

Implementierung einer Ampelsteuerung

**Klausurersatzleistung Embedded Systems in der
Automation**

im Studiengang Elektrotechnik

an der Dualen Hochschule Baden-Württemberg Mosbach

von

Cedric Franke

20.06.2022

Matrikelnummer	2685192
Ausbildungsfirma	Mlog Logistics GmbH, Neuenstadt am Kocher
Gruppenmitglieder	Niklas Stein, Sven Mößner, Timo Kempf
Dozent:in	Claudia Heß

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine Klausurersatzleistung mit dem Thema: *Implementierung einer Ampelsteuerung* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1	Einleitung	1
1.1	Aufgabenstellung	1
1.2	Vorgehensweise	3
2	Hardware	4
2.1	Allgemeines	4
2.2	Aufbau	4
2.3	Pin-Belegung	5
3	Styleguide	6
3.1	Bezeichnungen	6
3.2	Kommentare	6
4	Systemarchitektur	7
4.1	Klassendiagramm des Projekts	7
4.2	Zustandsautomat	10
5	State Pattern	11
5.1	Context, Interface und Concrete Klassen	11
5.2	Singleton	11
6	Hardwarezugriff - GPIO	14
7	Wechsel zwischen Hardware- und Softwarebetrieb	17
7.1	Definition	17
7.2	Umsetzung	17
7.2.1	Ausgabeformat	18
7.2.2	Eingabeformat	19
Abbildungsverzeichnis		III
Quellcodeverzeichnis		IV

Quellen

V

1 Einleitung

1.1 Aufgabenstellung

Die Aufgabe dieser Klausurersatzleistung ist es, mit Hilfe eines STM32F401 Nucleo Boards eine Ampelsteuerung zu implementieren. Die Ampelsteuerung soll Softwareseitig über `print`-Befehle und Eingabe über die Tastatur, sowie Hardwareseitig über das Nucleo Board per Taster und LED's funktionieren.

Als Vorgabe und Ausgangssituation dient ein Laborskript, welches die Schritte der Implementierung, als auch die zu erstellenden Diagramme beinhaltet.

Folgende Meilensteine sind während der Bearbeitungszeit zu bearbeiten:

- Für die Ansteuerung der LED's und das Auslesen der Taster soll eine `GPIO`-Klasse entworfen werden, welche auf die `GPIO`-Register des STM32F401 zugreift.
- Damit die Implementierungen `GPIO`-Register Funktionen verwendet werden können, müssen zwei weitere Klassen für die LED's und Taster erstellt werden. Innerhalb dieser Klassen werden die `GPIO`-Funktionen aufgerufen.
- Zur einfachen Verständlichkeit der zu implementierenden Ampelsteuerung, soll ein Zustandsautomat in UML angefertigt werden. Anschließend soll aus dem Zustandsautomat ein Klassendiagramm mit Hilfe von State-Pattern angefertigt werden.
- Im nächsten Schritt soll die Ampelsteuerung mit Hilfe der State-Pattern implementiert werden. Beginnend mit dem "äußerer Zustandsautomat", sprich der Auswahl der beiden Betriebszuständen und endend mit dem inneren Zustandsautomat. Zusätzlich kann eine Auswahl über den Benutzer implementiert werden.

tiert werden, in dem er zwischen Softwareein-/ ausgabe und Hardwareein-/ ausgabe entscheiden kann.

- Abschließend sollen alle implementierten Klassen, welche benötigt werden in die Ampelsteuerung integriert werden.

Über den ganzen Implementierungszeitraum hinweg, soll zudem eine Quellcodedokumentation mit Doxygen erstellt werden.

