# Der Arduino steuert klassenbewusst


## Peter Beater



Mit dem Arduino lassen sich auch komplexere Automatisierungsvorhaben realisieren, wenn man bei Aufbau und Programmierung Vorgehensweisen der industriellen Steuerungstechnik anwendet. Am Beispiel einer Fußgängerampel folgt nach einem typischen Arduino Skript der permanent zyklische Betrieb einer SPS und anschließend eine Kapselung durch Klassen im Sinne der objektorientierten Programmierung.


## Worum geht es?

Automatisierungsaufgaben im Haushalt oder Garten lassen sich häufig gut mit einem Arduino lösen. Das Angebot an Sensoren ist fast schon legendär. Die erforderliche Rechenleistung und Speichergröße sind häufig gering und mit dem Mega oder dem Due sind ggf. auch leistungsfähigere Arduinos als der Uno R3 verfügbar. Aufsetzbare Erweiterungen, sogenannte shields, gibt es zu Ansteuerung von kleinen Elektromotoren oder Relais. Geräte am 230 V Netz können gefahrlos mittels einer Funksteckdose vom Arduino über einen Sender im 433 MHz Band ein- oder ausgeschaltet werden. LCD oder OLED-Displays lassen sich leicht an den Arduino anschließen, um Statusmeldungen auszugeben.

Die Programmierung erfolgt über die Arduino IDE. Am Beispiel einer Fußgängerampel werden drei Programmstrukturen vorgestellt. Es geht los mit einem „klassischen“ Arduino Programm oder im Slang „Skript“, in dem die einzelnen Ampelphasen mit der Anweisung **`delay()`** gesteuert werden. Nachteil dieser Vorgehensweise ist, dass während der Wartezeit der Prozessor keine anderen Aufgaben ausführen kann, z. B. eine Anzeige aktualisieren. Die zweite Programmstruktur verwendet daher eine Vorgehensweise aus der industriellen Steuerungstechnik, bei der der Ablauf in einzelne Schritte gegliedert wird, die nacheinander ausgeführt werden.

Bei einer großen Anzahl an Ein- und Ausgängen wird es wichtig, den Überblick zu behalten. Dann bietet es sich an, die Strukturierungsmöglichkeiten des von der IDE verwendeten avr-gcc Compilers zu nutzen. Dazu werden nach den Regeln der objektorientierten Programmierung Klassen erstellt, die jeweils Parameter und Vorgehensweisen für eine Ampel zusammenfassen und gegenüber der Umwelt kapseln. Im vierten Schritt wird das Programm modularisiert, so dass die dabei erstellten Bibliotheken einfach wiederverwendet werden können.

## Der problemvermeidende Aufbau

Bevor es aber mit der Programmierung losgehen kann, sind einige Worte zum Aufbau sinnvoll.

Das typische Arduino-Projekt beginnt mit einem Uno, Steckbrett und vielen Steckkabeln. Wenn wir ein Startkommando geben wollen, bietet sich ein einfacher Taster an. Wir können den Pin 3 als Eingang deklarieren und mit dem Taster eine Verbindung zum 5 V Pin herstellen. Als Ausgabe wählen wir eine Leuchtdiode, LED, am Pin 13. Die Ausgangsspannung des Arduinos beträgt 5 V, so dass für die LED ein Vorwiderstand erforderlich ist.

