7.3. Vollständige Automatisierung per Pipeline

In diesem Kapitel wird die Implementierung einer pipelinebasierten Automatisierung für CI/CD, die den in Kapitel 5 [TODO: Verweis] beschriebenen Arbeitsablauf ermöglicht, erörtert. Dieses Pipelinesystem wurde darauf konzipiert, sowohl Teile des Eclipse-Plug-In-Projekts „ArduinoCLIUtilizer“ aus Kapitel 6.2 „Integration der ArduinoCLI“ und der MUML-Plug-Ins, mit denen Stürner [TODO: Verweis] gearbeitet hat oder die von ihm erstellt wurden, zu nutzen als auch eigene Teile für die Verwendung von anderen Plug-Ins bereitzustellen. Bei der Implementierung sind jedoch Probleme, die nicht im Rahmen dieser Ausarbeitung gelöst werden konnten, aufgetreten und deshalb werden in diesem Kapitel auch die vorgenommenen Improvisationen bzw. Workarounds beschrieben und erklärt. Die in diesem Kapitel beschriebenen Implementierungen sind auf die beiden Eclipse-Plug-In-Projekte „MUMLACGPPA“ (für „MUML Arduino Code Generation and PostProcessing Automatisation“) und „PipelineExecution“ aufgeteilt, müssen jedoch für die volle Funktionalität zusammen verwendet werden. Die Aufteilung in zwei Eclipse-Plug-In-Projkte hat den folgenden Grund: Die Workarounds sollten so weit wie möglich in dem separaten Projekt „PipelineExecution“ gebündelt werden, damit diese später bei einer Verbesserung von „MUMLACGPPA“ mit einer ordentlichen Implementierung der Problemstellen leichter entfernt werden können.

Dieses Kapitel ist in folgende Abschnitte unterteilt:
In 7.3.1 [TODO: Verweis] werden die aufgetretenen Probleme beschrieben und deren Improvisationen erklärt. Zusätzlich wird der Aufbau der umgesetzten Implementierung mit den Workarounds beschrieben.
In 7.3.2 [TODO: Verweis] wird die Implementierung das Eclipse-Plug-In-Projekts „PipelineExecution“ erläutert.
In 7.3.3 [TODO: Verweis] wird die Implementierung des Eclipse-Plug-In-Projekts „MUMLACGPPA“ bzw. des eigentlichen Systems der Pipeline, also z.B. das Interpretieren der Schritte oder die Verwaltung der Variablen und deren Daten, erklärt.
In 7.3.4 wird auf die Export-Änderung an einem bestehenden MUML-Plug-In „mechatronicuml-cadapter-component-container“ eingegangen.

7.3.1 Aufgetretene Probleme und Improvisationen:

Im Laufe des Implementierungsprozesses traten verschiedene Probleme auf, die weder direkt gelöst noch umgangen werden konnten. Deshalb musste improvisiert werden, damit dennoch die beabsichtigten Ziele bzw. Fähigkeiten der Automatisierung umgesetzt werden konnten.
Folglich wurde die ursprünglich geplante Implementierung angepasst. Die beiden Fälle, die auftraten, und die jeweilige Improvisation werden in den folgenden Abschnitten beschrieben.

7.3.1.1: Gescheitertes Verstehen und Nachahmen der automatischen Auswahlen bei der Kontainer-Transformation und der Komponentencodes:

Mangels weitreichenderen Kenntnissen bei der Export-Plug-In-Programmierung, den Arbeitsweisen der MUML-Plug-Ins und verfügbarer Zeit für das hierfür erforderliche ausführliche Einarbeiten konnten keine automatisierbaren Ressourcenauswahlen der Systemallokationen und Komponenteninstanzkonfigurationen aus den MUML-Dateien entwickelt werden.
Deshalb wurde beschlossen, von den gegebenen bereits automatisch erfolgenden Auswahlen in den Export-Wizards „Container Model and Middleware Configuration“ für die Kontainer-Transformation und „Source Code_Workspace“ für die Generation der Komponenten-Codes Gebrauch zu machen, indem aus deren Quellcode ein Export-Wizard namens „PipelineExecutionAsExport“ und die Hilfsklassen „ContainerTransformationImprovisation“, „ComponentCodeGenerationImprovisation“ und „ContainerCodeGenerationImprovisation“ erstellt wurden.

