Karlsruhe Institute of Technology

Design, implementation and evaluation of a visual interface to guide the knowledge discovery process within large biological datasets

Bachelor Thesis of

Manuel Traub

at the Department of Informatics
Institute for Applied Computer Science (IAI)

Reviewer:	Prof. Dr. Veit Hagenmeyer
Second reviewer:	apl. Prof. Dr. Ralf Mikut
Advisor:	Johannes Stegmaier
Second advisor:	Benjamin Schott

12. Januar 2016 – 11. Mai 2016

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, May 11th, 2016

. . . . . .

(Manuel Traub)

Abstract

Multi-dimensional image data from modern microscopes allow studying the embryogenesis of various model organisms. For instance, image analysis methods like automatic or semi-automatic segmentation and tracking are used to analyze embryonic development by extracting cell movement trajectories. Most available software tools, however, aren't capable of processing those trajectories interactively and intuitively in combination with calculated features like cell division rate or trajectory length. In this thesis, a novel Matlab VTK interface is developed in order to guide a knowledge discovery process within large biological datasets. This interface extends Matlab about fast and interactive VTK 2D/3D views written in C++ to utilize both major advantages of Matlab and C++, namely usability and speed. The generic design of this interface allows rapid prototyping of various graphical user interfaces (GUIs) from Matlab through the use of wrapped C++ callbacks for user interactions. This enables feature-based selections through Matlab plots, which are directly linked to VTK views. The developed framework was successfully used to develop customized GUIs that allow interactive trajectory selection, single-cell trajectory analysis on maximum intensity projections as well as tracking error visualizations and corrections.

Zusammenfassung

Multidimensionale Bilder aus aktuellen Mikroskopen ermöglicht es die Embryogenese verschiedenster Modellorganismen zu studieren. Erreicht wird dies durch automatisches bzw. teilweise automatisches Segmentieren und Tracken dieser Bilder, wobei Zellbewegunstrajektorien generiert werden. Die meisten aktuellen Software Tools verfügen jedoch nicht über die Möglichkeit diese Trajektorien interaktiv und intuitiv in Kombination mit berechneten Eigenschaften wie Zellteilungsrate oder Trajektorienlänge zu bearbeiten. In dieser Arbeit wird eine neue Matlab VTK Schnittelle entwickelt um die Wissensentdeckung in großen biologischen Datensätzen zu unterstützen. Die entwickelte Schnittelle erweitert Matlab um in C++ geschriebene 2D/3D VTK Ansichten, um die jeweiligen Vorteile, Nutzerfreundlichkeit und Geschwindigkeit, von Matlab bzw. C++ zu nutzen. Der allgemein gehaltene Entwurf dieser Schnittstelle erlaubt durch Rapid-Prototyping das einfache erstellen verschiedenster graphischer Benutzerschnittellen (GUIs) in dem gekapselte C++ Callbacks aus Matlab heraus genutzt werden. Dies ermöglicht das erstellen von Clustern basierend auf den Eigenschaften von Trajektorien welche in Matlab-Schaubildern visualisiert werden. Mit Hilfe der entwickelte Schnittelle war es möglich verschiedene anwendungsbezogene GUIs zu erstellen, die z.B. zur interaktiven Trajektorienauswahl, zur Anzeige und Analyse einzelner Zellen inklusive ihrer mit Mikroskopbildern unterlegten Trajektorien, sowie zur Anzeige und Korrektur von Trackingfehlern.

Contents

List of Figures

List of Tables

1. Introduction

Since biologists explore nature through microscopes, their structure has largely developed from simple optical microscopes [7] to a wide range of digital microscopes [38] that rapidly produce large amounts of image data. These microscopes allow studying the embryogenesis of various model organisms, which can be used for drug development by examining the impact of certain drugs to the natural growth of embryos [1]. By running such an experiment, the described microscopes can produce several terabytes of data [31]. It's clear that these huge amounts of data need to be automatically or semi-automatically processed to generate meaningful information [10].

1.1. Knowledge discovery within large biological datasets

The main target of this thesis is to guide a knowledge discovery process using full tracks from a zebrafish as a biological dataset. These full tracks are trajectories for each moving cell of a developing embryo. They can be used to examine the impact of specific drugs or developmental differences of mutant and wild type embryos by comparing the characteristics from different embryos. In order to discover knowledge, it is important to have an interactive and intuitive user interface where a user can visualize and analyze the data. It is as important to have a 3D view, as to have a 2D view in combination with the original image data, since this is the data representation biologists are familiar with. Within these different views a user should be able to manually generate clusters by selecting subsets of the visualized data. A selection from one view should automatically be generated within all other linked views. Another important feature is the possibility to select clusters through features like full track length or cell division rate. In order to have this functionality, these features should be made visible in a feature plot which is also linked with all other windows. A user should then be able to select clusters within the feature plot and also fine-tune them manually by selections in different views.

1.2. From embryo to cell lineages

In order to generate the described full tracks, there are several steps involved which are shown in Figure 1.1. This section describes those steps which are namely: embryo selection and preparation, image generation with a microscope, followed by cell segmentation and tracking.

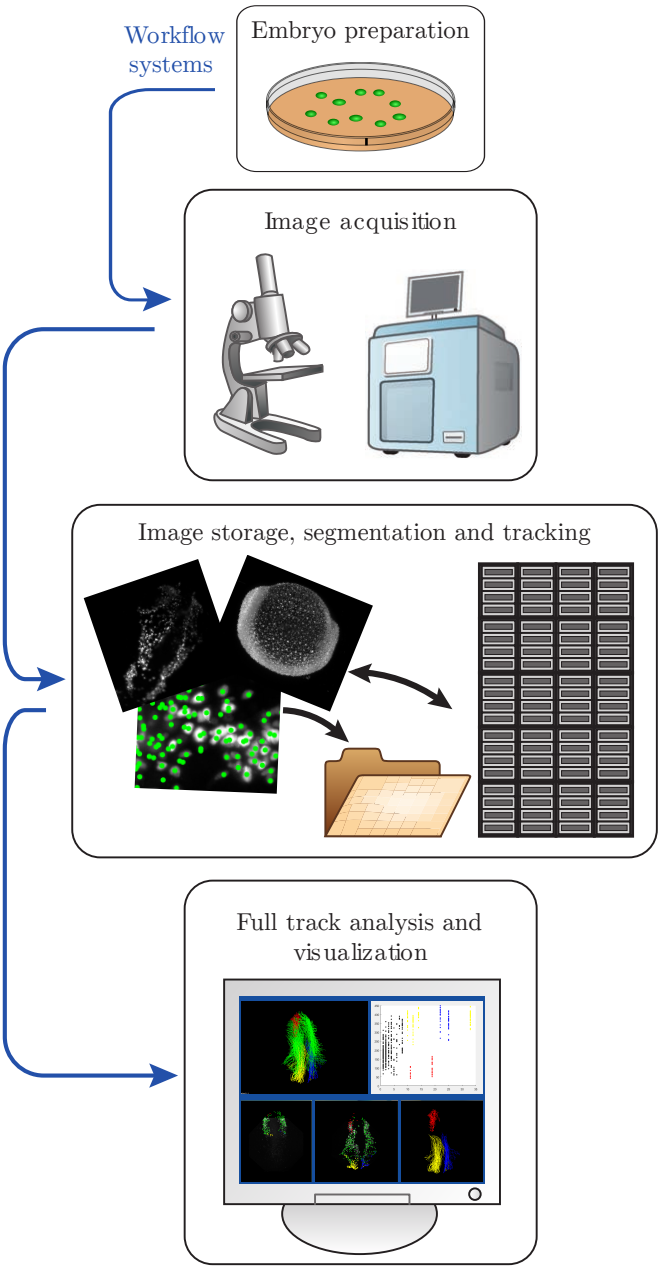

Figure 1.1.: In order to generate full tracks, first select and prepare a suitable embryo. Second run an experiment with a microscope to generate 3D time resolved images. Third store and process these images to generate the desired full tracks, which can then be analyzed and visualized (image adapted from [10]).

1.2.1. Why zebrafish are used in research

The full tracks used in this thesis originate from zebrafish since they are widely used as model organisms [26]. This is due to the fact that zebrafish are small, translucent freshwater animals [11]. They grow to an early larva within only 72 hours making it feasible to explore their early stages of development within a reasonable time lapse [24]. Another valuable characteristic of zebrafish is the fact, that they are fast and easy to breed. They are fertile at an age of 3 months and females can lay up to 200 eggs per mating [42]. But the outstanding feature which truly supports the written experiments are genetically modified zebrafish that produce fluorescent proteins upon expression of specific genes within their cells making them even better visible for microscopes [20]. Another improvement is the treatment of phenylthiocarbamide at early stages of development to suppress generation of pigments which enhances the transparency of the zebrafish embryo [22].

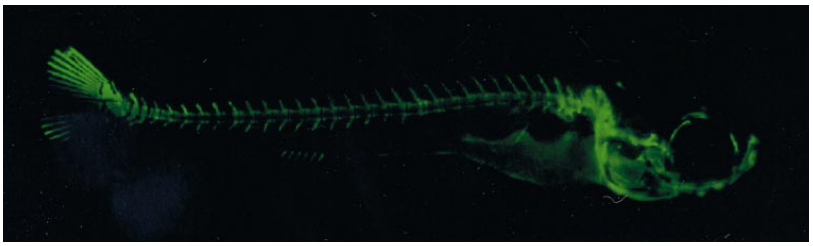

Figure 1.2.: Zebrafish with chromophore calcein as fluorescence marker to highlight its skeletal structure [8].

1.2.2. General recording techniques to generate *in vivo* time resolved images

Once a suitable embryo is chosen for the experiment, the next step is to generate *in vivo* time resolved images. There are several different types of microscopes which are capable for this purpose.

Scanning electron microscope (SEM)

SEM generates images by scanning probes with a focused beam of electrons. Recent studies showed that SEM can be also used for *in vivo* experiments [27]. The advantage of using electrons in comparison to visible light in optical microscopes is that one can reach higher magnification [15].

Confocal microscopy (CF)

CF is an optical microscope that uses a laser to scan the probe instead of illuminating the whole probe by once. By using this technique, it increases the contrast within thick probes [43, 41].

Stimulated emission depletion (STED)

STED is a special optical micrsoscope which can break Abbe's diffraction resolution limit and thus reach a significantly higher resolution than confocal microscopy [25].

Single-plane illumination microscopy (SPIM)

SPIM is an optical microscope that uses a thin plane of laser light to illuminate a specific layer within a probe. As shown in Figure 1.3, the light that is used to stimulate the fluorescence proteins within the embryo shines from an orthogonal angle with respect to the detection objective. This setup allows to generate an image from a single thin layer within the embryo [35, 21].

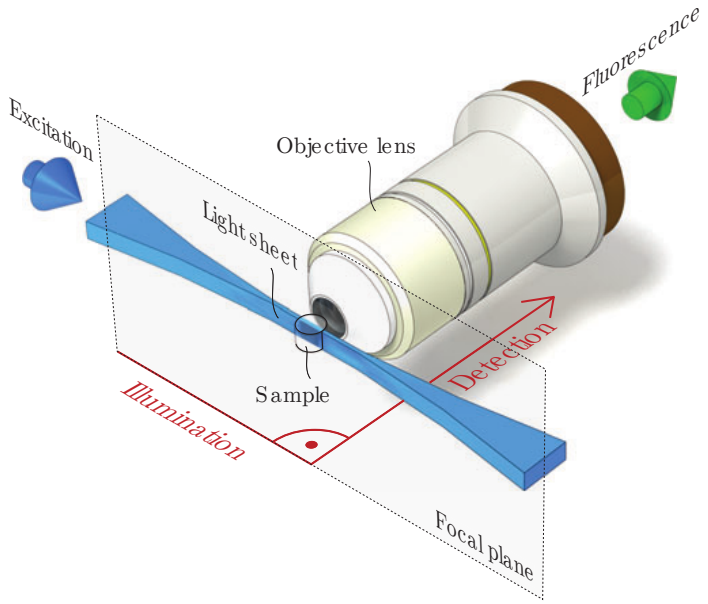

Figure 1.3.: Single-plane illumination microscopy. A thin laser light-sheet stimulates the fluorescent protein within a specific layer of the embryo [21].

1.2.3. Creating 3D time resolved images

In order to generate tracklets from time resolved images, it is necessary to have 3D images, this are images of different layers of an object at the same time point. This is the reason why using SPIM is an optimal choice for generating such 3D time resolved images [23]. Thus, the full tracks used in this thesis are generated from SPIM images.

1.2.4. Cell segmentation and tracking

After the generation of the 3D time resolved images, the next step is to detect and track the cells. This process of finding specific regions within an image that represent living cells is called cell segmentation. While there exists a wide range of algorithms capable of this task [19], it is not trivial to develop one that is also capable of processing large

amounts of image data up to several terabytes in reasonable time [39].
After segmentation the next step is to track each cell within different images over time to generate a trajectory for each cell. Unfortunately, creating a perfect cell lineage is still a difficult task [12]. There exists a bunch of software tools to support a semi-automatic segmentation and tracking process [12, 36, 34].

1.2.5. Interactive classification and knowledge discovery

The last step from Figure 1.1 is the classification and the visualization of the full tracks generated within the previous step. For this purpose one needs an adequate data representation, a 3D view of the full tracks for example. To do the actual classification one needs to divide the dataset into clusters, each representing a specific functionality or otherwise defined region of interest. The clustering could be done manually through the visual data representation, or better based on some calculated features like track length or movement speed, which are already dividing the dataset in some way [37].
There are already some software tools which are partially capable of this sort of tasks as shown in Table 1.1.

	Feature based clustering	Connected feature plots	Matlab interface	Open source
Fiji	-	-	unidirectional	X
CATMAID	X	-	-	X
Mov-IT	-	-	-	X
Andrienko	X	X	-	-

Table 1.1.: Software tools for interactive clustering.

Fiji

Fiji focuses on image processing like semi-automatic segmentation and tracking. Although it has a unidirectional interface to Matlab and 3D visualization, it lacks the feature based group selection [36].

CATMAID

CATMAID is another bioinformatics tool which is specialized for semi-automatically segmentation. CATMAID's main focus are images of neuronal cells from which they generate 3D representation of this cells [34].

Mov-IT

Mov-IT is a software designed for semi-automatic segmentation and tracking of cells over time. The generated tracks can be viewed in 3D and groups can be manually selected, but it has no feature based selection and no interface to Matlab [12].

Andrienko

The Graphical User Interface (GUI) described by Andrienko et al. comes most close to the desired functionality developed in this thesis. Their software is capable of connecting different data views including 3D views and feature plots, but their software isn't compatible with Matlab and also isn't open source making it unfeasible for extensions [3, 2].

1.3. Targets of this thesis

As pointed out in Section 1.2.5, none of the existing software tools for interactive clustering is capable of all requirements. For instance, none of the tools was really suitable for use with Matlab. So there was a need for developing a custom software.

1.3.1. Main targets

The main targets, which are necessary to overcome the existing limitations, are listed below:

Matlab compatibility

Having Matlab compatibility as requirement is due to the fact that it is widely used by researchers for its powerful data processing capability [18]. Additionally, the tracking algorithm used to generate the full tracks used within this thesis is written in Matlab code [40].

3D and 2D data representation

besides a 3D view, the software should be capable of presenting the data in 2D along with the original image data, since this is the data representation biologists are used to.

Interactive clustering

Within different views a user should be able to generate clusters manually through selections.

Connected views

Clusters from different views that represent the same sub dataset should be connected in a way, that updating a cluster in one view also updates all connected once.

Feature based clustering

Another important ability that half of the programs shown in Section 1.2.5 lack, is the possibility to select clusters through features like full track length or cell division rate.

Feature plot

In order to be able to select clusters through features, these features should be made visible in a feature plot which is also linked with all other windows. A user should then be able to

select clusters within the feature plot and also fine tune them manually through selections within other views.

1.3.2. Additional targets

In addition to the main target, the software itself should be compatible to already used methods. This is the origin of the written requirement for Matlab compatibility. Since features could already be calculated and visualized through Matlab there is no need to reinvent this [32]. On the other hand, Matlab's 3D plots are not capable of handling large datasets interactively. Thus, the new software should extend Matlab about a bidirectional interface to allow the described connections between feature plots and other views.
Other sub targets are:

Easy extensibility

The software should be modular to allow modifications and extensions.

Handle large datasets

As mentioned above, the software should be interactive and still responsive when working with large datasets.

Controlled by Matlab

Since the experience of the developer of the existing code lies on the Matlab side, the GUI should optimally be build using Matlab code and only outsource heavy tasks like 3D visualization.

2. Methods

As shown in the last chapter, there was a need to develop a new software capable of extending Matlabs 3D visualization ability. The already used method to interactively explore large 3D data from Matlab was to export those data to a Visualization Toolkit (VTK) file and then open this file with Paraview (a user interface based on VTK to visualize large 3D data). Although this method was applicable for just visualizing the full tracks in 3D there was no option to export selected clusters from Paraview back to Matlab.

