Hochschule
Augsburg University of
Applied Sciences

Visualisierung eines RFID-3D-Scanners
Technische Dokumentation

von

Falk Alexander, Tobias Scholze, Miriam Berschneider, Felix Wagner, Thomas Hipp, Valon Berisha, Michael Stadelmeier, Manuel Feyrer, Maximilian Schrupp, Jennifer Meier

der

Fakultät für Informatik

12. Juli 2012

Inhaltsverzeichnis

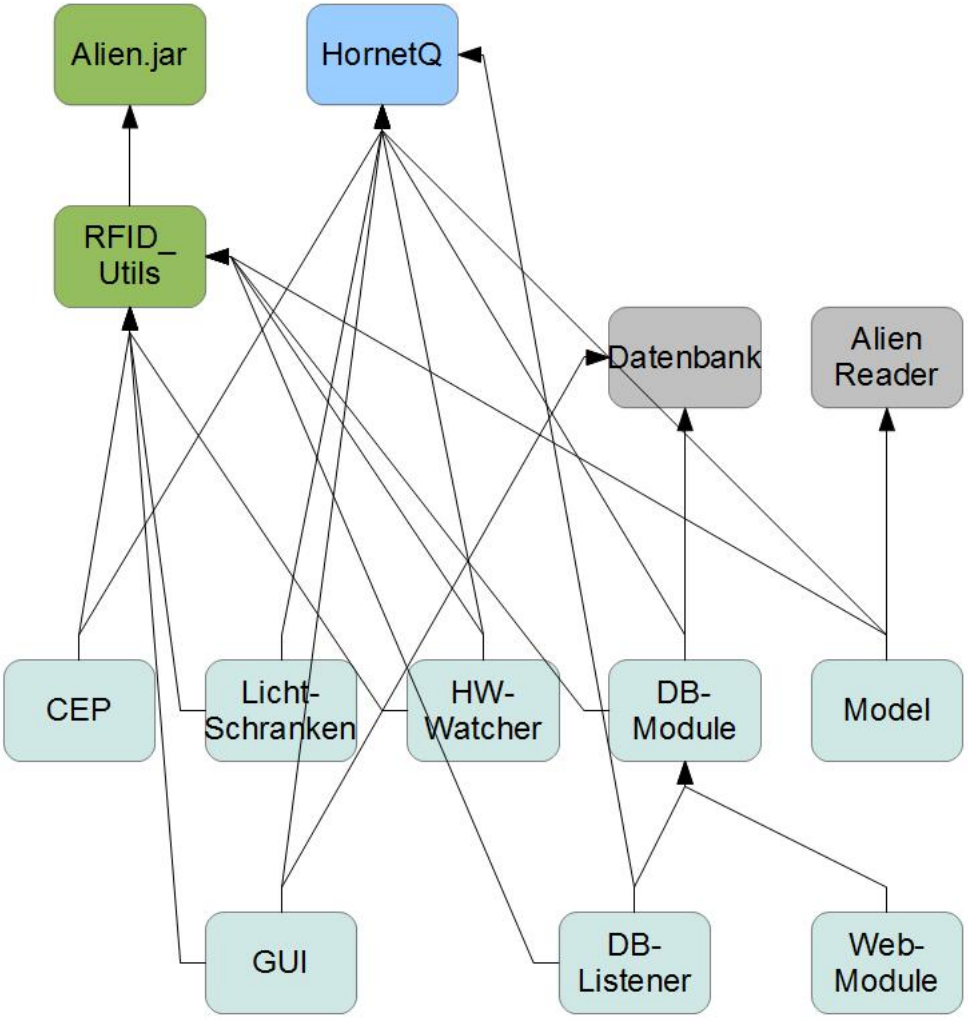
Abbildungsverzeichnis

1 Abhängigkeiten

1.1 Übersicht

Abbildung 1.1 zeigt, wie wir *HornetQ* und *rfid_utils* benötigen um alle anderen Packete zu bauen.

Abbild 1.1: Abhängigkeiten



1.1.1 HornetQ

Die oberste Abhängigkeit ist *HornetQ*. Dadurch wird die Kommunikation der Komponenten erst ermöglicht.

1.1.2 RFID_Utils

In diesem Paket befinden sich Klassen die von vielen Komponenten benötigt werden. Das ist z.B. TopicConnector um auf JMS Topics zuzugreifen. RFID_Utils muss mit mvn install allen anderen Paketen bereitgestellt werden.

1.1.3 CEP-Server

User CEP-Server hängt von HornetQ und RFID_Utils ab.

1.1.4 DB$_M$*odule*

Startet die DB ggf. in-memory-DB und füllt sie mit Einträgen. Hängt von RFIDund HornetQ ab. *mvn install* muss aufgerufen werden, da andere Bereiche Abhängigkeiten auf *db_module* besitzten.

1.1.4.1 Reihenfolge

Zuerst die DB Instanz: *mvn exec:java -Dexec.mainClass="org.hsqldb.Server" -Dexec.args="-database.0 file:target/data/db_module"* Danach wird die DB mit Einträgen gefüllt: *Test-Launcher* Die *Listener* werden mit *RunListener* gestartet.

1.1.5 Hw_Encapsulater

Startet alle Prozesse die HW Informationen an die CEP Schicht weitergeben oder diese Informationen simulieren. Hängt von *HornetQ* und *RFID_Utils* ab. Bei einem Einsatz mit Hardware müssen die Lichtschranken und der Alienreader erreichbar sein. Im *hw_encapsulater* startet die App ein Model, das auf den AlienReader zugreift. *hw_encapsulater* 's *StartLightBarrier* fragt die Lichtschranken ab.

1.1.6 Web_Module

Erzeugt eine Webseite, die die DB Einträge anzeigt und somit eine mobile GUI ermöglicht. Hängt ab von *HornetQ* und *DB_Module* (DB-Server läuft und ist erreichbar). *mvn install* (Es muss ein *WAR* [1] erstellt werden) Die Weboberfläche wird mit *mvn jetty:run* gestartet.

1.1.7 Hw_Watcher

Zeigt die Informationen an, die an die CEP-Schicht gehen. Hängt von *HornetQ* und *RFID_Utils* ab. Ist logisch gesehen von der HW abhängig. Ohne HW wird eben nichts angezeigt.

1.1.8 GUI

Zeigt Events grafisch an. Hängt von *HornetQ* und *RFID_Utils* ab und muss nach dem Model gestaret werden. Die Gui fragt daten von der DB ab. Die GUI benötigt eine erreichbare DB. Die letzte Version der GUI befindet sich in *branches/GraphicalUserInterface* Das erstellte Jar-Archiv muss ausgeführt werden.

2 HornetQ

2.1 Über HornetQ

HornetQ ist eine Implementierung eines Java Messaging Service's.

2.1.1 MessageListener

Pro Consumer kann es nur einen *MessageListener* geben.

2.1.2 Verbindung sauber beenden

Es ist wichtig die Verbindung ordnungsgemäß zu schließen. Wird dies nicht getan, so scheint es bei zukünftigen Verbindungen Probleme beim Zustellen von Nachrichten zu geben.

```
connection.stop();
consumer.close(); //ggf. producer
session.close();
connection.close();
initialContext.close(); //falls jndi verwendet wird
```

Listing 2.1: Verbindungsabbau

2.1.3 Serialisierte Objekte

Werden Objekte versandt müssen sie nicht nur aus der gleichen Klasse erzeugt werden, sondern sogar aus der selben Klasse.

2.1.3.1 Beispiel

Projekt A

- MainA.java
- GuiA.java
- Message.java

Projekt B

- MainB.java
- GuiB.java
- Message.java

Dieses Beispiel schlägt fehl. Message.java muss aus einem dritten Projekt kommen, zu dem die Projekte *A* und *B* eine Abhängigkeit besitzen. Dies lässt sich gut mit Maven lösen.