So können die MUML-Werkzeuge ausgeführt und auch die Ausführung der Pipeline verwaltet werden.
Für mögliche spätere Entwicklungen und Problembehebungen wurde jedoch beschlossen, mit Ausnahme des Startens der Pipeline und der Verwendung der MUML-Werkzeuge dennoch das „MUMLACGPPA“ separat zu belassen und so viel wie möglich vollständig zu implementieren und selbstständig die jeweilige Aufgabe ausführen lassen zu können.
Folglich wurde der Export-Wizard „PipelineExecutionAsExport“ in „PipelineExecution“ darauf konzipiert, aus „MUMLACGPPA“ das eigentliche Pipelinesystem für das Interpretieren der Pipelinekonfigurationen und dessen Bereitstellen der auszuführenden Schritte zu nutzen. So übernimmt er bei Schritten, die für ihre geplante Funktionsweise eine Improvisation erfordern, deren beabsichtigte Funktionsweise.
Bei den Pipelineschritten, die ihre Arbeit selber durchführen können, lässt „PipelineExecutionAsExport“ diese ihre Aufgabe eigenständig erfüllen.

7.3.1.2: Blockiertes Anzeigen von Nachrichten- und Text-Fenstern:
Während der Implementierung von Export-Wizard „PipelineExecutionAsExport“ und dabei erfolgten Tests stellte sich heraus, dass das Erstellen von z.B. Nachrichtenfenstern weder von den Klassen der Pipelineschritte noch von dem programmierten Export-Wizard möglich ist. Bei allen Versuchen zeigte Eclipse eine Warnmeldung mit dem Titel „Internal error“ und dem Text „Invalid thread access“ an und brach die weitere Ausführung ab.
Möglicherweise kann dieses Problem gelöst werden, aber aus Zeitmangel wurde beschlossen, die Nachrichten und Texte in der Konsole ausgeben zu lassen und beim Starten der Pipeline auf diesen Umstand hinzuweisen.

7.3.1.3: Ausführungsprobleme bei Pfadfindung: [(((Neu)))]
Während der Programmierung der Plug-Ins schlug das Heraussuchen des gesuchten Datei über Anweisungen wie

```
IWorkbenchWindow window = PlatformUI.getWorkbench().getActiveWorkbenchWindow();
		IStructuredSelection selection = (IStructuredSelection)
window.getSelectionService().getSelection();
		final IFile selectedFile = (IFile) ((IStructuredSelection) selection).getFirstElement();
Path result = Paths.get(selectedFile.getLocation().toString());
```

bizarrerweise fehl, wenn durch „MUMLACGPPA“ etwas aus „ArduinoCLIUtilizer“, das die ausgewählte Datei heraussucht, verwendet wird. Für einzelne verwendete Dateien können die jeweils verantwortlichen „Action“-Klassen das Heraussuchen zuverlässig übernehmen und die Information als Parameterbelegung weiter geben. Für größere Unternehmungen hingegen wird der Ordnerpfad des aktiven Projekts benötigt. Also wurde beschlossen, diesen global zugänglich zu speichern. Dies wurde über das Eclipse-Plug-In „ProjectFolderPathStoragePlugIn“ mit seiner Klasse „ProjectFolderPathStorage“ mit dem statischen Feld „projectFolderPath“ vom Typ Path ermöglicht. Jede im Rahmen dieser Ausarbeitung erstellte funktionale „Action“-Klasse ermittelt den Projektorderpfad und legt diesen in dem statischen Feld dafür ab. So können mehrere Komponenten gleichzeitig diesen abrufen und mit den jeweiligen Dateien arbeiten. Später wurde für den Typ „IProject“ das statische Feld „project“ hinzugefügt.
[(((Ende neu.))))]