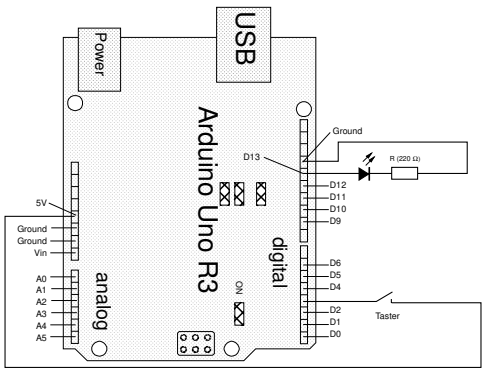


Bild 1 Einfacher Aufbau mit Taster und LED

Bei diesem Aufbau kann es vorkommen, dass die LED aufblinkt, obwohl wir den Taster nicht betätigt haben. Dies liegt an dem starken elektrischen 50 Hz-Feld, das die überall vorhandenen Netzleitungen erzeugen. Für Bild 2 wurden an den Analogeingang A5 ein 30 cm langes Kabel angeschlossen und dann mit **`analogRead(5)`** so schnell wie möglich Werte erfasst. Das 50 Hz-Signal ist deutlich zu sehen. Gleichzeitig wurde der nicht beschaltete Eingang D1 mit **`digitalRead(1)`** ausgelesen und ebenfalls im Bild 2 dargestellt.

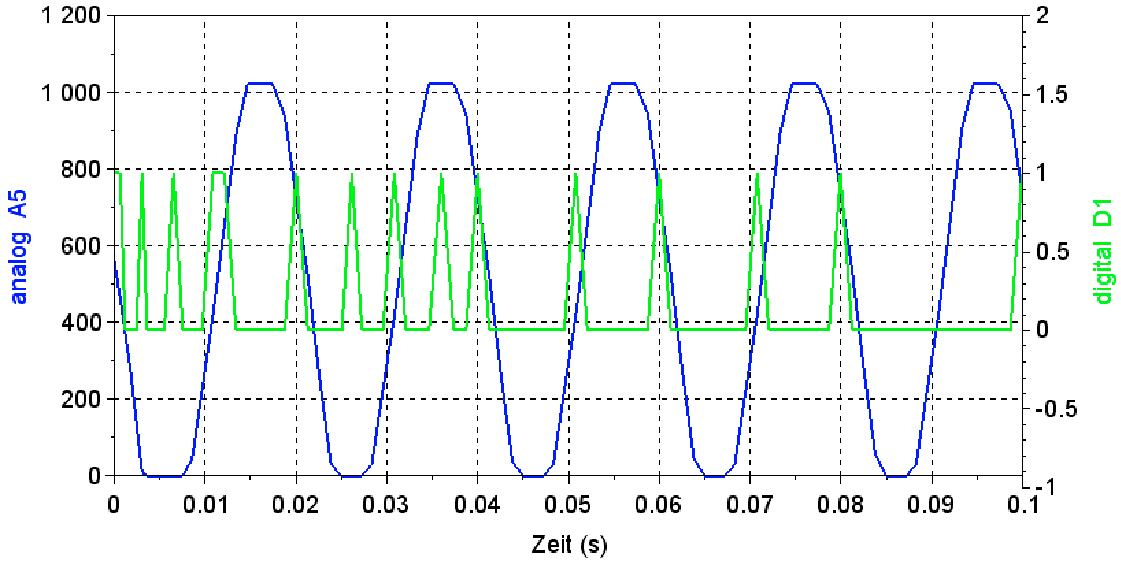


Bild 2 Gemessenes Analogsignal (blau) und Digitalsignal (grün)

Da diese Störungen bei einem Aufbau nach Bild 1 immer auftreten, ist in den im Arduino Uno verwendeten Prozessor ATmega328/P bereits eine Abhilfe eingebaut worden. Diese nennt sich **`INPUT_PULLUP`** und besteht aus einem hochohmigen Widerstand, der den Pin mit der Versorgungsspannung verbindet. Der Digitaleingang ist unbeschaltet daher immer logisch 1. Um den Zustand logisch 0 zu erreichen, verbinden wir den Pin mit dem Bezugspotential **`GROUND`**.
Im **`setup`**-Teil des Arduinoskriptes verwendet wir die Anweisung

```
pinMode(tasterPin, INPUT_PULLUP);
```

und im Programm negieren wir den gelesenen Zustand gleich mit dem **`!`** , d. h.

```
! digitalRead(tasterPin)
```

Diese Vorgehensweise bedarf allerdings einer gewissen Umsicht, [1]. Wenn wir irrtümlich den **`tasterPin`** als digitalen Ausgang deklarieren, auf logisch 1 setzen und auf den Taster drücken, fließt ein unzulässig großer Strom, der zur Zerstörung führt. Daher ist in der Schaltung in Bild 3 noch ein 180 Ω Widerstand vorgesehen, der in diesem Fall den Strom auf einen verträglichen Wert begrenzt. Bei der vorgesehenen Verwendung als Eingang stört dieser Widerstand nicht.