2.1. Programming language and library choices

In the early stages of development there were several options: One could go with the Paraview solution and extend it about an bidirectional interface to Matlab. Another option would be to develop an own user interface based on the VTK library. At first the Paraview solution seemed to be the easiest. However, it turned out that Paraview's plugin system was mainly focused on file access. So a stand-alone VTK application was the way to go. Since Matlab already has a bidirectional Python engine and VTK has a Python interface, the first few tests were written in Python. During those tests several drawbacks occurred. First not all functionality of the VTK C++ interface is also present within the VTK Python interface, second Python turned out to be noticeably slow with the reduced test dataset. So finally, C++ was selected as the programming language along with the VTK library to build the external software component. This not only gave a huge speed improvement within the simple tests, it also enabled the use of all VTK functionality since VTK is also written in C++.

2.2. Project organization and software build methods

After some initial tests with Matlab's C++ interface (Mex interface), the actual software development started. In order to easily handle the VTK dependency during the software build process, the CMake build system [30] was used, since it's also used to build the VTK libraries. With CMake one could just include VTK as an dependency and it automatically detects the VTK library installation. Additionally, git was used as version control system [28]. As working title for the Matlab VTK interface the name Simple VTK Interface (SVI) was chosen. Simple in this context refers to the complex functionality provided by VTK, which makes it hard for developers understand it. SVI on the other hand should be as powerful as needed, but also as simple to use as possible.

2.3. Software component models with design patterns

This section will show the detailed software models used to build the actual Matlab VTK interface. To help understanding the different components, the main software parts are visualized using Unified Modeling Language (UML) diagrams [5]. Additionally the structure of most software parts are partially or directly based on well known design patterns [14]. For a complete list of the used design patters see Appendix A.3.

2.3.1. The basic Matlab user / developer interface

The basic functionality of the Matlab VTK interface is to create a 3D view in a separate window and to display Matlab specific 3D data in it, just like with a normal Matlab figure. To achieve this functionality, an interface developer uses the classes shown in Figure 2.1. The `SVI` class is implemented using the Singleton design pattern [9] encapsulating the very basic functionality like creating windows and properties. The `SVIWindow`, `SVIProperties`, `SVIID` and `SVIRawIDSelection` are best described by the Wrapper design pattern [9], encapsulating the underlying C++ data types. These classes are the four basic data types needed to create and control 3D visualizations, which consist of points or lines. The first thing needed to display data is a window represented by `SVIWindow`. This window can have different properties (represented by `SVIProperties`) like background color, size, camera position and so on. Once a window is created, a user can add data to it. This data have to be in a specific Matlab format to be accepted. For instance, a three dimensional vector can represent a set of 3D lines: NxMx3 with N being the number of lines with M points each, for each point x, y and z coordinates.
The type and properties parameter from the `add` method can be omitted to use default values for them. After execution, the `add` function returns an `SVIID` object representing the visualization of the added data. Through this ID, the user can now request and change properties like color, opacity and others.
Another major functionality is that from each `SVIID` one can create a selection by calling the `addSelectionType` function. Selections are either point or line selections.
Through a built-in selection mode, a user can define a frustum (basically a rectangular prism) by drawing a rectangle on a window. This frustum is then used to create a subset of the data represented by the `SVIID`, the selection was created from.
The `SVISelectionCallbackProcessor` and the `SVIInteractionCallback` are wrapped interfaces to create custom GUIs and allow rapid prototyping by handling user interactions and processing selections from Matlab. These interfaces are described in Section 2.3.11 and 2.3.6.

2.3.2. Connecting to C++, Matlabs Mex interface

To utilize the speed of natively compiled languages like C, C++ or Fortran, Matlab has a so-called Mex interface that allows developers to write programs in the desired language and use them as plug-ins. Matlab therefore provides an interface that one has to implement. This interface actually is a single function, which can return and can be called from Matlab with arbitrary data. The data is provided in Matlabs native C / C++ / Fortran format, and

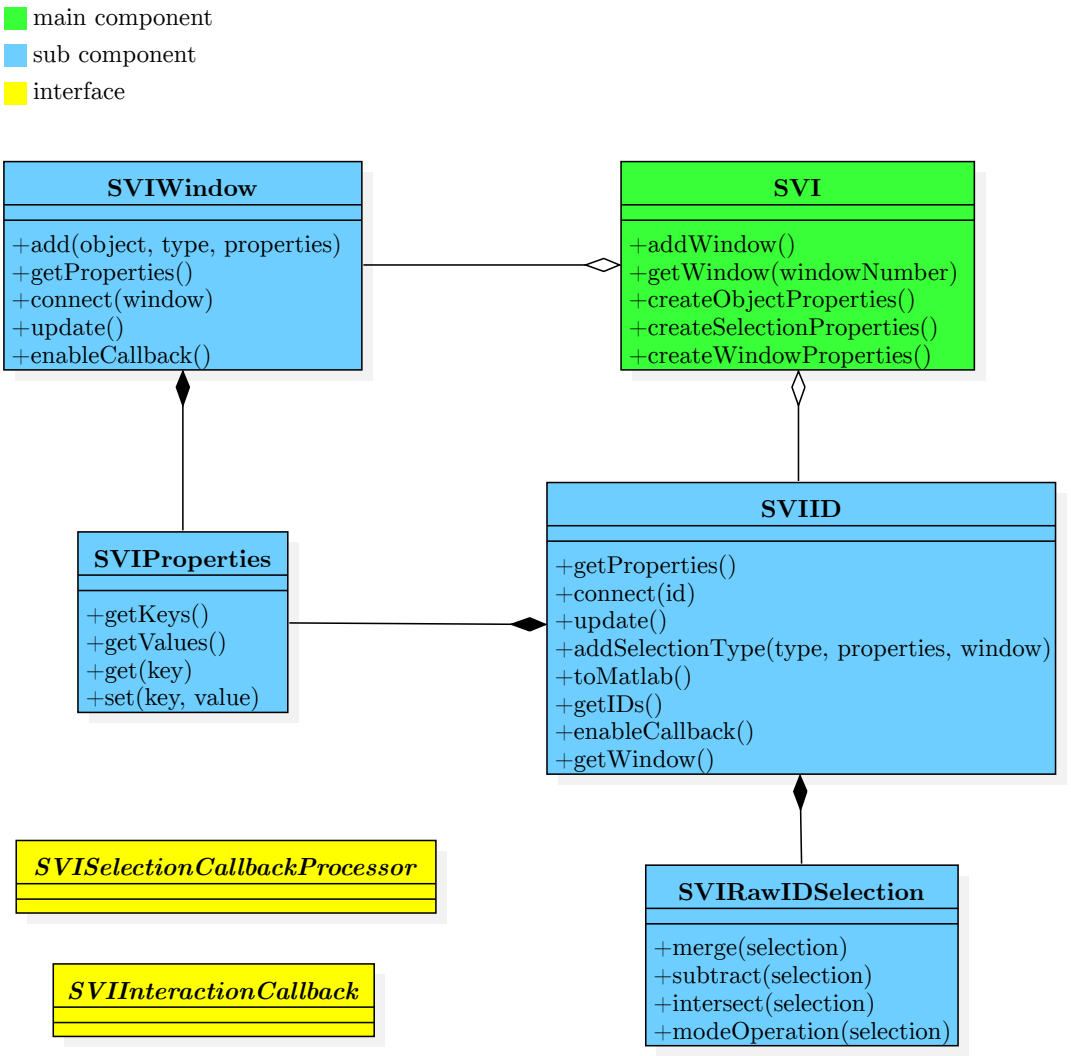

Figure 2.1.: SVI represents the plain C++ wrapper, when a Matlab object gets added to a `SVIWindow` it creates a `SVIID` as its representation. Windows and IDs can have a set of properties represented by the `SVIProperty` class. `SVIRawIDSelection` represents the visible IDs of an object or selection.

can be read / created through some helper functions. Finally, the written code can be compiled with Matlabs built-in Mex compiler. Since this Mex compiler effectively produces a shared library it can be circumvented by explicitly providing Matlabs Mex headers. This was done to integrate the compilation of the Mex file into the CMake build process. Once a Mex file is created it can be called from Matlab by using the Mex filename as a function name.
Within the native code the developer is free to create several threads, but he must not use any Matlab helper functions outside the thread from which Matlab called the Mex interface function. Thus, the Mex interface is only unidirectional from Matlab to native code. It is not intended to call Matlab from native code after the Mex interface-function

has returned. In addition, just holding execution within the Mex interface function is not an option since Matlab will also hold until the function returned.

2.3.3. Achieving a bidirectional connection from and to Matlab

As mentioned in Section 2.3.2, Matlab does not natively support a bidirectional interface to C++, so there was a need to circumvent this. Fortunately, Matlab supports local TCP sockets with custom callbacks that get called once data is available.
So it was surprisingly easy to open a local TCP socket within Matlab, register a callback, and connect to this socket from a separate thread from within the Mex code. Figure 2.2 shows this setup. This bidirectional interface is mainly needed to react to user interactions from within Matlab.
As mentioned in Section 1.3.1, an important feature of this VTK-Matlab interface is to connect selections from different visualized objects, so that they can react to changes of one another. But doing so creates a huge amount of possible interactions between them. So it wasn't feasible to encode all these possible interactions directly within the interface. Instead, the actual selection arithmetic is outsourced through callbacks. By handling these callbacks from Matlab, developers have the possibility of easily creating new interactions between selections, therefore rapidly creating new GUIs without the need of recompiling the whole interface.
Figure 2.3 shows the general process of selection processing within Matlab: Once a user selects something in a VTK window, the created selection data including the data of all connected selections gets handled over to the `SVIMatlabSelectionProcessor`.
The `SVIMatlabSelectionProcessor` then informs Matlab over the local TCP-server. Matlab then calls the Mex function (from the `ConcreteSelectionProcessor`) which returns the selection data to Matlab. Within Matlab the selection interactions can be processed and even Matlab specific calculations like updating a plot with similar data can be preformed. After processing the selections, Matlab again calls the Mex function with the result of the selection calculation, telling the `SVIMatlabSelectionProcessor` to return from its callback. To handle even more control over to Matlab a developer can register Mouse or Key interaction callbacks, which then also get sent to Matlab and handled by the `SVIInteractionCallback`.

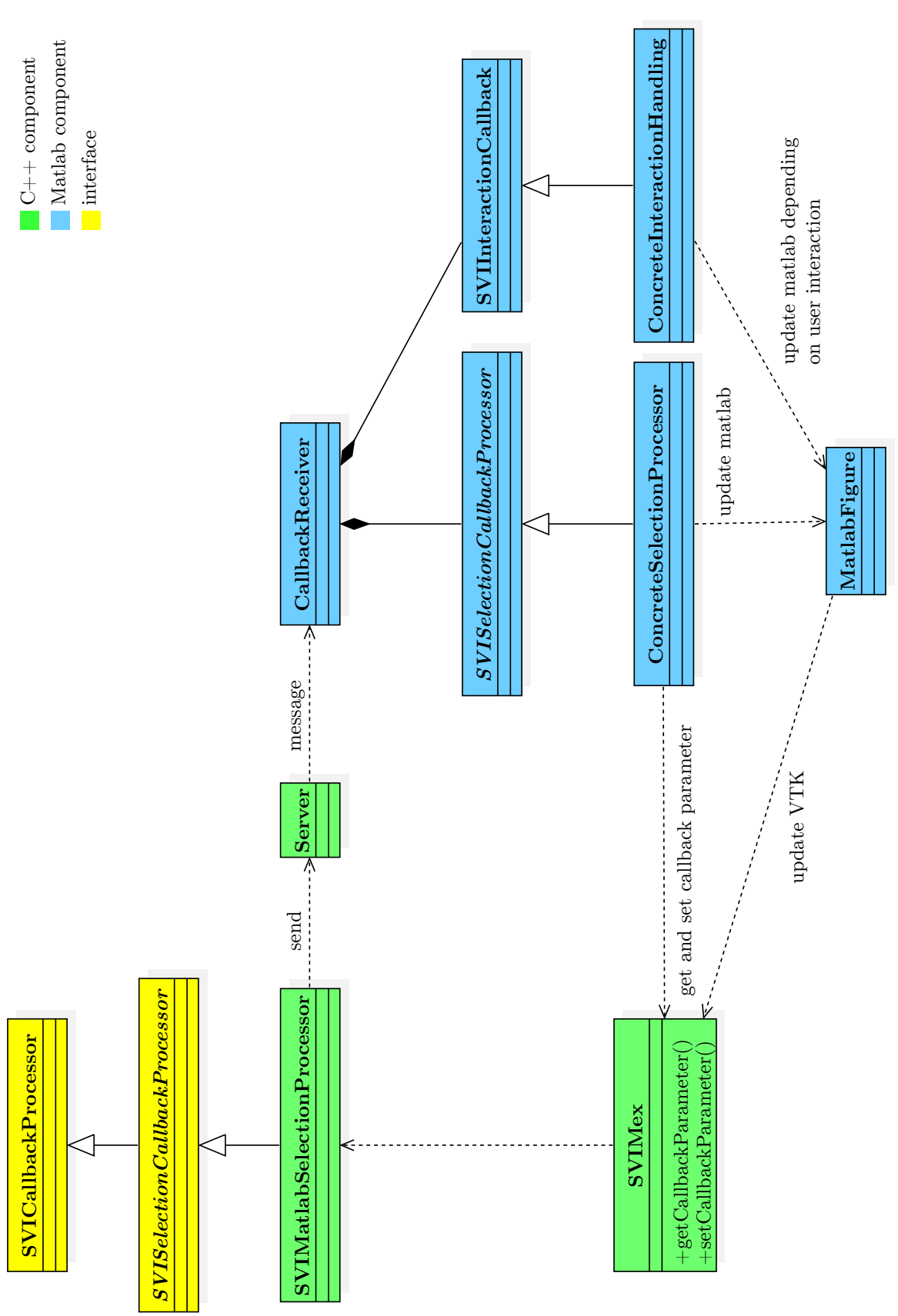

Figure 2.2.: Matlab C++ bidirectional interface: Because Matlabs Mex interface is unidirectional, a local TCP server and Matlab callbacks are used to achieve a bidirectional interface.

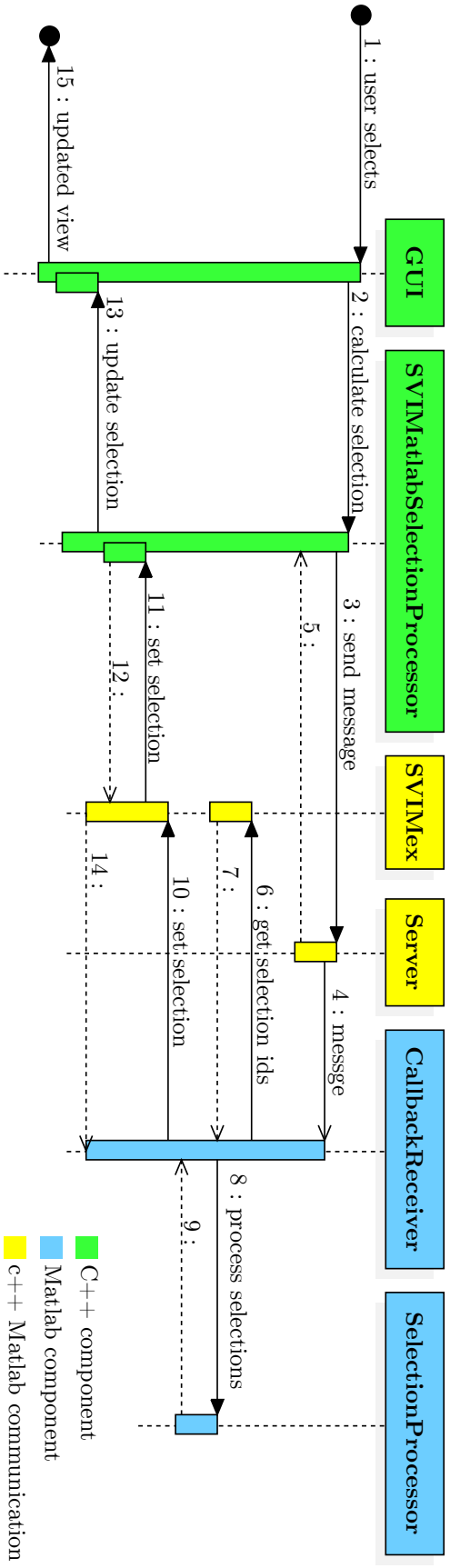

Figure 2.3.: Selection processing within Matlab, to allow rapid prototyping.