7.3.2: Implementierung der Pipelineausführung:
Es wurde das Eclipse-Plug-In-Projekt „PipelineExecution“ angelegt, das den Export-Wizard „PipelineExecutionAsExport“ enthält. Dieser ist darauf konzipiert, aus „MUMLACGPPA“ das eigentliche Pipelinesystem bzw. dessen Konfigurationslesesystem mit einer Konfigurationsdatei zu instanziieren, deren Konfigurationen validieren zu lassen und nacheinander die auszuführenden

Schritte anzufordern. Dies erfolgt innerhalb der Funktion „addPages()“, also beim Aufbauen der verschiedenen Fenster, wie sie normalerweise für das Konfigurieren von Exportvorgängen verwendet werden. Zu jedem erhaltenen Schritt soll „PipelineExecutionAsExport“ die Ausführung der jeweiligen beabsichtigten Funktionsweise sicherstellen. Abseits der Ausführung von der Pipeline wurden für die Nutzerschaft Informationsmeldungen entworfen, die nach Notwendigkeit angezeigt werden, und es wurde eine mögliche Optimierung vorgenommen.

7.3.2.1: Pipelineschritte mit improvisierter Durchführung der jeweiligen Funktion:
Bei den Pipelineschritten, die nicht eigenständig ihren Zweck erfüllen können, übernimmt „PipelineExecutionAsExport“ die jeweilige Arbeit.
Für beispielsweise ContainerTransformation, das der Kontainer-Transformation entspricht, erstellt „PipelineExecutionAsExport“ das aus dem Originalcode übernommene Auswahlfenster für die Auswahl des Systemallokationseintrages aus der Datei „roboCar.muml“, aber die weiteren Daten für die Middleware der Kommunikation bzw. den Ansatz für die Kommunikation und für den Speicherort des zu generierenden Kontainers werden aus den gegebenen Konfigurationen übernommen und bei der Nutzung von ContainerGenerationJob aus den MUML-Werkzeugen eingetragen.

7.3.2.2: Starten von funktionierenden Pipelineschritten:
Bei den Pipelineschritten, die ihre Arbeit selber durchführen können, lässt „PipelineExecutionAsExport“ sie ihre jeweilige Aufgabe eigenständig erfüllen, indem es bei diesen Instanzen jeweils die Ausführungsfunktion „execute()“ aufruft.

7.3.2.3: Situationsabhängige Informationsmeldungen:
Um bei fehlenden oder fehlerhaften Konfigurationsdateien die Ausführung der Pipeline nicht zu starten, sondern stattdessen einen Hinweis auf den entsprechenden Missstand anzuzeigen, übernimmt „PipelineExecutionAsExport“ auch das Überprüfen der Konfigurationsdateien auf deren Existenz. Wenn aber beispielsweise keine Schritte aufgespürt werden, die auf einer Nutzung von „ArduinoCLIUtilizer“ beruhen, so wird weder nach dessen Konfigurationsdatei gesucht noch ihre Existenz erfordert. Zusätzlich wird die Überprüfung der Pipelinekonfigurationen gestartet. Wenn hierbei ein Fehler aufgespürt wird, so wird dieser gemeldet. Nur wenn alle Prüfungsschritte keinen Fehler finden konnten, kann die Pipeline gestartet werden. Weitere Details zu dem System der Pipeline und den Korrektheitsprüfungen werden in 7.3.3 erläutert.

7.3.2.4: Weitere Varianten:
Die Varianten für das jeweilige alleinige Ausführen der MUML-Werkzeuge und der Post-Processing-Schritte funktionieren ähnlich, wenn auch auf die jeweilige Aufgabe reduziert. Hierfür erben sie von „PipelineExecutionAsExport“, verwenden jedoch nur die jeweils relevanten Funktionen, verwenden von diesen jedoch nur, was jeweils relevant ist. Zusätzlich sind jeweils der Aufbau und die Folge der verschiedenen Fenster sowie der interne Ausführungsablauf auf den entsprechenden Zweck angepasst bzw. vereinfacht.

