Documentation of Shape Analyzer

Version 1.0

Emanuel Laude[1] und Zorah Lähner[2]

May 9, 2015

1 emanuel.laude@in.tum.de
2 laehner@in.tum.de

Contents

CHAPTER 1

Introduction

This report gives an overview over the functionalities of the Shape Analyzer application. The code can be found in the CVPR SVN under the project name ShapeAnalyzer. The main purpose of this document is to give an introduction into integrating a new algorithm into the framework and does therefore not contain information about every class. The code is throughly documented though. Instructions how to get the documentation can be found in chapter 6.2.1.

CHAPTER 2

Installation

ShapeAnalyzer is completely written in `C++` heavily making use of the recent `C++11` standart in the code. Moreover it builds up on the frameworks `VTK` version 6.1.0 or greater and `Qt` version 5.0 or greater for the visualization and rendering of the shapes and the creation of the graphical user interface respectively. For all matrix and vector related computations including the computation of the Laplace-Beltrami eigenvectors the libraries `PETSc` (basic linear algebra including basic solvers for linear systems) and `SLEPc` (sparse eigensolver) are used. (PETSc and SLEPc - the funny dogs. This sounds like the names of funny cartoon series characters for kids.)
In order to compile *ShapeAnalyzer* it has to be ensured that both, a recent `C++` compiler that supports the major part of the `C++11` standard (e.g. `gcc` 4.7 or newer) as well as `cmake` version greater 2.8 is available . Furthermore all the aforementioned libraries and frameworks have to be installed.

2.1 Installation of Qt5

Since the most recent version of Qt currently (Dezember 2014) available via apt-get on Ubuntu is less than 5, it is recommended to download the most recent pre-compiled Qt5 package from the homepage of Qt: For installation just launch the installation assistant.

Hint: In case the installation file cannot be opened and executed you may have to make it executable via

```
sudo chmod +x <filename>
```

2.2 Installation of VTK

As it is the case for Qt the most recent version of VTK currently (Dezember 2014) available via apt-get on Ubuntu is less than 6.1. Therefore it is recommended to compile and install `VTK` version 6.1.0 or newer from source. It is also important that VTK is compiled with a link to your Qt version.

- First download the source code and extract it via

```
cd ~/Downloads
wget http://www.vtk.org/files/release/6.1/VTK-6.1.0.tar.gz
```

```
tar xvf VTK-6.1.0.tar.gz
cd ./VTK-6.1.0
```

- Execute CMake using the following options:

  ```
  cmake -DVTK_QT_VERSION:STRING=5 \
        -DQT_QMAKE_EXECUTABLE:PATH=~/Qt/5.4/gcc_64/bin/qmake \
        -DVTK_Group_Qt:BOOL=ON \
        -DCMAKE_PREFIX_PATH:PATH=~/Qt/5.4/gcc_64/lib/cmake \
        -DBUILD_SHARED_LIBS:BOOL=ON \
        ~/Downloads/VTK-6.1.0
  ```

- Compile the code via

  ```
  make
  ```

2.3 Installation of PETSc

Since PETSc is designed for the usage with MPI it has to be compiled explicitly without MPI.

- First download the source code and extract it via

  ```
  cd ~/Downloads
  wget http://ftp.mcs.anl.gov/pub/petsc/release-snapshots/petsc-3.5.2.tar.gz
  tar xvf petsc-3.5.2.tar.gz
  cd ./petsc-3.5.2
  ```

- Execute the configure script as follows (By setting the option `--with-mpi=0` we tell the compiler that PETSc is used without MPI in single-processor-mode)

  ```
  ./configure --with-fc=0 --download-f2cblaslapack --with-mpi=0
  ```

- The code is compiled via

  ```
  make PETSC_DIR=~/Downloads/petsc-3.5.2 PETSC_ARCH=arch-linux2-c-debug all
  make PETSC_DIR=~/Downloads/petsc-3.5.2 PETSC_ARCH=arch-linux2-c-debug test
  ```

  If this does not work, follow the instructions given by PetsC.

- Finally the following environment variables have to be set (e.g. in `~/.pam_environment`)

  ```
  export PETSC_DIR=~/Downloads/petsc-3.5.2
  export PETSC_ARCH=arch-linux2-c-debug
  ```

2.4 Installation of SLEPc

SLEPc is an add-on for PETSc and, as stated in the beginning, it is responsible for solving sparse eigenvalue problems. Therefore SLEPc can only be installed after PETSc has been installed.

