Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Institute of Clinical Chemistry,
Mass Spectrometric Proteomics

Hochschule für Angewandte Wissenschaften Hamburg

Fakultät Life Sciences

Entwicklung eines Softwaremoduls zur Steuerung eines Verfahrtischsystems für die automatisierte Laserablation von Gewebeproben

Bachelor of Science

im Studiengang Medizintechnik

vorgelegt von

Lea Song-I Park

[redacted]

Hamburg

am 20. September 2021

1. Gutachter: Prof. Dr. Petra Margaritoff (HAW Hamburg)

2. Gutachter: Dr.-Ing. Jan Hahn (Universitätsklinikum Hamburg-Eppendorf)

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

IMID	Immune-mediated inflammatory disease
SRP	Single responsibility principle
GUI	Graphical user interface
PI	Physik Instrumente

Glossar

Begriff	Definition/Erklärung
Omics-Verfahren	Omics-Verfahren sind Methoden, die sich auf die umfassende oder globale Evaluierung von Molekülsets fokussieren. Abhängig von dem betrachteten Molekülset (Genabschnitte, Proteine, Metaboliten etc.) werden die Omics-Verfahren in Kategorien eingeteilt: z.B. Genomics, Proteomics, Metabolomics u.v.m. [1]
Optische Kohärenztomografie (OCT)	OCT ist ein nicht-invasives, bildgebendes Verfahren zur 2D- und 3D-Darstellung von Gewebe. Es verwendet Licht im infrarotnahen Wellenlängenbereich, das in einem Interferometer in einen Referenzstrahl und den Probenstrahl geteilt wird. Beim Durchtreten des Probenstrahls durch das Gewebe wird er abhängig von der Durchdringungstiefe unterschiedlich stark zerstreut. Das reflektierte Licht wird mit dem Referenzlicht am Ausgang des Interferometers kombiniert, um eine Interferenz zu erreichen. Die Lichtintensität des resultierenden Strahls kann durch einen Photodetektor aufgenommen und für die Bildgebung analysiert werden. Durch das laterale Scannen des Gewebes können die Schichten zu einem 3D-Bild kombiniert werden. [2]
Design Pattern	Entwurfsmuster: Wiederverwendbare Lösungsschablonen für gängige Probleme in der Softwarearchitektur/-entwicklung. Hier referenziert Design Pattern zu den sogenannten Creational Patterns, die eine Unterkategorie von den Entwurfsmustern ist, die sich mit der Erzeugung von Objekten befasst.
Instanziierung	Erzeugung eines Objekts während der Laufzeit einer Software
Pure virtual function	Virtuelle Funktion: eine Funktion von der erwartet wird, dass sie in den abgeleiteten Klassen neu definiert wird. Stellt sicher, dass die richtige Funktion für das Objekt aufgerufen wird, unabhängig vom Ausdruck, mit dem die Funktion aufgerufen wird. Rein virtuell bedeutet, dass die Methode gleich Null gesetzt wird und damit die Methoden nicht mehr aufgerufen werden können, da kein Objekt erstellt werden kann. Die abgeleiteten Klassen sind also gezwungen die Methoden erst zu implementieren, um ein Objekt zu erzeugen.
Parent, Child	Die sog. Parent-Klasse oder Objekt ist eine übergeordenete Klasse/Objekt. Die Child-Klasse/-Objekt ist die untergeordnete oder auch erbende Klasse/Objekt.
Flags	Statusindikatoren, die als Kennzeichen für bestimmte Zustände gesetzt werden können. Häufig wird eine Boole'sche Variable verwendet.
Tupel	Hier der Variablentyp std::tuple, der mehrere Elemente von verschiedenen Typen enthalten kann. Auf die Elemente des std::tuple kann durch verschiedene Methoden zugegriffen werden (z.B. std::get(tuple)).

Struktur	Zusammenfassung von Datenelementen (auch unterschiedlichen Typs) unter einem Namen
Coverage Analyse	Eine Coverage Analyse in Bezug auf die Softwareentwicklung überprüft ob die Anforderungen einer Softwareentwicklungsphase in die nächste aufgenommen (Lastenheft zu Softwareanforderungen) bzw. durch Testfälle überprüft wurden. Die Deckung der Anforderungen wird in Prozent angegeben.

Die Begriffe aus dem Glossar sind im Text in *kursivem* Format markiert.

1. Einleitung und Aufgabenstellung

1.1 Einleitung

Ungefähr 5-7% der heutigen Bevölkerung leidet an immun-vermittelten Entzündungskrankheiten (IMID, engl. Immune-mediated inflammatory disease), wie zum Beispiel Rheumatoide Arthritis, Autoimmune Hepatitis, Multiple Sklerose und Morbus Crohn, nur um ein paar zu nennen [3]. Patienten mit diesen Krankheiten zeigen eine Gruppe von typischen Symptomen und stark behindernde, chronische Beschwerden, die einen gemeinsamen entzündlichen Signalweg besitzen. Ein Signalweg ist ein physiologischer Prozess bei dem eine Reihe von Wechselwirkungen zu einer Änderung in der Zelle führen.
Die derzeitig verfügbaren Therapien für IMIDs basieren hauptsächlich auf der Einsetzung von immunsuppressiven Medikamenten mit starken Nebenwirkungen, die allerdings keine Heilung der Krankheiten darstellen. Aus diesem Grund ist die Erforschung von alternativen Therapien notwendig, auch um effektive biologische Therapien für Patienten verfügbar zu machen [4].
Die Forschungsgruppe „The Molecular Inter- and Intranet of Inflammation“ (M3I) am Universitätsklinikum Hamburg-Eppendorf (UKE), befasst sich in Kooperation mit dem Heinrich-Pette-Institut und dem Max-Planck-Institut für Struktur und Dynamik der Materie mit der Analyse von Proteinen und Lipiden mittels quantitativer Massenspektrometrie sowie anderer *Omics-Verfahren*. Ziel ist es, die molekularen Prozesse und den Zusammenhang von Entzündungen und der Regeneration von Gewebe in beispielloser Auflösung zu erforschen und durch das erlangte Verständnis neue Therapien für IMIDs zu entwickeln.

1.2 PIRL-basierte Massenspektrometrie

Zur Entnahme von Gewebeproben wird ein Pikosekundeninfrarotlaser (PIRL) eingesetzt, der den Effekt der Desorption durch impulsive Schwingungsanregung (DIVE, engl. Desorption By Impulsive Vibrational Excitation) verwendet. Aufgrund der Verwendung der Wellenlänge von 2,92 – 3,15 µm wird die Energie des Lasers von den Wassermolekülen direkt in Translationsbewegungen umgewandelt [5]. Der Pikosekundenpuls des PIRLs erzeugt einen ultraschnellen Übergang zwischen flüssigem und gasförmigem Aggregatzustand von Wasser und ermöglicht dabei, die Übertragung von thermischer und akustischer Energie an das Umgebungsgewebe minimal zu halten [5].
Dies resultiert in der Fähigkeit des PIRL’s, ohne signifikante Wärmebildung Gewebe vaporisieren zu können, wodurch die Bestandteile der Zelle nicht zerstört werden. Das entstehende Aerosol kann dann auf einem Objektträger oder auf einem Glasfaserfilter aufgefangen und anschließend mittels einer Massenspektrometrie analysiert werden.
Im Vergleich mit der herkömmlichen Methode zur Gewinnung von Gewebeproben für die Massenspektrometrie, der mechanischen Homogenisierung, resultiert die Probenaufbereitung durch den PIRL in einer höheren Menge an intakten Proteinen und geringeren enzymatischen Abbaureaktionen [6]. Die PIRL-DIVE-Methode ist vergleichbar mit einer Momentaufnahme der dynamischen Änderungen in der Zelle zum Zeitpunkt der Laserbestrahlung.

1.3 Aufgabenstellung

Eine ausgewählte Probenentnahme ist wichtig, um Gewebe effektiv auf bestimmte Merkmale zu untersuchen, wie zum Beispiel der Zustand in Tumorgewebe. Ideal wäre eine Analyse ausschließlich von Tumorgewebe, um es mit gesundem Gewebe vergleichen zu können.
Hierzu wird derzeit am UKE ein Verfahren entwickelt, welches eine automatisierte Ablation auf Basis von Bilddaten ermöglichen soll. Durch dieses Verfahren soll es möglich werden die Grenze zwischen verschiedenen Gewebearten (z.B. Tumor- und gesundem Gewebe) deutlich zu machen.
Ziel dieser Arbeit ist es, ein Softwaremodul zu entwickeln, das die Gewebeprobe automatisch auf Grundlage von gegebenen Datenpunkten über ein Verschiebetischsystem verfährt, um so die vorher festgelegt Fläche mit dem Laser ablatieren zu können.
Die Datenpunkte werden durch ein anderes Plugin mittels dem optischen Kohärenztomografie-System (OCT, Optical Coherence Tomography) erarbeitet und über die existierende Hauptsoftware smartLab zur Verfügung gestellt.
Derzeit werden die Versuche durch die Lenkung des Laserstrahls durchgeführt. Dies hat allerdings den Nachteil, dass ca. ein Viertel des Gewebeaerosols bei der Ablation verloren geht. Der Grund ist, dass das Gewebeaerosol auf einem Objektträger aufgefangen wird (Abbildung 1). Da sich aber der Laserstrahl und nicht der Tisch bewegt, zerstört der Laser jegliches Material auf dem Objektträger, das in den Strahlengang kommt.
Die Entwicklung des Plugins würde dazu beitragen, den Forschungsprozess deutlich zu beschleunigen und hat darüber hinaus das Potenzial für andere Versuche einsetzbar zu sein, die Verfahrtischsysteme verwenden.

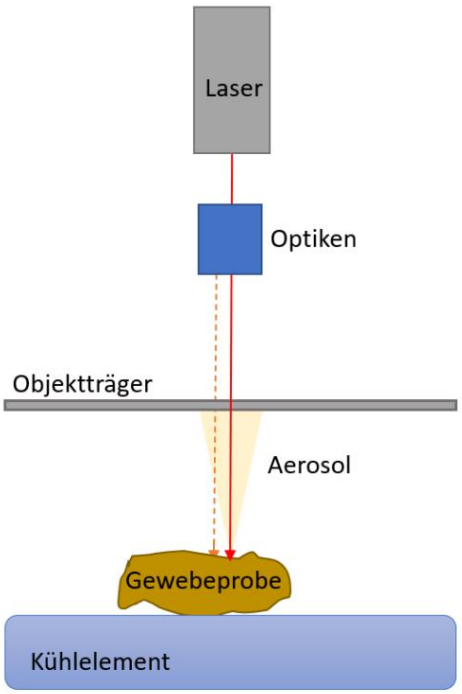

Abbildung 1: Schema des derzeitigen Versuchsaufbaus

1.4 Stand der Technik

1.4.1 Stand Softwareprojekts

smartLab

Die Hauptsoftware ist smartLab. Programmiert wurde smartLab in der Programmiersprache C++ mit Qt und ist eine am Laser Zentrum Hannover e.V. entwickelte Desktopanwendung zum Einbinden von Softwaremodulen für das Verwenden und Bedienen von verschiedenen Hardwareprodukten. Die Softwaremodule werden als Plugins eingebunden und verwendet. Weiterhin ist es möglich durch sogenannte „Jobs“ im smartLab mehrere Plugins über smartLab für einen bestimmten Versuchsaufbau zu verwenden. Beispielsweise könnte ein Job definiert werden, der das Verfahren der Stage zu einer bestimmten Position und die

Aufnahme der Gewebeprobe durch eine Kamera kombiniert, ohne dass manuell etwas bedient werden muss.

Plugin

Zu Beginn der Arbeit waren die Klassen für das manuelle Verfahren der Tische angelegt (qMotionPlugin, controllerFactory, controller, stage). Die Funktionalität des Verfahrens war noch nicht entwickelt und es existierte lediglich ein grobes Konzept.

1.4.2 Verwendete Hardware

Folgende Controller und Vefahrtische wurden für die Entwicklung des Plugins verwendet:

- Physik Instrumente Controller:
  - C-884 SN 121005110
  - PI C-867 Piezomotor Controller SN 0120063595
- PI-Linearverfahrtische:
  - M112.2DG1 (max. Geschwindigkeit: 1,5 mm/s, Stellweg: 23 mm)
  - M112.2DG1 (max. Geschwindigkeit: 1,5 mm/s, Stellweg: 23 mm)
  - M112.2DG1 (max. Geschwindigkeit: 1,5 mm/s, Stellweg: 23 mm)
  - M-687.UL:X (max. Geschwindigkeit: 50 mm/s, 135 mm)
  - M-687.UL:Y (max. Geschwindigkeit: 50 mm/s, 85 mm)

Derzeit sind die drei Tische des Modelltyps M112.2DG1 für das Verfahren während der Ablation selbst bestimmt (in Zusammenhang mit dem Controller C-884). Die anderen zwei Tische sind ein Tischsystem vom Modelltyp M-687.UL (angeschlossen am Controller PI C-867) und werden für die Probensammlung eingeplant (Verfahren des Objektträgers). Diese Tische wurden gewählt, da sie weitaus schneller sind, größere Distanzen überbrücken können und dadurch die Probensammlung schnell und effizient stattfinden kann.

2. Material & Methoden

Dieses Kapitel beschreibt die gegebenen Anforderungen, die verwendeten Tools und die *Design Patterns* für die Entwicklung des Plugins. Außerdem wird auch die Gesamtarchitektur der Anwendung und die Einbindung in die smartLab-Software zum Startzeitpunkt der Arbeit skizziert.

2.1 Anforderungen (Lastenheft)

Folgende Anforderungen waren im Lastenheft gegeben:

Tabelle 1: Lastenheftanforderungen

Key	Anforderung
LH1	Es soll ein Softwaremodul entwickelt werden, das mehrere Controller bedienen kann, welche jeweils mehrere Tische bedienen können.
LH2	Das Softwaremodul soll als Plugin für die existierende Hauptsoftware smartLab in C++ mit Qt entwickelt werden.
LH3	Alle angeschlossenen Controller und Tische sollen zusammengefasst in einem Userinterface angezeigt und bedienbar sein.
LH4	Die Tische sollen per Buttonklick mit einer bestimmten Geschwindigkeit für eine bestimmte Strecke fahren.
LH5	Das Timing der Fahrt soll bestimmbar sein, um das Triggern des Lasers synchronisieren zu können.
LH6	Die Tische sollen per Buttonklick bis zur maximalen/minimalen Position verfahrbar sein.
LH7	Es soll eine Möglichkeit zum automatisierten Abfahren in einem bestimmten Fahrmuster (Makro) geben.
LH8	Es soll möglich sein über eine Schnittstelle ein bestimmtes Lasermuster zu übergeben, anhand dessen das Plugin die Fahrstrecke berechnet.
LH9	Es soll möglich sein die Tische sofort in ihrer Bewegung zu stoppen.
LH10	Es soll eine Klasse geben, die die Informationen über die angeschlossenen Tische und Controller beinhaltet.

2.2 Verwendete Tools

Programmiert wurde in Microsoft Visual Studio 2019 in C++ mit der Qt Version 5.15.2.

2.2.1 Visual Studio

Microsoft Visual Studio ist eine integrierte Entwicklungsumgebung (IDE) für verschiedene Programmierhochsprachen. Dies umfasst Visual Basic, C, C++, C# u.v.m.

2.2.2 C++

C++ ist eine ISO-genormte Programmiersprache, die im Jahre 1979 von Bjarne Stroustrup als Erweiterung von C entwickelt wurde [7]. Sie ist eine der weit verbreitetsten Programmiersprachen deren Besonderheit es ermöglicht laufzeiteffizient und systemnah, aber auch objektorientiert zu programmieren. Im Vergleich mit anderen populären Sprachen

wie Java oder Python kann C++ mehrere Basisklassen haben (Multiple Inheritance). Zur Anwendung kommt dieses Prinzip bei der Schnittstelle zwischen smartLab und dem Plugin. Ein Vorteil von C++ ist, dass die Programmiersprache nicht an einen Compiler gebunden ist, sondern von einer Umgebung auf eine andere übertragen werden kann. Für dieses Projekt wurde der Visual C++ Compiler verwendet.

2.2.3 Qt

Qt ist ein Anwendungsframework mit umfangreichen Bibliotheken zur Entwicklung von Softwareprogrammen und grafischen Benutzeroberflächen [8].
Gewählt wurde dieses Framework aufgrund der Architektur von smartLab. Qt kann für verschiedene Software- und Hardware-Plattformen wie Linux, Windows, macOS, Android oder eingebettete Systeme angewendet werden und unterstützt verschiedene Compiler, unter anderem auch den hier verwendeten Visual Studio C++ Compiler.
Der Qt-Designer bietet die Möglichkeit die grafische Benutzeroberfläche (GUI) grafisch (wie zum Beispiel bei Windows Forms), aber auch per Code zu programmieren. Die grafische Programmierung findet über den Designer statt und wird als .ui Datei angelegt. Aus dieser generiert der Compiler dann eine ui_classname.h Datei, welche den Backend-Code für die grafischen Elemente und deren Anordnung beinhaltet. Es kann allerdings auch ohne grafische Ansicht gearbeitet und die GUI direkt durch Code definiert werden. Das Design kann durch eine textbasierte Kodierung erzeugt werden.
Qt verwendet das Signal-Slot-Prinzip, um die ereignisgesteuerte Kommunikation zwischen Objekten herzustellen, wobei ein Objekt als Sender und ein anderes als Empfänger agiert. Dies hat den Vorteil, dass die Abhängigkeiten zwischen Klassen verringert werden. Die Variablentypen und Funktionen aus den Qt-Bibliotheken beginnen immer mit dem Buchstaben Q (z.B. QString, QVector, QPushButton).