1.2 Vorgehensweise

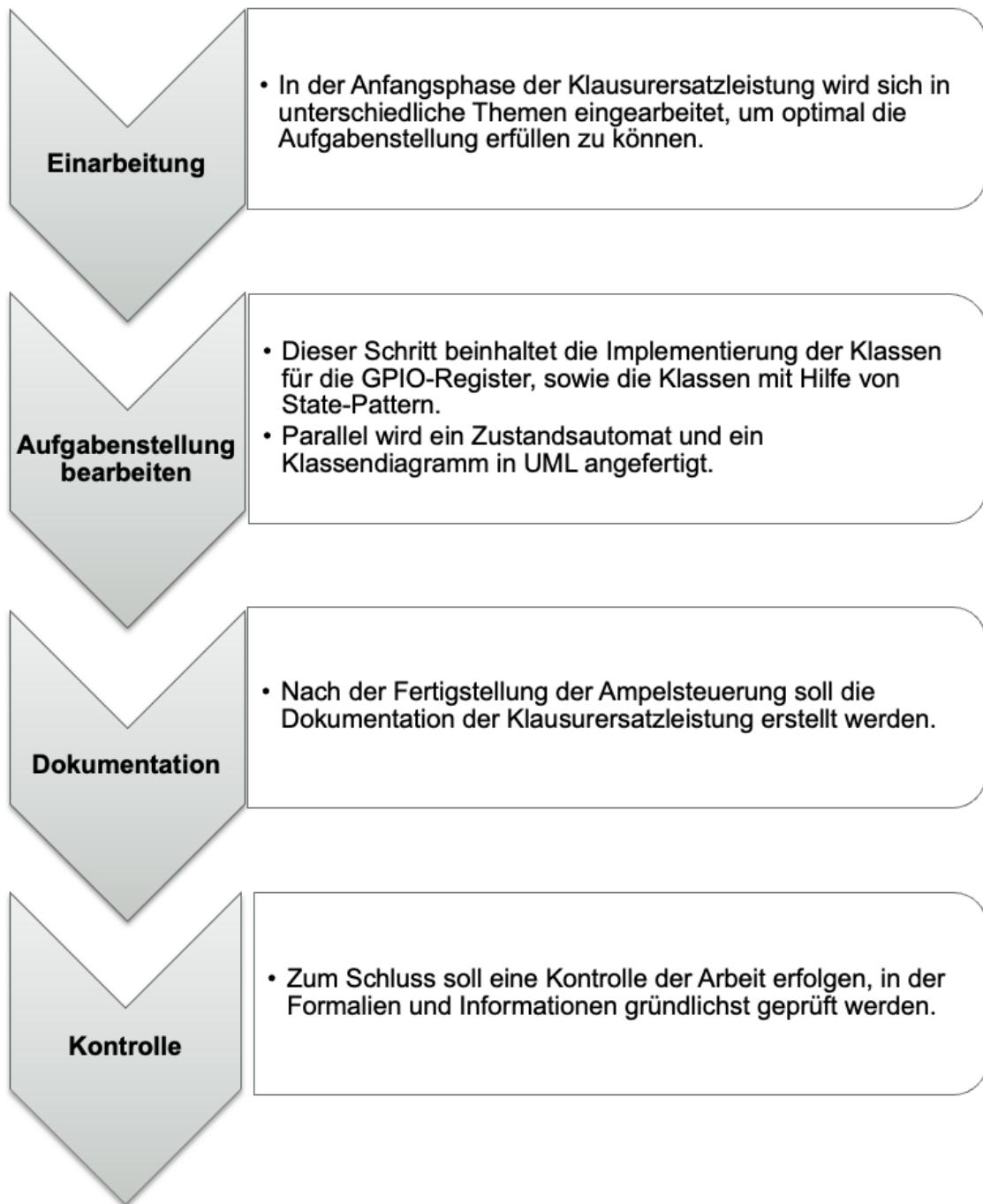


Abbildung 1.1: Vorgehensweise der Klausurersatzleistung [ED22]

2 Hardware

2.1 Allgemeines

Verwendet wird das STM32F401RE Nucleo Board der Firma STMicroelectronics. Weiterhin wird ein Shield verwendet, welches auf den Microcontroller aufgesteckt werden kann. So wird eine Verbindung zwischen einigen GPIOs des Boards und dem Shield ermöglicht. Des Weiteren werden LEDs und Buttons mitgeliefert. Zuletzt wird noch ein USB-Kabel zur Verbindung zwischen Laptop und Mikrocontroller benötigt. Dieses versorgt den Controller mit Spannung und ermöglicht eine serielle Kommunikation, welche später benötigt wird.

2.2 Aufbau

Im Folgenden wird nun auf den Aufbau der Hardware eingegangen. Das Shield wird auf das Board aufgesteckt. Die mitgelieferten LEDs und Buttons können nun einfach am Shield eingesteckt werden und erhalten damit eine leitende Verbindung zu Versorgungsspannung, Masse und einer Signalleitung. Die Verbindung über das Shield bietet eine höhere Festigkeit und Stabilität gegenüber einzelnen Jumperverbindungen, welche direkt am Board gemacht werden könnten.

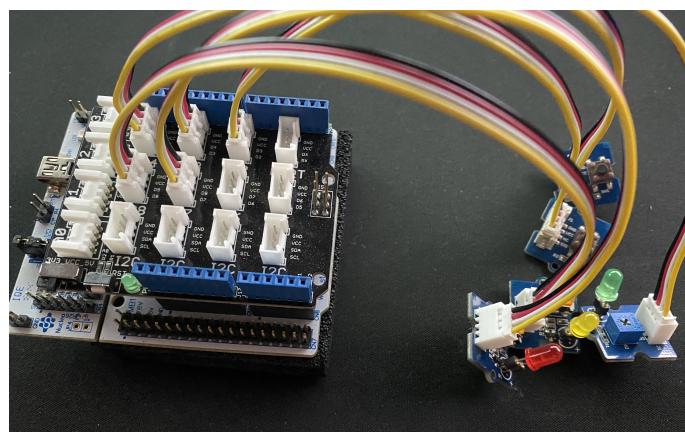


Abbildung 2.1: Hardwareaufbau der Ampelsteuerung [ED22]

2.3 Pin-Belegung

Da der Arbeitsaufwand dieses Projektes auf vier Studierende aufgeteilt wird, muss abgestimmt werden, welcher Pin jeweils den LEDs und den Buttons zugeordnet werden soll. Deshalb wird gleich zu Beginn des Projektes eine Vereinbarung der Pinbelegungen getroffen. Diese sind in der untenstehenden Tabelle zu sehen.

Port	Pin	Shield	Hardware
A	8	D7	LED Rot
A	9	D8	LED Orange
A	10	D2	LED Grün
B	3	D3	Taster 1
B	5	D4	Taster 2

3 Styleguide

3.1 Bezeichnungen

3.2 Kommentare

4 Systemarchitektur

4.1 Klassendiagramm des Projekts

In diesem Abschnitt wird das Klassendiagramm der Ampelschaltung (siehe Abbildung 4.1) erläutert. Es wird darauf eingegangen, wie die einzelnen Klassen in Beziehung zueinanderstehen und wie diese gruppiert werden können.

Das Klassendiagramm lässt sich in vier Bereiche unterteilen:

- Auswahl Betriebsmodus (blau)
- Steuerung Farben (grün)
- Output (rot)
- Input (schwarz)

Der Bereich „Auswahl Betriebsmodus“ bildet die Logik der beiden Betriebsmodi ab, in denen sich die Ampel befinden kann. Zum einen der aktive Modus, in welchem die Ampel die Farben durchschaltet und zum anderen den Modus, in welchem die Ampel orange blinkt. Der Bereich „Steuerung Farben“ übernimmt die Aufgabe die Zustände im Hinblick auf die Farben der Ampel bereitzustellen. Die Ampel kann Rot ausgeben, Rot-Organge, Orange, Grün oder ausgeschaltet sein. Der Output übernimmt die Funktion, die einzelnen Farben zu setzen und dies entweder software- oder hardwaremäßig. Der Input übernimmt die Aufgabe, die Benutzereingabe bereitzustellen, die ebenso entweder software- oder hardwaremäßig eingegeben werden kann. Die Klasse GPIO, die nicht in einen der Bereiche zugeordnet wird, bildet die Schnittstelle für den Hardwarezugriff ab.