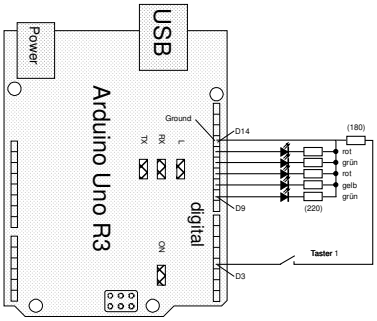


Bild 3 Verbesserter Aufbau mit Taster und LEDs für die Ampel

## Eine Fußgängerampel als Aufgabenstellung

Es gilt, eine klassische Fußgängerampel zu steuern. Nach dem Druck auf den Starttaster „erwacht“ das grüne Licht für die Autos und dann folgt der bekannte Ablauf. Die Zeiten sind im Bild 4 dargestellt.

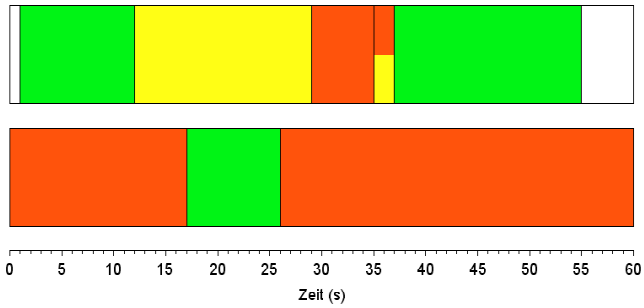


Bild 4 Farbfolge der Autoampel (oben) und der Fußgängerampel (unten) als Zeitverlauf

## Das klassische Arduino Skript

Im klassischen Arduino Skript deklarieren wir zuerst die Aufgaben der einzelnen Pins:

```
const byte ledRotPin = 11;          // rote LED  fuer Autos an Pin 11
pinMode(ledRotPin, OUTPUT);         // Pin als Ausgang
digitalWrite(ledRotPin, LOW);       // LED ausschalten
```

Den Taster deklarieren wie oben:

```
const byte tasterPin = 3;
pinMode(tasterPin, INPUT_PULLUP);    // Pin D3 als Eingang
```

Solange der Taster nicht gedrückt wird, ist der Wert logisch 1. Wir nutzen dies als Bedingung für eine **`while`**-Schleife, um das Programm auf der Stelle treten zu lassen, bis der Taster gedrückt wird.

```
while (digitalRead(tasterPin) ) {  // wenn gedrueckt wird, geht es nach diesem Block weiter

  }
```

Danach folgen die Zuweisungen für die Ausgänge und Wartezeiten. Dabei setzen wir für die folgende Phase die LEDs und halten dann mit der Anweisung **`delay()`** das Programm an. Für die Grün-Phase folgt:

```
digitalWrite(ledGruenPin, HIGH);  // Autos jetzt gruenes Licht angeschaltet
  delay(11000); // Wartezeit, das Programm steht still und wartet
```

Der **`loop`**-Block des Skriptes **`Skript_mit_delay.ino`** beginnt wie folgt:

```
// -------------------- LOOP --------------

void loop() {

  // Anfang, Autos fahren, da alle Lichter aus; Fussgaenger stehen, da rotes Licht
  digitalWrite(ledRotPin, LOW);       // LED ausschalten
  digitalWrite(ledGelbPin, LOW);      // LED ausschalten
  digitalWrite(ledGruenPin, LOW);     // LED ausschalten, da Ampel auss
  digitalWrite(ledRotRundPin, HIGH);  // LED als einzige einschalten
  digitalWrite(ledGruenRundPin, LOW); // LED ausschalten


  // Warten, bis der Taster gedrueckt wird. In dieser Zeit wird das Programm quasi angehalten,
  // da die while-Schleife ausgefuehrt wird, bis digitalRead(tasterPin) logisch 0 ist.

  while (digitalRead(tasterPin) ) {  // wenn gedrueckt wird, geht es weiter

  }
```

```
  delay(1000); // Wartezeit, das Programm steht still und wartet

  digitalWrite(ledGruenPin, HIGH);  // Autos jetzt gruenes Licht angeschaltet
  delay(11000); // Wartezeit, das Programm steht still und wartet

  // Auto wechselt von gruen auf gelb
  digitalWrite(ledGruenPin, LOW);
  digitalWrite(ledGelbPin, HIGH);
  delay(5000);

  // Auto wechselt von gelb auf rot
  digitalWrite(ledGelbPin, LOW);
  digitalWrite(ledRotPin, HIGH);
  delay(3000);

  ...
```