2.3.4. Basic C++ interface

In order to keep the connection between Matlab and C++ clean and simple, it was decided to separate the VTK dependency within a C++ interface. This interface is completely independent from Matlab and thus allows reusing the code for possibly other interfaces to other languages. To also make the compilation independent from Matlab, the C++ interface is built as a static library during the compilation process. This also allows Matlab independent testing of the whole VTK functionality.
The main functionality is exposed through the abstract class `SimpleVtkInterface` which allows creation and connection of windows as well as adding, removing and connecting objects and selections. To be independent from VTK headers, some simple data structures are provided to construct the equivalent of a `vtkPolyData` object (VTK's main data representation). Just like in VTK a `SVIPolyData` object consists of so-called `SVICells` which consist of `SVIPoints`. In VTK, cells are used to represent dependent 3D points like lines or simple polygons. In order to choose the specific visualization of the data an `SVIDataType` object is used in `SVIPolyData` to distinguish between lines and points. In order to effectively create dependencies between the points or cells of different objects, each `SVIPoint` and each `SVICell` has its own ID. This ID is used within the selection processing, for example to select same parts from different objects. The `SVIRawIds` class is used to store the currently visible point and cell IDs of an object or selection and can be generated by calling the `getIds` function.
Adding objects and creating selections works petty similar to the Matlab interface: by calling `add` or `addSelectionType` one gets an `SVIId` object, which can then be used in other functions. Thus, the `SimpleVtkInterface` class as shown in Figure 2.4 acts just like the Factory Method design pattern [14], encapsulating the creation process of the different 3D visualizations represented by the `SVIId` class.
As in Matlab, the `SVIProperties` class controls the appearance of windows, objects and selections. Also, the `SVISelectionCallbackProcessor` and the `SVIInteractionCallback` are the C++ equivalent interfaces to create custom GUIs and allow rapid prototyping. These interfaces will be explained in detail later on.

2.3.5. VTK's single thread architecture and how to achieve multi-threading

A major problem that occurred during the start of this project was the fact that VTK is not thread-safe, so VTK dependent code can't be executed by several threads at the same time. This alone could have been handled easily using some sort of locking mechanism for thread synchronization. Unfortunately, OpenGL, the render library used by VTK, also requires to be exclusively executed from the same thread. So there was the need of developing something like a Producer Consumer design pattern [17] with one VTK thread functioning as the Consumer getting synchronized requests form one or more Producer threads. Figure 2.5 shows the general concept used in this work, which implements the described pattern. To be able to interact with the VTK thread, one has to implement a custom event loop by subclassing the VTK class which implements the described loop. This is done in the `SVIInteractor` class, which subclasses the Linux and Windows specific `vtkRenderWindowInteractors`. The reimplemented event loop in `SVIInteractor` handles

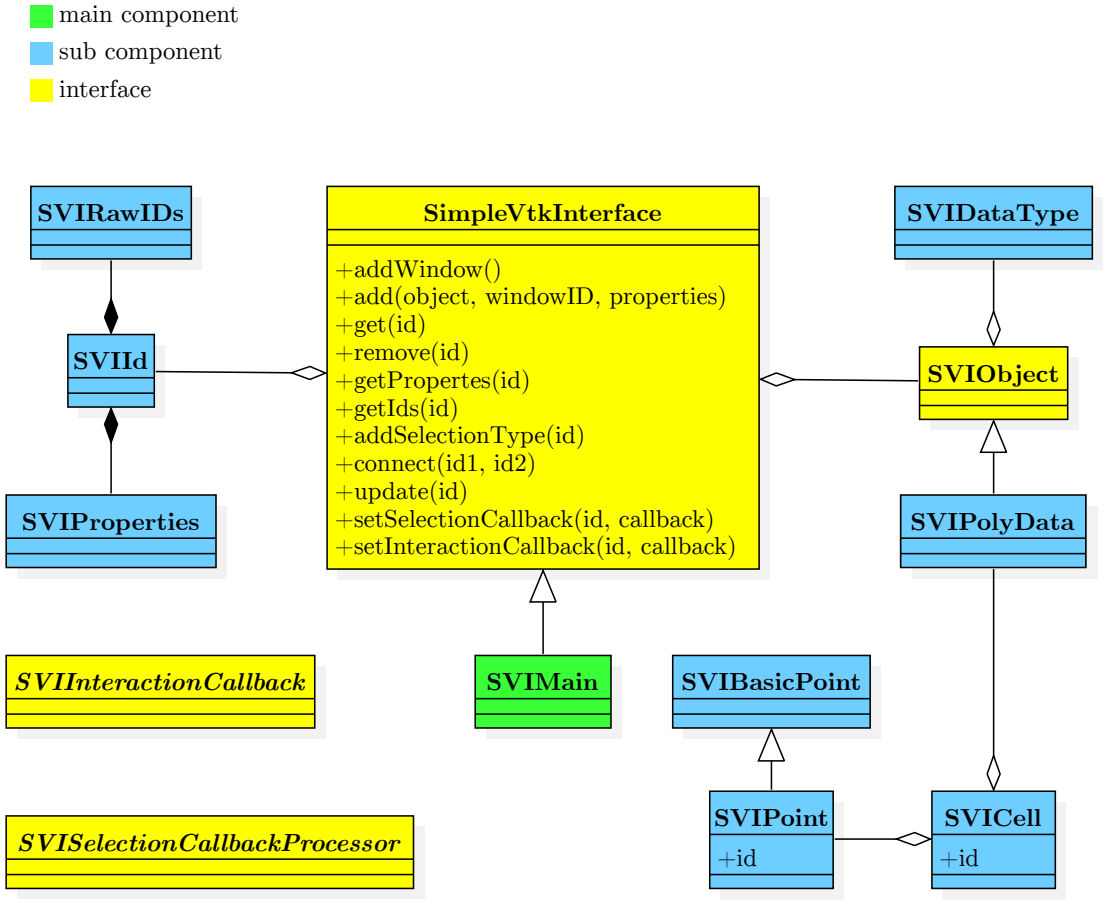

Figure 2.4.: The C++ library interface is used to encapsulate the VTK functionality. The interface methods are: creating windows, adding removing and connecting objects and selections.

user events, mouse and keyboard events, just like VTK did, but also frequently checks a message queue. This message queue is implemented by `SVIMessenger` and is used do deliver interface commands, like adding a window or object, to the VTK thread.
As shown in Figure 2.6, once an interface method is requested, its specific call parameters are saved within the `SVISyncParameter` class, then a message is sent informing the VTK thread about the called interface method. Right after the message is sent, the main thread, typically the Matlab thread, requests the return parameter from the `SVISyncReturn` class. The `SVISyncReturn` class then blocks the main thread until the actual return parameter is set by the VTK thread.
Figure 2.7 shows how the `SVIMain` class processes the actual interface call within the VTK thread. This happens after the event loop implemented in `SVIInteractor` receives the previously send message and then calls back to `SVIMain`.

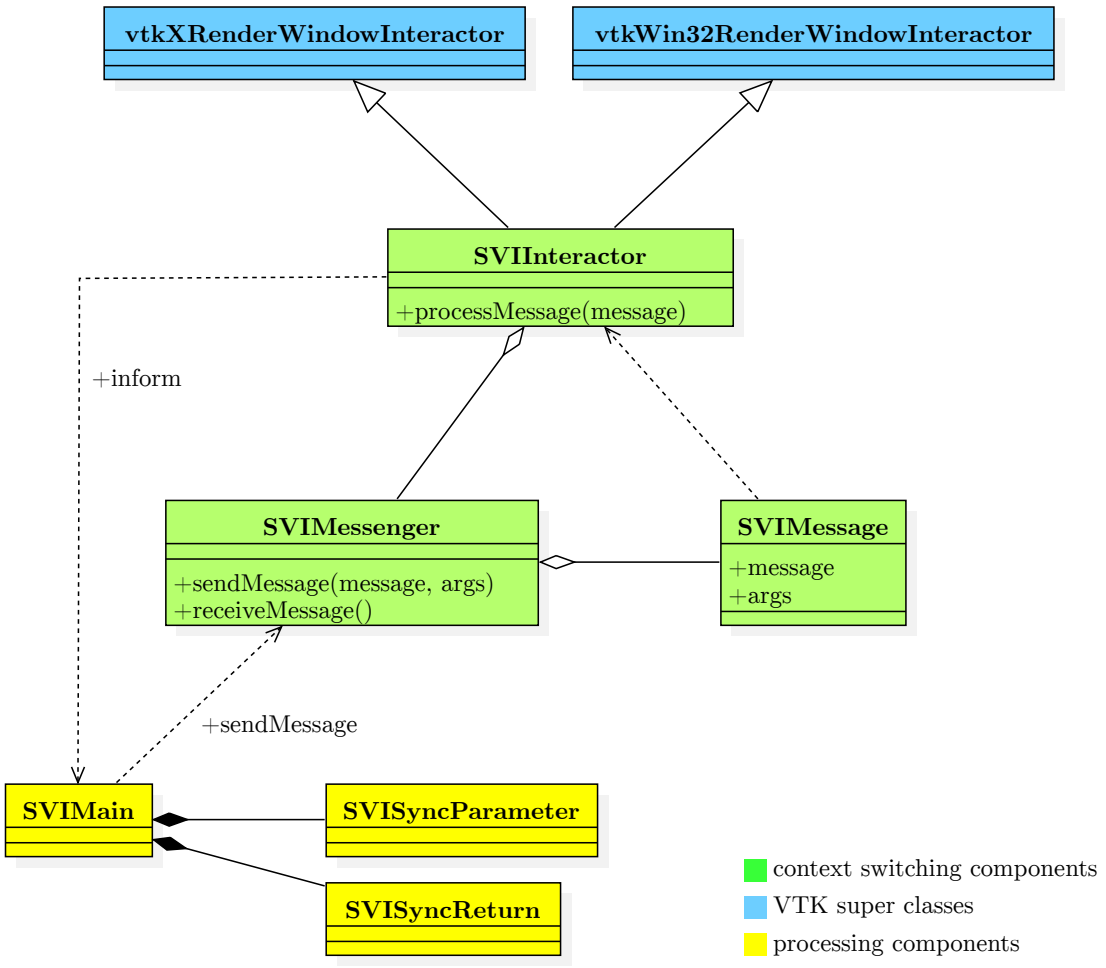

Figure 2.5.: Achieving multi-threading by using messages and synchronization structures.

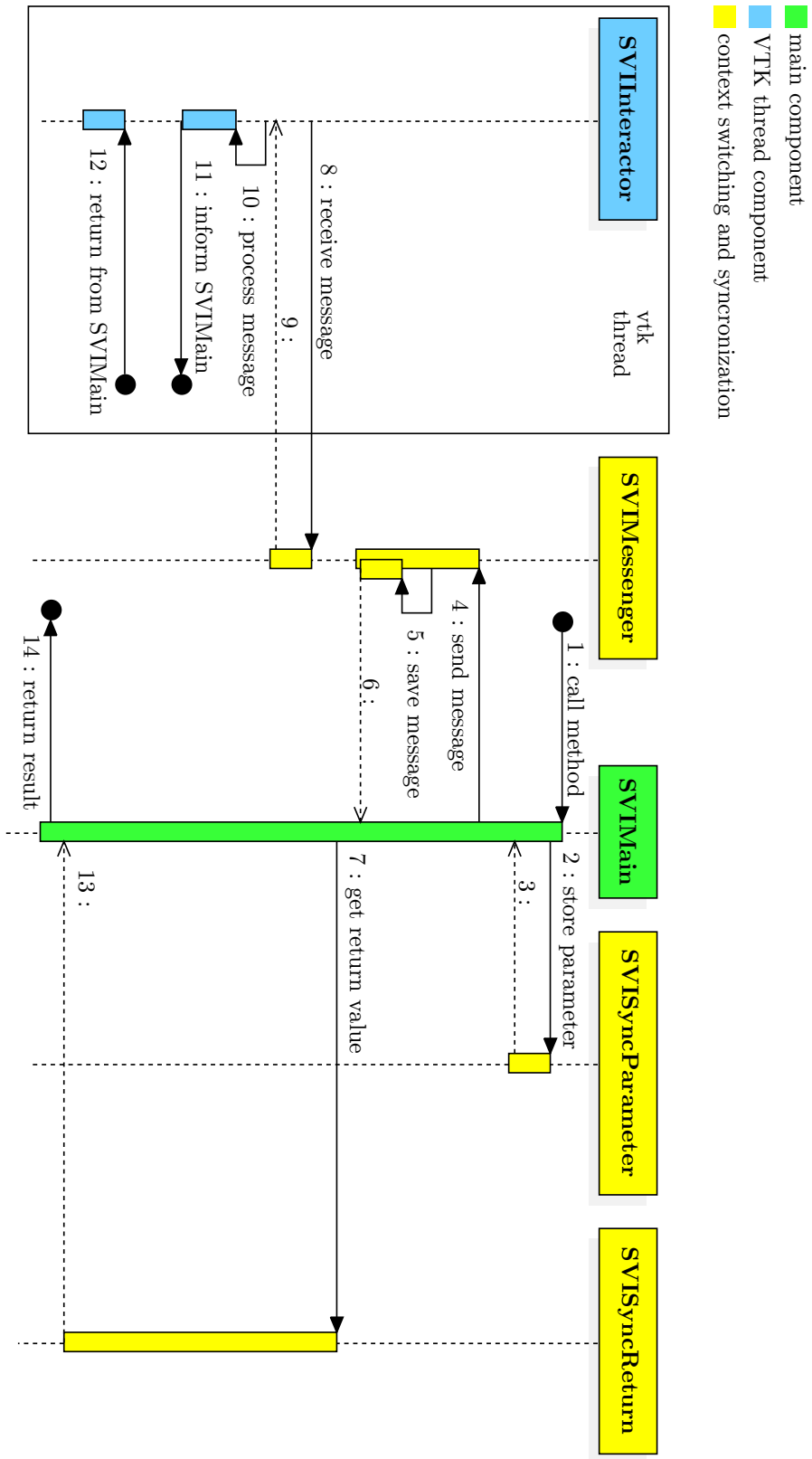

Figure 2.6.: Process of handling an interface call, by passing control over to the VTK thread.

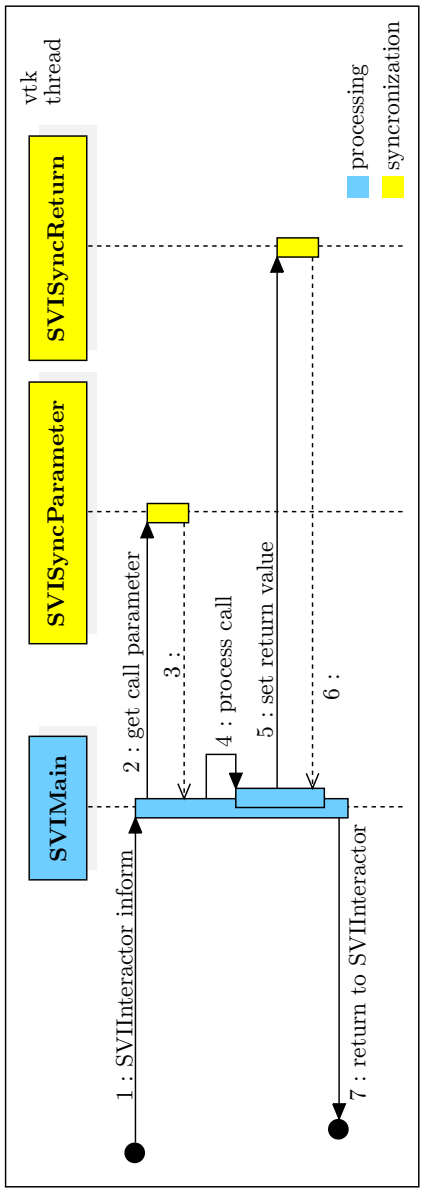

Figure 2.7.: VTK thread processing of the interface call, using the synchronization structures.

2.3.6. How to customize VTK's visualization model

To actually represent a window, the `SVIWindow` class is used. It stores all visualized objects and selections and handles camera and rendering requests. It also holds a `SVIInteractorStyle` object, which is a subclass of VTK's interaction handling class `vtkInteractorStyleRubberBandPick` as shown in Figure 2.8. This VTK class comes with a built-in rubber band frustum selection and provides abstract methods to customize the behavior of user actions like keyboard and mouse inputs.
Through the `SVIInteractionCallback`, a developer can register an own callback and will then be informed about the various user actions.

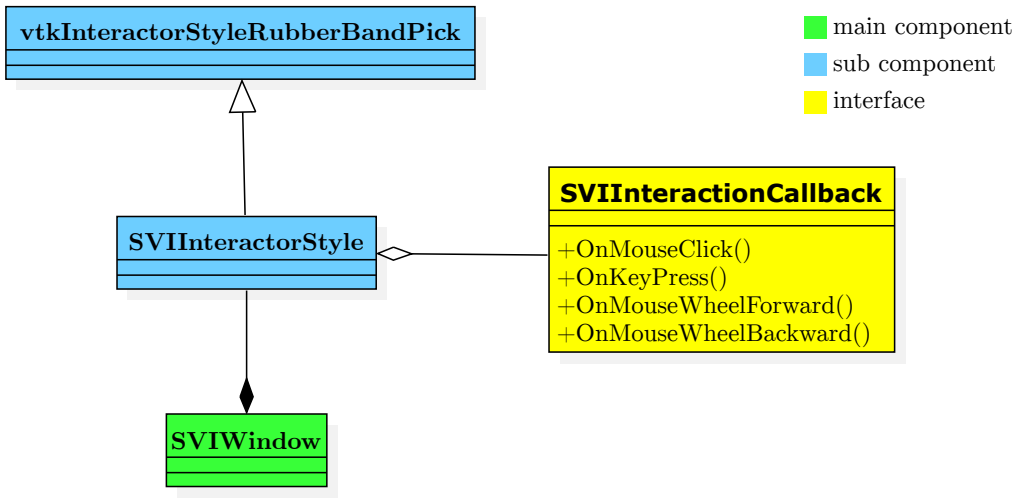

Figure 2.8.: Allow developers to react to user events by implementing the `SVIInteractionCallback` interface.