7.3.3: Implementierung des eigentlichen Systems der Pipeline:
Das eigentliche Pipelinesystem wurde als das Eclipse-Plug-In-Projekt „MUMLACGPPA“ implementiert und im Folgenden wird auf seine Konzepte und Teile eingegangen.

7.3.3.1: Variablen- und Werte-Verwaltung

Die Klasse „VariableHandler“ wurde dafür entworfen, die Variablen und deren Nutzung wie dem Festlegen von Werten zu verwalten. Hierbei werden eine Liste mit initialisierten Variablennamen und zwei Tabellen, die zu den verschiedenen Namen entsprechend den jeweiligen Typ oder Inhalt speichert, verwendet.
In Folge von mangelnden Kenntnissen mit Generics unter Java und gescheiterten Implementierungsversuchen sind die Variableninhalte selbst Instanzen der Klasse „VariableContent“. Diese sieht das Speichern von Daten als String bzw. Zeichenkette vor und stellt für das Lesen bzw. Abrufen die Getter-Funktionen mit den Umwandlungen für die verschiedenen Datentypen bereit.
Weil das globale Zugreifen auf die Variablenverwaltung als gefährlich bewertet wurde, wird beim Lesen der Konfigurationsdatei ein „VariableHandler“ angelegt und jeder Pipelineschritt erhält bei seiner Instanziierung eine Referenz darauf.

7.3.3.2: Konzept hinter den Pipelineschritten:
Zu den einzelnen Typen von Pipelineschritten wurden verschiedene Klassen mit der jeweiligen Funktionalität geschrieben. Diese erben jedoch gemeinsame Funktionen von der abstrakten Klasse „PipelineStep“. So wird durch das Prinzip des Polymorphismus die Nutzung vereinfacht, indem z.B. eine mit eben dieser Klasse als Datentyp festgelegte Liste alle Pipelineschritte speichern kann. Beim Iterieren über diese (Liste) kann bei allen darin enthaltenen Schritten die Ausführung leicht aufgerufen werden.
Die gemeinsamen Funktionen behandeln die Instanziierung, das interne Festlegen der zu konfigurierenden Parameter, das Überprüfen der eigenen Einstellungen, das Lesen und Schreiben von Variablen, das Interpretieren von Pfadangaben sowie das Ausführen der jeweiligen Aufgabe.

[(((Muss ich hier überhaupt bei allen noch weiter ins Detail gehen?)))]

„getRequiredInsAndOuts()“ ist erforderlich, weil innerhalb der verbliebenen Zeit keine Möglichkeit gefunden wurde, wie über statische Felder innerhalb der Unterklassen von „PipelineStep“ der jeweilige erwartete Konfigurationsaufbau festgelegt werden kann. Mehr zu den Überprüfungsabläufen wird in 7.3.3.4 „Überprüfung der Korrektheit der Pipelinekonfigurationen“ ([TODO: Verweis]) beschrieben.
Es ist aber zu Beachten, dass „getContentOfInput(String key)“, alle Varianten von „setContentOfOutput“ und „getResolvedPathContentOfInput(String key)“ Hilfsfunktionen sind, die wegen der Improvisation bei der Nutzung von MUML-Werkzeugen per Export-Wizard (siehe 7.3.1 „Aufgetretene Probleme und Improvisationen“) hinzugefügt wurden.