2.2.4 Git/Gitlab

Git ist eine Open Source Software zur verteilten Versionsverwaltung von Dateien. Zur Entwicklung von Software in einem Team ist es wichtig, alle Änderungen nachverfolgbar bzw. auch reversibel zu machen [9]. Aus diesem Grund wurde für dieses Projekt zur Nachverfolgung git verwendet. Für die Anwendung wurde git auf GitLab eingerichtet.
GitLab ist ein Webanwendung, die einen Hosting Server für git anbietet. Über GitLab können zum Beispiel Code Reviews über den Browser visualisiert werden.

2.2.5 PI-Geräte-Bibliothek

Die verwendeten Controller und Tische wurden von dem Hersteller Physik Instrumente (PI) erworben, der seine eigene Bibliothek mit Funktionen zur Anwendung zur Verfügung stellt.
Verwendet wird die Bibliothek durch die Headerdatei PI_GCS2_DLL.h und die Einbindung der lib- und dll-Datei ins Projekt.

2.3 Architektur & Design

2.3.1 Projektbeschreibung

Wie schon erwähnt basiert das Projekt auf der Grundlage der Desktopanwendung smartLab, die verschiedene Plugins einbinden und verwenden kann.

Im Folgenden wird dieses Konzept näher erläutert.

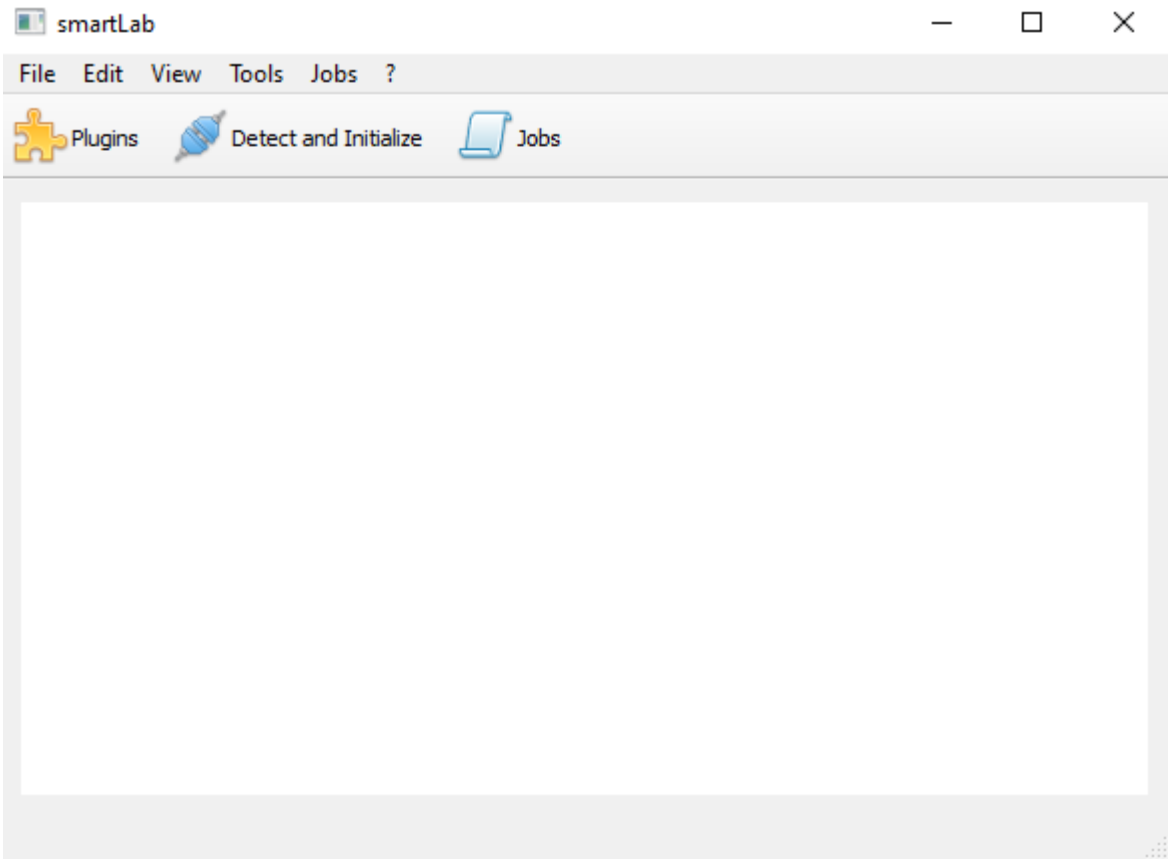

Abbildung 2: Grafische Oberfläche von smartLab ohne geladene Plugins

smartLab bietet eine grafische Benutzeroberfläche (siehe Abbildung 2) durch welche mehrere Plugins (Hardware) eingebunden und benutzt werden können. Die Plugins können dann über sogenannte „Jobs“ vordefinierte Aufgaben durchführen. Jobs sind smartLab-interne Anwendungen, die sich dadurch charakterisieren, dass durch sie von der Hauptsoftware auf die verschiedenen Plugins und ihre Interface-Funktionen zugegriffen werden kann. Ein Job besteht aus einer Qt-GUI-Datei (.ui), einer Header- und einer Cpp-Datei. Eingebunden werden die Job-Dateien über das IJob Interface (Job-Klasse erbt von IJob), durch welches dann die Job-Instanz im Job-Manager von smartLab erzeugt werden kann.
Die Plugin-Bibliotheken (.dll-Dateien) werden über die interfaces.h-Datei eingebunden. Diese Headerdatei enthält alle Interfaces, die bisher für smartLab entwickelt wurden. Sie muss im smartLab und im Plugin eingebunden sein und übereinstimmen. Ein Plugin-Interface muss von dem IHardware-Interface erben, deren Funktionen im Plugin implementiert werden müssen, damit das Plugin funktionsfähig wird.

Um das Plugin zu verwenden, werden diese Funktionen, z.B. die detect()- und initialize()-Funktion aus IHardware zur Erkennung und Initialisierung der Hardware, implementiert. Dies wird durch den Klick auf den Button „Detect and Initialize“ ausgelöst (Abbildung 2).
Des Weiteren können in der Interface-Datei Funktionen sowie Signal-Slot-Verhalten definiert werden. Zum Beispiel können Informationen zwischen Plugin und smartLab gesendet und empfangen oder Funktionen aus dem Plugin durch den Job ausgelöst werden.

Listing 1: Ausschnitt aus interfaces.h, IStage-Klasse

```
class IStage : public IHardware
{
    Q_OBJECT
public:
    virtual bool moveTo(double pos, int id = 0) = 0;
    virtual bool moveTo(double* pos, const char* axes) = 0;
    virtual bool moveSteps(double steps, int id = 0) = 0;
    virtual double* getPosition(const char* axes) = 0;
    virtual double* getPosition() = 0;
    virtual bool stop() = 0;

signals:
    void newPositionAvailable(QVector<qreal>* position, int id);
};
```

In Listing 1 sind die für dieses Plugin definierten Funktionen und ein Signal zu sehen. Da schon in vorhergehenden Projekten PI-Controller und Tische verwendet wurden, gibt es schon ein angelegtes Interface, welches teilweise *Legacy code,* wie beispielsweise die erste moveTo-Funktion, beinhaltet.
Die Definition der Interface-Methoden als rein virtuell (*pure virtual function*) bedeutet, dass im Plugin selbst diese Funktionen implementiert werden müssen.

2.3.2 Design Patterns

Factory bzw. Factory Pattern

Eine Factory ist eine Programmkomponente, meist eine Klasse, welche verantwortlich für die Erstellung von Objekten ist [10]. Durch die Factory wird es möglich, die Instanziierung von Objekten von der Verwendung zu trennen. Die factory-Klasse erlaubt es, dynamisch Speicherplatz für das neue Objekt zuordnen zu können. Dies ist eine notwendige Funktionalität für die Umsetzung der Anforderung LH1. Die Anwendung findet in der controllerFactory-Klasse statt, auf die im folgenden Kapitel näher eingegangen wird.

Zur Begriffsabgrenzung wird im Folgenden das Factory Design Pattern oder Factory Method beschrieben, da in der Literatur die Abgrenzung nicht immer konkret aufgeführt wird:
Das Factory Design Pattern ist ein *Design Pattern*, das die Factory verwendet, wobei die *Objektinstanziierung* allerdings auf eine *virtuelle*-Methode verlagert wird [10]. Hierbei entsteht dann eine sogenannte „abstract factory“ von der spezifische Factory Klassen abgeleitet werden können, die gemeinsame abstrakte Eigenschaften haben, deren konkrete Details sich aber unterscheiden.
In Falle des Plugins ist allerdings keine „abstract factory“ notwendig, weshalb nur die Factory Einsatz findet.

Einzelinstanz einer Klasse vs. Singleton Klasse

Die Lastenheftanforderung LH10 fordert die Erstellung einer einzelnen Instanz, welche alle Informationen zu den angeschlossenen Controllern und Tischen beinhaltet.
Hierbei wurde das sogenannte Singleton Design Pattern in Betracht gezogen, welches im Buch „Design Patterns: Elements of Reusable Object-Oriented Software“ von dem Autorenteam „The Gang of Four“ vorgestellt wurde [11].
Das Singleton Pattern ist ein Designkonzept, welches in der Coding Community sehr umstritten ist [12]. In der Kritik steht zum Beispiel, dass das Single Responsibility Prinzip (SRP) nicht eingehalten wird. Dieses besagt, dass jedes Modul, jede Klasse oder Funktion nur Verantwortung über einen Teil der Programmfunktionalität haben soll. Das Singleton Pattern verstößt gegen das SRP, weil es gleichzeitig zuständig für alle Funktionen innerhalb der Klasse ist, aber auch für dessen *Instanziierung* und Lebenszeit.
Die Eigenschaften des Singleton Patterns erschweren die Wartung und Änderung des Codes und erzeugen ungewollte Abhängigkeiten zwischen Klassen.
Aus diesem Grunde wurde eine Alternative gewählt, nämlich nur eine einzelne Instanz einer Klasse zu erstellen und zu verwenden. Zur Anwendung kommt dieses Prinzip in der Klasse allDetectedStages, die als Logger des Plugins fungiert. Die allDetectedStages-Instanz sammelt alle Informationen zu den Controllern und Stages und macht gezielt spezifische Informationen zugänglich, beispielsweise die aktuelle Position der Stages.

3. Ergebnisse

Im folgenden Kapitel werden die Entwicklungsschritte des Softwaremoduls aufgeführt. Angefangen mit den erarbeiteten Softwareanforderungen werden darauf basierend die Lösungsansätze und die Entwicklung eines Konzepts herausgearbeitet. Im zweiten Teil des Kapitels wird dann die konkrete Implementierung in den verschiedenen Klassen vorgestellt. Als Letztes werden die derzeitige Funktionalität, die Tests und Coverage Analysen, Abweichungen sowie Mängel in der Umsetzung der Anforderungen dargestellt.

3.1 Umsetzung

3.1.1 Softwareanforderungen

Folgende Softwareanforderungen konnten aus dem Lastenheft abgeleitet und konkretisiert werden:

Tabelle 2: Softwareanforderungen

Key	Anforderung	Verifiziert
SW1	Das Softwaremodul soll als Plugin in smartLab auf Windows 10 lauffähig sein.	LH2
SW2	Die Software soll als Dynamic Library (.dll) programmiert werden.	LH2
SW3	Das Plugin soll in C++ und Qt programmiert werden.	LH2
SW4	Das Plugin soll die PI Bibliothek einbinden.	LH1
SW5	Das Plugin soll mehrere Controller von PI einbinden können, welche in der GUI ausgewählt werden können.	LH1
SW6	Das Plugin soll mehrere Stages einbinden können, welche in der GUI auswählbar sind.	LH1, LH3
SW7	Jede Stage soll einzeln per Buttonklick in zwei Richtungen verfahrbar sein.	LH4
SW8	Die Fahrgeschwindigkeit soll für jede Stage einstellbar sein.	LH4
SW9	Es soll eine Stopp-Funktion für den Controller geben.	LH9
SW10	Jede Stage soll eine Option zum Verfahren mit einer individuellen/manuell anpassbaren Schrittweite haben.	LH4
SW11	Es soll eine Anzeige für jede Stage geben, die die aktuelle Position der Stage anzeigt.	LH3
SW12	Es soll eine Möglichkeit geben per Buttonklick zu den Randpositionen zu fahren.	LH6
SW13	Es soll eine Klasse geben, die alle Daten zu der angeschlossenen Hardware sammelt und abrufbar macht.	LH10
SW14	Es soll möglich sein, die Tische über ein vorgegebenes Muster zu verfahren.	LH7
SW15	Es soll eine Schnittstelle für die Übergabe von Datenpunkten für die abzufahrende Fläche geben.	LH8
SW16	Das Plugin soll eine möglichst effiziente Fahrstrecke für die Datenpunkte berechnen.	LH8, LH7
SW17	Das Timing der Fahrt soll bestimmbar sein, um die Lasertriggerung synchronisieren zu können.	LH5

Weiß = Allgemeine Anforderungen, Grün = Plugin-GUI Anforderungen, Gelb = Logger Klasse, Rot = Makro Anforderungen

Um zu überprüfen, ob die Deckung aller Lastenheftanforderungen durch die Softwareanforderungen gewährleistet ist, wurde eine Coverage Analyse durchgeführt.

Tabelle 3: Coverage Analyse, Überprüfung der Deckung der Lastenheftanforderungen durch Softwareanforderungen

Schlüssel	Lastenheftanforderung	Schlüssel	Softwareanforderung
		Coverage 100%	
LH1	Es soll ein Softwaremodul entwickelt werden, das mehrere Controller bedienen kann, welche jeweils mehrere Tische bedienen können.	SW4	Das Plugin soll die PI Bibliothek einbinden.
		SW5	Das Plugin soll mehrere Controller von PI einbinden können, welche man in der GUI auswählen kann.
		SW6	Das Plugin soll mehrere Stages einbinden können, welche in der GUI auswählbar sind.
LH2	Das Softwaremodul soll als Plugin für die existierende Hauptsoftware smartLab in C++ und Qt entwickelt werden.	SW1	Das Plugin soll als Plugin in smartLab auf Windows 10 lauffähig sein.
		SW2	Die Software soll als Dynamic Library (.dll) programmiert werden.
		SW3	Das Plugin soll in C++ und Qt programmiert werden.
LH3	Alle angeschlossenen Controller und Tische sollen in einem Userinterface angezeigt und bedienbar sein.	SW6	Das Plugin soll mehrere Stages einbinden können, welche in der GUI auswählbar sind.
		SW11	Es soll eine Anzeige für jede Stage geben, die die aktuelle Position der Stage anzeigt.
LH4	Die Tische sollen per Buttonklick in einer bestimmten Geschwindigkeit für eine bestimmte Strecke verfahrbar sein.	SW7	Jede Stage soll einzeln per Buttonklick in zwei Richtungen verfahrbar sein.
		SW8	Die Fahrgeschwindigkeit soll für jede Stage einstellbar sein.
		SW10	Jede Stage soll eine Option zum Verfahren mit einer Schrittweite haben, welche vorher eingegeben werden kann.
LH5	Das Timing der Fahrt soll bestimmbar sein, um das Triggern des Lasers synchronisieren zu können.	SW17	Das Timing der Fahrt soll bestimmbar sein, um die Lasertriggerung synchronisieren zu können.

LH6	Die Tische sollen per Buttonklick bis zur maximalen/minimalen Position verfahrbar sein.	SW12	Es soll eine Möglichkeit geben per Buttonklick zu den Randpositionen zu fahren.
LH7	Es soll eine Möglichkeit zum automatisierten Abfahren in einem bestimmten Fahrmuster (Makro) geben.	SW14	Es soll möglich sein die Tische über ein vorgegebenes Muster zu verfahren.
LH8	Es soll möglich sein über eine Schnittstelle ein bestimmtes Lasermuster zu übergeben anhand dessen das Plugin die Fahrstrecke berechnet.	SW15	Es soll eine Schnittstelle für die Übergabe von Datenpunkten für die abzufahrende Fläche geben.
		SW16	Das Plugin soll eine möglichst effiziente Fahrstrecke für die Datenpunkte berechnen.
LH9	Es soll möglich sein die Tische sofort in ihrer Bewegung zu stoppen.	SW9	Es soll eine Stop-Funktion für den Controller geben.
LH10	Es soll eine Klasse geben, die die Informationen über die angeschlossenen Tische und Controller beinhaltet.	SW13	Es soll eine Klasse geben, die alle Daten zu der angeschlossen Hardware sammelt und abrufbar macht.

Die rechte Seite (Softwareanforderungen) der Tabelle 3 verifiziert die linke Seite (Lastenheftanforderungen). Die Coverage Analyse zeigt, dass eine hundertprozentige Deckung der Lastenheftanforderungen durch die Softwareanforderungen gewährleistet ist.