2.3.7. First draft of visualizing objects and selections

After having a single VTK thread and a render window representation, the next step was to actuality visualize something. This section describes why the first approach of doing so failed by getting to vast and complex.
In the beginning of this project, there was only the need for visualized objects and selections on these objects. Figure 2.9 shows the first approach modeling them. Since VTK has a very good abstraction layer, a visualized object can be represented by an abstract `vtkDataObject` and its properties. To create a selection from a VTK object one needs a frustum. By applying several VTK filters one can then create different types of selections, like point or line selections. So it seemed to be the easiest solution, to define a selection as a pipeline of abstract VTK filters.
As mentioned in Section 1.3.1, it is important to connect selections from different visualized objects to form more powerful GUIs. With the existing classes from Figure 2.9 this wasn't quite easy to do.
As shown in Figure 2.10 each `VisualizedObject` got a `ConnectedWindow` object which itself

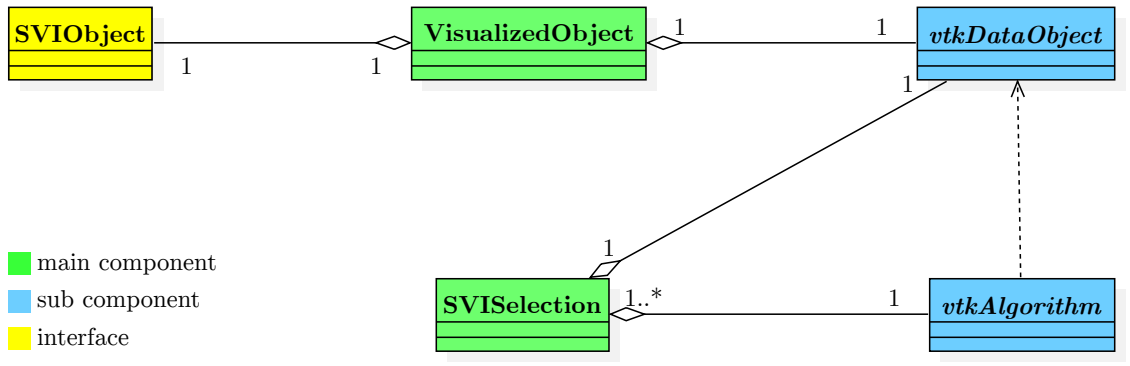

Figure 2.9.: Simple concept for handling visualized objects and selections, without much relations between them.

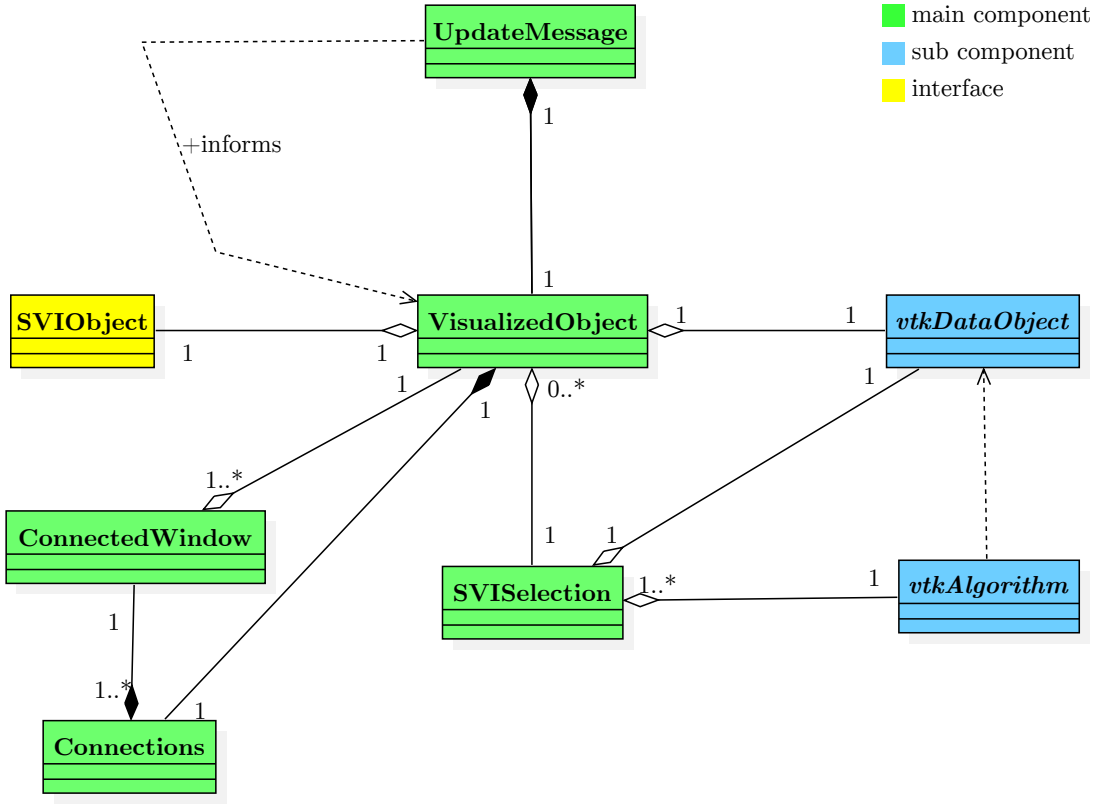

Figure 2.10.: The first approach failed by getting to complicated and messy when trying to extend the simple concept about connected selections, connected windows and a hardly comprehensible and understandable update mechanism.

stores several `VisualizedObject` from a different window. Several `ConnectedWindows` are pooled together within a `Connections` object.
Once a `VisualizedObject` gets updated it creates a `UpdateMessage` containing a reference to it so as information about the selection mode, to increase or subtract the selections. This `UpdateMessage` then gets delivered to all other `VisualizedObjects` listed in the

Connections object. Not only creates this a circular dependency, also the code to connect two VisualizedObjects objects was extremely difficult to understand. Another pitfall was that all selection arithmetic was directly encoded within the SVISelections class, which made it inflexible and hard to understand since it had to mange different selection modes in combination with connected selections.

2.3.8. Second approach: Modeling connected objects through a single instance

As shown in Section 2.3.7, there was a huge need for a redesign of the visualization objects and selection encoding, something that takes into account that selections and windows can be connected. To easily connect windows and selections, a class template called SVICollection as shown in Figure 2.11 was introduced. Just like the Multiton design pattern [29], this class controls the number of SVIBasicCollection instances through the static attributes collections, instances and indexCounter in a way that when one connects (merges) two instances they become one instance. This removes the previous struggle, where each VisualizedObject holds a list of its connections. Now there is only one list of objects which are connected to each other.
The state attribute of SVIBasicCollection was introduced to model the common state shared by several connected selections.

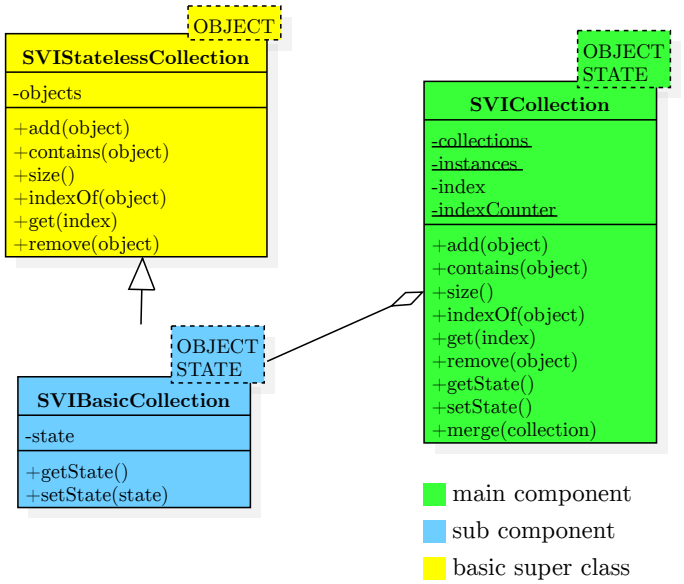

Figure 2.11.: SVIBasicCollection models a collection of connected objects that share a single state. SVICollection ensures that only one instance with the same connected objects exists.

2.3.9. The visualization pipeline

As mentioned in Section 2.3.7, the original design wasn't very flexible to changes. To circumvent this a new design shown in Figure 2.12 was created based on the Pipes and Filters design pattern [6]. The general idea of applying this pattern is to construct visualized objects and selections from small and reusable parts to form a flexible and easy adaptable system.

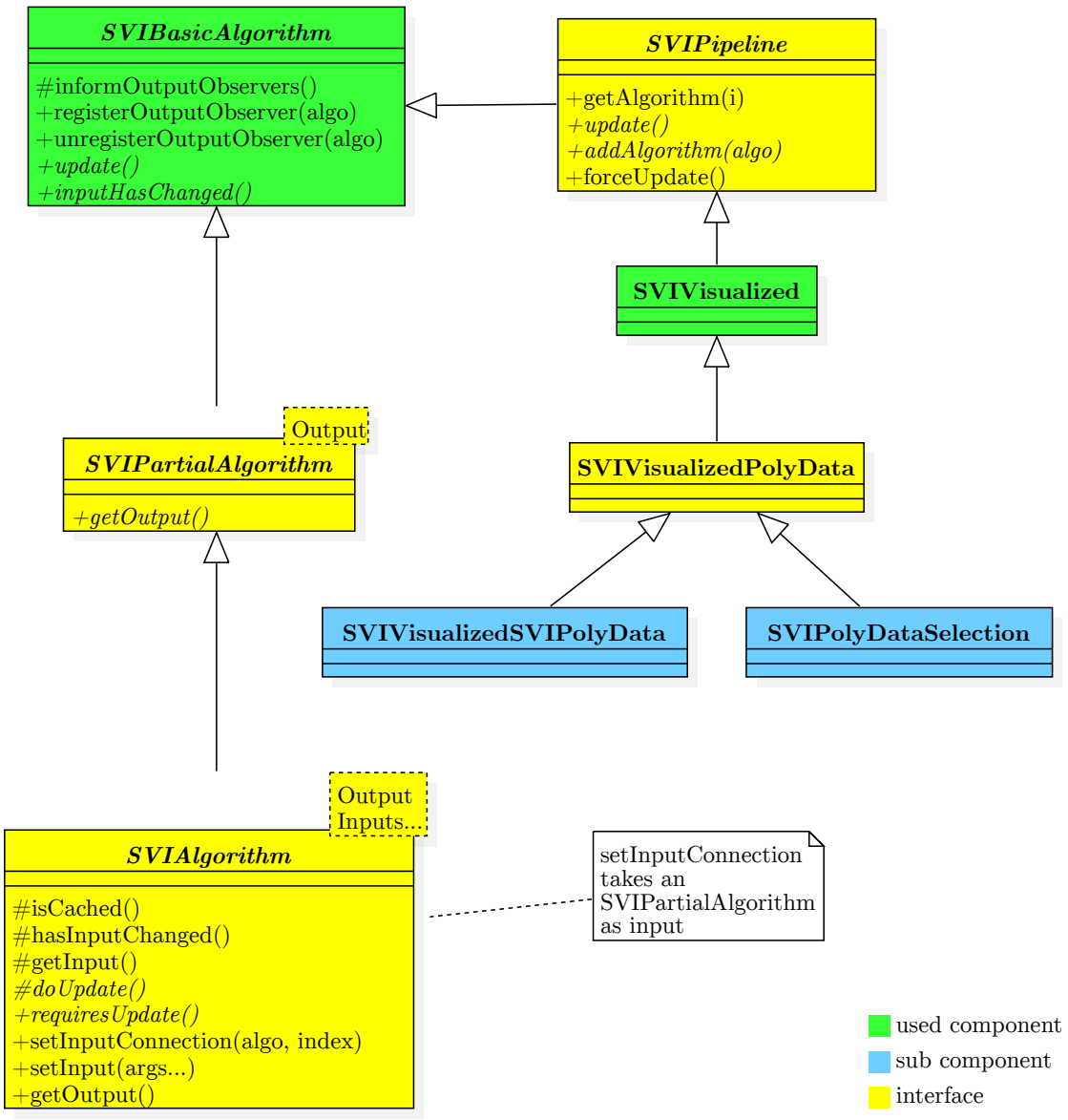

Figure 2.12.: New concept (Pipes and Filters design pattern [6]): creating small and reusable components connected in a linear manner.

The `SVIAlgorithm` class is the abstract form of a filter template that has several inputs and exactly one output. Through the `getOutput` function one can retrieve the last calculated output value. To calculate this output from the inputs, subclasses of `SVIAlgorithm` should

implement the abstract `doUpdate` function.
Due to the fact that `SVIAlgorithm` is a class template and therefore one can't create pointers or references to it, the `SVIBasicAlgorithm` was introduced. This base class specifies the update method, which in `SVIAlgorithm` calls the `doUpdate` function at some point.
To complete the pattern, `SVIPipeline` is an abstract implementation of a pipeline constructed of several filters (subclasses of `SVIBasicAlgorithm`). Once a pipeline gets updated it sequentially updates all its containing algorithms to calculate the final result of the last algorithm. However, updating all filters of a pipeline, is not always necessary. It is easy to imagine an example where a selection processing filter needs to recalculate its output, but a filter that just applies properties, like colors and opacity, doesn't need to run. Inspired by the Lazy Load pattern [13] a `SVIAlgorithm` can be a cached algorithm, meaning that the `update` function will only call `doUpdate` from the subclass if its really necessary, for example when the input has changed, modeling some sort of lazy calculation. Figure 2.13 shows this concept.

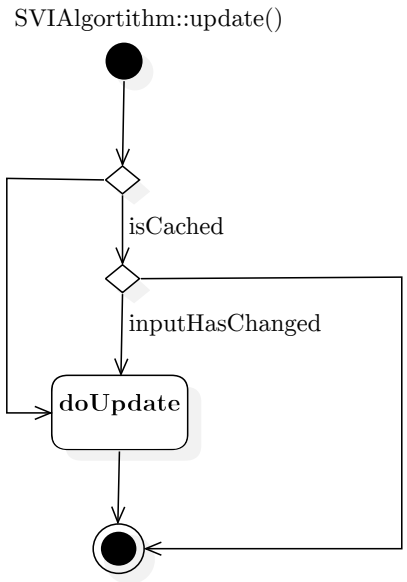

Figure 2.13.: Through lazy calculation a `SVIAlgorithm` only updates the output value if the inputs have changed.

Another important part is that a filter within the pipeline mostly directly depends on the calculation of a previous filter. To model this behavior, `SVIPartialAlgorithm`, a template class specifying only the output of a filter, was introduced. This allows to set a `SVIPartialAlgorithm` as a dependency for a specific input. The function `setInputConnection`, which registers this dependency, then automatically registers itself by the connected algorithm as an output observer [16], so that it will be informed when the connected algorithm has calculated a new output. Figure 2.14 shows the general update process for two algorithms connected in the described manner.
Once a `SVIAlgorithm` has called the `doUpdate` function of its subclass, it then informs all output observers through the `inputHasChanged` function that they have to update themselves. An `SVIAlgorithm` that has to update itself then collects the outputs of its input

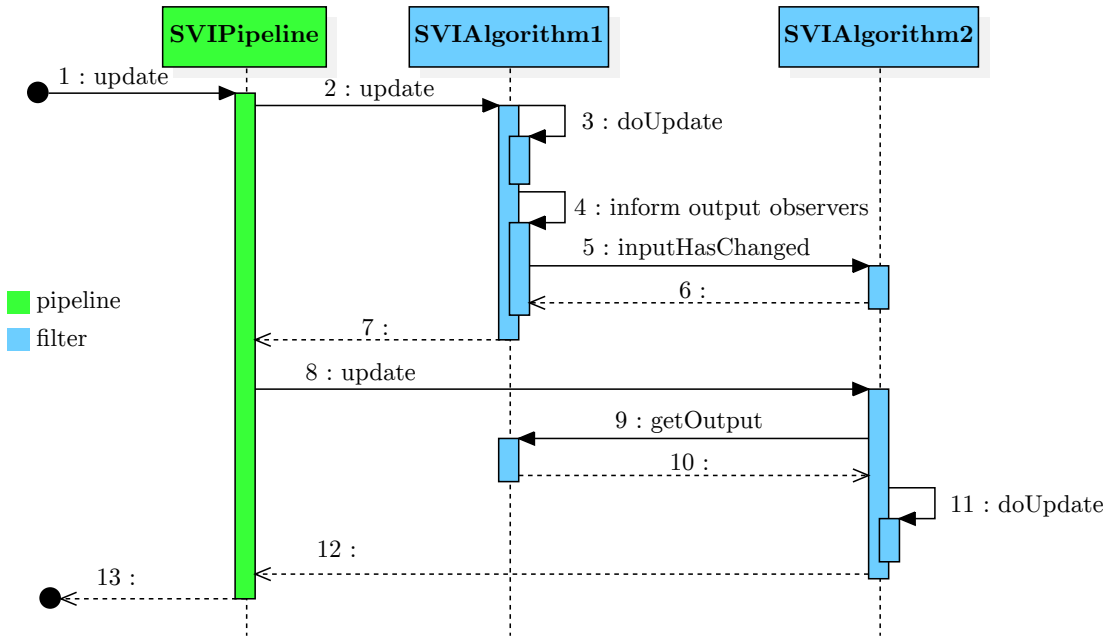

Figure 2.14.: Once an `SVIAlgorithm` updates its output value, it automatically informs all other `SVIAlgorithms` that depend on this output.

connections before calling the `doUpdate` function.
For a complete overview of the implementation of the Pipes and Filters design pattern see Appendix Figure A.1.

