Pyduino

[Kurzfassung 1](#_Toc124270427)

[Problem 2](#_Toc124270428)

[Lösungsidee 3](#_Toc124270429)

[Umsetzung 3](#_Toc124270430)

[Transpiler 4](#_Toc124270431)

[Verbindung zwischen PC und Arduino 9](#_Toc124270432)

[VS Code Erweiterung 9](#_Toc124270433)

[Features 11](#_Toc124270434)

[Pyduino 11](#_Toc124270435)

[Programmstruktur 11](#_Toc124270436)

[Variablen 12](#_Toc124270437)

[Arrays 12](#_Toc124270438)

[Listen 13](#_Toc124270439)

[Funktionen 14](#_Toc124270440)

[Builtins 14](#_Toc124270441)

[Kontrollstrukturen 15](#_Toc124270442)

[if-Bedingung 15](#_Toc124270443)

[while-Schleife 15](#_Toc124270444)

[for-Schleife 15](#_Toc124270445)

[Getting Started 16](#_Toc124270446)

[Ergebnisse 16](#_Toc124270447)

[Ausblick 17](#_Toc124270448)

[Dank 17](#_Toc124270449)

[Quellen 17](#_Toc124270450)

# Kurzfassung

In meinem Projekt habe ich eine einfachere Programmiersprache für den Microcontroller Arduino entwickelt. Sie soll mit einer Python-ähnlichen Syntax den Einstieg ins Programmieren erleichtern, da für den Arduino sonst die schwierigere Programmiersprache C verwendet werden muss. Um dies möglich zu machen, habe ich einen Transpiler in Python programmiert, der die Pyduino Syntax in C Syntax übersetzt und auf Fehler überprüft. Dieses C Programm wird dann für den PC und den Arduino kompiliert. So kann die selbe Syntax auf beiden Plattformen ausgeführt werden. Wenn der PC mit dem Arduino verbunden ist, kann der PC auf die Ports des Arduinos zugreifen und der Arduino kann Text in der Konsole des PCs ausgeben, auf Dateien zugreifen und Funktionen aufrufen. Mit Pyduino kann der Arduino vom PC aus gesteuert werden, ohne das Programm jedes mal hochladen zu müssen und selbst auf die Rechenleistung eines PCs zugreifen, was die Einsatzmöglichkeiten des verbreiteten Microcontrollers stark erweitert.

# Problem

Der Arduino ist ein sehr beliebter Microcontroller, mit dem sehr viele die Grundlagen des Programmierens lernen. Auch im NWT-Unterricht in Baden-Württemberg ist er ein wichtiger Bestandteil. Die Popularität des Arduinos hängt damit zusammen, dass er einfach, benutzerfreundlich und universell einsetzbar ist. Auch der Einstieg ist sehr einfach; Innerhalb von wenigen Minuten kann selbst ein Anfänger die ersten Ergebnisse, wie zum Beispiel eine blinkende LED sehen. Der Arduino ist aber trotzdem auch für Fortgeschrittene Anwendungen geeignet. Dabei ist er aber immer noch sehr benutzerfreundlich, vor allem durch die neue Arduino IDE 2.x, die Features wie Syntax Highlighting und Autovervollständigung für die Arduino Programmiersprache zur Verfügung stellt. Diese Programmiersprache basiert auf C und ist mit einigen Hilfsfunktionen, mit denen zum Beispiel die Ein- und Ausgänge des Arduinos angesteuert werden können, ausgestattet [Q1].

Da der Arduino oft von Anfängern verwendet wird, um in die Programmierung einzusteigen ist auch die Programmiersprache ein Problem. C ist eine low-level Programmiersprache, bei der dem Nutzer weniger „abgenommen“ wird als bei high-level Programmiersprachen wie zum Beispiel Python. Das führt dazu, dass Programmieranfänger nicht nur grundlegende Programmierkonzepte, sondern auch spezielle Eigenheiten von C lernen müssen. Um zum Beispiel die Länge eines Arrays zu ermitteln muss die Länge der gesamten Datenstuktur in Bytes durch die Länge eines Array-Elements in Bytes geteilt werden. In Python dagegen kann einfach die len() Funktion benutzt werden. Listen sind generell ein erschwerender Faktor bei der Programmierung des Arduinos. Da C eine low-level Programmiersprache ist, sind zum Beispiel keine verketteten Listen implementiert. Das heißt, wenn man auf dem Arduino eine solche Liste verwenden will, müsste man diese selbst implementieren. Die in C implementierte Datenstruktur ein Array, bei dem aber zum Beispiel nicht ohne weiteres Elemente hinzugefügt oder entfernt werden können. In Python dagegen sind solche Datenstrukturen wie Listen oder Dictionarys schon standartmäßig implementiert. Das macht es sehr einfach in Python eine Liste anzulegen, bei der unkompliziert Elemente mit verschiedenen Datentypen angehängt und entfernt werden können. Beim Arduino dagegen können Arrays nur mit fester Länge und einem vorher festgelegten Datentypen definiert werden. Dies ist zwar deutlich schneller und effizienter als das Python Gegenstück, für Anfänger ist es aber deutlich intuitiver, wenn Elemente aus Listen einfach entfernt und eingefügt werden können. Aufgrund dieser Vereinfachungen braucht Python aber deutlich mehr Ressourcen und ist langsamer als C. Deshalb ist es nicht möglich Python auf dem Arduino auszuführen.

Die Arduino IDE bietet außerdem die Möglichkeit, Programme zu kompilieren und auf den Arduino hochzuladen *[Q2]*. Diese Funktionen basieren auf dem Arduino CLI, dass zum Beispiel für das Kompilieren und Hochladen der Programme zuständig ist. Diese Vorgänge sind aber sehr zeitaufwändig, was das Programmieren erschwert:

|  |  |  |  |
| --- | --- | --- | --- |
| Plattform *(siehe Quellen)* | Kompilieren | Hochladen | Gesamt |
| PC | 0.74s | 1.85s | 3.59s |
| Laptop 1 | 4.43s | 2.78s | 7.21s |
|  |  |  |  |

*(Die Zeiten wurden mit diesen Programmen ermittelt:* [*https://github.com/Bergschaf/Arduino-Benchmark*](https://github.com/Bergschaf/Arduino-Benchmark)*)*

Man kann feststellen, dass die Zeit, die für das Kompilieren benötigt wird, bei größeren Programmen nur leicht steigt. Die Zeit, die zum Hochladen benötigt wird, steigt dagegen etwas bei größeren Programmen. Diese Zeiten sind aber trotzdem sehr lang, da der Nutzer jedes Mal bis zu 7s warten muss, nur um den Effekt einer kleinen Änderung im Programm zu sehen.

Dieses Problem lässt sich mit der bestehenden Methode aber nicht lösen, da die Arduino Programme kompiliert und auf den Arduino Microcontroller hochgeladen werden müssen. Python Programme werden dagegen von einem Interpreter ausgeführt, was fast ohne Zeitverzögerung funktioniert.

# Lösungsidee

Um diese Probleme zu lösen könnte man natürlich einfach mit einer high-level Programmiersprache wie zum Beispiel Python anfangen. Dabei bleibt aber ein großer Vorteil des Arduinos auf der Strecke. Er bietet die Möglichkeit, nicht nur theoretisch zu programmieren, sondern auch mit Elektronik zu experimentieren. Es können zum Beispiel LEDs und Sensoren angesteuert werden. Es gibt zwar Ansätze diese zwei Aspekte der Elektronik und der einfachen Programmierung zusammenzuführen. Da Python aber deutlich mehr Ressourcen benötigt als C/C++ sind solche Microcontroller wie der ESP oder sogar der Raspberry Pi deutlich teurer als der Arduino.