Hierarchisch ganz oben ist die Kontextklasse „Traffic Light“ zu sehen. Ein Objekt

dieser Klasse wird in der Main erstellt und beinhaltet die Logik der gesamten Ampelschaltung. Wird die Handle Funktion dieses Objekts aufgerufen, läuft die Ampelschaltung wie gefordert ab. Die Aufgabe Benutzereingaben einzulesen, wird an das Interface „InputFormat“ delegiert. Je nachdem, ob eine Hardware angeschlossen ist, liefern die Klassen „SoftwareInput“ und „HardwareInput“ den jeweiligen Tasterwert, bzw. die Tastatureingabe. Die Klasse „UserButtons“ frägt beide Pins (für die beiden Taster) ab und stellt der Klasse „HardwareInput“ nur einen Wert zur Verfügung, der die beiden Tasterzustände abbildet. Damit die Klasse „UserButtons“ auf die Registerinhalte des Mikrocomputers zugreifen kann, steht sie in Beziehung mit der Klasse „GPIO“. Ein Objekt der „Traffic Light“ Klasse kann den Zustand in Betrieb oder außer Betrieb annehmen. Über das Interface „state“ werden die Zustandsklassen „flashing“ und „active“ hierfür bereitgestellt. Diese beiden Klassen greifen über das Interface „LightControl“ auf die Farbenklassen „Off“, „Red“, „RedAmber“, „Amber“ und „Green“ zu. Diese Unterklassen sind dafür verantwortlich, dass die Ampel den Farbzustand annimmt. Des Weiteren bieten diese Klassen die Funktion an, Objekte für den nächsten Farbzustand zurückzuliefern. Noch eine Hierarchieebene weiter darunter greifen die Farbklassen über das Interface „OutputFormat“ auf die „SoftwareOutput“ bzw. auf die Klasse „HardwareOutput“ zu. Diese realisieren die Ausgabe der Ampelschaltung. Im Fall, dass Hardware angeschlossen ist, greift die „HardwareOutput“ Klasse über die „UserLEDs“ Klasse auf die „GPIO“ Klasse zu. Damit werden die entsprechenden Bits in den Registern des Mikrocontrollers gesetzt, damit die entsprechenden Pins der LEDs auf HIGH geschaltet werden können.