7.3.3.3: Interpretation der Pipelinekonfiguration:
Zuerst wird im Konstruktor die Konfigurationsdatei geladen, die daraufhin interpretiert wird. Hierbei werden die Bereiche für die Einstellungen in folgender Reihenfolge interpretiert:

- Zuerst „VariableDefs“, um die Variablen und ihre Werte zu erhalten.
- Dann „TransformationAndCodeGenerationPreconfigurations“, um für die einzelne vorkonfigurierte Ausführung von MUML-Werkzeugen die Einstellungen zu erhalten, falls sie von späteren Teilen der Konfigurationsdatei geladen werden.
- Danach „PostProcessingSequence“. So können einzelne Einstellungen aus „TransformationAndCodeGenerationPreconfigurations“ geladen werden, aber in „PipelineSequence“ kann es komplett eingefügt werden. Hierbei ist das Einbinden von einzelnen Schritten aus der Post-Processing-Sequenz zwecks Zuverlässigkeit bei Änderungen nicht erlaubt.
- Zuletzt „PipelineSequence“. So können sowohl einzelne Einstellungen aus „TransformationAndCodeGenerationPreconfigurations“ als auch die komplette Schrittliste von „PostProcessingSequence“ mit eingebunden werden.

Es ist wichtig zu erwähnen, dass alle Versuche, zu den eingestellten Pipelineschritten die jeweiligen Klasse anhand des angegeben Namens herauszusuchen und zu instanziieren, fehlgeschlagen sind. Deshalb wird sozusagen mit einem Wörterbuch gearbeitet, also „PipelineStepDictionaryMUMLPostProcessingAndArduinoCLIUtilizer“. Dieses führt mittels einer langen „switch … case ...“-Abfrage jeweils das Heraussuchen und Instanziieren der erforderlichen Unterklasse von „PipelineStep“ durch. Um für eigene Zwecke leichter eigene Klassen verwenden zu können, wurde eine Superklasse „PipelineStepDictionary“ angelegt, sodass leichter eigene Implementationen für das Heraussuchen der Klassen erstellt und diese in dem Konstruktor für eine „PipelineSettingsReader“-Instanz eingetragen werden können.

[(((Evtl. noch das Lesen von Daten, die nicht als String dastehen, erläutern. Oder nicht?))))]

7.3.3.4: Überprüfung der Pipelinekonfigurationen auf Fehler:
Vor dem möglichen Ausführen der Pipeline wird von „PipelineExecutionAsExport“ oder einer ihrer Unterklassen eine Fehlersuche gestartet. Falls ein Konfigurationsfehler gefunden wird, wird zu diesem eine Meldung ausgegeben und die Pipeline wird nicht gestarten.
Die implementierten Mechanismen der Fehlersuche können folgendes aufspüren:

- Fehlende Konfigurationsdateien
- Syntax-Fehler
- Typen-Fehler
- Versuchte oder versehentliche Typenänderungen bei Variablen
- unerlaubte Strukturen wie z.B. „from direct“ oder sonstige Strukturverstöße
- Versuchtes rekursives Einbinden der Einträge aus „PostProcessingSequence“ in „PostProcessingSequence“, d.h. nur in „PipelineSequence“ darf der Inhalt aus „PostProcessingSequence“ eingebunden werden.
- fehlplatzierte oder falsche Schlüsselwörter
- fehlende oder überschüssige Parameterangaben
- fehlerhafte Werteangaben
- Verwendung von nicht vorhandenen oder noch nicht initialisierten Variablen
- Verwendung von nicht vorhandenen Einträgen aus „TransformationAndCodeGenerationPreconfigurations“
- Unerlaubte Pipelineschritttypen in „TransformationAndCodeGenerationPreconfigurations“
- [(((Hoffentlich habe ich nichts vergessen.))))]

Die ersten drei Einträge dieser Liste erfolgen teils durch die verwendete YAML-Bibliothek „Snakeyaml“ (Version 2.2) und teils durch das Prüfen der Strukturen und Inhalte beim Interpretieren.
Die Fehlersuche beim Datenfluss erfolgt hinsichtlich des Existierens oder Fehlens von Variablen, der Typenprüfung und des versuchten oder versehentlichen Typenänderns. Der logische Pipelineablauf wird auch beachtet. Eine Variable namens „ifSuccessfulContainerTransformation“, die dynamisch in dem Bereich „TransformationAndCodeGenerationPreconfigurations“ in „ContainerTransformation“ erstellt wird, ist in dem Bereich „PipelineSequence“ erst nach dem Eintrag „from TransformationAndCodeGenerationPreconfigurations: ContainerTransformation“ verfügbar.
Hierbei wird ebenfalls die Überprüfung der Parameterangaben bei den verschiedenen Instanzen der den konfigurierten Pipelineschritten entsprechenden Klassen gestartet. Wenn nicht alle der erwarteten Parameter der jeweiligen Klasse oder unerwartete Einträge gefunden werden, so wird dies als Fehler gemeldet und dabei ein Vergleich der jeweils angegebenen und der erwarteten Einträge angezeigt.