- First download the source code and extract it via

```
cd ~/Downloads
wget http://www.grycap.upv.es/slepc/download/download.php?filename=slepc
-3.5.3.tar.gz -O slepc-3.5.3.tar.gz
tar xvf slepc-3.5.3.tar.gz
cd ./slepc-3.5.3
```

- Execute the configure script

  ```
  ./configure
  ```

- The code is compiled via

  ```
  make SLEPC_DIR=$PWD PETSC_DIR=~/Downloads/petsc-3.5.2 PETSC_ARCH=arch-linux2
  -c-debug
  ```

  If this does not work, follow the instructions given by SlepC.
- Finally the following environment variables have to be set (e.g. in `~/.pam_environment`)

  ```
  export SLEPC_DIR=~/Downloads/slepc-3.5.3
  ```

2.5 Compiling the project

- First download the source code from the CVPR SVN repository.
- Create a directory for the build and execute `CMake` using the following options:

  ```
  cd ./src
  mkdir build
  cd ./build
  cmake -DVTK_DIR:PATH=~/Downloads/VTK-6.1.0 \
        -DQT_QMAKE_EXECUTABLE:FILEPATH=~/Qt/5.4/gcc_64/bin/qmake \
        -DCMAKE_PREFIX_PATH:PATH=~/Qt/5.4/gcc_64/lib/cmake \
        ..
  ```

- Compile the code via

  ```
  make
  ```

- Have fun!

CHAPTER 3

Classes

All classes are organized in 3 folders: domain, view and custom. Domain contains all classes that purely contain some algorithm logic. View contains classes that are responsible for the GUI and custom also contains GUI classes, but only menu items and tabs which are created by the Factory. Each folder has its own namespace.

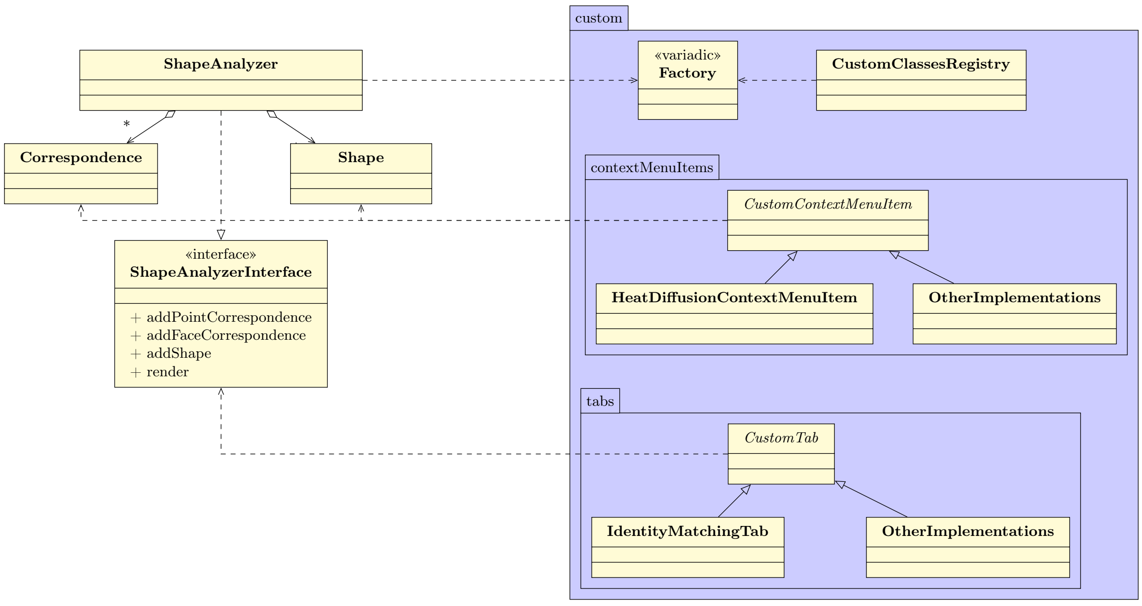

Figure 3.1: UML diagram of the most important classes in view and custom.

3.1 ShapeAnalyzer

The ShapeAnalyzer is the main application and manages all user input as well as the existing Shapes and Correspondences. The ShapeAnalyzer should only be changed if really necessary. It is also a QWidget, so it can (and should) be used as the parent for further QWidgets or Qt objects that have a parent widget input. All new Shapes and Correspondences have to be added through this main application and not by directly adding them into the data structures (which is not possible most of the time anyway) in order to keep the whole application up-to-date.

3.1.1 ShapeAnalyzerInterface

The ShapeAnalyzerInterface is an interface that provides functions to add new Shapes, Correspondences and Render the GUI, basically everything that the custom objects are allowed to do. The custom objects only get a reference to the interface. Theoretically the ShapeAnalyzerInterface can be forcefullly cast into a ShapeAnalyzer object but you should think about whether there might be a different solution. It also prevents cyclic dependencies.

