4.2 Auswertung bestehender Ansätze für die Integration von Arduinosoftware und Automatisierung in Eclipse

Um einen Überblick über mögliche bereits vorhandene Lösungsansätze für die Verwendung von Arduinocode innerhalb der Eclipse-IDE, die OTA-Programmierung der Mikrokontroller und die Automatisierung von Abläufen mit Eclipse dieser Ausarbeitung zu erhalten, wurden jeweils Recherchen durchgeführt. In diesem Kapitel wird beschrieben, welche Ansätze gefunden wurden und aus welchen Gründen stattdessen die in den folgenden Kapiteln (siehe [TODO: Verweise auf 7. „Integration der Arduino-CLI“ und 8. „Over-the-Air-Programmierung der Arduino-Mikrocontroller“]) Umsetzungen durchgeführt wurden.

Ansätze für die Verwendung von Arduinocode innerhalb der Eclipse-IDE

Um das Verlassen der Eclipse-IDE und das Wechseln in die Arduino-IDE zu vermeiden, wurde im Vorfeld nach einer Möglichkeit für das Arbeiten mit Arduinocode und den passenden Mikrokontrollern durchgeführt. Wegen der in der Konsole der Arduino-IDE aufgelisteten Meldungen und Befehle wurde zunächst die Sucheingabe „Arduino command line“ verwendet und so die Arduino-CLI gefunden. Diese ist eine offizielle Software von Arduino ist und wurde wegen ihrer Automatisierbarkeit per programmatischer Nachstellung des Terminals bzw. Kommandozeile als leicht automatisierbar erkannt. In [TODO: Verweis auf 7.1 „Über die Arduino-CLI“] wird sie näher beschrieben werden.
Es wurde auch eine Suche nach bereits vorhandenen Möglichkeiten für das Arbeiten mit Arduinocode in Eclipse zum Gegenvergleich mittels der Eingaben „eclipse plug in arduino“ und „eclipse arduino“ durchgeführt und hierbei wurde das Plug-In „Eclipse C++ Tools for Arduino 2.0“ gefunden. Aufgrund der leichten Testbarkeit und Neugierde wurde dieses bereits während des Proposals zu dieser Ausarbeitung getestet. Bei diesen Vorabtests ist jedoch ihre Installation gescheitert, weil nicht mehr alle der erforderlichen Bibliotheken von dem Plug-in-Installer gefunden werden konnten.

Ansätze für die OTA-Programmierung der Mikrokontroller

Die in Unterkapitel 4.2 „Over-the-Air-Programmierung“ aufgelisteten Ansätze wurden getestet, um praktische Erfahrungen mit diesen zu sammeln und nicht funktionierende Ansätze herauszufiltern, sodass nur die funktionierenden Möglichkeiten genauer betrachtet und miteinander verglichen werden. Hierbei hat sich jedoch gezeigt, dass keiner der drei Ansätze funktioniert hat. Deshalb werden im folgenden alle Ansätze und die jeweiligen fatalen Probleme aufgelistet und beschrieben:

Ansatz 1: Herstellen einer Verbindung über das Internet nach „On Flashing Over The Air “FOTA” for IoT Appliances – An ATMEL Prototype“:

In diesem Fall konnte der Sourcecode, der für den programmierenden Mikrokontroller verwendet wird, nicht gefunden werden. Zusätzlich erfordert die verwendete Software Netburner, die nur mit einem Account und einer Seriennummer heruntergeladen werden kann. Dies stellt eine hohe Bindung an eine einzelne Firma dar und falls sie diese Software irgendwann einstellt, so würde eine mögliche auf ihr basierende Umsetzung nicht mehr funktionieren. Zusätzlich würde im Kontext des autonomen Fahrens außerhalb von Simulationen ein Updatevorgang auf dem eigenständigen Herunterladen und Installieren erfolgen. Insgesamt stehen die Faktoren gegen diesen Ansatz, dass weitere Versuche, ihn zu nutzen, abgebrochen wurden.

Ansatz 2: Anwendung der Firmware esp-link
Für diesen Ansatz wurde geplant, die esp-link-Firmware auf einem ESP8266 zu installieren, per Browser unter Beobachtung der Kommunikation den OTA-Update-Vorgang durchzuführen und anhand einer passenden Bibliothek von Java die Kommunikation passend für den Anwendungsfall nachzustellen. Bei einem Funktionstest hat sich jedoch gezeigt, dass Konfigurationswebseite, die dabei benutzt werden musste, nicht aufgerufen werden konnte, sondern blank blieb. Es wurden mehrere Versuche durchgeführt, um dieses Problem zu beheben, aber keiner davon war erfolgreich. So konnte also nicht garantiert werden, dass diese alternative Firmware korrekt arbeitet und deshalb wurde dieser Ansatz verworfen.

