UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

AGAS
Arbeitsgruppe Aktives Sehen

Redesign und Refactoring einer GUI für das Projekt Primus

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von

Marcel Haeselich

Betreuer: Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik

Koblenz, im Juli 2008

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien der Arbeitsgruppe für Studien- und Diplomarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	ja ☐	nein ☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	ja ☐	nein ☐

Koblenz, den .... Unterschrift

Inhaltsverzeichnis

1 Einleitung **9**

1.1 Überblick 9

1.2 Primus 10

1.3 Aufbau der Arbeit 11

2 Stand der Technik und eigener Ansatz **13**

2.1 Stand der Wissenschaft 13

2.2 Stand der Technik 14

2.3 Eigener Ansatz 15

3 Grundlagen **17**

3.1 Qt 17

3.1.1 Trolltech 17

3.1.2 Warum Qt gewählt wurde 17

3.1.3 qmake und moc 18

3.1.4 Vorgefertigte Dialoge 18

3.1.5 Signal-Slot-Prinzip 19

3.1.6 QEvent 20

3.1.7 QtSql 20
3.2 Das Primus-Framework 21
3.2.1 Messages 21
3.2.2 Worker 22

4 Umsetzung **23**
4.1 Systemarchitektur 23
4.1.1 Ordnerstruktur 23
4.1.2 Datenkapselung 24
4.2 Redesign und Refactoring der GUI 25
4.2.1 Fortschrittdialoge 25
4.2.2 Bildoptionen 26
4.2.3 Settings 28
4.2.4 Region of interest 30
4.2.5 Icons der GUI 30
4.3 Neue Elemente der GUI 31
4.3.1 Tracer für die GUI 31
4.3.2 Objektorientierte Dialoge 32
4.3.3 Darstellung von Objekten anstelle von Pixeln 32
4.3.4 GeometricObjectConverter (goc-Worker) 33
4.3.5 Arbeiten auf den Objekten 35
4.3.6 Dokumentation 35

5 Zusammenfassung **37**

Literaturverzeichnis **40**

Verzeichnis der Bilder **41**

A Installationshinweise **45**

A.1 Systemvorraussetzungen 45

A.2 svn und Pfade 45

A.2.1 svn Revision 45

A.3 Pakete 46

A.4 Umgebungsvariablen 47

A.5 Qt 48

A.6 Settingsdatei 48

A.7 Programmstart 48

B Anhang: Dokumentation **49**

Kapitel 1

Einleitung

1.1 Überblick

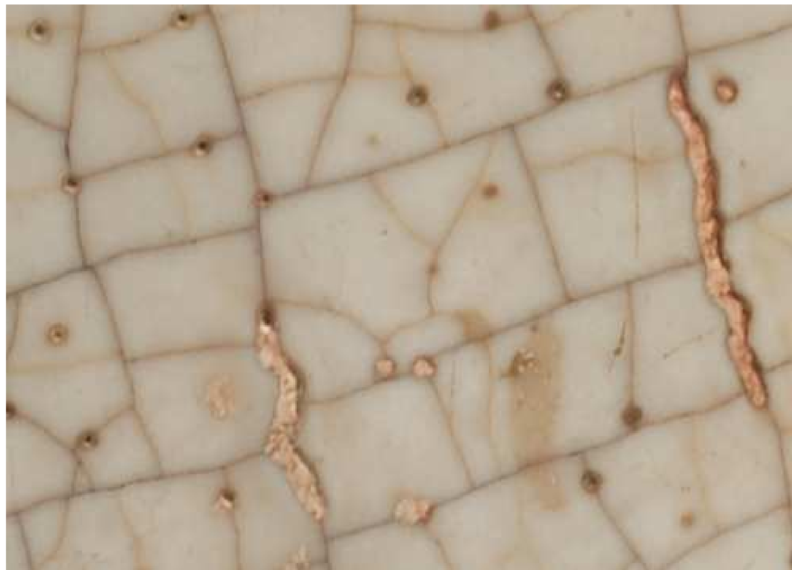

Bild 1.1: Fotografie eines Rissmusters

Bislang gibt es kein Verfahren zur objektivierten Auswertung und Analyse von Rissmu-

stern. Diese enthalten jedoch zahlreiche Informationen über deren Entstehung, die für die Technik oder die Kulturgeschichte von großem Wert sein können. Teilweise existieren bereits Klassifizierungsverfahren, die sich für die computergestützte Auswertung einsetzen lassen. Durch automatische oder semi-automatische Klassifizierung von Rissmusterabbildungen könnte beispielsweise die Echtheitsprüfung von antiken Artefakten oder die Materialforschung vereinfacht werden. Bild 1.1 zeigt eine Fotografie eines typischen Rissmusters.

1.2 Primus

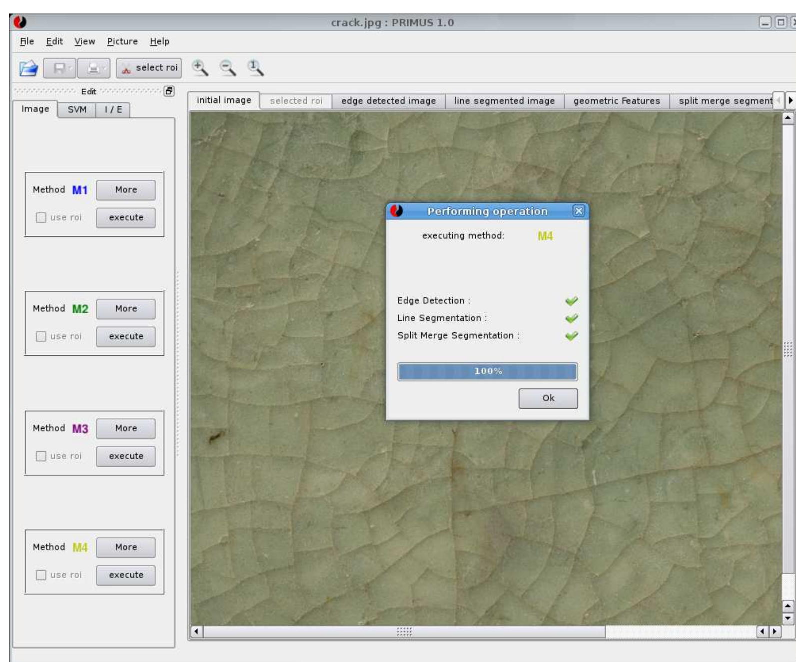

Bild 1.2: Screenshot von Primus

In Zusammenarbeit mit Prof. Dr. Gerhard Eggert von der Staatlichen Akademie der Bildenden Künste Stuttgart entstand so 2007 das **P**rojektpraktikum **RI**ss**MUS**teranalyse. Ziel des Projekts war die Entwicklung eines Bildverarbeitungsverfahrens, das die automatische Klassifikation von Rissmuster-Aufnahmen ermöglicht. Daran angebunden sollte ein

Datenbanksystem die Bilder samt ihrer Analyseergebnisse verwalten und darstellen können. In einer einfach zu bedienenden grafischen Benutzeroberfläche, Bild 1.2, werden 4 verschiedene Methoden angeboten, die mit jeweils unterschiedlichen Bildverarbeitungsverfahren eine robuste Klassifikation der Rissmuster und den anschließenden Transfer in die Datenbank ermöglichen.

1.3 Aufbau der Arbeit

Der Aufbau der Arbeit ist wie folgt: In Kapitel 2 werden die aktuelle Situation des Projekts Primus und die gewünschten Veränderungen dargestellt. Im darauf folgenden Kapitel 3 werden die grundlegenden Strukturen des Projekts mit besonderem Hinblick auf die verwendete Programmiersprache Qt dargestellt. Dabei werden bereits erste Besonderheiten der Sprache und ihrer Klassen erklärt, die im Kapitel 4 dann wieder aufgegriffen werden um die Veränderungen und Neuerungen der grafischen Benutzeroberfläche zu verdeutlichen. Im letzten Kapitel 5 werden die Ergebnisse des Redesigns und Refactorings reflektiert.

Kapitel 2

Stand der Technik und eigener Ansatz

2.1 Stand der Wissenschaft

“Im Gegensatz zu Krakelee in Malschichten und Firnis wurden Rissmuster in Gläsern und Glasuren bisher als Informationsquelle für die Restaurierung vernachlässigt. Ein genauerer Blick zeigt Unterschiede auf, die überraschende Hinweise geben können, z.B. auf das Abschrecken von Beigaben der Leichenverbrennung, gegen thermisches Vorspannen von Gläsern bei den Römern oder für ein besseres Verständnis von Rissen in authentischen Glasuren. Eine Terminologie zur Beschreibung muss einerseits Unterschiede in der Form der einzelnen Risse, andererseits Besonderheiten des Rissmusters erfassen. Dies könnte ein besseres Verständnis ihrer fraktalen Geometrie und der in ihnen verborgenen Informationen ermöglichen.“ [Egg06]

Eine Software zur Rissmusteranalyse ermöglicht dem Kunsthistoriker eine neue Form der Verarbeitung von digitalen Bilddaten. Zum einen ist es möglich, die Bilder durch unterschiedliche Vorverarbeitungsmethoden besser auf ihre wesentlichen Merkmale zu reduzieren. Dabei sind Farbinformationen, Schatten und andere Fragmente aus dem Bild herauszurechnen, um das Rissmuster bestmöglich zu isolieren. Bild 2.1, Bild 2.2, Bild 2.3 und Bild 2.4 zeigen vier typische Rissmuster. Anhand der Bildbeipiele lässt sich erkennen, dass jedes Rissmuster über charakteristische Merkmale verfügt, die herauskristallisiert werden

sollen. Darüber hinaus ermöglicht die Software auch eine verbesserte Repräsentation der gewonnenen Daten. Zum anderen ergibt sich aus einer Applikation mit integrierter Mehrfachverarbeitung die Option, hunderte Bilder nacheinander zu verarbeiten. Bereits klassifizierte Rissmuster können auf diese Weise ohne Zeitaufwand für den Anwender in die Datenbank übertragen werden und trainieren somit die Software.