Deshalb ist meine Idee, eine Programmiersprache zu entwickeln, deren Syntax an Python inspiriert, also möglichst einfach, ist, die aber trotzdem so wenig Ressourcen verbraucht, dass sie auch auf dem Arduino ausgeführt werden kann. Um dies zu ermöglichen, müssen zwar einige Vereinfachungen gegenüber Python unternommen werden, die die Programmiererfahrung aber nicht signifikant einschränken. Diese Vereinfachungen machen es möglich, dass die Sprache von einem Transpiler in C++ Syntax übersetzt werden kann, der wiederum für den Arduino kompiliert werden kann. Ein weiterer Vorteil der Sprache soll sein, dass sie nicht nur auf dem Arduino, sondern auch auf einem herkömmlichen Computer ausgeführt werden kann. Die Programme auf dem PC und auf dem Arduino können dann auch parallel ausgeführt werden. Der Programmteil, der auf dem PC ausgeführt wird, hat dabei, sofern der Arduino verbunden ist, die Möglichkeit die Pins auf dem Arduino anzusteuern. Der Programmteil auf dem Arduino kann über die Verbindung dann zum Beispiel Ausgaben an den PC senden, die dann in der Konsole angezeigt werden.

Diese Verbindung gibt dem Programmierer die Möglichkeit, den Arduino auch nur als Steuerungseinheit, die Befehle von einem Programm auf dem PC erhält, für die Pins zu verwenden. Da so das Programm nicht jedes Mal von dem langsamen Arduino CLI kompiliert und hochgeladen werden muss können so die Wartezeiten minimiert werden. Das Programm müsste nur von einem Transpiler in C++ übersetzt werden und dann von einem C++ Compiler für den PC kompiliert werden, was deutlich schneller als der Arduino Compiler ist.

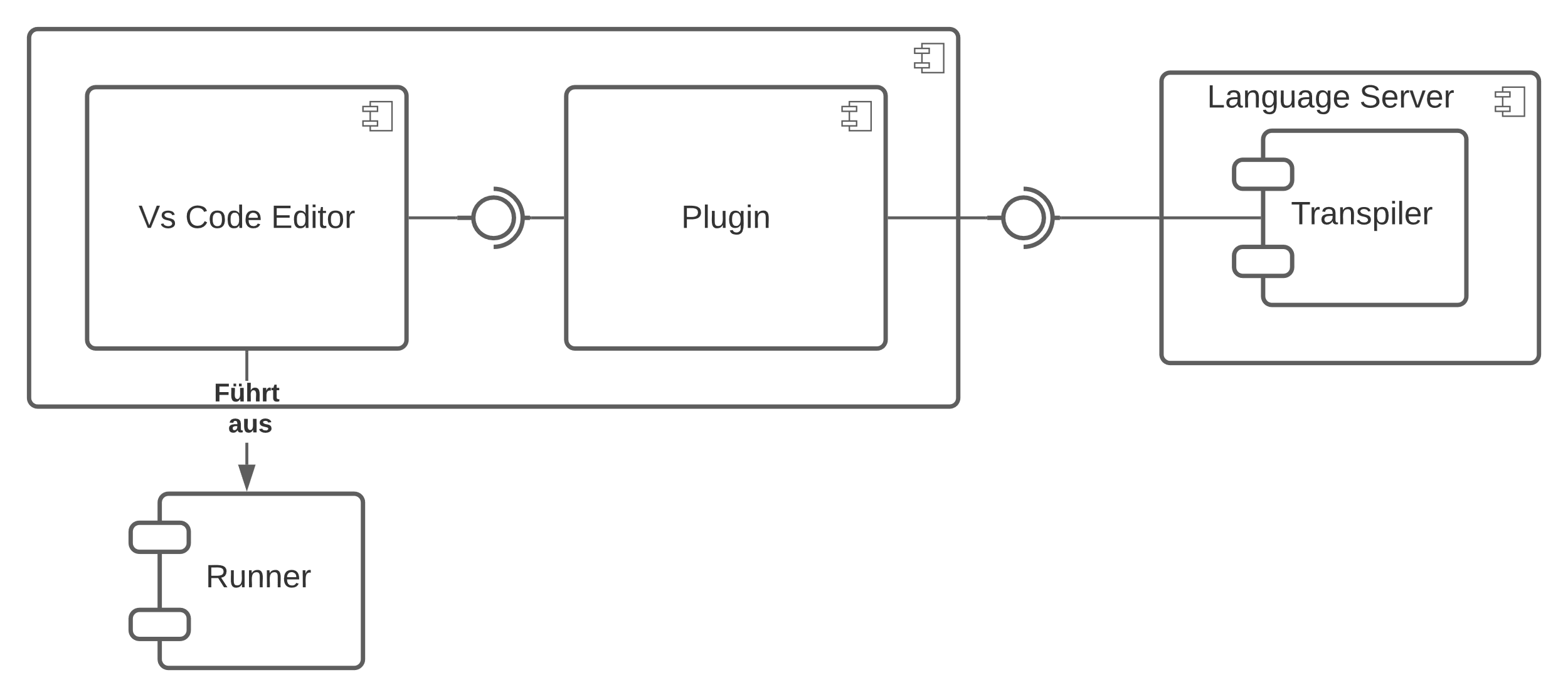
Außerdem ist meine Idee eine Entwicklungsumgebung zu programmieren, die mit modernen Features wie Syntax Highlighting, Autovervollständigung und automatischer Fehlererkennung eine möglichst gute Programmiererfahrung bietet. Da es sehr schwierig ist, eine eigene IDE zu entwickeln ist meine Idee, ein Plugin für die bekannte IDE Visual Studio Code von Microsoft zu entwickeln. Sie ist frei verfügbar und bietet eine gute API, die es ermöglicht mit geringem Aufwand neue Programmiersprachen einzubinden.

# Umsetzung

Als Grundlage verwende ich die IDE Visual Studio Code. Sie stellt eine API für Erweiterungen zur Verfügung, mit der auch Unterstützung für neue Programmiersprachen implementiert werden kann. Um erweiterte Funktionen für diese Sprachen zur Verfügung zu stellen, gibt es die Möglichkeit einen Language Server mithilfe des Language Server Protocols (LSP) zu verwenden [Qutemp1] . Die Idee hinter dem LSP ist, die Entwicklung von Features wie Autovervollständigung und Fehlererkennung für verschiedene Texteditoren so einfach wie möglich zu machen [Qutemp2]. Deshalb wird die Implementierung dieser Features von den Texteditoren und IDEs getrennt, indem nur ein Language Server für jede Programmiersprache entwickelt werden muss. Dieser Language Server kann dann über das Language Server Protocol mit einem Texteditor kommunizieren und so Features wie Autovervollständigung, Fehlererkennung und Dokumentation zur Verfügung stellen. Das bringt auch den Vorteil, dass der Language Server in einer beliebigen Programmiersprache entwickelt werden kann, unabhängig von der API des Texteditors.