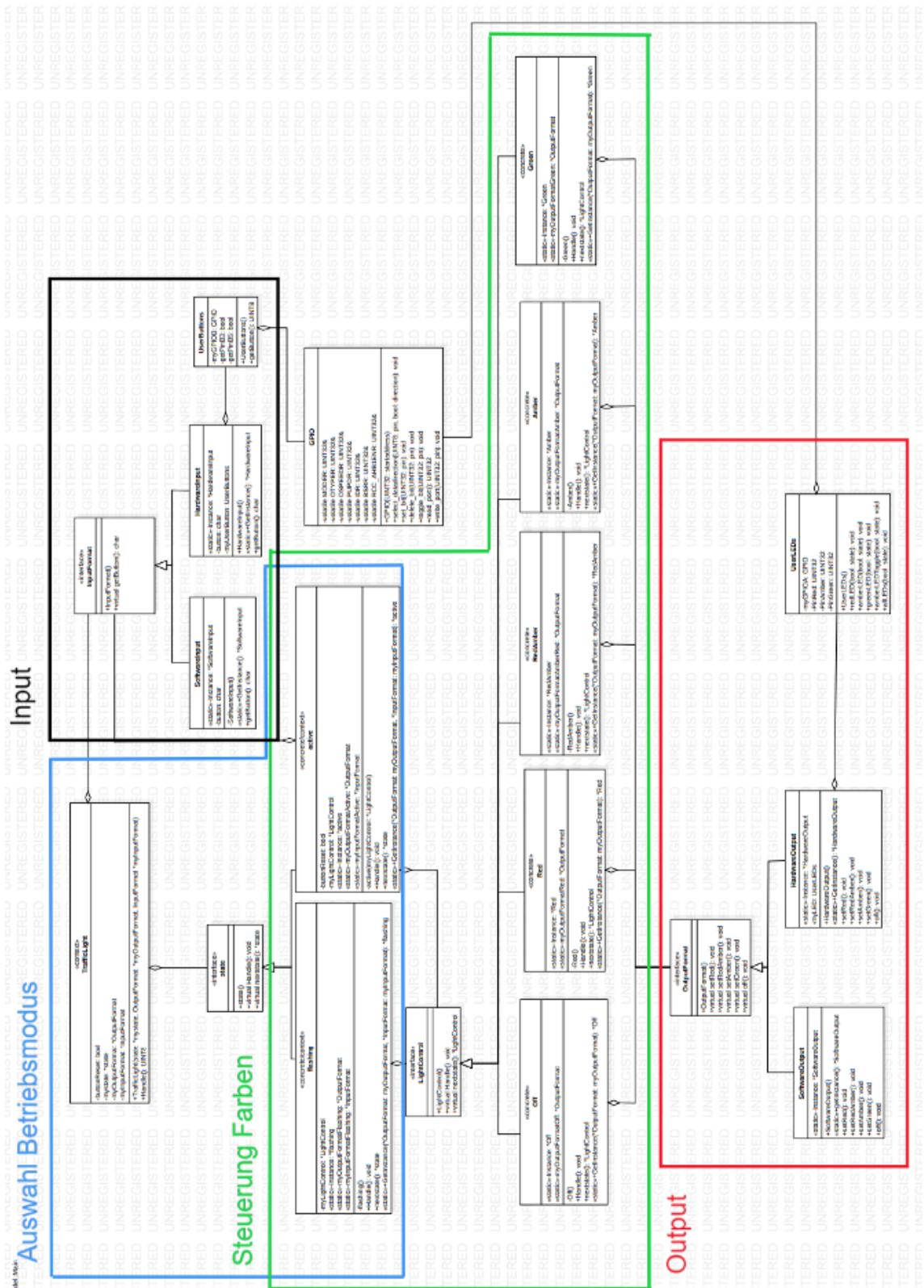


Abbildung 4.1: Klassendiagramm der Ampelsteuerung [ED22]

4.2 Zustandsautomat

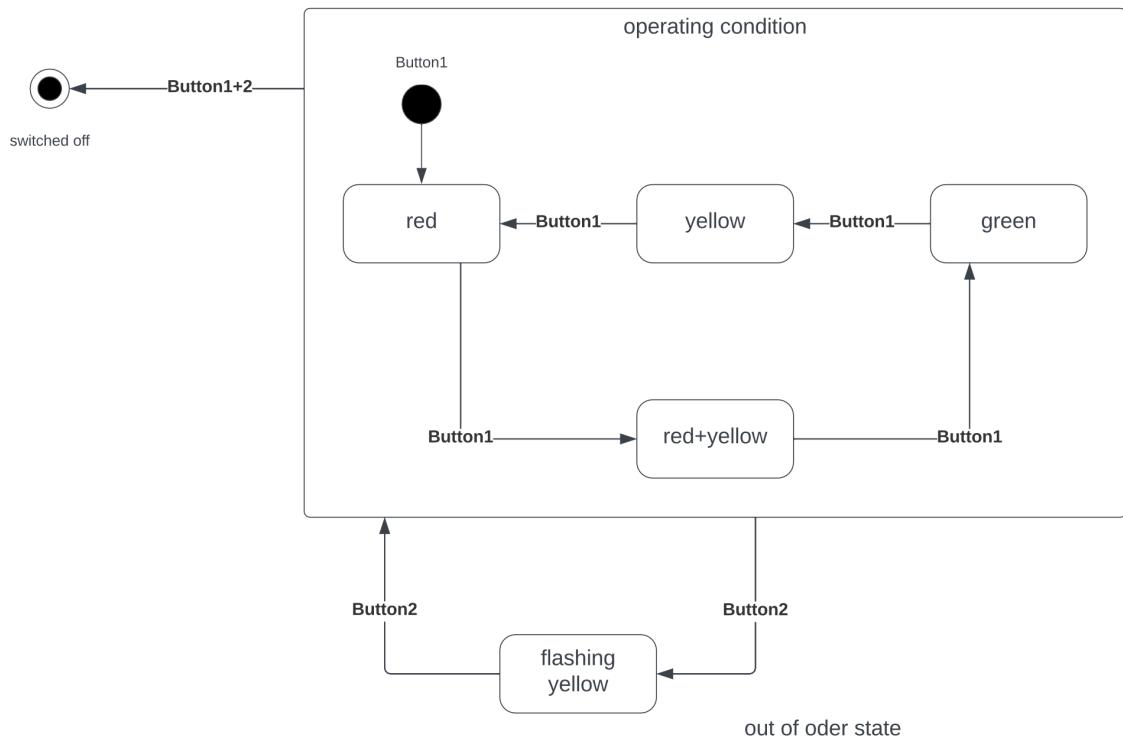


Abbildung 4.2: Zustandsautomat der Ampelsteuerung [ED22]

Ein Zustandsdiagramm ist eins der 14 Diagrammartarten der Sprache UML für Software und andere Systeme. Es ermöglicht den Entwicklern sich einen Gesamtüberblick zu verschaffen, über die Zustände und Operationen, welche in dem Projekt/Programm erfolgen sollen.

Die zu implementierende Ampelsteuerung soll zwei Zustände „In Betrieb“ und „Außer Betrieb“ annehmen können. Im Betrieb soll die Ampelsteuerung die Ampelphasen Rot, Rot-Orange, Grün und Gelb durchlaufen (siehe Abbildung 4.2 mittig). Die Ampelphasen sollen per Taster „T1“ oder Tastatureingabe „F“ gesteuert werden. Zwischen jeder Ampelphase kann die Ampelsteuerung mit dem Taster „T2“ oder Tastatureingabe „B“ in den „Außer Betrieb“ Zustand gesetzt werden. Innerhalb dieses Zustandes, soll die orange LED blinken (siehe Abbildung 4.2 unten). Immer wenn die Ampelsteuerung neugestartet oder in den Betriebsmodus „In Betrieb“ wechselt, soll sie in der roten Ampelphase starten.