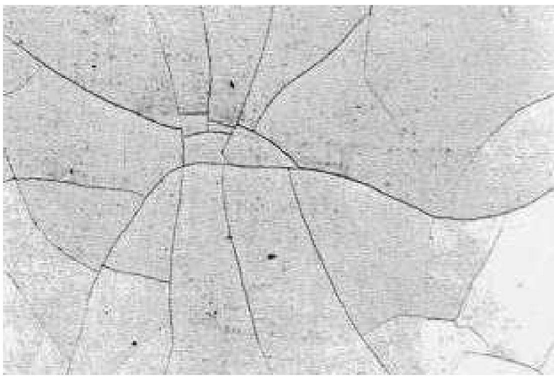

Bild 2.1: Typ Spiderweb

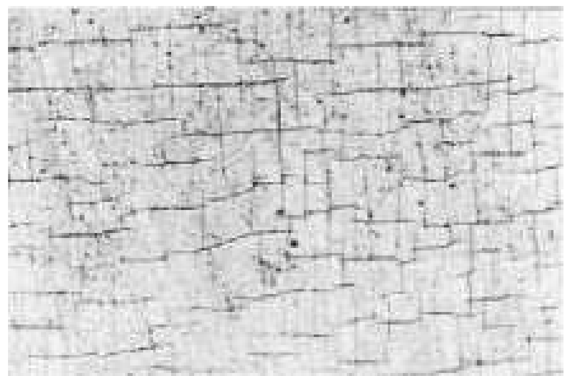

Bild 2.3: Typ Rectangle

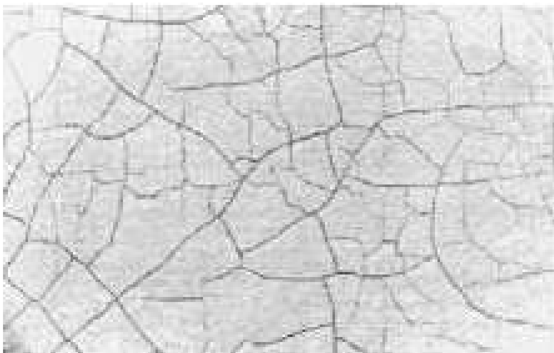

Bild 2.2: Typ Circular

Bild 2.4: Typ Unidirectional

2.2 Stand der Technik

Die grafische Benutzeroberflächen des Projekts Primus ist vollständig während des Projektpraktikums entstanden. Dabei wurde das Grundgerüst bereits zu Beginn ins Frame-

work integriert und von da an stetig ausgebaut. Die GUI(**G**raphical **U**ser **I**nterface) wuchs während des gesamten Projekts stetig weiter, denn je umfangreicher das Framework wurde, desto umfangreicher wurde die GUI. Als zentrales Instrument der Benutzerinteraktion bildet sie automatisch die Schnittstelle zwischen den einzelnen Teilen des Frameworks. Darunter fallen sämtliche Vorverarbeitungsmethoden der Bilddaten, das Auswählen einer *region of interest*, das Einstellen aller Konfigurationsoptionen, die Anbindung an die SVM, die Interaktion mit der Datenbank und zu guter Letzt auch der Multimodus. Da dies alles während der Arbeitszeit von sieben Personen angebunden werden musste, wurde leider oft die schnellste oder einfachste Lösung eines Problems gewählt. Softwareergonomische Anforderungen an die Benutzeroberfläche wurden vernachlässigt und statische Bereiche erweitert, die sich dynamisch viel eleganter und vor allem intelligenter realisieren lassen. Die Datenstruktur des GUI beschränkte sich auf 3 C++-Dateien und ihren zugehörigen Headern mit insgesamt ca. 7.000 Zeilen.

2.3 Eigener Ansatz

Der eigene Ansatz dieser Arbeit umfasst zwei Schwerpunkte: Zum einen das Redesign und zum anderen die Erweiterung der GUI um neue Komponenten wie einen separaten Tracer und Qt-Objekte anstelle von Bildern in Pixeldarstellung. Die GUI soll sich durch neue grafische Elemente und eine verbesserte Darstellung auszeichnen. Für die optische Verschönerung wurde eine kostenlose, aber umfangreiche Icon-Bibliothek verwendet, die in Kapitel 4.2.5 beschrieben wird. Außerdem wird sich die GUI mehr an softwareergonomischen Standards orientieren, um dem Benutzer eine verbesserte Interaktion zu ermöglichen. Es existiert eine Vielzahl an Richtlinien für eine gute grafische Benutzeroberfläche. Die Hersteller von Betriebssystemen besitzen jeweils eigene Veröffentlichungen von GUI Styleguides[1]. Generell sollte eine GUI möglichst zielgruppenorientiert aufgebaut sein. Bei der Erstellung des Redesigns werden besonders softwareergonomische Merkmale berücksichtigt[2].

[1] http://www.procontext.com/de/richtlinien/styleguides.html

[2] eine Übersicht mit Veröffnetlichungen findet sich auf http://www.gui-design.de/swlist.htm

“Die Anforderungen an eine grafische Benutzungsschnittstelle im Rahmen der Mensch-Computer-Kommunikation sind in der europäischen Norm EN ISO 9241-110 ff. geregelt. Dabei muss die Schnittstelle folgende Merkmale aufweisen:

- *Aufgabenangemessenheit*
- *Selbstbeschreibungsfähigkeit*
- *Steuerbarkeit*
- *Erwartungskonformität*
- *Fehlertoleranz*
- *Individualisierbarkeit*
- *Lernförderlichkeit*

Ferner ist in der Norm EN ISO 9241 die Umsetzung von Benutzungsschnittstellen für Web-Applikationen und deren Evaluation im Rahmen der Benutzbarkeit definiert.“ [3]

Die GUI von Primus wird daher bestrebt sein, den Anforderungsmerkmalen bestmöglich gerecht zu werden. Besondere Aufmerksamkeit wird auf Erwartungskonformität , Selbstbeschreibungsfähigkeit, Steuerbarkeit und Fehlertoleranz gelegt werden. Die Verwendung von Qt garantiert zudem, dass die Applikation unter jedem Betriebssystem automatisch an das verwendete Erscheinungsbild angepasst wird.

[3]Quelle: http://de.wikipedia.org/wiki/Grafische_Benutzeroberfl%C3%A4che

Kapitel 3

Grundlagen

3.1 Qt

3.1.1 Trolltech

Qt wurde von der Firma Trolltech zur GUI-Entwicklung in C++ hergestellt. 1995 erschien das erste öffentliche Toolkit. Trolltech vertreibt Qt unter zwei verschiedenen Lizenzen, einer GPL-Lizenz für Open-Source-Software und einer kostenpflichtige Lizenz speziell für kommerzielle Applikationen. Die von Trolltech entwickelte Software kommt bereits bei vielen namhaften Anwendungen zum Einsatz: Die freie Desktopumgebung KDE, der Opera-Webbrowser, die VoIP-Software Skype und der virtuelle Globus Google Earth. [1]

3.1.2 Warum Qt gewählt wurde

Qt ermöglicht eine perfekte Integration der grafischen Benutzeroberfläche in bereits bestehenden C++-Code. Obwohl es unter Open-Source Lizenz für nicht kommerzielle Anwender verfügbar ist, steht trotzdem die komplette Funktionalität zur Verfügung. Zudem glänzt Qt durch den Qt-Assisstant, durch den die ohnehin sehr gute Dokumentation aller

[1] http://trolltech.com/

Qt-Klassen um eine komfortable GUI mit schneller Suchfunktion dargestellt wird. Im Internet und innerhalb des Qt-Assisstant existiert außerdem eine stetig wachsende Zahl an Tutorials zu den unterschiedlichsten Anforderungen an grafischen Benutzeroberflächen.

3.1.3 qmake und moc

Jede Entwicklungsumgebung hat eigene Tools: Compiler, Linker, make-Programm. Zusätzlich haben die Tools auch noch andere Namen, andere Parameter und unterschiedliche Optionen. Qt funktioniert dank qmake mit einer Vielzahl von Plattformen und Compilern. qmake sucht im aktuellen Verzeichnis nach Dateien und generiert dann aus der Projektdatei ein passendes Makefile und fügt diesem Regeln für die Kompilierung der .uic-Dateien hinzu. Dabei wird der meta object compiler(moc) von Qt verwendet. Dieser untersucht alle Quellcodedateien des Projekts und erstellt daraus Meta-Informationen über die Klassen und andere Programmteile. Aus den gesammelten Informationen erstellt der meta object compiler dann den C++-Code, in dem die Funktionen implementiert werden und die ohne weitere Bibliotheken in C++ nicht möglich wären. Auf diese Weise realisiert der meta object compiler automatisch die Introspektion aller mit qmake kompilierten Programme. Introspektion bedeutet im Sinne der Informatik, dass ein Programm Erkenntnisse über seine eigenen Struktur gewinnen kann und ermöglicht bei objektorientierter Programmierung z.B., dass zur Laufzeit Informationen über Klassen oder deren Instanzen abgefragt werden können [Bla04] .

3.1.4 Vorgefertigte Dialoge

Qt verfügt über eine Vielzahl von vorgefertigten Dialogen zur Erstellung der häufigsten Elemente einer GUI. Es existieren viele Klassen zur Erstellung von Informationsdialogen. Zum Beispiel die Klasse QMessageBox die mit ihren 40 abgeleiteten Klassen nahezu jeden erdenklichen Dialog zum Darstellen oder Abfragen kleinerer Eingaben ermöglicht und dabei HTML und Bilder enthalten kann.

3.1.5 Signal-Slot-Prinzip

Eines der elementarsten Prinzipien der Qt-Programmierung ist das Signal-Slot-Prinzip. Es ermöglicht das Verbinden eines oder mehrerer Signale mit einem oder ebenfalls mehreren Slots. Das Besondere daran ist, dass weder der Sender des Signals den Empfänger(und dessen Slot) kennen muss, noch umgekehrt. Diese Verbindungen werden über die connect-Methode hergestellt und können auch erst zur Laufzeit des Programms erstellt oder wieder getrennt werden. Für das Trennen gibt es die disconnect-Methode. Der Code zum Verbinden eines Signals mit einem Slot sieht dabei folgendermassen aus:

```
connect( sender,   SIGNAL( signal type ),
         receiver, SLOT(   slot type   ) );
```

Um die Verbindung aufzuheben, besteht entweder die Möglichkeit, genau diese Verbindung mit dem disconnect-Aufruf zu beenden(a), alle Verbindungen des Senders zu einem bestimmten Empfänger(b) oder generell alle Verbindungen eines Senders(c):

```
(a) disconnect( sender,   SIGNAL( signal type ),
                receiver, SLOT(   slot type   ) );

(b) disconnect( sender, SIGNAL( signal type ),
                0,      0 );

(c) disconnect( sender );
```

Die Makros SIGNAL und SLOT nehmen als Argument die Deklaration einer Methode an und geben einen speziellen String vom Typ const char* zurück. Dieser String wird durch QObject als Identifikator dieser Methode verwendet und somit wird die Art des Kommunikationswegs innerhalb des Programms definiert. Signale und Slots können Parameter von beliebigem Typ besitzen und ein Slot kann weniger Parameter als eines seiner Signale besitzen. Jedoch muss jeder Parameter des Slots sich auf derselben Stelle und desselben Typs sein wie im Signal. An einen Slot ohne Parameter kann also jedes Signal angeschlossen werden. Der *meta object compiler* sieht alle Slots und baut auf dieser Basis eine Liste

und realisiert die Implementierung des Signals. Die Implementation ruft alle Slots auf, die im gegebenen Moment an das Signal angeschlossen sind. Das Entfernen des Objektes entfernt automatisch alle Verbindungen, an denen es teilnimmt [SB06] .