3.2 Shape

A Shape contains the mesh data as well several VTK related objects. It has a unique id and a (not necessarily unique) name. The data (vtkPolyData object) can be obtained through `getPolyData()`. Additionally to some self-explanatory functions the surface can be colored with `setColoring`, see 3.2.1.

3.2.1 Coloring

Coloring is a struct in the Shape class. It contains a Type and a vtkDataArray. Either Type is possible as Point or Face. Then the vtkDataArray has to have as many entries as the shape has vertices or faces respectively and every vertex or face will be colored with the value(s) in the same row as their id. It is not possible (to my knowledge) to leave some vertices or faces uncolored. The possible Types are:

- **`Coloring::Type::PointScalar/FaceScalar`** `vtkDoubleArray` The minimum value will be blue, the maximum red and all values inbetween have a smooth color gradient.
- **`Coloring::Type::PointRGB/FaceRGB`** `vtkUnsignedCharArray` Set the number of components of the array to 3 with `SetNumberOfComponents(3)`, the components will be interpreted the R, G and B parts and have to be values between 0 and 255.
- **`Coloring::Type::PointSegmentation/FaceSegmentation`** `vtkIntArray` Every integer denotes one region that will be colored in the same color. VTK will automatically color in a way that allows to differ separate regions. If the coloring is set to an segmentation, the region can be extracted as a new shape in the GUI.

These are just the standard behaviors of each type. The vtkMapper and vtkLookUpTable of every Shape have more options that can be explored.

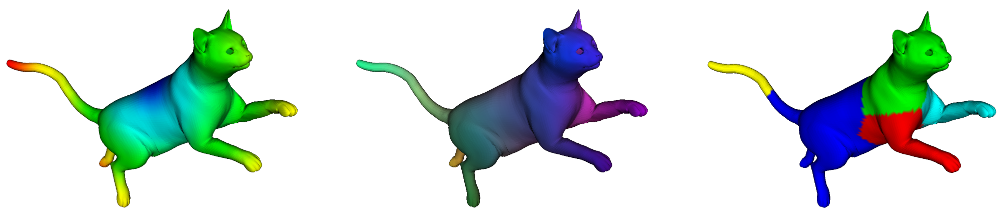

Figure 3.2: (i) PointScalar, (ii) PointRGB using the coordinates as RGB values, (iii) PointSegmentation

3.3 Correspondence

A Correspondence is a set of tupels, each containing a shape and an id. There are PointCorrespondences and FaceCorrespondences, the id either belonging to a vertex or a face. The tupels are realized as two vectors where objects in the same row belong to the same tupel. Correspondences can be visualized in the GUI with corresponding VisualCorrespondence objects which are organized (and created) by the ShapeAnalyzer.

3.4 Factory

The Factory is responsible for automatically creating objects of Custom classes (see chapter 4) and making them visible in the GUI. It is a variadic template that allows to specify the number and type of input parameters the constructors of those objects will get. It is realizing the Factory (you might have guessed it from the name) and the Singleton design pattern.

If you only want your menu item / tab to appear in the GUI, no deeper knowledge of the Factory is required. Take a look at chapter 4.3.

```
template<class T, class... Args>
class Factory {
    ...
    /// \brief Returns the unique instance of Factory<T>.
    /// \details This is the only way to obtain the Factory object for type T.
    static Factory<T, Args...>* getInstance() {
        static Factory<T, Args...> instance;
        return &instance;
    }
    ...
};
```

Listing 3.1: Factory.h

Due to the Singleton pattern there is only one Factory instance of every type. You can retrieve it by calling Factory<T, Args...>::getInstance() with the right template parameters. For example for CustomContextMenuItems there exists this Factory:

```
typedef Factory<CustomContextMenuItem, shared_ptr<Shape>, ShapeAnalyzerInterface
*> CustomContextMenuItemFactory;
```

This Factory can produce CustomContextMenuItem objects that get a Shape and a ShapeAnalyzerInterface pointer as input parameters. As CustomContextMenuItem is an abstract class, the concrete classes have to be registered with a string identifier and a label. The string identifier has to be unique among all registered classes, the label will be shown in the GUI.

```
template<class C>
    void Register(const string& identifier, const string& label) {
        // inserts a pair consisting of the identifier and a c++11 lambda
        // expression calling the constructor of class C into the map
        // constructors
        function<T*(Args...)> constructor([](Args... args)->T*
            { return new C(args...); });
        contructors_.insert(pair<string, function<T*(Args...)
            >>(identifier, constructor));

        labels_.push_back(pair<string, string>(identifier, label));
        labelIndex_.insert(pair<string, int>(identifier, labels_.size() - 1));
}

T* create(const string& identifier, Args... args) {
        // execute create function pointer given the arguments in tuple args.
        return contructors_.at(identifier)(args...);
    }
```