3.1.2 Lösungsansatz

Aus den Anforderungen ergeben sich zunächst diese Fragestellungen:

- Wie kann eine generische Objektinstanziierung abhängig von der Benutzereingabe gewährleistet werden?
- Wie können die Informationen über die ausgewählte Hardware und deren Zustand am besten während der Laufzeit gespeichert und abrufbar gemacht werden?
- Wie kann die Makrofunktionalität optimal umgesetzt werden?

Erstellung der Objektinstanzen für Controller und Stages

Wie in den Anforderungen SW5 und SW6 (siehe Tabelle 2) beschrieben, sollen mehrere Controller und Stages eingebunden werden können (siehe Abbildung 3). Das heißt, es sollte eine Möglichkeit geben, die Objekte automatisiert instanziieren zu können, je nachdem wie viele Controller oder Stages in der GUI ausgewählt werden oder nicht.

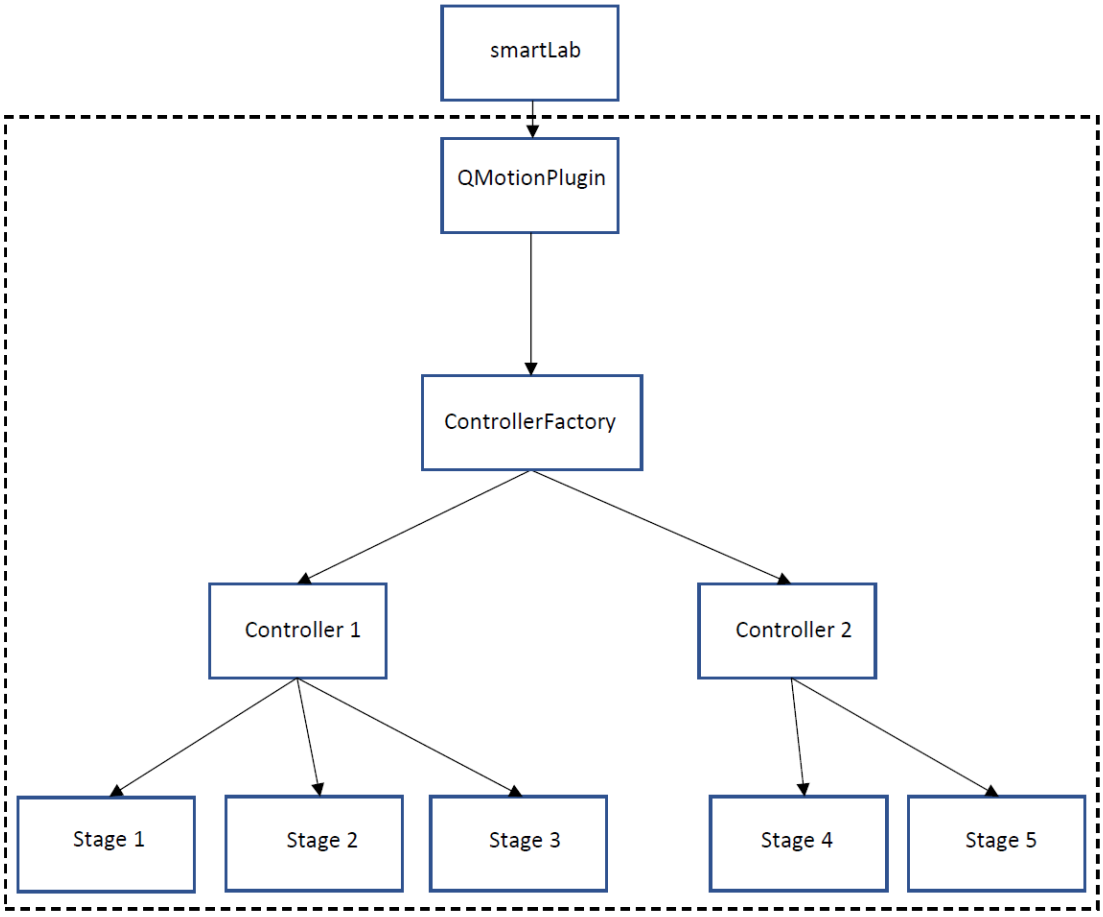

Abbildung 3: Schema der Anwendung des Plugins

Der Lösungsansatz für diese Fragestellung ist die Factory (siehe Kapitel 2.3.2), die die nötigen Instanzen der Controller erstellt. Die Controller selbst erstellen dann jeweils die nötigen in der GUI ausgewählten Stage-Instanzen.

Logger-Klasse mit allen Informationen über Hardware

Im Kapitel 2.3.2 wurde geschlussfolgert, dass die Nutzung einer einzelnen Instanz für die Problemstellung der Logger-Klasse die gewählte Lösung ist. Konkret soll diese Klasse Zugang zu den Informationen aller Hardwarekomponenten des Plugins haben und diese speichern bzw. abrufbar machen können. Daraus folgt die Entscheidung, dass die Instanziierung des Klassenobjekts am besten in der controllerFactory stattfindet. Die Klasse hat hier den nötigen Zugriff zu jedem Controllerobjekt und damit auch zu den Stage-Objekten. Weiterhin soll der Klasse nicht eine Kopie, sondern die Adresse des Objektes übergeben werden, sodass ein manuelles Updaten der Informationen nicht notwendig ist.

Umsetzung der Makros

In dem Plugin wird zunächst nur ein Fahrmuster im Rahmen dieser Arbeit angedacht, nämlich das Meandermuster. Hierbei stellte sich die Frage, inwieweit mit den gegebenen Funktionen aus der PI-Bibliothek eine geeignete Lösung erarbeitet werden kann.

Um das Fahren der Tische mit einer bestimmbaren Fahrtgeschwindigkeit zu programmieren und diese Bestimmbarkeit möglichst wenig durch die Kommunikationszeit zwischen Soft- und Hardware zu beeinflussen, wurde die Trajektorien-Funktion aus der PI-Bibliothek ausgewählt. Die Funktion kann durch eine Abfolge von Befehlen die Stages in einem Muster verfahren. Die Controller besitzen einen Buffer, also einen Zwischenspeicher, in dem Datenpunkte eingelesen werden können, die dann in einer eingestellten Geschwindigkeit (trajectory rate) abgefahren werden.

Für die Laserablation ist eine gleichmäßige und damit bestimmbare Geschwindigkeit notwendig. Somit muss zusätzlich erst auf die Zielgeschwindigkeit beschleunigt werden, bevor die Laserablation beginnen kann. Für eine möglichst reibungslose Anfahrt der Trajektorie wurde folgende Beschleunigungsrampe vor der eigentlichen Ablationsstrecke festgelegt:

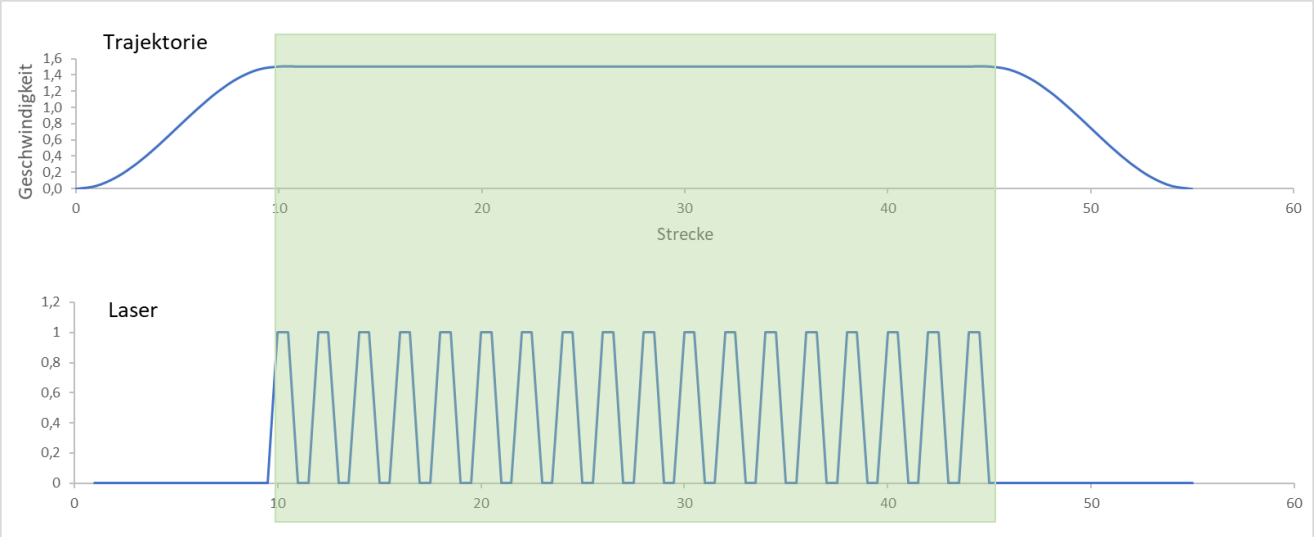

Abbildung 4: Gewünschte Geschwindigkeitskurve der Trajektorie (oben), Lasertriggersignale (unten)

Für die Formel der Beschleunigungsrampe wurde eine trigonometrische Ansatzfunktion gewählt, um die Anfahrt nicht zu abrupt zu gestalten. Ergebnis der Überlegungen, um die gewünschte Form der Beschleunigungsrampe (Abbildung 4, oberer Graph) zu erhalten, war folgende Formel:

$$f(x) = \frac{v}{2} \bullet 1 - \left(\cos\left(\frac{x\pi}{s}\right)\right)$$

v steht hierbei für die Geschwindigkeit, während das s für die Breite der Rampe und x für den Index des Trajektorienpunktes steht.

Eine Trajektorie besteht also aus der Fahrt von 0 bis zur Rampe, der Beschleunigungsrampe (steigende Flanke der oben gegebenen Funktion), der eigentlichen Laserablationsfahrt (Abbildung 4, grün markiert), der Entschleunigungsrampe (sinkende Flanke der oben gegebenen Funktion), der Fahrt bis zur nächsten Reihe, sowie aus derselben Sequenz aber in die entgegengesetzte Richtung. Wiederholt wird dies, bis die gegebenen Datenpunkte abgefahren sind. Der Buffer ist begrenzt, allerdings können nach und nach weitere Punkte nachgeschoben werden.

3.1.3 Konzeptionierung

Basierend auf Abbildung 3 wurden folgende Klassen erarbeitet:

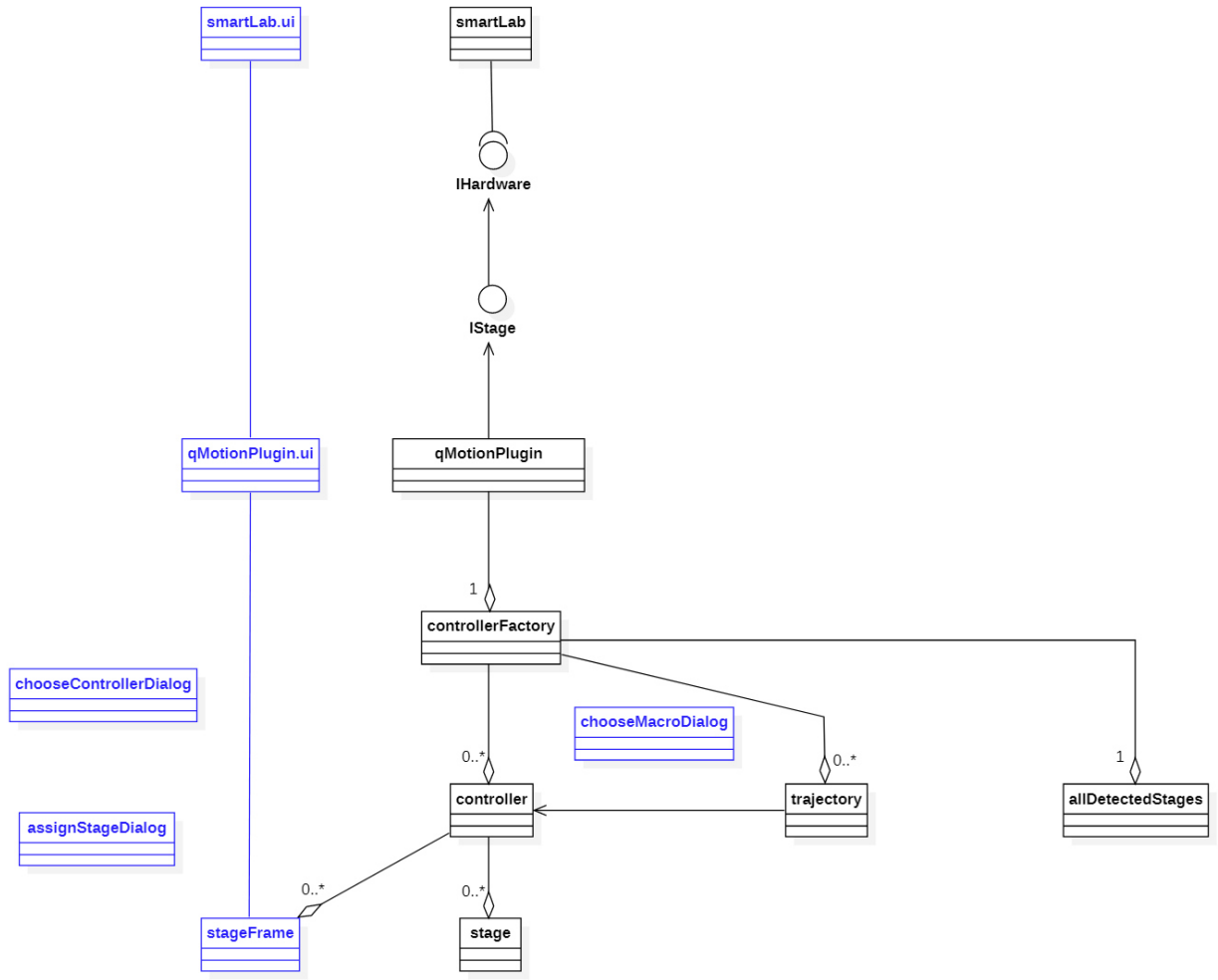

Abbildung 5: Klassenübersicht QMotionPlugin, GUI Elemente blau markiert

Die Software smartLab kann, wie in Kapitel 2.3.1 erläutert, über die Schnittstelle auf das Plugin zugreifen. Die Hauptklasse ist die qMotionPlugin-Klasse, in der das GUI-Element für die Einbindung in die smartLab GUI erstellt wird. In der qMotionPlugin-Klasse wird die controllerFactory-Instanz erstellt, die wiederum zuständig ist, für die Erstellung der Controller Instanzen, der Logger-Instanz (allDetectedStages) und der nötigen Trajektorieninstanzen.

Zur Erstellung der Controller-Instanzen und den sukzessiven Stage-Instanzen gibt es mehrere Dialoge, die per Buttonklick aufrufbar sind (chooseControllerDialog, assignStageDialog) und durch welche der Benutzer die gewünschten Controller und Stages auswählen kann. Entsprechend der Benutzerauswahl werden dann die Instanzen generiert.

Die Stage-Klasse beinhaltet keine Funktionalität, sondern ist ein reiner Informationsspeicher. Notwendig ist diese Klasse, da die Stage-Anzahl abhängig von der Benutzerauswahl ist und die Informationen automatisch strukturiert gespeichert werden sollen. Basierend auf der Stage-Auswahl werden in einer Controllerinstanz dann die korrespondierenden Stageframes erstellt. Diese sind die GUI-Elemente der Stages, welche dann während der Laufzeit in die qMotionPlugin-GUI eingebunden werden.

Die Logger-Klasse allDetectedStages sammelt die Informationen der Controller und der Stages für das Schreiben in eine Log-Datei oder zum Abruf von Informationen aus smartLab. Auch sollen für smartLab die Stage-IDs nicht pro Controller festgelegt werden, sondern fortlaufend als eine Liste sichtbar sein. Daher wird auch hier die globale Stage-ID in eine interne Stage-ID (pro Controller) zum Ansteuern über die PI-Bibliothek übersetzt.

Da die allDetectedStages-Klasse keine Kopie, sondern Pointer der Controller- und Stage-Instanzen speichert, steht immer die aktuelle Information zur Verfügung.

Die Controller-Klasse ist neben der trajectory-Klasse, die von der Controller-Klasse erbt, die einzige Klasse, die Befehle über die PI-Bibliothek ausführen kann. Die trajectory-Klasse hat die Funktionalität zum Ausführen der Trajektorien.

3.2 Implementierung

In Kapitel 3.1.3 Konzeptionierung wurde der allgemeine Zusammenhang der Klassen erläutert. Im Folgenden werden die einzelnen Klassen in Bezug auf die im Lösungsansatz gestellten drei Fragestellungen näher betrachtet.

3.2.1 Allgemeine Plugin-Klassen

Bei der allgemeinen Plugin-Struktur kommt der Lösungsansatz für die generische Objektinstanziierung zum Tragen. Dazu werden im Folgenden die einzelnen Klassen erläutert.

QMotionPlugin-Klasse

Die Hauptklasse des Plugins ist die qMotionPlugin-Klasse. Wie bereits erläutert ist für die Plugin-Einbindung notwendig, dass die Hauptklasse qMotionPlugin von IStage und somit von IHardware erbt.