3.1.6 QEvent

QWidget ist die Basisklasse für alle Klassen, die auf dem Bildschirm gezeichnet werden können, und besitzt einen Mechanismus zur Bedienung von Ereignissen. Jedes QWidget hat die virtuelle Funktion

```
bool QWidget::event( QEvent *event );
```

Die Funktion gibt genau dann true zurück, wenn der Parameter event erkannt wurde. Informationen über ein Ereignis werden im Parameter vom Typ QEvent* übergeben. Um festzustellen, mit welchem Ereignis man es zu tun hat, wird der Rückgabewert der Methode QEvent::type() betrachtet. Dieser kann dann anschließend auf die entsprechende Klasse gecastet werden. Im Allgemeinen stellt die Standardimplementation von QWidget die Ereignisart selbst fest und ruft die zugehörige virtuelle Methode auf. Die Ereignisse werden in einer Ereignisschleife abgearbeitet. Widgets können mit Hilfe von:

```
QApplication::postEvent( QObject *receiver, QEvent *event );
```

miteinander kommunizieren. Diese Funktion stellt dem Objekt receiver das Ereignis event im nächsten Zyklus der Ereignisschleife bereit. Zum Beispiel fügt die Funktion:

```
QWidget::update()
```

der Ereignisschleife ein an sich adressiertes QPaintEvent hinzu infolgedessen wird das ganze Widget im nächsten Zyklus der Ereignisschleife neu gezeichnet.

3.1.7 QtSql

Qt bietet ein Modul zur Integration einer Datenbank in die grafische Benutzeroberfläche. Durch einbinden des Moduls lassen sich alle gängigen Verbindungen, Anfragen und Ope-

ration auf einer Datenbank schnell und einfach realisieren. Über die Klasse QSqlDatabase lassen sich folgende Datenbankmodelle vorgefertigt einbinden:

Driver Type	Description
QDB2	IBM DB2
QIBASE	Borland InterBase Driver
QMYSQL	MySQL Driver
QOCI	Oracle Call Interface Driver
QODBC	ODBC Driver (includes Microsoft SQL Server)
QPSQL	PostgreSQL Driver
QSQLITE	SQLite version 3 or above
QSQLITE2	SQLite version 2
QTDS	Sybase Adaptive Server

Bild 3.1: QSqlDatabase

3.2 Das Primus-Framework

3.2.1 Messages

Die Nachrichtenstruktur des Primus-Frameworks ist während des Projektpraktikums Rissmusteranalyse entstanden und wurde teilweise aus der Architektur des Projekts Robbie 6 entnommen. Die Messages dienen zur Kommunikation im System der grafischen Benutzeroberfläche und des restlichen Frameworks. Jede Message besitzt eine Zählvariable und wird ausschließlich als Pointer versendet. Jeder Zugriff auf eine Message inkrementiert ihre Zählvariable um Speicherlecks zu vermeiden. Erreicht die Zählvarible einen definierten Wert, so wird die gesamte Message gelöscht. Die Nachrichten besitzen zudem eine einzigartige Id um eine exakte Identifizierung zu ermöglichen. Im Header jeder Message wird ihr Typ deklariert, unabhängig vom Typ der Nachricht gilt jedoch für alle, dass sie

nur get- und keine set-Funktionen besitzen dürfen.

3.2.2 Worker

Die Anbindung unterschiedlicher Systemteile wird fast immer über Worker realisiert. In den Worker befinden sich unter anderem Funktionen, die an vielen Stellen und von unterschiedlichen Programmteilen verwendet werden. In Hinblick auf die GUI eigenen sich Worker besonders gut für die Schnittstellen zum Framework. Wird die grafische Benutzeroberfläche geändert oder komplett ausgetauscht, so bleibt die Schnittstelle bestehen und kann von der neuen GUI wieder verwendet werden. In Kapitel **??** wird die Integration der GUI durch Worker weiter erläutert.

Kapitel 4

Umsetzung

4.1 Systemarchitektur

4.1.1 Ordnerstruktur

Die Ordnerstruktur von Primus ist wie folgt aufgebaut. Es handelt sich jedoch um eine gekürzte Fassung, da das Augenmerk auf die Elemente der GUI gelenkt werden soll:

```
30_prog
       /Architecture
                    /Dispatcher
                    /Message
                    /Module
                    /Singleton
                    /Thread
       /Config
       /GUI
                    /InitialLabel
                    /GraphicsObjects
                                   /GraphicsLine
```

```
                                          /GraphicsPath
                                          /GraphicsPolygon

                    /GraphicsScene
                    /GraphicsView
                    /GuiImages
                    /GuiTracer
                                          /GuiTracerLogFiles
        /Images
        /include
                    /MatrixOperations
        /lib
```

4.1.2 Datenkapselung

Bei der Erstellung des Frameworks und der GUI wurde das Konzept der Datenkapselung eingehalten. Unterschiedliche Programmkomponenten kommunizieren ausschließlich über fest definierte Schnittstellen mit Messages untereinander. Der Quellcode der einzelnen Komponenten liegt in unterschiedlichen Dateien in verschiedenen Ordner. Der jeweilige Ordner beinhaltet zudem mindestens ein eigenes Makefile, sodass ein Austausch einer Komponente nur an zwei Stellen erfolgen muss. Zum einen muss die Komponente selbst verändert werden, zum anderen muss im zentralen Makefile die Komponente gegebenenfalls neu konfiguriert werden. Momentan erfolgt die Anbindung der GUI und des Frameworks an die Datenbank noch über Qt. Dabei ist jedoch auch die Datenbankanbindung bereits gekapselt und der Code ausgelagert. Somit ist es möglich, die Verbindung zur Datenbank beispielsweise auf Java umzustellen, ohne dafür an der GUI oder dem Framework Veränderungen durchführen zu müssen.

4.2 Redesign und Refactoring der GUI

4.2.1 Fortschrittdialoge

In der bisherigen Version von Primus wurde die Bearbeitung der Bilder durch das Klicken des entsprechenden Buttons in der GUI gestartet. Daraufhin öffnete sich ein Fortschrittsdialog vom Typ QDialog der über die Funktion “setWindowModality()“ die Eigenschaft “Qt::ApplicationModal“ zugewiesen bekam. Diese Eigenschaft blockierte sämtliche Ein- und Ausgaben an andere Komponenten der GUI und setzte das ihr zugewiesene Objekt in den Vordergrund der Anwendung. Zusätzlich wurde das “closeEvent“ des Dialogs überschrieben und solange blockiert, bis die Verarbeitungsschritte vollständig abgeschlossen waren. Innerhalb des Dialogs wurden die jeweiligen Schritte illustriert und nach ihrer Fertigstellung, wie in Bild 4.1 zu sehen, mit Häkchen versehen. Darunter befand sich noch ein Fortschrittsbalken, in dem in Prozenten die Verarbeitung dargestellt wurde. In der Praxis hatte sich dieser Dialog mit seinen Einschränkungen für den Anwender als unnötiges Feature herausgestellt. War ein kleines Bild verarbeitet oder eine schnelle Methode gewählt worden, so war der Dialog direkt bei 100% und somit im Grunde genommen überflüssig. Bei einem großen Bild oder einer zeitaufwändigen Methode hingegen war der Dialog viel zu lange geöffnet. Im Hintergrund wurden beispielsweise die Tabs 2 bis 5 geöffnet und wären schon vollständig zu Betrachten gewesen, doch die Eigenschaft “Qt::ApplicationModal“ verhinderte alle Benutzerinteraktion bis auch der letzte Tab, in diesem Fall Tab 6, fertiggestellt war. Das Redesign von Primus verzichtet daher völlig auf diesen Fortschrittsdialog. Beim Ausführen einer Methode oder sonstigen Interaktion mit der GUI entstehen keine blockierenden Dialoge mehr. Die Verarbeitung wird von der GUI nun so gehandhabt, dass Ergebnisse nach ihrer Reihenfolge gezeichnet und anschließend direkt in einem neuen Tab dargestellt werden. Somit verhält sich die GUI so, dass bei schnellen Verarbeitungschritten direkt alle Ergebnisse vorhanden sind und bei langsamen nach und nach die Ergebnisse eingeblendet werden und dadurch nur noch minimale Wartezeiten entstehen. Derzeit ist keinerlei Rückmeldung an den Anwender erforderlich, da erste Ergebnisse quasi augenblicklich verfügbar sind. Die Schritte der Methoden sind zu Beginn ihrer jeweiligen Verarbeitung sehr schnell und stellen für moderne Computer keinen wirklichen Rechenaufwand dar. Sollte jedoch eine weitere Methode integriert werden,

deren erster Verarbeitungsschritt viel Zeit beansprucht, so würde es sich empfehlen, eine kleine Rückmeldung bei Verarbeitungsstart an den Benutzer auszugeben. Zum Beispiel würde sich dafür dann QDialog sehr gut eignen, der sich durch das Drücken eines Ok- oder Schließen-Buttons sofort wieder ausblenden lässt.

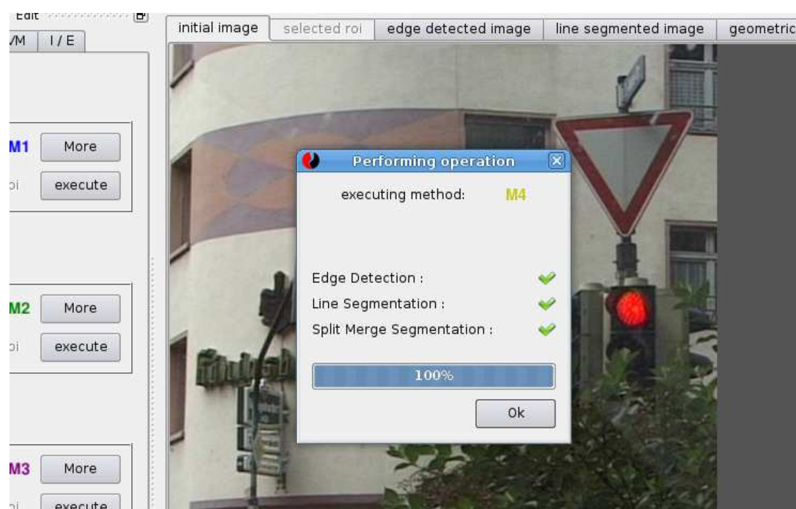

Bild 4.1: Fortschrittdialog

4.2.2 Bildoptionen

Die Darstellung der unterschiedlichen Verarbeitungsschritte der Bilder erfolgt nach wie vor in nebeneinander arrangierten Tabs im zentralen Bereich der GUI. Bislang waren die Bildoptionen jedoch global, d.h. die Option zum Vergrößern wirkte sich auf alle momentan geladenen Bilder aus. Bilder, die durch die Verarbeitung erst nach dem Vergrößern hinzukommen, wurden direkt vergrößert dargestellt. Erst beim Öffnen eines neuen Bildes wurde die Orginalgröße wiederhergestellt. Im Redesign hat sich dies aus Gründen der Performance und Wartbarkeit grundlegend geändert. Die Bildoptionen Drucken, Vergrößern, Verkleinern, Originalgröße wiederherstellen und Speichern befinden sich nun bei jedem dargestellten Bild und nicht mehr oberhalb der Tabs. Dadurch existieren zwar mehr Buttons, diese sind jedoch wesentlich effizienter und verhalten sich mehr nach den Erwar-

tungen des Anwenders. Zudem ist es durch die neue Anbindung der Optionen möglich, ein und dasselbe Bild in verschiedenen Verarbeitungsstufen mit unterschiedlichen Zoomfaktoren zu betrachten.