Wenn dieses Skript alle Anforderungen erfüllt, gibt es keinen Grund, weiterzuentwickeln. Die wichtigste Anforderung an ein Rechnerprogramm ist erfüllt: Einfach lesbar; die korrekte Arbeitsweise wird dabei stillschweigend vorausgesetzt. Ein besonders trickreich programmiertes Programm rächt sich spätestens dann, wenn man einige Zeit später eine Änderung vornehmen muss. Dann braucht man einiges an Zeit, um die früheren Gedankengänge wieder nachzuvollziehen.

In diesem Programm tut der Arduino bei **`delay()`** ganz aktiv nichts. Er wartet, bis die Zeit abgelaufen ist und kann in dieser Wartzeit keine weiteren Aufgaben übernehmen. Wir können daher einen Teil des Verhaltens realer Ampeln nicht realisieren: Wenn erneut gedrückt wird und die Ampelanlage noch nicht im Ruhezustand ist, wird die übliche Grünphase abgewartet und dann beginnt der obige Ablauf mit gelb. Bei unserem Programm findet immer noch das Ausschalten und die Ruhephase statt und erst dann wird auf einen Tastendruck reagiert. Daher folgt die Variante zwei, bei der auch komplexere Bedingungen realisiert werden können.

## Ein Blick zu den Profis

Im Sondermaschinenbau wird nicht für jede Maschine ein neues Steuergerät entwickelt, sondern ins Regal gegriffen und eine Speicherprogrammierbare Steuerung, SPS, von der Stange verwendet. Diese Geräte erfüllen die Ansprüche an Industrieelektronik, was z. B. den Temperaturbereich oder die Festigkeit gegen Vibrationen betrifft. Eine weit verbreitete SPS-Software, Codesys, ist auch für den raspi verfügbar; die Leistungsfähigkeit des Arduinos reicht dazu nicht aus.

Eine übliche Betriebsart ist bei SPSen der „permanent zyklische Betrieb“. Dabei wird ein Programm ausgeführt und unmittelbar nach Abarbeiten wieder von vorne angefangen. Verglichen mit z. B. Windows auf einem PC ist dies ein sehr einfaches, aber damit auch sehr schnelles und wenig anspruchsvolles Betriebssystem. Unser Arduino bietet genau diese Funktion mit der Endlosschleife **`loop()`** als Hauptprogramm.

Eine Möglichkeit, Abläufe zu beschreiben, ist der Funktionsplan (DIN EN 60848), bei dem die einzelnen Phasen des Ablaufs in Schritte unterteilt werden [2]. Bei der Ampel ist das z. B. grün für die Autos und rot für die Fußgänger. Ein Schritt wird verlassen, wenn die dazugehörige Übergangsbedingung erfüllt ist. Bei der Ampel kann das der Tasterdruck am Beginn des Zyklus sein, oder das zeitliche Ende der rot-gelb Phase. Dieser Schritt wird dann passiv und der folgende Schritt aktiv. Bei SPS-Systemen gibt es eine eigene Programmiersprache, genannt Ablaufsprache, um diese Art von Steuerungen einfach erstellen zu können. Beim Arduino können wir entweder **`if`**-Anweisungen oder übersichtlicher das **`switch`**-Konstrukt verwenden.