2.1.4 HornetQ Beispiel

2.1.4.1 Beispiele

Im SVN unter Examples befinden sich Beispiele. Die Beispiele sind alles eigene Maven Projekte mit einer eigenen *pom.xml*, die von der *pom.xml* des Prototypen getrennt ist. Will man die Beispiele ausführen muss man das Beispiel als eigenes Projekt ausführen (in einer IDE) oder von der Shell zu der *Beispiel-pom* wechseln und dort ein *mvn package* ausführen. Die pom's wurden getrennt um nicht relevante Abhängigkeiten aus dem Hauptprojekt fern zu halten.

2.1.4.2 HornetQ-Core-Test

Ein einfacher Nachrichtenaustausch über eine Queue. Zum Einsatz kommt die HornetQ-Core-API. Es wird ein passend konfigurierter HornetQ Server benötigt.

2.1.4.3 JMS-Test

Ähnlich wie HornetQ-Core-Test, nur dass die JMS API Verwendung findet. Hier wird ebenfalls ein passend konfigurierter Server benötigt.

2.1.4.4 Groupchat

In diesem Beispiel wird ein consolenbasierter Chat erstellt. Alle Teilnehmer klinken sich in einem Topic auf dem JMS ein. Der Nachrichtenaustausch wird mit Hilfe von einer eigenen Nachrichtenklasse bewältigt. Damit soll gezeigt werden, wie Serialisierung mit JMS funktioniert. Es wird ebenfalls ein Server benötigt. Der Topic lautet:

```
<topic name="ChattingTopic">
         <entry name="/topic/Chatting"/>
</topic>
```

Listing 2.2: hoernetq-jms.xml

Einfach auf der Kommandozeile tippen und mit Return absenden. Wird als Nachricht *:q* eingegeben, beendet sich der Client.

2.1.4.5 Groupchat_extended

Dieses Beispiel ist sehr ähnlich wie der Groupchat. Jedoch wird er um die Benutzung der Properties der Messages erweitert. Es ist möglich mit *:greet ¡botname¿* einen Bot zu grüßen und dieser antwortet dann. Beispiele und Erklärungen zu der Query-Syntax gibt es hier: [2] und [3]

3 HornetQ Connectoren

3.1 HwEventQueue

In diese Queue schreiben Model und LightBarrier ihre Daten. Diese gelangen an den CEP-Server. Damit hat die Queue einen Empfänger aber mehrere Sender. Es ist eine Queue, damit keine Meldungen an die CEP-Schicht verloren gehen. Es gibt 2 Arten von Messages auf dieser Queue:

1. RFIDTagArray
2. LightSensor

Durch das *StringProperty* type können sie unterschieden werden. Desweiteren gibt es noch 2 weitere *StringProperty* class für die Klasse im *MessageBody* und bei *LightSensor* noch den *sensorName* für den Sensor, der ausgelöst wurde. Ein *LongProperty timeStamp* gibt noch den Zeitpunkt an.

3.2 controltopic

Von einer Admin-GUI können Befehle an den AlienReader gesendet werden. Dieser antwortet auf dem selben Topic. Dieser Topic findet nur Verwendung falls ein Alien-Reader benutzt wird.

3.3 GuiTopic

Events werden von der CEP-Schicht an die Gui geschickt. Auf diesem Topic hört noch zusätzlich die DB mit um Events zu protokollieren. Es existiert wieder ein *timeStamp* als *LongProperty* und das *StringPorperty* class gibt an von welcher Klasse, das Event erstellt wurde. Sehr wichtig ist das *StringProperty uniqueName*. Jede Regel besitzt einen einzigartigen Namen. Dieser Name wird jedem Event mitgegeben, damit klar ist, welche Regel ausgelöst hat. Damit bekommt der Inhalt eine Bedeutung. Der Inhalt jeder *ObjectMessage* auf diesem Topic ist ein RFID-Tag-Array.

3.4 GuiInstructionTopic

Jede Regel die erstellt oder gelöscht wird, wird von der Gui zur CEP-Schicht auf diesem Topic versandt. Die DB hört auch mit und protokolliert alles Regel mit. Damit kann in der DB einem Event eine Regel zugeordnet werden. Versandt werden *RuleContainer* und als *StringProperty* wird der *uniqueName* gesetzt.

3.5 HwWatcherTopic

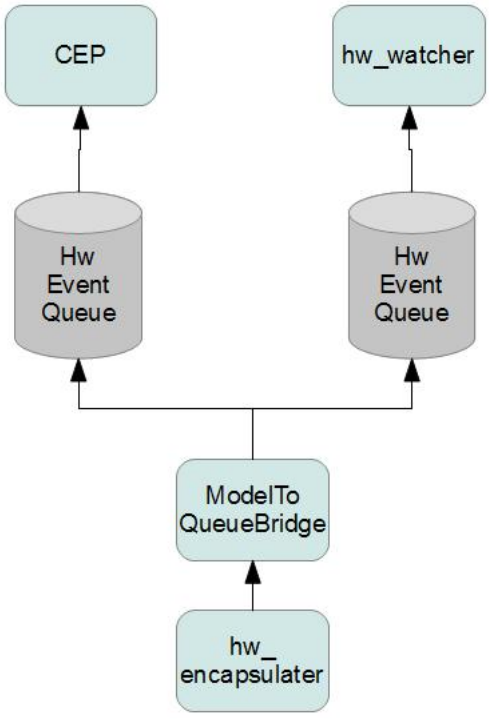
Dieser Topic erhält den exakt gleichen Input wie die *HwEventQueue*. Dieser Topic dient der Überwachung und dem Debugging der Hardware. Das Senden auf diesem Topic ist optional und muss in den Klassen beim Instanziieren explizit angegeben werden. Es existiert neben der *HwEventQueue* ein weiterer Topic, weil keine Lösung für mehrere Empfänger auf einer Queue praktikabel war.

4 hw_watcher

4.1 Zweck

Um direkt die Geschehnisse auf der Hardware zu verfolgen kann der hw_watcher benutzt werden. Er gibt alle Hardware Meldungen aus, die auch an die CEP-Schicht gehen. Verdeutlichen sollte dies Abbildung 4.1