Listing 3.2: Factory.h

The register function is a template taking the concrete class as the template parameter `C`. It will create and store a lambda expression that calls the constructor of `C` with input parameters that were given as template parameters to the Factory template. Afterwards the create function will take the identifier and input parameters and use the stored constructors to create new objects. As the identifiers are stored separately, they can be used to automatically create objects of all registered classes. This happens when the menu is called.

```
CustomContextMenuItemFactory::getInstance()>
Register<VoronoiCellsContextMenuItem<GeodesicMetric>>
("voronoicells_geodesic", "Segmentation>>Voronoi Cells>>Geodesic");
```

The label parameter allows to create layered menus. For menu items the label will be separated at every », creating a new sub menu. Menu items that have the same label up to some level will be put in the same sub menu. For tabs the label has to start with `"Shapes»"` or `"Correspondences»"` and no further sub menus will be created.

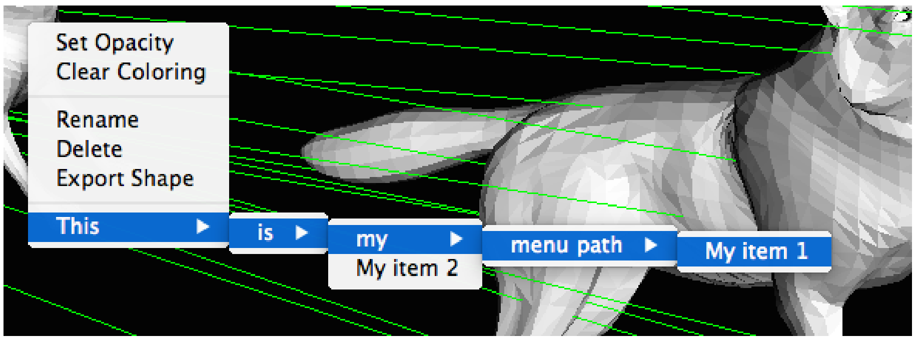

Figure 3.3: The label "`This»is»my»menu path»My item 1`" will produce a menu like this.

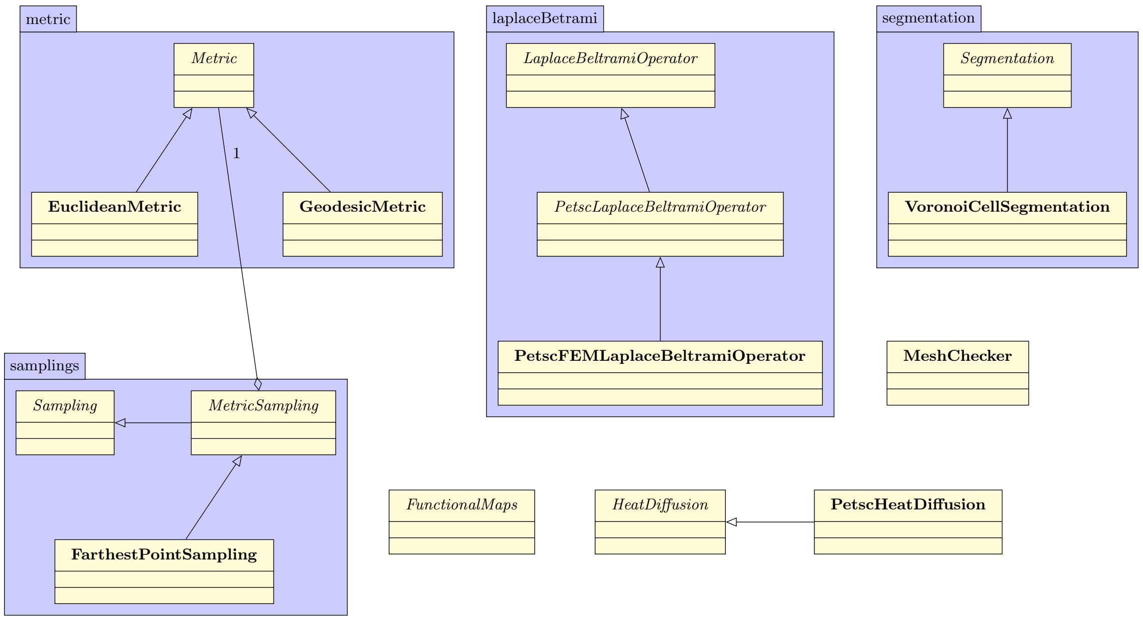

Figure 3.4: Subset of classes in the domain folder.

3.5 Domain

The domain folder consists of several subfolders which contain different kinds of classes that inherit from the same abstract class or belong together somehow. Feel free to add more folders if you think it makes sense. Each folder has its own namespace and Error class (see 5.2).