5 State Pattern

5.1 Context, Interface und Concrete Klassen

In der vorliegenden Ampelschaltung werden zwei State Patterns verwendet. Zudem wird beim Input und Output sich eines Interfaces bedient, welches Polymorphie zulässt. Das heißt, je nachdem ob Hardware angeschlossen ist, werden die Hardware oder Software Klassen verwendet. Näheres dazu wird in dem Kapitel „Wechsel zwischen Hardware- und Softwarebetrieb“ beschrieben. Das hierarchisch höhere State Pattern besteht aus der Kontextklasse „TrafficLight“, dem Interface „state“ und den Concrete Klassen „flashing“ und „active“. Je nach Benutzereingabe verweist der Zeiger vom Typ „state“ auf eine Instanz der Klasse „flashing“ oder „active“. Wird die Handle Funktion des Interfaces aufgerufen, ist die Ampel entweder im konkreten Zustand „in Betrieb“ oder „außer Betrieb“. Fordert der Benutzer ein Wechsel des Betriebsmodus, so ist der nächste Zustand der jeweils andere Betriebsmodus. Das zweite State Pattern wechselt zwischen den Farbzuständen. Dieses State Pattern besteht entweder aus der Kontextklasse „flashing“ oder „active“, dem Interface „LightControl“ und den Concrete Klassen („Off“, „Red“, „RedAmber“, „Amber“, „Green“). Die Klasse „active“ ruft die Funktionen „nextstate“ des Interfaces auf, damit die Ampelfarben durchgeschaltet werden. Im außer Betrieb Zustand wechselt der Farbzustand der Ampel zwischen Off und Amber.

5.2 Singleton

Das Singleton Design Pattern hilft in der objektorientierten Programmierung dem Entwickler mit vielfachen Vorlagen das Lösen von Programmieraufgaben. Es ist sehr leistungsfähig und gehört zu der Kategorie der Erzeugungsmuster unter den Design Pattern. Der Ausdruck Singleton wird oftmals als „Einzelstück“ betitelt. Dieses Synonym bildet sehr gut die Funktion eines Singeltons ab, da die Aufgabe darin besteht nicht mehr als ein Objekt einer Klasse zu erstellen. Wurde mit der Klasse über das Singleton Pattern nur eine Instanz der Klasse erzeugt, sorgt es auch

dafür, dass es ledlich bei dieser einen Instanz der Klasse bleibt. Oftmals dient eine Funktion, wie `GetInstance` dafür, dass nur eine Instance der Klasse erstellt wird.

Im Bezug auf die Ampelsteuerung wurde das Singleton Pattern in sieben unterschiedlichen Klassen verwendet. Diese lauten wie folgt:

- active.h
- flashing.h
- Red.h
- Amber.h
- RedAmber.h
- Green.h
- Off.h

Da der Aufbau in jeder Klasse identisch ist, folgt die Erläuterung im Code nun am Beispiel der Klasse `Red.h`.

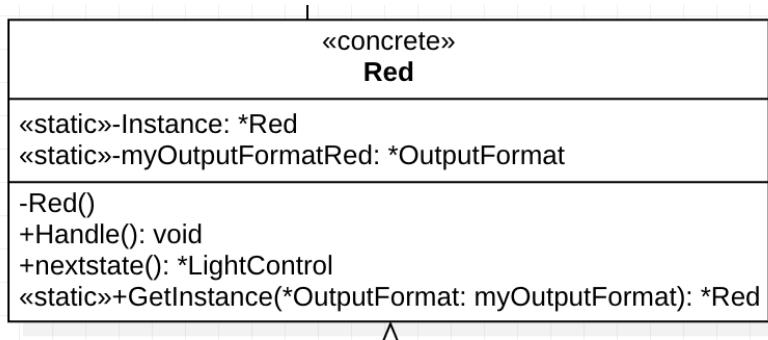


Abbildung 5.1: Singleton Patten am Beispiel der Red.h Klasse [ED22]

In der Abbildung 5.1 sind die Funktionen und Instanzen der `Red.h` Klasse zu sehen, welche mit Hilfe des Singleton Pattern implementiert wurde. Das Schlüsselwort `static` bedeutet einmal, dass die als `static` deklarierte Variable oder Funktion ihren Wert innerhalb zwischen Aufrufen behält und andermal, dass eine globale statische Variable oder Funktion nur in der Datei „gesehen“ wird, in der sie deklariert

wurde. Um das Singleton Pattern nutzen zu können, wird die statische Instanz und die Funktion `GetInstance` benötigt. Die Funktion `GetInstance` hat die Aufgabe, bei jedem Aufruf eine Instance zu erzeugen (siehe Quellcodeausschnitt 5.1).

```
1     ...
2     Red *Red::Instance = NULL;
3
4     Red *Red::GetInstance(OutputFormat *myOutputFormat)
5     {
6         myOutputFormatRed = myOutputFormat;
7         if (Instance == NULL)
8         {
9             Instance = new Red();
10        }
11        return Instance;
12    }
13    ...

```

Listing 5.1: Red.cpp - Singleton in der Klasse Red.h

Wie in Zeile 2 zu erkennen ist, wird die Instanz der Klasse immer auf `NULL` deklariert. In Zeile 4 der cpp-Datei wird ist die Implementierung der Funktion `GetInstance` zu sehen. Innerhalb dieser Funktion wird der übergebene Parameter `myOutputFormat` der Klasse Red zugewiesen. Folgend findet eine `if` Überprüfung statt, ob es schon eine Instance der Klasse Red.h gibt. Wenn nicht, wird in Zeile 9 eine neue Instanz der Klasse Red.h erzeugt. Anschließend besitzt die Funktion `GetInstance` als `return`-Wert die erzeugte Instanz. Wichtig ist es, dass die statische Instanz als `private` und die Funktion `GetInstance` als `public`.

6 Hardwarezugriff - GPIO

Der Hardwarezugriff erfolgt über drei Klassen. Die untergeordnete Klasse `GPIO` greift dabei direkt auf die Register des Mikrocontrollers zu und steuert somit sämtliche Hardwarezugriffe. Die zwei übergeordneten Klassen `UserLEDs` und `UserButtons` greifen dann auf die Klasse `GPIO` zu und steuern diese. Übergeordnet heißt hier jedoch nicht, dass es sich um Oberklassen einer Vererbung handelt. Die hier herrschende Beziehung ist eine Aggregation. Außer zu diesen beiden Klassen hat die `GPIO`-Klasse keine weiteren Verbindungen. Eine Hardwareansteuerung kann somit nur durch die beiden genannten übergeordneten Klassen durchgeführt werden.