## Der permanent zyklische Betrieb mit dem Arduino

Im Gegensatz zum ersten Programm verwenden wir nicht die **`delay()`**-Anweisung, sondern nutzen die Funktion **`millis()`**. Sie gibt die Zeit seit Einschalten des Arduinos in Millisekunden an. Wir rufen sie zu Beginn eines Schrittes auf und speichern den Wert ab. Wenn die aktuelle Betriebsdauer größer ist als Startzeit plus gewünschte Schrittdauer, wechseln wir in den nächsten Schritt.

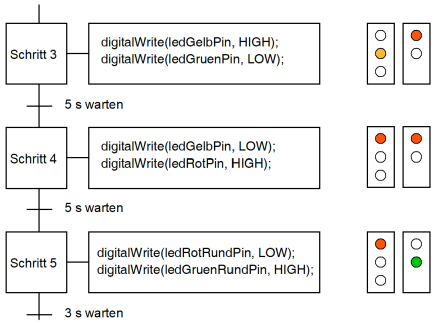


Bild 5 Drei Schritte des Funktionsplans und rechts daneben die aktiven Lampen

Wir können diesen Ablauf mit einer Folge von **`if`**-Anweisungen programmieren.

```
if ( (schritt = 3) && (millis() >= zeit + 5000) ) {
          // Auto wechselt von gruen auf gelb
          digitalWrite(ledGruenPin, LOW);
          digitalWrite(ledGelbPin, HIGH);
          zeit = millis();
          schritt = 4;
        }
```

Übersichtlicher wird es, wenn wir die **`switch`**-Anweisung verwenden. Wir deklarieren die Variable **`schritt`**

```
int schritt = 1;
```

und verwenden dieses Konstrukt, um von einem Schritt zum nächsten zu schalten.

```
switch ( schritt ) {
    case  1: {
        // erster Block, schritt = 1
               ... Anweisungen
               schritt = 2; // weiterschalten zum nächsten Block
      }
      break;
    case  2: {
        // zweiter Block, schritt = 2
               ... Anweisungen
               schritt = 3; // weiterschalten zum nächsten Block
      }
      break;
    ...
    case  9: {
        // letzter Block, schritt = 9
               ... Anweisungen
               schritt = 1; // zurück zum ersten Block
      }
  } // Ende Switch
```

Die Verzögerungszeiten setzen wir mit **`if`**-Anweisungen um:

```
zeit = millis();

...

if ( millis() >= zeit + 11000 ) {
        zeit = millis();
        ...
        }
```

Das gesamte Steuerungsprogramm ist in **`Funktionsplan_mit_switch.ino`** enthalten.

Wenn dieses Programm alle Anforderungen erfüllt, gibt es keinen Grund, weiterzuentwickeln. Die wichtigste Anforderung an ein Rechnerprogramm ist erneut erfüllt: Einfach lesbar. Von der Struktur ist es flexibler als die erste Version. Es benötigt auch wenig Speicherplatz für Programm und Daten, was beim Einsatz von Mikrocontrollern immer ein wichtiger Punkt ist.

Ein Nachteil dieses Programms ist, dass es bei Erweiterungen unübersichtlich werden kann. Wenn wir nicht eine, sondern an einer großen Kreuzung fünf Ampeln haben, muss man beim Programmieren sehr aufpassen, immer die richtigen Leuchten zu erwischen. Die Informatik hat sich dieser Problematik angenommen und objektorientierte Programmiersprachen entwickelt. Einige Möglichkeiten, die der avr-gcc Compiler über die bisher verwendete Sprache C hinaus bietet, nutzen wir im folgenden Programm.

## Objekte als höhere Programmierkunst

Für das Steuerungsprogramm der Ampel benötigen wir Parameter und Anweisungen. Bei einer größeren Anlage kann das sehr unübersichtlich werden. Um leichter den Überblick zu behalten, fasst man bei der objektorientierten Programmierung alles Erforderliche an einer Stelle zusammen und spricht dann von einer Klasse. Diese Klasse können wir als Gussform betrachten, mit der wir beliebig viele Abgüsse erstellen können. Diese nennt die Informatik Objekte, Exemplare oder Instanzen. Ziel dieser Programmierweise ist es, das Programm übersichtlich und fehlarm zu machen.