Listing 2: Übersicht der QMotionPlugin-Klasse, qMotionPlugin.h

```
class QMOTIONPLUGIN_EXPORT QMotionPlugin : public IStage
{
    Q_OBJECT
        Q_PLUGIN_METADATA(IID IStage_iid)
        Q_INTERFACES(IStage)
        Q_INTERFACES(IHardware)

public:
    QMotionPlugin();
    ~QMotionPlugin();

    virtual bool detect();
    virtual bool initialize();
    virtual QDockWidget* getView();
    virtual bool moveTo(double pos, int axis);
    virtual bool moveTo(double* pos, const char* stages);
    virtual bool moveSteps(double steps, int axis) { return false; }
    virtual double* getPosition();
    virtual double* getPosition(const char* globalStageID);
    virtual bool stop();
    virtual bool release();
    virtual IOutput* getOutput();
    virtual QMap<QString, QString> getSaveValueInformation();
    virtual void setLoadValueInformation(QMap<QString, QString> map);
    virtual void showSettingsWindow();

private:
    Ui::Dock ui;
    QDockWidget* dock;
    controllerFactory* manager;
    QString className;
    void _connectSignals();

public slots:
    bool choose();
    void initGUI(int controllerID);
    void sendSignalFinished();

signals:
    void signal_Moved();
    void signal_ConnectionClosed();
    void signal_Stopped();
};
```

In Listing 2 ist die Definition der *virtual*-Methoden aus der interfaces.h für die Implementierung zu sehen. Die Methoden aus dem IStage-Interface (pluginspezifische Methoden) wurden grün und die aus dem IHardware-Interface (allgemeine Methoden für die Hardwareerkennung und -initialisierung) orange markiert. Das IHardware-Interface hat außerdem noch einige *flags,* die gesetzt werden können. Beispielsweise wird durch die setDetected(true), die Plugin-Hardware als erkannt markiert.

Listing 3: qMotionPlugin.cpp, Konstruktor und Destruktor

```
#include "qMotionPlugin.h"

QMotionPlugin::QMotionPlugin()
{
    setName("Q-Motion Plugin");
    setDetectable(true);
    setDetected(false);
    setInitialized(false);
    setOutput(false);
    setType(STAGE);
    xmlName = "Q-Motion";
    className = "QMotionPlugin";

    dock = new QDockWidget();
    manager = new controllerFactory();
    ui.setupUi(dock);
    _connectSignals();
}
QMotionPlugin::~QMotionPlugin()
{
    delete manager;
}
```

Listing 3 zeigt den Konstruktor und den Destruktor der QMotionPlugin-Klasse. Hier werden alle nötigen Variablen (z.B. für die GUI-Anzeige) und flags gesetzt, die aus dem IHardware-interface stammen. Außerdem wird das dock-Objekt und das manager-Objekt instanziiert. Das dock-Objekt ist ein Pointer vom Typ QDockWidget, der die GUI-Elemente in einem Fenster bzw. in einem sogenannten dock in die smartLab-GUI integrieren kann.
Das manager-Objekt ist ein Pointer vom Typ controllerFactory und wird im nächsten Kapitel näher erklärt. Da es sich bei dem manager-Objekt um einen nicht-Qt Pointer handelt, wird dieser nicht automatisch beim Zerstören des qMotionPlugin-Objektes gelöscht, weshalb er im Destruktor explizit nochmal zerstört wird. Qt-Pointer werden beim Destruieren der Instanz automatisch zerstört, da Qt die Funktionalität besitzt, dass beim Zerstören des *Parent* auch jedes *Child*-Objekt zerstört wird.

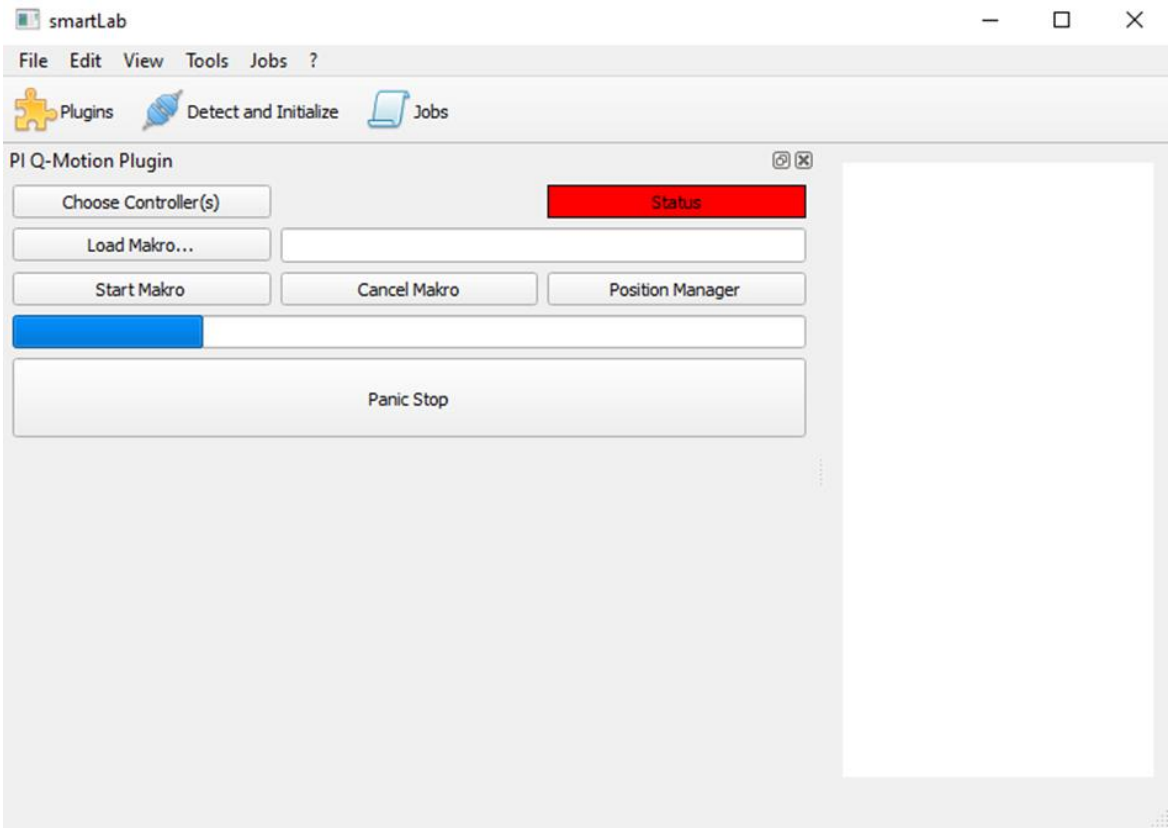

Abbildung 6: QMotionPlugin-GUI

Die setupUI-Funktion ist eine Funktion, die sich in der durch Qt erstellten GUI-Datei (ui_qMotionPlugin.h) befindet. Hier wird die primäre, grafische Oberfläche des Plugins definiert (siehe Abbildung 6).

Tabelle 4: Signal-Slot Verbindungen qMotionPlugin Klasse

Sender	Signal	Empfänger	Slot
ui.pushButton_chooseController	released()	this	choose()
ui.pushButton_loadMacro	released()	manager	loadMacro()
ui.pushButton_startMacro	released()	manager	startMacro()
ui.pushButton_cancelMacro	released()	manager	cancelMacro()
manager	signal_controllerReady()	this	initGUI()
manager	signal_allStagesMoved()	this	sendSignalFinished()
manager	signal_controllersStopped()	this	sendSignalFinished()
manager	signal_connectionClosed()	this	sendSignalFinished()

Die _connectSignals()-Funktion stellt alle Signal-Slot-Verknüpfungen her (siehe Tabelle 4), die in der Klasse verwendet werden. Beispielsweise werden hier auch die Buttons der GUI mit der entsprechenden Funktion verknüpft, die der Button auslösen soll (siehe Beispiel Listing 4).

Listing 4: qMotionPlugin.cpp, Beispiel Signal-Slot-Verbindung

```
connect(ui.pushButton_chooseController, &QPushButton::released, this,
&QMotionPlugin::choose);
```

Der QPushButton pushButton_chooseController löst die QMotionPlugin-Funktion choose() aus. Diese Funktion (Listing 5) ruft aus dem controllerFactory-Objekt manager die Funktion connectToControllers() auf, welche verantwortlich ist für die Benutzerabfrage der gewählten Controller über den Dialog chooseControllerDialog und die anschließende Generierung der controller Objekte.

Listing 5: qMotionPlugin.cpp, choose()-Funktion

```
bool QMotionPlugin::choose()
{
    // Connect controller
    if (!(manager->connectToControllers()))
    {
        qDebug("PI PLUGIN Error connecting to controllers.");
        return false;
    }
    return true;
}
```

Wie schon in Kapitel 2.3.1 beschrieben, wird die Funktion detect() (und dann initialize()) aufgerufen, sobald der „Detect and Initialize“-Button auf der smartLab-GUI gedrückt wird. In Listing 6 ist zu sehen, dass in der detect()-Funktion über das controllerFactory-Objekt die Anzahl der angeschlossenen Controller abgefragt wird. Soweit Controller angeschlossen sind, wird die isDetected-Flag auf true gesetzt. In der initialize()-Funktion wird der isInitialized-Flag auf true gesetzt und in der release()-Funktion werden die Verbindungen zum Controller geschlossen.

Listing 6: qMotionPlugin.cpp, Ausschnitt Funktionsimplementierungen der IHardware virtual Funktionen

```
bool QMotionPlugin::detect()
{
    long numberOfControllers = manager->getNumberOfControllers();
    if (numberOfControllers < 1)
    {
        qDebug("PI PLUGIN Error: No Controllers were detected.");
        this->setDetected(false);
        return false;
    }
    this->setDetected(true);
    return this->isDetected();
}
bool QMotionPlugin::initialize()
{
    qDebug("PI PLUGIN: Plugin initializing...");
    this->bIsInitialized = true;
    return this->isInitialized();
}
bool QMotionPlugin::release()
{
    if (!manager->closeConnection())
    {
        return false;
    }
    return true;
}
```

Die weiteren IHardware-Funktionen sind leer implementiert, da sie als *pure virtual function* definiert sind und implementiert werden müssen. In diesem Plugin wird die Funktionalität aber nicht verwendet.

Listing 7: qMotionPlugin.cpp, Ausschnitt Funktionsimplementierung der IStage virtual Funktionen

```
bool QMotionPlugin::moveTo(double* pos, const char* stages)
{
    if (!manager->moveTo(pos, stages))
    {
        return false;
    }
    return true;
}
double* QMotionPlugin::getPosition(const char* globalStageID)
{
    double* pos = manager->getPosition(globalStageID);
    if (pos != nullptr)
        return pos;
    else
    {
        qDebug("PI PLUGIN Error getting position from manager.");
        return nullptr;
    }
}
bool QMotionPlugin::stop()
{
    if (!manager->stopControllers())
    {
        return false;
    }
    return true;
}
```

Des Weiteren gibt es die Implementierung der IStage-Funktionen (Listing 7). Die moveTo()-Funktion erlaubt es einem Job Tische zu einer Position zu verfahren. Es können hier auch gleichzeitig mehrere Stages angegeben werden, da die Übergabe durch Pointer erfolgt.
Die Position von Stages lässt sich über getPosition() abfragen. Hier wird in der controllerFactory auf die allDetectedStages-Klasse zugegriffen. Es gibt noch eine zusätzliche moveTo()-Funktion, sowie getPosition()-Funktion, die aber leer implementiert sind und *Legacy Code* darstellen.
Um smartLab zu signalisieren, dass die moveTo-Funktion erfolgreich war, die Stages gestoppt wurden oder die Controllerverbindung geschlossen wurde, wird der sendSignalFinished()-Slot ausgelöst (vgl. Tabelle 4). Dieser Slot wiederum emittiert abhängig vom Sendernamen das entsprechende Signal für smartLab (Listing 8).

Listing 8: qMotionPlugin.cpp, sendSignalFinished()-Funktion

```
void QMotionPlugin::sendSignalFinished()
{
      QMetaMethod senderMethod = sender()->metaObject()-
      >method(senderSignalIndex());
      QString senderName = QString(senderMethod.name());
      if (senderName == "signal_allStagesMoved")
      {
            emit signal_Moved();
      }
      if (senderName == "signal_controllersStopped")
      {
            emit signal_Stopped();
      }
      if (senderName == "signal_connectionClosed")
      {
            emit signal_ConnectionClosed();
      }
}
```

Sobald die Controller ausgewählt wurden, und die weiteren Schritte, die über die choose()-Funktion ausgelöst werden abgeschlossen sind, werden die Controllerinstanzen in die GUI geladen. Dies geschieht durch das Signal signal_controllerReady (vgl. Tabelle 4), das den Slot initGUI() aufruft (Listing 9, nächste Seite).

Listing 9: qMotionPlugin.cpp, initGUI()-Funktion

```
void QMotionPlugin::initGUI(int controllerID)
{
    ui.mainVerticalLayout->addWidget(manager->c[controllerID]->getWidget(), 0,
    Qt::AlignTop);
}
```

In diesem Slot werden die GUI-Elemente der Controller als Widgets (orange markiert) in die Plugin-GUI eingefügt (siehe Abbildung 7).

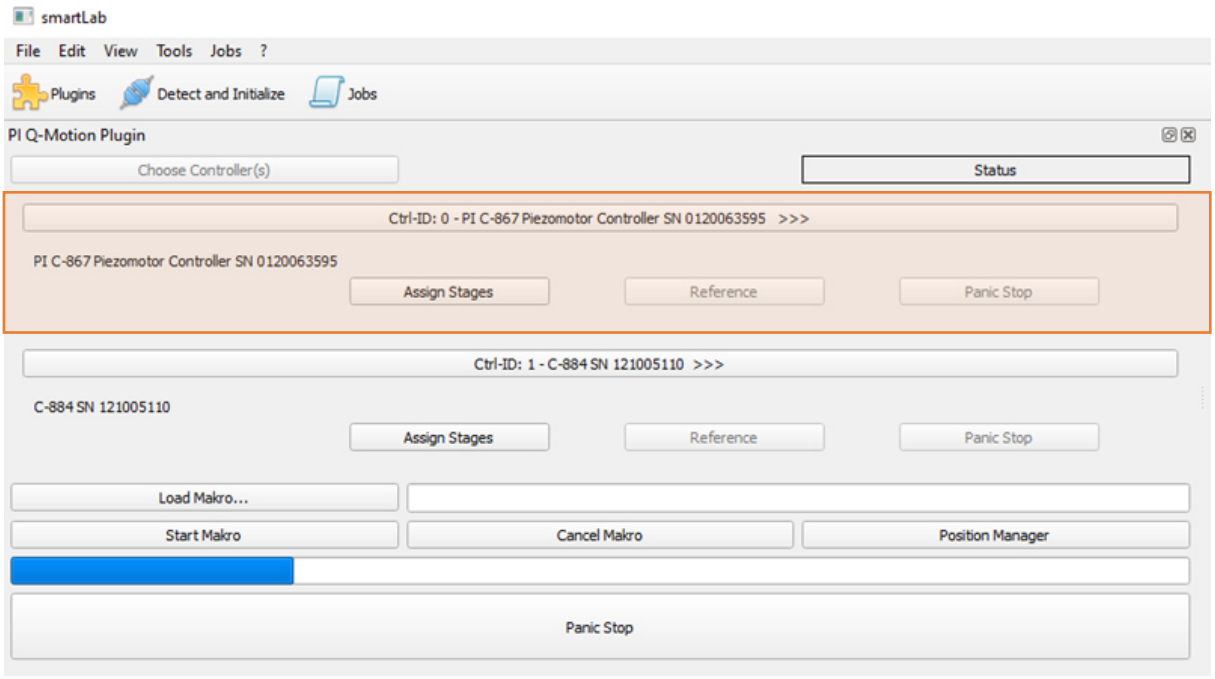

Abbildung 7: In die Plugin-GUI geladene Controller-Elemente, einzelnes Widget orange markiert

ControllerFactory-Klasse – Implementierung des Lösungsansatzes zur dynamischen Generierung der Controller-Objekte

Die controllerFactory wird als Objekt „manager“ in der Hauptklasse qMotionPlugin instanziiert und ist zuständig für die Generierung der controller-Objekte, der trajectory-Objekte und dem allDetectedStages-Objekt.
Die controller-Objekte werden für die ausgewählten Controller erstellt und bestehen so lange wie die controllerFactory existiert. Zusätzlich werden auch temporäre bzw. lokale controller-Objekte erstellt.
Ein Beispiel der lokalen Instanziierung ist die Bestätigung der Erkennung der Hardware, um smartLab ein erfolgreiches Detektieren zurückmelden zu können, indem in der detect()-Funktion der isDetected-Flag auf true gesetzt wird (Listing 6).

Bei der detect()-Funktion wird im manager-Objekt die Funktion getNumberofControllers() aufgerufen, die wiederum ein lokales controller-Objekt erstellt. Dieses hat Zugriff auf die PI-Bibliothek und somit auf den Controller selbst hat.
Wie in Listing 10 zu sehen kann durch den Befehl PI_EnumerateUSB der PI Bibliothek die Anzahl und die Namen der angeschlossenen Controller abgerufen werden.

Listing 10: controller.cpp, Ausschnitt _getControllers()-Funktion

```
bool controller::_getControllers()
{
    char names[128];
    _numberOfControllers = PI_EnumerateUSB(names, 128, "");
    _names = *_makeControllerList(names);
    return true;
}
```

Als erster Schritt nach dem Einbinden des Plugins müssen die zu nutzenden Controller ausgewählt und für die Nutzung aktiviert werden. Dieser Prozess startet durch das Drücken des „Choose Controller(s)“-PushButtons (siehe Abbildung 6) und der dadurch ausgelösten choose()-Funktion. Diese löst die Funktion connectToControllers() des manager-Objekts aus (Listing 11).

Listing 11: controllerFactory.cpp, Ausschnitt connectToControllers()-Funktion

```
bool controllerFactory::connectToControllers()
{
    // START - open choose controller dialog
       chooseControllerDialog* pChooseControllerDialog = new
       chooseControllerDialog();
       QStringList chosenControllers = pChooseControllerDialog-
       >runDialog(_getControllerNames());
       if (!chosenControllers.length()) {
         qDebug("PI PLUGIN Error on controller list!");
         return false;
       }
    // END - open choose controller dialog
       _controllerNum = chosenControllers.length();
       if (_controllerNum > 0)
       {
         _createController();
         // connect controller
         for (int i = 0; i < _controllerNum; i++)
         {
             emit c[i]->signal_connectController(chosenControllers[i]);
         }
       }
       else
       {
         qDebug("PI PLUGIN Error: No controller chosen.");
         return false;
       }
       return true;
}
```

Als Erstes wird ein Dialog-Objekt erstellt und aufgerufen, in welchem der Benutzer aus einer Liste von Controllern die gewünschten Controller auswählen kann (Listing 11, orange markiert; Abbildung 8). Die _getControllerNames()-Funktion ruft diese Liste über die controller-Funktion_getControllers() ab (Listing 10).

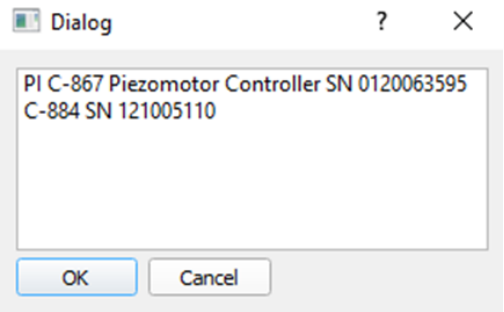

Abbildung 8: chooseControllerDialog mit geladener Controller-Liste

Nachdem die Controller ausgewählt wurden, können nun die controller-Objekte erstellt werden, die während der Laufzeit bestehen bleiben. Dies geschieht durch die Funktion _createControllers() (Listing 11, grün markiert, und Listing 12).

Listing 12: controllerFactory.cpp, Ausschnitt _createController()-Funktion

```
void controllerFactory::_createController()
{
    if (_controllerNum >= 0)
    {
      c = new controller * [_controllerNum - 1];
      for (int i = 0; i < _controllerNum; i++)
      {
            c[i] = new controller();
       }
    }
}
```

In der Klasse wurde ein globales Array von Pointern definiert. Der Vorteil eines globalen Pointer-Arrays ist es, dass es dynamisch deklariert werden kann. Dies führt dazu, dass abhängig von der Anzahl der Controller die Größe des Arrays bestimmt werden kann (Listing 12). Jeder Eintrag des Arrays beinhaltet einen Pointer zu einem neuen Controller-Objekt, der aufgrund der globalen Definition für die gesamte Laufzeit bestehen bleibt.
Um das sofortige Funktionieren der Methoden sicherzustellen und einen redundanten Code zu vermeiden, werden an dieser Stelle auch (wie im Beispiel in Listing 4) die Signal-Slot Verbindungen zwischen controllerFactory und den Controller-Objekten hergestellt (Tabelle 5).

Tabelle 5: Signal-Slot Verbindungen der controllerFactory, c[i] ist ein Eintrag aus dem Array von Pointern

Sender	Signal	Empfänger	Slot
c[i]	signal_connectController()	c[i]	connectToController()
c[i]	signal_assignStages()	c[i]	assignStages()
c[i]	signal_closeConnection()	c[i]	closeConnection()
c[i]	signal_stopController()	c[i]	stopController()
c[i]	signal_moveToPosition()	c[i]	moveToPosition()
c[i]	signal_connectionClosed()	this	_slot_signalReceived()
c[i]	signal_isStopped()	this	_slot_signalReceived()
c[i]	signal_stagesMoved()	this	_slot_signalReceived()
c[i]	signal_controllerConnected()	this	_slot_GuiReady()
c[i]	signal_stagesAssigned()	this	setLogData()

Controller-Klasse

Nach dem Erstellen der Controller-Objekte wird nun das erste Signal pro Controller emittiert (Listing 11, gelb markiert), um die Controller-interne Funktion connectToController() auszuführen (Listing 13). Diese Funktion verbindet die vom Benutzer ausgewählten Controller und ermöglicht deren Bedienung.

Listing 13: controller.cpp, Ausschnitt connectToController()-Funktion

```
void controller::connectToController(QString controllerName)
{
    bool connected = false;
    QByteArray ba_des = controllerName.toLatin1();
    _name = controllerName;
    sprintf( description, ba_des.data());
    _connectUSB();
    if (_checkConnection())
        connected = true;
    //check configuration of all stages
    if (!_getStageIndex()) {
        qDebug( "PI PLUGIN Error on stages query");
        return;
    }
    _initWidget();
    if (connected)
        emit signal_controllerConnected(_controllerID);
}
```

Die Methode _connectUSB() (Listing 13, blau markiert) ruft eine Funktion der PI-Bibliothek auf, um den Controller zu Nutzung über USB-Verbindung bereitzustellen. _checkConnection() stellt sicher, dass die Verbindung erfolgreich war. Die _getStageIndex()-Funktion lädt die Indizes aller angeschlossenen Stages aus dem Controller.

Als Letzter Schritt der Methode wird das Signal signal_controllerConnected() gesendet, welches in der controllerFactory-Klasse empfangen (siehe Tabelle 5) und schließlich an die qMotionPlugin-Klasse weitergeleitet wird (vgl. Tabelle 4). Das in der initWidget()-Funktion gebaute Controller-GUI-Element wird dann als Widget in die Plugin-GUI eingebaut (Abbildung 7). Der Nutzer kann nun die Stages für jeden Controller auswählen, indem bei jedem Controller der „Assign Stages“-Button (Abbildung 7) gedrückt wird. Dieser Button ruft den entsprechenden Stage-Auswahldialog für den Controller auf. Im Stage-Dialog (Abbildung 9 und Listing 14 gelb markiert) werden auf der linken Seite die erkannten Stages und auf der rechten die ausgewählten sowie dem Index zugewiesenen Stages aufgelistet.

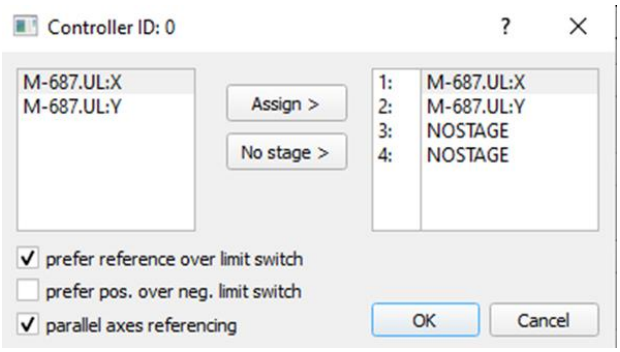

Abbildung 9: assignStageDialog, Auswählen der erkannten Stages für den Controller mit der ID = 0

Nach dem Auswählen werden die zugewiesenen Stages in einer Textdatei gespeichert, um dem Benutzer die Bedienung beim nächsten Aufruf zu erleichtern (siehe Listing 14, rot markiert). Die Stage-Auswahl wird dann direkt aus der Textdatei geladen und muss nicht erst zugewiesen werden.

Listing 14: controller.cpp, Ausschnitt assignStages()-Funktion

```
void controller::assignStages()
{
    // getting stage names
    if (!_checkAssignedStages()){return;}
    if (preChoiceStages.isEmpty())
    {
        _getPreChoice();
    }
    // START - open assign stage dialog
    assignStageDialog* pAssignStageDialog = new assignStageDialog();
      QStringList chosenStagesList = pAssignStageDialog-
      >runDialog(_getStagesFromDB(), preChoiceStages, _controllerID);
    // set prechoice
    preChoiceStages = chosenStagesList;
    _setPreChoice();
    QString chosenStages = _makeStringFromStringList(chosenStagesList);
    _convertDataForController(chosenStages);
    // END - open assign stage dialog
    // create instances of stages
    if (stages.isEmpty())
        _createStages();
    else
        _renameStages();
    //assign stages
    if (!_assignStages()){return;}
    //check all axes again to check assigned axes
    if (!_checkAssignedStages()){
        qDebug("PI PLUGIN Error on name query!");
        return;
    }
    //check configured axes
    if (!_getStageIndex()){
        qDebug("PI PLUGIN Checking axes failed!");
        return;
    }
    emit signal_stagesAssigned(_controllerID);
    _slot_addStagesToGui();
    posTimer->start(1000);
}
```

Sobald die Stages über den Stage-Dialog (Listing 14, gelb markiert) ausgewählt und in die Textdatei gespeichert wurden (rot markiert), werden nun die Stage-Instanzen für die Speicherung der Stage-Informationen erstellt.
Der grün markierte Code in Listing 14 setzt die dynamische Instanziierung abhängig von der Benutzereingabe um. Hier werden die ausgewählten Stages auf die gleiche Weise wie die Controllerobjekte in Listing 12 als Objekte der Klasse Stage erstellt.
Um die Nutzung der Stages über den Controller zu ermöglichen, werden nach der Instanziierung die Stages aktiviert (blau markiert). Die Zuweisung wird überprüft und dann wird das Signal signal_stagesAssigned() (Tabelle 5) gesendet, das die Übernahme der Stage-Objekte in die Logger-Instanz auslöst.

Als nächstes wird nun die Funktionalität der Stages für die Nutzung über die GUI vorbereitet. In der Funktion _slot_addStagesToGui() werden die Stageframes gebaut und in die Controller-GUI Elemente eingebunden (Abbildung 10).

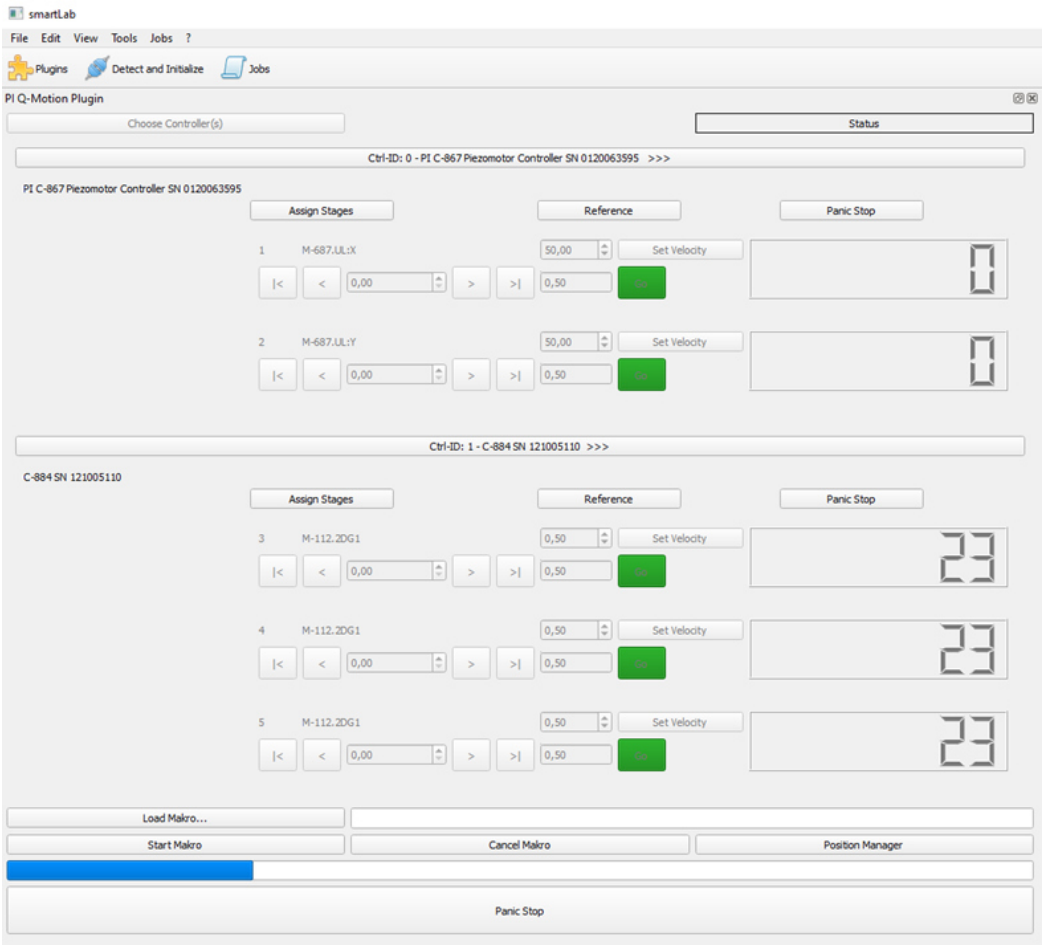

Abbildung 10: Plugin-GUI mit eingebundenen Stages

Schließlich wird, um immer die aktuelle Position der Stages anzeigen zu können, ein Timer (posTimer) gestartet, der in Intervallen von 1000 ms die Funktion _updateGui() aufruft (Listing 15). Da die Positionsanzeige nicht auf die Millisekunde genau sein muss, reicht es aus, die GUI im Sekundentakt zu aktualisieren.

Listing 15: controller.cpp, _updateGui()-Funktion

```
void controller::_updateGui()
{
    _getPosition();
    for (int i = 0; i < _stageNum; i++)
    {
        _currentPos[i] = round(_currentPos[i], 2);
        if (_currentPos[i] == stages[i]->get_currentPosition())
        {
            _selectedStage = i;
            for (int i = 0; i < _stageNum; i++)
            {
                stageFrames[i]->stageFrameWidget->setEnabled(true);
            }
            _setGuiStatusReferenced();
        }
        stageFrames[i]->lcdNumber_position->display(_currentPos[i]);
    }
}
```

Der Position-Timer läuft ab dem Startzeitpunkt während der gesamten Laufzeit des Softwaremoduls. Die aufgerufene Funktion (Listing 15) fragt die aktuellen Positionen der Stages ab und speichert sie in einem Array.
Die Stage-Instanz beinhaltet die erste aktuelle Position, sobald die Referenzfahrt abgeschlossen wurde. Sobald eine Position über die Stage-Instanz abrufbar ist, werden auch die Stageframes in der GUI aktiviert und zur Verfügung gestellt. Vorher sind die GUI-Elemente noch ausgegraut (Abbildung 10). Die Positionsangabe wird nach jeder Bewegung in der Stage-Instanz aktualisiert.
Nach der Auswahl der Stages müssen sie zur Ermittlung der Position referenziert werden. Diese Funktion wird durch den Button „Reference“ ausgelöst (siehe _referenceStages()-Funktion, Listing 16).

In der Reference-Funktion wird als Erstes der Servomotor der Stages eingeschaltet und überprüft. Danach wird der Referenzierungstyp der Stages (Referenzschalter oder Endschalter) abgefragt. Abhängig vom abgefragten Typ wird dann ein char-Array mit den entsprechenden Übergabewerten im richtigen Format für den Referenzierungsbefehl der PI-Bibliothek gebaut. Danach wird die Referenzfahrt nach dem entsprechenden Typ (_referenceWithSwitch() oder _referenceWithLimit()) durchgeführt.

Listing 16: controller.cpp, _referenceStages()-Funktion

```
void controller::_referenceStages()
{
    _setGuiStatusReferencing();
    //servo
    _setServo();
    _checkServo();

    //check reference type of each axis
    _checkReferenceType();

     //build strings for assigned axes with reference switch, limit switch and
     without switch
    _buildReferenceString();

    //reference with reference switch
    if (ref_ctr > 0) {
        _isReferenced(REFERENCE);
        _referenceWithSwitch();
    }
    //reference with limit switch
    if (limit_ctr > 0) {
        _isReferenced(LIMIT);
        _referenceWithLimit();
    }
    referenceTimer->start(1000);
}
```

Als Letztes wird in der _referenceStages()-Funktion ein Timer gestartet, der in regelmäßigen Intervallen (1000 ms) den Slot _updateReferencingStatus() (Listing 17) aufruft. Auch hier ist der Sekundentakt ausreichend, da hier lediglich überprüft wird, ob die Referenzfahrt noch läuft.

Listing 17: controller.cpp, _updateReferencingStatus()-Funktion

```
void controller::_updateReferencingStatus()
{
    allStagesMoved = 0;
    _areStagesMoving();
    for (int i = 0; i < _stageNum; i++)
    {
        if (!_ReferencingFlags[i])
        {
            allStagesMoved++;
        }
    }
    if (allStagesMoved == _stageNum)
    {
        referenceTimer->stop();
        _isReferenced(REF_AND_LIM);
        for (int i = 0; i < _stageNum; i++)
        {
            if (_bRefFlags[i])
            {
                _selectedStage = i;
                _setGuiStatusReferenced();
            }
            else
            {
                _selectedStage = i;
                _setGuiStatusUnreferenced();
            }
        }
        //switch on servos
        _setServo();
        _checkServo();
        //set referenced axes to home position
        _goHome();
    }
}
```

Die _updateReferencingStatus()-Funktion kontrolliert, ob sich die Stages in Bewegung sind. Sobald sich alle Stages nicht mehr bewegen, wird der Reference-Timer gestoppt und überprüft, ob die Stages ordnungsgemäß referenziert wurden (isReferenced()-Funktion). Ist die Referenzierung erfolgreich, wird der Status der Referenzierung in der GUI aktualisiert. Danach wird der Servomotor nochmal neu aktiviert.

Listing 18: controller.cpp, _goHome()-Funktion

```
bool controller::_goHome()
{
    _setGuiStatusHoming();
    if (!PI_GOH(_controllerID, NULL)) {
        return false;
    }
    for (int i = 0; i < _stageNum; i++)
    {
        stages[i]->set_currentPosition(round(0.00000, 2));
    }
    return true;
}
```

Als letzter Schritt der Referenzierung werden die Stages auf ihre im Controller gespeicherte Home-Position gefahren (Listing 18). Nach diesem Schritt wird die Home-Position als erste aktuelle Position in den Stage-Instanzen gespeichert. Somit sind die Stageframes dann über den Position-Timer durch die _updateGui()-Funktion aktivierbar(Listing 15).

GUI-Funktionalität des Stageframes:

Abbildung 11 zeigt eine referenzierte Stage mit der ID 1. Zu erkennen sind die verschiedenen Buttons zur manuellen Bedienung der Stage über die Plugin-GUI. Zur Vereinfachung wurden die Richtungen als links und rechts definiert. Die eigentliche Orientierung ist abhängig von der Hardware-Positionierung der Stage.

Abbildung 11: Stageframe der Stage mit ID = 1

Folgende Funktionalitäten sind gegeben:

- Bewegung der Stage nach rechts um das eingegebene Inkrement
- Bewegung der Stage nach links um das eingegebene Inkrement
- Bewegung der Stage an das linke Ende
- Bewegung der Stage an das rechte Ende
- Bewegung der Stage an die eingegebene Position (Eingabefeld und Go-Button)
- Ändern der Bewegungsgeschwindigkeit der Stage (Eingabefeld und Set Velocity-Button)
- Anzeige der aktuellen Position (Listing 15)

In Listing 19 ist die _moveRight()-Funktion als Beispiel gezeigt. Die anderen Bewegungsfunktionen sind vergleichbar aufgebaut. Nach der Abfrage der aktuellen Position wird abhängig von der ID für diese Stage die Funktion movePosition() aufgerufen. Die interne Stage-ID wird aus der angezeigten globalen ID ermittelt (für mehr Informationen siehe nächstes Kapitel). Die Arrays, die für diese Funktion notwendig sind, werden geleert und vor Beginn der Fahrt mit der gewünschten Zielstage und -position gefüllt (siehe Listing 19, gelb markiert).

Listing 19: controller.cpp, _moveRight()-Funktion

```
bool controller::_moveRight()
{
	getPosition();
	QPushButton* btn = (QPushButton*)sender();
	int stageButtonIndex = btn->objectName().split("_").last().toInt()-1;
	getSelectedStageIndex(stageButtonIndex);
	double position = _currentPos[_selectedStage];
	double increment = stageFrames[_selectedStage]->doubleSpinBox_stepSize-
	>value();
	setCharEmpty();
	_currentStages[0] = (_selectedStage + 1) + '0';
	_currentPos[0] = position + increment;
	stages[_selectedStage]->set_currentPosition(round(_currentPos[0],2));
	_setGuiStatusBusy();
	if (!_movePosition())
	{
		qDebug("PI PLUGIN Error moving to defined position.");
		return false;
	}
	return true;
}
```

3.2.2 Logger-Klasse

Die in den Anforderung SW14 beschriebene Logger-Klasse ist die allDetectedStages-Klasse (Listing 20).

Sie beinhaltet folgende Funktionalitäten:

- Speichern von Pointern zu den Controller-Instanzen in einen QVector._controllers
- Speichern von Pointern zu den Stage-Instanzen in einen QVector _stages
- Speichern der Controller- und Stages-IDs in einen QVector _controllerInfo
- Speichern und Abrufen der internen Stage-Indices anhand der globalen Stage-ID
- Ausgeben der aktuellen Position einer Stage
- Schreiben der Controller und Stages in eine Logdatei
- Speichern der Controller-Namen in einer StringList

Listing 20: allDetectedStages.h

```
class allDetectedStages : public QObject
{
    Q_OBJECT
public:
    allDetectedStages();
    ~allDetectedStages();

    void setStages(controller* c);
    void setControllers(controller* c);
    void setNames(QStringList names);
    std::tuple<int, int> getStageIndex(int globalIndex);
    double getCurrentPosition(int globalIndex);
    void writeToFile();

private:
    QVector<controller*> _controllers;
    QVector<stage*> _stages;
    QVector<QString> _stageIndex;
    QVector<QStringList> _controllerInfo;
    QVector<int> _controllerID;
    QStringList _controllerNames;
};
```

Eine wichtige Funktionalität der Klasse ist es, die interne Stage-ID in einer globalen Stage-ID abrufbar zu machen, um unter anderem das Abrufen der Funktion mit der Stage-ID über den Job für den Benutzer zu vereinfachen. Die interne Stage-ID wird daher zusammen mit der Controller-ID als QString in einem QVector gespeichert (Listing 21, gelb markiert). Die globale Stage-ID entspricht dann dem Index des QVectors. Durch die Übergabe der globalen Stage-ID kann über die Funktion getStageIndex() die interne Stage-ID und die entsprechende Controller-ID abgerufen werden (Listing 22). Da die interne Stage-ID und die zugehörige Controller-ID als QString zusammengefasst wurden, wird dieser getrennt und die zwei Rückgabewerte werden als *Tupel* ausgegeben.

Listing 21: allDetectedStages.cpp, _setStages()-Funktion

```
void allDetectedStages::setStages(controller* c)
{
    int numOfStages = c->get_StageNum();
    QStringList active_stages;
    // First entry of QStringList is controllerID
    active_stages.append(QString::number(c->get_controller_ID()));
    for (int i = 0; i < numOfStages; i++)
    {
      _stages.append(c->stages.at(i));
      // set global Index of stage
      int globalIndex = _stages.count() - 1;
      _stages.at(globalIndex)->set_global_ID(globalIndex);
      // saving internal stage number
      active_stages.append(QString::number(i));
      _stageIndex.append(QString::number(c->get_controller_ID()) + "/n" +
      QString::number(i));
    }
    // save this controller's info to QVector
    _controllerInfo.append(active_stages);
}
```

Listing 22: allDetectedStage.cpp, getStageIndex()-Funktion

```
std::tuple<int, int> allDetectedStages::getStageIndex(int globalIndex)
{
    QString str = _stageIndex.at(globalIndex);
    int cID = str.split("/n").at(0).toInt();
    int sID = str.split("/n").at(1).toInt();
    return  std::make_tuple(cID, sID);
}
```

Sobald die Stages ausgewählt wurden, wird das Signal signal_stagesAssigned gesendet (siehe Tabelle 5), und die Funktion setLogData() in der controllerFactory aufgerufen (Listing 23). Diese Funktion speichert die Controller- und Stage-Pointer sowie die Controller-Namen.

Listing 23: controllerFactory.cpp, setLogData()-Funktion

```
void controllerFactory::setLogData(int controllerID)
{
    // connect pointers of allDetectedStages for controllers
    log->setControllers(c[controllerID]);
    log->setNames(_controllerNames);
    // connect pointers of allDetectedStages for stages
    log->setStages(c[controllerID]);
}
```

Listing 24: allDetectedStages.cpp, writeToFile()-Funktion

```
void allDetectedStages::writeToFile()
{
	QDate today = today.currentDate();
	int year = today.year();
	int month = today.month();
	int day = today.day();
	QDateTime x = x.currentDateTime();
	QString path = QCoreApplication::applicationDirPath() + "/logfiles/";
	QDir dir;
	if (!dir.exists(path))
		dir.mkpath(path);
	QString logName = QString::number(year) + "_" + QString::number(month) + "_"
	+ QString::number(day) + "_log.txt";
	QFile data(path + logName);
	int numOfControllers = _controllerNames.size();
	if (!data.open(QIODevice::WriteOnly | QIODevice::Append |
	QIODevice::Text))return;
	QTextStream out(&data);
	for (int i = 0; i < numOfControllers; i++)
	{
		out << "ControllerID: " << QString::number(i) << ",    Controller
		name: " << _controllerNames.at(i) << "\n";
		int numOfStages = _stages.size();
		for (int j = 0; j < numOfStages; j++)
		{
			auto x = getStageIndex(j);
			int cID = std::get<0>(x);
			if (cID == i)
			{
				int sID = std::get<1>(x);
				out << "Stage: " << QString::number(j) << ", Internal
				StageID: " << QString::number(sID) << "\n";
			}
		}
	}
}
```

Die letzte Funktion ist die Logdateierstellung (Listing 24). Das entstehende Log beinhaltet das Erstellungsdatum und die Erstellungszeit (gelb markiert), die Controller-IDs, Controllernamen und internen Stage-IDs (grün markiert). Hier kommt die getStageIndex()-Funktion (Listing 22) zum Einsatz. Diese ermöglicht das Laden der Controller und der entsprechenden internen Stage-IDs anhand der gesamten Stage-Liste.

3.2.3 Implementierung der Makros

Die PI-Bibliothek beinhaltet Funktionen zum Abfahren von Bewegungsbahnen (sog. Trajektorien). Zur Implementierung der Anforderungen SW14-17 (Tabelle 2) wurde diese Funktionalität, wie im Kapitel 3.1.2 erläutert, gewählt.

Die Benutzung der Trajektorien ist durch eine bestimmte Reihenfolge von PI-Bibliotheksbefehlen möglich. In Abbildung 13 ist das in der Bedienungsanleitung gezeigte Flowchart zu sehen, welches die Reihenfolge der PI-Befehle abbildet.

Die trajectory-Funktionalität wird durch Buttons in der GUI aufgerufen. Die Verbindungen zwischen den GUI-Buttons und den Slots zum Aufrufen der Trajektorien sind in der Tabelle 4: Signal-Slot Verbindungen qMotionPlugin Klasse gezeigt.

Eine wichtige Definition für die Makro Implementierung ist die Schnittstelle zu smartLab. Die Datenpunkte werden in einer Textdatei übergeben. Dies gibt dem Benutzer die Auswahlmöglichkeit, über ein anderes Plugin die Datenpunkte zu ermitteln und zu übergeben oder manuell Datenpunkte zu definieren, sodass zur Anwendung nicht unbedingt erst ein Job definiert werden muss. Das Format ist eine Tabelle von x-, y-, und z-Koordinaten. Pro Zeile werden die Werte für einen Punkt gespeichert und das Trennzeichen ist ein Komma (siehe Abbildung 12). Hierbei müssen die Datenpunkte der Textdatei nicht sortiert sein.

```
1,1,1
2,1,1
3,1,1
2,2,1
3,2,1
1,2,1
3,3,1
2,3,1
4,3,1
```

Abbildung 12: Beispiel einer Textdatei mit Makrodatenpunkten

Die trajectory-Klasse hat folgende Funktionalitäten:

- Lesen der Textdatei mit Datenpunkten
- Sortieren der Datenpunkte
- Bauen der Trajektorie mit den notwendigen Beschleunigungs- und Entschleunigungsrampen
- Verfahren der Stage über die Reihenfolge der Funktionen (siehe Flowchart Abbildung 13)

Da das Plugin für die Nutzung der Trajektorie betriebsbereit sein muss, wurden die ersten Schritte des Flowcharts (Servomotor anschalten, Referenzieren) bereits beim Auswählen und Referenzieren der Stages durchgeführt (Kapitel 3.2.1).

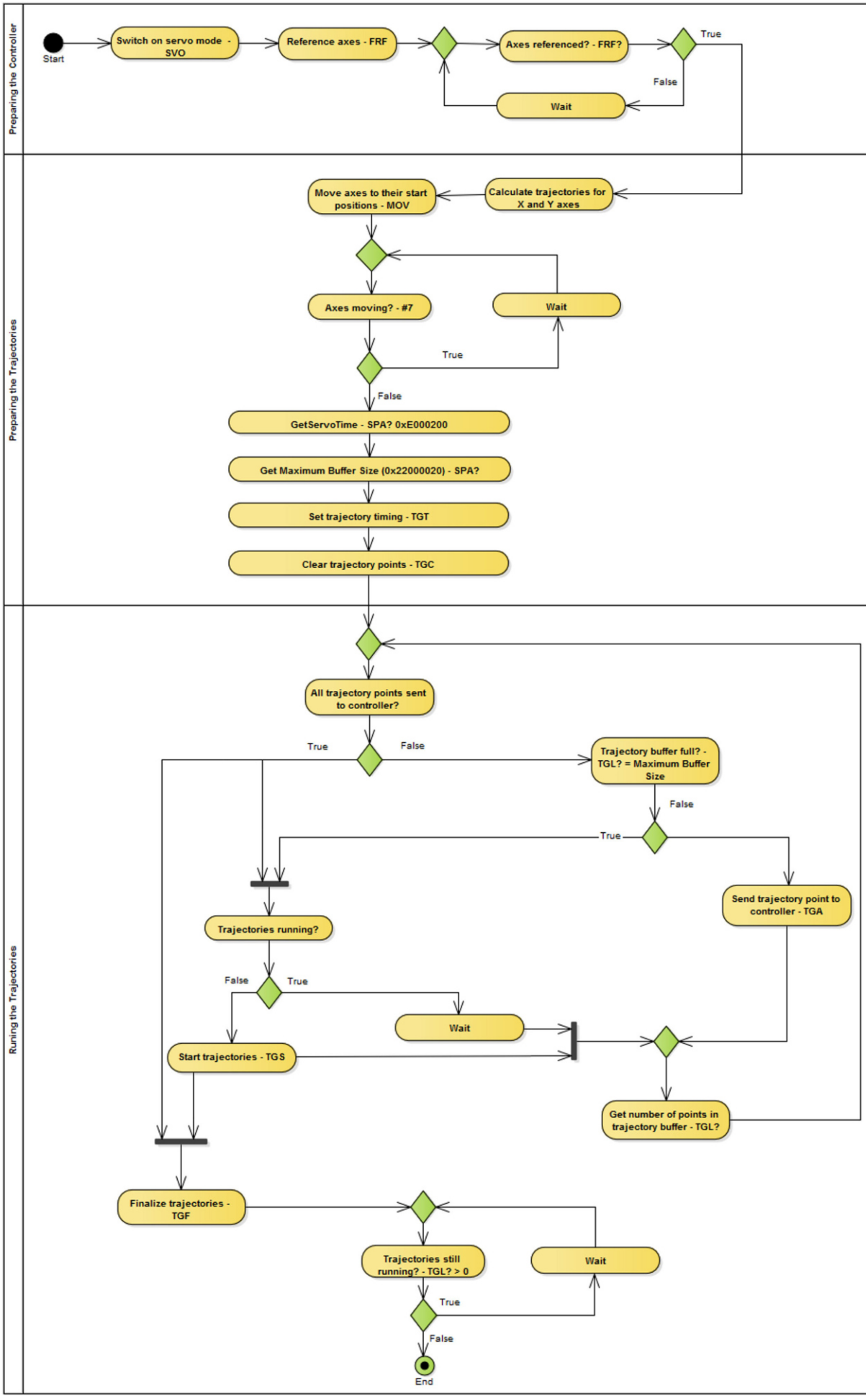

Abbildung 13: Flowchart für die PI-Bibliothek-Funktionsaufrufe zur Benutzung der Trajektorien [13]

Die controllerFactory-Instanz manager erstellt eine trajectory-Instanz, über die die Funktionen der trajectory-Klasse aufgerufen werden können. Es gibt Funktionen zum Laden, Starten und Stoppen von Makros, die über Buttons in der Plugin-GUI aufgerufen werden können. Die Signal-Slot-Verbindungen werden in Tabelle 4 gezeigt.
Das Verwenden der Makros beginnt mit der Funktion slot_loadMacro() (Listing 25), in der über einen Dialog der gewünschte Makrodatensatz ausgewählt werden kann.

Listing 25: trajectory.cpp, slot_loadMacro()-Funktion

```
void controllerFactory::slot_loadMacro()
{
    QStringList macros;
    chooseMacroDialog* pChooseMacroDialog = new chooseMacroDialog();
    QString chosenMacro = pChooseMacroDialog->runDialog(macros);
    macro->readFromFile(chosenMacro);
}
```

Nach der Auswahl wird dann die readFromFile()-Funktion (Listing 26) von dem gewählten Makro durchgeführt. Abhängig vom übergebenen Makronamen wird die entsprechende Textdatei geladen.