Nun wird etwas genauer auf die die `GPIO`-Klasse eingegangen. Sie ist wie folgt aufgebaut:

GPIO
<pre> -volatile MODER: uint32& -volatile OTYPER: uint32& -volatile OSPEEDR: uint32& -volatile PUPDR: uint32& -volatile IDR: uint32& -volatile BSRR: uint32& -volatile RCC_AHB1ENR: uint32& </pre>
<pre> +GPIO(uint32: startaddress) +select_datadirection(uint8: pin, bool: direction): void +set_bit(uint32: pin): void +delete_bit(uint32: pin): void +toggle_bit(uint32: pin): void +read_port(): uint32 +write_port(uint32: pin): void </pre>

Abbildung 6.1: GPIO Klasse [ED22]

Alle Attribute dieser Klasse sind als „private“ gekennzeichnet. Das liegt daran, dass es sich bei den Attributen um Zeiger handelt, mit welchen man auf die `GPIO`-Register des Mikrocontrollers zugreifen kann. Hier wird das Prinzip des Information Hidings angewendet, um zu verhindern, dass die Register von außen manipuliert werden könnten. Weiterhin wird den Attributen das Schlüsselwort `volatile` vorangestellt. Dieses gibt dem Compiler die Information, dass sich der Inhalt auf den die Zeiger zeigen, ändern kann, auch wenn das Programm nicht schreibend darauf zu-

greift.

Der Konstruktor der Klasse konfiguriert einen GPIO-Port. Ihm muss lediglich die Startadresse des gewünschten Ports übergeben werden. Diese Startadresse kann dem Datenblatt auf Seite 53 entnommen werden. Auf Basis der Startadresse adressiert dieser dann die benötigten Register. Weiterhin wird dort das Clock Signal des ausgewählten Ports aktiviert. Über die Methode `select_datadirection()` kann dann für jeden Pin des Ports einzeln ausgewählt werden ob, dieser als Input oder als Output fungieren soll. Dazu muss der Methode nur die Pinnummer und die Art des GPIOs übergeben werden. Mit `set_bit()` kann ein ausgewählter Output gesetzt werden. Dazu muss lediglich die Pinnummer bekannt sein. `delete_bit()` macht genau das Gegenteil. Dort wird ein Output zurückgesetzt. `toggle_bit()` vereint beide Methoden. Dort wird der Status des Outputs abgefragt und danach umgekehrt. Mit `read_port()` werden die Zustände jedes einzelnen Pins des gesamten Ports gelesen und als 32-Bit Wert zurückgegeben. Bei der Ausführung von `write_port()` hingegen, wird der gesamte Port beschrieben. Wirksam ist der Schreibzugriff jedoch nur auf definierte Outputs.

Nun wird auf die Klasse „UserLEDs“ betrachtet. Das dazugehörige Klassendiagramm kann folgender Abbildung entnommen werden:

UserLEDs
-myGPIOA: GPIO
-PinRed: UINT32
-PinAmber: UINT32
-PinGreen: UINT32
+UserLEDs()
+redLED(bool state): void
+amberLED(bool state): void
+greenLED(bool state): void
+amberLEDToggle(bool state): void
+allLEDs(bool state): void

Abbildung 6.2: UserLEDs Klasse [ED22]

Auch hier sind alle Attribute als private definiert, um das Information Hiding zu wahren. Das erste Attribut ist eine Instanz der `GPIO`-Klasse. Es dient als Schnittstelle zur Ansteuerung der Pins, an welchen die LEDs angeschlossen sind. In den restlichen drei Attributen werden die jeweiligen Pinnummern der LEDs hinterlegt. Dies

erfolgt im Konstruktor über eine Initialisierungsliste. Danach wird dort noch, über die `select_datadirection()`-Methode der `GPIO`-Klasse festgelegt, dass es sich bei den vorliegenden Pins um Outputs handelt. Mit den restlichen Methoden werden dann die LEDs angesteuert. Ihnen muss lediglich der gewünschte Zustand übergeben werden.

Zuletzt wird die Klasse `UserButtons` betrachtet. Auch hier wird ein UML Klassendiagramm zur Erläuterung herbeigezogen:

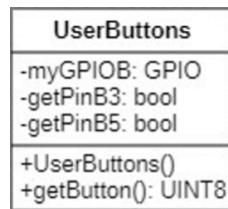


Abbildung 6.3: UserButtons Klasse [ED22]

Dort wird auch wieder eine private Instanz der Klasse `GPIO` erstellt. Damit können die Buttons abgefragt werden. Weiterhin existieren zwei private Methoden, welche den Status des jeweiligen Buttons zurückgeben. Sie werden in der Methode `getButton()` aufgerufen. Dort erfolgt eine Entprellung. Dazu wird der Wert zunächst ein erstes Mal abgefragt. Danach wird 3000 Schleifendurchgänge gewartet und der Status erneut abgefragt. Liefern beide Abfragen das gleiche Ergebnis, so wird der Wert übernommen. Andernfalls wird die Entprellung erneut durchgeführt. Wie im Klassendiagramm zu sehen hat die Methode einen Rückgabewert vom Datentyp `UINT8`, dies entspricht einem 8 Bit Charakter. Je nachdem, welcher Button nun gedrückt wird, wird ein bestimmter Rückgabewert zurückgegeben. Eine Auflistung aller möglichen Rückgabewerte kann folgender Tabelle entnommen werden:

Welche Buttons gedrückt?	Rückgabewert
Keiner	O
Nur Button an PB3 (D3)	B
Nur Button an PB5 (D4)	F
Beide Buttons	X

7 Wechsel zwischen Hardware- und Softwarebetrieb

7.1 Definition

Wie bereits erwähnt, soll es dem Anwender möglich sein, die Ampelsteuerung sowohl mit Hardware, als auch komplett ohne Hardware nutzen zu können. Um die beiden Fälle näher zu erläutern, wird der Hardware- und Softwarebetrieb im Folgenden konkret spezifiziert.