4.2.3 Settings

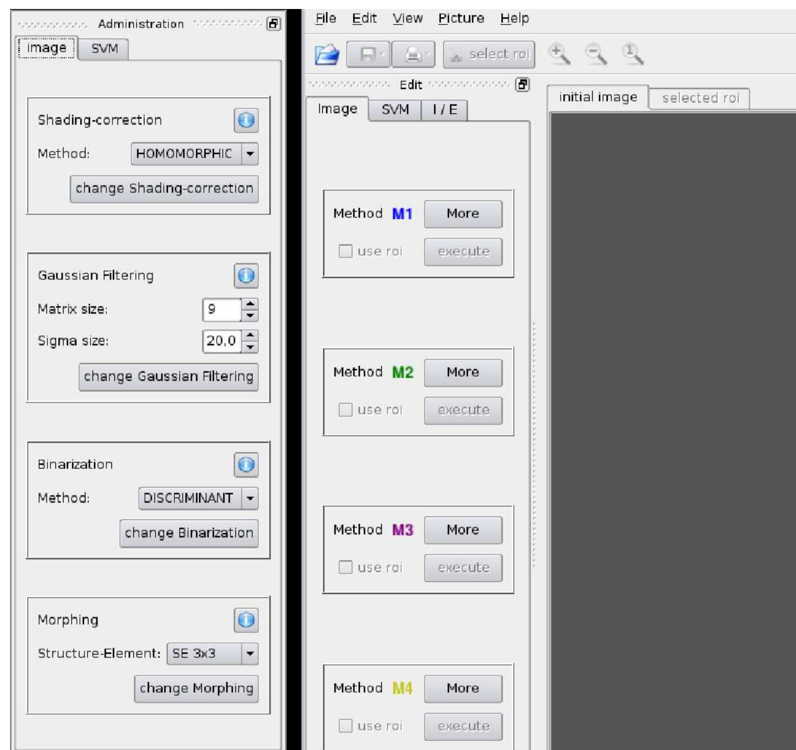

Bild 4.2: Settingsoptionen

Bild 4.2 zeigt die alte Version der Settings aus dem Projektpraktikum. Die möglichen Einstellungen waren in 2 Reitern organisiert, die sich in einem extra Fenster befanden. Bei dem Fenster handelte es sich um einen abgeleiteten QDockwidget, der auf der rechten Seite der GUI per "Drag and Drop" andocken und ausklinken konnte. Das Fenster war bei Programmstart unsichtbar, da eine Veränderung der Settings anfangs nur durch die Softwareentwickler möglich sein sollte. Man konnte über das Menü unter "View -> Administrator Options" den Widget einblenden und anschließend die Optionen manipulieren. Die Optionen waren dabei nicht nach ihrer Zugehörigkeit zu den Schritten geordnet, sondern in Bildvorverarbeitung und SVM unterteilt. Unterhalb eines jeden Reiters war ein *Change*-Button, der die Einstellungen bei Aktivierung in die Settingsdatei schrieb. In der neuen Version der GUI hat sich, wie in Bild 4.3 zu sehen, das Ändern der Option völlig

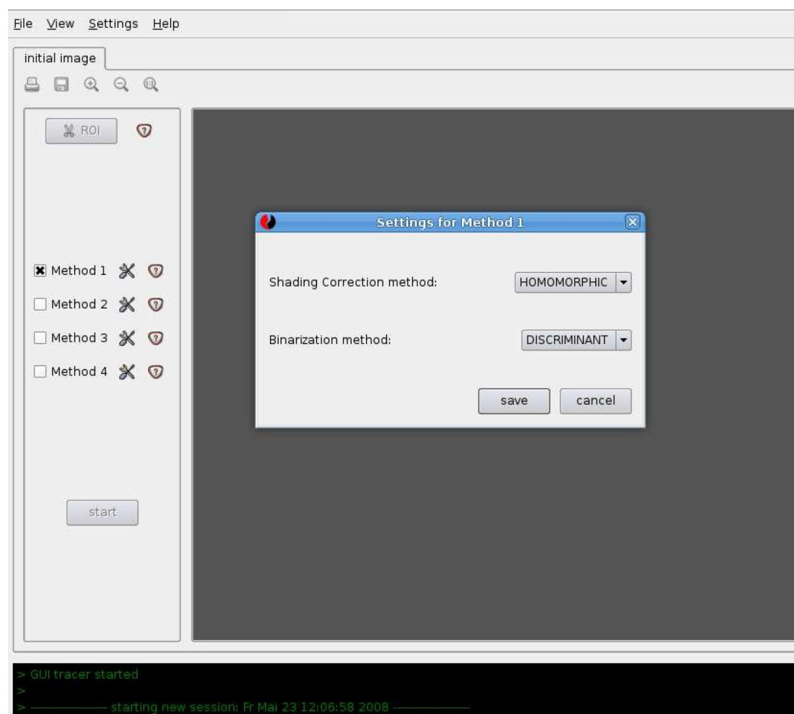

Bild 4.3: Settingsoptionen

verändert. Die Dialoge wurden aufgesplittet und befinden sich nun in kleinen, übersichtlichen Fenstern. Diese Fenster lassen sich durch drücken der Settings-Buttons neben dem jeweiligen Objekt öffnen und beinhalten ausschließlich Optionen zu den zugehörigen Objekten. Dadurch kommt es zu gelegentlichen Redundanzen einiger Dialoge, da beispielsweise der Vorverarbeitungsschritt “Binarisierung“ von Methode 1, 2 und 3 verwendet wird und somit auch in den Settingsdialogen von allen 3 Methoden verändert werden kann. Für den Anwender jedoch ist diese Form der Darstellung eine enorme Erleichterung, da er nicht von einem kompletten Optionsfenster mit dutzenden Einstellungsmöglichkeiten erschlagen wird, sondern genau die Settings vorfindet, die er zu dem entsprechenden Abschnitt gerade erwartet. Darüber hinaus ergänzen sich die Settings-Buttons sehr gut mit den Informations-Buttons, die in Kapitel 4.3.2 vorgestellt werden.

4.2.4 Region of interest

Die Auswahl der Region of interest(ROI) hat sich ebenfalls geringfügig geändert. Nunmehr öffnet sich kein neuer Tab in dem die ROI bei korrekter Auswahl angezeigt wird, sondern das Bild wird im aktuell bereits geöffneten Tab modifiziert. Zwar entsteht daraus der Nachteil, dass der Benutzer für jede ROI ein neues Bild laden muss, dennoch überwiegt der Faktor der Übersichtlichkeit und des Handlings der GUI. Der Benutzer muss nun zu einer Methode nicht mehr angeben, ob er das Originalbild oder die ROI des Bildes verwenden möchte, da immer das aktuell geladene Bild verarbeitet wird. Durch das Einsparen eines Tabs wird außerdem die umfangreiche Vorverarbeitung transparenter und besser überschaubar. Diese neue Implementierung entspricht daher mehr dem Selbstverständnis von Primus, das kein zweites Gimp oder Photoshop sein soll, sondern vielmehr ein Experimentiertisch für unterschiedliche Bildverarbeitungsverfahren auf Rissmustern.

4.2.5 Icons der GUI

Eine ansprechende grafische Oberfläche zeichnet sich zu einem großen Teil durch ihre Übereinstimmungen mit bekannten Programmen und ihren logischen Aufbau aus. Icons haben einen hohen Wiedererkennungswert und ermöglichen Anwendern eine schnelle Adaption der Bedienelemente. Das Symbol “Schere“ wird sofort mit Ausschneiden assoziiert und das Diskettensymbol wird als Speicherknopf automatisch akzeptiert. Als Entwickler möchte man sich jedoch weder in Unkosten stürzen, um an anschaulich und qualitativ hochwertige Icons zu gelangen, noch möchte man sie alle selber mit einem Grafiktool erstellen. Für die Benutzeroberfläche von Primus fiel die Wahl auf das Icon-Paket von Ximian, welches unter LGPL Lizenz vertrieben wird.[1]

“When putting together an application, many developers consider the visual appearance to be almost as important as the internal functionality of the application itself. After all, who wants to look at 16-color icons? [..] Ximian recognized this problem and made over a thousand high-color icons doused in alpha-channel goodness. [..] If you’re putting together an application, you are free to use every one of these icons. Ximian has released

[1] http://www.novell.com/coolsolutions/nnlsmag/assets/ooo-stock.zip

the icons under the LGPL. This means that you can use the icons in your program free of charge. If you want to fork the icons into your own icon collection, you need to retain the license and the original copyrights, but from there you're free to do what you want.“[2]

Diese Icons zeichnen sich durch eine Sammlung von mehr als Tausend Grafiken in unterschiedlichen Größen und Variationen aus. Die von Primus verwendeten Icon liegen im Ordner 30_prog/GUI/GuiImages und werden in QT wie folgt integriert:

```
setIcon(QIcon(QDir::currentPath() +
              "/GUI/GuiImages/stock_open.png"));
```

In diesem Beispiel wird das Icon stock_open.png als QIcon geladen und über die Funktion setIcon zugewiesen. Innerhalb von Primus liefert QDir::currentPath() immer den Pfad der main-Methode des Projekts, also den Speicherort von 30_prog/ auf der Festplatte.

4.3 Neue Elemente der GUI

4.3.1 Tracer für die GUI

Die grafische Benutzeroberfläche besitzt einen separaten Tracer. Dieser ist in einer objektorientierten Klasse namens “GuiTracer“ angelegt und ist von der QT-Klasse QTextEdit abgeleitet. Der Tracer wird bei Laufzeit gestartet und mit einem Zeitstempel initiiert. Der Zeitstempel ist der erste Eintrag und wird somit zu Beginn einer Sitzung als erstes protokolliert. Der Tracer speichert alle seine Einträge auf der Festplatte in einer Textdatei unter dem Verzeichnis “GUI/GuiTracer/GuiTracerLogFiles“. Er läuft permanent und zeichnet die wichtigsten Benutzeraktionen auf. Somit ist auch bei einem eventuellen System- und/oder GUI-Crash nachvollziehbar, wo der Fehler lag bzw. welches die zuletzt ausgeführten Aktionen war. Die Logdatei des Tracers kann anschließend auf Wunsch des Anwenders an einen Fehlerbericht z.B. per Mail angehangen werden. Auf ein automatisches Versenden der Logdatei bei einem Absturz oder Fehler wurde aus Datenschutzgründen bewusst verzichtet. Der GuiTracer kann zur Lauftzeit jederzeit über die Menüoption “View“

[2] http://www.novell.com/coolsolutions/feature/1637.html, Zitat von Kevin Breit

-> “Show/Hide GUI tracer“ in die GUI ein- und ausgeblendet werden. Die Klasse QTextEdit ermöglicht einige Standardoperationen auf dem eingeblendeten GuiTracer-Dialog, beispielsweise Kopieren oder Auswählen, die in Primus übernommen wurden. Ein direktes Editieren des Dialogs ist jedoch abgeschaltet.

4.3.2 Objektorientierte Dialoge

Bislang wurden Informationsdialoge statisch über die GUI gehandhabt. Jede Funktion musste erst programmiert werden, bevor anschließend vom Programmierer die Informationen dazu an den GUI-Entwickler weitergegeben wurden. Im Redesign wurden diese Dialoge objekorientiert gestaltet: Jede Funktion besitzt nun Selbstbeschreibungsfähigkeit. Wird eine neue Funktion entwickelt oder eine bestehende Funktion geändert, muss keine neue Funktionalität zur GUI hinzugefügt werden. Alle beschreibenden Information sind durch die *getInfo* jeder Funktion abrufbar. Somit wird eine Änderung der Funktionsbeschreibung automatisch von der grafischen Benutzeroberfläche übernommen.