Diese Vorgehensweise wenden wir beim Arduino an, ohne uns dessen bewusst zu sein. Bei der seriellen Ausgabe mittels **`Serial`** auf den Monitor merken wir überhaupt nicht, dass wir ein Objekt einer Klasse erzeugen und verwenden. Etwas deutlicher wird dies, wenn wir zusätzlich zur seriellen Schnittstelle an den Pins 0 und 1 noch weitere Schnittstellen benötigen, z. B. um mit externer Hardware zu kommunizieren. Dann verwenden wir die in der IDE bereits vorhandene Klasse **`SoftwareSerial`** und fügen am Programmanfang die Zeile

```
SoftwareSerial mySoftwareSerial(10, 11); // RX, TX
```

ein. Damit definieren wir ein Objekt mit dem Namen **`mySoftwareSerial`**. Jetzt dient Pin 10 zum Lesen und Pin 11 zum Senden. Die Baudrate dieser Schnittstelle legen wir mit der Anweisung

```
mySoftwareSerial.begin(9600);
```

fest. Dabei handelt es sich bei **`begin()`** um eine sog. Methode. In anderen Sprache würde man sie als Unterprogramm oder Subroutine bezeichnet. Ein Methodenaufruf hat immer die Form **`Objektname.Methodenname (ggf. Parameter)`**. Wenn wir weitere Schnittstellen benötigen, können wir weitere Objekte erzeugen, z. B.

```
SoftwareSerial mySecondSoftwareSerial(5, 6); // RX, TX
```

und mit

```
mySecondSoftwareSerial.begin(115200);
```

auch mit anderen Parametern initialisieren.

Für unsere Ampel erstellen wir eine Klasse, die wir auch **`Ampel`** nennen.

```
class Ampel { }
```

Bei der Deklaration folgen die Variablen, die in der Klasse benötigt werden. Charakteristisch für objektorientierte Programmierung ist es, dass wir auf die Daten einer Klasse nicht direkt zugreifen können, sondern dafür die Methoden der jeweiligen Klasse aufrufen müssen. Diese Kapselung verhindert, dass wir aufgrund eines Programmierfehlers unbeabsichtigt Variablen ändern. Das Schlüsselwort **`private`** sorgt für diese Kapselung. Das Gegenstück ist **`public`**, wenn die Variablen von außen verfügbar sein sollen.

```
private:
    byte pinRot;
    byte pinGelb;
    byte pinGruen;
    boolean freieFahrt;
```

Die Variable **`freieFahrt`** zeigt an, dass diese Ampel grün zeigt. Sie dient in erster Linie zur Erweiterung unseres recht speziellen Beispiels, da sie das Ergebnis einer Zuweisung enthält. Im nächsten Abschnitt beschreiben wir die nach außen sichtbare Schnittstelle unserer Klasse. Daher das Schlüsselwort **`public`**. In unserem Fall weisen wir den für die digitalen Ausgänge benötigten Variablen die beim Aufruf angegebenen Pinnummern zu. Wir rufen auch unsere Methode **`init()`** auf, um die Pins als Ausgänge zu deklarieren, und die Methode **`zeigeRot()`**, um das rote Licht einzuschalten.

```
public:
    Ampel(byte pRot, byte pGelb, byte pGruen) {
      pinRot   = pRot;
      pinGelb  = pGelb;
      pinGruen = pGruen;
      init();
    }
```

Diesen ersten Teil nennt man auch Konstruktor. Er hat den gleichen Namen wie die Klasse und wird genutzt, um die Eigenschaften des Objektes zu initialisieren.
Dann erst kommen die eigentlichen Codeblöcke, also die Rechenanweisungen. Diese Methoden besitzen jeweils eigene Schnittstellen und gehören fest zu ihrer Klasse. Die Methode **`zeigeGelb()`** schaltet beispielsweise die gelbe Lampe ein, alle anderen aus. Außerdem setzt sie die lokale Variable **`freieFahrt`** auf **`true`**.

```
void zeigeGelb() {
      digitalWrite(pinRot, LOW);
      digitalWrite(pinGelb, HIGH);
      digitalWrite(pinGruen, LOW);
      freieFahrt = true;
    }
```