2.3.10. Constructing visualized objects and selections as pipelines

As shown in Figure 2.12, the subclass `SVIVisualized` of `SVIPipeline` is the base class actually used for all visualizations (objects and selections). The class `SVIVisualizedPolyData` is a specialization for VTK's main data format `vtkPolyData`. `SVIVisualizedSVIPolyData` and `SVIPolyDataSelections` are the actual implementations of visualized objects and selections as a pipeline of `SVIBasicAlgorithms`.

Composition of visualized objects

The class `SVIVisualizedSVIPolyData`, which represents a visualized object, forms a pipeline of the algorithms shown in Figure 2.15. The first algorithm gets a `SVIPolyData` object as input and generates a `vtkPolyData` object as output. The second algorithm then creates a `vtkActor`, which is VTK's equivalent for a visible object. The last filter then decides whether the object should be rendered and if so it generates a render request.

Composition of selections

A selection needs as input a `vtkPolyData` object. The fact that the `SVI2vtkPolyData-Algorithm` from `SVIVisualizedSVIPolydata` produces such an output, it can be used as an input connection for the first algorithm of `SVIPolyDataSelection`. Modeling a selection in this way leads to the fact that it will get automatically notified if the visualized object

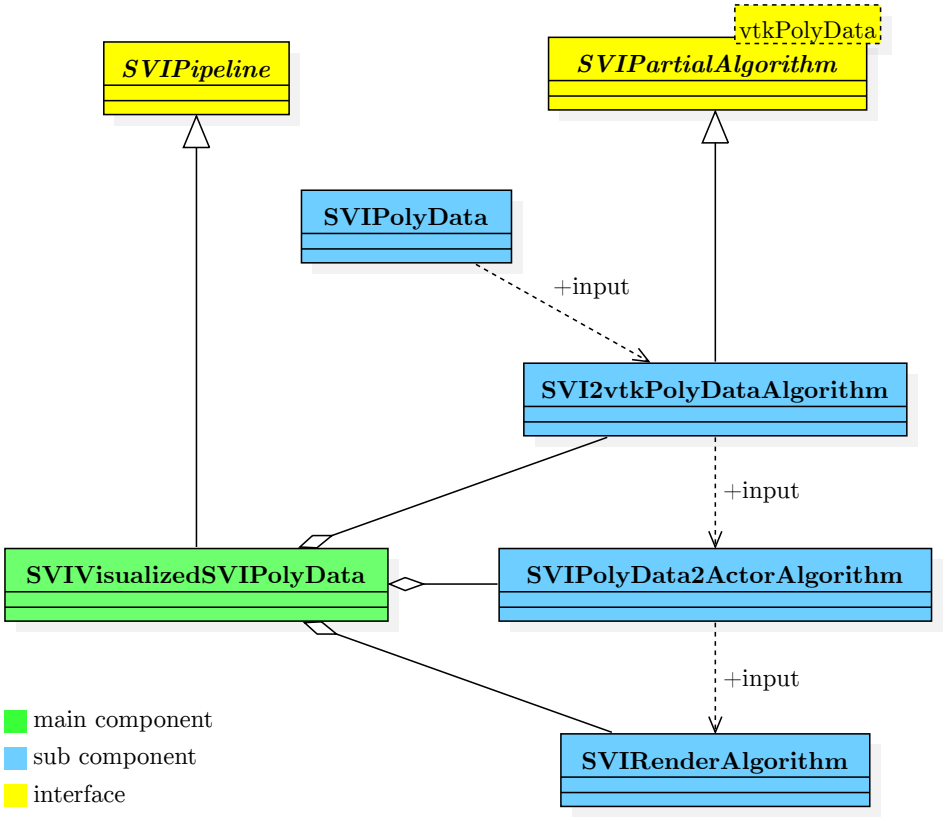

Figure 2.15.: A visualized object consists of several `SVIBasicAlgorithms`.

changes.
The first algorithm of `SVIPolyDataSelection` takes as input a `vtkPolyData`. It also takes a frustum created from the user selection on a window. From these two inputs, it calculates the IDs which lies inside this frustum. The next filter then takes these IDs and the IDs of all connected selections and calculates the final IDs for this selection that should be visible. The third filter takes the IDs as input and again calculates a `vtkPolyData` object from it. The last two algorithms are the same as used in `SVIVisualizedSVIPolydata`, which directly demonstrates the reusability of the new design. The fact that the third algorithm produces a `vtkPolyData` object representing the final selection, makes it possible to use a selection as input for a new selection as shown in Figure 2.16. This feature could theoretically be used to create a tree like selection structure, dividing a set of data from coarse-grained groups to fine-grain ones.

2.3.11. Allowing rapid prototyping through selection calculation

As described in Section 2.3.7, one pitfall in the first approach was that all selection arithmetic was directly coded within the `SVISelection` class, making it hard to extend. To circumvent this a Visitor design pattern [14] was introduced to separate the selection arithmetic from the actual selection implementation and make it interchangeable. Figure 2.17 shows the general structure of the implemented design pattern. The

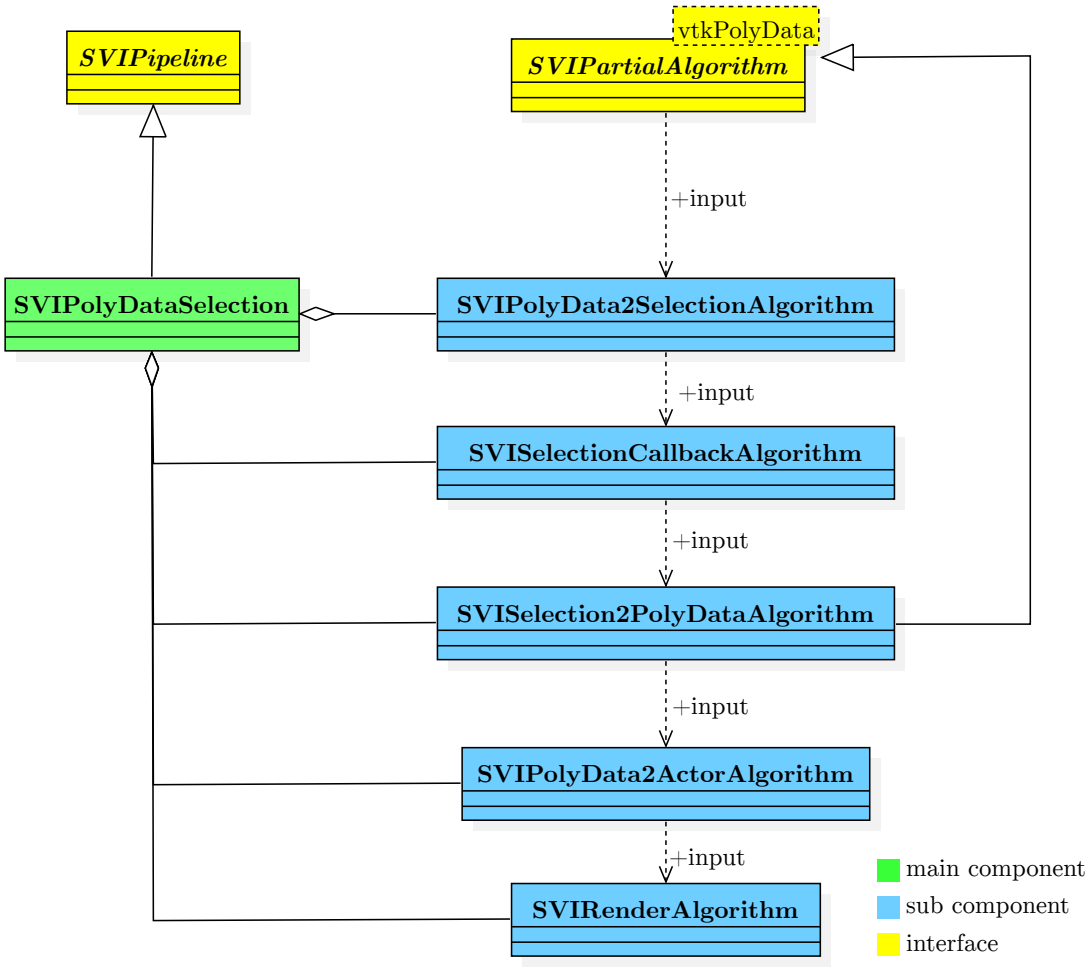

Figure 2.16.: A selection consists of several `SVIBasicAlgorithms` starting with the `vtkPolyData` output of a `SVIPartialAlgorithm`, which can be from a visualized object or selection.

`SVISelectionCallbackAlgorithm` is the second algorithm of the `SVIPolyDataSelection` pipeline described in Figure 2.16. Its input is a collection of connected selection IDs that share a common state. This state consists of the actual IDs which are visible in each of the connected selection.
The interface `SVICollectionAlgorithm` uses the `SVICollection` introduced in Figure 2.11 to create an abstract algorithm with a `SVICollection` as input. `SVISelectionCallbackAlgorithm` specializes the `SVICollectionAlgorithm` interface to use a `SVICollection` of selection IDs. `SVISelectionCallbackAlgorithm` itself uses in its `doUpdate` function a `SVICallbackAlgorithm` which is a generalized implementation of the Visitor design pattern, where the `SVICallbackProcessor` is the generalized Visitor interface [14]. `SVISelectionCallbackProcessor` specializes this interface to take a `SVICollection` of selection IDs and generate a new common state. Finally, `SVIStandartSelectionProcessor` is the default Visitor, which supports basic selection arithmetic like increasing or decreasing the common state about the nearly selected IDs.

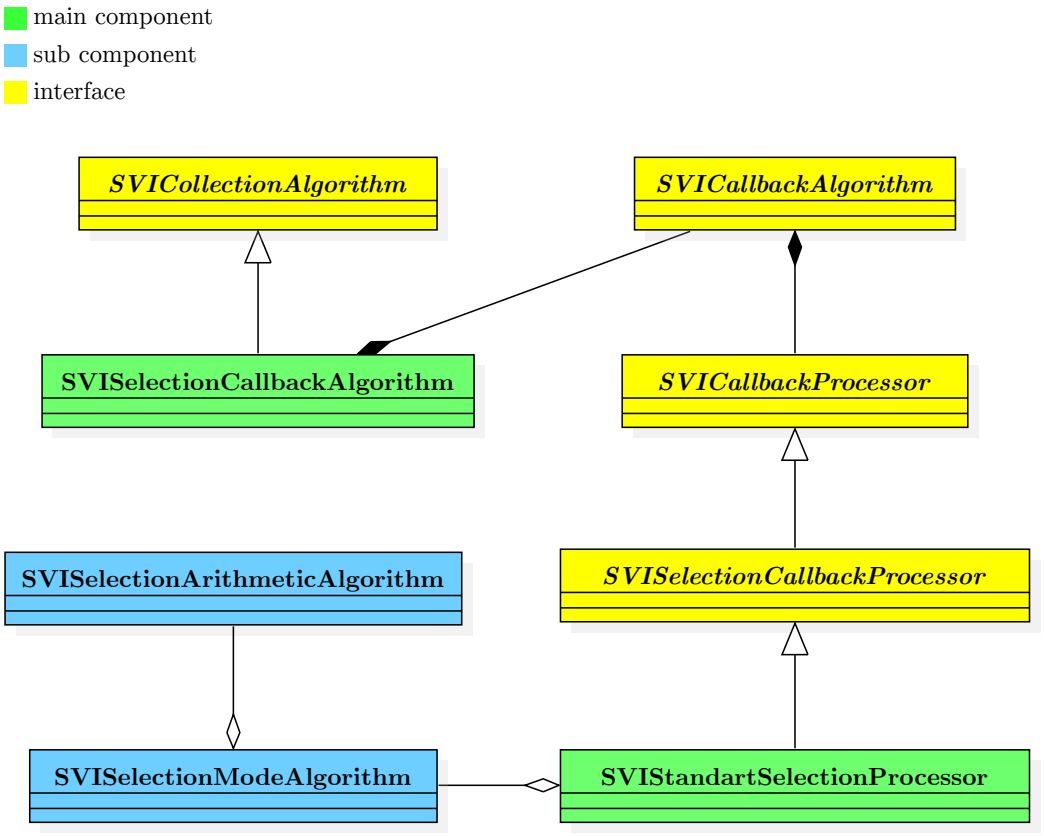

Figure 2.17.: The actual selection calculation is outsourced through the `SVISelectionCallbackProcessor` interface, in order to allow rapid prototyping. The `SVIStandartSelectionProcessor` is the default implementation of the selection processing.

2.3.12. Modeling properties of different data types

As mentioned in Section2.3.1, the `SVIProperties` wrapper in Matlap provides an interface to window, object and selection settings, like camera position, background color, opacity, visibility, etc. All these properties are basically modeled as a key-value storage, where the keys are Enums (a Enum is a set of unique identifiers) and the values can have different types like int, double, etc. For each different value type there is a different Enum class in order be able to determine the value type from the key. This simplifies polymorphism (using the same method name for different actions, which are determined by the method parameters). For example, using only the key to determine the return type of a get function. Within the C++ interface, properties were first modeled in the single class `SVIProperties` shown in Figure 2.18. Through the use of overloaded methods, one could easily set and obtain different types of properties. One must only write a set and get method for each different type and the C++ compiler chooses the right method at compile time.
However this model also had some pitfalls. First, since all property changes were directly applied, a context switch into the VTK thread was always necessary. Second, a class

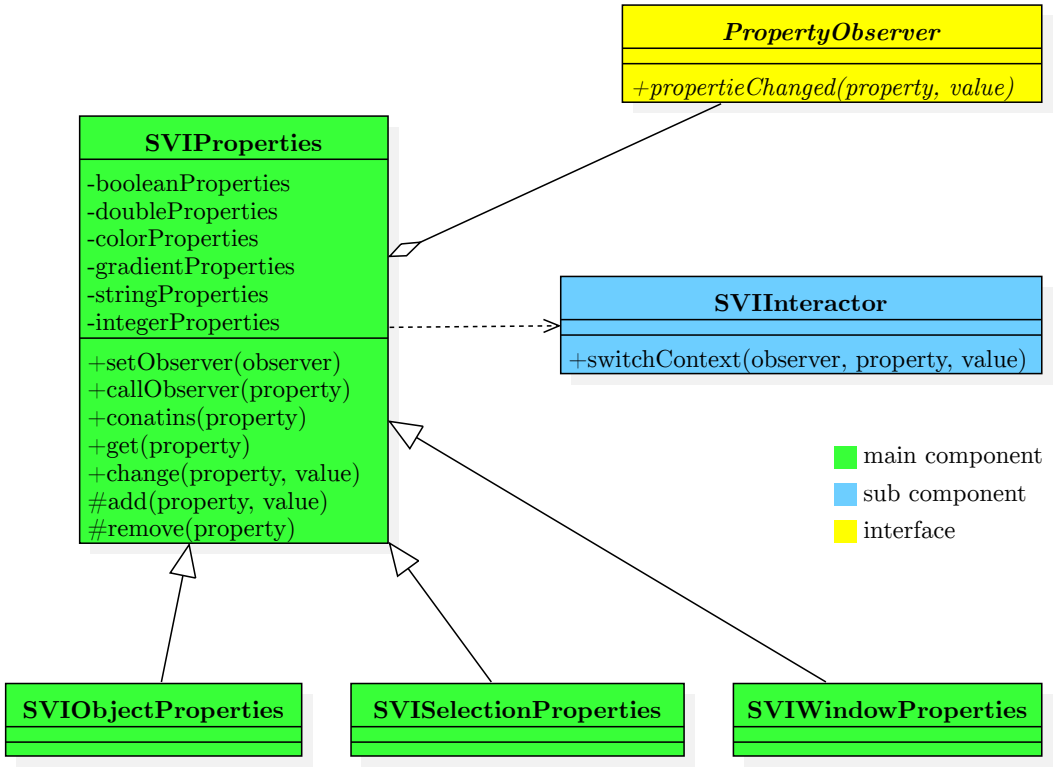

Figure 2.18.: Old properties implementation. For each new property implemented, all subclasses of PropertyObserver also had to be updated.

that implemented the `PropertyObserver` interface had to implement a `propertieChanged` method for each property data type. The big drawback was that if a new property type should be added, all classes implementing the `PropertyObserver` interface had to be modified. To circumvent this drawbacks the `SVIProperties` class as shown in Figure 2.19 was redesigned.
Now the observer is just a `SVIBasicAlgorithm` and once a property changes the `informObservers` method just calls the `inputHasChanged` function of the `SVIBasicAlgorithm` class. So the next time this algorithm gets updated it knows that the properties input has changed and processes an update.
To avoid the risk of infinity recursion if an algorithm would call `informObservers` on its input properties, the super class `SVIBasicProperties` was introduced to model a properties class without observers. Finally, to make it easy to add new property types the template class `SVISingleTypeProperties` was introduced, which handles the properties of a single type. `SVIBasicProperties` then implements easy access to the different property types through overloaded methods just like in the old design.
For a complete list of all available properties see Appendix A.1.