Ansatz 3: Eigenständiges Herunterladen und Installieren nach dem Ansatz von SYSTEMS O.R.P [Ansatz von SYSTEMS O.R.P]:
Bei dieser Möglichkeit ist der Funktionstest bei einem der Aufbauschritte gescheitert: Der ESP8266 wurde mit dem Arduino-Uno verbunden und beim Prüfen der Verbindung per AT-Befehl konnten nur unleserliche Zeichen erhalten werden. Auch weitere Versuche mit einem anderen ESP8266, neuer Installation aktueller offizieller Firmware, einem anderen Arduino-Uno und verschiedenen Programmierungen für diesen (den Arduino-Uno) konnten dieses Problem nicht lösen. Deshalb wurde auch dieser Ansatz für gescheitert befunden.

Auswertung:
Die Forschungsziele FZ4.1 „Welche Kriterien sind für die Auswertung von OTA-Programmierungsansätze relevant?“, FZ4.2 „Welche Anpassungen an den Modellautos und deren Verhaltensmodellen sind erforderlich?“ und FZ4.3 „Welcher OTA-Programmierungsansatz erfüllt die Kriterien am besten und wird implementiert werden?“ wurden mangels einer Auswahl bzw. Möglichkeiten für die Umsetzung verworfen und die Aufgaben „A4.1 Definieren von Kriterien für das Auswerten von OTA-Programmierungsansätzen.“, A4.2 Erforschung und Auswertung der Hardware und Verhaltensmodelle der Modellautos.“ und A4.3 „Auswerten der OTA-Programmierungsansätze und Auswählen des zu implementierenden Ansatzes.“ können nicht durchgeführt werden. Als Kompensation wird das Programmieren über die typischen Verbindungsmöglichkeiten für Geräte wie USB-Kabel erfolgen.

Ansätze für die Automatisierung von Abläufen mit oder innerhalb der Eclipse-IDE:

Für dieses Ziel wurde zunächst nach bereits vorhandenen Möglichkeiten für das Automatisieren von Abläufen mit Eclipse gesucht und dann auf ihre Durchführbarkeit getestet. Hierbei wurden offizielle Ansätze priorisiert, weil sie offiziell unterstützt werden und nicht von Drittanbietern stammen. Die Automatisierungsmöglichkeiten von Drittanbietern wurden ursprünglich nur als als Reserve für den Fall, dass keine offizielle Lösung funktioniert, vorgesehen, aber wegen der schlechten Resultate zu EASE (siehe [TODO: Verweis]) doch getestet. An dieser Stelle sei auch erwähnt, dass eingestellte Software nicht als Ansatz zugelassen wurde wurde. So soll weder eine weitere Hürde gegen einen möglichen Umstieg auf aktuelle Eclipse-Versionen geschaffen werden noch das Risiko von möglicherweise gelöschter oder aus sonstigen Gründen nicht mehr installierbaren Automatisierungswerkzeugen in das Ergebnis dieser Ausarbeitung eingebracht werden.
Durch die in diesem Abschnitt bzw. im Folgenden beschriebenen Analysen und Auswertungen werden folgende Forschungsziele beantwortet:
FZ6.2: Welche Kriterien sind für die Auswertung der Build-Automatiserungsmöglichkeiten für Eclipse relevant?
FZ6.3 Welche der Build-Automatiserungsmöglichkeiten für Eclipse erfüllt die Kriterien am besten?

Offizielle Automatisierungsansätze:

###EASE [EASE, https://eclipse.dev/ease/downloads/]:

Die Eclipse Advanced Scripting Environment, kurz EASE, beruht auf der Interpretation von Scripten durch eine Java-Runtime innerhalb einer laufenden Eclipse-Instanz [EASE]. Es stellt also praktisch gesehen nur eine Entwicklungsumgebung für ein leichteres Heraussuchen von zugreifbaren Teilen der Eclipse-IDE und das Aufrufen von diesen dar.
Die Automatisierung von Vorgängen in Eclipse erfolgt genauer gesagt übe die Skript-Module, die die Skripte enthalten [Foub]. Verschiedene Sprachen wie Java werden mittels verschiedener Script-Engines unterstützt, wobei Entwickler eigene Engines schreiben können [Foub]. EASE verfügt nicht über die Fähigkeiten einer per Konfigurationsdatei einstellbaren Pipeline.
Es wird auf der Webseite von TEA [Tea] erwähnt und wurde deshalb zwecks Vollständigkeit auch überprüft. Aufgrund der verwendeten „Neon“-Edition der Eclipse-IDE wurde hierfür die Version 0.4.0 verwendet [EASE].
Bei der Ausführung bzw. Interpretation von diesen kann es auch auf Java-Klassen und Teile der Eclipse-IDE zugreifen [Foua]. Recherchen der Dokumentation und Tests haben gezeigt, dass es von sich aus keinen direkten Zugriff auf die Inhalte von Projekten und Plug-Ins erhalten kann, weil es bei diesen entweder deren Integration als Modul oder das Erstellen von Wrappern auf Instanzen einer Javaklasse mittels wrap([Instanz der Java-Klasse]) [https://eclipse.dev/ease/documentation/writing_modules/]. Die Nutzung ist jedoch in beiden Fällen auf Schnittsstellen mit der Sichtbarkeit public beschränkt.
Folglich müssen bei existierendem Java-Code entweder entweder Module deklariert und Wrapper eingefügt werden oder bei jeder Instanz einer Klasse aus diesen wrap([Instanz]) verwendet werden. Dies wiederum bedeutet, dass bestehender Code hinsichtlich der Sichtbarkeiten und der von außen aufrufbaren Funktionen angepasst werden müsste.
Bei Nutzung wären jedoch in beiden Fällen Aufrufe und Programmierungen analog zu den Interaktionen der Plug-Ins untereinander erforderlich.
Möglichkeiten für automatisierte Zugriffe auf GUI-Elemente wie bei den Exporten, die bei der Automatisierung von diesen eine große Zeitersparnis darstellen würden, konnten nicht gefunden werden. Die hierfür verwendeten Suchanfragen lauteten „eclipse ease automate export“, „eclipse ease export“, „eclipse ease access ui“ und „eclipse ease access ui“.
Beim Arbeiten mit Eclipse-Plug-In-Projekten stellt es folglich praktisch gesehen nur einen Umweg für programmatische Aufrufe bereit.
Aufgrund dieser Problematik und der oben beschriebenen Schwächen ist Ansatz eine prinzipiell funktionierende, aber umständliche Basis für eine Automatisierung dar.

TEA [https://eclipse.dev/tea/index.php]:

TEA ist ein offizielles Set aus Erweiterungen für die Eclipse-IDE, deren Hauptaufgabe die Automatisierung von Aufgaben oder gar Aufgabenketten mit und ohne Nutzung der grafischen Oberfläche ist. Im Gegensatz zu EASE soll es komplexere Anwendungen besser umsetzen können. Hierfür werden die Aufgaben gewöhnlich als Eclipse-Plug-Ins geschrieben.
Es kann auf Teile der IDE zugreifen und auch ohne einem Workspace arbeiten. Es wird auch eine Bibliothek bereitgestellt, die beispielsweise auch zu Jenkins kompatibel ist. Bei diesem Ansatz hat das Alter von Eclipse Neon deutliche Auswirkungen auf die Installierbarkeit: Offizielle automatische Installationsmöglichkeiten, die eine Eclipse-Installation mit TEA vorbereiten konnten, funktionieren nicht mehr, weil nicht mehr alle von deren verwendeten Adressen die alten Dateien bereitstellen können. Beispielsweise muss man für das optionale LcDsl nicht den Link https://mduft.github.io/lcdsl-latest/ aufrufen, sondern https://mduft.github.io/lcdsl-leagacy/.

Wichtige Tauglichkeitstests von TEA schlugen fehl, z.B. weil die Erkennung von Bibliotheken nicht funktionierte. Deshalb wurde auch dieser Ansatz für diese Ausarbeitung verworfen, aber er könnte bei neueren Versionen bzw. Editionen von Eclipse korrekt funktionieren.

Ansätze von Drittanbietern:

###Apache Ant bzw. Ant:
Ant ist gleichermaßen eine Bibliothek und Kommandozeilenwerkzeug, das anhand von Konfigurationsdateien Prozesse steuert [Fouf]. Es wird hauptsächlich für Java-Anwendungen verwendet, kann aber auch außerhalb von Java jeglichen Typ von Prozess steuern, solange dieser in Form von Zielen und Tätigkeiten beschrieben werden kann. Es ist auch möglich, Erweiterungen in Form von "antlibs" zu programmieren.
Bei der Nutzung von Ant können .jar-Dateien und deren Methoden aufgerufen werden. Beim Durchgehen der Dokumentation konnten jedoch keine Möglichkeit für das Aufrufen von Inhalten von unkompilierten java-Dateien gefunden werden. Dies ist für die Ziele dieser Arbeit wichtig, weil die Plug-In-Projekte der MUML-Erweiterungen eine lange Liste an Errors und Duplikaten (durch das „alles importieren“-Vorgehen bei den MUML-Plug-Ins gibt es nicht nur die org.*-Projekte, sondern auch die Projekt-Ordner, die diese ebenfalls enthalten) haben. Alle Versuche, sie zu .jar-Dateien zu kompilieren, schlugen fehl.