Um auf die als **`private`** deklarierte lokale Variable **`freieFahrt`** zugreifen zu können, benötigen wir eine eigene Methode und später dann im Hauptprogramm einen Methodenaufruf.

```
boolean getFreieFahrt() {
      return freieFahrt;
    }
```

Ein direkter Zugriff im Hauptprogramm auf die Variable **`ampelAuto.freieFahrt`** ist nicht zulässig und wird vom Compiler mit folgendem Kommentar verhindert

```
'boolean Ampel::freieFahrt' is private within this context
```

Leider ist diese Kapselung beim Arduino etwas löcherig, da wir auf die Pins aus unserem Programm ungehindert zugreifen können; Zugriffe verhindert der Compiler nur auf Variable.

Um eine Klasse im Hauptprogramm verwenden zu können, erzeugen wir am Programmanfang ein Objekt dieser Klasse. Wir nehmen die Gussform **`Ampel`** und erzeugen einen Abguss, ein Objekt namens **`ampelAuto`** :

```
Ampel  ampelAuto(ledRotPin, ledGelbPin, ledGruenPin);
```

Im Hauptprogramm rufen wir die Methoden auf, z. B.:

```
ampelAuto.zeigeGelb();
```

oder

```
if (ampelAuto.getFreieFahrt() && ampelFussgaenger.getFreieFahrt() ) ...  .
```

Das gesamte Steuerungsprogramm ist in **`Funktionsplan_oo.ino`** enthalten.

Wenn unser Steuerungsprogramm für die Ampel alle Anforderungen erfüllt, gibt es keinen Grund, weiterzuentwickeln. Die wichtigste Anforderung an ein Rechnerprogramm ist erneut erfüllt: Einfach lesbar, wenn man sich mit den Klassen in C++ beschäftigt hat. Bei größeren Projekten wäre es sinnvoll, jetzt die erforderlichen Ressourcen für die unterschiedlichen Programmieransätze zu bestimmen und bei speicher- oder zeitkritischen Echtzeitanwendungen die geeignetste Version auszuwählen. Bei diesem Minibeispiel ergeben sich keine signifikanten Unterschiede bezüglich Speichernutzung oder Ausführungszeit. Wenn man häufig gleichartige Steuerungen entwickelt, und dies u. a. mit mehreren Personen, ist es sinnvoll, das Programm zu modularisieren und die Teile, die die Klassen beschreiben, in einzelne Dateien auszulagern. Dann können alle auf sie zugreifen und sie verwenden. Nicht abgesprochene Änderungen, die im betrieblichen Alltag zu Friktionen führen können, entfallen.

## Modularisierung und Wiederverwendung als Bibliothek

Der letzte Schritt bei der Entwicklung unseres Steuerungsprogramms besteht darin, dass wir eine Bibliothek für den Arduino erstellen.

Wir erstellen eine Textdatei **Ampel.h** und fügen am Anfang folgenden Zeilen ein:

```
#ifndef Ampel_lib_H
#define Ampel_lib_H

#include <Arduino.h>
```

Mit den ersten zwei Zeilen stellen wir sicher, dass die Datei nur einmal gelesen wird, indem wir die Variable **Ampel_lib_H** erzeugen. Bei einem zweiten Aufruf dieser Datei ist diese Variable dann vorhanden und die Anweisung **#ifndef** bewirkt, dass die Datei nicht ein zweites Mal ausgeführt wird. In der letzen Zeile müssen wir, sozusagen als Gegenstück zur Abfrage in der ersten Zeile, den Block abschließen, mit
**#endif**

