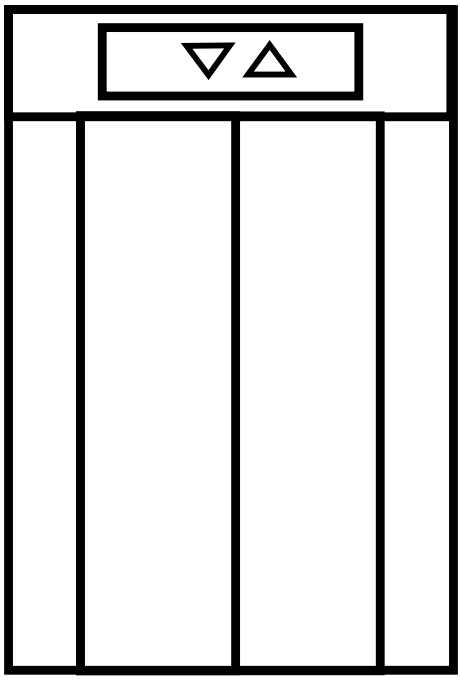
NVS Projekt: Aufzugsteuerung

Ralf Kühmayer

January 2021



Contents

1 Aufgabenstellung

Ein kleines Beispiel für eine Aufzugssteuerung. Jedes Stockwerk und jeder Aufzug wird durch einen Thread repräsentiert. Es gibt einen zentralen Koordinator, der entscheidet, welcher Aufzug in welches Stockwerk fahren soll. In diesen Threads werden die in einem Aufzug oder einem Stockwerk gedrückten Knöpfe, um den Aufzug zu bewegen, vom Anwender simuliert. In der CLI gibt es Optionen, um auszuwählen, wie viele Aufzüge und Stockwerke simuliert werden. Es gibt auch Optionen für die Fahrzeit zwischen den Etagen und ob es die Möglichkeit eines Overrides in der CLI gibt. Des Weiteren kann in der CLI angegeben werden, dass eine Simulation anstelle des REPLs verwendet wird. Es gibt auch Optionen für logging und die Konfiguration kann auch durch eine JSON oder TOML Datei vorgenommen werden.

2 Verwendung

Das Programm wird durch den Aufruf von ./elevator_control in der Kommandozeile gestartet. Beim Aufruf ohne Parameter startet das Programm mit 3 Stockwerken, 1 Aufzug und 3 Sekunden Fahrzeit zwischen den Stockwerken. Das Programm kann entweder durch die Eingabe des end Kommandos in das REPL beendet werden oder durch Abbruch mit Strg+c. Wird das Programm abgebrochen, wird es sofort geschlossen und die Aufzüge arbeiten ihre Queue nicht mehr ab. Wird das Programm jedoch mit dem end Kommando geschlossen, wartet das Programm noch solange, bis alle Aufzüge ihre Queue abgearbeitet haben, bevor es sich schließt.

2.1 Parameter

Das Programm hat einige optionale Parameter, mit welchen der Anwender das Programm Konfigurieren, das Logging ändern und die Bedienung des Programmes verändern kann.

2.1.1 Programmparameter

Programmparameter sind Parameter, welche das Verhalten oder Anzahlen in dem Programm beeinflussen.

-s, --seconds-between-floors $<$ *positive Kommazahl* $>$
Mit dem Parameter -s oder --seconds-between-floors und einer positiven Kommazahl kann der Anwender einstellen, wie lange ein Aufzug von einem Stockwerk zum nächstgelegenen Stockwerk in Sekunden braucht. Wenn der Parameter nicht angegeben wird, ist standardmäßig eine Fahrzeit von 3 Sekunden zwischen den Stockwerken festgelegt.

-f, --floor-number $<$ *positive ganze Zahl* $>$
Mit dem Parameter -f oder --floor-number und einer positiven ganzen Zahl kann der

Anwender einstellen, wie viele Stockwerke es gibt. Wenn der Parameter nicht angegeben wird, sind standardmäßig 3 Stockwerke vorhanden.

-e, --elevators $< positive\ ganze\ Zahl >$
Mit dem Parameter -e oder --elevators und einer positiven ganzen Zahl, kann der Anwender einstellen, wie viele Aufzüge es gibt. Wenn der Parameter nicht angegeben wird, ist standardmäßig nur ein Aufzug vohanden.

-o, --override
Mit dem Parameter -o oder --override, wird eingestellt, dass die Befehle Move und Call auch mit einem override Flag ausgeführt werden können. Wir ein Befehl mit einem Override ausgeführt, hat das zur Folge, dass der ausgewählte Aufzug das angegebene Stockwerk als nächstes anzusteuerndes Stockwerk in seine Queue einfügt.

2.1.2 Simmulationsparameter

Simulationsparameter sind Parameter, mit welchen man das REPL durch eine Simulation ersetzen kann und diese damit auch konfiguriert.

--simulation
Wird der Parameter --simulation gesetzt, dann wird das REPL durch eine Simulation ersetzt. Der Anwender kann diese Simulation durch eine Eingabe der Entertaste beenden, jedoch werden alle Aufzüge ihre Queue noch leeren und dann nach wird das Programm geschlossen.

--simulation-time $< positive\ Kommazahl >$
Mit dem Parameter --simulation-time und einer positiven Kommazahl kann der Anwender einstellen, wie lange es nach einem Befehl dauert, bis der nächste Befehl gesendet wird.

2.1.3 Konfigurationsparameter

Konfigurationsparameter sind Parameter, welche durch eine angegebene Konfigurationsdatei die Programmparameter ersetzen. Es dürfen nicht beide Konfigurationsparameter gleichzeitig verwendet werden. Immer nur einer ohne Programmparameter.

-j, --config-file-json $< Dateipfad >$
Mit dem Parameter -j oder --config-file-json und dem Pfad zu der Konfigurationsdatei kann angegeben werden, dass das Programm mit einer JSON-Datei konfiguriert werden soll. Wird dieser Parameter angegeben, darf kein Programmparameter verwendet werden, Simulations- und Loggingparameter können verwendet werden.

-t, --config-file-toml $< Dateipfad >$
Mit dem Parameter -t oder --config-file-toml und dem Pfad zu der Konfigurationsdatei kann angegeben werden, dass das Programm mit einer TOML-Datei konfiguriert werden soll. Wird dieser Parameter angegeben, darf kein Programmparameter verwendet werden, Simulations- und Loggingparameter können verwendet werden.

2.1.4 Loggingparameter

Mit den Loggingparametern wird das Logging in eine Datei aktiviert und es kann dieses konfiguriert werden. Standardmäßig ist das Logginglevel auf Info gesetzt, das kann jedoch genau so wie die Datei konfiguriert werden.

-l, --log-to-file
Mit dem Parameter -l oder --log-to-file, kann der Anwender einstellen, dass das Logging in eine Datei aktiviert wird. Das default Logginglevel ist Info und es wird standardmäßig in die Datei control.log in dem aktuellen Ordner geschrieben. Existiert diese Datei nicht, wird sie erstellt.

-d, --log-level-debug
Mit dem Parameter -d oder --log-level-debug wird das Logginglevel, für das Loggen in eine Datei vom Level Info auf das Level Debug geändert. Dadurch werden mehr Informationen in der Logdatei gespeichert.

--log-file $< Dateipfad >$
Mit dem Parameter --log-file und dem Pfad zu einer Datei kann die Datei spezifiziert werden, in welche der Logger schreiben soll. Wenn diese Datei bereits existiert, hängt der Logger seine Nachrichten einfach am Ende der Datei an. Existiert die Datei jedoch nicht, wird eine neue Datei erstellt.

3 Objekte

3.1 Stockwerk

```
1  class Floor
2  {
3  private:
4      std::string name{"Floor"};
5      unsigned int id{};
6      MessageQueue* message_queue;
7      MessageQueue* coordinator_queue;
8      std::shared_ptr<spdlog::logger> file_logger;
9      bool running{true};
10
11 public:
12     Floor(unsigned int id
13     , MessageQueue* coordinator_queue
14     , std::shared_ptr<spdlog::logger> file_logger);
15     void operator()();
16     void push(Message message);
17 };
```

Das Stockwerk Objekt bekommt bei der Initialisierung eine ID, welche gleichzeitig die Nummer des Stockwerkes ist, die Nachrichtenqueue des Koordinators, um Nachrichten an den Koordinator zu schicken und einen shared Pointer für das Logging in eine Datei. Des Weiteren wird auch der Name auf Floor gesetzt. Mit dem Namen und der ID kann das Stockwerk eindeutig identifiziert werden. Des Weiteren hat ein Stockwerk auch einen eigene Nachrichtenqueue, in welche das Repl oder die Simulation Nachrichten stellen, welche von dem Stockwerk bearbeitet und an den Koordinator gesendet werden. Um diese Nachrichten in die Queue einzufügen, hat das Objekt eine öffentliche Methode push, welche eine Nachricht als Parameter benötigt. Die öffentliche Methode operator wird verwendet, um einen Thread für das Stockwerk zu starten. In dieser Methode wartet das Objekt auf eine Nachricht vom REPL oder der Simulation und wenn es eine Nachricht erhält, wird eine Ausgabe getätigt, die Nachricht verändert und an den Koordinator weiter gesendet. Die boolsche Variable running ist dafür da, dass der Thread so lange eine Schleife aufruft, um auf Nachrichten zu warten und sie weiter zu senden, bis eine Nachricht mit dem Befehl stop kommt. Dann wird diese Variable auf false gesetzt und der Thread schließt sich.

3.2 Aufzug

```
1  Class Elevator
2  {
3  private:
4      std::string name{"Elevator"};
5      unsigned int id{1};
6      unsigned int current_floor{1};
7      bool moving{false};
8      float travel_time{3.0};
9      MessageQueue* message_queue;
10     MessageQueue* coordinator_queue;
11     NextFloor_Queue* next_floors;
12     std::shared_ptr<spdlog::logger> file_logger;
13
14 public:
15     Elevator(unsigned int id, float travel_time
16     , MessageQueue* coordinator_queue
17     , std::shared_ptr<spdlog::logger> file_logger);
18     void move_to(unsigned int floor);
19     void first(unsigned int floor);
20     unsigned int get_current_floor();
21     bool is_moving();
22     void operator()();
23     void buttons();
24     void push(Message message);
25 };
```

Ein Aufzug bekommt bei der Initialisierung einen Namen, eine ID, in welchem Stock er sich gerade befindet, die Fahrzeit, die er zwischen zwei Stockwerken benötigt, die Koordinatorqueue, an welche er Nachrichten sendet und einen shared Pointer für das Logging in eine Datei. Durch den Namen und die ID kann jeder Aufzug eindeutig identifiziert werden. Des Weiteren erstellt er bei der Initialisierung eine Nachrichtenqueue, in welche mit der öffentlichen Methode push eine Nachricht eingefügt werden kann. Ebenfalls wird eine Aufzugsqueue erstellt, in diese Queue wird mit den öffentlichen Methoden first und move_to etwas eingefügt, beide Methoden brauchen eine positive ganze Zahl als Parameter. Die Methode first wird nur dann aufgerufen, wenn ein Befehl mit einem Override Flag versehen wurde, dadurch werden alle Elemente, die gerade in der Aufzugsqueue sind, nach hinten verschoben und das übergebene Stockwerk wird an erster Stelle eingefügt. Bei der Methode move_to wird zu erst überprüft, ob das Stockwerk zwischen dem Aktuellen und dem Nächsten liegt, dann wird es als Erstes in die Aufzugsqueue eingefügt, ansonsten wird es an die Queue übergeben, um es an einer passenden Stelle einzufügen. Diese beiden Methoden werden nur von dem Koordinator

aufgerufen. Damit der Koordinator weiß, ob sich ein Aufzug gerade bewegt oder nicht, hat ein Aufzug die Methode is_moving, welche true zurückgibt, wenn er sich bewegt, und wenn er gerade in einem Stockwerk steht, wird flase zurückgegeben. Aus einem Aufzugsobjekt werden zwei Threads erstellt. Der erste Thread ruft die Methode buttons auf. Diese Methode wartet auf Nachrichten vom REPL oder der Simulation, bearbeitet diese und gibt sie an den Koordinator weiter. Dadurch wird das Drücken eines Knopfes in einem Aufzug simuliert. Diese Methode führt das warten, bearbeiten, senden so lange in einer Schleife aus, bis eine Nachricht mit dem Kommando stop erfolgt. Erfolgt diese Nachricht, sendet das Objekt die Nachricht mit dem stop Kommando weiter an den Koordinator und beendet sich danach. Die Methode operator hingegen simuliert die Bewegungen des Aufzuges. Es wird in einer Schleife überprüft, ob man das Ziel erreicht hat, ob es ein neues Ziel gibt oder ob überhaupt noch Ziele in der Queue sind. Hat der Aufzug sein Ziel erreicht, gibt er eine Nachricht aus und löscht dieses Element aus seiner Queue. Wenn sich nach dem Löschvorgang keine Ziele mehr in der Queue befinden, setzt er die Variable moving auf false. Jedes Mal, wenn ein Aufzug sein Ziel erreicht, wartet er auf diesem Stockwerk eine Sekunde, was das Ein- und Aussteigen von Personen simulieren soll. Danach wartet er, bis das nächste Ziel in der Queue ist. Ist die Queue noch nicht leer, holt er sich das nächste Ziel und überprüft, ob es dem Zeichen für stop entspricht. Entspricht es diesem Zeichen, beendet sich der Thread. Entspricht es jedoch nicht dem Zeichen, dann wird die Variable moving auf true gesetzt und der Aufzug fährt weiter. Es wird auch überprüft, ob sich das nächste Ziel in der Zwischenzeit geändert hat und falls es sich geändert hat, in der lokalen Variable die Änderung als neues Ziel gesetzt.

3.3 Koordinator

```
1  class Coordinator
2  {
3  private:
4      std::vector<Elevator>& elevators;
5      MessageQueue* message_queue;
6      std::string name{"Coordinator"};
7      std::shared_ptr<spdlog::logger> file_logger;
8      unsigned int get_closest_elevator(Message message);
9
10 public:
11     Coordinator(std::vector<Elevator>& elevators
12     , MessageQueue* message_queue
13     , std::shared_ptr<spdlog::logger> file_logger);
14     void operator()();
15 };
```

Ein Koordinator bekommt bei der Initialisierung alle Aufzüge, seine Nachrichten-

queue und den shared Pointer für das Logging in eine Datei. Der Koordinator wird mit der Methode operator in einem Thread gestartet und wartet auf Nachrichten von einem Aufzug oder einem Stockwerk in seine Nachrichtenqueue. Wenn er eine Nachricht bekommt, muss er das Kommando auslesen. Ist das Kommando ein Override dann wird die boolsche Variable für override auf true gesetzt und danach aufgrund der Aufzugsid geschaut, ob ein Call Kommando oder ein Move Kommando mit Override aufgerufen wurde. Das kann dadurch erreicht werden, da bei einem Call die Aufzugsid immer 0 ist, es aber keinen Aufzug mit dieser ID gibt. Ist es ein Kommando ohne Override, dann kann die Art des Kommandos einfach ausgelesen werden. Ist es ein Move Kommando wird in die Aufzugqueue des Aufzuges, in welchem der Knopf gedrückt wurde, das Stockwerk eingefügt. Wenn es ein Override Kommando war mit der Methode first des Aufzuges ansonsten mit der Methode move_to. Ist das Kommando jedoch kein Move, sondern ein Call, muss der Koordinator zuerst schauen, welcher Aufzug sich bewegt und welcher am nächsten ist. Das erfolgt durch die private Methode get_closest_elevator. Diese Methode ruft die statischen Methoden closest_elevator_not_mooving und closest_elevator_with_mooving auf. Diese Methoden werden asynchron in einem eigenen Thread gestartet und durch ein future-promis-paar wird das Ergebnis abgefragt. Es wird zuerst überprüft, ob sich ein Aufzug nicht bewegt, wenn das der Fall ist, wird der nächste sich nicht bewegende Aufzug ausgesucht. Bewegen sich jedoch alle Aufzüge, wird der Aufzug ausgesucht, der sich dem Stockwerk am nähesten befindet. Steht der Aufzug jetzt fest, so wird wie bei einem Move das Stockwerk in die Aufzugsqueue des Aufzuges eingefügt. Bei einem Override wieder mit der Methode first und wenn es kein override ist mit der Methode move_to.

3.4 REPL (Read–eval–print loop)

```
1  class Repl
2  {
3  private:
4      std::vector<Floor>& floors;
5      std::vector<Elevator>& elevators;
6      unsigned int floor_number{};
7      unsigned int elevator_number{};
8      bool override{false};
9      bool use_simulation{false};
10     bool& running;
11     std::shared_ptr<spdlog::logger> file_logger;
12     void move(std::string floor_number
13     , std::string elevator_number, bool override);
14     void call(std::string number, bool override);
15     void show_help();
16     void stop();
17
```

```
18 public:
19     Repl(std::vector<Floor>& floors
20     , std::vector<Elevator>& elevators
21     , unsigned int floor_number
22     , unsigned int elevator_number, bool override
23     , std::shared_ptr<spdlog::logger> file_logger
24     , bool use_simulation, bool& running);
25     void operator()();
26 };
```

Das REPL oder Read–eval–print loop ist eine Schleife, die Anwendereingaben list diese evaluiert, danach etwas ausgibt und das ganze in einer Schleife solange macht, bis der Anwender es abbricht oder das end Kommando eingibt. Bei der Initialisierung des REPLs werden alle Stockwerke, alle Aufzüge, die Anzahl der Stockwerke, die Anzahl der Aufzüge, ob override aktiv ist, der shared Pointer für das Logging in die Datei, ob die Simulation aktiv ist und ob das Programm läuft übergeben. Zuerst wird überprüft, ob eine Simulation läuft und wenn eine Simulation läuft, wird nur auf eine Eingabe gewartet, damit die Variable running auf false gesetzt werden kann uns somit die Simulation weiß, dass das Programm beendet wurde. Ist die Simulation jedoch nicht aktiv, wird zuerst die Grammatik für das REPL definiert. Dabei wird überprüft, ob Override aktiv ist. Ist es Aktiv wird es in die Grammatik eingebunden, ist es nicht aktiv, gilt die Eingabe eines Override Kommando als Fehler. Nachdem die Grammatik definiert ist, startet die Schleife. In dieser Schleife wird auf eine Eingabe des Anwenders gewartet. Ist eine Eingabe vorhanden, wird diese mit der Grammatik verglichen. Dadurch werden nicht vorhanden Kommandos ignoriert und es wird eine Information für den Anwender angezeigt, dass seine Eingabe keinem Kommando entspricht und welche Kommandos es gibt. Ist das Kommando jedoch als gültig erklärt worden, dann wird überprüft, um welches Kommando es sich handelt. Handelt es sich um das Kommando help, wird eine Infonachricht mit allen verfügbaren Kommandos ausgegeben. Handelt es sich um einen Override Kommando, wird eine lokale boolsche Variable auf true gesetzt und override wird aus dem string gelöscht, um in weiter zu überprüfen. Danach wird unterschieden zwischen einem call und einem move befehl. Bei einem Call befehl wird danach eine Zahl für die Stockwerksnummer angegeben, diese Muss von einem String zu einer ganzzahligen positiven Zahl konvertiert werden. Ist das erledigt, wird die Methode call aufgerufen. Diese Methode überprüft, ob es überhaupt soviel Stockwerke gibt, wie vom Anwender angegeben. Gibt es nicht so viele Stockwerke, wird eine Warnung ausgegeben mit der Anzahl der existierenden Stockwerke und der vom Anwender eingegebenen Anzahl. Ist die angegebene Zahl im Bereich des Möglichen, wird noch überprüft, ob override verwendet wurde, also dass die Variable true ist und dann die dementsprechende Nachricht an das dementsprechende Stockwerk geschickt. Ist es kein Call Kommando, sonder ein Move Kommando, müssen zwei Zahlen konvertiert werden und es wird die Methode move aufgerufen. Diese Methode überprüft ebenfalls, ob die eingegebene Zahl für das Stockwerk und die eingegebene Zahl für den Aufzug im Bereich des Möglichen sind. Sind sie es

nicht, wird eine Warnung ausgegeben, mit wie viele Aufzüge oder Stockwerke es gibt und was ihre Eingabe war. Ist die Eingabe jedoch korrekt, wird überprüft, ob override verwendet wurde, also dass die Variable true ist und dann die dementsprechende Nachricht an den dementsprechenden Aufzug geschickt. Wird das Kommando end eingegeben, sendet das REPL an alle Aufzüge und an alle Stockwerke die Nachricht stop. Außerdem setzt es die Variable running auf flase und beendet somit die Schleife und den Thread.