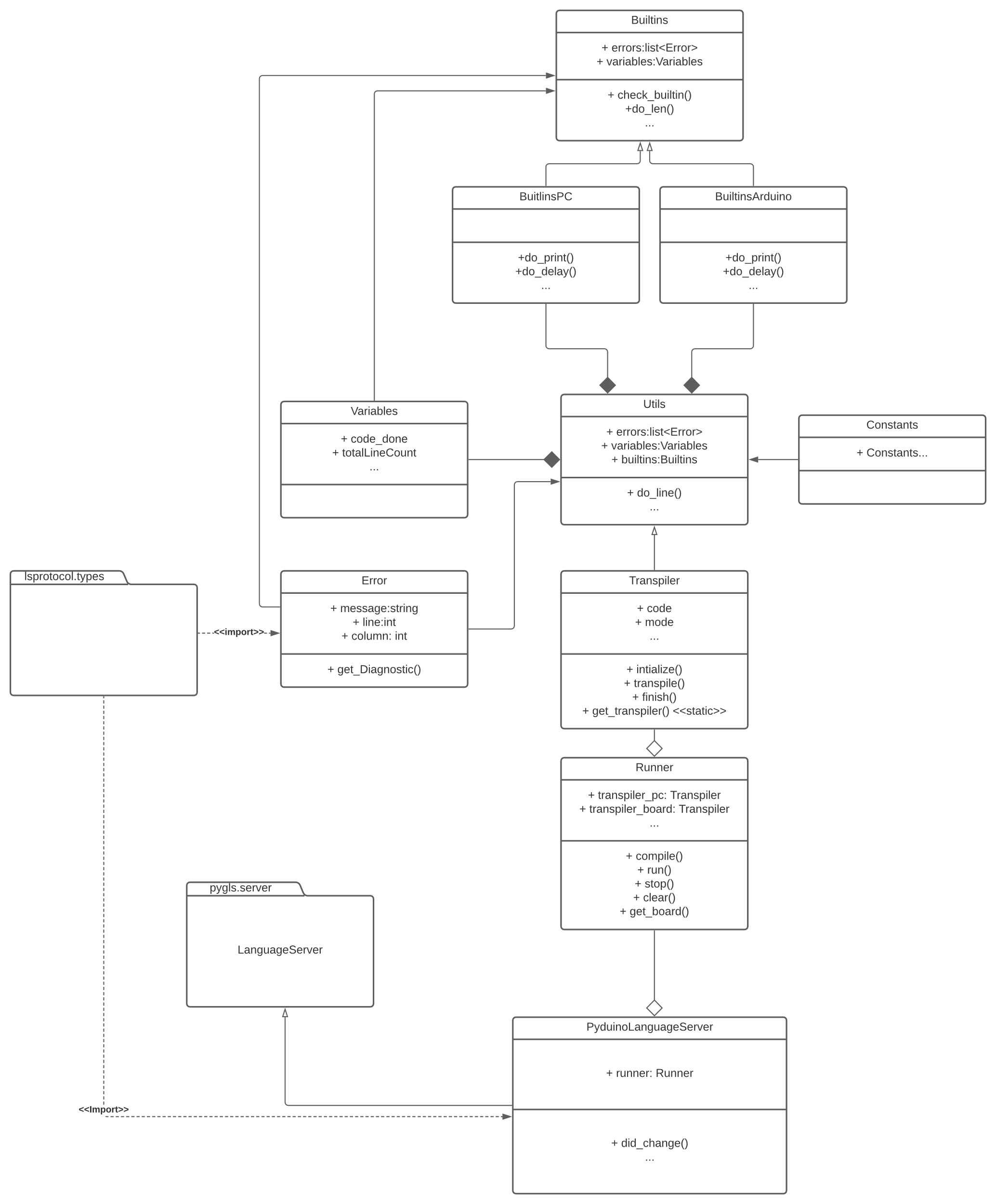
Daher ist meine Idee, mithilfe der Bibliothek pygls (<https://github.com/openlawlibrary/pygls>) einen Language Server für Pyduino in Python zu implementieren. Die Implementierten Features können dann mit einer Erweiterung in Visual Studio Code verwendet werden.

Um Pyduino Programme auszuführen, wird aber auch ein Transpiler benötigt, der den Pyduino Code in C++ übersetzt. Der Code muss beim Übersetzten in C/C++ auch auf Fehler überprüft werden, was eng mit den Aufgaben des Language Servers zusammenhängt. Deshalb habe ich mich entschieden, den Language Server und den Transpiler als ein Python Modul zu implementieren. Wenn ein Programm auf Fehler überprüft werden soll, wird es transpiliert und die Fehler werden gespeichert. Anhand von den Daten, die dabei über das Programm gesammelt werden, kann dann auch Autovervollständigung implementiert werden.