Abbild 4.1: hw_watcher Architektur



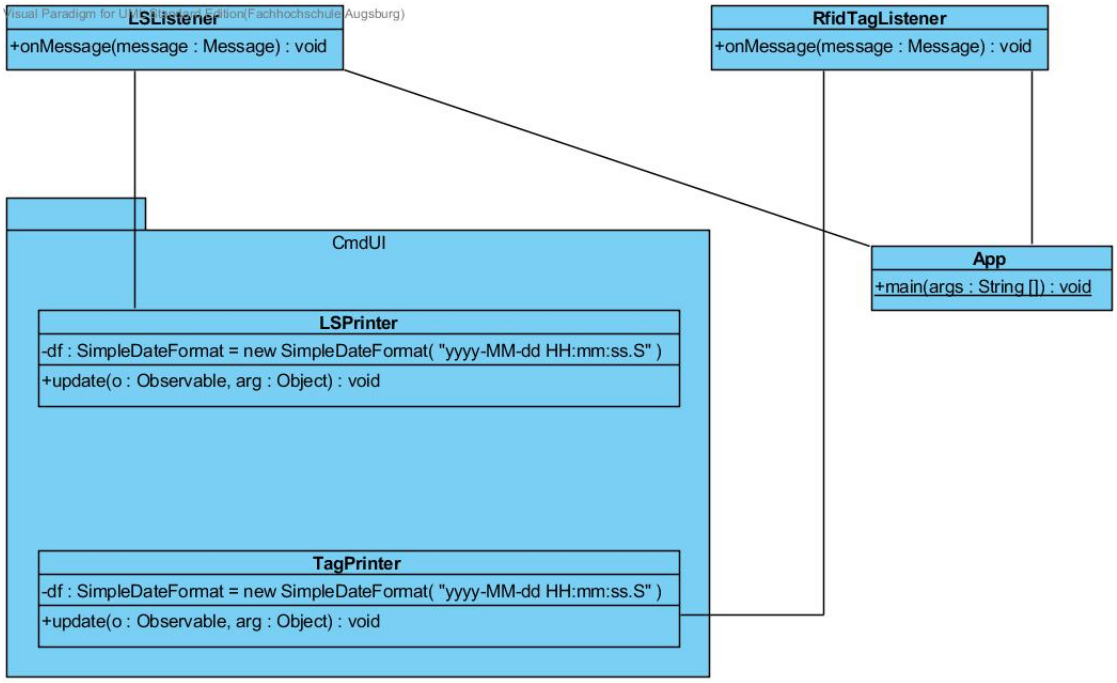
4.2 Aufbau

ModelToQueueBridge muss einen Topic bekommen, auf dem es zusätzlich die Informationen veröffentlicht. Angedacht ist der Topic */topic/HwWatcherTopic* dafür. Über die Methode setActiveWatcher wird ein TopicConnector übergeben. Auf der anderen Seite des Topics werden *MessageListener* auf *RFIDTagArray* und *LightSensor* gesetzt.

4.3 Eigenes Topic

Warum verwendet *hw_watcher* eine eigenes Topic und nicht die *HwEventQueue*? In einer Queue sind mehrere Empfänger problematisch. Mit einem *QueueBrowser* gäbe es die Möglichkeit über die Nachrichten einer Queue zu iterieren ohne sie zu entfernen, jedoch nur solange die Nachrichten nicht bestätigt werden. Die *HwEventQueue* soll auch nicht zum Topic werden, da die Nachrichten an die CEP-Schicht sicher zugestellt werden sollen. Eine Übersicht befindet sich in Abbildung 4.2.

Abbild 4.2: Listener Architektur



5 Model.java

5.1 Der Service

Um die gelesenen Tags vom Reader zu bekommen ist es nötig einen *MessageListenerService* zu erstellen und diesen zu starten. Der Service schickt dann, sobald er eine TagList hat, diese zu der Methode *messageReceived*, welche von der Klasse implementiert werden muss.

5.2 Der Reader

5.2.1 AlienClass1Reader

Um die Tags vom Reader als Stream zu erhalten ist es nötig, diesen zu initialisieren. Hierzu muss man ein Reader-Objekt mit IP und Port erstellen, welches eine TCP Verbindung zum Netzwerk aufbaut und die Antennen beinhaltet.

5.2.2 Notifiy

Damit die Tags als Stream empfangen werden können, muss man dem Reader-Objekt mit *setNotifyAddress* die IP des Rechners übergeben, welcher die gelesenen Tags als *Message-Objekt* entgegen nehmen soll.

Mit *setNotifyTrigger(Art)* ist es möglich die Art und Weise festzulegen wann die Tags gesendet werden. Hierzu gibt es folgende Möglichkeiten bzw. Arten:

Add Sobald ein neuer Tag gelesen und der TagList hinzugefügt wurde wird dieser Tag übermittelt.

Remove Sobald ein Tag gelesen und von der TagList entfernt wurde wird dieser Tag übermittelt.

Change Sobald ein Tag hinzugefügt bzw. entfernt wurde wird die komplette TagList übermittelt.

True Sobald ein Tag der TagList hinzugefügt wurde wird die komplette TagList übermittelt.

False Wenn kein Tag der TagList hinzugefügt wurde wird die komplette TagList übermittelt.

TrueFalse Es wird immer die komplette TagList übermittelt.

Damit das ganze funktioniert ist es erforderlich den *NotifyMode* auch zu aktivieren. (*setNotifyMode(AlienReader1Class.ON)*)

5.2.3 AutoMode

Mithilfe des AutoMode kann man den Datenfluss und das Monitoring steuern, indem man dem Reader mitteilt, wie er die Tags lesen soll, wann er sie Lesen soll und wenn Tags gefunden wurden von wem sie kommen. Z.B. *setAutoStopTimer(2000)* bedeutet das der Reader erst alle 2 Sekunden die *TagList* übermitteln soll. Da der *AutoMode* sehr umfangreich und komplex ist, benötigt es noch etwas Zeit diesen zu verstehen und richtig einzusetzen.

5.2.4 TagListAntennaCombine

Mithilfe von TagListAntennaCombine kann man festlegen welche Einträge in die *TagList* geschrieben werden z.B. bei 2 Antennen:

TagListAntennaCombine = ON: - Nur eine Antenne erstellt einen Eintrag. *TagListAntennaCombine = OFF:* - Beide Antennen erstellen einen Eintrag.

Dies kann später bei der Positionsbestimmung und zur Vermeidung von nicht gelesenen Tags von nutzen sein.

6 Native Java GUI

6.1 Enum: SettingParameter

SettingParameter definiert alle Einstellungen die in dem Programm vorgenommen werden können. Aus dieser Liste werden die Methoden-Aufrufe dynamisch generiert. Jedem Eintrag ist einer oder mehrere Default-Werte zugewiesen, um das Programm ausführen zu können ohne Anfangs Einstellungen tätigen zu müssen.

Attribute *private String[] defaultValue* Enthält die Standard-Parameter für den SettingParameter in Form eines String Array. Die Werte sind alle als String gespeichert, und müssen eventuell später in das Zielformat umgewandelt werden.

Methoden *public String[] getDefault()* Liefert das String Array *DefaultValue* zurück.

6.2 Enum: MethodType

MethodType definiert alle Filter welche in der Klasse *ReflectionManager* angewendet werden können.

Werte

SET Es wird nur nach *Set-Methoden* gesucht.

GET Es wird nur nach *Get-Methoden* gesucht.

ALL Es wird nach allen Methoden gesucht.

6.3 Enum: Instruction

Instructions definiert Befehle welche durch das Drücken der Buttons in der GUI ausgelöst werden und im Controller ausgeführt werden.

6.4 Model

Model ist die Daten-Schnittstelle des Programms. Es enthält alle aktuellen Werte, wie Einstellungen. Model ist ebenfalls dafür zuständig die gelesenen Tags bereitzustellen und weiter zu leiten. Weiterhin instanziiert und hält Model das Reader-Objekt, auf welchem Einstellungen für den RFID-Reader vorgenommen werden können. Model implementiert das Interface GenericModel.

Attribute

privateboolean running Definiert ob der Stream aktiv ist oder nicht. Wird von der StartStop-Methode geändert.

Private GenericReader reader GenericReader ist eine Generische Klasse, welche das Interface für das Reader-Objekt definiert. Reader Objekte müssen von GenericReader abgeleitet werden um die Funktionen zu erfüllen.

private MessageListenerService service Der Service, der die Nachrichten vom Reader entgegennimmt.

private Tag[tagList] Das Tag Array welche alle gelesenen Tags als Objekt enthält. Wird immer wieder bei Erhalt von neuen Tags überschrieben.

private Log log Das Log Objekt, welches die erhaltenen Tags auswertet und in einer Log-Datei speichert.

privateboolean stdOut Definiert ob Ausgaben auf Kommandozeilen-Ebene erfolgen sollen oder nicht.

privateboolean simpleLog Legt fest ob ein Log-File angelegt werden soll.