- **Hardwarebetrieb:** Der Programmcode der Ampelsteuerung wird auf das Nucleo Board geflasht und auf diesem ausgeführt. Die Nutzereingabe erfolgt über zwei an das Board angeschlossenen Taster. Die Ausgabe der Ampelfarben erfolgt über die drei angeschlossenen LEDs.
- **Softwarebetrieb:** Das Programm der Ampelsteuerung wird auf dem PC ausgeführt. Die Nutzereingabe erfolgt über die Tastatur. Die Tasten „F“ und „B“ repräsentieren die beiden Hardwaretaster. Die Taste „X“ steht für das Betätigen beider Taster gleichzeitig. Das Einlesen der Tastatureingaben erfolgt über den Eingabestream `cin`, die Ausgabe der Ampelfarben über den Ausgabestream `cout` in der Kommandozeile.

7.2 Umsetzung

Ob die Ampelsteuerung im Hardware- oder Softwarebetrieb arbeitet, wird über Pointer bestimmt, welche in der `main.cpp` des Programmes angelegt und bis zur relevanten Stelle im Programmcode durch alle Klassen übergeben werden. Ob die Pointer für den Hardware- oder Softwarebetrieb initialisiert werden, wird wiederum über eine Präprozessoranweisung festgelegt, welche der Nutzer setzen oder auskommen kann. Dies ist in 7.1 dargestellt.

```

1   ...
2   #define _HARDWAREPRESENT
3   int main(){
4     #ifdef _HARDWAREPRESENT
5       OutputFormat *myOutputFormat =
6         HardwareOutput::GetInstance();
7       InputFormat *myInputFormat =
8         HardwareInput::GetInstance();
9     #else
10    OutputFormat *myOutputFormat =
11      SoftwareOutput::GetInstance();
12    InputFormat *myInputFormat =
13      SoftwareInput::GetInstance();
14  #endif
15  ...

```

Listing 7.1: main.cpp - Hardwarebetrieb oder Softwarebetrieb

Im weiteren Programmcode wurde die Umschaltung zwischen Hardware- und Softwarebetrieb nochmals aufgeteilt. Zum einen wird beim Ausgabeformat der Ampelfarben zwischen Hardware- und Softwareausgabe, zum anderen beim Eingabeformat der Nutzereingabe zwischen Hardware- und Softwareeingabe unterschieden.

7.2.1 Ausgabeformat

7.1 zeigt den Ausschnitt aus dem Klassendiagramm, welcher für die Umschaltung zwischen Hardware- und Softwareausgabe zuständig ist. Die Klasse „OutputFormat“ ist eine Interface Klasse und beinhaltet ausschließlich virtuelle Methoden. Die Unterklassen „SoftwareOutput“ und „HardwareOutput“ erben von der Oberklasse „OutputFormat“ und verwenden deren Interface. Die Concrete Klassen des übergelagerten State Patterns, welche die unterschiedlichen Ampelfarben repräsentieren, besitzen den erläuterten übergebenen Zeiger, welcher in der `main.cpp` initialisiert wurde und auf eine Instanz der Klasse „HardwareOutput“ oder „SoftwareOutput“ zeigt. Je nachdem werden die Methoden von „HardwareOutput“ oder „SoftwareOutput“ aufgerufen, um die Ampelfarben hardware- oder softwareseitig anzusegnen.

Beim Hardwarebetrieb ruft die Klasse „HardwareOutput“ schließlich die Methoden der Klasse „UserLEDs“ auf und steuert so die LEDs an. Befindet sich die Ampelsteuerung im Softwarebetrieb, so sorgen die Methoden in der Klasse „SoftwareOutput“ für die entsprechenden Ausgabestreams in der Kommandozeile.