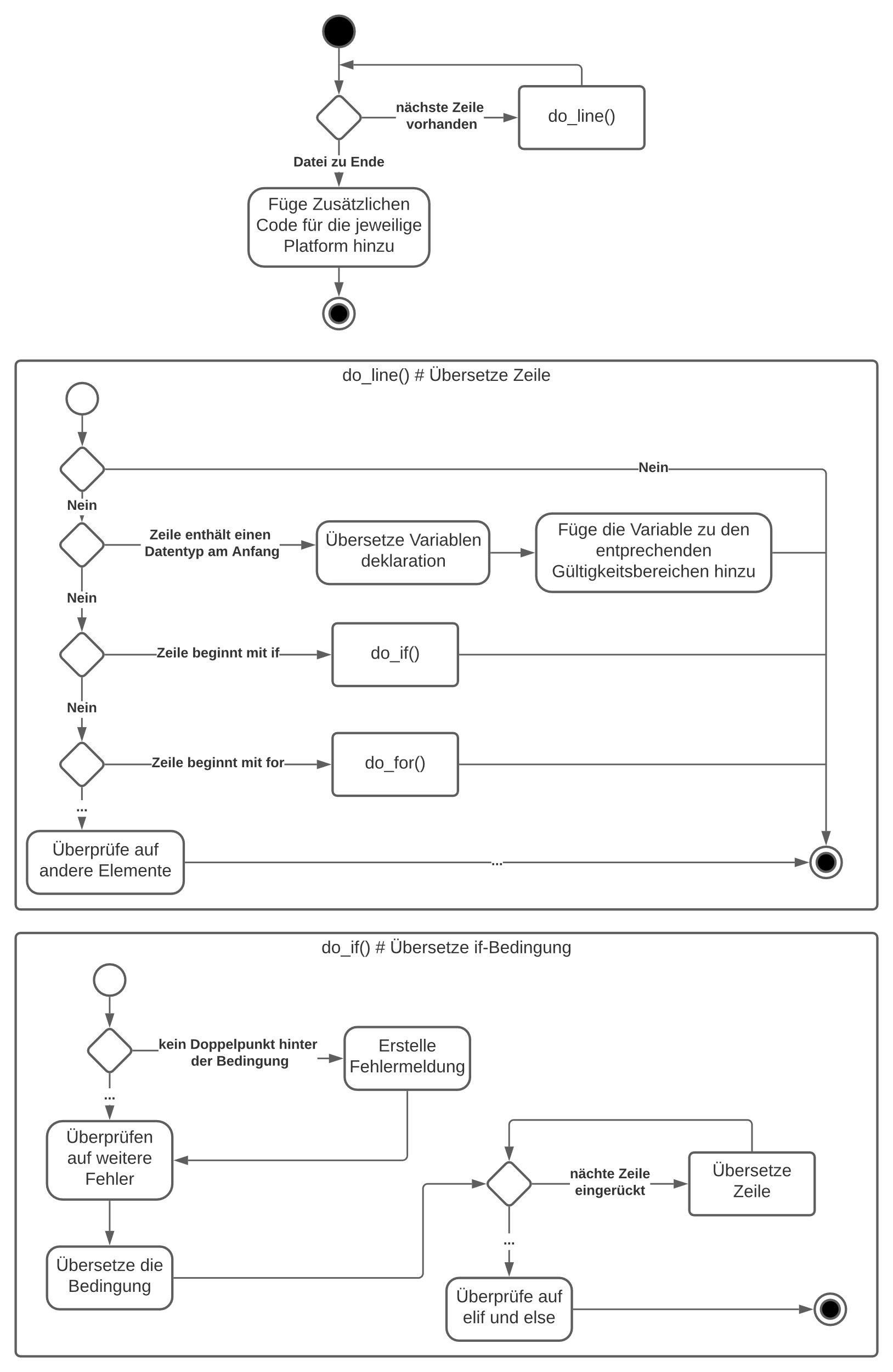


## Transpiler

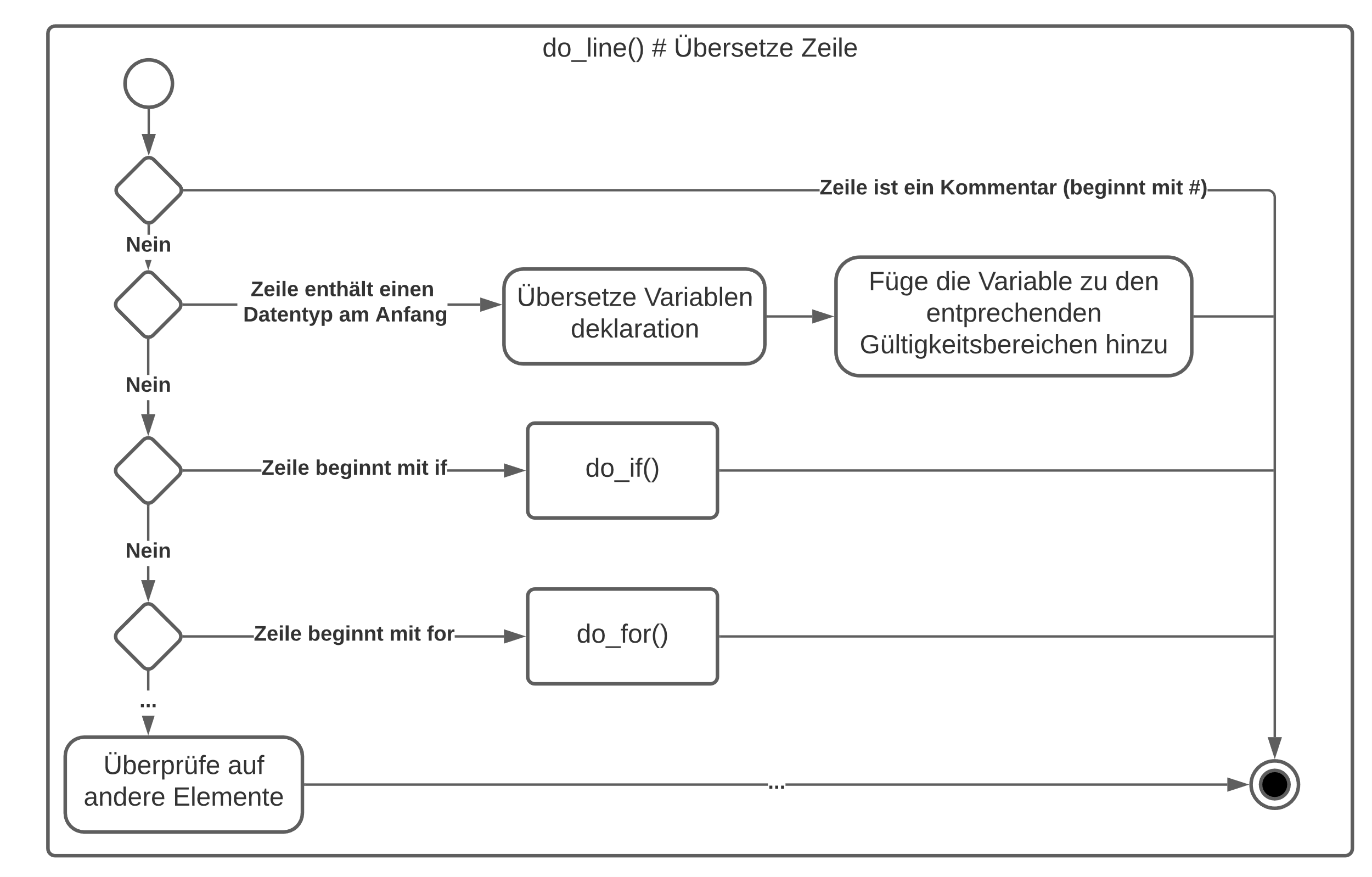
Ich habe mich dafür entschieden, den Transpiler in Python zu implementieren, da Python eine einfache und übersichtliche Programmiersprache ist, die es einfach macht, zum Beispiel mit Strings zu arbeiten. Python ist zwar langsamer als Kompilierte Sprachen wie C++, die Performance reicht für den Language Server aber aus, er kann ein Pyduino Programm innerhalb von wenigen Millisekunden übersetzen. Das folgende Diagramm zeigt den Aufbau des Transpilers:



Für das Transpilieren von einem Pyduino Programm zu C++ ist die Transpiler Klasse verantwortlich. Sie wird jeweils für den #main und #board Teil des Programms instanziiert. Dann erstellt jede Instanz der Klasse ein Variables Objekt, in dem wichtige Daten über das Programm gespeichert sind. Es wird zum Beispiel eine Liste für den fertig übersetzten C++ Code angelegt. Außerdem wird in dem Variables Objekt ein Iterator angelegt, mit dem der Code in der richtigen Reihenfolge Zeile für Zeile übersetzt werden kann und auf den von allen Klassen aus zugegriffen werden kann. Das ist wichtig, da der Transpiler rekursiv arbeitet. Für das Übersetzten des Programms ist nach der Initialisierung die transpile() Funktion zuständig. Sie funktioniert nach folgendem Prinzip:



Der Code wird mithilfe der do\_line() Funktion, die die Transpiler Klasse von der Utils Klasse erbt, zeilenweise übersetzt. Anschließend wird mit der finish() Funktion der Transpiler Klasse der Plattformspezifische Code, der zum Beispiel für die Verbindung zwischen PC und Arduino zuständig ist, hinzugefügt. Dieses Diagramm zeigt das Grundlegende Prinzip der do\_line() Funktion:



Die do\_line() Funktion überprüft, welchem Syntax Element oder welcher Art von Anweisung am Anfang Zeile steht. Dementsprechend wird dann die passende Funktion aus der Utils Klasse aufgerufen. Ein Beispiel dafür ist eine if-Bedingung. Wenn eine Zeile mit „if“ beginnt, dann wird die do\_if() Funktion aufgerufen, die nach folgendem Prinzip arbeitet:



Zuerst wird die Bedingung in C übersetzt. Anschließend werden alle Zeilen hinter der if-Bedingung, die eingerückt sind, mit der do\_line() Funktion übersetzt. Danach wird überprüft, ob nach dem if-Block ein elif oder else Segment zu finden ist. Für diese Segmente wird dann ggf. auch die Bedingung und der eingerückte Teil übersetzt.