Methoden

public Model() Konstruktor

public void setServicePort(String port) Instanziiert den *MessagListenerService* und legt den entsprechenden Listener-Port fest, an welchen die Nachrichten (Tags) gesendet werden.

public void messageReceived(Message message) Diese Methode erhält die Tags vom Reader, die mittels Notification gesendet werden.

public void observeData() Diese Methode ist Teil des Observer-Patterns, signalisiert den Objekten76 welche dieses Objekt beobachten, dass Änderungen vorliegen und ruft anschließend die Update-Methode in den entsprechenden Objekten auf.

private void waitWhileReading() throws InterruptedException Hält den Model-Thread am leben, solange bis der Stop-Button gedrückt wird.

public void startStopReader(boolean running) throws Exception Setzt das running Attribut auf den übergebenen Parameter. Anschließend startet bzw. stoppt sie den Service und Stream.

public void start() throws Exception Ruft die StartStopReader-Methode mit *true* auf.

public void stop() throws Exception Ruft die StartStopReader-Methode mit *false* auf.

public boolean isStdOut() Gibt den Wert von *stdOut* zurück.

public void setStdOut(boolean stdOut Setzt den Wert von *stdOut*

public boolean isSimpleLog() Gibt den Wert von simpleLog zurück.

public void setSimpleLog(String simpleLog) Setzt den Wert von SimpleLog.

publicvoid applySettings(...) throws Exception Erhält als Übergabeparameter ein Dictionary mit den Einstellungsnamen (=Mehodennamen) und den zugehörigem(n) Wert(en). Alle Einstellungen welche auf dem Model gesetzt werden müssen, werden mithilfe einer ReflectionManager-Instanz gesetzt. Anschließend werden die Settings an den Reader übergeben.

public void initDefaultSettings() Instanziiert das settings-Dictionary mit den Default-Werten.

public void setDefaultSettings() throws Exception Überschreibt das settings-Dictionary mit den Default Werten und setzt diese anschließend.

public void run() Startet den Model-Thread und hält ihn am Leben.

6.5 Reader

Der Reader hält das AlienReader Objekt, auf welchem die Einstellung des RFID-Reader gesetzt werden können. Der Reader implementiert das GenericReader Interface.

Attribute

private AlienClass1Reader alienReader Das AlienClass1Reader-Objekt.

Methoden

public Reader(String alienIp) throws ClassNotFoundException Konstruktor

public void setSettings(...) throws Exception Erhält als Übergabeparameter ein Dictionary mit den Einstellungsnamen (=Mehodennamen) und den zugehörigem(n) Wert(en). Alle Einstellungen welche auf dem Reader gesetzt werden müssen werden mithilfe einer ReflectionManager-Instanz gesetzt.

public void turnOffReader() throws AlienReaderException Stoppt den Stream des RFID-Readers.

public void setReaderIp(String alienIp) Setzt die Netzwerk-Adresse des alienReaders auf einem neu instanziierten AlienReader-Objekt, weshalb diese Methode nicht vom ReflectionManager aufgerufen werden kann.

6.6 ReflectionManager

Der ReflectionManager ist sehr allgemein (generisch) programmiert, und somit unabhängig. Er regelt die Methodenaufrufe über das Reflection Pattern. Er parst automatisch die übergebenen Parameter aus einem String in das notwendige Format, und ruft anschließend die Methode auf.

Attribute

private T targetClass Das Objekt der Klasse in dem die Methoden ausgeführt werden sollen. Der Typ T ist generisch und wird beim erstellen des ReflectionManager definiert.

private MethodType methodType Der Enum-Wert welcher festlegt, nach welchen Methoden gefiltert werden soll.

private Method[classMethods] Ein Array welches alle verfügbaren Methoden der Klasse, in der die Aufrufe stattfinden sollen, beinhaltet.

Methoden

public ReflectionManager(T targetClass, MethodType methodType) Der Konstruktor der Klasse ReflectionManager. Er erhält ein Objekt vom generischen Typ T sowie einen MethodType Enum Wert, welcher den Filter für die Methoden festlegt.

privat evoid initReflecionManagerObjects() Ruft die gewünschte Filter-Methode auf.

privat evoid cleanNoSetMethods() Filtert nur die Set-Methoden heraus, und speichert sie in den jeweiligen Listen.

private void cleanNoGetMethods() Filtert nur die Get-Methoden heraus, und speichert sie in den jeweiligen Listen.

private void completeMethodList() Sucht alle Methoden heraus und speichert sie in den jeweiligen Listen.

private Object[getParamTypes(...)] Wandelt die Übergabe-Parameter vom Typ String in das benötigte Format um.

public void invokeMethodsFromMethodList(...) Erhält den Methodennamen der aufzurufenden Methode als String, sowie die/den Übergabeparameter als Object. Ruft die Methode zum umwandeln des Übergabeparameters auf und führt anschließend die Methode aus.

7 Complex-Event-Processing (CEP)

7.1 Begrifflichkeiten

7.1.1 Rule

Als eine Rule wird eine Regel in Esper bezeichnet. Sie hat einen einzigartigen Namen (uniqueName) mit dem sie eindutig identifiziert werden kann. Der einzigartige Name wird beim Erzeugen der Regel vergeben. Es ist möglich das zwei syntaktisch identische Regeln mit unterschiedlichen Namen existieren. Wird eine Regel als ObjectMessage verschickt, so wird ein StringProperty mit dem Key *uniqueName* auf den vergebenen Namen gesetzt. Ihr Repräsentation liegt im Projekt rfid_uitils und heißt RuleContainer. MessageListener, die auf eine bestimmte Rule warten, wenden Filter auf dieses Property an. Ein Beispielfilter ist: `uniqueName = 'all tags'`

7.1.2 Event

Ein Event ist ein einmaliges Zutreffen einer Regel in Esper. Ein Event besitzt den einzigartigen Namen der Regel und eine Referenz auf die Objekte, die auf die Regel zutreffen.

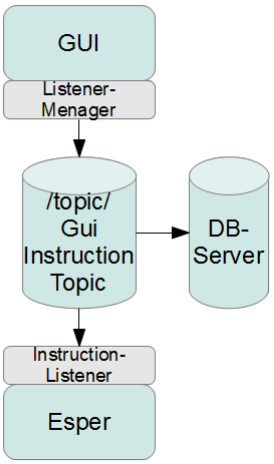
Wird ein Event als ObjectMessage verschickt, so soll die Regel, aus der das Event hervorgeht, mit dem Event verknüpft werden, d.h. der *uniqueName* der Rule soll als StringProperty an die ObjectMessage angefügt werden.

MessageListener, die auf ein bestimmtes Event warten, wenden Filter auf dieses Property an.

7.1.3 Neue Regeln

Ein möglicher Anwendungsfall ist, das auf der GUI eine neue Regel (Rule) erstellt wird und diese soll in Esper (CEP) aufgenommen werden. Gleichzeitig hört der DB-Server mit, ob neue Regeln erstellt werden. Er erfasst und speichert die Meta-Inforamtionen (Beschreibung ect.) ab. In der GUI wird vom ListenerManager eine neue Regel an den CEP-Server gesandt. Diese Regel ist eine Instanz von RuleContainer (rfid_utils). Die

Abbild 7.1: Neue Regeln in der GUI



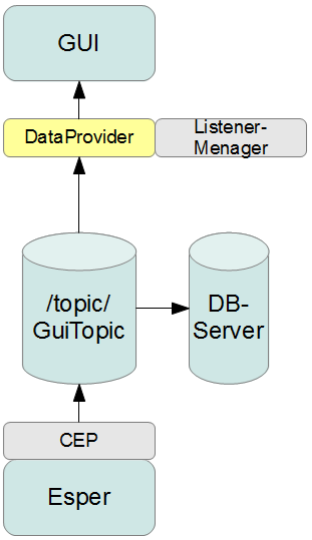
Meta-Daten der Regel, werden von der DB abgegriffen und gespeichert. Der CEP-Server erstellt eine neue Instanz von *MyUpdateListenerImpl*, die als Attribut den *uniqueName* der Regel hat. Da dieser Listener bei einem Eintreffen der Regel aufgerufen wird, kann an ein späteres Event dem *uniqueName* wieder beigefügt werden.