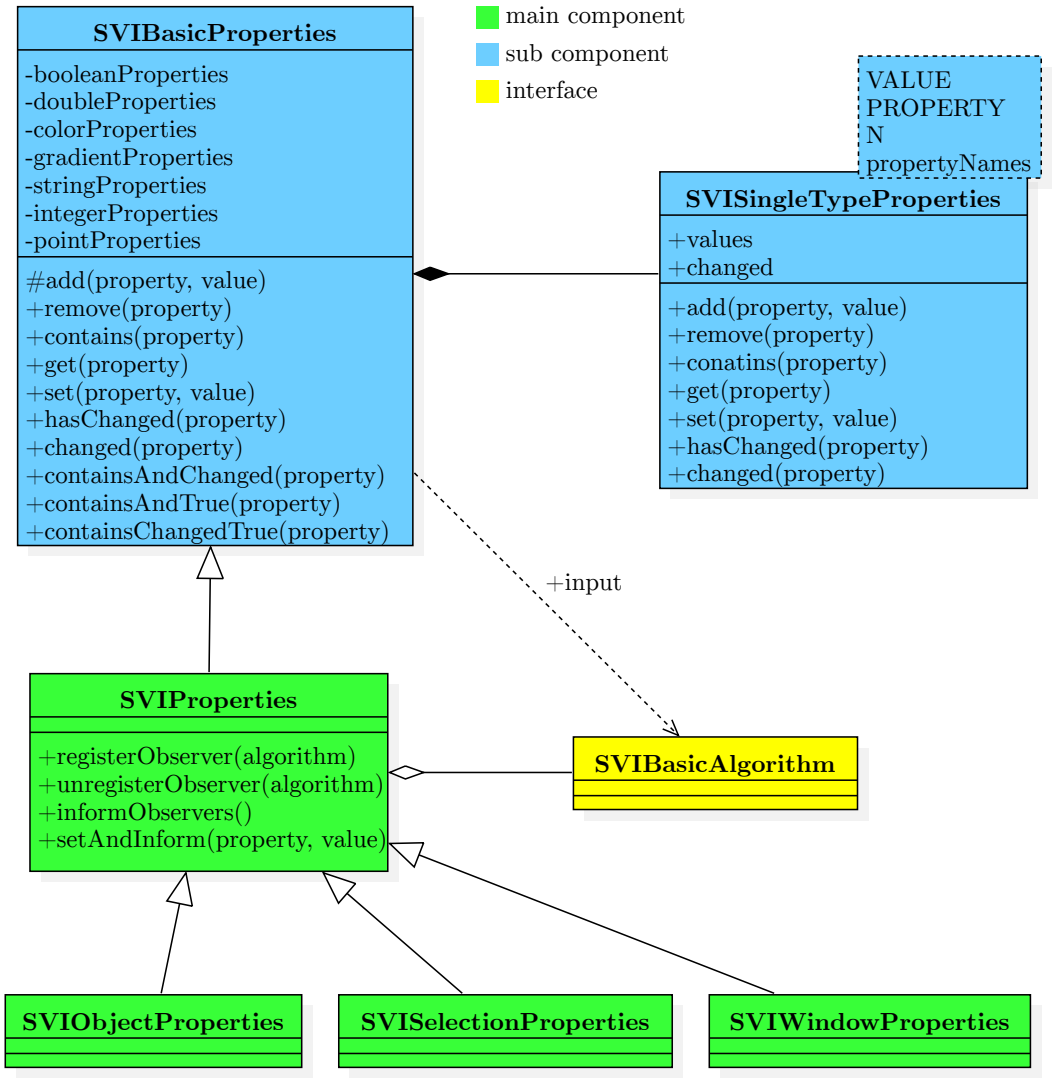

Figure 2.19.: New properties design making it easy to add a new property type, and make it work nicely with the SVIBasicAlgorithms as observers.

2.4. Capabilities of the developed Matlab VTK interface

The developed interface, namely SVI, as introduced in Section 2.3, enables a developer to easily build different user interfaces. These GUIs can be build on top of the components introduced in the Section 2.3.1. Through the Matlab interfaces `SVISelectionCallbackProcessor` and `SVIInteractionCallback`, described in Section 2.3.11 and 2.3.6, developers can react to user actions and model different visualization approaches by changing the default way in which connected selections interact.
Chapter 3 shows three GUIs build on top of the developed Matlab VTK interface.

3. Evaluation

This chapter will discuss the results of the software developed in this thesis. Besides presenting the different GUIs, which were build based on the developed Matlab VTK interface, also the requirements defined in Chapter 1 will be discussed.

3.1. Use cases

Based on the Matlab interface described in Figure 2.1 there were several user interfaces created within Matlab. The aim of these user interfaces was to assist a user within different tasks applicable on the datasets used within this thesis.

3.1.1. Using the interface to find groups of cells

The main intention of this Matlab VTK interface was to guide a knowledge discovery process with in large biological data by extending Matlab with VTK's capability of interactively visualizing data [37]. Figure 3.1 shows an example GUI built with this interface. Figure 3.1 A shows visualized full tracks. Figure 3.1 C and D show the single cells at two different developmental steps of the embryo. These two windows have a complete set of all developmental steps, but display only one of them. A user can change the visualized developmental step, by simply using the mouse wheel. For example, the user can scroll through all time points, getting a visual feedback on how the embryo develops over time. To visualize a specific time point, use Matlabs built-in command processor and call a specific GUI function.
More information about mouse and keyboard interactions can be found in Appendix A.2.
There are already three different groups of cells selected. View E shows only the full tracks from these groups allowing to inspect them easier. View B is a Matlab scatterplot showing in x direction the cell division rate, and the total track length in y direction. These feature combinations could already be used to detect cell groups, which form distinct local areas within the embryo.
To have a user-friendly interface, the groups from the scatterplot as well as from the VTK windows are connected. These groups are modeled by `SVIPolyDataSelection`, which is described in Section 2.3.10. They are also connected using the described techniques. As shown in Figure 3.2, the Matlab plot is modeled by the `SVIExampleScatterPlotProcessor` class, which implements the selection callback interface `SVISelectionCallbackProcessor`. So once a user selects a group within a VTK window, the `SVIExampleScatterPlotProcessor` will get called with the IDs of the selection. It can then update the scatterplot and return the selection IDs to the C++ interface. If a user selects within the scatterplot, it then forces a selections update, so that it gets called with the selection IDs. Afterwards it changes the

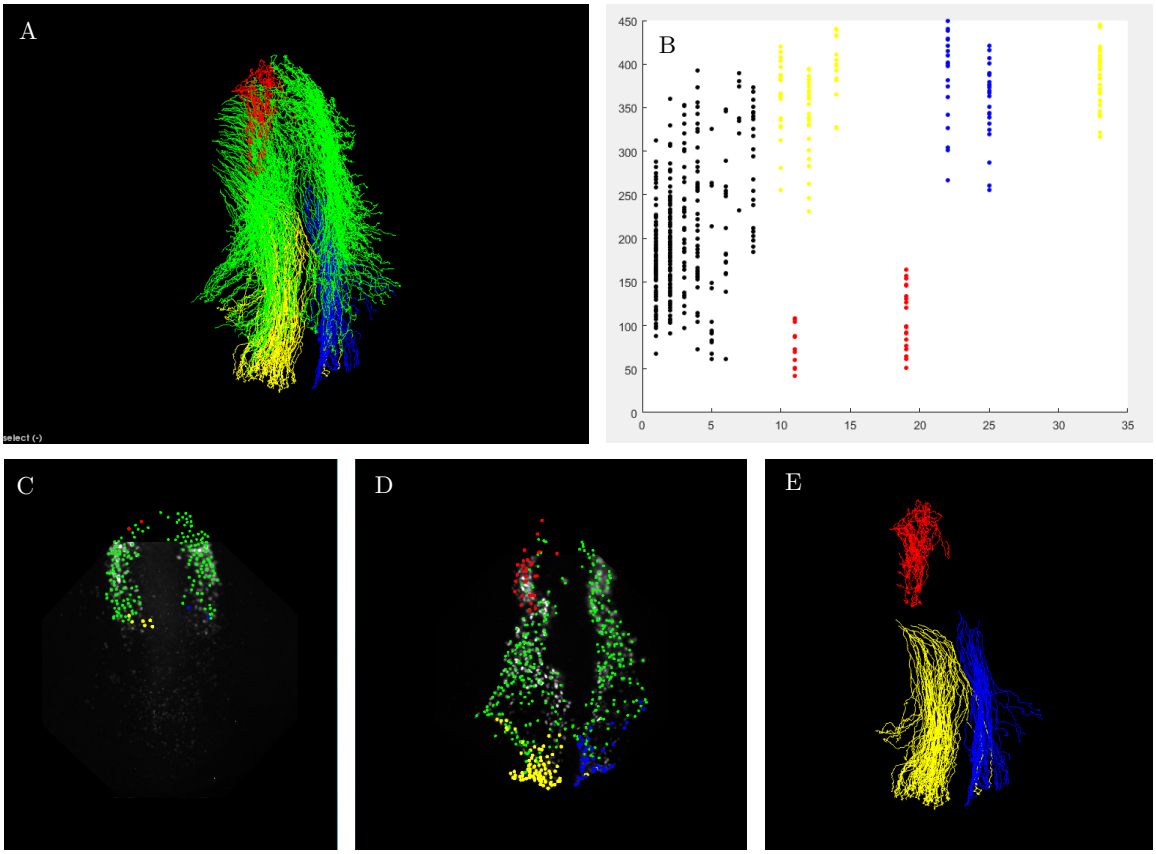

Figure 3.1.: Finding different cell groups through connected features and selections.

given selection IDs according to the user selection within the scatterplot and returns the new IDs to C++.
The two scrollable windows mentioned above, are modeled by the `SVIScrollableIDs` class. An object of this class can be configured for several different time steps. For each of these time steps a developer can add IDs, representing objects or selections. The `SVIScrollableIDs` class also implements the `SVIInteractionCallback` interface, enabling it to react to user actions like keyboard or mouse events. This makes it possible to change the visible time step, according to the use of the mouse wheel. To show one active time step within a VTK window, the `SVIScrollableIDs` class uses the properties of its `SVIIDs` to set active IDs to visible and inactive IDs to invisible. Through an object of `SVIScrollableIDs` the user can set a specific time step by calling the `setTimeStep` method with the desired time step number. The `SVIGroupSelection` is the overall control class which puts together the different windows and the scatterplot.

3.1.2. Inspecting tracklets on a maximum projection

Another interesting case where the framework can be useful to guide a knowledge discovery process, is the evaluation of full track generation. Originally, the algorithm generates tracklets, short trajectories of moving cells. To investigate why the algorithm loses the

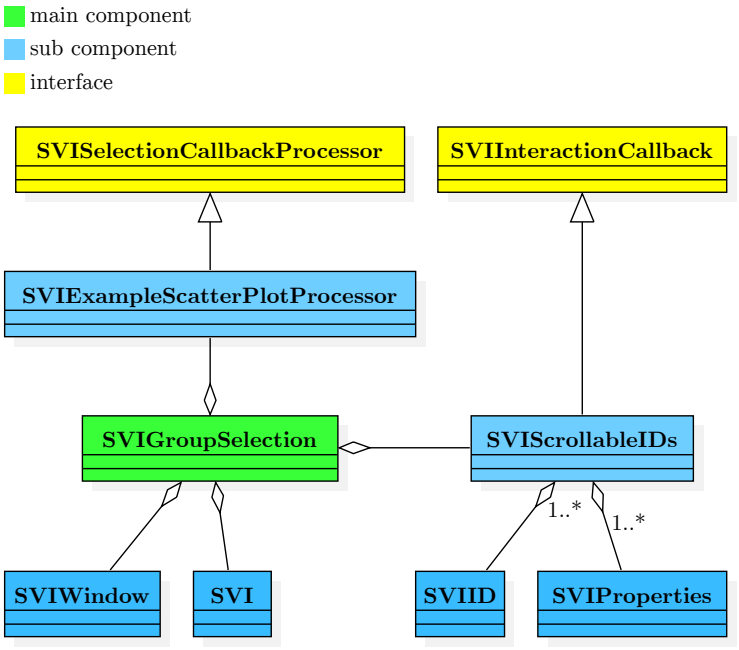

Figure 3.2.: General structure of the GUI shown in Figure 3.1. The class `SVIScrollable` models the scrollable views, `SVIExampleScatterPlotProcessor` models the Matlab feature plot and `SVIGroubSelections` puts all components together.

track of a cell, the GUI shown in Figure 3.3 was created by Tomas Antritter based on the Matlab VTK interface developed within this thesis.
Within this GUI, the user can manually select a cell on any of the three windows. The selected cell and its depending tracklet then gets highlighted within all windows. A cell ID can also be specified using the Matlab command input window. Also, for all currently selected cells the user can get the depending cell IDs via Matlab. By scrolling within any window, the user can see how the cell moves along its tracklet. Unlike the previous example where the two scrollable windows were independent, in this GUI all windows scroll simultaneously showing the user the same developmental step of the embryo from different point of views.

3.1.3. Inspecting tracklet successors in 3D within the whole embryo

The intention of the GUI introduced in Section 3.1.2 is to create a measurement to connect tracklets automatically in order to generate valid full tracks. However, if this measurement reveals that it is not clear how to automatically connect a given set of tracklets, the developed Matlab VTK interface can be used to inspect those tracklets in 3D and connect them manually. Such a basic approach is shown in Figure 3.4.
In this GUI, the user specifies a set of tracklets. The GUI, then automatically jumps to this tracklets and highlights them by the use of window and object properties. For example, it sets the camera position and focus, to let the user navigate around the end / start points of the given tracklets. In the shown case the segmentation algorithm lost the track of the

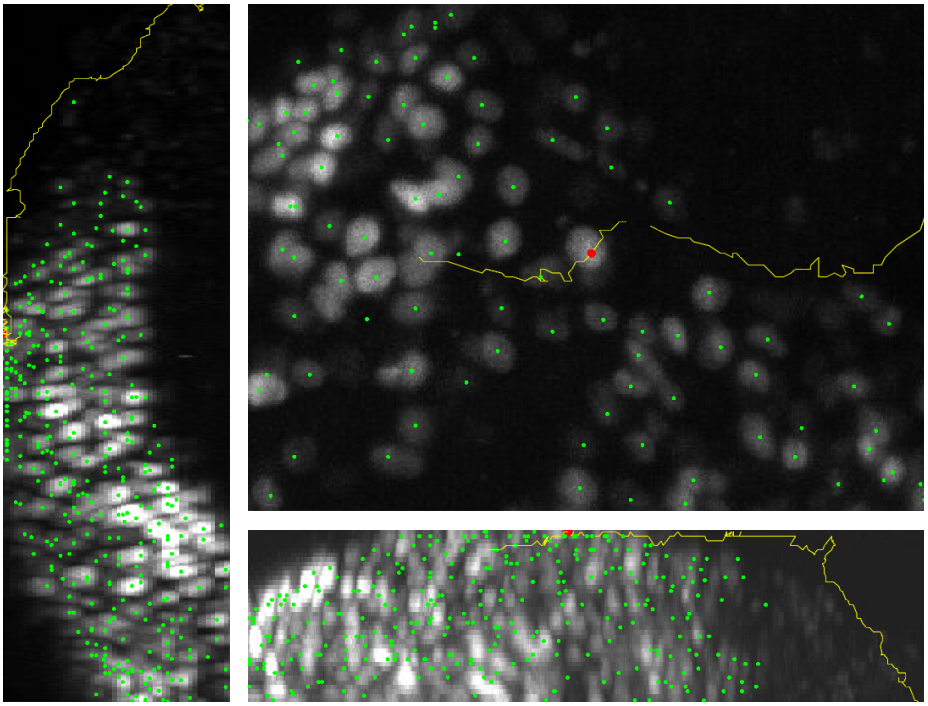

Figure 3.3.: inspecting one tracklet on the maximum projection.