7.3.3.5: Generierung der Beispiele:
Um der Nutzerschaft den Einstieg in die Nutzung dieses Pipelinesystems zu erleichtern, wurden zwei Dateigeneratoren für eine Liste mit Beispielkonfigurationen aller Pipelineschritte und für eine auf den Anwendungsfall dieser Ausarbeitung ausgerichtete Beispiel-Pipelinekonfiguration konzipiert.
Für das erste erfolgt der Aufruf per Rechts-Klick auf „roboCar.muml“]/“MUMLACGPPA“/“Generate all pipeline step example settings“ und dann wird eine Textdatei mit allen Pipelineschritten und direkten Beispielbelegungen der Parameter generiert.

Die Generierung der Beispiel-Pipelinekonfiguration wird per Rechts-Klick auf „roboCar.muml“]/“MUMLACGPPA“/“Generate default/example pipeline settings“ gestartet. Hierbei werden jedoch für die Seriennummern der zu verwendenden Boards, WLAN-Informationen und MQTT-Informationen keine reellen Daten, sondern Dummy-Werte eingetragen. Diese fangen mit „Dummy“ an und ein Informationsfenster weißt auf diesen Umstand hin. So können die zu ersetzenden Dummy-Werte mit einer Suche nach „Dummy“ leicht und schnell gefunden werden.

7.3.3.6: Generierung einer lokalen SofdCar-Hal-Konfigurationsdatei:
Die Bibliothek „Sofdcar-HAL“ erfordert u.a. Konfigurationen, die in einer als Konfigurationsdatei dienenden Codedatei „Config.hpp“ eingetragen sind. Der ursprüngliche Post-Processing-Ablauf [Link] sah hierfür das Herunterladen von ihr aus dem Repository [Link] und dem Platzieren von ihr in jedem „*CarDriverECU“-Ordner vor. Wegen möglicher Versionsunterschiede zwischen [TODO: Formulierung verbessern] den Dateien in der Versionsverwaltung und der lokalen Kopie kann dies beispielsweise Kompilierfehler oder unerwartetes Verhalten verursachen. Deshalb wurde ein zu dem Herunterladen alternativer Schritt konzipiert, durch den stattdessen eine lokale Instanz aus dem Ordner „automatisationConfig“ kopiert werden kann. Zur leichteren Nutzung wurde ein Generator für diese Konfigurationsdatei konzipiert, der per per Rechts-Klick auf „roboCar.muml“]/“MUMLACGPPA“/“Generate local settings for Sofdcar-Hal“ gestartet werden kann. Für das intuitive Verständnis dieser Konfigurationsdatei wird sie innerhalb von „automatisationConfig“ unter dem Namen „LocalSofdcarHalConfig.hpp“ aufgelistet, aber ihre Kopien an den Zielorten unter dem korrekten Namen „Config.hpp“.

7.3.4: Export-Änderung an MUML-Plug-In „mechatronicuml-cadapter-component-container“
Es wurde die Klasse „GenerateAll“ aus dem Package „org.muml.arduino.adapter.container.ui.common“ aus dem MUML-Plug-In-Projekt „mechatronicuml-cadapter-component-container“ bzw. dessen Unter-Projekt „org.muml.codegen.componenttype.export.ui“ benötigt. Deshalb wurde in den Einstellungen von diesem Unterprojekt das genannte Package in den Plug-In-Einstellungen per „Exported Packages“ als nach außen sichtbar eingestellt. So kann es für die automatiserte Generation des Kontainercodes verwendet werden.