Alle weiteren Assoziationen zu der Regel müssen über die DB erledigt werden.

7.1.4 Ausgelöste Events

Wenn eine Regel zutrifft, wird von Esper das zugehörige Event ausgelöst. Mit dem Aufruf von *MyUpdateListenerImpl* wird eine ObjectMessage mit dem *uniqueName* der Regel erstellt und an die GUI gesandt. In der CEP.java wird eine neue OjectMessage erzeugt, die als StringProperty den *uniqueName* der Rule besitzt, weswegen sie ausgelöst wurde. Damit ist auf der Empfängerseite möglich auf bestimmte Events zu filtern. Der ListenerManager hängt an jede Regel die erstellt wurde einen MessageListener z.B. einen DataProvider (speziell dieser reicht seine Daten an die GUI weiter). Der DB-Server nutzt keinen Filter und nimmt alle Events in seine DB auf, die erzeugt wurden.

Abbild 7.2: Fluss der Events



7.2 Entwicklungshilfen

7.2.1 Was müssen Dritte über meinen Code wissen, um damit zu arbeiten?

Esper (CEP.java) und der Controller (Controller.java) spielen zusammen. Eine Controller Instanz verlangt eine GUI-, eine GenericModel- und eine CEP-Instanz. Die Observer werden dann so gesetzt, wie sie im Architektur Diagramm dargestellt sind. Die Kommunikation von GUI nach CEP läuft über den Controller, da die CEP-Schicht nicht wissen soll bzw. kann, was über ihr ist. Sie weiß lediglich, dass sie Events von einem GenericModel, einem Controller und von UpdateListenern bekommt.

CEP — hört auf → Controller (Regeln) CEP — hört auf → Model (Tags) CEP — hört auf → sämtliche UpdateListener (TagEvents)

Controller — hört auf → Gui (Regeln und Instructions)

GUI — hört auf → CEP (TagEvents)

7.2.2 Wie gebe ich Daten an Esper weiter?

Daten können nur via Controller an Esper übergeben werden. Momentan funktioniert das allerdings nur mit Regeln. Es muss noch darüber nachgedacht werden, wie man beispielsweise Regeln aktiviert und deaktiviert.

7.2.3 Wie bekomme ich Daten von Esper?

Daten bekommt man von Esper, wenn man sich als Observer registriert hat, da Esper ja eine Unterklasse von Observable ist. Die GUI wird vom Controller automatisch als Observer von CEP festgelegt.

```
...
CEP cep = new CEP();
cep.addObserver(MyObserver);
Controller ctrl = new Controller(new Gui(), new Model(), cep);
...
```

Listing 7.1: Daten von Esper

7.2.4 Wie erstelle ich eine neue Regel in Esper?

Eine neue Regel wird erstellt, indem die GUI via notifyObservers, ein String Array der Länge drei sendet. Der erste String beinhaltet die Regel, der zweite den Pfad zum UpdateListener und der dritte die eigentliche Klasse.

```
...
String[] s = {
        "select * from TagEvent",
        "classes/prototype/control",
        "prototype.control.MyUpdateListenerImpl"};
...
```

Listing 7.2: neue Regeln

Es ist wichtig zu wissen, dass ein UpdateListener eine Unterklasse von Observable sein muss und das UpdateListener-Interface benötigt wird. Die Datei MyUpdateListener-Impl.java ist ein Beispiel eines solchen UpdateListeners.

```
import java.util.Observable;
import com.espertech.esper.client.*;
public class MyUpdateListener extends Observable implements UpdateListener {
        ...
}
```

Listing 7.3: neuer Listener

8 Graphische Benutzeroberfläche

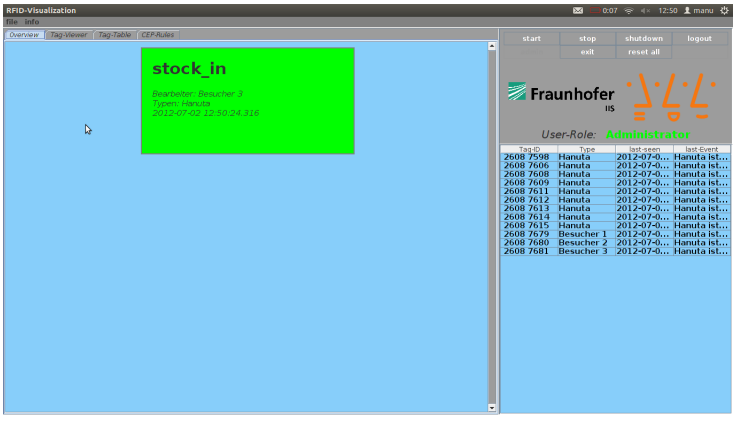
8.1 Übersicht

Die Benutzeroberfläche (GUI) ist in mehrere Bereiche aufgeteilt. Wie unten in den Bildern dargestellt befindet sich auf der linken Seite der GUI ein Fenster mit vier Reitern. In der oberen rechten Hälfte befinden sich die Steuerelemente, die nötig sind um die verschiedenen Situationen zu steuern. In der unteren rechte Hälfte befindet sich eine Tabelle, die die Informationen darstellt, die der jeweilige Usecase enthält (mehr davon im Reiter *Overview* 8.1.1).

8.1.1 Die Reiter

Overview Im Reiter "Overview" werden die einzelnen Usecases (hier stock_in für Wareneingang) dargestellt. Durch einen Klick mit der Maus auf den farbigen (hier: grün) Container, wird in der rechten unteren Bildschirmhälfte eine Tabelle mit den Inhalten dieses Usecases angezeigt. Der Reiter dient dazu, die aktuelle Situation darzustellen, zu

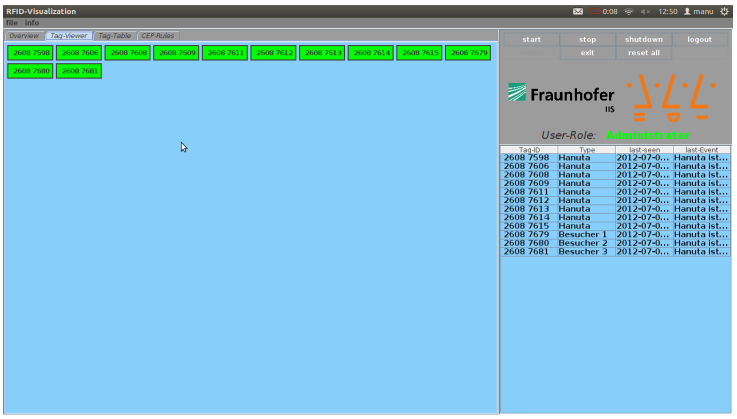
Abbild 8.1: Overview



analysieren und im gegebenen Fall auf Fehlerfälle zeitnah einzugreifen.