CHAPTER 4

Customs

Customs refer to Qt objects that you can add to the GUI without changing the main application. There are CustomContextMenuItems (4.1) and CustomTabs (4.2). CustomContextMenuItems allow you add menu items to the right click menu on shapes. It is possible to open input dialogs to get certin parameters for the function but if you need more complex inputs or outputs you are better off with a CustomTab. To make your Customs visible in the GUI you have to register them (see 4.3).

4.1 CustomContextMenuItem

- **Have to implement** CustomContextMenuItem
- **Found in** custom/contextMenuItems
- **Use when** your algorithm works with exactly one shape and has limited input parameters
- **Naming convention** <MeaningfulName>ContextMenuItem

The abstract CustomContextMenuItem class is very limited and there is only one function to implement. See 4.1.3 for an example.

4.1.1 Abstract Functions

- **`onClick(vtkIdType pointId, vtkIdType faceId, QWidget* parent)`** Is triggered when the menu item was clicked.
  - `pointId` refers to the id of the closest vertex of the shape in the vtkPolyData of the shape to where the click was.
  - `faceId` refers to the id of the face of the shape in the vtkPolyData of the shape under the click.
  - `parent` is a reference to the ShapeAnalyzer Widget which is needed to open QInputDialogs.

4.1.2 Class attributes

- `shared_ptr<Shape>` **`shape_`** Reference to the clicked shape

- `ShapeAnalyzerInterface*` **`shapeAnalyzer_`** Reference to the ShapeAnalyzerInterface, see **??**

4.1.3 Example: VoronoiCellsContextMenuItem

```
template<class T = Metric>
class VoronoiCellsContextMenuItem : public CustomContextMenuItem {
public:
    VoronoiCellsContextMenuItem<T>(
         shared_ptr<Shape> shape,
         ShapeAnalyzerInterface* shapeAnalyzer)
    : CustomContextMenuItem(shape, shapeAnalyzer) {}

    virtual void onClick(vtkIdType pointId, vtkIdType faceId, QWidget* parent) {
        bool ok;
        vtkIdType source = QInputDialog::getInt(
                         parent,
                         "Source vertex",
                         "Choose ID of source vertex.",
                         0, 0,
                         shape_->getPolyData()->GetNumberOfPoints()-1,
                         1, &ok);
        if (!ok) {
            return;
        }
        vtkIdType numberOfSegments = QInputDialog::getInt(
                           parent,
                           "Number of segments",
                           "Choose number of segments",
                           0, 0,
                           shape_->getPolyData()->GetNumberOfPoints()-1,
                           1, &ok);
        if(ok) {
            try {
                auto m = make_shared<T>(shape_);
                auto fps = make_shared<FarthestPointSampling>(
                                shape_, m, source, numberOfSegments);
                VoronoiCellSegmentation segmentation(shape_, m, fps);

                // save current segmentation for being able to create
                // new shapes out of the segments
                shared_ptr<Shape::Coloring> coloring
                             = make_shared<Shape::Coloring>();
                coloring->type = Shape::Coloring::Type::PointSegmentation;
                coloring->values = segmentation.getSegments();
                shape_->setColoring(coloring);
            } catch(metric::MetricError& e) {
                QMessageBox::warning(parent, "Exception", e.what());
            }
        }
    }
};
```

Listing 4.1: VoronoiCellsContextMenuItem.h

In `line 2` we of course inherit from CustomContextMenuItem.

The constructor (`line 4 to 7`) is mandatory like this (not really, for more information look up 4.3). Every normal CustomContextMenuItem should be constructed with those two parameters and should call the super constructor.

The `onClick` function (`line 9 to 48`) is the important one. There are two QInputDialogs that get the source point and the number of segments. There are 4 different types of QInputDialogs (see the static public member functions for details) that restrict the type of inputs you can get for your functions via context menu items. It is important to use the `parent` input as the parent widget here. Then you can call any code you want using the Shape and the ShapeAnalyzerInterface as well as the results from the dialogs as inputs. Notice that all the objects we constructed here (except the Coloring) will be destroyed afterwards. If you want to reuse a computationally intense object later in an algorithm, use a CustomTab instead. The Metric object might throw MetricError which is catched here. All error messages should be created on this level (see 5.2).

This class is special because it is a template. The template parameter is the Metric that should be used for calculating the Voronoi cells. If you have a template menu item you have to register every possible parameter individually for them to appear in the GUI (4.3).

