# NVS Projekt2:
# Remote Method Invocation System
# 43

Ralf Kühmayer, 13, 5BHIF

April 2021

# Inhaltsverzeichnis

# 1 Aufgabenstellung

Impelentierung eines Remote Method Invocation Systems basierend auf asio. Der Server und der Client müssen mit spdlog Logdateien anlegen können und JSON soll verwendet werden, entweder für die Kommunikation oder für die Konfiguration.

# 2 Erweiterung durch Bibliotheken

### asio

Die Bibliothek asio wird für die generelle Kommunikation zwischen dem Server und dem Client verwendet.

### CLI11

Mit CLI11 wird die Schnittstelle beim Starten des Programms zum Benutzer implementiert. CLI11 hilft auch beim Validieren der Parameter und überprüfen, ob Dateien zur Konfiguration vorhanden sind.

### doctest

Mit der Bibliothek doctest, werden die Testfälle der Unit- und Integration-Tests implementiert und es gibt die Möglichkeit, dem Anwender mit Parametern zu Konfigurieren, welche Tests ausgeführt werden sollen und wie es sich bei Fehlern verhalten soll.

### fmt

Fmt wird zum Formatieren von Strings und zur synchronisierten und farbigen Ausgabe verwendet.

### grpc

Mit der grpc Bibliothek kann man von einem Client eine Funktion Remote am Server ausführen und von dieser einen Rückgabewert bekommen.

### json

Durch die json Bibliothek kann eine JSON-Datei an das Programm übergeben werden und das Programm basierend auf dieser konfiguriert.

**magic_enum**

Mit der Bibliothek magig_enum wird das Arbeiten mit enums in c++ erleichtert und die enums im Generellen um Funktionalität erweitert.

**peglib**

Mit peglib können Benutzer eingaben, validiert und geparst werden. Man kann eine Grammatik angeben und für jeden Befehl eine eigene Funktion starten.

**protobuf**

Mittels protobuf können Message-Objekte erstellt werden, welche komplexe Objekte beinhalten können. Diese Message-Objekte kann man dann mit protobuf zu einem String serialisieren, was für die Übertragung von Objekten praktisch ist.

**rang**

Die Bibliothek rang, wird verwendet, um den Konsolenoutputstream von c++ zu Manipulieren und formatierte und farbige Ausgabe möglich zu machen. Da die Standard c++ Streams verwendet werden, ist die Ausgabe nicht Threadsafe.

**spdlog**

Mit spdlog können Log-Dateien erstellt und beschrieben werden. Spdlog ist Threadsafe und fügt viele nützliche Features zum Loggen hinzu, wie z. B. Uhrzeit und Threadnummer.

**toml++**

Durch die toml++ Bibliothek kann eine TOML-Datei an das Programm übergeben werden und das Programm basierend auf dieser konfiguriert.

# 3 Remote Method Invocation

## 3.1 Theorie

### 3.1.1 RPC (Remote Procedure Call)

#### Hintergrund

Ein Remote Procedure Call (RPC)[1] ist eine Methodik zum Ausführen von Prozeduren auf einem entfernten Rechner. Es wird meistens zur Realisierung von Client/Server-Architekturen verwendet und beruht auf dem UDP- oder TCP-Protokoll. Die ersten RPCS, wurden von Sun Microsystems für ein Network File System entwickelt. Schon im Jahre 1976 wurde die ersten Ansätze der Technik des remoten Aufrufens von Prozeduren von James E. White in de RFC 707 niedergeschrieben. Diese Technik wurde in der Folge von weiteren Netzwerk Systemen wie Xerox Network Systems (XNS) oder Novells Netware eingesetzt. Im Jahre 1994 bekamen dann sogar die Herren Andrew Birrell und Bruce Nelson den ACM Software System Award für ihre Entwicklung. Durch den Gewinn dieses Preises, wurde sowohl die RPC-Variante Sun-RPC als auch die RPC-Variante ONC-RPC (Open Network Computing Remote Procedure Call) bekannt. Neben diesen beiden schaffte es auch die Variante Distributed Computing Environment Remote Procedure Call (DCE-RPC) bekannt zu werden und diese galt zum damaligen Zeitpunkt als verbreitetste Variante. Von dieser Variante leitet Microsoft ihre eigene Variante ab, den Microsoft-RPC (MSRPC). Die Grundidee hinter der Technik des RPC, galt der Speicherplatz Erweiterung. Man wollte über ein Netzwerk Speicherplätze zur Verfügung stellen, dieser sollte so zuverlässig wie der stationäre Speicherplatz sein. Für dieses Vorhaben wurde mithilfe von RPC eine Reihe von Befehlen erstellt, welche über ein Client-Server-Modell die Speicher-Routinen abarbeiten sollten.

#### Wie funktioniert RPC?

Die Kommunikation bei einem RPC verläuft nach dem folgenden Prinzip: Der Client sendet dem Server einen RPC. Dieser Server empfängt die RPC-Anfrage mittels eines installierten Portmappers (Endpointmapper). Die in dieser Anfrage enthaltenen Daten werden dann weiter an die jeweilige Server-Anwendung weiter gegebnen. Diese Server-Anwendung verarbeitet dann den Auftrag und sendet dem Client das Ergebnis, eine Fehlermeldung oder einen Status zurück.