4.3.3 Darstellung von Objekten anstelle von Pixeln

Ein besonderes Merkmal des Refactorings liegt in der Darstellung von Objekten anstelle von Pixeln. Die bisherige Version von Primus hat ausschließlich auf vollständigen Bildern gearbeitet. Diese wurde mit unterschiedlichen Methoden verarbeitet und anschliessend eins zu eins pixelweise in der GUI dargestellt. Im Refactoring werden die Bilddaten aus dem PUMA-Segmentierungsobjekt über einen Worker, in Kapitel 4.3.4 beschrieben, in Qt-Objekte umgewandelt. Die Objekte besitzen dann spezifische Eigenschaften und Methoden. Außerdem ist es möglich, über ein Qt-Objekt an sein korrespondierendes PUMA-Objekt zu gelangen, da alle Objekte eine eindeutige Id besitzen. Die grafischen Objekte werden einer *GraphicsScene* vom Typ *QGraphicsScene* hinzugefügt. In der GUI wird ein *GraphicsView* vom Typ *QGraphicsView* angezeigt, der die *GraphicsScene* darstellt. Die beiden abgeleiten Klassen lassen sich beliebig um Funktionalität erweitern .

4.3.4 GeometricObjectConverter (goc-Worker)

Der GeometricObjectConverter wurde als Worker speziell für die Umwandlung von PUMA Segmentierungsobjekten(SegObj) konzipiert. Die unterschiedlichen Funktionen des Workers sollen eine Umwandlung in alle darstellbaren Qt-Objekte ermöglichen.

```
#include <sstream>
#include <string>

#include "hippos/Chain.h"
#include "hippos/AtomLine.h"
#include "hippos/PointXY.h"
#include "hippos/LineArc.h"

#include "../../Architecture/Singleton/Tracer.h"
#include "../../Architecture/Singleton/Clock.h"

#include "../../GUI/MainWindow.h"
#include "../../GUI/GraphicsObjects/
                   GraphicsPath/GraphicsPath.h"

#include "GeometricObjectConverter.h"
```

Durch das Einbinden des Headers MainWindow.h erübrigt sich das Inkludieren der Klassen QVariant, QPointF und QPainterPath. Zudem muss die Klasse GraphicsPath einbezogen werden, da der Worker in der Funktion computePath bereits die Qt-Objekte berechnet und in einem QVector zurückgibt.

```
QVector<GraphicsPath*> THIS::computePath(
                                   SegObj* inputSegObj) {
 NIHCL_NS::Set alines =
               inputSegObj->PartsOfClass(META(AtomLine));
 QVector< GraphicsPath* > m_PathVector;
```

```
 int amountOfLines = 0;
 DO(alines,AtomLine,alptr)
  Represent* rep = Represent::castdown(alptr->getRep());
  if(rep == NULL) {
   rep = Chain::castdown(alptr->getRep(&META(Chain)));
  }
  if(rep != NULL && rep->isKindOf(META(Chain))) {
   Chain* chain = Chain::castdown(rep);
   chain->setToFirstLink();
   PointXY firstPoint = chain->nextPoint();
   QVariant startX = firstPoint.x();
   QVariant startY = firstPoint.y();
   QPointF start(startX.toDouble(), startY.toDouble());
   QPainterPath path(start);
   while(!chain->endOfChain()) {
    chain->Chain::setToNextLink();
    PointXY currentPoint = chain->nextPoint();
    QVariant currentX = currentPoint.x();
    QVariant currentY = currentPoint.y();
    QPointF currentPointF(currentX.toDouble(),
                          currentY.toDouble());
    path.lineTo(currentPointF);
   }
   GraphicsPath* currentPath =
                 new GraphicsPath(amountOfLines,
                                  path, m_MainWindow);
   m_PathVector.append(currentPath);
  }
  amountOfLines++;
 OD
 return m_PathVector;
}
```

Beim Erstellen des Vektors werden alle AtomLines aus dem PUMA-Segmentierungsobjekt umgewandelt. Der erste Punkt wird als Startpunkt für den Konstruktor des Qt-Objekts QPainterPath verwendet. Danach wird jeder weitere Punkt einer Atomline über die Funktion lineTo(QPointF, QPointF) dem QPainerPath hinzugefügt. Wenn der letzte Punkt einer jeweiligen Linie erreicht ist, im Falle chain->endOfChain() = true, wird ein neues Objekt vom Typ GraphicsPath als Pointer konstruiert und dem Ergebnisvektor angehängt.

4.3.5 Arbeiten auf den Objekten

Die von der grafischen Benutzeroberfläche zur Darstellung verwendeten Objekte sind alle in Subklassen abgelegt, die von den Qt-Klassen abgeleitet wurde. Die Subklassen wurden um einige Optionen, Variablen und Funktionen erweitert. Die Klasse GraphicsPath aus Kapitel 4.3.4 soll dies im folgenden verdeutlichen. Der Konstruktor besitzt 3 Parameter, von denen der erste die Id des Objekts repräsentiert. Jede Id ist einzigartig und ermöglicht neben einer Identifizierung des Grafikobjekts zudem das Zurückrechnen zum PUMA Segmentierungsobjekt. Der zweite Parameter ist das darzustellende Objekt vom Typ QPainterPath. Ein QPainterPath kann eine Vielzahl von geometrischen Formen beinhalten und mit einander verknüpfen. Der Worker aus Kapitel 4.3.4 beschränkt sich jedoch vorerst auf atomare Linien und fügt diese über die Funktion *pathTo* dem Grafikpfad hinzu. Als letzten Parameter bekommt jedes Objekt noch einen Pointer auf das Hauptfenster als *Parent* übergeben

4.3.6 Dokumentation

Die Dokumentation aller Klassen der GUI wurde komplett in Doxygen erstellt[3]. Die Konfigurationsdatei *doxygen.conf* befindet sich im Verzechnis 30_prog und ist momentan so eingestellt, das nur die Inhalte der Header der GUI rekursiv mit in die Dokumentation eingebunden werden. Bei der Dokumentation wurde standardkonform unter Verwendung eines Howtos von Kim Kulling[4] der Qt-Style verwendet. Ein Ausdruck der Dokumentation

[3] http://www.stack.nl/ dimitri/doxygen/
[4] http://www.selflinux.org/selflinux-devel/pdf/doxygen.pdf

befindet sich im Kapitel B.

Kapitel 5

Zusammenfassung

Die Ziele für die grafische Benutzeroberfläche wurden komplett umgesetzt. Wichtig dabei war, dass die GUI mindestens die gleichen Operationen ermöglicht wie ihr Vorgängermodel. Die Dialoge wurden neu gestaltet und das gesamte GUI-Framework objektorientiert aufgebaut. Die Funktionen zu den geladenen Bildern und ihren einzelnen Verarbeitungsschritten sind neu arrangiert worden. Buttons zum Speichern, Drucken, Zoomen, etc. wurden zu den jeweiligen Bildern hinzugefügt. Die überflüssigen Fortschrittsdialoge sind entfernt und durch eine simplere und übersichtlichere Struktur ersetzt worden. Die Funktionalität der SVM und der Datenbankanbindung bleibt komplett erhalten, jedoch wurde die grafische Darstellung neu visualisiert. Die *region of interest* hat eine komplette Umstellung erfahren. Die Auswahl erfolgt zwar immer noch in einem *QLabel*, jedoch wird kein weiterer Tab geöffnet, sondern das bearbeitete Bild im aktuellen Tab geladen. Die Methodenauswahl wurde anschaulicher gestaltet und wird nun ohne auf- und zuklappende Frames realisiert. Die Einstellungen zum Framework und den Vorverarbeitungsmethoden sind nun von einander getrennt. Dadurch bedingt öffnen sich nur noch kleine komfortable Dialogfenster anstelle eines einzigen großen Tab-Widgets. Neben dem Redesign wurden aber auch eine Reihe völlig neuer Funktionalitäten realisiert. Die GUI hat einen eigenen Tracer erhalten. Dieser protokolliert wichtige Ereignisse eigenständig und schreibt diese in eine Textdatei, die bei Bedarf per Mail an die Entwickler geschickt werden kann. Des weiteren wurde neben den *QLabels* ein weiteres Darstellungs-

element integriert: *QGraphicsView*. Durch diese Erweiterung ist es nun möglich, grafische Objekte in einer Szene darzustellen anstelle der Pixel. Diese Objekte sind wesentlich flexibler als einfache per-Pixel Darstellungen und können eigene Eigenschaften besitzen. Es ist somit möglich, Objekte zu selektieren und miteinander in Relation zu setzten.

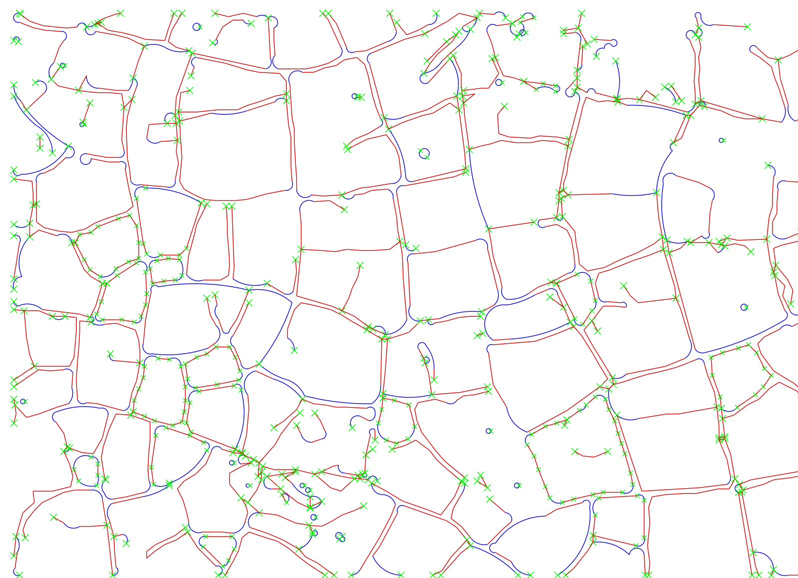

Bild 5.1: segmentiertes Rissmuster

Für weitere Arbeiten an Primus bleibt viel Raum. Die Segmentierung der Rissmuster ist sicherlich das spannendste Gebiet. Wie in Bild 5.1 zu erkennen ist, gibt es noch eine Vielzahl an Segmentierungsobjekten. Beispielsweise Flächen, in Form von beliebigen Polygonen, Kreisbogensegmenten oder beliebigen geometrischen Aneinanderreihungen von Objekten. Bild 5.2 zeigt ein beliebiges n-eckiges geschlossenes Polygon. Auf den Flächen von geschlossenen Objekten lassen sich viele weitere Aspekte herausarbeiten, sowohl visuell über die Darstellung in der GUI, als auch mathematisch durch das Objekt selbst.