4.2 CustomTab

- **Have to implement** CustomTab
- **Found in** custom/tabs
- **Use when** your algorithm is too complex for a menu item or when you might want to store information
- **Naming convention** <MeaningfulName>Tab

CustomTabs can be loaded into the menu on the right side of the ShapeAnalyzer. Every CustomTab is its own QWidget and can be freely designed (but maybe think about the space restrictions in the menu) in the QtCreator (or by hand if you like that).

The actual use of the abstract functions is to keep your GUI updated about which objects (mainly shapes) exist. This is not done automatically, but you can find a pattern in the example (4.2.4). Of course you only need to do that if you have a list of shapes or something similar in your tab.

4.2.1 Abstract Functions

- `onShapeAdd(Shape* shape)` Is triggered when a new shape was added to the ShapeAnalyzer.
  - `shape` Reference to Shape that was added.

- **onShapeEdit(Shape* shape)** Is triggered when an attribute of a shape was changed (for example the name).
  - `shape` Reference to Shape that was edited.
- **onShapeDelete(Shape* shape)** Is triggered when a shape was deleted.
  - `shape` Reference to Shape that was deleted.
- **onClear()** Is triggered when the GUI is cleared, this means all shapes and correspondences will be deleted.

4.2.2 Class attributes

- `const HashMap<vtkActor*, shared_ptr<Shape»&` **shapes_** Read-only reference to all shapes
- `const HashMap<shared_ptr<PointCorrespondence>, bool>&` **pointCorrespondences_-** Read-only reference to all point correspondences
- `const HashMap<shared_ptr<FaceCorrespondence>, bool>&` **faceCorrespondences_-** Read-only reference to all face correspondences
- `ShapeAnalyzerInterface*` **shapeAnalyzer_** Reference to the ShapeAnalyzerInterface, see **??**

4.2.3 Qt User Interface

You probably want to create the layout for your tab with the QtCreator. Put your .ui file in the custom/tabs folder and name it correspondingly to your tab: <MeaningfulName>TabWidget.ui. During the creation make sure that you named the Widget accordinglly, too. If you didn't do that or you are not sure, the beginning of the .ui should look like this:

```
...
<class>IdentityMatchingTabWidget</class>
<widget class="QWidget" name="IdentityMatchingTabWidget">
...
```

Listing 4.2: IdentityMatchingTabWidget.ui

Then you can inherit from UI::<YourUIClassName> and your widget will be layouted appropriately. You have to include ui_<YourUIClassName>.h for the class to be found. This file will be automatically created so don't worry if you can't find it before compiling.

4.2.4 Example: IdentityMatchingTab

```
#include "ui_IdentityMatchingTabWidget.h"

class IdentityMatchingTab :
                              public QWidget,
                              private Ui::IdentityMatchingTabWidget,
                              public CustomTab
```

```
 7 {
 8     Q_OBJECT
 9
10 public:
11     IdentityMatchingTab(
12       const HashMap<vtkActor*, shared_ptr<Shape>>& shapes,
13       const HashMap<shared_ptr<PointCorrespondence>, bool>& pointCorrespondences,
14       const HashMap<shared_ptr<FaceCorrespondence>, bool>& faceCorrespondences,
15       ShapeAnalyzerInterface* shapeAnalyzer);
16
17     virtual ~IdentityMatchingTab();
18
19     virtual void onShapeDelete(Shape* shape);
20     virtual void onShapeAdd(Shape* shape);
21     virtual void onShapeEdit(Shape* shape);
22     virtual void onClear();
23 private slots:
24     virtual void slotMatch();
25     virtual void slotToggleMode();
26 };
```

Listing 4.3: IdentityMatchingTab.h

Every CustomTab implementation has to inherit from `QWidget` and `CustomTab`. Additionally, if you created a .ui file for the layout, it has to inherit from `Ui::<YourUIClassName>`. See 4.2.3 for more details about that.

Put the `Q_OBJECT` keyword as in `line 8` if your widget is interactive i.e. you want to send signals for example from buttons.

Different from the context menu items, the constructor has to be implemented to setup the widget as well as all abstract functions. For those who have no experience with Qt take a look at the .cpp files for some impressions especially of the signal and slot system.

```
custom::tabs::IdentityMatchingTab::IdentityMatchingTab(...)
        :   QWidget(dynamic_cast<QWidget*>(shapeAnalyzer), 0),
            CustomTab(...)
{
    this->setupUi(this);

    QStringList labels  = getShapeIdentifierList();

    if(shapes_.size() < 2) {
        buttonMatch->setEnabled(false);
    }

    comboBoxShape1->insertItems(0, labels);
    comboBoxShape2->insertItems(0, labels);

    connect(this->buttonMatch,    SIGNAL(released()),
                   this,    SLOT(slotMatch()));
    connect(this->buttonGroupMatch, SIGNAL(buttonClicked(int)),
                    this,    SLOT(slotToggleMode()));
}
```

Listing 4.4: IdentityMatchingTab.cpp, Constructor

In the constructor you have to set up the widget and connect all signals and slots (Signals & Slots) as well as other variable output, here two combo boxes that contain lists of all shapes.