###Apache Maven, kurz Maven [Foug] [Foue]: Maven wird als „Softwareprojektmanagement- und Verständnis-Werkzeug“ beschrieben. Es kann Build-Prozesse, Reports und Dokumentationen managen. Zusatzlich kann es Tätigkeiten von Ant verwenden, mit mehreren Projekten gleichzeitig arbeiten und verschiedene Versionsverwaltungsysteme für den Quellcode integrieren. Benötigte Bibliotheken können auch automatisch heruntergeladen werden. Eigene Erweiterungen werden als Maven-Plug-Ins geschrieben. Zur Konfiguration werden mehrere Dateien verwendet [Foud]. Jedoch wird auch hier kompilierter Javacode, also .jar-Dateien, benötigt, woran auch der vorherige Ansatz gescheitert ist.
Bei Versuchen mit m2e-extras konnten auf http://ifedorenko.github.com/m2e-extras/ keine Daten gefunden werden und bei der Installation über https://download.eclipse.org/technology/m2e/releases/latest/ oder einer der dafür automatisch erfolgten Installation von benötigten Daten wurde etwas aus der gegeben Eclipse-Installation für MUML entfernt oder ersetzt, weil Teile davon ihre Ausführung mit Nullpointer-Fehlern (Nullpointer exceptions) abbrachen. Die Kommunikation mit den Download-Seiten und die Installations-Historie funktionierten nicht mehr und evtl. wurde auch mehr gestört. Die Fehlfunktionen konnten nicht behoben werden, also wurde die gestörten Eclipse-Installation durch ein Backup ersetzt.
Somit ist auch Maven keine mögliche Basis für die Automatisierung per Pipeline.

###Gradle:
Gradle Build Tool, kurz Gradle: Dies ist ein Abhängigkeitenverwaltungs- und Build-Prozess-Automatisierungs-Werkzeug, das fär das Festlegen der Build-Prozesse in den Konfigurationsdateien entweder Groovy oder Kotlin verwendet [Bae]. So können manche Domänenprobleme gelost werden [Bae]. Der Großteil der Funktionalität wie die Verwendbarkeit für z.B. Java erfolgt durch Plug-Ins [Bae] und es ist auch möglich, eigene Plug-Ins zu entwickeln [Inca]. Gradle kann auch mit mehreren Projekten gleichzeitig

arbeiten [Inca]. Laut Angaben in dem Benutzerhandbuch [Incb] soll es durch seine Schnelligkeit und Skalierbarkeit große und komplexe Projekte verarbeiten können.
Beim Durchgehen der Dokumentation konnte keine Möglichkeit für das Nutzen von unkompilierten Eclipse-Plug-Ins bzw. unkompiliertem Javacode gefunden werden. Es wurde nur auf kompilierte Dateien bzw. .jar-Dateien und deren Ausführung hingewiesen.
Somit ist auch dieser Ansatz verworfen worden.

Auswertung:
Von den offiziellen und inoffiziellen Automatisierungsmöglichkeiten wurden alle bis auf EASE verworfen. EASE hat funktioniert, aber die Tests haben gezeigt, dass das Konzept der Eclipse-Plug-Ins eine Konkurrenz darstellt (siehe [TODO: Verweis]). Die reine Eclipse-Plug-In-Nutzung erfordert nämlich keinen Installationsaufwand und erreicht ohne Integrationsaufwand an anderen Eclipse-Plug-In-Projekten dasselbe Potential. Das Hinzufügen von GUI-Elementen und das Durchführen von Aktionen bei Verwendung von diesen ist in beiden Fällen möglich. Dies gilt ebenfalls im Hinblick auf das Ermöglichen von CI/CD per pipelinebasierter Automatisierung, weil die Entwicklung eines Systems für die Pipeline selbst bei beiden Ansätzen erforderlich wäre. Die Antwort auf FZ6.2 „Welche Kriterien sind für die Auswertung der Build-Automatiserungsmöglichkeiten für Eclipse relevant?“ sind folglich die Unterschiede, also der Installations- und der Integrations-Aufwand.
In beiden Punkten ist das Entwickeln Eclipse-Plug-Ins überlegen, weil die Installation von zusätzlicher Software bei der vorbereiteten Eclipse-Installation für MUML über das Lubuntu-Image [Link] nicht mehr durchgeführt werden muss und die Integration erfordert beispielsweise keine Wrapper. Dies beantwortet FZ6.3 „Welche der Build-Automatiserungsmöglichkeiten für Eclipse erfüllt die Kriterien am besten?“.
So wurde EASE abgelehnt und und die Entwicklung eines eigenen Plug-Ins, das auf andere Plug-Ins zugreift und so deren Nutzung automatisiert, als Basis für die Automatisierung per Pipeline gewählt.

[(((TODO: Link-Check erfolgte am 7.6.24)))]