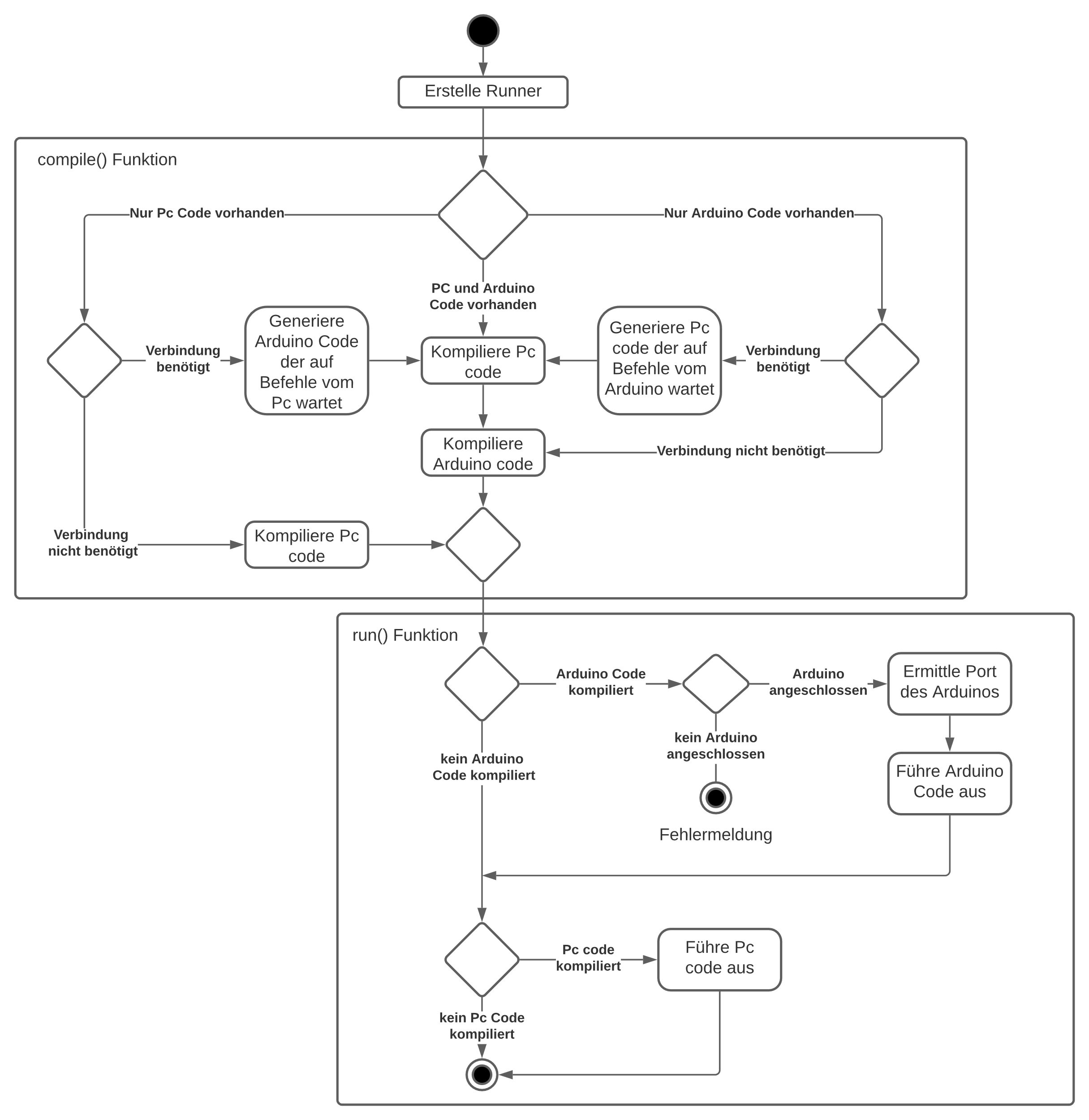
Falls beim Transpilieren Fehler entdeckt werden, wie zum Beispiel ein fehlender Doppelpunkt nach einer if-Bedingung, werden diese gespeichert. Dafür wird ein Error Objekt erstellt, in dem die Fehlermeldung, die Zeile und die Spalte des Fehlers gespeichert sind. Wenn ein Fehler entdeckt wird, wird das Programm nicht abgebrochen, sondern der Transpilierungsprozess wird fortgeführt. Das ist wichtig, da so alle Fehlermeldungen im Texteditor angezeigt werden können.

In der Utils Klasse sind Funktionen definiert, die für das Transpilieren der grundlegenden Programstruktur wichtig sind. Außerdem enthält sie ein Variables Objekt, in dem die Daten, die während dem Transpilieren wichtig sind, gespeichert sind und ein Builtins Objekt.

Die Builtins Klasse ist die Oberklasse für die Funktionen, die in Pyduino standartmäßig implementiert sind. Sie enthält dabei die Funktionen, deren C äquivalent auf dem Pc und dem Arduino gleich sind. Ein Beispiel dafür ist die len() Funktion, die die Länge eines Arrays ermittelt.

Die BuiltinsPC und die BuiltinsArduino Klasse erben von der Builtins Klasse. Sie sind für die Funktionen zuständig, deren Implementation sich zwischen Pc und Arduino unterscheidet. Ein Beispiel dafür ist die print-Funktion. In der Pc-Version werden einfach die Argumente in der Konsole ausgegeben, in der Arduino Version muss über die Serielle Verbindung eine Anfrage an den Pc geschickt werden. So kann dieser dann die Werte vom Arduino in der Konsole ausgeben.

Um ein Pyduino Programm auszuführen, wird die Runner Klasse instanziiert. Sie erhält für den #main und den #board Teil des Programms jeweils eine Instanz der Transpiler Klasse. Das Diagramm zeigt den Ablauf, um ein Pyduino Programm auszuführen:

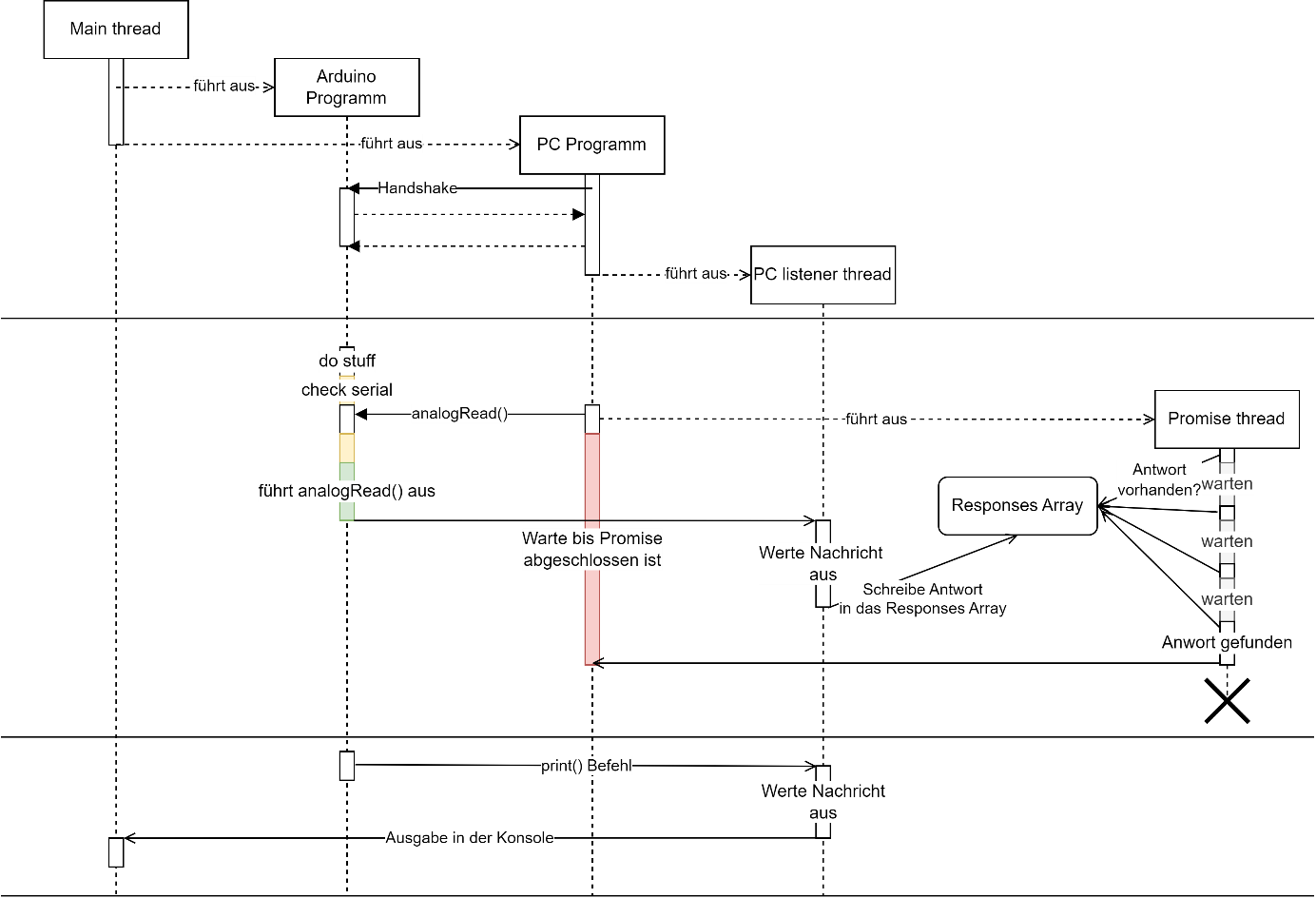


Zuerst wird eine Instanz der Runner Klasse erstellt. Sie enthält jeweils für den #main und #board Programmteil ein Transpiler Objekt, falls der jeweilige Teil vorhanden ist. Mit der transpile() Funktion dieser Transpiler Instanzen können dann die Programmteile in den Syntax für die jeweilige Plattform übersetzt werden. In der compile() Funktion der Runner Klasse wird zuerst überprüft, welche Programmteile vorhanden sind. Wenn ein #main und ein #board Teil existiert, werden beide für die jeweilige Plattform kompiliert. Wenn nur ein Programmteil vorhanden ist, dann wird überprüft, ob die Verbindung zum PC oder zum Arduino benötigt wird. Dies wäre zum Beispiel der Fall, wenn im #main Teil auf die Pins des Arduinos zugegriffen wird oder wenn im #board Teil die print() Funktion aufgerufen wird. Wenn dies der Fall ist, dann wird für die Plattform, die keinen Programmteil im Pyduino Programm hat, Code generiert, der auf Befehle der anderen Plattform wartet. Anschließend werden die Programme für beide Plattformen kompiliert. Falls die Verbindung nicht benötigt wird, wird nur der vorhandene Programmteil kompiliert.

In der run() Funktion wird zuerst überprüft, ob ein kompilierter Arduino Programmteil vorhanden ist. Wenn dies der Fall ist, wird mithilfe des Arduino Clients ermittelt, an welchem Port der Arduino angeschlossen ist. Falls kein Arduino angeschlossen ist, bricht das Programm mit einer Fehlermeldung ab, andernfalls wird das Programm über den entsprechenden Port hochgeladen und auf dem Arduino ausgeführt. Dasselbe wird für den #main Teil wiederholt. Falls ein kompiliertes PC-Programm vorhanden ist, wird dieses ausgeführt.

## Verbindung zwischen PC und Arduino

Dieses Diagramm beschreibt die Funktionsweise der Verbindung zwischen PC und Arduino:



Wenn ein PC- und ein Arduino-Programmteil vorhanden sind und die Verbindung benötigt wird, dann Verbinden sich PC und Arduino zuerst über einen Handshake. Dabei werden vorher festgelegte Character ausgetauscht, um die Verbindung zu überprüfen. Erst wenn der Handshake abgeschlossen ist, starten beide Programme. Dabei wird auf dem PC ein listener-Thread gestartet, der auf Nachrichten des Arduinos über die Serielle Verbindung wartet. Da der Arduino kein Multithreading unterstützt, wird nach jedem Befehl überprüft, ob Daten im Seriellen Buffer [Qserial] vorhanden sind. Diese werden dann verarbeitet und ggf. werden die Befehle ausgeführt.

Wenn in dem PC-Programmteil ein Befehl auf dem Arduino ausgeführt werden soll, dann wird der Befehl über die Serielle Verbindung an den Arduino geschickt. Dieser führt dann den Befehl aus und schickt die Antwort zurück an den PC. Diese wird dann von dem listener-Thread ausgewertet und dann anhand der Nachrichten-ID in das „Responses“ Array geschrieben. Um den Rückgabewert des Befehls zu erhalten hat der PC-Programmteil einen Promise gestartet, nachdem er den analogRead() Befehl an den Arduino geschickt hat:

template<typename T>  
class Promise {  
public:  
 typedef T(bytesToType)(char\* bytes);  
 thread \*t;  
 static void resolve(int requestID, bytesToType bytesToType, T\* targetVariable, char

Responses[MaxRequests][MaxDataLength]) {  
 int millisecondsWaited = 0;  
 while (Responses[requestID][0] == 0) {  
 sleep\_for(milliseconds(1));  
 millisecondsWaited++;  
 if (millisecondsWaited > MaxMillisecondsToWaitForData) {  
 cout << "Error: Timeout while waiting for response" << endl;  
 return;  
 }  
 }  
 if(targetVariable == nullptr || bytesToType == nullptr) {  
 Responses[requestID][0] = 0;  
 return;  
 }  
 \*targetVariable = bytesToType(Responses[requestID]);  
 Responses[requestID][0] = 0;  
 }  
  
 Promise(T\* targetVariable,bytesToType bytesToType, int requestID, char

Responses[MaxRequests][MaxDataLength]) {  
 // start a thread with the resolving function  
 t = new thread(resolve, requestID, bytesToType, targetVariable, Responses);  
 }  
  
 // Destructor  
 ~Promise() {  
 // join the thread  
 t->join();  
 }  
};

Die Promise-Klasse wird mit einem Pointer zur Zielvariable, einer Funktion, die die Antwort zu dem entsprechenden Datentyp der Zielvariable konvertiert, einer Nachrichten ID und dem Responses Array initialisiert. Dabei wird ein neuer Thread mit der resolve() Funktion gestartet. Diese Funktion wartet bis für die Nachricht mit der Nachrichten ID eine Antwort im Response Array vorhanden ist, oder das Zeitlimit überschritten ist. Anschließend wird die Antwort zu dem Entsprechenden Datentyp konvertiert und in die Zielvariable geschrieben. Um sicherzustellen, dass eine Antwort eingegangen ist und ein Wert in die Zielvariable geschrieben wurde, muss der Promise gelöscht werden, was den Destructor aufruft. In diesem wird gewartet, bis der Thread mit der resolve() Funktion abgeschlossen ist.

Um die Antwort auf eine Synchrone Anfrage, bei der das Programm wartet, bis eine Antwort vom Arduino eingegangen ist, zu erhalten, wird der Promise direkt nach seiner Initialisierung wieder gelöscht, was den Destructor aufruft und den main-thread damit anhält, bis der Promise aufgelöst ist. Diese Methode wird auch in der zurzeit implementierten analogRead() Pyduino-Funktion angewendet. Das bedeutet, dass das Pyduino Programm anhält, bis die Antwort der analogRead() Funktion eingegangen ist.

Mit der Promise Klasse ist es aber auch möglich, Asynchrone Anfragen, bei denen das Pyduino Programm nicht angehalten wird, bis eine Antwort eingegangen ist, zu implementieren. Dabei würde der Promise gestartet werden, ohne ihn direkt wieder zu löschen. Da der Promise einen separaten Thread mit der resolve() Funktion startet, wird das Pyduino Programm nicht angehalten. Wenn der Rückgabewert dann gebraucht wird, dann kann der Promise gelöscht werden, was sicherstellt, dass

## VS Code Erweiterung

Dieses Diagramm zeigt die wichtigsten Dateien der VS Code Erweiterung, die auch aus dem Extension Marketplace heruntergeladen werden kann (<https://github.com/Bergschaf/Pyduino-Plugin>):

D:.  
| language-configuration.json  
| main.py  
| package-lock.json  
| package.json

In der „language-configuration.json” Datei ist die Grundlegende Konfiguration für die Pyduino Sprache, die mit der Erweiterung als neue Programmiersprache registriert sind. Es wird zum Beispiel festgelegt, dass Kommentare mit „#“ beginnen oder dass der Editor Klammern beim tippen automatisch schließen soll.

Die „main.py“ Datei kann in der Konsole mit dem Befehl „env/Scripts/python.exe main.py [Pyduino Datei]“ aufgerufen werden, um eine Pyduino Datei zu kompilieren und auszuführen.

In der „package.json“ Datei sind die Eckdaten der Erweiterung wie Name, Version und Veröffentlicher festgelegt. Außerdem ist beschrieben, dass die Erweiterung die Sprache „Pyduino“ zur Verfügung stellt und immer dann aktiviert wird, wenn eine „.pino“ Datei geöffnet ist.