In several GUI elements only a string (a QString actually) is stored and then we need to get the right shape using only this string when interaction with this element takes place. The identifier string looks like this: <shapeId>:<shapeName>. Because this is tedious work, the CustomTab class has functions to deal with that:

- **`QString getShapeIdentifier(Shape*)`** returns the identifier in the right form
- **`vtkIdType getIdFromIdentifier(QString)`** returns the id from a valid identifier
- **`shared_ptr<Shape> getShapeFromId(vtkIdType)`** returns the shape from an id
- **`QStringList getShapeIdentifierList()`** returns a QStringList with all shape identifiers
- **`shared_ptr<Shape> getShapeFromIdentifier(QString)`** returns shape from an identfier

In the example we get all shape identifiers (`line 7`) and fill both combo boxes with them (`line 13 + 14`). The combo boxes are not updated automatically though, that's what done in the abstract functions.

But why don't we use just the names for the shape identifier? Because it is possible that two shapes have the same name and then they can't be distinguished. The id is unique but it's not userfriendly.

```
void custom::tabs::IdentityMatchingTab::onShapeAdd(Shape* shape) {
    QString label = getShapeIdentifier(shape);
    comboBoxShape1->insertItem(0, label);
    comboBoxShape2->insertItem(0, label);

    if(shapes_.size() >= 2) {
        this->buttonMatch->setEnabled(true);
    }
}
```

Listing 4.5: IdentityMatchingTab.cpp, onShapeAdd

This function is called everytime a shape is added to the main application and this tab was added to the menu before. It creates an identifier for the new shape and adds it both combo boxes. Additionally, the match button is activated if more than two shapes exist. The source code of the other abstract functions is not listed here, but you can basically just go and copy it if you update shape lists.

4.3 Registering Customs

Besides from implementing a concrete menu item or tab, the custom has to be registered in order to be shown in the GUI. This can be done in the class `CustomClassesRegistry` in the folder `custom`. There is a standard way to do it explained below but if it does not suffice for your purpose take a look at the Factory section (3.4).

4.3.1 Register a CustomContextMenuItem

In `CustomClassesRegistry.h` add your menu item to the `registerContextMenuItems()` function. Your registration must have the form

```
CustomContextMenuItemFactory::getInstance()->Register< <NameOfYourClass> >
(<UniqueStringIdentifier>, <LabelPath>)
```

where `<NameOfYourClass>` is the specific class you want to create, for example ColorMetricContextMenuItem<GeodesicMetric>. `<UniqueStringIdentifier>` is a string that should be meaningful and not identical to any other string in the registry list. `<LabelPath>` is a string that describes how the menu item should be labeled as well as its path, for example `"Coloring»Metric»Geodesic"` will produce a 3-layered menu (see Figure 3.3 for an example). If several menu items have the same path (up to some layer), these will of course be merged.

4.3.2 Register a CustomTab

In `CustomClassesRegistry.h` add your menu item to the `registerTabs()` function. Your registration must have the form

```
CustomTabFactory::getInstance()->Register< <NameOfYourClass> >
(<UniqueStringIdentifier>, <LabelPath>)
```

where `<NameOfYourClass>` is the specific class you want to create, for example CorrespondenceColoringTab. `<UniqueStringIdentifier>` is a string that should be meaningful and not identical to any other string in the registry list. `<LabelPath>` is a string that describes the label of the tab and whether it is a correspondence or shape tab, for example `"Correspondences»Correspondence Coloring"` or `"Shapes»Mesh Checker"`. Different from menu items not arbitrary many layers are possible and only Shapes and Correspondences are possible groups. The groups just loosely describe whether the tab deals more with shapes or more with correspondences.

CHAPTER 5

Exceptions

There are two types of exceptions you might want to throw: Exceptions (5.1) and Errors (5.2). The first one is kind of optional, it includes all normal exceptions C++ provides, the second one is for non-fatal problems that might occur and that can actually be handled. In both cases do not use the `throws` keyword in the function signature.

When you are using VTK classes you can use the ErrorObserver (5.3) to handle VTK specific exceptions.