Listing 26: controllerFactory.cpp, readFromFile()-Funktion

```
void trajectory::readFromFile(QString macro)
{
    macroData.clear();
    QString path = QCoreApplication::applicationDirPath();
    QDir dir;
    if (!dir.exists(path))
        qDebug("PI PLUGIN: macro file path does not exist.");
    QFile data(path + "/" + macro + ".txt");
    if (data.exists())
    {
        if (data.open(QIODevice::ReadOnly | QIODevice::Text))
        {
            QTextStream in(&data);
            QString line;
            while (!in.atEnd())
            {
                line = in.readLine();
                QStringList coordinates = line.split(", ");
                float x = coordinates.at(0).toFloat();
                float y = coordinates.at(1).toFloat();
                float z = coordinates.at(2).toFloat();
                macroData.push_back({x, y, z});
            }
        }
        else
        {
            qDebug("PI PLUGIN: macro.txt does not exist.");
        }
        data.close();
    }
}
```

In Listing 26 ist gelb markiert zu sehen, wie die Datenpunkte aus der Textdatei in Variablen vom Typ float konvertiert und in einen Vektor geschrieben werden.

Die Funktion slot_loadMacro() ruft die Methoden readFromFile() und prepareTrajectory() auf, um die Datenpunkte für die Trajektorie vorzubereiten. slot_startMacro() startet dann die vorbereitete Trajektorie und durch die Funktion runTrajectory(). Durch die Funktion slot_cancelMacro() werden alle Controller in ihrer Bewegung gestoppt.

Die prepareTrajectory()-Funktion beinhaltet mehrere Schritte, um die Trajektorie für das Verfahren vorzubereiten (entsprechend dem Prepare Trajectories Abschnitt im Flowchart, Abbildung 13).

Konkret für dieses Projekt bedeutet das Folgendes:

- Sortieren der Datenpunkte aus dem QVector, der in der readFromFile()-Funktion die Daten eingelesen hat
- Festlegen der Startkoordinaten der Trajektorie
- Bauen der Trajektorie in zwei QVectoren (X- und Y-Koordinaten)
- Ermitteln der Servozeit
- Setzen des Trajectory Timings (zeitlicher Abstand zwischen den Punkten der Trajektorien) anhand er gewünschten Geschwindigkeit und der ermittelten Servozeit
- Leeren des Trajectory Buffers im Controller

Listing 27: Ausschnitt aus trajectory.h

```
struct vect3
{
    float x;
    float y;
    float z;
};
struct vect2
{
    float x;
    float y;
};
std::vector<vect3> macroData;
std::vector<vect2> layerData;
std::vector<vect2> rowData;
```

Das Sortieren der Datenpunkte geschieht durch die Verwendung von mehreren Vektoren, die wiederum verschiedene *Struktur*typen enthalten (Listing 27). Es wurde eine *Struktur* mit drei Koordinaten und eine mit zwei Koordinaten definiert.
In der readFromFile()-Funktion (Listing 26) werden die Datenpunkte in den „macroData“-Vektor mit der Struktur „vect3“ eingelesen. Dieser muss nun nach jeder Komponente sortiert werden (Listing 28, gelb markiert).
Als Erstes wird der Vektor nach der z-Komponente sortiert. Danach wird ermittelt wie viele z-Elemente der Datensatz enthält. Hierbei wird die Funktion std::set verwendet, um alle einzigartigen Elemente zu identifizieren und in den Vektor „zCountVector“ zu speichern. Durch das Bestimmen der Größe dieses Vektors ist nun die Anzahl der einzigartigen z-Elemente bekannt.
Jetzt kann über „macroData“ iteriert und die x- und y-Werte mit den gleichen z-Elementen in den „layerData“-Vektor mit der Struktur „vect2“ geschrieben werden.
Der „layerData“-Vektor enthält nun die x- und y-Werte einer Laserablationsebene z.

Listing 28: trajectory.cpp, Ausschnitt aus prepareTrajectory()

```
std::sort(macroData.begin(), macroData.end(), sortDataByZ);
std::vector<float> zCountVector;
for (int i = 0; i < macroData.size(); i++)
{
	zCountVector.push_back({macroData.at(i).z});
}
zCount = std::set<float>(zCountVector.begin(), zCountVector.end()).size();
for (int j = 0; j < zCount; j++)
{
	layerData.clear();
	for (int i = 0; i < macroData.size(); i++)
	{
		if (macroData.at(i).z == macroData.at(j).z)
		{
			layerData.push_back({ macroData.at(i).x, macroData.at(i).y});
		}
	}
	std::sort(layerData.begin(), layerData.end(), sortDataByY);
	std::vector<float> yCountVector;
	for (int i = 0; i < layerData.size(); i++)
	{
		yCountVector.push_back({ layerData.at(i).y });
	}
	yCount = std::set<float>(yCountVector.begin(), yCountVector.end()).size();
	int x = 0;
	for (int j = 0; j < yCount; j++)
	{
		rowData.clear();
		for (int i = 0; i < layerData.size(); i++)
		{
			if (layerData.at(i).y == layerData.at(j).y)
			{
				rowData.push_back({ layerData.at(i).x,
				layerData.at(i).y });
			}
		}
		std::sort(rowData.begin(), rowData.end(), sortDataByX);
		int s = rowData.back().x * 0.1;
		double startPoint[2];
		bool evenRow;
		if (j % 2 == 0)
		{
			startPoint[0] = rowData.at(0).x - s;
			startPoint[1] = rowData.at(0).y;
			evenRow = true;
		}
		else
		{
			startPoint[0] = rowData.at(0).x + s;
			startPoint[1] = rowData.at(0).y;
			evenRow = false;
		}
		buildTrajectoryRampAsc(startPoint, v, s);
		buildLineTrajectory(rowData.size());
		startPoint[0] = _trajectoryDataX.last();
		buildTrajectoryRampDesc(startPoint, v, s);
		buildMoveToNextRow(evenRow, v, s);
	}
}
```

„layerData“ wird nun nach der y-Komponente sortiert und es wird wieder die Anzahl der einzigartigen y-Elemente bestimmt. Die x-Werte für den gleichen y-Wert werden in den Vektor „rowData“ geschrieben. Wie der Name schon andeutet, beinhaltet „rowData“ die Werte einer Ablationsreihe. Zuletzt wird „rowData“ noch nach dem x-Element sortiert.

Die Implementierung der Trajektorien beschränkt sich auf das Abfahren der Ablationsfläche durch das Meandermuster (Abbildung 14).

Abbildung 14: Meandermuster

Der grün markierte Teil in Listing 28 zeigt die Bestimmung des Startpunktes einer Reihe. Abhängig davon, ob es eine Hinreihe oder eine Rückreihe ist, ist der Startpunkt der Reihe entweder vor oder hinter der Beschleunigungsrampe. Also wird bei jedem geradzahligen Reihenindex die Rampenstrecke addiert oder subtrahiert.

Als nächster Schritt wird die Trajektorie gebaut. Für jede Reihe werden die blau markierten Funktionen aus Listing 28 ausgeführt.

Folgende Funktionen gibt es:

- buildTrajectoryRampAsc(): Bauen der Beschleunigungsrampe in einer Hinreihe (Listing 29)
- buildReversedTrajectoryRampAsc(): Bauen der Beschleunigungsrampe in einer Rückreihe
- buildTrajectoryRampDesc(): Bauen der Entschleunigungsrampe in einer Hinreihe
- buildReversedTrajectoryRampDesc(): Bauen der Entschleunigungsrampe in einer Rückreihe
- buildLineTrajectory(): Bauen der Laserablationsstrecke einer Reihe
- build MoveToNextRow(): Bauen der Fahrstrecke zwischen den Reihen

Listing 29: trajectory.cpp, buildTrajectoryRampAsc()

```
void trajectory::buildTrajectoryRampAsc(double* startingPoint, float v, int s)
{
	for (int i = 0; i < s; i++)
	{
		startingPoint[0] = startingPoint[0] + (v / 2) * (1 - cos((i * M_PI) / s));
		_trajectoryDataX.append(startingPoint[0]);
		_trajectoryDataY.append(startingPoint[1]);
	}
}
```

Die Koordinaten für die Trajektorie werden in diesen Funktionen in zwei QVektoren (x-Koordinaten, y-Koordinaten) geschrieben. Listing 29 zeigt die buildTrajectoryRampAsc()-Funktion, welche die im Lösungsansatz erläuterte Formel verwendet, um eine steigende Flanke als Beschleunigungsrampe zu erstellen. Die anderen Rampenfunktionen sind auf ähnliche Weise implementiert und unterscheiden sich in der Wahl des Iterationsstart- und -endpunktes, sowie der Iterationsrichtung (i++/i--) der for-Schleife. Die buildLineTrajectory()-Funktion baut die eigentliche Laserablationsfläche und übernimmt die Koordinaten aus dem Datensatz. In der buildMoveNextRow()-Funktion werden abhängig von der Distanz zwischen

den Reihen entweder Beschleunigungsrampen oder bei einer kurzen Distanz eine lineare Strecke gebaut. Nachdem über alle Daten iteriert wurde, enthalten diese Vektoren (Datentyp QVektor) nun alle Datenpunkte für jede Ebene mit Beschleunigungs- und Entschleunigungsrampen, sodass die eigentliche Ablationsfläche, wie in SW17 gefordert, eine konstante, eingestellte Geschwindigkeit enthält.

Listing 30: trajectory.cpp, runTrajectory()-Funktion

```
void trajectory::runTrajectory(int iD, char* stages, float v)
{
	int trajectoryPointCount = _trajectoryDataX.size();
	bool isTrajectoryRunning = false;
	double startPoint[2] = { _trajectoryDataX.at(0), _trajectoryDataY.at(0) };
	moveToStart(iD, stages, startPoint);
	waitForMotionDone(iD);
	getMaxBufferSize(iD);
	unsigned int idx = 0;
	while (idx < (unsigned int)trajectoryPointCount)
	{
		currentTrajectoryPoint[0] = _trajectoryDataX.at(idx);
		currentTrajectoryPoint[1] = _trajectoryDataY.at(idx);
		getNumberOfTrajectoryPoints(iD, trajectoryIDs);
		if (currentPointsInTrajectory[0] < (int)idBufferSize &&
		currentPointsInTrajectory[1] < idBufferSize)
		{
			appendTrajectoryPoints(iD, trajectoryIDs);
			idx++;
		}
		else
		{
			if (!isTrajectoryRunning)
			{
				isTrajectoryRunning = StartTrajectory(iD, trajectoryIDs);
			}
			else
			{
				Sleep(1);
			}
		}
	}
	if (!isTrajectoryRunning)
	{
		StartTrajectory(iD, trajectoryIDs);
	}
	while (currentPointsInTrajectory[0] > 0 && currentPointsInTrajectory[1] > 0)
	{
		Sleep(10);
		getNumberOfTrajectoryPoints(iD,currentPointsInTrajectory);

	}
}
```

Die runTrajectory()-Funktion im slot_startMacro() wird nun aufgerufen, um die vorbereitete Trajektorie abzufahren (Listing 30). Der erste Schritt des Abfahrens ist das Fahren der Tische zum Startpunkt der Trajektorie durch die MoveToStart()-Funktion. Nach dem Erreichen der Startposition und dem Abfragen der maximalen Buffergröße werden die Koordinatenpunkte einzeln in den Buffer geladen. Sobald die maximale Kapazität des Buffers erreicht wurde, wird das Abfahren der Trajektorie gestartet. Wurde ein Punkt abgefahren, wird dieser aus dem Buffer entfernt und die Anzahl der geladenen Punkte sinkt, sodass neue Punkte nachgeladen werden können. Nachdem sichergestellt wurde, dass alle Punkte in der Trajektorie abgefahren wurden, wird die Funktion beendet.

3.2.4 Tests

Die Plugin-Funktion muss überprüft werden, um die geforderte Funktionalität sicherzustellen. Dafür wurde ein Akzeptanztest geplant, spezifiziert und durchgeführt.

Tabelle 6: Übersicht der Testfälle

Schlüssel	Funktionalität	Verwendete Klassen	Status
TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab	qMotionPlugin, controllerFactory, allDetectedStages, controller, stage, stageFrame, chooseControllerDialog, assignStageDialog	Bestanden
TC2	Automatisiertes Verfahren über Trajektorien	qMotionPlugin, controllerFactory, allDetectedStages, controller, stage, stageFrame, chooseControllerDialog, assignStageDialog, trajectory, chooseMacroDialog smartLab-Software und pluginspezifischer Job	In Planung
TC3	Verwendung der Pluginfunktionen über einen Job in smartLab	qMotionPlugin, controllerFactory, allDetectedStages, controller, stage, stageFrame, chooseControllerDialog, assignStageDialog, trajectory, chooseMacroDialog smartLab-Software und pluginspezifischer Job	In Planung, noch nicht spezifiziert

Tabelle 6 zeigt eine Übersicht der geplanten Tests. Testfall TC1 überprüft die manuelle Fahrfunktion. Der Testfall TC2 konnte nicht durchgeführt werden, da die PI-Bibliotheksbefehle für die Trajektorien in einer neueren Version der Bibliothek verfügbar waren, die allerdings nicht mit dem vorhandenen Softwaremodul kompiliert werden konnte. Die Methoden wurden, wie in Kapitel 3.2.3 beschrieben, angelegt. Die Zwischenschritte konnten nicht kompiliert und daher auch nicht überprüft werden.

Abbildung 15: Ausschnitte der Konsolenausgabe aus der laufenden PI Beispielsoftware

```
Turned servo on
Moved to start position of trajectory.
Set trajectory rate to 400 (3 / 150 / 5e-05.
Successfully prepared trajectory.
Buffer size: 128
Number of values in buffer: 0, 0
1 / 150 points added
Number of values in buffer: 1, 1
2 / 150 points added
Number of values in buffer: 2, 2
3 / 150 points added
Number of values in buffer: 3, 3
4 / 150 points added
Number of values in buffer: 4, 4
5 / 150 points added
Number of values in buffer: 5, 5
6 / 150 points added
Number of values in buffer: 6, 6
7 / 150 points added
```

```
Number of values in buffer: 22, 22
Number of values in buffer: 20, 20
Number of values in buffer: 17, 17
Number of values in buffer: 15, 15
Number of values in buffer: 14, 14
Number of values in buffer: 12, 12
Number of values in buffer: 10, 10
Number of values in buffer: 8, 8
Number of values in buffer: 6, 6
Number of values in buffer: 4, 4
Number of values in buffer: 1, 1
Number of values in buffer: 0, 0

End of trajectory.
Turned servo off
Closing connection.
```

Die Funktionalität der Trajektorien mit dem Controller und den Stages konnte durch das Testen der Hardware mit dem Beispielprojekt, das von Physik Instrumente mit dem Controller geliefert, wurde sichergestellt werden (siehe Abbildung 15).

Der Job, der zum Durchführen des Testfall TC3 notwendig ist, wurde aus Zeitgründen nicht implementiert und der Testfall noch nicht spezifiziert. Zum Testen muss der Job in der Software smartLab angelegt werden. Die Funktionen sind über das Interface für den Job abrufbar.