Die Datenbank könnte ebenfalls eine Umstrukturierung erfahren. Die Webschnittstelle von Primus funktioniert momentan nur in eine Richtung: Es ist zwar möglich, Ergebnisse mit Primus in die Datenbank zu laden und im Web anzuzeigen, jedoch lassen sich keine Bilder

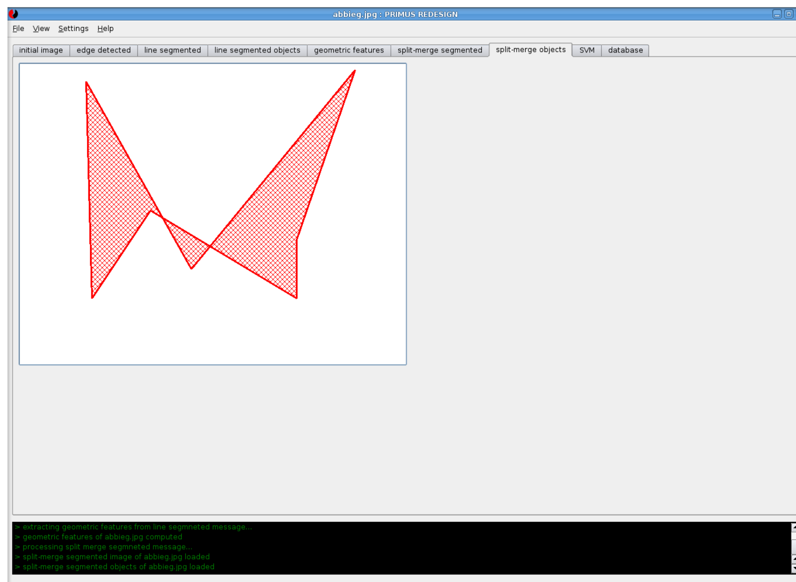

Bild 5.2: Beispiel eine beliebigen Polygons

unabhängig von Primus in die Datenbank schreiben. Außerdem lassen sich die Möglichkeiten der SVM weiter ausschöpfen und zusätzliche Eigenschaften der Rissmuster in die Analyse integrieren.

Literaturverzeichnis

[Bla04] Blanchette, Jasmin ; Summerfield, Mark (Hrsg.): *C++ GUI-Programmierung mit Qt3*. Addison-Wesley, 2004

[Egg06] *Kapitel* p. 69-75. In: Eggert, Gerhard: *Studies in conservation: To whom the cracks tell : A closer look at craquelure in glass and glaze*. London : International institute for conservation of historic and artistic works, 2006, S. 69–75. – ISSN 0039-3630

[SB06] Summerfield, Mark ; Blanchette, Jasmin: *C++ GUI Programming with Qt 4*. Pap/Cdr. Prentice Hall International, 2006

Verzeichnis der Bilder

1.1 Ausschnitt einer Aufnahme eines Rissmusterfotos aus dem Keramikmuseum Westerwald. Die Aufnahme wurde mit freundlicher Genehmigung des Museums von den Teilnehmern des Projektpraktikums zur Datensammlung erstellt. Insgesamt wurden mehrere hundert Fotografien erstellt. Die Internetpräsenz des Keramikmuseums Westerwald befindet sich aud der Webseite: http://www.keramikmuseum.de 9

1.2 Screenshot der ersten grafischen Benutzeroberfläche von Primus, die während des Projektpraktikums entstanden ist 10

2.1 Bild eines Spiderweb-Rissmuster, aus: http://eprints.ecs.soton.ac.uk/10040/ PDF: http://eprints.ecs.soton.ac.uk/10040/1/fazlythesis.pdf 14

2.2 Bild eines Circular-Rissmuster, aus: http://eprints.ecs.soton.ac.uk/10040/ PDF: http://eprints.ecs.soton.ac.uk/10040/1/fazlythesis.pdf 14

2.3 Bild eines Rectangle-Rissmuster, aus: http://eprints.ecs.soton.ac.uk/10040/ PDF: http://eprints.ecs.soton.ac.uk/10040/1/fazlythesis.pdf 14

2.4 Bild eines Unidirectional-Rissmuster, aus: http://eprints.ecs.soton.ac.uk/10040/ PDF: http://eprints.ecs.soton.ac.uk/10040/1/fazlythesis.pdf 14

3.1 Screenshot vom Qt Assistant der Firma Trolltech in dem die Dokumentation der Klasse QSqlDatabase dargestellt wird 21

4.1 Screenshot eines Fortschrittdialogs aus der grafischen Benutzeroberfläche des Projektpraktikums 26

4.2 Screenshot der alten Settingsoptionen aus der grafischen Benutzeroberfläche des Projektpraktikums 28

4.3 Screenshot der neuen Settingsoptionen des Redesigns 29

5.1 Bild eines bearbeiteten und segmentieren Rissmuster 38

5.2 Screenshot einer mit Primus erstellten Polygons 39

Anhang A

Installationshinweise

A.1 Systemvorraussetzungen

Ein derzeitiger Standard Pc ist für Primus völlig ausreichend. Linux ist zur Zeit noch das einzige Betriebssystem, auf dem Primus kompiliert. Zusätzlich müssen KONIHCL und PUMA auf dem System installiert sein. Eine Bildschirmauflösung von 1024x768 ist empfohlen, aber nicht zwingend notwendig.

A.2 svn und Pfade

Die Url zum svn lautet: https//svn.uni-koblenz.de/agas/projects/crackpattern. Das Redesign befindet sich im Ordner crackpattern/90_redesign/30_prog und lässt sich über ein dort liegendes Makefile kompilieren.

A.2.1 svn Revision

Die für das Redesgin verwendete svn Revision lautet: 25854.

A.3 Pakete

Folgende Pakete müssen zum Kompilieren auf dem Rechner vorhanden sein:

- libmagick9
- libmagick9-dev
- libmagick++9c2a
- libmagick++9-dev
- lapack3
- lapack3-dev
- atlas3-base
- atlas3-base-dev
- atlas3-headers
- libqt3-mt
- refblas3
- refblas3-dev
- libqt3-mt-dev
- libqt3-mt-headers
- libtqt4-qt3support
- libtqt4-core
- libqt4-dev
- libtqt4-gui
- libcv0.9.7-0
- libcv-dev
- libcvaux0.9.7-0
- libcvaux-dev
- libhighgui0.9.7-0
- libhighgui-dev
- fftw3
- fftw3-dev
- fftw2
- fftw-dev
- libg2c0
- libg2c0-dev
- libtool
- automake1.9
- tex4ht
- tk8.4
- libgsl0
- libgsl0-dev
- tetex-base
- tetex-bin
- tetex-doc
- tetex-extra
- transfig
- jpeg2ps
- cvs
- g++

A.4 Umgebungsvariablen

Um Primus erfolgreich kompilieren zu können, müssen einige Variablen auf bestimmte Ordner im System zeigen. Alle Einträge werden in die Datei *.bashrc* eingefügt und sind je nach Linux-Distribution erst nach einen Neustart des Terminals wirksam. Es handelt sich dabei um folgende Zeilen:

```
export ROBBIEDIR=<1.>
export LD_LIBRARY_PATH=<2.>
export PUMADIR=<3.>
export PUMAARCH=Linux
```

Erklärung:

1. : Der Pfad in dem der Ordner 90_redesign der lokalen SVN-Kopie von Primus liegt, in ihm befindet sich der Ordner 30_prog. Beispiel:

   ```
   export ROBBIEDIR=/home/USERNAME/svn/primus/crackpattern/
   90_redesign
   ```

2. : Der Pfad in dem die lokale SVN-Kopie von KONIHCL liegt. Beispiel:

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/USERNAME/
   KONIHCL/lib/.libs/

   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROBBIEDIR/
   30_prog/lib

   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
   ```

3. : Der Pfad in dem die lokale SVN-Kopie von Puma liegt. Beispiel:

   ```
   export PUMADIR=/home/USER/svn/puma
   ```

A.5 Qt

Qt sollte in Version 4.3.2 oder höher vorhanden sein. Der Auruf `qmake -version` sollte folgendens Ergebnis liefern: ”QMake version 2.01a” [1]. Ist dies nicht der Fall und die GUI sollte nicht kompilieren, hilft es bei installiertem Qt die Datei qmake durch eine aktuellere zu erstezen[2]: `sudo cp /usr/bin/qmake-qt4 /usr/bin/qmake`. Dieser Fehler tritt jedoch nur bei veralteten Versionen auf oder bei einer bereits eingerichteten Qt-3 Umgebung.

A.6 Settingsdatei

Vor dem ersten Start von Primus muss noch eine Datei kopiert werden: Die Datei test.dat muss vom Verzeichnis 90_redesign/30_prog/Config/manualTestDat/ nach 90_redesign/30_prog/Config/ kopiert oder verschoben werden[3].

A.7 Programmstart

Für den Programmstart existiert das ausführbare Shell-Script *Primus.sh*.

---

[1] oder eine höhere Version

[2] diese Dateien befinden sich im Verzeichnis /usr/bin/

[3] Mit einer Aktualisierung des Frameworks wird dies überflüssig werden

Anhang B

Anhang: Dokumentation

Die Dokumentation “Redesgin und Refactoring of the Primus GUI Refence Manual“ wurde mit Doxygen 1.5.3 erstellt und beginnt auf der nächsten Seite.

Redesign and Refactoring of the Primus GUI Reference Manual

Generated by Doxygen 1.5.3

Thu Jun 26 21:24:22 2008

Contents

1 Redesign and Refactoring of the Primus GUI Class Index **1**

1.1 Redesign and Refactoring of the Primus GUI Class List 1

2 Redesign and Refactoring of the Primus GUI Class Documentation **3**

2.1 GraphicsLine Class Reference 3

2.2 GraphicsPath Class Reference 5

2.3 GraphicsPolygon Class Reference 7

2.4 GraphicsScene Class Reference 8

2.5 GraphicsView Class Reference 9

2.6 GUIEvent Class Reference 10

2.7 GUIThread Class Reference 11

2.8 GuiTracer Class Reference 13

2.9 InitialLabel Class Reference 14

2.10 MainWindow Class Reference 16

Chapter 1

Redesign and Refactoring of the Primus GUI Class Index

1.1 Redesign and Refactoring of the Primus GUI Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

GraphicsLine (This class creates and represents line objects) 3
GraphicsPath (This class creates and represents path objects) 5
GraphicsPolygon (This class creates and represents polygon objects) 7
GraphicsScene (This class creates a graphics scene) 8
GraphicsView (This class creates a graphics view) 9
GUIEvent (This class manages incoming messages and forwards them to the GUI) . . 10
GUIThread (This class implements the thread which runs the GUI) 11
GuiTracer (This class creates a tracer within the GUI) 13
InitialLabel (This class creates the first label) 14
MainWindow (This class is derived from QMainWindow and creates the GUI for the system) 16

Chapter 2

Redesign and Refactoring of the Primus GUI Class Documentation

2.1 GraphicsLine Class Reference

This class creates and represents line objects.