5.1 Exceptions

- **Throw when** you have special cases that are due to previous faulty programming or unexpected errors
- **Documentation** not necessary
- **Found in** C++ std library

Any exception will be caught by the main program and shown within an error message. The program will then terminate. In order to make debugging easier a short description of the problem and the function name is useful.

```
if(s >= points_->size()) {
        throw invalid_argument("Source point (" + to_string(s) + ")
        larger than number of points (" + to_string(points_->size()) + ")
        in " + __PRETTY_FUNCTION__);
}
```

5.2 Errors

- **Throw when** you notice faulty data or the like that can not be processed, but it is not necessary to terminate the whole program
- **Documentation** use the `@throws` command and define the Error class
- **Found in** util or the namespaces folder

Error is a ShapeAnalyzer specific class that inherits from exception but does not add any functionality. Every namespace has its own Error class that is named `<namespace>Error`. This allows more specific error handling by catching the different classes.

If your exception is not due to faulty programming logic or you think it can be solved by a higher level class, use this classes instead of a std exception. This can be for example detecting a mesh with borders but your algorithm does not support borders. Most of the time this means you terminate your algorithm without a result and let the user know about the problem with an pop-up. Again a short description of the problem and the function name is useful. You can use `QMessageBox::warning` for the pop-up but you must give the QMessageBox the ShapeAnalyzer as the parent widget, so catch the Error in your Tab or ContextMenuItem (see **??**) and create the QMessageBox there.

It is possible to give null as the parent widget but you must not do that. It might cause unwanted behavior like the QMessageBox not appearing in the foreground.

If your function is throwing an Error use the `@throws` or `@exception` keyword in the Doxygen documentation. Give the class of the error and a short description of what might cause this error.

```
if(ierr != 0) {
    throw LaplaceBeltramiError(string(
    "Error occured in computation of Laplace-Beltrami eigenfunction
    (Probably due to corrupted shape) in ")
    .append(__PRETTY_FUNCTION__));
}
```

```
try {
    PetscFEMLaplaceBeltramiOperator laplacian(shape_, 100);
    vtkSmartPointer<vtkDoubleArray> eigenfunction
        = laplacian.getEigenfunction(i);
} catch(LaplaceBeltramiError& e) {
    QMessageBox::warning(parent, "Exception", e.what());
}
```

Some namespaces do not have their own Error class yet. If you need it, just create it. You can use `MetricError` as a template.

5.3 ErrorObserver

- **Use when** you want to handle VTK specific errors
- **Found in** util

VTK classes in general do not throw exceptions instead they write error and warning messages to observers. If you want to handle those problems use the ErrorObserver class.

```
// vtk object that we want to observe
vtkSmartPointer<vtkTriangleFilter> triangleFilter
  = vtkSmartPointer<vtkTriangleFilter>::New();
// new ErrorObserver object
```

```
vtkSmartPointer<ErrorObserver> triangleObserver
  = vtkSmartPointer<ErrorObserver>::New();

// list the observer to the triangle filter
// if you are not interested in warnings leave the second line out
triangleFilter->AddObserver(vtkCommand::ErrorEvent, triangleObserver);
triangleFilter->AddObserver(vtkCommand::WarningEvent, triangleObserver);

// do the things you want to do with your vtk object
triangleFilter->SetInputConnection(input);
triangleFilter->Update();

// see if messages were written into the observer and handle that
if (triangleObserver->GetError()) {
        showErrorMessage("The file cound not be opended",
                       triangleObserver->GetErrorMessage());
        return;
}
if (triangleObserver->GetWarning()) {
        showErrorMessage("There was a warning reading the file",
                        triangleObserver->GetWarningMessage());
}
```

CHAPTER 6

Coding Style

Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.

Of course we don't know where you live but other people might actually want to use your code. The Google C++ Style Guide is a good guideline.

6.1 C++11

C++11 introduced a lot of useful new features. If you are not familiar with C++11 take a look here and try to make use of them. Especially the new pointer types.

6.2 Documentation

Documentation should be done in Doxygen. You have to use the C++ comment blocks starting with ///, JavaDoc and QtDoc are not activated.

```
///
/// \brief HashMap is a wrapper class for unordered_map with some extra features,
/// e.g. a proper contains function and a random subset of the map can be
/// generated.
/// \details A HashMap<KEY, VALUE> maps objects from type KEY to objects from
/// type value. It manages a unordered_map<KEY, VALUE> object and has some
/// special features. The first one is that the [] operator does not add keys to
/// the hash map when you try to access them although they did not exist.
/// Additionally you can get random subsets of the elements, keys or values.
///
/// @tparam KEY type used as the key in the hash map
/// @tparam VALUE type of the mapped values
/// \author Emanuel Laude and Zorah L\"ahner
///
```

You can find a list of all possible commands here.

6.2.1 Installing and Using Doxygen

After installing Doxygen, you can update the documentation with the Doxyfile found in the projects root folder. If you have problems installing Doxygen with the binaries, try to compile it yourself.

Use

```
doxygen Doxyfile
```

in the root folder.

Do not add all the files produced by Doxygen to the repository.