3.5 Simulation

```
1  class Simulation
2  {
3  private:
4      std::vector<Floor>& floors;
5      std::vector<Elevator>& elevators;
6      bool override{false};
7      bool& running;
8      std::shared_ptr<spdlog::logger> file_logger;
9      float sim_time{};
10     unsigned int floor_number{};
11     unsigned int elevator_number{};
12     void move(unsigned int floor_number
13     , unsigned int elevator_number, bool override);
14     void call(unsigned int number, bool override);
15
16 public:
17     Simulation(std::vector<Floor>& floors
18     , std::vector<Elevator>& elevators
19     , unsigned int floor_number
20     , unsigned int elevator_number, bool override
21     , std::shared_ptr<spdlog::logger> file_logger
22     , float sim_time, bool& running);
23     void operator()();
24 };
```

Die Simulation benötigt bei Initialisierung alle Stockwerke, alle Aufzüge, die Anzahl der Stockwerke, die Anzahl der Aufzüge, ob es override gibt oder nicht, den shared Pointer für das Logging in eine Datei, die Zeit zwischen den Kommandos und ob die Simulation beendet ist. Die Simulation wird mit der Methode operator als Thread gestartet. Es wird eine Schleife ausgeführt, die zufällig ein Kommando generiert, wenn override aktiv ist auch override Kommandos. Diese Kommandos werden dann mit den privaten Methoden move und call weiter bearbeitet. Wenn ein call Kommando mit einer Stockwerksnummer generiert wurde, wird die Methode call aufgerufen, um dieses Kommando in eine Nachricht zu verpacken und es an das dementsprechende Stockwerk

zu schicken. Bei einem Move Kommando wird die Methode move aufgerufen, um dieses generierte Kommando in eine Nachricht zu verpacken und an den dementsprechenden Aufzug zu verschicken. Wird ein override Kommando, generiert wir einfach eine boolsche Variable auf true gesetzt und beide Methoden setzten dieses Kommando in die Nachricht ein. Die Simulation läuft solange, bis eine Anwendereingabe das Programm beendet. Das wird durch die boolsche Variable running realisiert, welche bei einer Anwendereingabe auf false gesetzt wird.

3.6 Aufzugsqueue

```
1  class NextFloor_Queue
2  {
3  private:
4      std::vector<unsigned int> next_floors{};
5      std::mutex m{};
6      std::condition_variable con_empty{};
7      std::shared_ptr<spdlog::logger> file_logger;
8
9  public:
10     NextFloor_Queue
11         (std::shared_ptr<spdlog::logger> file_logger);
12     unsigned int front();
13     unsigned int get();
14     void erase(unsigned int floor);
15     void insert_first(unsigned int floor);
16     void insert(unsigned int floor);
17     bool empty();
18
19 };
```

Die Aufzugsqueue bekommt bei der Initialisierung den shared Pointer für das Logging in Dateien und es wird ein Vector angelegt, welcher positive ganze Zahlen speichern kann. Um in diesen Vector etwas einfügen zu können, gibt es zwei öffentliche Methoden, insert_first und insert. Beide Methoden benötigen als Parameter eine positive ganze Zahl. Der Unterschied dieser beiden Methoden ist das insert_first den Parameter immer als Erstes einfügt. Wohingegen insert den Parametern nach seinem numerischen Wert einordnet. Um aus diesem Vector auch wieder etwas auslesen zu können, gibt es die Methoden first und get. Bei der Methode first wird der erste Wert des Vectors zurückgegeben und wenn keiner existiert 0. Bei get wird ebenfalls der erste Wert zurückgegeben, existiert aber kein wert, dann wird gewartet, bis einer eingefügt wird. Des Weiteren gibt es noch die Methode erase um einen Wert aus dem Vector zu löschen und die Methode empty die zurückgibt, ob der Vector leer ist oder nicht.