Tag-Viewer Im Reiter "Tag-Viewer" werden alle, also nicht nur für diesen Usecase gelesenen Tags dargestellt. Die einzelnen Tags werden grün dargestellt, wenn sie aktuell gelesen wurden, rot falls sie beim letzten mal nicht mehr gelesen wurden. Der Tag-

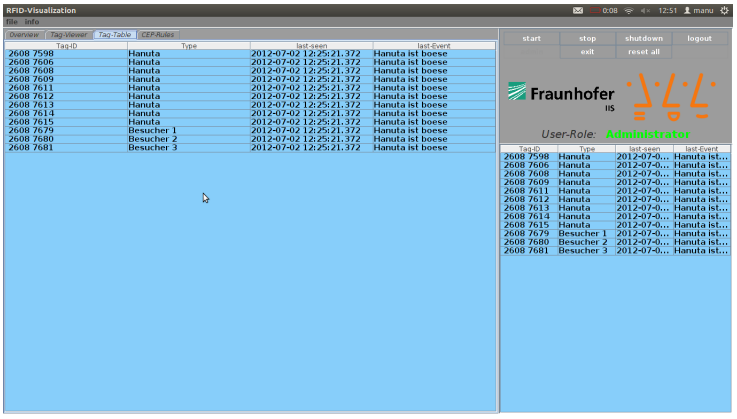
Abbild 8.2: Tag-Viewer



Viewer dient zur Übersicht und zur Kontrolle ob sich evtl. noch Tags im Feld bzw. Zwischen den Antennen befindet.

Tag-Table Die Tag-Table ist eine Tabelle, die alle gelesenen Tags mit Informationen über das jeweilige Produkt bzw. über die Person enthält, unabhängig von jeweiligen Usecase. Die Tag-Table dient zum Überwachen der einzelnen Tags und zur Analyse im

Abbild 8.3: Tag-Table



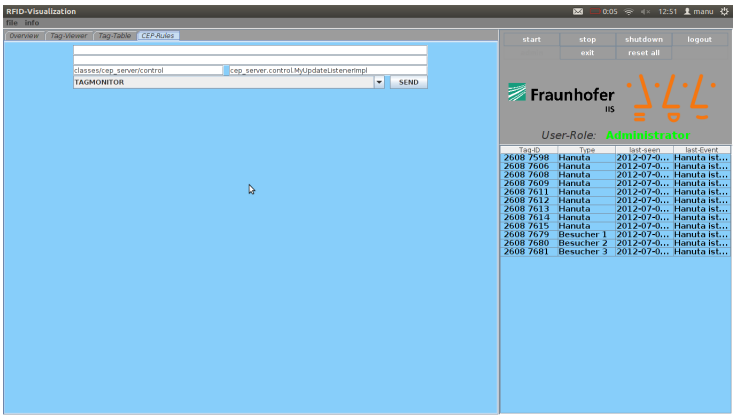
Fehlerfall.

CEP-Rules Der Reiter CEP-Rules, dient zum Setzen von verschiedenen Filterregeln mit denen sich die Usecases steuern lassen.

8.1.2 Die Steuerungselemente

Zunächst einmal gilt es zu erwähnen, dass es für die Steuerelemente nur ein beschränkter Zugriff erlaubt ist d.h. es gibt einen Administrator, welcher die Steuerung der Geräte

Abbild 8.4: CEP-Rules



übernimmt. Zum anderen gibt es Observer, welche lediglich zum Monitoring gedacht sind. Falls sich die Admin-GUI ausloggen sollte, ist es auch für einen Observer möglich sich als Admin-GUI anzumelden. Es ist nur ein Administrator pro System möglich, da es sonst zu widersprüchlichen Befehlen kommen würde (z.B. Start – Stop).

8.1.2.1 Nur Admin

Start Der Alienreader beginnt Tags einzulesen und schickt diese an die CEP - Schicht weiter.

Stop Der Alienreader stopt den Lesevorgang und schickt keine weiteren Tags an die CEP - Schicht.

Shutdown Der Alienreader beendet sich ordnungsgemäß und fährt herunter. Es ist nun ohne manuellen Start nicht möglich den Alienreader erneut zu starten.

Logout Der Administrator gibt die Rechte ab und eine andere GUI kann Adminrechte anfordern.

8.1.2.2 Nur Observer

Admin Falls es noch keinen Administrator gibt, ist es möglich den *Admin* – Button anzuklicken, welcher einem Zugriff auf die oben genannten Steuerungselemente gibt.

8.1.2.3 Alle Benutzergruppen

Exit Die Benutzeroberfläche schießt sich und beendet alle Verbindungen.

Reset All Leert alle Monitore. Dient zur Erhaltung der Übersicht.

8.2 Kommunikation

Damit die Benutzeroberfläche agieren kann und Daten erhält die sie anzeigen kann, muss sie über das Netzwerk mit den verschiedenen Schichten kommunizieren. Für die Kommunikation wurde JMS mit verschiedenen Topics bzw. Queues benutzt.

8.2.1 Kommunikation mit dem Model

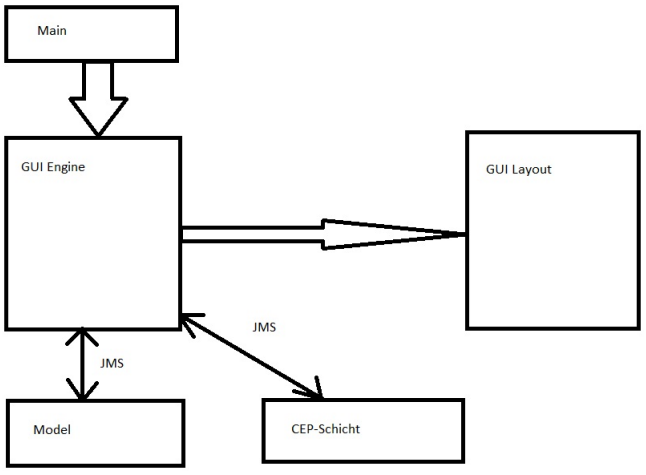
Die Kommunikation mit dem Model erfolgt über ein Topic. In dieses Topic schreibt die Benutzeroberfläche den jeweiligen Befehl der ausgeführt werden soll. Alle verfügbaren Befehle sind im Instruction-Enum definiert. Auch die Antworten vom Model erhält die GUI über dieses Topic wie z.B. die aktuellen Einstellungen des Readers.

8.2.2 Kommunikation mit der CEP-Schicht

Die Kommunikation mit der CEP-Schicht erfolgt über zwei Topics. Ein Topic ist zum setzten der verschiedenen CEP-Rules gedacht, das andere zum Empfangen der Events die von der CEP-Schicht, aufgrund der gesetzten Rules, generiert werden.

8.3 Architektur

Abbild 8.5: vereinfachte Architektur der Gui



Die GUI-Engine ist für die Organisation der eingehenden Daten verantwortlich. Wenn die Daten ausgewertet wurden und festgestellt wurde für welche Ansicht sie gedacht sind, werden die Daten an das GUI-Layout zur Darstellung weiter geschickt. Weiterhin beinhaltet die Engine alle Referenzen auf die Layoutobjekte um so die Daten bequem zu aktualisieren. Auch die Kommunikation mit den anderen Schichten wie Model und CEP wird über die Engine gehandhabt.

Das GUI-Layout erhält von der Engine alle Layoutobjekte und bestimmt die jeweilige Position und sonstige Layoutparameter wie Hintergrundfarben etc..

8.4 Evaluation

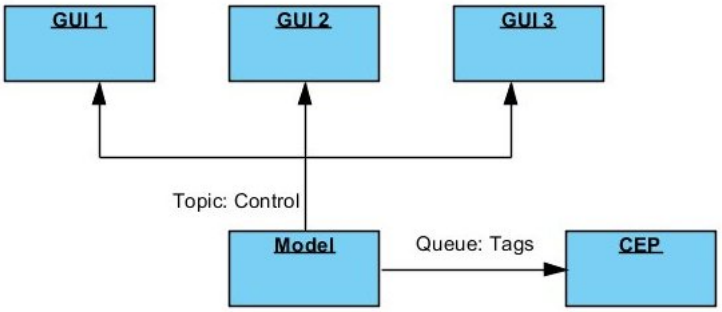
Wir haben uns für die Java Swing API entschieden, da sie im Gegensatz zu AWT modernen und schöner gestaltet ist und keine Lizenzschwierigkeiten verursacht wie andere kommerzielle Sparten.