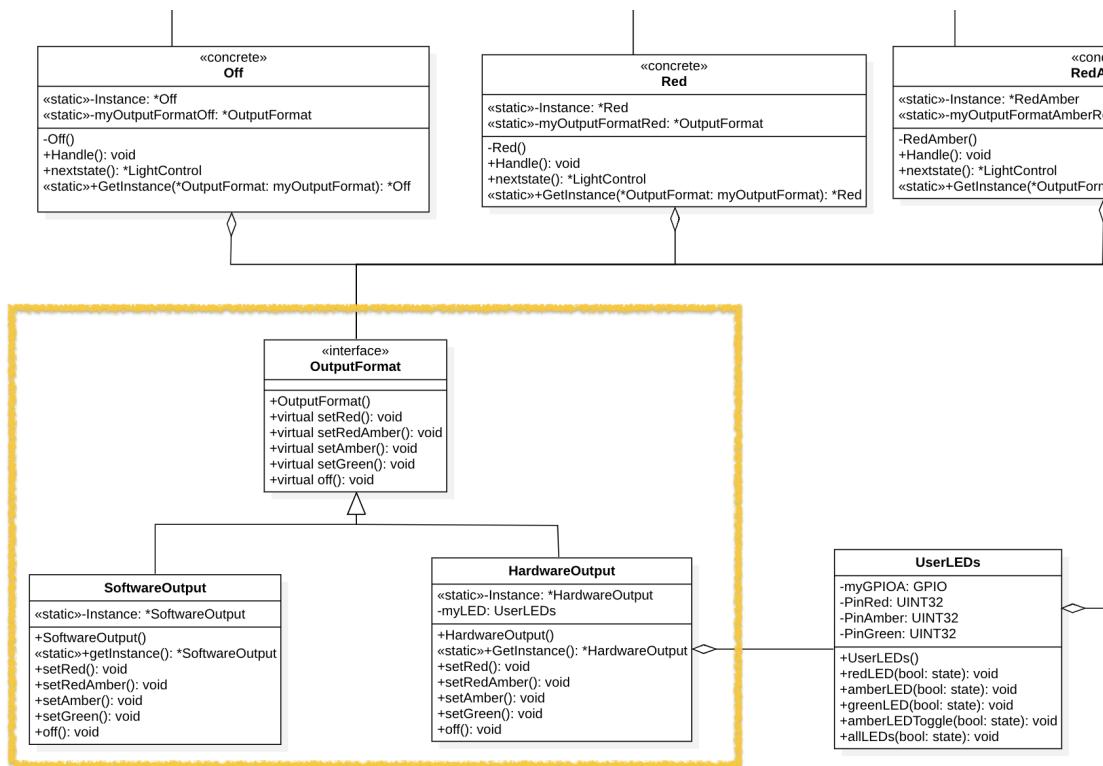


Abbildung 7.1: Auszug aus dem Klassendiagramm - Ausgabeformat [ED22]

7.2.2 Eingabeformat

7.2 zeigt einen Auszug aus dem Klassendiagramm, welcher für den Wechsel zwischen Hardware- und Softwareeingabe zuständig ist. Auch hier sind eine Interface Klasse „InputFormat“ und zwei erbende Unterklassen „HardwareInput“ und „SoftwareInput“, welche das Interface verwenden, vorzufinden. Der Wechsel zwischen Hardware- und Softwareeingabe funktioniert dabei identisch zur Logik der Hardware- und Softwareausgabe, weshalb darauf an dieser Stelle nicht weiter eingegangen wird.

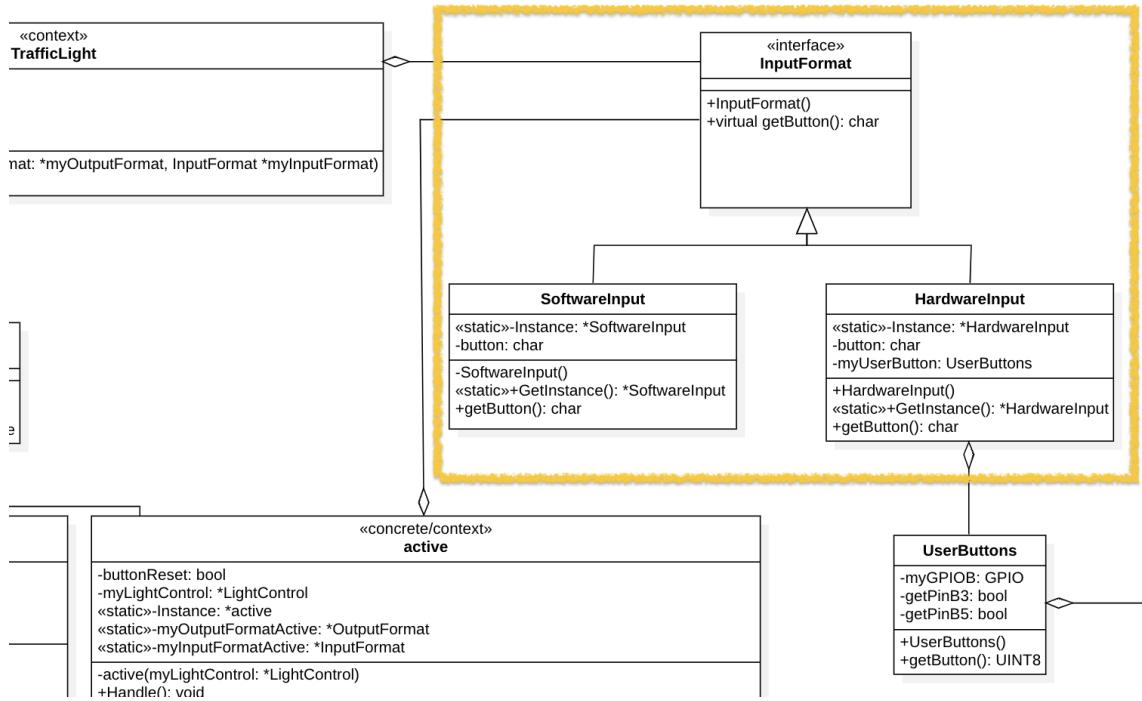


Abbildung 7.2: Auszug aus dem Klassendiagramm - Eingabeformat [ED22]

Abbildungsverzeichnis

1.1	Vorgehensweise der Klausurersatzleistung [ED22]	3
2.1	Hardwareaufbau der Ampelsteuerung [ED22]	4
4.1	Klassendiagramm der Ampelsteuerung [ED22]	9
4.2	Zustandsautomat der Ampelsteuerung [ED22]	10
5.1	Singleton Patten am Beispiel der Red.h Klasse [ED22]	12
6.1	GPIO Klasse [ED22]	14
6.2	UserLEDs Klasse [ED22]	15
6.3	UserButtons Klasse [ED22]	16
7.1	Auszug aus dem Klassendiagramm - Ausgabeformat [ED22]	19
7.2	Auszug aus dem Klassendiagramm - Eingabeformat [ED22]	20

Quellcodeverzeichnis

5.1	Red.cpp - Singleton in der Klasse Red.h	13
7.1	main.cpp - Hardwarebetrieb oder Softwarebetrieb	18

Literatur

[ED22] *Eigene Darstellung.* (Besucht am 20.06.2022).