+---client  
| \---src  
| | extension.ts  
|  
+---env  
| | pyvenv.cfg  
| +---Include  
| +---Lib  
| +---Scripts  
| | | python.exe

Der „env“ Ordner enthält die Virtuelle Umgebung mit allen Bibliotheken, die benötigt werden, um den, in Python implementierten, Language Server zu starten. Dabei kann die „python.exe“ Datei im Scripts Ordner verwendet werden, um Python Dateien auszuführen. Um die insgesamte Größe der Datei kleiner zu halten, ist in der Erweiterung kein Python Interpreter enthalten. Daher muss dieser auf dem System installiert sein. Der Ort, an dem der Interpreter auf dem System zu finden ist, ist in der „pyvenv.cfg“ Datei festgelegt. Da sich dieser Ort aber von System zu System unterscheidet, muss er bei jedem System neu bestimmt werden.

Die activate() Funktion der „extension.ts“ Datei wird von VS Code aufgerufen, wenn die Erweiterung aktiviert wird, d.h. es wurde eine „.pino“ Datei geöffnet. Sie ist dafür zuständig den Language Server zu starten. Dafür wird die „python.exe“ Datei in der Virtuellen Python Umgebung verwendet. Dafür muss aber zuerst der Pfad des Python Interpreters auf dem jeweiligen System bestimmt werden. Dieser wird dann in die „pyvenv.cfg“ Datei geschrieben. Anschließend kann der Language Server ausgeführt werden.

+---mingw  
| | MinGW.7z  
| \---MinGW  
| +---bin  
| c++.exe  
|  
+---node\_modules  
|

Da die Programme für den PC in C++ übersetzt werden muss ein C++ Compiler verwendet werden um die Programme zu ausführbaren „.exe“ Dateien zu kompilieren. Da nur wenige Programmieranfänger einen C++ Compiler installiert haben, ist in der Extension der mingw C++ Compiler enthalten. Um die Dateigröße zu reduzieren ist nur die „.7z“ Datei enthalten, die bei Bedarf entpackt wird. In dem „mingw/MinGW/bin“ Ordner ist die „c++.exe“ Datei zu finden, mit der die C++ Programme kompiliert werden können.

+---server  
| | server.py  
| +---transpiler  
| | | arduino-cli.exe  
| | | builtin\_functions.py  
| | | constants.py  
| | | error.py  
| | | runner.py  
| | | transpiler.py  
| | | utils.py  
| | | variables.py  
| | +---SerialCommunication  
| | | ArduinoSerial.ino  
| | | Serial.cpp  
| | | Serial.txt  
| | | SerialClass.h  
| | | SerialPc.cpp  
| | |   
| | +---test  
| | | test.py  
|   
+---syntaxes  
| pyduino.tmLanguage.json

Der Language Server, der von der „extension.ts“ Datei gestartet wird, ist in der „server.py“ Datei implementiert. Bei jeder Änderung der „.pino“ Datei wird die did\_change() Funktion aufgerufen, die mithilfe der Transpiler Klasse das Programm auf Fehler überprüft. Die vorher genannten Klassen wie Transpiler oder Runner sind in den Entsprechenden Python Dateien implementiert.

Um Arduino Programme zu kompilieren, den Port des Arduinos zu ermitteln und die Programme auf den Arduino hochzuladen wird die „arduino-cli.exe“ Datei verwendet.

Die Programme im „SerialCommunication“ Ordner sind für die PC-Seite der Seriellen Verbindung zwischen Arduino und PC zuständig.

Im „test“ Ordner sind Unit Tests zu finden, die Verschiedene Features des Transpilers testen. Dabei werden Pyduino Programme in C++ übersetzt, kompiliert und ausgeführt. Die Ausgaben werden dann mit den erwarteten Ausgaben verglichen. Da das Kompilieren der Programme auf nur einem Prozessorkern aber sehr zeitaufwändig ist, werden die Tests mithilfe der multiprocessing Bibliothek parallel ausgeführt, was die Geschwindigkeit signifikant erhöht.

In der „pyduino.tmLanguage.json“ Datei sind die Regeln für das Syntax Highlighting, wenn eine Pyduino Datei in VS Code geöffnet ist, festgelegt.

Um Programme in VS Code auszuführen können in der „.vscode/launch.json“ Datei Run-Konfigurationen festgelegt werden. Daher wird von dem Language Server eine „Pyduino“ Konfiguration angelegt. Sie startet die „main.py“ Datei mit der Python Installation im „env/Scripts“ Ordner und der entsprechenden „.pino“ Datei als Argument. Die „main.py“ Datei kompiliert und startet das Programm, wenn die Pyduino Run-Konfiguration ausgeführt wird.

# Features

Welche Features bereits implementiert sind, kann über diesen Link eingesehen werden:

<https://trello.com/b/QFvFXErk/pyduino-features>

## Pyduino

Der Name Pyduino ist aus Python und Arduino abgeleitet und bezeichnet die Programmiersprache, die ich entwickle.

### Programmstruktur

Pyduino Programme sind in zwei Teile aufgeteilt, den main-Teil und den board-Teil. Die Einzelnen Teile werden mit „#main“ und „#board“ eingeleitet.

#main

print("Hello World Pc")

#board

print("Hello World Arduino")

Ein Programm kann auch nur einen der Teile enthalten. Wenn die Funktionen des anderen Teils gebraucht werden, dann wird automatisch ein Programm auf der jeweiligen Plattform gestartet, dass zwar kein Pyduino Programm ausführt, aber trotzdem die Funktionen der jeweiligen Plattform zur Verfügung stellt.

#main

analogWrite(11,255)

Hier wird trotzdem ein Programm auf dem Arduino gestartet, dass auf Befehle vom PC (hier „analogWrite“, also „schalte LED an“) wartet.

#board

analogWrite(11,255)

Hier wird kein Programm auf dem PC gestartet, da keine der Funktionen des PCs benötigt werden.

# Das ist ein Kommentar

Kommentare werden mit „#“ eingeleitet und werden vom Transpiler nicht berücksichtigt.

### Variablen

Variablen werden mit dem Datentyp und einem Namen definiert und ihnen kann ein Wert zugewiesen werden.

int x = 42

float y = 2.0

float z = x / y

string s = "Hello World"

Um Datentypen ineinander umzuwandeln können folgende Funktionen verwendet werden:

float y = 3.14

int z = int(y)

float w = float(z)

#### Arrays

Für alle dieser Datentypen können mit folgender Syntax auch Arrays erstellt werden.

int[] a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

float[] b = [3.14,4.2]

string[] c = ["hello", "world"]

Arrays können auch nur mit der Länge ohne Werte initialisiert werden.

int[10] a  # integer Array mit der Länge 10

float[4] b

string[2] c

Auf einzelne Elemente von Arrays kann mithilfe von Indices, die bei 0 beginnen, zugegriffen werden.

int[] array = [42, 15, 4, 8, 23, 16]

print(array[0]) # -> 42

print(array[3]) # -> 8

#### Listen

(Werden eventuell in Zukunft implementiert)

Neben Arrays, die zwar eine schnelle Zugriffszeit (O(1)) haben, bei denen aber nicht einfach Elemente eingefügt oder entfernt werden können, gibt es Listen.

list<int> d = [1, 2, 3, 4, 5]

list<float> e = [1.0, 2.0, 3.0, 4.0, 5.0]

list<string> f = ["a", "b", "c", "d", "e"]

Listen bieten die Möglichkeit Elemente hinzuzufügen und zu entfernen.

list<int> liste = [1,2,3,4,5,6,7,8,9,10]

liste.append(42)  # 42 wird am Ende der liste angehängt

print(liste.pop())  # das letzte Element wird entfernt und ausgegeben

liste.insert(0, 42)  # 42 wird an der Stelle 0 eingefügt

liste.remove(42)  # 42 wird aus der Liste entfernt (die erste 42)

liste.index(42)  # gibt die Stelle, an der 42 in der Liste vorkommt, zurück

liste.count(42)  # gibt die Anzahl zurück, wie oft 42 in der Liste vorkommt

Auch bei Listen kann auf einzelne Elemente zugegriffen werden.

print(liste[0]) # 1

print(liste[8]) # 9

### Funktionen

Funktionen werden mit dem Datentyp, der zurückgegeben wird, dem Namen und den Parametern definiert. Der Programmteil, der nach der Funktionsdeffinition eingerückt wird, gehört zur Funktion. Werte werden mit „return“ zurückgegeben.

int f(int x):

    return x + 1

print(f(1))

Funktionen werden mit Klammern, in denen die Parameter stehen, aufgerufen.