[1] Geißler, *Was ist ein Remote Procedure Call - RPC?*

#### Probleme

Jedoch können bei einem RPC Aufruf verschiedene Fehler in der Kommunikation entstehen. Diese Fehler müssen von Programmieren in der Entwicklung einer solchen Anwendung auf jeden Fall beachtet werden. Es könnt zum Beispiel passieren, dass der Server keine Antwort auf den RPC Aufruf eines Clients gibt. Doch nicht nur auf dem Client können Probleme auftreten, der Server sollte sich merken, welcher Client welche Funktion schon ausgeführt hat, um einen mehrfachen Aufruf dieser Funktion zu vermeiden. Jedoch gibt es bei der RPC Technik verschiedene Möglichkeiten, solche Probleme zu verhinder und dadurch die Datenintegrität zu gewährleisten. Jedoch muss man dafür erst mal zwischen der Technik des synchronem RPC und des asynchronen RPC unterscheiden. Ist die Kommunikation für synchrones RPC ausgelegt, muss der Server die Ausführung schnell durchführen, da ansonsten der Client blockiert wird, da er auf die Antwort vom Server wartet. Dieses Problem tritt bei der Technik des asynchronen RPCs nicht auf, da der Client, während er auf die Antwort wartet, weitere Operationen ausführen kann. Jedoch ist es schwieriger, asynchrone RPCs zu implementieren, da, wenn ein Fehler auftritt, dass Programm aber zwischenzeitlich schon weiter gearbeitet hat, dieser Fehler nicht behoben werden kann und dadurch das Programm beendet werden muss.

### 3.1.2 RMI (Remote Method Invocation)

#### Hintergrund

Remote Method Invocation [2] bezeichnet das entfernte Aufrufen von Methoden und Zugriffsvariablen und ist eine spezifische Methode der Programmiersprache Java. Die Anfänge des RMI liegen in der Realisierung eines verteilten Objektmodells in Java, welches die Interaktion von Remote-Objekten erlauben sollte. Diese Interaktion mit den Remote-Objekten sollte möglichst nahe an der Interaktion mit lokalen Java-Objekten liegen. In diesem Kontext werden Remote-Objekte verstanden, welche sich entweder auf entfernten externen Rechner-Systemen befinden oder aber getrennt auf einer anderen Java Virtual Maschine (JVM) laufen. Für die Kommunikation gibt es verschiedene für eine bestimmten Zweck zugeschnittene Protokolle. Unter diesen Protokollen gitb es zum Beispiel das Java Remote Method Protocol (JRMP), welches auch unter dem Namen Wire-Protocol bekannt ist. Dieses Protokoll ist für die RMI das implementierte Standardprotokoll. Aber es gibt auch andere, für einen bestimmten Zweck angepasste Protokolle. So gibt es zum Beispiel das RMI IIOP Prtotkoll, welches zur Integration in der Common Object Request

[2] Buchverlag, *RMI (remote method invocation)*

Broker Architecture (CORBA), welche weitere Mechanismen zur verteilten Anwendungsprogrammierung bereitstellt. Die Kommunikation der RMI kann über HTTP getunnelt werden und somit steht eine Möglichkeit bereit, eine Firewall zu umgehen. RMI ist durch die Verschlüsselung mit Secure Socket Layer (SSL) eine sichere Möglichkeit der Kommunikation.

#### Verwendung von RMI

Durch RMI wird das transparente Verschicken von Objekten und deren Funktionen sowie das dynamische Nachladen von Objektklassen unterstützt. Die Java-Standardumgebung beinhaltet alle notwendigen Methoden und Mechanismen der RMI-Klassenbibliothek. Dadurch ist die Verwendung von entfernten Objekten auf gleiche Weise wie lokale Objekte möglich und auch der Aufruf von Methoden der Objekte ist remote sehr ähnlich zu dem lokalen Aufruf. Durch einen höher entwickelten Mechanismus werden die Methoden der Remote-Objekte aufgerufen. Diesen Methoden kann man Objekte vollständig als Wert oder als Referenz übergeben. Kommunikationsfehler oder andere Fehler können durchaus vorkommen, weshalb es vordefinierte Error-Messages RMI-Klassenbibliothek gibt.

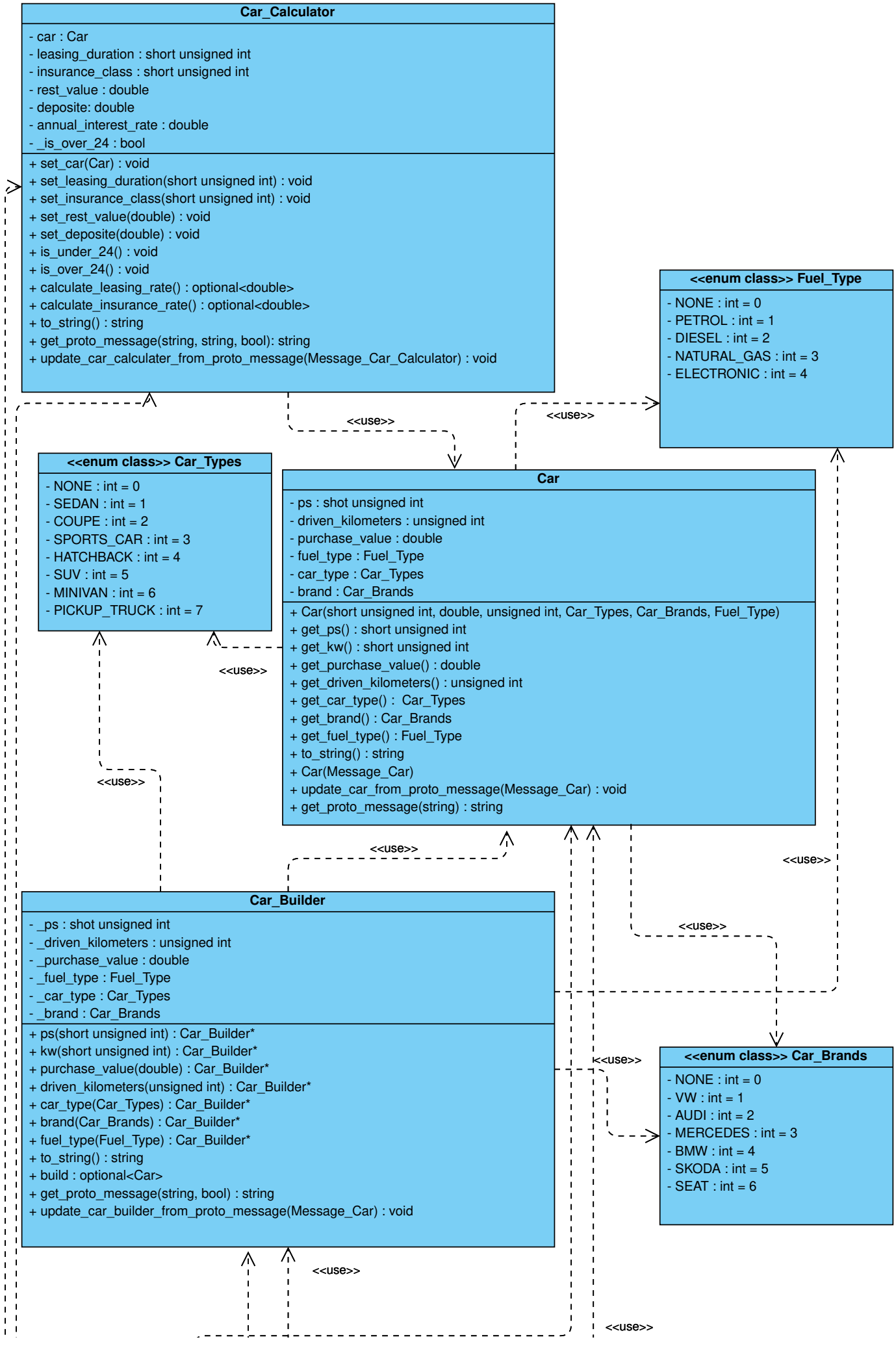
#### RMI-Ebenen

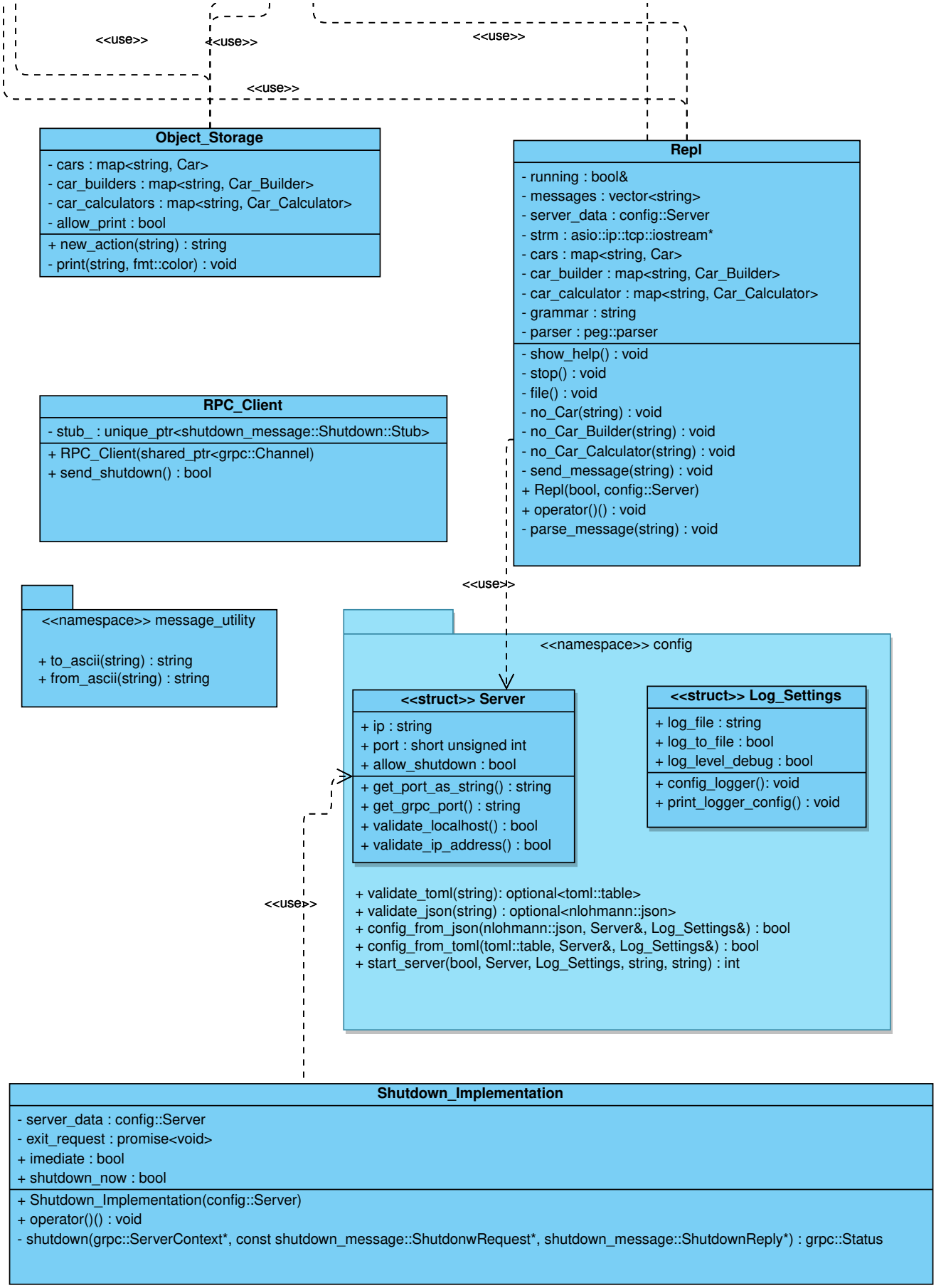
RMI integriert drei verschiedene Ebnen, welche abhängig von den spezifischen Anwendungen, ob es auf einem Externen oder demselben System in einer anderen JVM liegt, von jedem einzelnen Methodenaufruf durchlaufen. Diese drei Ebenen sind: die Transportebene, die Remote Reference Ebene und die 'Stubs'. Die Transportebene ist für die Behandlung der Netzwerkverbindung zwischen den verschiedenen Systemen zuständig. Während die Remote Reference Ebene für die Zusammenstellung des Methodenaufrufs für die Aufbereitung der Parameter und die Aufbereitung der späteren Rückgabewerte zuständig ist. Die Ebene der 'Stubs' ist die Ebene, auf welcher der Server und der Client agieren und auf beiden Seiten quasi Stellvertreter-Objekte, die die Daten von der eigentlichen Implementierungsklasse übernehmen, bereitstellen. Dadurch wird das Stub-Objekt auch der lokale Stellvertreter für das Remote-Objekt genannt.

## 3.2 Umsetzung

In diesem Programm wurden drei Objekte angelegt. Welche dem Server und dem Client bekannt sein müssen. Diese Objekte werden zwischen Server und Client synchronisiert und der Client kann Funktionen der Objekte am Server

aufrufen. Zu diesen Objekten gehört der Car_Builder, welcher als Grundobjekt dient. Mit diesem Car_Builder, kann man ein Objekt, ein Car, erzeugen, welches im weiteren Verlauf an den Car_Calculator übergeben wird. Das Car an sich kann nur angesehen werden und es gibt keine Möglichkeit, es zu verändern. Der Car_Calculator benötigt dann noch ein paar zusätzliche Infos und kann, basierend auf diesen Informationen dann zwei Berechnungen ausführen. Diese Berechnungen wird der Client am Server durchführen. Um ein paar Attribute des Cars besser darstellen zu können, gibt es drei Enums welche die jeweiligen Typen besser benennen. Die drei Grundobjekte werden dann am Server in dem Object_Storage verwaltet. Dieser führt die vom Client gesendeten Operationen aus und überprüft, ob alle Bedingungen gegeben sind, gegebenenfalls gibt er das erwartete Ergebnis oder eine Fehlermeldung zurück. Damit die Kommunikation über Streams in assio mit Protobuff funktioniert, gibt es den Namespace message_utility, welcher dafür sorgt, dass die zu String serialisierten Protobuff Objekte über einen Stream übertragen werden können. Dies ist notwendig, da in den serialisierten Protobuff Objekte \n vorkommt, welches zu auslesen des Streams am Server führen würde. Daher werden die serialisierten Protobuff Objekt mit der Methode to_ascii codiert und dann beim Empfänger wieder mit der Methode from_ascii decodiert.Damit der Benutzer über den Client mit dem Server kommunizieren kann und Objekte erstellen und Methoden ausführen kann, gibt es das Repl, welches die Eingaben des Benutzers liest, diese validiert und die dem entsprechend Methoden, Änderungen, Ausgaben oder Kommunikationen mit dem Server vornimmt. Um generelle Konfigurationen auf dem Server und auf dem Client vor nehmen zu können, gibt es den Namespace config. In diesem Namespace liegen zwei Structs und mehrere Konfigurationsfunktionen. Eines der beiden Structs ist das Server Struct. In diesem Struct werden die Informationen bezüglich der IP-Adresse, des Portes und ob der Server vom Client geschlossen werden darf, gespeichert. Die Information über die IP-Adresse und des Portes ist für den Client wichtig, damit er eine Verbindung aufbauen kann. Die Information über den Port und ob der Server vom Client geschlossen werden darf, hingegen ist für den Server wichtig. Das zweite Struct, Log_Settings, legt sowohl am Client als auch am Server die Konfiguration des Loggers fest und gibt diese in einem lesbaren Format aus. Die zusätzlichen Funktionen in dem config Namespace dienen der allgemeinen Konfiguration. Sie erlaben es sowohl den Client als auch den Server über eine JSON- oder eine TOML-Datei zu konfigurieren und validieren diese beiden Dateiarten auch. Des Weiteren gibt es noch die Funktion, den Server zu starten, welche benötigt wird, wenn der Client angibt, dass er den Server mit starten will.


Car_Calculator
- car : Car
- leasing_duration : short unsigned int
- insurance_class : short unsigned int
- rest_value : double
- deposite: double
- annual_interest_rate : double
- _is_over_24 : bool
+ set_car(Car) : void
+ set_leasing_duration(short unsigned int) : void
+ set_insurance_class(short unsigned int) : void
+ set_rest_value(double) : void
+ set_deposite(double) : void
+ is_under_24() : void
+ is_over_24() : void
+ calculate_leasing_rate() : optional<double>
+ calculate_insurance_rate() : optional<double>
+ to_string() : string
+ get_proto_message(string, string, bool): string
+ update_car_calculater_from_proto_message(Message_Car_Calculator) : void
<<enum class>> Fuel_Type
- NONE : int = 0
- PETROL : int = 1
- DIESEL : int = 2
- NATURAL_GAS : int = 3
- ELECTRONIC : int = 4
<<enum class>> Car_Types
- NONE : int = 0
- SEDAN : int = 1
- COUPE : int = 2
- SPORTS_CAR : int = 3
- HATCHBACK : int = 4
- SUV : int = 5
- MINIVAN : int = 6
- PICKUP_TRUCK : int = 7
Car
- ps : shot unsigned int
- driven_kilometers : unsigned int
- purchase_value : double
- fuel_type : Fuel_Type
- car_type : Car_Types
- brand : Car_Brands
+ Car(short unsigned int, double, unsigned int, Car_Types, Car_Brands, Fuel_Type)
+ get_ps() : short unsigned int
+ get_kw() : short unsigned int
+ get_purchase_value() : double
+ get_driven_kilometers() : unsigned int
+ get_car_type() : Car_Types
+ get_brand() : Car_Brands
+ get_fuel_type() : Fuel_Type
+ to_string() : string
+ Car(Message_Car)
+ update_car_from_proto_message(Message_Car) : void
+ get_proto_message(string) : string
Car_Builder
- _ps : shot unsigned int
- _driven_kilometers : unsigned int
- _purchase_value : double
- _fuel_type : Fuel_Type
- _car_type : Car_Types
- _brand : Car_Brands
+ ps(short unsigned int) : Car_Builder*
+ kw(short unsigned int) : Car_Builder*
+ purchase_value(double) : Car_Builder*
+ driven_kilometers(unsigned int) : Car_Builder*
+ car_type(Car_Types) : Car_Builder*
+ brand(Car_Brands) : Car_Builder*
+ fuel_type(Fuel_Type) : Car_Builder*
+ to_string() : string
+ build : optional<Car>
+ get_proto_message(string, bool) : string
+ update_car_builder_from_proto_message(Message_Car) : void
<<enum class>> Car_Brands
- NONE : int = 0
- VW : int = 1
- AUDI : int = 2
- MERCEDES : int = 3
- BMW : int = 4
- SKODA : int = 5
- SEAT : int = 6
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>
<<use>>


<<use>>
<<use>>
<<use>>
<<use>>
Object_Storage
- cars : map<string, Car>
- car_builders : map<string, Car_Builder>
- car_calculators : map<string, Car_Calculator>
- allow_print : bool
+ new_action(string) : string
- print(string, fmt::color) : void
Repl
- running : bool&
- messages : vector<string>
- server_data : config::Server
- strm : asio::ip::tcp::iostream*
- cars : map<string, Car>
- car_builder : map<string, Car_Builder>
- car_calculator : map<string, Car_Calculator>
- grammar : string
- parser : peg::parser
- show_help() : void
- stop() : void
- file() : void
- no_Car(string) : void
- no_Car_Builder(string) : void
- no_Car_Calculator(string) : void
- send_message(string) : void
+ Repl(bool, config::Server)
+ operator()() : void
- parse_message(string) : void
RPC_Client
- stub_ : unique_ptr<shutdown_message::Shutdown::Stub>
+ RPC_Client(shared_ptr<grpc::Channel)
+ send_shutdown() : bool
<<use>>
<<namespace>> message_utility
+ to_ascii(string) : string
+ from_ascii(string) : string
<<namespace>> config
<<struct>> Server
+ ip : string
+ port : short unsigned int
+ allow_shutdown : bool
+ get_port_as_string() : string
+ get_grpc_port() : string
+ validate_localhost() : bool
+ validate_ip_address() : bool
<<struct>> Log_Settings
+ log_file : string
+ log_to_file : bool
+ log_level_debug : bool
+ config_logger(): void
+ print_logger_config() : void
+ validate_toml(string): optional<toml::table>
+ validate_json(string) : optional<nlohmann::json>
+ config_from_json(nlohmann::json, Server&, Log_Settings&) : bool
+ config_from_toml(toml::table, Server&, Log_Settings&) : bool
+ start_server(bool, Server, Log_Settings, string, string) : int
<<use>>
Shutdown_Implementation
- server_data : config::Server
- exit_request : promise<void>
+ imediate : bool
+ shutdown_now : bool
+ Shutdown_Implementation(config::Server)
+ operator()() : void
- shutdown(grpc::ServerContext*, const shutdown_message::ShutdonwRequest*, shutdown_message::ShutdownReply*) : grpc::Status

## 3.3 Bedienung

### 3.3.1 Message_utilitys

Damit der Server und der Client kommunizieren können und dabei auch Objekte versendet werden können, gibt es für alle Objekte, welche auch übertragen werden, eine Protobuff-Message, welche an den Request oder den Reply übergeben werden kann. Um diese Messages mittels asio übertragen zu können, werden sie zu Strings serialisiert. Da diese serialisierten Strings allerdings auch \n und andere ASCII-Steuerzeichen beinhalten können, werde diese Strings codiert, um dann ein \n anzuhängen, um sie zu versenden. Damit die codierte Message möglichst kompakt, aber die Erstellung auch möglichst schnell geht, werden in diesem serialisierten String jedes Zeichen in seinen entsprechenden ASCII-Code umgewandelt. Die Kompaktheit und die Geschwindigkeit sind notwendig, damit beim Senden und Empfangen eine kürzest mögliche Decodierung, also eine kürzest mögliche Verzögerung stattfindet.

```
string message_utility::to_ascii(string data) {
    stringstream sstream;

    for(char ch : data) {
        sstream << " " << (int)ch;
    }
    sstream << "\n";

    return sstream.str();
}


string message_utility::from_ascii(string data) {

    data = pystring::lstrip(data);
    stringstream sstream{};
    string new_string{};
    string temp{};

    sstream << data;

    while (sstream >> temp) {
```

```
        new_string.push_back((char) stoi(temp));
    }

    return new_string;
}
```

### 3.3.2 Server / Object-Storage

Der Server bekommt die codierten Strings, welche decodiert werden und an den Object-Storage weiter gegeben werden. In diesem Object-Storage wird dann versucht, diesen String, welcher ein serialisierter String des Proto-Requests sein sollte, zu einem Proto-Request Objekt zu parsen. Kann der String nicht geparst werden, gibt der Object-Storage eine Fehlermeldung zurück, welche der Server codiert und an den Client sendet. Kann der String jedoch in das Proto-Request Objekt geparst werden, dann wird dieses im Object-Storage verarbeitet. Das heißt, dass zuerst überprüft wird, um welche Art von Aufgabe sich es handelt. Diese Überprüfung wird durch ein switch-case mit dem MessagTypen, welcher den Wert einer Enum beinhaltet, durchgeführt. Kann dieser Typ dann einem Case zugeordnet werden, dann werden die für diese Aktion notwendigen Überprüfungen und Aktionen durchgeführt. Wie in dem Beispiel zu sehen, werden bei fast allen Cases zuerst überprüft, ob überhaupt alle notwendigen Objekte vorhanden sind. In manchen Fällen müssen diese Objekte vorhanden sein, sonst wird eine Error-Message an den Client zurückgegeben. In anderen Fällen wie dem Beispiel wird überprüft, ob das Objekt vorhanden ist, ist es vorhanden, dann wird es aktualisiert, ist es nicht vorhanden, dann wird es erstellt. Sind alle Überprüfungen erfolgreich und die Operation konnte erfolgreich durchgeführt werden, dann wird das Erstellte Protp-Reply Objekt an den Client zurückgegeben.

```
...
case Request::MessageType::Request_MessageType_BUILDER:
    spdlog::debug(fmt::format("Message is MessageType Builder"));

    if (this->car_builders.find(msg.name()) != this->car_builders.end()) {
        spdlog::info(fmt::format("Builder {} existsts and will be updated",
                    msg.name()));

        print(fmt::format("Builder {} existsts and will be updated",
                    msg.name()), fmt::color::medium_sea_green);
```

```
        this->car_builders.at(msg.name())
            .update_car_builder_from_proto_message(msg.car());
    } else {
        spdlog::info(
            fmt::format("Builder {} does not existst and will be created",
                msg.name()
            )
        );

        print(
            fmt::format("Builder {} does not existst and will be created",
                msg.name()
            ), fmt::color::medium_sea_green);

        Car_Builder builder{};
        builder.update_car_builder_from_proto_message(msg.car());
        this->car_builders.insert_or_assign(msg.name(), builder);
    }

    return this->car_builders.at(msg.name())
            .get_proto_message(msg.name(), true);
...
```

### 3.3.3 Client file input

Damit es dem Benutzer leichter fällt, bestimmte aufgaben häufiger auszuführen, gibt es die Möglichkeit, ein Script an Befehlen zu erstellen und dieses an den Client mittels des File-Befehls zu übergeben. Diese Script-Datei wird dann vom Client gelesen und wenn die Datei vorhanden ist, wird jeder Befehl in dieser Datei mit der Grammatik validiert und interpretiert. Ist die Datei nicht vorhanden, wird dem Benutzer eine Fehlermeldung ausgegeben.

```
void Repl::file() {
    string input{};
    fmt::print("Filepath: ");
    getline(cin, input);
    ifstream infile(input.c_str());

    if (infile.good()) {
```

```
        string line{};
        while (getline(infile, line) && this->running) {
            fmt::print("{}\n", line);

            line = pystring::lower(line);

            line = pystring::strip(line);

            spdlog::debug(fmt::format("File input: {}", line));

            this->parser.parse(line.c_str());

            this_thread::sleep_for(chrono::seconds(1));
        }
    } else {
        fmt::print("File {} does not exist\n", input);

        spdlog::info(fmt::format("File {} does not exist", input));
    }
}
```

### 3.3.4 Client send Message

Wenn der Client eine Nachricht an den Server senden soll, überprüft er vor dem senden, ob der Server die aktuelle Verbindung noch aufrecht erhalten hat oder ob die Verbindung abgebrochen ist. Falls die Verbindung nicht abgebrochen ist, codiert der Client zu erst die Nachricht. Ist die Nachricht codiert, wird sie in einem Vektor gespeichert. Dieser Vektor existiert damit, falls die Verbindung abgebrochen ist und der Benutzer die Verbindung wieder Aufbauen will und der Server noch aktiv ist, der Client den Server auf den Aktuellen stand bringen kann. Nachdem die Nachricht gespeichert ist, wird diese auch an den Server gesendet. Danach wartet der Client auf eine Antwort vom Server. Sobald er diese Antwort bekommt, decodiert er sie und interpretiert diese. Wenn die Verbindung abgebrochen ist und der Benutzer probiert, den Client wieder mit dem Server zu verbinden, aber der Server nicht mehr aktiv ist, dann schließt sich der Client mit der Fehlermeldung, dass der Server nicht mehr aktiv ist.

```
void Repl::send_message(string msg) {
```

```
    if (*strm) {

        spdlog::debug(fmt::format("Client encodes message '{}'", msg));

        string dec_msg{message_utility::to_ascii(msg)};

        spdlog::debug(fmt::format("Client sends encoded message '{}'",
        pystring::replace(dec_msg, "\n", "")));

        this->messages.push_back(dec_msg);

        *strm << dec_msg;

        string data;

        getline(*strm, data);

        spdlog::debug(
            fmt::format("Client got   encoded message '{}'", data)
        );

        data = message_utility::from_ascii(data);

        spdlog::debug(fmt::format("Client decoded message '{}'", data));

        data = pystring::lstrip(data);

        if (data != "ok" && data != "") {
            parse_message(data);
        }

    } else {
        ...
    }
}
```

### 3.3.5 Proto-Message

Um die Kommunikation und Synchronisation von Objekten zwischen dem Client und dem Server zu implementieren, werden Proto-Messages verwendet. Diese Proto-Messages/-Objekte können einfach erstellt werden und ein-

fach zu Strings serialisiert werden. Da der Request an den Server und der Reply von dem Server leicht unterschiedlich sind, gibt es für beide Richtungen eine eigene Message. In beiden Messages gibt es eine Enum, welche der empfangenden Partei mitteilt, welche Operation für diese Message ausgeführt werden soll. Für die unterschiedlichen Aktionen sind dann weitere Daten in diesen Objekten abgebildet. Diese Daten werden dann ausgelesen und verwendet, um z. B. ein Objekt zu aktualisieren oder ein neues zu erstellen. Die Reply message, hat auch die Möglichkeit, einen Error an den Client zu übergeben, wenn eine Operation versucht wird, für welche nicht genügend Daten vorhanden sind.

```
message Request {

    enum MessageType {
        NONE = 0;
        BUILDER = 1;
        CALCULATOR = 2;
        BUILD = 3;
        CALC_LEASING = 4;
        CALC_INSURANCE = 5;
    }

    MessageType type = 1;

    oneof message {
        Message_Car car = 2;
        Message_Car_Calculator calculator = 3;
    }

    string name = 4;
    string builder = 5;
}

message Reply {

    enum MessageType {
        NONE = 0;
        BUILDER = 1;
```

```
        CALCULATOR = 2;
        CAR = 3;
        DOUBLE = 4;
        ERROR = 5;
    }

    MessageType type = 1;

    oneof message {
        Message_Car car = 2;
        Message_Car_Calculator calculator = 3;
    }

    string text = 4;
    double value = 5;
}
```

# 4 Programm

## 4.1 Parameter

Bei beiden Programmen kann man die Parameter in Logging-Parameter, Connection-Parameter und Konfiguration-Parameter einteilen. Wobei diese Parameter bis auf einige Spezifische auf dem Server und dem Client gleich sind.

### Logging-Parameter

#### -l,--log-to-file

Mit diesem Parameter kann der Benutzer das Logging in eine externe Datei aktivieren. Wird nur dieser Parameter angegeben, ist das Log-Level per Default auf Info gestellt und der Logger erstellt eine Datei in einem Unterverzeichnis log mit dem Namen server.log oder client.log.

#### -d,--log-level-debug

Will man mehr Daten sehen, um z. B. einen Fehler im Programm zu finden. Kann man mit diesem Parameter das Log-Level auf Debug setzen. Dadurch werden mehrere Informationen in die Log-Datei geschrieben. Dieser Parameter kann nur angegeben werden, wenn auch der Parameter l,--log-to-fileängegeben wurde.

#### --log-file $< Dateipfad >$

Soll die Log-Datei in einem bestimmten Verzeichnis mit einem bestimmten Namen platziert werden, muss dieser Parameter und ein Dateipfad mit Dateinamen angegeben werden. Existiert diese Datei bereits, wird sie überschrieben. Wenn diese Datei größer als 10 MB wird, legt der Logger bis zu drei Dateien an. Sind alle drei Daten größer als 10 MB, wird begonnen, die erste zu überschreiben. Dieser Parameter kann nur angegeben werden, wenn auch der Parameter l,--log-to-fileängegeben wurde.

### Connection-Parameter

#### -p,--port $< positive\ ganze\ Zahl >$

Mit diesem Parameter wird für den Server der Port angegeben, auf welchem er auf Anfragen hören soll. Für den Client gibt dieser Parameter an, auf welchem Port er die Anfragen schicken soll. Es wird nicht nur der angegebene

Port, sondern auch der nächsthöhere Port verwendet. Wird dieser Parameter nicht angegeben, wird der Defaultport verwendet. Dieser Defaultport ist Port 1112.

**Client: -s,--server-ip** $< Hostname\,oder\,IP-Adresse >$

Dieser Parameter ist nur für den Client bestimmt, da der Server die IP-Adresse des Gerätes, auf welchem er gestartet wurde, annimmt. Wird an diesem Parameter eine IP-Adresse übergeben, dann wird sie validiert, ob sie allen Richtlinien einer IP-Adresse entspricht. Wird dieser Parameter nicht angegeben, wird der Client mit Localhost als IP-Adresse gestartet.

### Konfiguration-Parameter

**-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, werden alle Parameter überschrieben, welche in der Konfigurationsdatei konfiguriert 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, werden alle Parameter überschrieben, welche in der Konfigurationsdatei konfiguriert werden.

### Spezifische Parameter

**Client: --start-server**

Mit diesem Parameter kann man angeben, dass der gestartete Client auch gleich einen Server startet. Dieser gestartete Server wird auch geschlossen, wenn der Client geschlossen wird. Wird dieser Parameter angegeben, erweitert sich die Validierung der IP-Adresse darauf, dass nur Localhost IP-Adressen valide sind. Das ist daher zu erklären, da, wenn man den Server auf seinem eigenen PC startet, dieser immer nur mit Localhost als IP-Adresse angesprochen werden kann.