Die Anweisung **#include <Arduino.h>** ist erforderlich, damit auch in diesem Block die Arduino-Bibliothek verfügbar ist. Es folgt in der vierten Zeile die bereits bekannte Deklaration unserer Schnittstelle.

```
class Ampel {
  private:
    byte pinRot;
    byte pinGelb;
    byte pinGruen;
    boolean freieFahrt;

  public:
    Ampel(byte pRot, byte pGelb, byte pGruen);
    void init();
    void zeigeRot();
    void zeigeRotGelb();
    void zeigeGelb();
    void zeigeGruen();
    void zeigeAllesAus();
    boolean getFreieFahrt();


}; //Ende Ampel.h
```

Die ausführbaren Anweisungen schreiben wir in die Datei **Ampel.cpp**. Als erstes fügen wir **#include "Ampel.h"** ein, um auf **Ampel.h** zugreifen zu können und dann folgen die Methoden der Klasse.

```
#include "Ampel.h"

Ampel::Ampel(byte pRot, byte pGelb, byte pGruen) {
      pinRot   = pRot;
      pinGelb  = pGelb;
      pinGruen = pGruen;

      init();
    }

...

boolean Ampel::getFreieFahrt() {
      return freieFahrt;
    }

// Ende Ampel.cpp
```

Der doppelte Doppelpunkt **::** ist der Bereichsauflösungsoperator, damit der Compiler weiß, für welche Klasse die Methode definiert wird. Als nächstes erstellen wir mit Windows einen ZIP-komprimierten Ordner **Ampel.zip** und fügen die zwei Dateien **Ampel.h** und **Ampel.cpp** ein. Diesen Ordner fügen wir zur Arduino IDE hinzu ( **Sketch/Bibliothek einbinden/.ZIP-Bibliothek hinzufügen** ). Als letzten Schritt müssen wir in unser Steuerungsprogramm noch die Anweisung aufnehmen, unsere Bibliothek auch zu verwenden:

```
#include "Ampel.h"
```

Jetzt können wir die Klasse Ampel als Bibliothek einfach selbst benutzen oder auch weitergeben, ohne sie jeweils in unser Programm mit dem Editor einfügen zu müssen. Wenn wir es perfekt machen wollen, ergänzen wir die Bibliothek noch um einen Hilfetext und ein Beispiel.

Das gesamte Steuerungsprogramm ist in **`Ampel_oo_library.ino`** , die Bibliotheken in **`Ampel.zip`** und **`Ampel2.zip`** enthalten.

## Fazit

Der Arduino ist für viele Steuerungsaufgaben sehr gut geeignet, wenn wir keine Internetverbindung benötigen und kein ausgesprochen mechanisch oder elektrisch robuster Aufbau erforderlich ist. Die Auswahl an Sensoren und Aktuatoren ist sehr groß und der Anschluss einfach. Beim Erstellen des Steuerungsprogramms kann man mit einer einfachen, linearen Struktur beginnen und Zeitverzögerungen mittels **`delay()`** realisieren. Bei komplexeren Aufgabenstellungen wird das Programm jedoch übersichtlicher, wenn man den Ablauf in einzelne Schritte gliedert, die nacheinander ausgeführt werden. Auch wird das System leistungsfähiger, da der Prozessor während der Wartephasen nicht blockiert wird, sondern andere Aufgaben ausführen kann. Die Möglichkeiten der objektorientierten Programmierung, die der avr-gcc Compiler der Arduino IDE bietet, kann man nutzen, um den Code wiederverwendbar zu gestalten. Als Bibliothek ist er dann auch einfach in Projekten einsetzbar, ohne dass man sich mit allen Details auseinandersetzen muss.
Berücksichtigen sollte man jedoch zum einen, dass auch bei der objektorientierten Programmierung am Anfang einige Hürden zu überwinden und ggf. neue Fehlermeldungen des Compilers zu entschlüsseln sind.

## Literatur

[1] Peter Beater - Physical Computing - Automatisieren mit dem Arduino - Eine kurze Einführung in die Welt eingebetteter Systeme, BoD Norderstedt, 2020

[2] Peter Beater - Grundkurs der Steuerungstechnik mit CODESYS Grundlagen und Einsatz Speicherprogrammierbarer Steuerungen, BoD Norderstedt, 2021