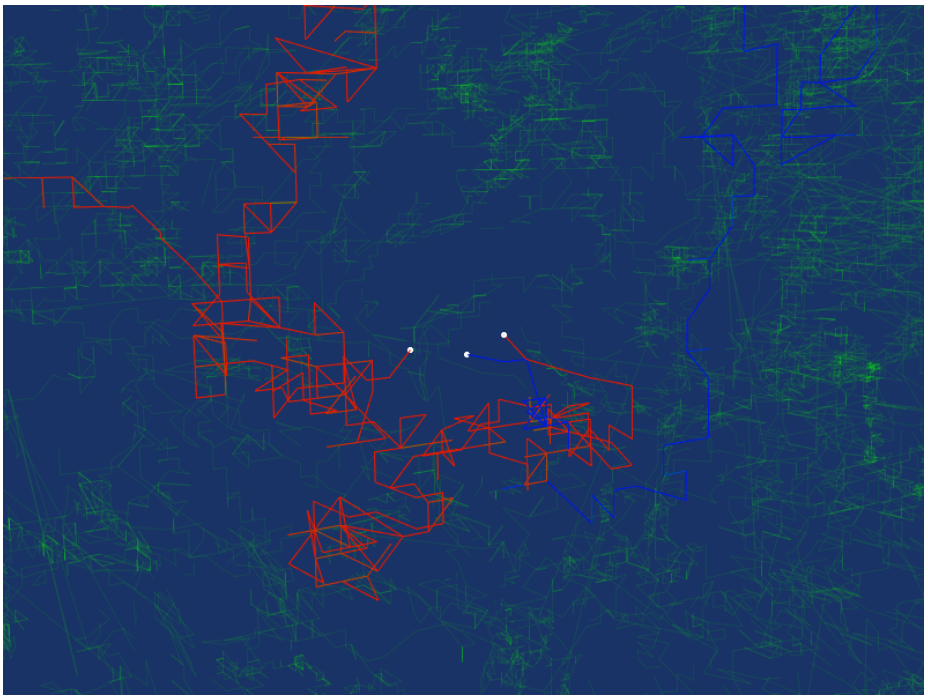

Figure 3.4.: Inspecting three tracklets that a part of a cell division.

blue tracklet due to the fact that a cell division occurred. In this case, the user could select both red tracklets as successors.

3.2. Software testing and requirements

To validate the correct function of the software, developed within this theses, unit test were used to test the basic components like selection processing, object connection, multi-threading, etc [33].

3.2.1. Benchmark creation

In order to test the actual user interface manual test were used. Since the result of these tests also depends on the input dataset, a benchmark creation software, shown in Figure 3.5, was developed. With this software, a developer could model an artificial embryo, and then start a simulation which generates full tracks. These full tracks mimic some characteristics of real biological full tracks like cell division and movement speed.

3.2.2. Original requirements

As introduced in Section 1.2.5, there already exist software tools which are capable of some but not all requirements of this thesis (see Table 1.1). Table 3.1 shows a comparison of these software tools with the developed Matlab VTK interface. It can be observed that the Matlab VTK interface (SVI) fulfills all main requirements, which will be described below.

	Feature based clustering	Connected feature plots	Matlab interface	Open source
Fiji	-	-	unidirectional	X
CATMAID	X	-	-	X
Mov-IT	-	-	-	X
Andrienko	X	X	-	-
SVI	X	X	X	X

Table 3.1.: Comparing the requirements with other software tools for interactive clustering

Matlab compatibility

There was not only a bidirectional interface developed, also rapid prototyping within Matlab is possible, so this target is completely fulfilled (see Section 2.3.3).

3D and 2D data representation

The GUI shown in Section 3.1.1 demonstrates that this goal was also reached.

Interactive clustering

As described in Section 3.1.1, interactive clustering is possible by selections within different views.

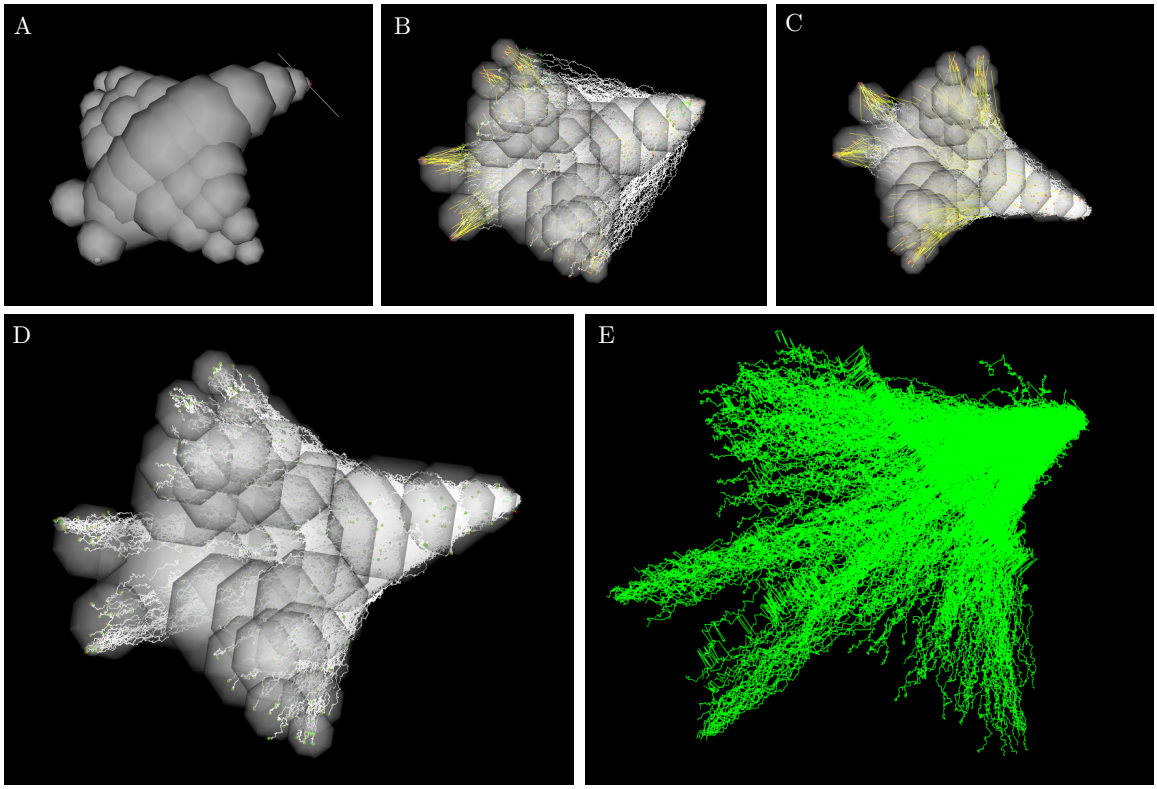

Figure 3.5.: Generating an artificial embryo to be used as a benchmark dataset. (A) Embryo modeling out of spheres. (B) First stage of simulation. Yellow lines visualize the distance for each trajectory between their endpoints to their desired endpoints. The algorithm tries to minimize the distance between each trajectory end point and its depending desired endpoint. The desired endpoints are randomly distributed across the embryo model in order to simulate cells. (C) Second stage of simulation. The algorithm takes into account that the trajectories should be inside the embryo model. Since the first stage is faster than the second one, the second stage is started after the first stage calculated a valuable result. (D) The finished simulation. (E) Visualization of the trajectories of the artificial embryo within a 3D view of the developed Matlab VTK interface.

Connected views

To form the GUI from Section 3.1.1, the different windows are connected in a way that an update of one window also triggers all other windows to update. Also the selection shown in Figure 3.1 are connected in a way that all windows show the same colored subsets of the original data.

Feature based clustering

The clusters from Figure 3.1 are actually selected through the feature plot, so this important target was also reached. However, it is only possible to connect one feature plot.

Feature plot

It is only possible to connect one feature plot, so this target was only partially reached. Unfortunately there still occurs an error when more then one plot is connected to a VTK view.

Easy extensibility

The different GUIs described in Section 3.1.1 - 3.1.3 demonstrate SVIs easy extensibility. So this side goal was also reached.

Handle large datasets

While it is possible to use VTK's full capability for 3D visualization, selection processing within Matlab is limited. Large datasets (several million data points) can't be handled in reasonable time in Matlab. In this case SVIs rapid prototyping capability should be used to generate a suitable GUI with a small dataset, and then rewrite the selection processing in C++ to increase the speed for the larger dataset.

Controlled by Matlab

As shown in Section 3.1, it is possible to write user interfaces in Matlab code. However in order to process large datasets it could be necessary to develop some parts in C++.

4. Conclusion

To interactively visualize and classify large 3D time resolved datasets, an adequate data representation is needed (e.g. a 3D view). In order to divide the dataset into specific regions of interest, manual and automatic cluster generation functionality is needed.
This thesis presents a novel Matlab VTK interface to guide a knowledge discovery process within large biological datasets. The main purpose is to interactively cluster the data based on specific features like cell division rate or trajectory length and to fine-tune those clusters manually. A GUI builds on top of the developed interface and uses the built-in capability of Matlab to generate visualizations (e.g. feature plots) and connects them with fast and interactive VTK 2D/3D views. This utilizes both major advantages of Matlab and C++, namely usability and speed. The developed approach is used to connect selections from different views, allowing a user to view and create clusters within different data representations. One of those representation is a 3D view for trajectories. In order to take advantage of the a priori knowledge of biologists, a novel view is introduced. In this view the tracked cells are represented along with the original microscopy image, since this is the data representation biologists are familiar with. In this view the user can visualize different developmental steps. This makes it possible to track a cell over a defined time span in an overlay visualization of cell tracks and microscopy images.
Through implementing a generic design, which makes use of callbacks for user interactions, the developed interface can easily be used to build different GUIs to fulfill highly specialized tasks. To quickly design and test various visualization approaches, the developed interface supports rapid prototyping within Matlab. By utilizing C++ callbacks from Matlab this prototypes are sufficient for small to medium dataset sizes. To process large datasets Matlab callbacks can easily be implemented as equivalent C++ callbacks, to process heavy tasks at the native speed of the VTK library.
To automate the process of image segmentation, tracking and interactive trajectory analysis, it is planned to include the Matlab VTK interface into the existing XPIWIT/Gait-CAD pipeline [4, 40]. Also the images in the superimposed views are currently maximum projections which could be adapted to support the original 3D images. Future development will also target user friendly packages for specialized use cases.

A. Appendix

A.1. Properties and their standard values

A.1.1. Boolean Properties

Visible (true) Controls whether an whole object or an selection is visible.

Active (true) Control whether a specific selection or an object is active (if it can be modified).

RenderWhenAdded (true) Controls whether an object should directly get rendered when added to the window.

ResetCameraWhenAdded (true) Controls whether the camera should be reset when an object is added.

ColorByScalars (false) Controls whether the color of an object should be defined by its scalars.

WindowParallelProjection (false) Controls whether the camera uses perspective or parallel projection.

RenderOnPropertyChange (true) Indicates whether to render an object / window on a property changed or not.

ResetCameraOnPropertyChange (true) Indicates that the camera should reset on a property change.

AllSelectionIDs (false) Indicates that one wants to have point IDs when using cell selection and vice versa.

OnCharUseIntern (true), OnKeyDownUseIntern (true), OnKeyUpUseIntern (true), OnKeyPressUseIntern (true), OnKeyReleaseUseIntern (true), OnMouseMoveUseIntern (true), OnLeftButtonDownUseIntern (true), OnLeftButtonUpUseIntern (true), OnMiddleButtonDownUseIntern (true), OnMiddleButtonUpUseIntern (true), OnRightButtonDownUseIntern (true), OnRightButtonUpUseIntern (true), OnMouseWheelForwardUseIntern (true), OnMouseWheelBackwardUseIntern (true) Indicates that the `SVIInteractor` should use the specific events.

UpdateSelectionOnStateChange (false) Indicates that the selection callback needs to be updated on a collection state change.

A.1.2. Double Properties

Opacity (1.0) Value between 0 an 1 to set the objects opacity.

PointSize (5.0) Value to set the point size of an object or selection.

ScalarRangeMin (not set) The minimum value of a scalar range.

ScalarRangeMax (not set) The maximum value of a scalar range.

CameraParallelScale (1.0) Zooming factor of the camera in parallel projection mode.

A.1.3. SVIColor Properties

PlainColor (random) The color of an object or selection).

WindowBackgroundColor (light blue) The background color for a specific window.

A.1.4. SVIColorFunction Properties

ColorFunction (red to white to blue) The color function of an object or selection.

A.1.5. String Properties

TextureImagePath (not set) The path to a texture image.

WindowInfoText (not set) The displayed info text of a window.

A.1.6. Integer Properties

WindowSelectionMode (SelectionModeAddition) The selection mode (addition or subtraction) for a specific window.

SelectionModeOverride (SelectionModeWindow) The selection mode for a specific selection (overrides the window selection).

RenderGroup (RenderDefault) The render priority of an object or selection.

WindowWidth (800) The width of a specific window.

WindowHeight (600) The height of a specific window.

A.1.7. SVIBasicPoint Properties

CameraPosition (not set) Camera position in a specific window.

CameraFocus (not set) Camera focus in a specific window.

CameraUpPosition (not set) Camera up position in a specific window.

A.2. Build in user commands

Changing the point of view By click and drag the user can move the camera around the focal point.

Zoom in and out Zooming is possible with the mouse wheel, or by right click-hold and moving the mouse.

Switching selection mode To switch to and from selection mode the user press the key r. This behavior is directly supported by VTKs `vtkInteractorStyleRubberBandPick`. Within the selection mode only 2D orientation is possible.

A.2.1. Build in Debug Mode

By pressing ESC the user / developer can enter an command / debug mode where he can enter commands starting with a colon.

:reset view Resets the camera focal point and position to its initial state and also resets the zoom to show all visible objects.

:set view X, :set view Y, :set view Z, :set view -X, :set view -Y, :set view -Z replaces the camera so that it shows the desired direction.

:set view X UP, :set view Y UP, :setview Z UP, :set view -X UP, :set view -Y UP, :setview -Z UP sets the cameras up position according to the typed command.

:list visible prints a list of all visible objects form the current window to the log file.

:list objects prints a list of all objects form the current window to the log file.

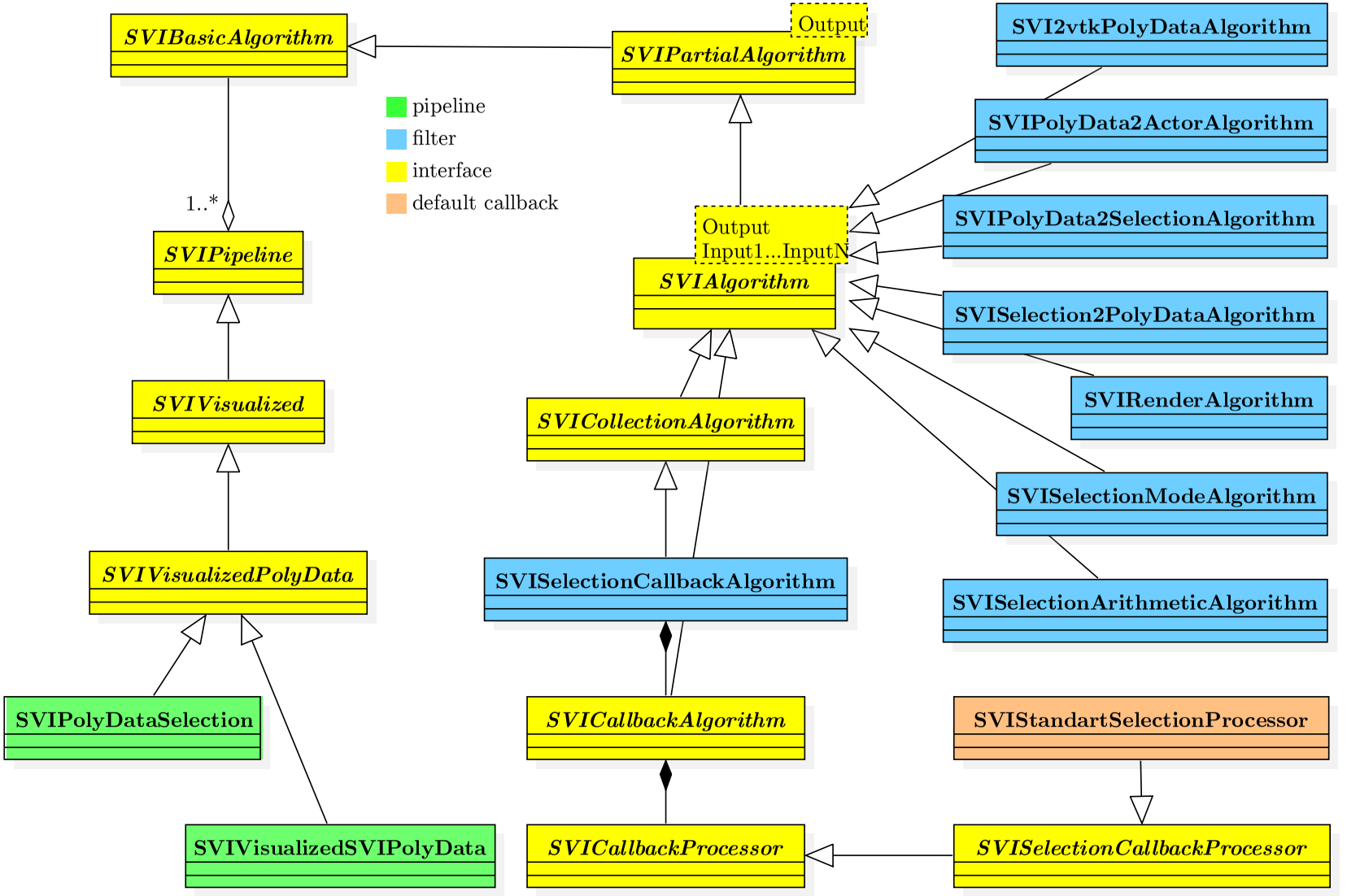

Figure A.1.: Overview of the implemented components and the general interaction of the Pipes and Filter design pattern [6]

A.3. Design patterns used within this thesis

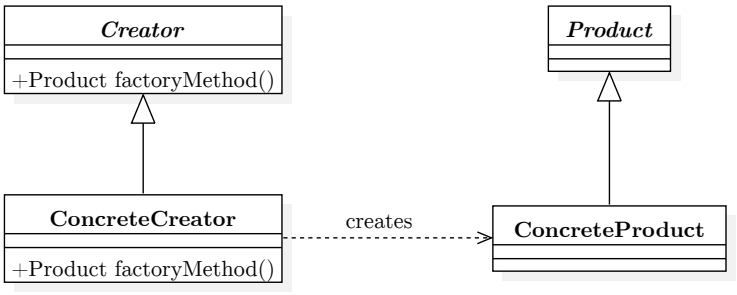

Figure A.2.: Factory Method design pattern. Creating specialized objects based on the call of method instead of an object constructor (adapted from [14]).