Um eine möglichst gute Übersicht zu behalten, haben wir den Datenfluss von der Darstellung getrennt. Dies führte dazu, das wir eine Engine eingeführt haben, welche sich um den Datenfluss kümmert und zugleich die Layout-elemente mit den richtigen Daten beliefert. Auch die Fehlersuche wurde dadurch vereinfacht und beschleunigt, da wird Anhand des Kontextes feststellen konnten ob es sich um ein Problem mit der Darstellung handelt (z.B. keine Daten angezeigt) oder mit der Datenbelieferung (z.B. falsche Daten angezeigt).

9 Model(JMS)

9.1 Datenkommunikation

Abbild 9.1: Konzept der Kommunikationsstruktur des Models



Es gibt für die Kommunikation 2 Topics:

1.1.Das Control-Topic“ ist für die Kommunikation zwischen der GUI mit Admin-Rechten und dem Model zuständig. 1.2.Die Tag-Queue“ ist für das Senden der Tags an die CEP-Schicht zuständig.

9.1.1 Control-Topic

Der Property-Wert der Nachrichten, welche über Control-Topic übertragen werden, muss immer in folgender Form vorliegen. Der Name des Property Wertes ist property“, und der zugehörige String enthält den Namen einer Instruction Enum Konstante. Abhängig von dieser Instruction wird das mit der Nachricht übertragene Objekt geparst oder ignoriert. Zudem wird die jeweilige Programmlogik ausgeführt, welche in einem Switch-Case abgearbeitet werden kann.

Instructions

START `null`

STOP `null`

APPLYSETTINGS `TreeMap<SettingParameter, String[]>`

DEFAULT `null`

EXIT `null`

SHUTDOWN `null`

ADMINREQUEST `String`

9.1.2 Tag-Queue

Über die Tag-Queue werden die Tags als RfidTag[]übertragen. Das Model ist nur Producer der Tag-Queue, das heißt dass das Model lediglich Daten über die Queue sendet und keine empfängt.

Die Tags müssen von einem Tag Array in ein RFID-Tag Array umgewandelt werden, da ein Tag nicht serialisierbar ist, was jedoch eine Voraussetzung von JMS ist.

9.2 Admin-Konzept

Um sicherzustellen, dass immer nur eine GUI Änderungen am Model vornimmt, gibt es eine GUI mit Administratorrechten, während alle weiteren GUIs nur zu Monitoring-Zwecken benutzt werden können. Die Funktionen, welche nur die Admin-GUI nutzen kann, zählen:

- Stream starten
- Stream stoppen
- Einstellungen am Reader/Model ändern
- Model herunterfahren

Die Anmeldung als Admin-GUI wird folgendermaßen durchgeführt:

Abbild 9.2: Veranschaulichung des Admin-Konzepts

9.2.1 Fall GUI 1

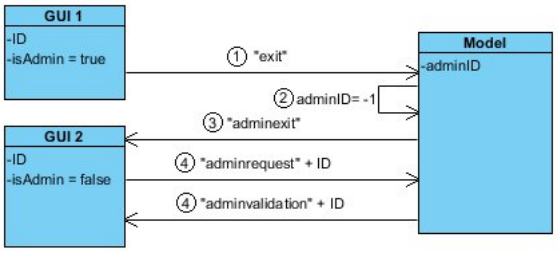
Als GUI 1 gestartet wird, ist noch keine andere GUI am Model registriert. Die GUI schickt an das Model den adminrequest“ mit ihrer ID. Model überprüft ob bereits eine GUI mit admin-rechten registriert ist. Da die GUI die erste ist, setzt model die ID als adminID und schickt adminvalidation“ mit der adminID zurück. Die gui vergleicht die beiden IDs, und aktiviert im Falle einer Übereinstimmung die Kontroll-Elemente.

9.2.2 Fall GUI 2

Als GUI 2 gestartet wird, besitzt GUI 1 bereits Administrations-Rechte. Sie schickt ebenfalls adminrequest“ mit ihrer ID an das Model, erhält jedoch als Antwort admindenied“ mit ihrer ID zurück, da bereits eine andere GUI als Administrator registriert ist. Stimmen die IDs überein, ist GUI 2 nur eine Montitoring-GUI.

9.2.3 Usecase: Admin-GUI wird beendet

Abbild 9.3: Admin-GUI wird beendet



1. Admin-GUI sendet exit an das Models
2. Das Model setzt die adminID zurück auf -1
3. Das Model schickt an alle GUIs (hier GUI 2) *adminexit*
4. Die GUIs (hier GUI 2) starten den Anmeldevorgang mit *adminrequest* erneut (siehe oben)

9.3 Instructions

Instructions (Enum) hält die Befehle, welche zur Steuerung des Models über die GUI zur Verfügung stehen. Die Instructions werden über das ControlTopic von der Admin-GUI an das Model gesendet, und im Model dann entsprechend abgearbeitet. Falls

benötigt schickt das Model ebenfalls angeforderte Daten über das ControlTopic an die GUI zurück.

START Startet den AlienReader, sodass die gelesenen Tags gesendet werden.

STOP Stoppt den AlienReader, sodass keine Tags mehr gesendet werden.

SETUP Fordert die aktuellen Settings an.

APPLYSETTINGS Weist das Model an, die übergebenen Settings anzuwenden.

DEFAULT Weist das Model an, die Default Werte zu setzen.

EXIT Informiert das Model darüber, dass sich die Admin-GUI beendet.

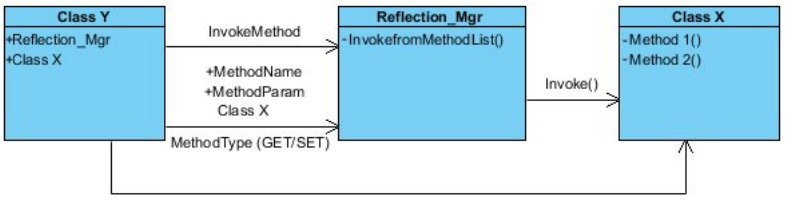
SHUTDOWN Befehl zum beenden des Models.

9.4 Reflection-Manager

Der ReflectionMgr bedient sich der Java internen Reflection API, um Methoden einer Klasse automatisch aufzurufen. Er benötigt hierfür eine Instanz der Klasse, in welcher die Methoden aufgerufen werden. Aus Performance-Gründen kann man festlegen, welche Methoden-Typen von der Klasse überprüft werden sollen (MethodType Enum). Dies können z.B. Get- oder Set-Methoden (MethodType.GET/MethodType.SET) sein.

Der ReflectionMgr fragt die gewünschten Methoden aus der Klasse ab und hält die Namen in einer Liste. Nun kann eine Methode anhand des Namens aufrufen, indem die InvokeMethodFromMethodList-Methode des ReflectionMgrs aufruft und ihr den Methodennamen sowie die entsprechenden Parameter als Object übergibt.

Abbild 9.4: Reflection-Manager



9.5 Reader

Die Reader-Klasse ist für das Setzen der Einstellungen auf der Hardware (AlienReader) zuständig. Sie benötigt eine Netzwerk-Adresse inklusive dem entsprechenden Port auf die die Hardware eingestellt ist. Der Reader hält das AlienClass1Reader-Objekt, welches alle Methoden für die Einstellungen enthält. Aufgrund der großen Anzahl der Methoden, verwenden wir den ReflectionMgr, welcher in der Lage ist die Methoden dynamisch aufzurufen. Dies versetzt uns in die Lage, die Einstellungen mit sehr wenig Änderungen zu erweitern.