```
#include <GraphicsLine.h>
```

Public Member Functions

- **GraphicsLine** (int id, qreal x1, qreal y1, qreal x2, qreal y2, **MainWindow** ∗parent)
  *The constructor for **GraphicsLine** (p. 3) objects.*

2.1.1 Detailed Description

This class creates and represents line objects.

GraphicsLine (p. 3) can be used to create simple lines consisting of only two points. Both the hoverEnterEvent and the hoverLeaveEvent are overwritten to enhance functionality, i.e. to display informations in the GUI. **GraphicsLine** (p. 3) inherits functionality of QGraphicsLineItem.

2.1.2 Constructor & Destructor Documentation

2.1.2.1 GraphicsLine::GraphicsLine (int *id*, qreal *x1*, qreal *y1*, qreal *x2*, qreal *y2*, MainWindow ∗ *parent*)

The constructor for **GraphicsLine** (p. 3) objects.

Parameters:

id An integer to identify each object within the GUI.

x1 qreal X-coordinate of the startpoint.

y1 qreal Y-coordinate of the startpoint.

x2 qreal X-coordinate of the endpoint.

y2 qreal Y-coordinate of the endpoint.

parent Sets the **MainWindow** (p. 16) as the parent of each constructed object.

The documentation for this class was generated from the following file:

- GUI/GraphicsObjects/GraphicsLine/GraphicsLine.h

2.2 GraphicsPath Class Reference

This class creates and represents path objects.

`#include <GraphicsPath.h>`

Public Member Functions

- **GraphicsPath** (int id, QPainterPath path, **MainWindow** *parent)

  *The constructor for **GraphicsPath** (p. 5) objects.*

- void **unSelect** ()

  This function unselects this object and redraws it.

- void **checkLength** (int value, int variation)

  This function checks the length to select this object or not.

- int **getId** ()

  Return the objects Id.

2.2.1 Detailed Description

This class creates and represents path objects.

GraphicsPath (p. 5) can be used to create complex lines consisting of a huge number of points. Both the hoverEnterEvent and the hoverLeaveEvent are overwritten to enhance functionality, i.e. to display informations in the GUI. **GraphicsPath** (p. 5) inherits functionality of QGraphicsPathItem.

2.2.2 Constructor & Destructor Documentation

2.2.2.1 GraphicsPath::GraphicsPath (int *id*, QPainterPath *path*, MainWindow * *parent*)

The constructor for **GraphicsPath** (p. 5) objects.

Parameters:

id An integer to identify each object within the GUI.

path QPainterPath that holds the whole path for this object. Due to the inherited functions from QGraphicsPathItem it is not necessary to construct the object at once, i.e. the function lineTo() can be used to add a line to an existing **GraphicsPath** (p. 5) object.

parent Sets the **MainWindow** (p. 16) as the parent of each constructed object.

2.2.3 Member Function Documentation

2.2.3.1 void GraphicsPath::checkLength (int *value*, int *variation*)

This function checks the length to select this object or not.

Parameters:

value An integer holding the chosen length of a line.

variation An integer holding the desired variation of the length given by the user input in the GUI.

The documentation for this class was generated from the following file:

- GUI/GraphicsObjects/GraphicsPath/GraphicsPath.h

2.3 GraphicsPolygon Class Reference

This class creates and represents polygon objects.

`#include <GraphicsPolygon.h>`

Public Member Functions

- **GraphicsPolygon** (int id, QPolygonF polygon, **MainWindow** *parent)
  The constructor for ***GraphicsPolygon*** *(*p. 7*) objects.*

2.3.1 Detailed Description

This class creates and represents polygon objects.

GraphicsPolygon (p. 7) can be used to create polygons. Both the hoverEnterEvent and the hoverLeaveEvent are overwritten to enhance functionality, i.e. to display informations in the GUI. **GraphicsPolygon** (p. 7) inherits functionality of QGraphicsPolygonItem.

2.3.2 Constructor & Destructor Documentation

2.3.2.1 GraphicsPolygon::GraphicsPolygon (int *id*, QPolygonF *polygon*, MainWindow * *parent*)

The constructor for **GraphicsPolygon** (p. 7) objects.

Parameters:

id An integer to identify each object within the GUI.

polygon QPolygonF that holds the whole polygon for this object. Therefore the polygon has to be constructed before the GraphicsPolygon's constructor is called.

parent Sets the **MainWindow** (p. 16) as the parent of each constructed object.

The documentation for this class was generated from the following file:

- GUI/GraphicsObjects/GraphicsPolygon/GraphicsPolygon.h

2.4 GraphicsScene Class Reference

This class creates a graphics scene.

`#include <GraphicsScene.h>`

Public Member Functions

- **GraphicsScene** (**MainWindow** *parent)
  *The constructor for a **GraphicsScene** (p. 8).*

2.4.1 Detailed Description

This class creates a graphics scene.

This **GraphicsScene** (p. 8) is meant to be used within a **GraphicsView** (p. 9). Any known graphical object can be visualized by adding it to the scene. Especially the objects **GraphicsLine** (p. 3), **GraphicsPath** (p. 5) and **GraphicsPolygon** (p. 7) will be added to the scene by the **MainWindow** (p. 16). **GraphicsScene** (p. 8) inherits functionality of QGraphicsScene.

2.4.2 Constructor & Destructor Documentation

2.4.2.1 GraphicsScene::GraphicsScene (MainWindow * *parent*)

The constructor for a **GraphicsScene** (p. 8).

Parameters:

parent Sets the **MainWindow** (p. 16) as the parent of this scene.

The documentation for this class was generated from the following file:

- GUI/GraphicsScene/GraphicsScene.h

2.5 GraphicsView Class Reference

This class creates a graphics view.

`#include <GraphicsView.h>`

Public Member Functions

- **GraphicsView** (**MainWindow** *parent)
  *The constructor for a **GraphicsView** (p. 9).*

2.5.1 Detailed Description

This class creates a graphics view.

This **GraphicsView** (p. 9) is meant to be used together with a **GraphicsScene** (p. 8). The view holds the scene and manages certain options. A more detailed description can be found in the Qt class documentation of QGraphicsView. **GraphicsScene** (p. 8) inherits functionality of QGraphicsScene.

2.5.2 Constructor & Destructor Documentation

2.5.2.1 GraphicsView::GraphicsView (MainWindow * *parent*)

The constructor for a **GraphicsView** (p. 9).

Parameters:

parent Sets the **MainWindow** (p. 16) as the parent of this scene.

The documentation for this class was generated from the following file:

- GUI/GraphicsView/GraphicsView.h

2.6 GUIEvent Class Reference

This class manages incoming messages and forwards them to the GUI.

`#include <GUIEvent.h>`

Public Member Functions

- **GUIEvent** (Message *message)
  The constructor for GUI events.
- Message * **getMessage** ()
  Returns a pointer to a message.

2.6.1 Detailed Description

This class manages incoming messages and forwards them to the GUI.

This class is derived from the class QEvent and is used for sending messages from the primus framework to the GUI. This mechanism has to be used because of the multi-threaded design of the architecture. **GUIEvent** (p. 10) inherits functionality of QEvent.

2.6.2 Constructor & Destructor Documentation

2.6.2.1 GUIEvent::GUIEvent (Message * *message*) `[inline]`

The constructor for GUI events.

The constructor can only be called with a pointer to a message.

Parameters:

message Pointer to the message that will be posted to the GUI.

2.6.3 Member Function Documentation

2.6.3.1 Message* GUIEvent::getMessage () `[inline]`

Returns a pointer to a message.

Returns:

Pointer to the message that this event carries.

The documentation for this class was generated from the following file:

- GUI/GUIEvent.h

2.7 GUIThread Class Reference

This class implements the thread which runs the GUI.

`#include <GUIThread.h>`

Public Member Functions

- **GUIThread** (MessageModule *module)
  The constructor of ***GUIThread*** *(p. 11).*
- virtual ~**GUIThread** ()
  The destructor deletes the QApplication object.
- virtual void **run** ()
  This functions starts the GUI.
- void **postGUIEvent** (Message *message)
  This function posts messages to the GUI.
- void **sendMessage** (Message *message)
  This function sends messages from the GUI.

Public Attributes

- MessageModule * **m_Module**
  This variable holds the instance of GUI module.

2.7.1 Detailed Description

This class implements the thread which runs the GUI.

2.7.2 Constructor & Destructor Documentation

2.7.2.1 GUIThread::GUIThread (MessageModule * *module*)

The constructor of **GUIThread** (p. 11).

The constructor only stores the pointer to GUI module in a member variable.

Parameters:

module Pointer to the GUIModule.

2.7.3 Member Function Documentation

2.7.3.1 virtual void GUIThread::run () `[virtual]`

This functions starts the GUI.

The run method creates the QApplication and the main window. It shows the main window and starts the QApplication.

2.7.3.2 void GUIThread::postGUIEvent (Message * *message*)

This function posts messages to the GUI.

This method posts the given message to the GUI by using the postEvent method of QApplication. Therefore the message is packed into a Qt event class that is capable of holding a pointer to a message (**GUIEvent** (p. 10)).

Parameters:

message Pointer to the message that will be posted to the GUI.

2.7.3.3 void GUIThread::sendMessage (Message * *message*)

This function sends messages from the GUI.

This method sends a given message to the module by using the pushIn() method of MessageModule.

Parameters:

message Pointer to the message sending to module.

The documentation for this class was generated from the following file:

- GUI/GUIThread.h

2.8 GuiTracer Class Reference

This class creates a tracer within the GUI.

`#include <GuiTracer.h>`

2.8.1 Detailed Description

This class creates a tracer within the GUI.

The **GuiTracer** (p. 13) logs certain events and writes to a logfile which is located in 30_-prog/GUI/GuiTracer/GuiTracerLogFiles. Just like the normal tracer it is very useful to trace bugs and grants the possibility to send the resulting .txt file to the developers. In addition GUI-internal events and possible errors are reported up to an eventual crash.

The documentation for this class was generated from the following file:

- GUI/GuiTracer/GuiTracer.h

2.9 InitialLabel Class Reference

This class creates the first label.

`#include <InitialLabel.h>`

Public Member Functions

- **InitialLabel** (**MainWindow** *parent)
  The constructor for an ***InitialLabel*** *(p. 14).*
- bool **getPressState** ()
  Return the press state of the mouse.
- bool **getReleaseState** ()
  Return the release state of the mouse.
- int **getXPressCoord** ()
  Returns the x-Coordinate of the latest Press-Event.
- int **getYPressCoord** ()
  Returns the y-Coordinate of the latest Press-Event.
- int **getXReleaseCoord** ()
  Returns the x-Coordinate of the latest Release-Event.
- int **getYReleaseCoord** ()
  Returns the y-Coordinate of the latest Release-Event.
- void **setMaxima** (int width, int height)
  Sets the maximum width and height for the selection.
- void **setRoiSelectionPerformed** ()
  Hides the rubberband and resets some flags.

2.9.1 Detailed Description

This class creates the first label.

This label holds the pixmap of an image. The three mouse events for this label are overwritten. **InitialLabel** (p. 14) inherits functionality of QLabel.

2.9.2 Constructor & Destructor Documentation

2.9.2.1 InitialLabel::InitialLabel (MainWindow * *parent*)

The constructor for an **InitialLabel** (p. 14).

Parameters:

parent Sets the **MainWindow** (p. 16) as the parent of this label.