Tabelle 7: Akzeptanztest für das Softwaremodul QMotionPlugin

Schlüssel	Testfall Zusammenfassung	Test Schritt	Test Schritt Beschreibung	Erwartetes Ergebnis	Test Status
TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab	Vorbereitung	PC anschalten, Software smartLab starten.	PC startet und lädt Windows 10, smartLab startet und lädt die Haupt-GUI	Bestanden
		Erkennen der Pluginbibliotheken und der Hardware	Button "Detect and Initialize" drücken	Detect()-Funktion der eingebundenen Plugins werden aufgerufen, der Auswahldialog der Plugins füllt sich mit den Plugins deren Hardware erkannt wurde.	Bestanden
		Laden der Plugin-GUI	Auswahl (Klicken) des Plugins "QMotionPlugin" im Plugin-Dialog.	Nach der Auswahl des Plugins aus der Liste wird die Plugin-GUI geladen.	Bestanden
		Laden der Controllerliste	Drücken des Buttons "Choose Controller(s)".	Liste der angeschlossenen Controller werden aus der Hardware geladen und im Dialog angezeigt.	Bestanden
		Auswahl der Controller	Es können mehrere Controller im Dialog selektiert werden. Zur Bestätigung der Wahl wird der Button "Ok" gedrückt.	Ausgewählte Controller werden als GUI-Widgets in die Plugin-GUI geladen.	Bestanden
		Laden der Stage-Liste	Drücken des Buttons "Assign Stages".	Stage-Dialog wird geöffnet und die Liste der Stages wird aus dem Controller abgefragt und in die linke Liste des Dialogs geladen. Sollte eine Vorauswahl der Stages für diesen Controller gespeichert sein, wird diese in die rechte Liste geladen.	Bestanden
		Auswahl der Stages	Auswählen der Stage auf der linken Liste und Auswählen des zuzuweisenden Indexes auf der rechten Liste. Drücken des Buttons "Assign". Klicken des "No stage" Buttons für keine Zuweisung des Indexes.	Die ausgewählte Stage auf der linken Seite wird auf den selektierten Index auf der rechten Seite übertragen. Wenn der "No stage"-Button gedrückt wurde, wird NOSTAGE in den Indexeintrag geladen.	Bestanden

Schlüssel	Testfall	Test Schritt	Test Schritt Beschreibung	Erwartetes Ergebnis	Test Status
		Laden der Stageframes	Drücken des "Ok"-Buttons im Stage-Dialog	Aus dem Dialog werden die zugewiesenen Stages aus der rechten Liste geladen. Aus den ausgewählten Stages werden nun Stageframes für die Plugin-GUI erstellt. Die Stageframes werden in die Plugin-GUI geladen, sind aber nicht bedienbar und ausgegraut.	Bestanden
		Referenzieren einer Stage	Drücken des "Reference"-Buttons in einem Controller-Widget	Die Stages des ausgewählten Controllers fahren zu ihrer gespeicherten Referenzposition. Dann fahren sie zur gespeicherten Home-Position. Sobald die Referenzfahrt zuende ist, werden die Stageframes zur Verwendung freigeschaltet.	Bestanden
		Verfahren einer Stage um ein Inkrement nach rechts	Eingabe des Inkrements in das Eingabefeld. Drücken der Pfeiltaste nach rechts.	Stage verfährt um das Inkrement nach rechts. Überprüft wird die Position durch die Anzeige in der GUI und der Bewegung der Stage.	Bestanden
		Verfahren einer Stage um ein Inkrement nach links	Eingabe des Inkrements in das Eingabefeld. Drücken der Pfeiltaste nach links.	Stage verfährt um das Inkrement nach links. Überprüft wird die Position durch die Anzeige in der GUI und der Bewegung der Stage.	Bestanden
		Verfahren einer Stage an die linke Grenze	Drücken der linken Grenzpfeiltaste	Stage verfährt bis zur maximal linken Position.	Bestanden
		Verfahren einer Stage an die rechte Grenze	Drücken der rechten Grenzpfeiltaste	Stage verfährt bis zur maximal rechten Position.	Bestanden
		Verfahren einer Stage an eine eingegebene Position	Eingabe der gewünschten Position in des Positionseingabefeld. Drücken des "Go"-Buttons	Stage verfährt zur gewünschten Position. Die Anzeige zeigt die gewünschte Position an.	Bestanden

Schlüssel	Testfall	Test Schritt	Test Schritt Beschreibung	Erwartetes Ergebnis	Test Status
		Einstellen einer anderen Geschwindigkeit der Stage	Eingabe der gewünschten Geschwindigkeit in das Velocity-Eingabefeld. Drücken des "Set Velocity"-Buttons. Verfahren der Stage auf beliebige Weise (z.B. über dem linken Grenzpfeilbutton)	Stage verfährt zur gewünschten Position in der eingestellten Geschwindigkeit.	Bestanden
		Stoppen der Controller	Drücken des "Panic Stop"-Buttons, während die Stage oder mehrere Stages fahren.	Alle Stages stoppen, der Das Status-Label zeigt "Controllers stopped". Sobald der Stopvorgang beendet ist, wird das Label wieder auf den Status "Referenced" aktualisiert.	Bestanden
TC2	Automatisiertes Verfahren über Trajektorien	Vorbereitung	PC anschalten, Software smartLab starten. QMotionPlugin einbinden, Controller und Stages einbinden und referenzieren.	Plugin lädt die Plugin-GUI ordnungsgemäß und das manuelle Fahren ist möglich.	In Planung
		Auswählen und Laden des Makros	Drücken des "Load Macro..."-Buttons, Auswahl des gewünschten Datensatzes, Drücken des "Ok"-Buttons im Dialog	Der Macro-Dialog wird geöffnet und zeigt alle zur Verfügung stehenden Datensätze an. Ein Datensatz kann ausgewählt werden.	In Planung
		Ausführen des Makros	Drücken des Buttons "Start Macro".	Die Trajektorienfahrt wird ausgeführt.	In Planung
		Beenden des Makros	Drücken des Buttons "Cancel Macro"	Stoppen der Macrofahrt.	In Planung

Zwecks Überprüfung der Deckung aller Lastenheftanforderungen durch die Softwareanforderungen, wurde eine Coverage Analyse durchgeführt.

Tabelle 8: Coverage Analyse, Überprüfung der Deckung der Softwareanforderungen durch Testfälle

Schlüssel	Softwareanforderung	Schlüssel	Testfall
		Coverage 100%	
SW1	Das Plugin soll als Plugin in smartLab auf Windows 10 lauffähig sein.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
		TC2	Automatisiertes Verfahren über Trajektorien
SW2	Die Software soll als Dynamic Library (.dll) programmiert werden.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
		TC2	Automatisiertes Verfahren über Trajektorien
SW3	Das Plugin soll in C++ und Qt programmiert werden.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
		TC2	Automatisiertes Verfahren über Trajektorien
SW4	Das Plugin soll die PI Bibliothek einbinden.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
		TC2	Automatisiertes Verfahren über Trajektorien
SW5	Das Plugin soll mehrere Controller von PI einbinden können, welche man in der GUI auswählen kann.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
SW6	Das Plugin soll mehrere Stages einbinden können, welche in der GUI auswählbar sind.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
SW7	Jede Stage soll einzeln per Buttonklick in zwei Richtungen verfahrbar sein.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
SW8	Die Fahrgeschwindigkeit soll für jede Stage einstellbar sein.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
SW9	Es soll eine Stop-Funktion für den Controller geben.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
SW10	Jede Stage soll eine Option zum Verfahren mit einer Schrittweite haben, welche vorher eingegeben werden kann.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab

SW11	Es soll eine Anzeige für jede Stage geben, die die aktuelle Position der Stage anzeigt.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
SW12	Es soll eine Möglichkeit geben per Buttonklick zu den Randpositionen zu fahren.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
SW13	Es soll eine Klasse geben, die alle Daten zu der angeschlossen Hardware sammelt und abrufbar macht.	TC1	Verwendung der manuellen Fahrfunktion des QMotionPlugins über smartLab
SW14	Es soll möglich sein die Tische über ein vorgegebenes Muster zu verfahren.	TC2	Automatisiertes Verfahren über Trajektorien
SW15	Es soll eine Schnittstelle für die Übergabe von Datenpunkten für die abzufahrende Fläche geben.	TC2	Automatisiertes Verfahren über Trajektorien
		TC3	Verwendung der Pluginfunktionen über einen Job in smartLab
SW16	Das Plugin soll eine möglichst effiziente Fahrstrecke für die Datenpunkte berechnen.	TC2	Automatisiertes Verfahren über Trajektorien
SW17	Das Timing der Fahrt soll bestimmbar sein, um die Lasertriggerung synchronisieren zu können.	TC2	Automatisiertes Verfahren über Trajektorien

4. Diskussion

Im Rahmen dieser Arbeit wurde ein Softwaremodul zur Steuerung eines Verfahrtischsystems bei der automatisierten Laserablation von Gewebeproben entwickelt. Als Erstes wurden die Anforderungen an das Programm gesammelt und dann ein Konzept für die Klassenarchitektur entwickelt. Für die wichtigen Problemstellungen wurden geeignete Lösungsansätze recherchiert und herausgearbeitet. Nach der Konzeptionierung wurden dann die Klassen mithilfe der Lösungsansätze implementiert. Für die Überprüfung der Funktionalität und der Erfüllung der Anforderungen wurde ein Akzeptanztest mit Testfällen entworfen. Testfall TC1 wurde erfolgreich durchgeführt und bestätigt die Funktionalität der manuellen Fahrfunktionen über die Plugin-GUI. Die Testfälle TC2 und TC3 sind in Planung.
Der Akzeptanztest (Testfall TC1, Tabelle 7) zeigt, dass das manuelle Fahren durch smartLab möglich ist. Die Funktionalität des automatisierten Fahrens ist noch nicht einsatzbereit, sodass weitere Tests nicht möglich waren (Testfälle TC2, TC3). Die Coverage Analyse der Softwareanforderungen bestätigt (Tabelle 8), dass durch die Testfälle eine hundertprozentige Deckung der Anforderungen erreicht wurde.
Wie im letzten Kapitel erläutert, ist der Testfall TC2 noch nicht durchführbar, da die PI-Bibliotheksversion, die für die Ausführung der Trajektorienfunktion notwendig ist, nicht in das Projekt eingebunden werden konnte.
Die Funktionalität wurde über eine Beispielprojekt sichergestellt (Abbildung 15). Die Trajektorie konnte in diesem Projekt mit der erarbeiteten Rampenformel verwendet werden. Hierbei wurde ein 32-Bit Compiler verwendet, zudem ist der Ausgabetyp des Projekts kein Plugin (.dll) und es wurde auch nicht im Zusammenhang mit Qt programmiert. Diese Spezifikationen bieten Ansätze für eine weitere Lösungsfindung zur erfolgreichen Einbindung der aktuellen PI-Bibliotheksversion.
Die Plugin-Funktionalität über den Job (Testfall TC3) ist relativ einfach umzusetzen. Es muss ein Job in smartLab angelegt werden. Das bedeutet eine Job-Klasse für die konkrete Aufgabe (z.B. Verfahren der Tische) muss erstellt werden, die vom Interface IJob erbt. In dieser Klasse können dann verschiedene Fahrfunktionen definiert werden, die über das IStage-Interface die Plugin-Funktionen (Listing 1, S. 17) aufrufen können. Beim Testen kann dann, nach der Einbindung der Stages, der Button „Jobs“ (Abbildung 2, S.16) geklickt werden. Es erscheint ein Dialog der eingebundenen Jobs. Nach der Auswahl kann dann der definierte Job verwendet und die Fahrfunktion über die Job- und die Plugin-GUI überprüft werden.

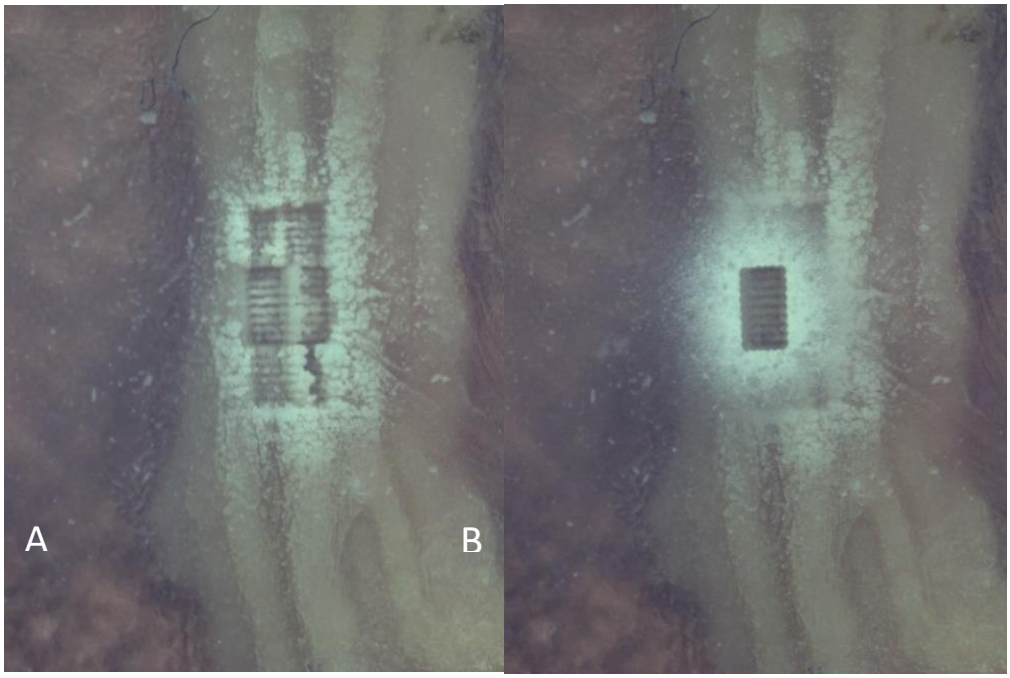

Abbildung 16: Gewebeprobe nach mehreren Laserablationen (A), Objektträger über der ablatierten Gewebeprobe mit gesammeltem Aerosol (B) (Quelle: UKE, 2021)

In der Aufgabenstellung wurde erklärt, wie die derzeitige Probengewinnung durch den PIRL funktioniert. Auch wurde auf den Nachteil eingegangen, dass durch die Ablation mittels der Laserstrahlbewegung ein Großteil der Gewebeprobe verloren geht.
Abbildung 16 zeigt eine ablatierte Gewebeprobe mit (B) und ohne Objektträger(A). Auf dem Bild B kann das aufgefangene Aerosol erkannt werden. Auch zu sehen ist ein rechteckiger Bereich in der Probe auf dem Objektträger. An dieser Stelle wurde das gesammelte Aerosol auf dem Objektträger durch den Laser abgetragen und kann nicht mehr analysiert werden. Durch die Anwendung des entwickelten Softwaremoduls, kann ein Großteil des Probenverlustes verhindert werden, indem die Gewebeprobe bewegt wird und nicht den Laserstrahl.
Der aktuelle Entwicklungsstand ermöglicht die Laserablation durch das manuelle Verfahren der Tische über die grafische Benutzeroberfläche des Plugins. Die Ablation durch das manuelle Verfahren ermöglicht eine weitaus größere Auflösung als die Ablation durch die Laserstrahllenkung, da sie den Verlust von großen Mengen der Gewebeprobe verhindert.

5. Ausblick

Die Weiterentwicklung des Plugins durch das Team im Laserlabor am UKE ist derzeit in Planung. Eine neue Stage vom Modelltyp M-531.EC wurde geliefert. Dieser Tisch soll die x-Achsenbewegung während der Ablation übernehmen, da sie eine maximale Geschwindigkeit von 100 mm/s und einem Stellweg von 306 mm besitzt. Dadurch können die geraden Bereiche des Meandermusters möglichst effizient gefahren werden. Auch bietet dieser Tisch die Möglichkeit die Proben unter dem Objektträgertischsystem herauszufahren. Das ermöglicht eine schnelle Be- und Entladung der Probe, sowie eine Aufnahme von *OCT*-Bildern.
Das endgültige Ziel des Teams ist es, ein System (den sog. Auto-Sampler) zu entwickeln, das durch 3D-Bildgebung das gewünschte Volumen identifiziert und dann automatisiert ablatiert. Es soll möglich sein, interessante Proben zu identifizieren, zu ablatieren und auf einem Glass-Array für die Sortierung in Reagenzgefäßen für die Massenspektrometrie zu sammeln. Diese sortierten Proben können dann dem Gewebe/Tumor topographisch zugeordnet werden. Dies ermöglicht eine weitaus größere Auflösung und eine 3D-Zuordnung der analysierten Proben zu dem Gewebe, sodass genau nachvollzogen werden kann, was in welchen Bereich des Gewebes vorhanden war.
Das Softwaremodul schafft eine Grundlage für die Entwicklung der automatisierten Fahrt während der Ablation und der Probensammlung, und letztendlich für das Ziel des Auto-Samplerprojekts der M3I-Forschungsgruppe. Auch bietet das Modul Möglichkeiten zur Weiterentwicklung für andere Verwendungszwecke von Tischen.

6. Literaturnachweise

1 Hasin Y et al. Multi-omics approaches to disease. Genome Biol 2017; 18(1):83. doi: 10.1186/s13059-017-1215-1.

2 Aumann S et al. (2019) Optical Coherence Tomography (OCT): Principle and Technical Realization. In: Bille J. (eds) High Resolution Imaging in Microscopy and Ophthalmology. Springer, Cham. https://doi.org/10.1007/978-3-030-16638-0_3

3 El-Gabalawy H et al. Epidemiology of Immune-Mediated Inflammatory Diseases: Incidence, Prevalence, Natural History, and Comorbidities. The Journal of Rheumatology 2010; 37 Suppl 85; doi:10.3899/jrheum.091461

4 Baumgart DC et al. Biological Therapies in Immune-Mediated Inflammatory Diseases: Can Biosimilars Reduce Access Inequities?. Frontiers in Pharmacology; 28 March 2019; https://doi.org/10.3389/fphar.2019.00279

5 Lu Y. Atmospheric Pressure Desorption by Impulsive Vibrational Excitation Mass Spectrometry (AP-DIVE-MS). Dissertation; 2017; https://ediss.sub.uni-hamburg.de/handle/ediss/7606

6 Kwiatkowski M et al. Homogenization of tissues via picosecond-infrared laser (PIRL) ablation. Giving a closer view on the in-vivo composition of protein species as compared to mechanical homogenization. J Proteomics 2016; 134:193–202. doi: 10.1016/j.jprot.2015.12.029.

7 Stroustrup B. Abgerufen 30.08.2021 von https://www.stroustrup.com/C++.html

8 The Qt Company. Abgerufen 30.08.2021 von https://www.qt.io/product

9 Chacon S, Straub B. Pro Git. Apress; 2014: S. 8-11

10 Ezust A, Ezust P. An Introduction to Design Patterns in C++ with Qt4. Upper Saddle River, NJ: 7 Pearson Education, Inc.; 2007

11 The "Gang of Four": Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional; 1994

12 Contieri M. Singleton Pattern: The Root of all Evil. Jun 2020

13 Physik Instrumente (2020). Benutzerhandbuch. C-884.XDC Digitaler Controller Für Positioniere Mit DC-Motor. Version: MS243DE, 07.08.2020: S. 87

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Bachelorarbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommen Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort, Datum

Unterschrift