#### Builtins

##### print

Die print-Funktion gibt wie in Python Werte aus. Mehrere Werte können mit einem Komma getrennt werden. Bei Arrays und Listen werden die einzelnen Werte ausgegeben.

print("Hello World") # Hello World

print("Hello", "World") # Hello World

int[] a = [1, 2, 3]

print("Array", a) # Array [1, 2, 3]

Wenn diese Funktion auf dem Arduino ausgeführt wird und der Arduino mit dem PC verbunden ist, dann wird die Ausgabe in der Konsole auf dem PC angezeigt.

##### len

Die len-Funktion gibt die Länge von Arrays und Listen zurück.

int[] array = [42, 15, 4, 8, 23, 16]

print(len(array)) # 6

##### delay

Die delay Funktion hält das Programm für eine bestimmte Zeit in Millisekunden an. Dabei wird aber trotzdem auf Anweisungen von jeweils Arduino oder PC gewartet, die dann während dieser Zeit auch ausgeführt werden können.

delay(1000) # warte 1 Sekunde

##### analogWrite

Steuert die Pins am Arduino mit einem Wert von 0 - 255 an. Der Arduino, der eigentlich nur digitale Pins besitzt kann die Spannung bestimmte Pins (die mit einer Welle markiert sind) durch Pulsweitenmodulation (PWM) steuern. Dadurch kann zum Beispiel die Helligkeit von LEDs gesteuert werden.

for i in range(255):

    analogWrite(11,i)

    delay(10)

Die LED an Port 11 wird langsam heller, egal ob das Programm auf dem PC oder auf dem Arduino ausgeführt wird.

##### analogRead

Liest die Spannung an den analogen Ports des Arduino in einem Wertebereich von 0 – 1023 aus.

print(analogRead(0))

### Kontrollstrukturen

#### if-Bedingung

#### while-Schleife

Eine Schleife, die ausgeführt wird, solange eine Bedingung wahr ist.

int x = 0

while x < 10:

    x = x + 1

    print(x)

Diese Schleife gibt die Zahlen von 1 bis 10 aus.

#### for-Schleife

Die for-Schleife iteriert mit einer Zählvariable über einen Bereich von Zahlen, der mit „range“ festgelegt wurde. Das Grundlegende Schema sieht folgendermaßen aus:

for Zählvariable in range(start,end,step):

    #code

Beispiele:

for i in range(10):

    print(i)

Diese for-Schleife gibt die Zahlen von 0 bis 9 aus.

for i in range(4,20,3):

    print(i)

Diese for-Schleife gibt die Zahlen von 4 bis 20 in 3er Schritten aus.

For-Schleifen können auch über Arrays und Listen iterieren.

int[] array = [42, 15, 314, 69, 20]

for i in array:

    print(i)

Diese for-Schleife gibt die Elemente aus dem Array einzeln aus.

# Getting Started

Um Pyduino Programme zu schreiben kann die VS Code Erweiterung verwendet werden. Diese Funktioniert aber zurzeit nur unter Windows.

Dafür muss als erstes VS Code installiert werden. Die Installationsdatei kann unter <https://code.visualstudio.com/download> heruntergeladen werden.

Danach kann die Pyduino Erweiterung installiert werden. Dafür muss zuerst an der Seitenleiste der „Erweiterungen“ Tab ausgewählt werden. Mit der Suchfunktion kann die Erweiterung unter „pyduino“ gefunden werden. Mit einem Klick auf „Install“ wird die neuste Version der Erweiterung installiert.

Um ein Pyduino Programm zu schreiben, muss dann in VS Code eine Datei mit der Endung „.pino“ erstellt werden. Der Pyduino Code wird dann automatisch auf Fehler überprüft, die dann rot unterstrichen sind und es wird Syntax Highlighting angewendet. Um das Programm auszuführen kann dann im Menu „Run and Debug“ die „Pyduino“ Konfiguration gestartet werden. In der Debug Konsole sind dann die Ausgaben von PC und Arduino zu sehen.

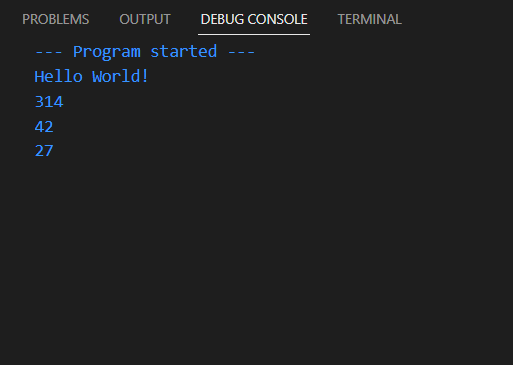
# Ergebnisse

Als Ergebnis kann ich nach 3 Monaten eine Programmiersprache präsentieren, deren grundlegende Funktionen bereits implementiert sind. Der Syntax ist an Python interpretiert und so für Anfänger intuitiver zu erlernen als C. Ein wichtiger Aspekt ist außerdem, dass das PC- und das Arduino-Segment eines Pyduino Programmes parallel ausgeführt werden kann. Über die Serielle Verbindung können PC und Arduino Daten und Befehle ausführen. So kann vom PC aus zum Beispiel mit der analogWrite() Funktion auf die Pins des Arduinos zugegriffen werden und der Arduino kann Werte in der Konsole am PC ausgeben.

Um Pyduino kann auf jedem Windows System verwendet werden, auf dem VS Code und Python installiert sind. Es muss lediglich die VS Code Erweiterung für Pyduino installiert werden. Dann können Pyduino Programme erstellt werden, bei denen die Erweiterung Syntax Highlighting und Fehlererkennung zur Verfügung stellt.



Diese Programme lassen sich mit der automatisch erstellten Run-Konfiguration einfach ausführen.



Auf virtueller Maschine getestet

# Ausblick

Neben der Fehlererkennung könnte man mit den Daten, die der Language Server über das Pyduino Programm sammelt auch Autovervollständigung implementieren. Das würde das Programmieren von Pyduino Programmen weiter erleichtern.

Außerdem besteht die Möglichkeit, die Pyduino VS Code Erweiterung als separate IDE zu implementieren, die mit einer „.exe“ Datei installiert werden kann. Dafür könnte die Theia Plattform [Qtheia] verwendet werden, mit der auch die Arduino IDE 2.x [Q2] implementiert ist.

* Asynchrone anfragen
* Arduino Receiver mode, Programme müssen nicht hochgeladen werden, Arduino wartet immer nur auf befehle

# Dank

Benno und Matthias nicht vergessen

# Quellen

*[Q1]* <https://en.wikipedia.org/wiki/Arduino#Sketch>

*[Q2]* <https://github.com/arduino/arduino-ide>

[Qutemp1] <https://code.visualstudio.com/api/language-extensions/overview>

[Qutemp2] <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>

[Qthea] <https://theia-ide.org>

[Qserial] <https://www.arduino.cc/reference/en/language/functions/communication/serial/available/> hier was besseres ^

PC: CPU: AMD Ryzen 5 5600X @3,7GHz 6-Cores 12-Threads; RAM: 16gb; OS: Windows 11

Laptop1: CPU: Intel Pentium N3710 @1,6GHz 4-Cores 4-Threads; RAM: 4gb; OS: Windows 10

Laptop2: