
Märklin Control

Projektvorstellung

Abschlusspräsentation

Vertiefung Informatik, 17.01.2017

Gliederung

1. Zielsetzung
2. Server
3. Client
4. GUI
5. Vergleich Anforderungen – Leistungen
6. Ausblick

Zielsetzung

Spezifikation und Implementierung eines Demonstrators zur Steuerung einer Märklin Modelleisenbahn für den Tag der offenen Tür, konkret:

- Bidirektionale Kommunikation vom Client über den Server zur Märklin Central Station
- Steuerung von mehreren Loks
- Geschwindigkeit, Richtung, Start, Stopp
- Stellen von Weichen
- Steuerung der Drehscheibe
- Notaus der Märklin Central Station
- Statusausgaben von Loks, Weichen und der Drehscheibe
- Einfaches GUI
- Alle Programme müssen auf den Labor PCs laufen

Server

1. Das Märklin-Protokoll (UDP)
2. Das Server-Client-Protokoll (TCP)

Das Märklin Protokoll

1.1 CAN Grundformat

Das CAN Protokoll schreibt vor, dass Meldungen mit einer 29 Bit Meldungskennung, 4 Bit Meldungslänge sowie bis zu 8 Datenbyte bestehen.

Die Meldungskennung wird aufgeteilt in die Unterbereiche Priorität (Prio), Kommando (Command), Response und Hash. Die Kommunikation basiert auf folgendem Datenformat:

Meldungskennung				DLC	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Prio	Command	Resp.	Hash	DLC	D-Byte 0	D-Byte 1	D-Byte 2	D-Byte 3	D-Byte 4	D-Byte 5	D-Byte 6	D-Byte 7
2+2 Bit	8 Bit	1 Bit	16 Bit	4 Bit	8 Bit	8 Bit	8 Bit	8 Bit	8 Bit	8 Bit	8 Bit	8 Bit
Message Prio	Kommando Kennzeichnung	CMD / Resp.	Kollisionsauflösung	Anz. Datenbytes	Daten	...						

vgl. cs2CAN, S.5

Das Märklin Protokoll

Im Ethernet werden immer Pakete mit 13 Bytes übertragen, unabhängig von der CAN - Datagramgröße, da das CAN - Ethernet - Gateway Pakete anderer Länge verwirft.

Die Bytes in der CAN- Botschaft werden folgendermaßen in dem UDP-Paket eingepackt:

- Bytes 1 bis 4 sind die Meldungskennung.
- Byte 5 entspricht dem DLC der CAN-Meldung.
- Bytes 6 - 13 sind die entsprechenden Nutzdaten.
Dabei nicht benötigte Bytes sind mit 00 zu füllen.

vgl. cs2CAN, S.7

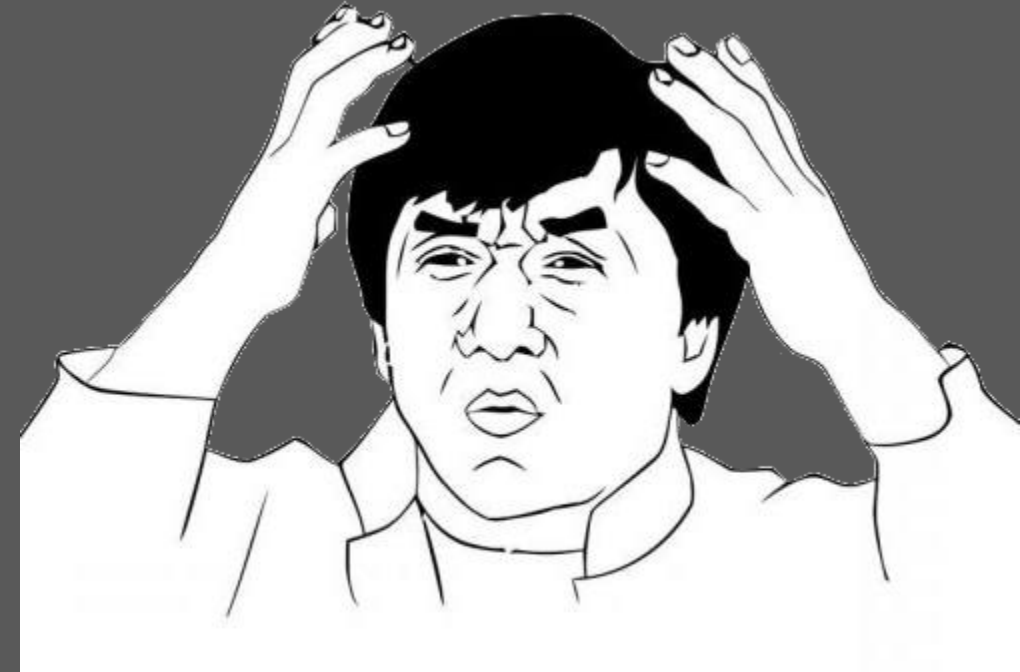
Das Mysterium der Meldungskennung

Märklin cs2CAN, Kapitel 1.2.7: „Bytes 1 bis 4 sind die Meldungskennung.“

Märklin cs2CAN, Kapitel 1.1: „29 Bit Meldungskennung“

Meldungskennung			
Prio	Command	Resp.	Hash
2+2 Bit	8 Bit	1 Bit	16 Bit
Message Prio	Kommando Kennzeichnung	CMD / Resp.	Kollisionsauflösung

Märklin: $4 * 8 = 29!$



Die Lüge des Hash



HATTU HASH?
SCHREIBSTU **0x0778!**

„Der Hash dient dazu, die CAN Meldungen mit hoher Wahrscheinlichkeit kollisionsfrei zu gestalten. Dieser 16 Bit Wert wird gebildet aus der UID Hash.

Berechnung: 16 Bit High UID XOR 16 Bit Low der UID.

Danach werden die Bits entsprechend zur CS1 Unterscheidung gesetzt.“

vgl. cs2CAN, S6

Dies widerspricht in jeder Hinsicht den mittels UDP Sniffer gewonnenen, praktischen Erkenntnissen.

Diese suggerieren eine statische Prüfsumme von 0x0778, womit diese effektiv nutzlos ist.

Die einzig wahre Meldungskennung

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12
PRI0	CMD	HASH		DLC	UID				Befehlsdaten			
0x00		0x07	0x78	Länge	0x00	0x00	Märklin ID					

Beispiel: Lok mit **MFX** Adresse **2** auf **Geschwindigkeit 300** (= 0x012C) setzen

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12
PRI0	CMD	HASH		DLC	UID				Befehlsdaten			
0x00	0x08	0x07	0x78	0x06	0x00	0x00	0x40	0x02	0x01	0x2C		

Beachtet man die Eigenarten der UDP Umsetzung des Märklin Protokolls, so können die Werte für CMD, UID und die Befehlsdaten einfach der cs2CAN Dokumentation entnommen werden. Diesbezüglich ist diese korrekt.

Server

Grundlage: Märklin Protokoll (drittes Praktikum) bestehend aus:

- Modelle für Loks und Weichen
- Kommunikationsklasse (sendet Datagramme zur Station)
- Listener (empfängt Datagramme der Station und updatet Modelle)

Server

Ansprüche an den Server:

- Sicherheit
- Multiclient-Fähigkeit
- Synchrones Statusupdate für alle Clients

Server

Sicherheit

Server

Multiclient-Fähigkeit:

- threadbasierter Ansatz
 - Server-Thread
 - Client-Thread (eigene Instanz je Client)
 - Handling-Thread (multiple Instanzen je Client möglich)
 - Update-Thread (eigene Instanz je Client; folgt)
- Verwaltung aller registrierter Clients in einer Array-List
- Separierung der Threads nach Aufgabenbereich in Executors
 - SERVER_THREAD_EXECUTOR: Server-Thread
 - CLIENT_THREAD_EXECUTOR: Client-Threads
 - UTILITY_THREAD_EXECUTOR: Handling- und Update-Threads

Server

Synchrones Statusupdate aller registrierten Clients:

- Eigener Thread für jeden Client
- Aufruf in definierten Zeitintervallen
- Formatierung der serverseitigen lokalen Abbilder der Elemente in Byte-Form
- Eindeutige Kennzeichnung durch Implementierung von Start- und Stop-Bytes

```
while (true){  
    if (System.currentTimeMillis()-timestamp >=  
        updateThreshold) {  
        server.statusUpdate();  
        timestamp = System.currentTimeMillis();  
    }  
    else {  
        try {  
            Thread.sleep(50);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Client

1. Kommunikation mit dem Server
2. Models - Statusabbilder der zu steuernden Objekte

Kommunikation mit dem Server

Kernelemente der Client-Server-Kommunikation:

- Herstellen und Trennen der Verbindung
- Aufrechthalten der Verbindung
- Update-Funktionalität

Kommunikation mit dem Server

```
public void establishConnetion () {
    if (connectionEstablished) {
        try {
            byte[] datagram = {Properties.SEPERATOR, (byte) 0x00, (byte) 0x00,
                               (byte) Properties.SESSION_ABORT, (byte) 0x00, (byte) 0x00};
            out.write(datagram);
            client.close();
            connectionEstablished = false;
            running = false;
            settingsController.updateSettingsStatus();

        } catch (Exception e) {
            setStatus("Error at: establishConnection (shutting down connection)");
            e.printStackTrace();
        }
    }
    else {
        UTILITY_THREAD_EXECUTOR.submit((new EstablishConnection(this)));
    }
}
```

Kommunikation mit dem Server

Aufrechterhalten der Verbindung:











































- Im Hintergrund mittels eines eigenen Threads (Handling ebenfalls via Executor-Services)
- Hält Input- und Output-Streams offen
- Submitted einmalig einen Update-Thread
- Zyklisches Update der GUI-Elemente

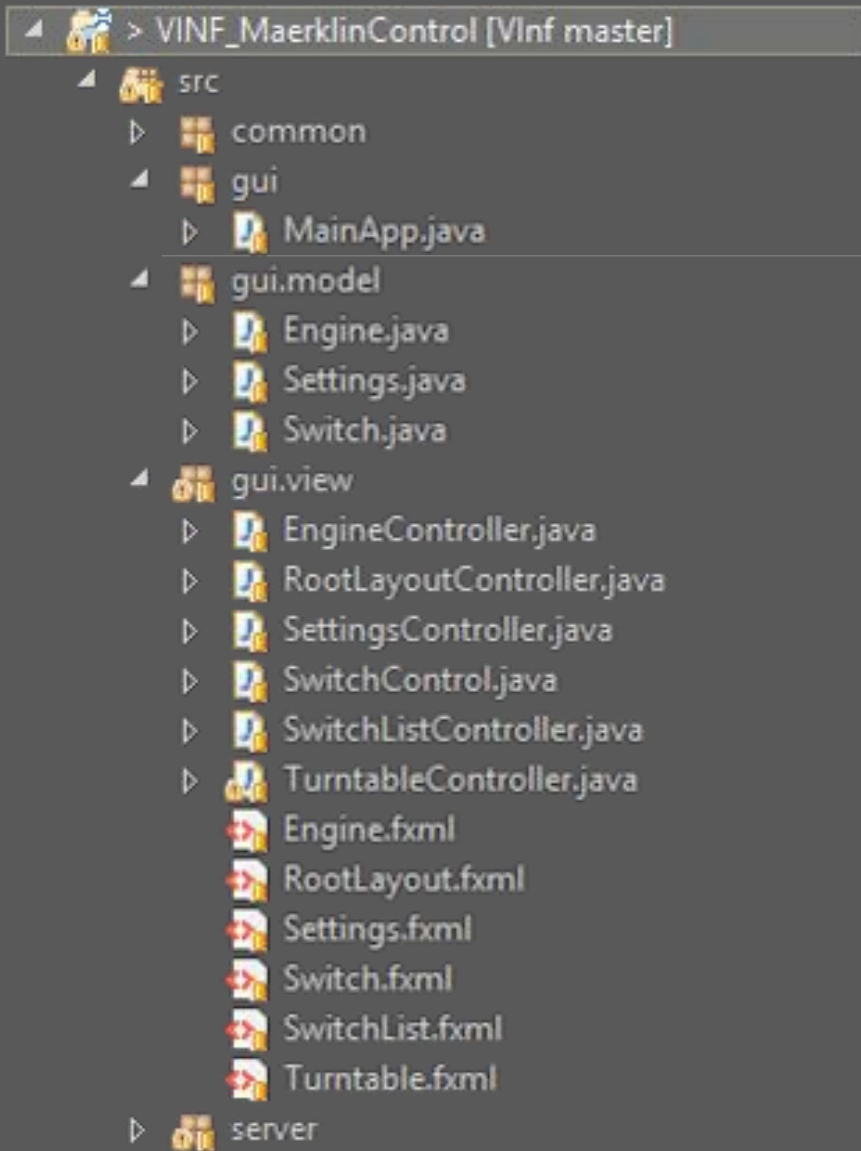
Kommunikation mit dem Server

Update-Funktionalität:

- Hört dauerhaft auf den Update-Port des Servers
- Decodiert das gesendete Datagramm ab einem erkannten Startbyte
- Updatet lediglich die Werte der Modelle (Persistenzebene) und nicht die GUI selbst
 - Vermeidung von Nebenläufigkeits-Problematiken

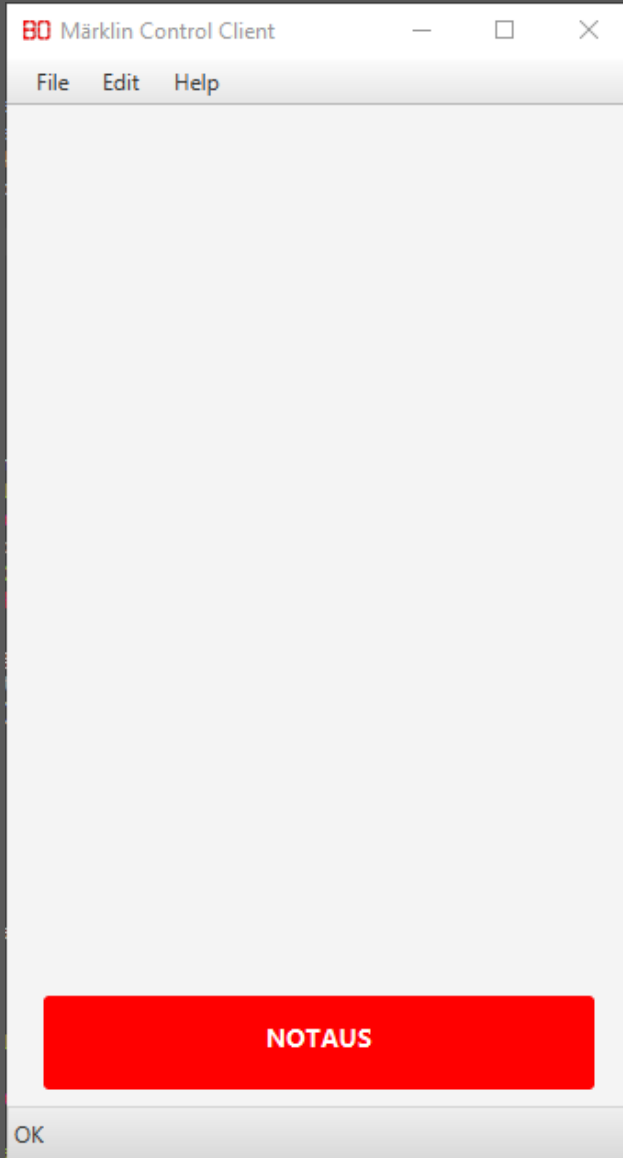
Models - Statusabbilder der Objekte

 Engine.java	 Switch.java	 Settings.java
 Engine	 Switch	 Settings
 fwd	 controller	 server
 id	 id	 Settings()
 img	 name	 getSocketAddress() : ObjectProperty<InetSocketAddress>
 name	 state	 setSocketAddress(InetSocketAddress) : void
 speed	 Switch()	
 Engine()	 Switch(String, Integer)	
 Engine(String, Integer)	 Switch(String, Integer, SwitchControl)	
 Engine(String, Integer, String)	 getController() : ObjectProperty<SwitchControl>	
 getDirection() : BooleanProperty	 getMaerklinID() : IntegerProperty	
 getImg() : ObjectProperty<Image>	 getName() : StringProperty	
 getMaerklinID() : IntegerProperty	 getState() : BooleanProperty	
 getName() : StringProperty	 setController(SwitchControl) : void	
 getSpeed() : IntegerProperty	 setName(String) : void	
 setDirection(Boolean) : void	 setState(Boolean) : void	
 setImg(Image) : void	 toString() : String	
 setSpeed(Integer) : void		
 toString() : String		



GUI – Grafische Benutzeroberfläche

1. Views – Dynamisches Einbinden von .fxml Dokumenten zur Laufzeit
2. Controller – Reagieren auf Ereignisse in der GUI und auf dem Gleis



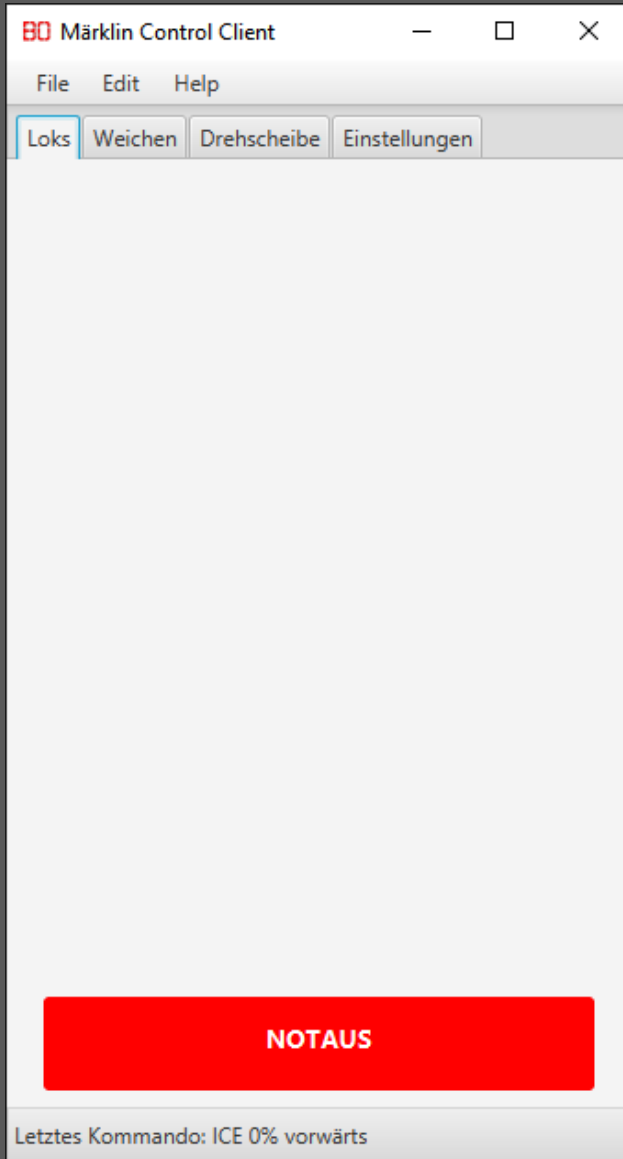
RootLayout.fxml

Das Basislayout der GUI

- Bereitstellen globaler Bedienelemente
 - Beispiel: Notaus, Statusleiste (controlsfx-Paket)
- Bereitstellen der TabPane

RootLayoutController.java

```
public void setMainApp(MainApp mainApp) {  
    this.mainApp = mainApp;  
}  
  
public void loadTabs(){  
    rootTabPane.getTabs().addAll(mainApp.getTabs());  
}  
  
public void setStatus(String status){  
    statusBar.setText(status);  
}  
  
public void emergencyStopHandler(){  
    mainApp.emergencyStopHandler();  
}
```



Erweiterung des Basislayouts

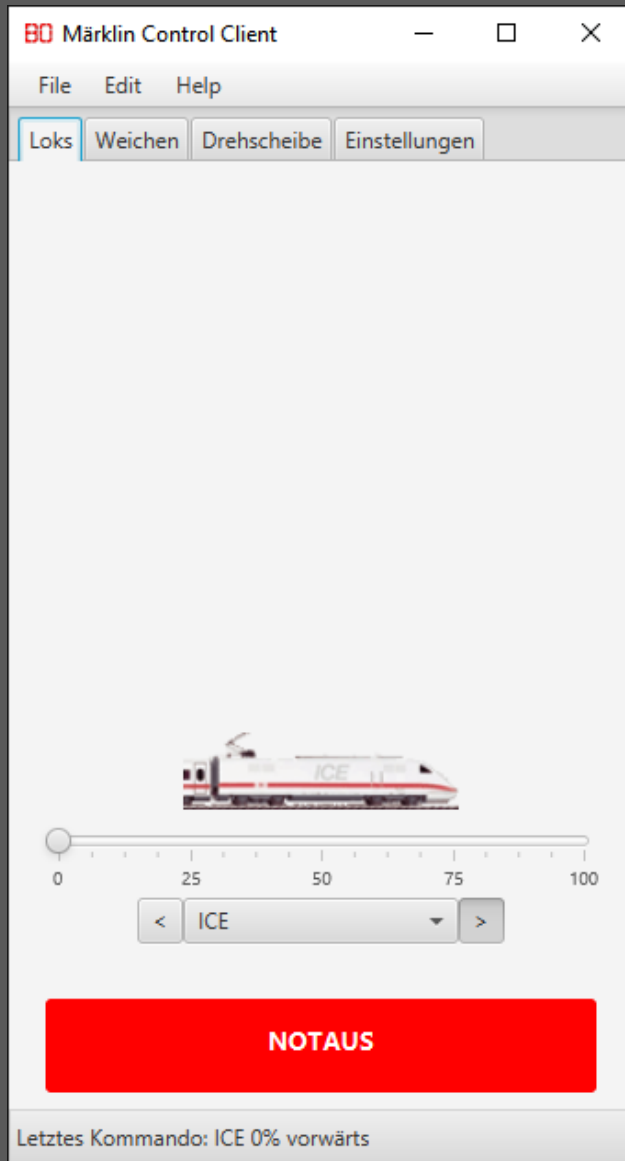
Dynamisch generierte Tabs

```
// Add the tabs
tabs.add(new Tab("Loks", enginePane));
tabs.add(new Tab("Weichen", switchListPane));
tabs.add(new Tab("Drehscheibe", turntablePane));
tabs.add(new Tab("Einstellungen", settingsPane));
rootController.loadTabs();
```


Engines.fxml Lokomotiven – Tab

Stellt Bedienelemente zur

- Auswahl und
 - ChoiceBox zur Auswahl der Lokomotive
 - ImageView zur Visualisierung der Auswahl
- Steuerung der Lokomotiven bereit
 - ToggleButtons in ToggleGroup zur Richtungswahl
 - Slider für die Steuerung der Geschwindigkeit

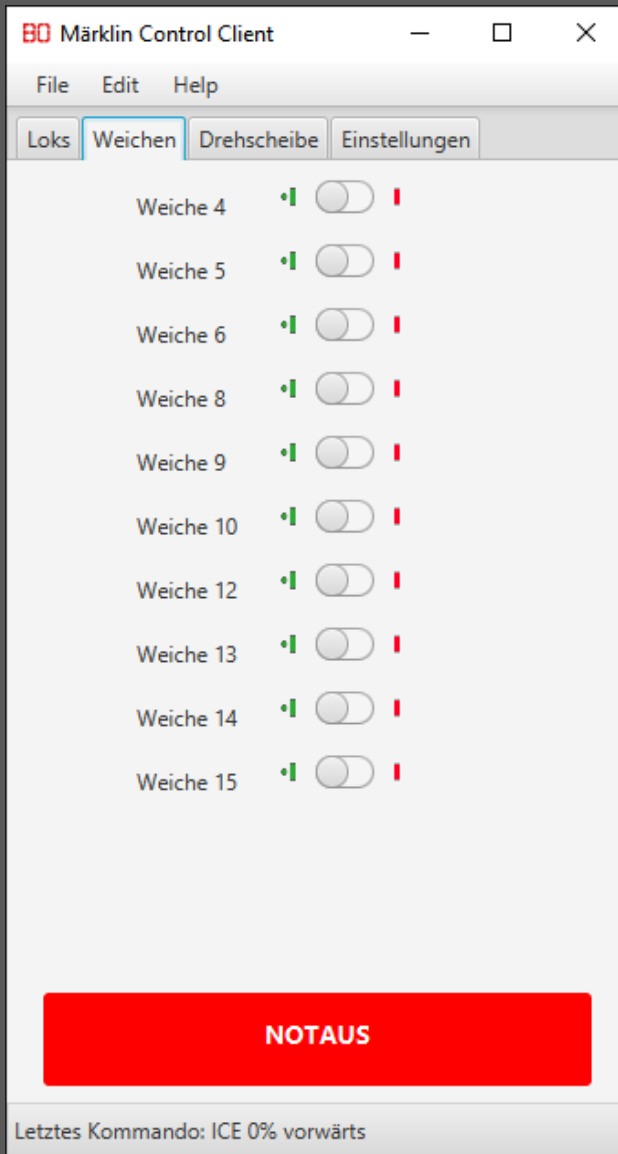


EngineController.java - Listener

@FXML

```
private void initialize() {
    engineSpeedSlider.setValue(0);
    engineFwdButton.setSelected(true); // initializes the forward Button as pressed
    engineChoiceBox.getSelectionModel().selectedItemProperty().addListener(
        (observable, oldValue, newValue) -> setSelectedEngine(newValue));

    engineSpeedSlider.valueProperty().addListener(new ChangeListener<Number>() {
        public void changed(ObservableValue<? extends Number> ov,
            Number old_val, Number new_val) {
            eng.setSpeed(new_val.intValue()*10);
            mainApp.setEngineSpeed(eng);
            updateEngineStatus();
        }
    });
}
```



SwitchList.fxml Weichen – Tab

- Stellt lediglich eine VBox in einer AnchorPane bereit
- Diese VBox wird dynamisch zur Laufzeit mit Weichenbedienelementen - Switch.fxml – gefüllt
- Ermöglicht eine effiziente GUI-Gestaltung, vgl. mit objektorientierter Programmierung -
- Keine „Code“ - Verdoppelung

SwitchListController.java

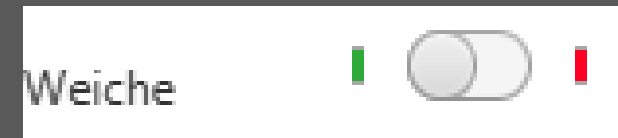
Dynamisches Anlegen von Bedienelementen

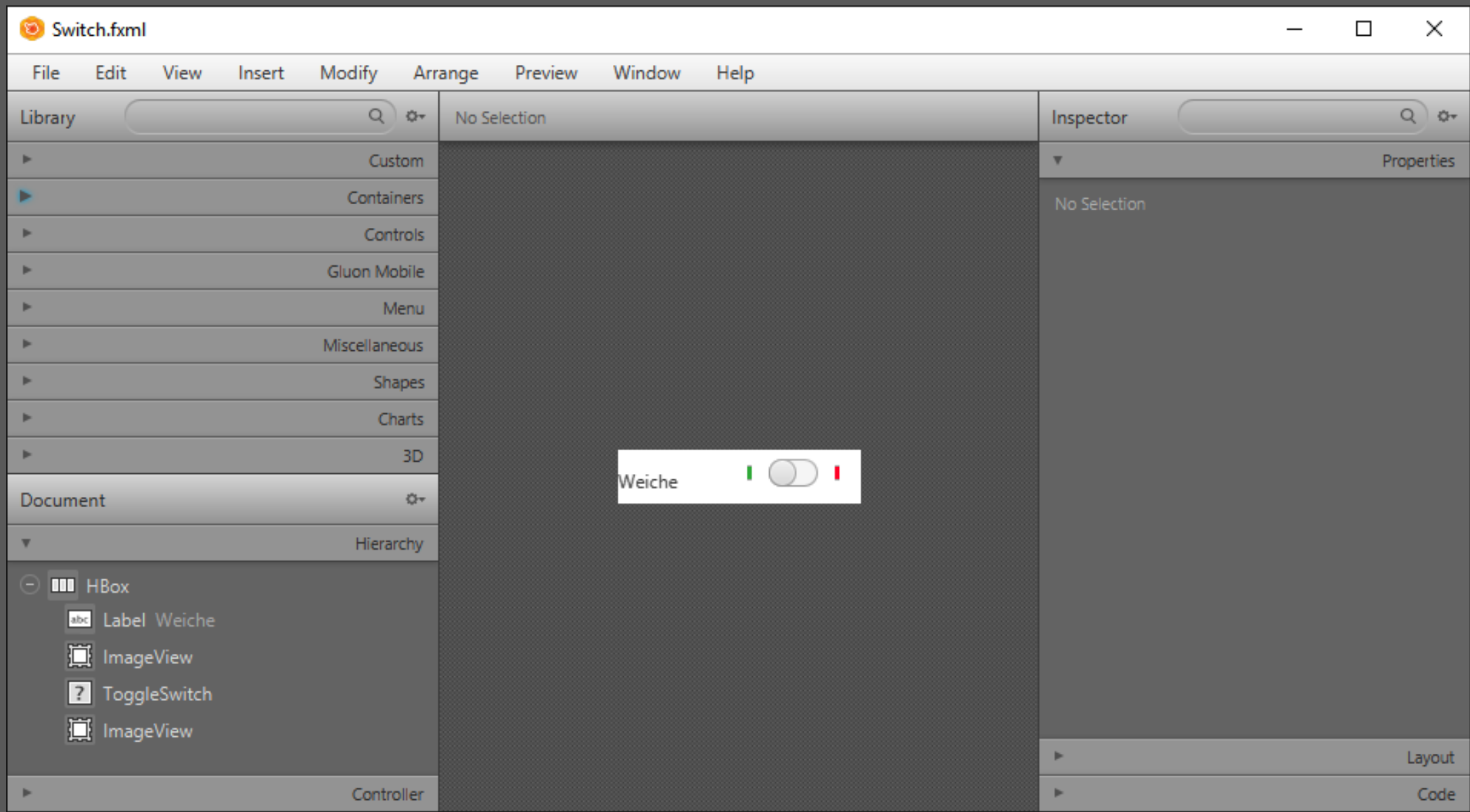
```
public void setSwitchControls(){
    ObservableList<Switch> switches = mainApp.getSwitches();
    ObservableList<SwitchControl> switchControls =
        FXCollections.observableArrayList();

    for ( int i=0; i<switches.size();i++ ){
        switchControls.add(new SwitchControl(switches.get(i)));
        switches.get(i).getController().get().setMainApp(mainApp);
    };
    switchesVBox.getChildren().addAll(switchControls);
}
```

Switch.fxml

- Zusammenfassung mehrerer Bedienelemente zu einem Sonderbedienelement
 - HBox
 - Label
 - zwei ImageViews
 - ToggleSwitch (controlsfx-Paket)



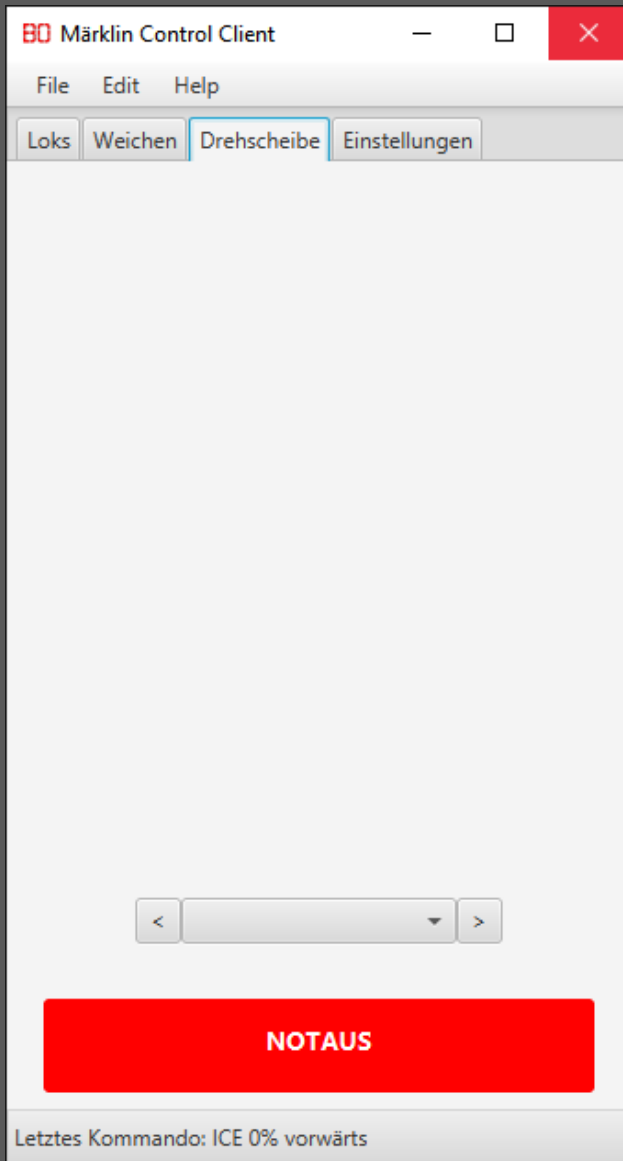


SwitchControl.java - Constructor

```
public SwitchControl(Switch sw) {  
    this.sw = sw;  
  
    FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource("Switch.fxml"));  
    fxmlLoader.setRoot(this);  
    fxmlLoader.setController(this);  
    sw.setController(this);  
  
    try {  
        fxmlLoader.load();  
    } catch (IOException exception) {  
        throw new RuntimeException(exception);  
    }  
}
```

SwitchControl.java – initialize()

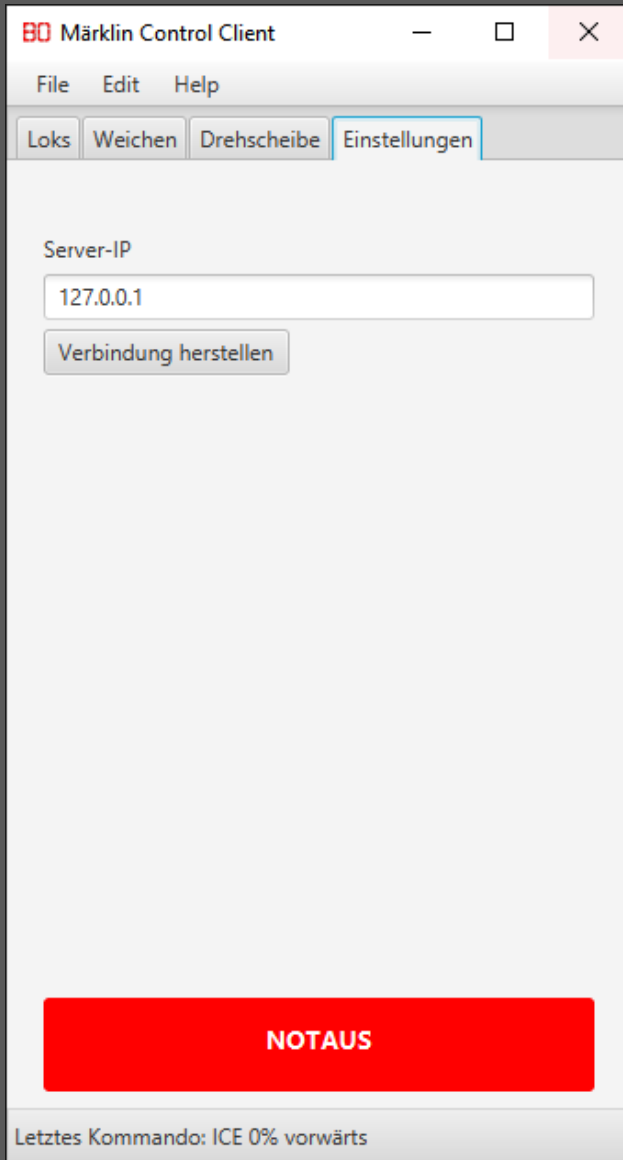
```
private void initialize() {  
    nameLabel.setText(sw.toString());  
    straightIcon.setImage(straightActive);  
    bentIcon.setImage(bentInactive);  
    stateToggleSwitch.selectedProperty().addListener(new ChangeListener<Boolean>(){  
        public void changed(ObservableValue<? extends Boolean> observable, Boolean  
            oldValue, Boolean newValue) {  
            sw.setState(newValue);  
            mainApp.setSwitchState(sw);  
            updateSwitchStatus();  
        }  
    });  
}
```

Turntable.fxml

Drehteller – Tab

- Stellt Bedienelemente zum Verfahren des Drehtellers bereit
 - Richtungsvorwahl durch ToggleButtons
 - Positionsvorwahl durch eine ChoiceBox



Settings.fxml Konfigurations – Tab

- Stellt Bedienelemente zur Herstellung einer Verbindung zum Server bereit
 - Eingabe des Hostnamens oder der IP
 - Herstellen und Trennen der Verbindung durch einen (1) Button
 - Bei erfolgter Verbindung Bestätigung dieser durch aktualisieren des Button Textes zu „Verbindung trennen“ und Meldung auf der Statusleiste

Vergleich Minimalanforderungen – Leistungen

- ✓ Bidirektionale Kommunikation vom Client über den Server zur Märklin Central Station
- ✓ Steuerung von mehreren Loks
- ✓ Geschwindigkeit, Richtung, Start, Stopp
- ✓ Stellen von Weichen
- ✓ Steuerung der Drehscheibe
- ✓ Notaus der Märklin Central Station
- ✓ Statusausgaben von Loks, Weichen und der Drehscheibe
- ✓ Einfaches GUI
- ✓ Alle Programme müssen auf den Labor PCs laufen

Vergleich Zusatzanforderungen – Leistungen

- Sonderfunktionen der Züge
 - Modellabhängig, z. B. Licht, Dampf, Signalhorn, etc.
- ✓ Multi-Client-Fähigkeit: Zugriff durch mehr als ein Bediengerät
- Benutzerverwaltung
- Einpflegen von Benutzern; Authentifizierung von Benutzern
- ✓ Statusausgaben von Loks, Weichen und der Drehscheibe synchron auf allen Clients
- ✓ GUI mit strukturierter Navigation und mehreren Screens

Ausblick

Potentielle Ergänzungen, auf die mangels einer Benotung verzichtet wurden:

- Nutzerverwaltung auf Client- und Serverseite
- Sonderfunktionen der Züge
- Erweiterte Drehscheiben-Steuerung – gezieltes Anfahren bekannter Positionen
- Performance-Evaluation – Laufzeitmessungen zwischen Client – Server – CS2

Vielen Dank für Ihre und
Eure Aufmerksamkeit!

Bildquellen

Lokomotiven-Bilder in der GUI
Gebr. Märklin & Cie. GmbH
<http://www.maerklin.de/de/>

BO-Logo & CVH Logo
Hochschule Bochum
<http://www.hs-bochum.de/campus-velbert-heiligenhaus.html>

Hattu Häschen
MOSES VERLAG
© Grosse Holtforth, Isabel

Alle Bildrechte verbleiben bei den
Eigentümern.

Eine Weitergabe ist nur
hochschulintern gestattet!

Quellen

[cs2CAN]
Märklin cs2CAN Protokoll
http://www.maerklin.de/fileadmin/media/service/software-updates/cs2CAN-Protokoll-2_0.pdf

[makery]
JavaFX 8 Tutorial
Model-View-Controller Konzept und Beispielumsetzung
<http://code.makery.ch/library/javafx-8-tutorial>

[benGale]
Creating a reusable JavaFX custom control
<https://programmingwithpassion.wordpress.com/2013/07/07/creating-a-reusable-javafx-custom-control/>

[javaFX]
JavaFX API Documentation
<http://docs.oracle.com/javase/8/javafx/api/toc.htm>

[controlsFX]
ControlsFX API Documentation
<http://controlsfx.bitbucket.org/>

[moodle]
Kursseite Vertiefung Informatik
<https://moodle.hs-bochum.de/course/view.php?id=125>

Alle Onlinequellen aufgerufen am 3.1.2017

Download

Zur hochschulinternen Verwendung:
<https://gitlab.cvh-server.de/lf.ps/VInf/>