**Server: -e,--enable-shutdown**

Mit diesem Parameter kann man angeben, dass der gestartete Server sich durch einen grpc Aufruf des Clients schließen lassen kann. Dies wird vor allem benötigt, da der Client den Server mit starten kann und dann diesen auch schließen können muss.

**Server: -a, --allow-print**

Erlaubt dem Server die zu erledigenden Aufgaben auf der Konsole auszugeben.

## 4.2 Server

Beim Starten des Servers werden zuerst alle Parameter auf ihre Richtigkeit überprüft und die Konfiguration basierend auf diesen Einstellungen vorgenommen. Wurde alles richtig konfiguriert, wird der Endpunkt des Servers erstellt. An diesen Endpunkt wird der eingetragenen Port übergeben. Ist dieser Port schon belegt, wirft der Endpunkt einen Error und das Programm beendet sich. Ist der Port jedoch nicht belegt, wird auf diesem Port auf anfragen gewartet. Dieses warten auf Anfragen passiert in einem eigenen Thread, sodass auch ein grpc-Server gestartet werden kann. Dieser Server überprüft auch zuerst, ob sein angegebener Port noch frei ist. Ist bei keinem der beiden Endpunkte ein Fehler aufgetreten, wird eine Nachricht ausgegeben, dass der Server gestartet ist. Jetzt wartet der Server auf Anfragen. Erhält der Server eine Anfrage, erstellt er für diese einen eigenen Thread und erstellt einen eigenen Object-Storage. Dieser überprüft jede Anfrage und sendet je nach dem eine Fehlermeldung oder ein Objekt zurück. Schließt der Client die Verbindung, wird der Object-Storage gelöscht und der Thread beendet sich. Der Server läuft solange weiter, bis er von einem Client oder durch eine Benutzereingabe das Zeichen bekommt, sich zu beenden. Erhält er dieses Zeichen, schließt er alle Verbindungen und beendet das Programm.

## 4.3 Client

Beim Starten des Clients werden zuerst alle Parameter auf ihre Richtigkeit überprüft und die Konfiguration basierend auf diesen Einstellungen vorgenommen. Wurde alles richtig konfiguriert, startet der Client das Repl. Beim Start des Repls wird überprüft, ob sich der Client zu dem Server verbinden kann. Kann keine Verbindung aufgebaut werden, schließt sich der Client. Kann eine Verbindung aufgebaut werden, werden die Benutzereingaben

überprüft und für jeden validen Befehl eine Funktion ausgeführt. Manche dieser Funktionen haben zur Folge, dass Objekte zwischen dem Server und dem Client synchronisiert werden müssen. Wird bei so einem Befehl festgestellt, dass der Server nicht mehr erreichbar ist, wird dem Benutzer die Option gegeben, das Programm zu beenden oder es erneut zu versuchen. Ist der Server jedoch erreichbar, werden die Objekte synchronisiert. Es gibt auch Befehle, welche es erfordern, dass der Server eine Methode ausführt und dem Client das Ergebnis zurückgibt. Will der Benutzer das Programm beenden, muss er den Endbefehl eingeben. Die Verbindung zum Server wird geschlossen und falls der Benutzer den Server mit dem Client mit gestartet hat, wird auch der Server geschlossen.

## 4.4 Unit-Tests

Die Unit Tests basieren auf der Bibliothek "doctest.h". Diese Bibliothek führt die geschriebenen Testfälle aus. Durch diese Bibliothek kann man dem Programm auch durch Parameter mitteilen, dass man nur einen bestimmten Test ausführen möchte oder in welcher Reihenfolge man sie ausführen möchte. Diese Tests testen die Funktionalität der Funktionen, welche keine Verbindung zum Server brauchen. Dadurch wird sicher gestellt, dass einfache Funktionen wie das Parsen eines Objektes zu einem Proto-Objekt, ohne Probleme Funktionieren.

## 4.5 Integration-Tests

Die Integration-Tests basieren ebenfalls auf der "doctest.h"Bibliothek, wodurch sie dieselbe Konfigurierbarkeit wie die Unit-Tests haben. Der einzige Unterschied zwischen diesen zwei Testmöglichkeiten ist, dass die Integration-Tests im Gegensatz zu den Unit-Tests einen eigenen Server starten, um die Funktion des Sendens und Empfangenes zu testen. Damit der Server gestartet werden kann, wird ein weiterer Prozess erstellt, welcher dann den Server startet. Dieser Server kann von einem Client herunter gefahren werden, denn mit dieser Technik wird der Server nach dem Erfolgen der Tests wieder geschlossen. Damit sicher gestellt wird, dass der Server auch läuft, wird vor den Tests die Verbindung überprüft. Diese Überprüfung bricht das Programm ab, wenn der Server nicht erreichbar ist. Kann der Server nicht gestartet werden, da schon ein Programm auf diesem Port lauscht und es das in diesem Projekt entwickelte Server-Programm ist, laufen die Integration-Tests trotzdem ab. Es kann nur zu Komplikationen führen, wenn der Server so konfiguriert ist, dass der Client ihn schließen kann, da die Integration-Tests den Server am Ende schließen.

# Literatur

[Buc] DATACOM Buchverlag. *RMI (remote method invocation)*. URL: https://www.itwissen.info/RMI-remote-method-invocation.html (besucht am 11. 04. 2021).

[Gei] Dipl. Betriebswirt Otto Geißler. *Was ist ein Remote Procedure Call - RPC?* URL: https://www.datacenter-insider.de/was-ist-ein-remote-procedure-call--rpc-a-712873/ (besucht am 11. 04. 2021).