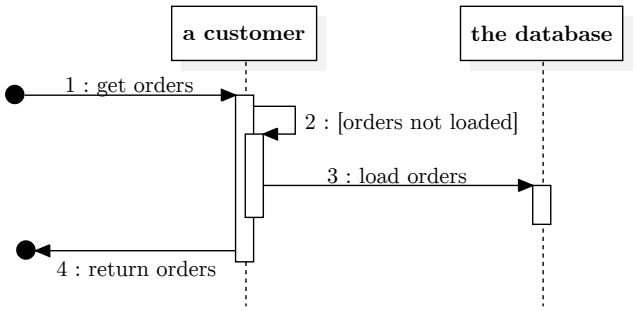

Figure A.3.: Lazy Load design pattern. The customer only loads an order the first time it is needed (adapted from [13]).

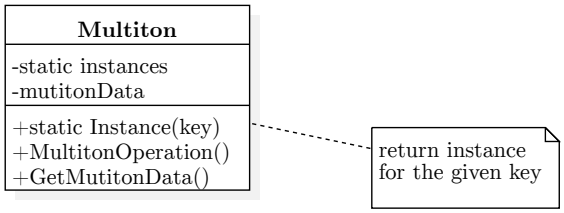

Figure A.4.: Multiton design pattern. Creating several distinguishable objects in a global manner. Trying to create an object with the same key results in a reference to existing object (adapted from [29]).

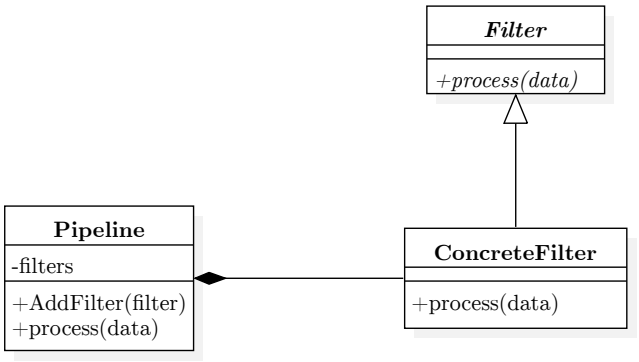

Figure A.5.: Pipes and Filters design pattern. Creating complex objects from small and reusable parts to form a flexible and easily adaptable system (adapted from [6]).

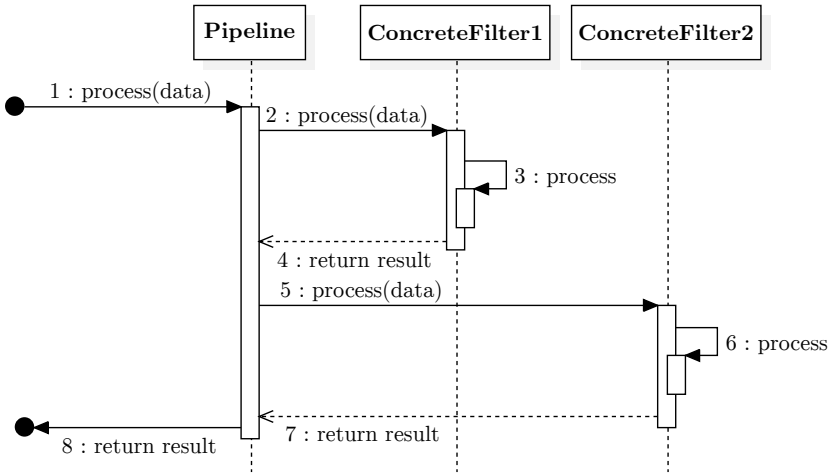

Figure A.6.: A pipeline in the Pipes and Filters design pattern processes data in a linear manner based on the order of its filters (adapted from [6]).

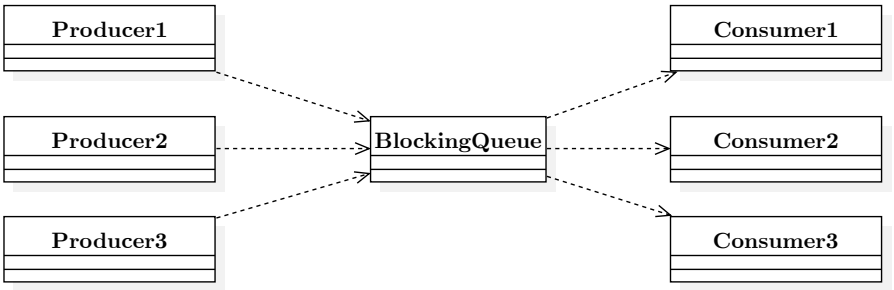

Figure A.7.: Producer Consumer design pattern. Synchronizing task creation and processing across threads (adapted from [17]).

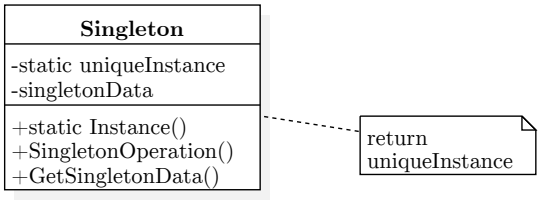

Figure A.8.: Singleton design pattern. Allows a class to have exactly one instance and provides global access to it (adapted from [9]).

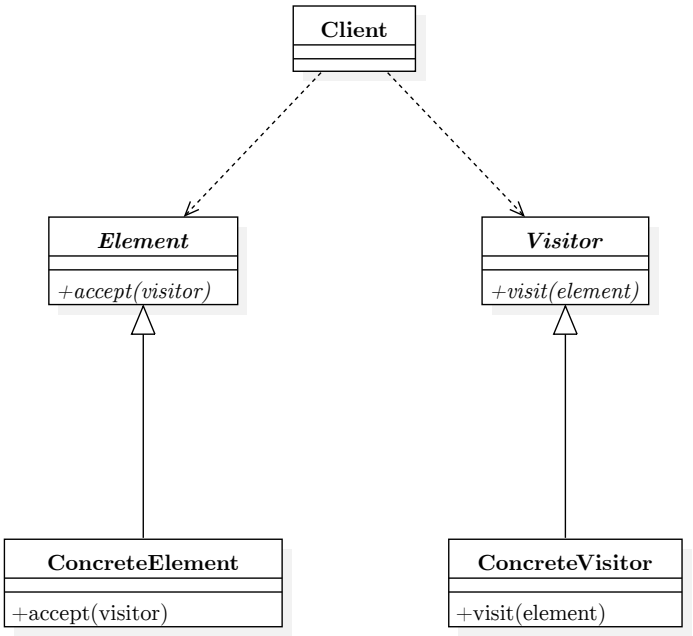

Figure A.9.: Visitor design pattern. Separating operations on a data structure from its implementation to make them interchangeable (adapted from [14]).

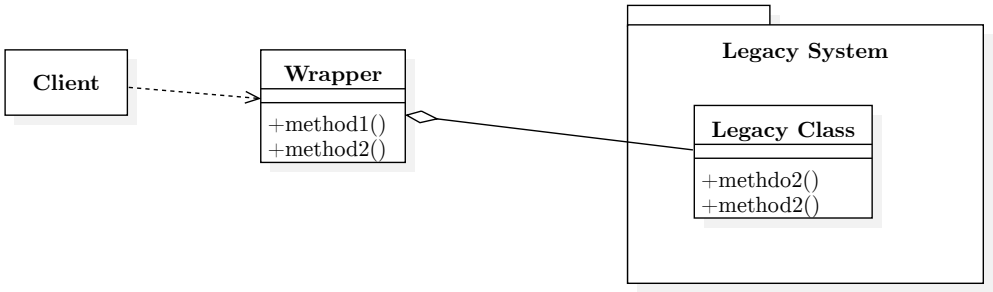

Figure A.10.: Wrapper design pattern. Making the implementation of a legacy system accessible to another system (adapted from [9]).

Bibliography

[1] Shaukat Ali et al. “Zebrafish Embryos and Larvae: A New Generation of Disease Models and Drug Screens”. In: *Birth Defects Research Part C: Embryo Today: Reviews* 93.2 (2011), pp. 115–133.

[2] Gennady Andrienko et al. “Interactive Visual Clustering of Large Collections of Trajectories”. In: *IEEE Symposium on Visual Analytics Science and Technology* (2009), pp. 3–10.

[3] Natalia Andrienko and Gennady Andrienko. *Exploratory Analysis of Spatial and Temporal Data: a Systematic Approach*. Springer Science & Business Media, 2006.

[4] Andreas Bartschat et al. “XPIWIT-An XML Pipeline Wrapper for the Insight Toolkit”. In: *Bioinformatics* (2015), btv559.

[5] Grady Booch. *The Unified Modeling Language User Guide*. Pearson Education India, 2005.

[6] Frank Buschmann et al. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[7] M. Daumas. *Scientific Instruments of the Seventeenth and Eighteenth Centuries and Their Makers*. 1972.

[8] Shao Jun Du et al. “Visualizing Normal and Defective Bone Development in Zebrafish Embryos Using Tthe Fluorescent Chromophore Calcein”. In: *Developmental Biology* 238.2 (2001), pp. 239–246.

[9] K. Eilebrecht and G. Starke. *Patterns Kompakt: Entwurfsmuster für Effektive Software-Entwicklung*. It Kompakt. Spektrum, Akad. Verlag, 2003.

[10] Kevin W Eliceiri et al. “Biological Imaging Software Tools”. In: *Nature Methods* 9.7 (2012), pp. 697–710.

[11] Longhou Fang and Yury I Miller. “Emerging Applications for Zebrafish as a Model Organism to Study Oxidative Mechanisms and Their Roles in Inflammation and Vascular Accumulation of Oxidized Lipids”. In: *Free Radical Biology and Medicine* 53.7 (2012), pp. 1411–1420.

[12] Emmanuel Faure et al. “A Workflow to Process 3D+ Time Microscopy Images of Developing Organisms and Reconstruct Their Cell Lineage”. In: *Nature Communications* 7 (2016), p. 8674.

[13] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[14] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] Joseph Goldstein et al. *Scanning Electron Microscopy and X-Ray Microanalysis: A Text for Biologists, Materials Scientists, and Geologists*. Springer Science & Business Media, 2012.

[16] J. Goll. *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java*. SpringerLink : Bücher. Springer Fachmedien Wiesbaden, 2014.

[17] M. Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*. Patterns in Java, Volume 1. Wiley, 2003.

[18] MATLAB Users Guide. "The Mathworks". In: *Inc., Natick, MA* 5 (1998), p. 333.

[19] Robert M Haralick and Linda G Shapiro. "Image Segmentation Techniques". In: *Computer Vision, Graphics, and Image Processing* 29.1 (1985), pp. 100–132.

[20] Shin-ichi Higashijima, Yoshiki Hotta, and Hitoshi Okamoto. "Visualization of Cranial Motor Neurons in Live Transgenic Zebrafish Expressing Green Fluorescent Protein Under The Control of The Islet-1 Promoter/Enhancer". In: *The Journal of Neuroscience* 20.1 (2000), pp. 206–218.

[21] Jan Huisken and Didier YR Stainier. "Selective Plane Illumination Microscopy Techniques in Developmental Biology". In: *Development* 136.12 (2009), pp. 1963–1975.

[22] Johnny Karlsson, Jonas von Hofsten, and Per-Erik Olsson. "Generating Transparent Zebrafish: A Refined Method to Improve Detection of Gene Expression During Embryonic Development". In: *Marine Biotechnology* 3.6 (2001), pp. 522–527.

[23] Philipp J Keller et al. "Reconstruction of Zebrafish Early Embryonic Development by Scanned Light Sheet Microscopy". In: *Science* 322.5904 (2008), pp. 1065–1069.

[24] Charles B Kimmel et al. "Stages of Embryonic Development of the Zebrafish". In: *Developmental Dynamics* 203.3 (1995), pp. 253–310.

[25] Thomas A Klar, Egbert Engel, and Stefan W Hell. "Breaking Abbes Diffraction Resolution Limit in Fluorescence Microscopy With Stimulated Emission Depletion Beams of Various Shapes". In: *Physical Review E* 64.6 (2001), p. 066613.

[26] Ulrike Langheinrich et al. "Zebrafish as a Model Organism for the Identification and Characterization of Drugs and Genes Affecting P53 Signaling". In: *Current Biology* 12.23 (2002), pp. 2023–2028.

[27] Shuang Fang Lim et al. "In Vivo and Scanning Electron Microscopy Imaging of Upconverting Nanophosphors in Caenorhabditis Elegans". In: *Nano Letters* 6.2 (2006), pp. 169–174.

[28] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. " O'Reilly Media, Inc.", 2012.

[29] J. Lott and D. Patterson. *Advanced ActionScript with Design Patterns*. Pearson Education, 2007.

[30] Ken Martin and Bill Hoffman. *Mastering CMake*. Kitware, 2015.

[31] Ralf Mikut et al. "Automated Processing of Zebrafish Imaging Data: A Survey". In: *Zebrafish* 10.3 (2013), pp. 401–421.

[32] Ralf Mikut et al. “Die MATLAB-Toolbox Gait-CAD”. In: 16 (2006), pp. 114–124.

[33] Roy Osherove. *The Art of Unit Testing*. MITP-Verlags GmbH & Co. KG, 2015.

[34] Stephan Saalfeld et al. “CATMAID: Collaborative Annotation Toolkit for Massive Amounts of Image Data”. In: *Bioinformatics* 25.15 (2009), pp. 1984–1986.

[35] Peter A Santi. “Light Sheet Fluorescence Microscopy a Review”. In: *Journal of Histochemistry & Cytochemistry* 59.2 (2011), pp. 129–138.

[36] Johannes Schindelin et al. “Fiji: an Open-Source Platform for Biological-Image Analysis”. In: *Nature Methods* 9.7 (2012), pp. 676–682.

[37] Benjamin Schott et al. “Challenges of Integrating A Priori Information Efficiently in the Discovery of Spatio-Temporal Objects in Large Databases”. In: *arXiv preprint arXiv:1602.02938* (2016).

[38] Greenfield Sluder and David E Wolf. *Digital Microscopy*. Vol. 114. Academic Press, 2013.

[39] Johannes Stegmaier et al. “Fast Segmentation of Stained Nuclei in Terabyte-Scale, Time Resolved 3D microscopy image stacks”. In: *PloS one* 9.2 (2014), e90036.

[40] Johannes Stegmaier et al. “Information Fusion of Image Analysis, Video Object Tracking, and Data Mining of Biological Images Using the Open Source MATLAB Toolbox Gait-CAD”. In: *Biomedical Engineering/Biomedizinische Technik* 57.SI-1 Track-B (2012), pp. 458–461.

[41] Robert H Webb. “Confocal Optical Microscopy”. In: *Reports on Progress in Physics* 59.3 (1996), p. 427.

[42] Monte Westerfield. *The zebrafish book: a guide for the laboratory use of zebrafish (Danio rerio)*. University of Oregon press, 2000.

[43] Tony Wilson. “Confocal Microscopy”. In: *Academic Press: London, etc* 426 (1990), pp. 1–64.