10 Lichtschranken

Zur Ergänzung der vorhandenen Antennen-Hardware war die Anschaffung von Geräten zur Bestimmung der Bewegungsrichtung nötig. Hier boten sich natürlich die Lichtschranken an.

10.1 Die Hardware

Eingesetzt wurden zwei einfache Lichtschranken-Relais mit zugehörigem Reflektor. Die Lichtschranken wurden im Komplett-Set beim Ingenieurbüro für Zeitmessung (zeitmessanlagen.de) bestellt. Enthalten war die vorgefertigte Schaltung inkl. Gehäuse für die Lichtschranken, 2 Reflektoren, 2 Kabel, 1 Serieller Stecker, 1 Seriell-USB-Adapter.

Die Lichtschranken selbst sind eine einfache Schaltung. Eine UV-Diode ist an ein Relais geschaltet, dass bei Auslösen der Diode (= Reflektion des von der Diode ausgehenden Lichts) ein entsprechendes Signal weitergibt. Das Signal kommt über eines von 3 angeschlossenen Kabeln, die anderen beiden Kabel dienen zur Stromversorgung. Teilweise mussten die Lichtschranken noch montiert werden.

10.2 Montage

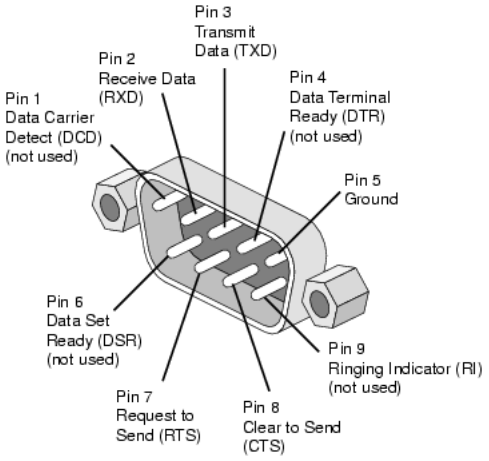
Für die Montage mussten die Lichtschranken noch mit den Kabeln an den seriellen Stecker angeschlossen werden. Die Lichtschranken selbst waren bereits an die Kabel montiert. Zum Anschließen konnten die Kabel an entsprechende Halterungen an der Rückseite des seriellen Steckers gelötet werden.

Pin 1 und Pin 6 (siehe Schema-Zeichnung unten) dienten als Datenleitungen für die Lichtschranken A und B. An diese Pins mussten die schwarzen Kabel von der Hardware kommend angelötet werden. Die braunen Kabel beider Lichtschranken waren die positiven Leiter. Sie mussten gemeinsam an Pin 4 angeschlossen werden, wo später Spannung anliegen soll. Dazugehörig mussten die blauen Kabel beider Lichtschranken (die negativen Leiter) angeschlossen werden, um den Stromkreis zu schließen. Die blauen Kabel kamen gemeinsam auf Pin 7.

Nach dem Löten musste das Gehäuse (einfache Plastik-Konstruktion) einfach nur noch um den seriellen Stecker geschlossen werden, die Kabel an der Rückseite des Gehäuses entsprechend gegen Zugkräfte gesichert, die die Isolierung entfernen könnten.

Durch entsprechend starkes Anziehen der Verschlussschrauben konnte man das Gehäuse sehr stabil machen.

Abbild 10.1: Pins des seriellen Anschlusses



10.3 Ansteuerung und Benutzung

Zur Ansteuerung der Schranken gibt es verschiedene Wege. Zunächst werde ich die allgemeine Vorgehensweise beschreiben, anschließend anhand von Beispiel-Code in JAVA erläutern.

Prinzipiell lässt sich die Funktion der Lichtschranken überprüfen, indem man an Pin 1 (DCD) Spannung anlegt und an Pin 6 (DSR) eben nicht. Da in den Lichtschranken eine gelbe Leuchtdiode verbaut ist, müsste diese Leuchtdiode nach entsprechender Konfiguration der Pins aufleuchten. Somit kann sichergestellt werden, dass der Stecker richtig verlötet wurde.

Das Setzen der beiden Spannungs-Pins ist bei jedem Programmstart notwendig.

Um nach dem Setzen auf das Geschlossen/Unterbrochen-Signal der Lichtschranken zu hören, müssen lediglich in genug hoher Frequenz die beiden Pins der Datenleitungen ausgelesen werden (Pin 4 – DTR und Pin 7 – RTS), um Änderungen an deren Zustand zu registrieren.

Durch das Auslesen der Pins erhält man einen boolschen Wert, also genau 1 Bit. Das Bit ist 0, also false, wenn die Lichtschranke geschlossen oder nicht unterbrochen ist. Das Bit

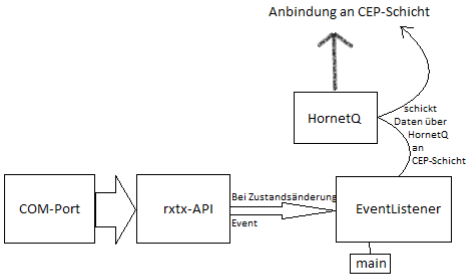
ist 1, sobald die Lichtschranke unterbrochen wurde (Im Gegensatz zu einer erwarteten, gegensätzlichen Annahme).

Das ist das grundlegende Prinzip, um die Lichtschranken an einem seriellen Port zu betreiben. Je nach Programmiersprache und Framework muss man natürlich vor dem Ansteuern der Pins den entsprechenden Port, an dem die Lichtschranke angeschlossen ist, auswählen.

Schließt man die Lichtschranke über den USB-Seriell-Adapter an, so ändert sich diese COM-Port (z.B. unter Linux normalerweise mit dem Namen `ttyUSB0`).

Die Implementierung in Java wählt folgenden Aufbau: Die rxt-API wird von einem

Abbild 10.2: Architektur der Lichtschrankenklassen

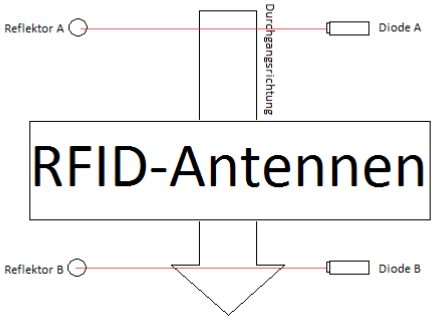


EventListener eingebunden und hört auf Zustandsänderungen. Die API selbst überwacht einen COM-Port. Der Eventlistener gibt bei triggern eines Events Daten über HornetQ-Topics an die CEP-Schicht weiter, wo je nach Anwendungsfall weiter verfahren wird.

Zum realen Aufbau muss man natürlich beachten, dass die Lichtschranken entlang einer Durchfahrts-Richtung aufgestellt werden und entsprechend Kontakt halten können.

Ein schematischer Aufbau:

Abbild 10.3: schematischer Aufbau des Lesefeldes mit Lichtschranken



Bibliography

[1] Various. War file format (sun). URL http://en.wikipedia.org/wiki/WAR_Sun_file_format.

[2] Oracle and/or its affiliates. The java ee 5 tutorial. URL http://docs.oracle.com/javaee/5/tutorial/doc/bnceh.html.

[3] IBM Corporation. Jms message selector. URL http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r1m0/index.jsp?topic=%2Fcom.ibm.etools.mft.doc%2Fac24876_.htm.