2.9.3 Member Function Documentation

2.9.3.1 bool InitialLabel::getPressState ()

Return the press state of the mouse.

Returns:

returns true if the mouse button is pressed.

2.9.3.2 bool InitialLabel::getReleaseState ()

Return the release state of the mouse.

Returns:

returns true if the mouse button is released.

2.9.3.3 void InitialLabel::setRoiSelectionPerformed ()

Hides the rubberband and resets some flags.

After a successfull selection the rubberband is hidden and the mouse-state flags are resetted.

The documentation for this class was generated from the following file:

- GUI/InitialLabel/InitialLabel.h

2.10 MainWindow Class Reference

This class is derived from QMainWindow and creates the GUI for the system.

`#include <MainWindow.h>`

Public Slots

- void **showLineSegmentedObjectsByLength** ()

  This slot marks any visible line segmented object that has a certain length.

Signals

- void **shadingCorrectedImageMessageArrived** (ShadingCorrectedImageM *shadingCorrectedImageMessage)

  This signal is emitted when a shading corrected image arrived.

- void **binarizedImageMessageArrived** (BinarizedImageM *binarizedImageMessage)

  This signal is emitted when a binarized image arrived.

- void **haralickMessageArrived** (DoHaralickFeatureClassificationM *haralickMessage)

  This signal is emitted when a haralick classification arrived.

- void **gaussFilteredImageMessageArrived** (GaussFilteredImageM *gaussFilteredImageMessage)

  This signal is emitted when a gaussian filtered image arrived.

- void **morphedImageMessageArrived** (MorphedImageM *morphedImageMessage)

  This signal is emitted when a morphed image arrived.

- void **edgeDetectedImageMessageArrived** (EdgeDetectedImageM *edgeDetectedImageMessage)

  This signal is emitted when an edge detected image arrived.

- void **lineSegmentedImageMessageArrived** (LineSegmentedImageM *lineSegmentedImageMessage)

  This signal is emitted when a line segmented image arrived.

- void **splitMergeSegmentedImageMessageArrived** (SplitMergeSegmentedImageM *splitMergeSegmentedImageMessage)

  This signal is emitted when a split merge segmented image arrived.

- void **trainResultMessageArrived** (TrainResultM *trainResultMessage)

  This signal is emitted when a train result arrived.

- void **predictResultMessageArrived** (PredictResultM *predictResultMessage)

  This signal is emitted when a predict result arrived.

Public Member Functions

- **MainWindow** (**GUIThread** ∗guiThread=0, QWidget ∗parent=0)

  *The constructor initializes the **MainWindow** (p. 16).*

- ∼**MainWindow** ()

  At the time the destructor is empty.

- void **trace** (QString traceString)

  *This function is used to address the **GuiTracer** (p. 13).*

- bool **getRoiQPushButtonStatus** ()

  Returns the state of the QPushButton roiQPushButton.

- void **setLineInfoLabel1Text** (QString text)

  This function sets the text of lineSegmentedObjectsInfoLabel1.

- void **setLineInfoLabel2Text** (QString text)

  This function sets the text of lineSegmentedObjectsInfoLabel2.

- void **unselectAllLineSegmentedObjects** ()

  This function unselects all possibly selected line objects of the line segmentation.

Public Attributes

- QString **imageName**

  QT-String to hold the name of the original image.

2.10.1 Detailed Description

This class is derived from QMainWindow and creates the GUI for the system.

The **MainWindow** (p. 16) is the central component of the GUI. Messages to and from the framework are received here. A number of images as well as grafics objects can be displayed within the main window. **MainWindow** (p. 16) inherits functionality of QMainWindow.

2.10.2 Constructor & Destructor Documentation

2.10.2.1 MainWindow::MainWindow (GUIThread ∗ *guiThread* = 0, QWidget ∗ *parent* = 0)

The constructor initializes the **MainWindow** (p. 16).

The constructor of the main window.

2.10.3 Member Function Documentation

2.10.3.1 void MainWindow::trace (QString *traceString*)

This function is used to address the **GuiTracer** (p. 13).

This function appends a string to the tracer and writes it to the logfile.

Parameters:

traceString Holds the string that will be posted to the tracer and written to the log file.

2.10.3.2 bool MainWindow::getRoiQPushButtonStatus ()

Returns the state of the QPushButton roiQPushButton.

This function returns true if the roiQPushButton is down, otherwise false.

Returns:

true if the roiQPushButton is pressed, otherwise false.

2.10.3.3 void MainWindow::setLineInfoLabel1Text (QString *text*)

This function sets the text of lineSegmentedObjectsInfoLabel1.

This function changes the text of one of the information boxes in the GUI. In this way, any other object can display data within the specified label. setLineInfoLabel1Text is responsible for the text within the QLabel lineSegmentedObjectsInfoLabel1 used in the line segmentation tab.

Parameters:

text Holds the string that will be shown in the upper information label of the line segmentation.

2.10.3.4 void MainWindow::setLineInfoLabel2Text (QString *text*)

This function sets the text of lineSegmentedObjectsInfoLabel2.

This function changes the text of one of the information boxes in the GUI. In this way, any other object can display data within the specified label. setLineInfoLabel2Text is responsible for the text within the QLabel lineSegmentedObjectsInfoLabel2 used in the line segmentation tab. In addition a line has to be selected to enable the length comparison. In order to do so, this function disables and enables the lineSegmentedObjectsLengthButton to block or allow the comparison.

Parameters:

text Holds the string that will be shown in the lower information label of the line segmentation.

2.10.3.5 void MainWindow::unselectAllLineSegmentedObjects ()

This function unselects all possibly selected line objects of the line segmentation.

This function changes the text of one of the information boxes in the GUI. In this way, any other object can display data within the specified label. setLineInfoLabel2Text is responsible for the text within the QLabel lineSegmentedObjectsInfoLabel2 used in the line segmentation tab. In addition a line has to be selected to enable the length comparison. In order to do so, this function disables and enables the lineSegmentedObjectsLengthButton to block or to allow the comparison. This is necessary, because the objects have no knowledge of each other. Only the view that contains them all knows each of them.

2.10.3.6 void MainWindow::showLineSegmentedObjectsByLength () `[slot]`

This slot marks any visible line segmented object that has a certain length.

This method marks all line segmented objects by the user selection given by the currently selected object and the desired value selected in the lineSegmentedObjectsLengthQSpinBox.

2.10.3.7 void MainWindow::shadingCorrectedImageMessageArrived (ShadingCorrectedImageM * *shadingCorrectedImageMessage*) `[signal]`

This signal is emitted when a shading corrected image arrived.

Parameters:

shadingCorrectedImageMessage This message transports a shading corrected image.

2.10.3.8 void MainWindow::binarizedImageMessageArrived (BinarizedImageM * *binarizedImageMessage*) `[signal]`

This signal is emitted when a binarized image arrived.

Parameters:

binarizedImageMessage This message transports a binarized image.

2.10.3.9 void MainWindow::haralickMessageArrived (DoHaralickFeatureClassificationM * *haralickMessage*) `[signal]`

This signal is emitted when a haralick classification arrived.

Parameters:

haralickMessage This message transports a haralick classification.

2.10.3.10 void MainWindow::gaussFilteredImageMessageArrived (GaussFilteredImageM * *gaussFilteredImageMessage*) `[signal]`

This signal is emitted when a gaussian filtered image arrived.

Parameters:

gaussFilteredImageMessage This message transports a gaussian filtered image.

2.10.3.11 void MainWindow::morphedImageMessageArrived (MorphedImageM * *morphedImageMessage*) [signal]

This signal is emitted when a morphed image arrived.

Parameters:

morphedImageMessage This message transports a morphed image.

2.10.3.12 void MainWindow::edgeDetectedImageMessageArrived (EdgeDetectedImageM * *edgeDetectedImageMessage*) [signal]

This signal is emitted when an edge detected image arrived.

Parameters:

edgeDetectedImageMessage This message transports an edge detected image.

2.10.3.13 void MainWindow::lineSegmentedImageMessageArrived (LineSegmentedImageM * *lineSegmentedImageMessage*) [signal]

This signal is emitted when a line segmented image arrived.

Parameters:

lineSegmentedImageMessage This message transports a line segmented image.

2.10.3.14 void MainWindow::splitMergeSegmentedImageMessageArrived (SplitMergeSegmentedImageM * *splitMergeSegmentedImageMessage*) [signal]

This signal is emitted when a split merge segmented image arrived.

Parameters:

splitMergeSegmentedImageMessage This message transports a split merge segmented image.

2.10.3.15 void MainWindow::trainResultMessageArrived (TrainResultM * *trainResultMessage*) [signal]

This signal is emitted when a train result arrived.

Parameters:

trainResultMessage This message transports a train result.

2.10.3.16 void MainWindow::predictResultMessageArrived (PredictResultM * *predictResultMessage*) [signal]

This signal is emitted when a predict result arrived.

Parameters:

predictResultMessage This message transports a predict result.

The documentation for this class was generated from the following file:

- GUI/MainWindow.h

Index

binarizedImageMessageArrived
- MainWindow, 19

checkLength
- GraphicsPath, 5

edgeDetectedImageMessageArrived
- MainWindow, 20

gaussFilteredImageMessageArrived
- MainWindow, 19

getMessage
- GUIEvent, 10

getPressState
- InitialLabel, 15

getReleaseState
- InitialLabel, 15

getRoiQPushButtonStatus
- MainWindow, 18

GraphicsLine, 3
- GraphicsLine, 3

GraphicsPath, 5
- checkLength, 5
- GraphicsPath, 5

GraphicsPolygon, 7
- GraphicsPolygon, 7

GraphicsScene, 8
- GraphicsScene, 8

GraphicsView, 9
- GraphicsView, 9

GUIEvent, 10
- getMessage, 10
- GUIEvent, 10

GUIThread, 11
- GUIThread, 11
- postGUIEvent, 12
- run, 11
- sendMessage, 12

GuiTracer, 13

haralickMessageArrived
- MainWindow, 19

InitialLabel, 14
- getPressState, 15
- getReleaseState, 15
- InitialLabel, 14
- setRoiSelectionPerformed, 15

lineSegmentedImageMessageArrived
- MainWindow, 20

MainWindow, 16
- binarizedImageMessageArrived, 19
- edgeDetectedImageMessageArrived, 20
- gaussFilteredImageMessageArrived, 19
- getRoiQPushButtonStatus, 18
- haralickMessageArrived, 19
- lineSegmentedImageMessageArrived, 20
- MainWindow, 17
- morphedImageMessageArrived, 19
- predictResultMessageArrived, 20
- setLineInfoLabel1Text, 18
- setLineInfoLabel2Text, 18
- shadingCorrectedImageMessageArrived, 19
- showLineSegmentedObjectsByLength, 19
- splitMergeSegmentedImageMessageArrived, 20
- trace, 18
- trainResultMessageArrived, 20
- unselectAllLineSegmentedObjects, 18

morphedImageMessageArrived
- MainWindow, 19

postGUIEvent
- GUIThread, 12

predictResultMessageArrived
- MainWindow, 20

run
- GUIThread, 11

sendMessage
- GUIThread, 12

setLineInfoLabel1Text
- MainWindow, 18

setLineInfoLabel2Text
- MainWindow, 18

setRoiSelectionPerformed
- InitialLabel, 15

shadingCorrectedImageMessageArrived
- MainWindow, 19

showLineSegmentedObjectsByLength
MainWindow, 19
splitMergeSegmentedImageMessageArrived
MainWindow, 20

trace
MainWindow, 18
trainResultMessageArrived
MainWindow, 20

unselectAllLineSegmentedObjects
MainWindow, 18