3.7 Nachrichtenqueue

```
1  class MessageQueue
2  {
3  private:
4      std::queue<Message> message_queue{};
5      std::mutex m{};
6      std::condition_variable empty{};
7
8  public:
9      Message pop();
10     void push(Message message);
11 };
```

Die Nachrichtenqueue ist die direkte Kommunikationsverbindung zwischen den Aufzugsthreads, den Stockwerk Threads, dem Koordinator Thread und dem REPL Thread oder dem Simulationsthread. Bei der Initialisierung muss nichts übergeben werden. Des Weiteren enthält die Nachrichtenqueue eine Queue, welche Objekte vom Typ Nachricht speichern kann. Damit auf diese Queue zugegriffen werden kann, gibt es die öffentlichen Methoden pop und push. Bei der Methode pop wird die nächste Nachricht aus der Queue geholt, aus der Queue gelöscht und zurück übergeben. Ist keine Nachricht in der Queue vorhanden, wird gewartet, bis eine eingefügt wird. Mit push fügt man eine Nachricht in die Queue ein und benachrichtigt das wartende pop.

3.8 Nachricht

```
1  class Message
2  {
3  private:
4      std::string message{};
5      std::string command{};
6      unsigned int floor{};
7      unsigned int elevator_id{};
8
9  public:
10     Message(std::string message, std::string command
11     , unsigned int floor, unsigned int elevator_id);
12     std::string get_message();
13     std::string get_command();
14     unsigned int get_floor();
15     unsigned int get_elevator_id();
16     std::string to_string();
17 };
```

Eine Nachricht wird verwendet, um zwischen den Stockwerken, den Aufzügen, dem Koordinator und dem REPL oder der Simulation eine einheitliche Kommunikationsbasis herzustellen. Beim Initialisieren einer Nachricht müssen eine Nachricht, ein Kommando, eine Stockwerknummer und eine Aufzugsid übergeben werden. Für jedes dieser Attribute gibt es eine Funktion, um ihren Wert Auslesen zu können.

4 Programmstart

Bei Programmstart werden zuerst alle Parameter überprüft und die dazu Angegebenen Werte des Benutzers. Ist irgend ein Wert nicht korrekt oder fehlerhaft, wird eine Info an den Benutzer ausgegeben und das Programm geschlossen. Danach wird überprüft, ob eine Konfigurationsdatei verwendet wird. Wird eine solche verwendet, wird diese validiert und die Werte werden übertragen. Ist die Datei jedoch nicht Valide, also sie weißt Fehler auf, wird ebenfalls eine Info an den Benutzer ausgegeben und das Programm beendet. Die nächste Überprüfung ist, ob das Loggen in eine Datei aktiviert wurde und welches Logginglevel verwendet werden soll. Dieser Logger wird erstellt und auf das Logginglevel konfiguriert. Des Weiteren wird das Pattern, mit welchem der Logger schreiben soll, für die Kommandozeilenausgabe genauso wie für das Logging in eine Datei konfiguriert. Der nächste Schritt ist das VeKtoren für die Stockwerke, die Aufzüge und die Threads erstellt werden. Diese Vektoren reservieren sich dann den Speicherplatz, den sie brauchen werden. Als Nächstes wird eine Ausgabe getätigt, welche den Status des Programmes ausgibt, also wie viele Aufzüge es gibt, ob Logging aktiviert ist und so weiter. Die nächste Überprüfung ist, ob der Benutzer die Simulation eingeschalten hat oder ob er das Programm per REPL bedienen will. Je nachdem wird eine Nachricht ausgegeben bei der Simulation, wie man sie beendet, beim REPL wie man die Hilfe aufruft. Als Nächstes werden die Threads für die Stockwerke, die Aufzugsbewegungen, die Aufzugsnachrichten, den Koordinator, das REPL und wenn aktiviert auch für die Simulation erstellt und gestartet. Mit dem Starten der Threads wird der Programmfluss an die Threads abgegeben und es wird nur noch gewartet, bis sich die Threads beenden. Nachdem alle Threads beendet sind, wird ausgegeben, dass das Programm sich beendet hat.