Erstellung eines Trainingssimulators für Radargeräte durch Restrukturierung, Modularisierung und Erweiterung einer OSGi-Anwendung

Marco Schäfer

30.1.2020

Inhaltsverzeichnis

1 Einleitung

1.1 Themenumfeld

Softwaresysteme sind heutzutage im ständigen Wandel und werden insbesondere, seitdem es die Möglichkeit gibt Updates und Patches über das Internet bereitzustellen, immer langlebiger. Wenn Quellcode eine lange Lebensdauer haben soll, ist es unabdingbar, dass dieser eine gute Wartbarkeit hat. Ansonsten können der Zeitaufwand und die Entwicklungskosten bis ins Unermessliche steigen. Diese Kosten entstehen, wenn Fehlerbehebungen am Code durchgeführt werden oder dieser um funktionelle Features erweitert wird. Ein wichtiges Konzept, um die Wartbarkeit von Softwaresystemsen zu verbessern ist eine Modularisierung der Softwarekomponenten.

Im Rahmen der Bachelorarbeit wird eine Anwendung mit einer modularen Architektur entworfen und implementiert. Um dies umzusetzen wird das Framework Equinox verwendet, welches die OSGi-Kernspezifikationen implementiert. „OSGi bietet ein Konzept zur Modularisierung und Komponentisierung von Java-Applikationen, und zwar nicht nur zur Entwicklungszeit, sondern auch zur Laufzeit“. Diese Applikation entsteht bei der Firma Thales Deutschland GmbH am Standort in Ditzingen in der Abteilung LAS (Land and Air Systems). In der Abteilung werden Radarsysteme für den zivilen, wie den militärischen Einsatz entwickelt. Die Abteilung ist in das Sensorteam und das Venusteam aufgeteilt. Das Venusteam ist für die Steuerungssoftware der Radarsensoren verantwortlich. Die Steuerungssoftware heißt Venus und bietet ein MMI (Man Machine Interface) zur Steuerung von verschiedenen Radargeräten. Zudem visualisiert die Venus Sensordetektionen auf einer Karte und liefert weitere relevante Sensorinformationen, wie z.B. Hardwarefehlermeldungen in Tabellen. Die Software läuft auf einem Laptop und ist über einen speziellen Anschluss mit dem Sensor verbunden. Zusätzlich entwickeln sie weitere Eclipse Applikationen zum Warten und Testen der Radarsensoren. Das Sensorteam entwickelt die Hardware und Software der verschiedenen Sensormodelle.

1.2 Aufgabenstellung

Das Sensorteam will einen Trainingssimulator erstellen, um neuen Benutzern die Funktionalitäten der Venus näher zu bringen. Bei der Trainingssimulation nehmen ein Trainer und ein bis mehrere Trainees teil. Jeder der beteiligten Personen verfügt über ein Laptop mit der entsprechenden Software. Während der Trainingssimulation, gibt der Trainer Simulationsdaten in die Benutzeroberfläche der Trainingssimulator Anwendung ein. Diese Informationen werden über eine Netzwerkschnittstelle an die Computer der Trainees übermittelt und dort in Echtzeit simuliert. Die Aufgabe des

Trainingssimulators ist es, dass die Trainees lernen, die simulierten Daten mit der Steuerungssoftware korrekt zu erfassen und richtig interpretieren. Ein detaillierterer Ablauf wird in Kapitel X.X beschrieben.

Das Ziel dieser Bachelorarbeit ist es, die beschriebene Trainingssimulator Anwendung zu implementieren. Um Wiederverwendbarkeit der Komponenten und eine einfache Erweiterung um funktionelle Features zu gewährleisten, steht eine modulare Architektur der Software im Vordergrund. Die Grundlage für diese Anwendung ist ein Radarsimulator, welcher einen realen Radarsensor simuliert. Dieser Radarsimulator wurde von früheren Praktikanten und Werkstudenten programmiert und wird derzeitig für interne Tests der Venus verwendet. Jedoch tendiert die Softwarearchitektur des Radarsimulators zu einer monolithischen Architektur, da einzelne Softwaremodule mehrere verschiedene Funktionen haben und wenig klare Schnittstellen zwischen den bereits vorhandenen Modulen definiert sind. Damit darauf sinnvoll aufgebaut werden kann, muss diese Architektur im Vorhinein überarbeitet werden. Zudem soll die Anwendung so angepasst werden, dass der Großteil der Softwarekomponenten in der Testanwendung und in der Trainingssimulator Anwendung verwendet werden können. Dazu muss eine entsprechende Architektur erstellt werden, die durch möglichst wenige Änderungen zu den verschiedenen Ausprägungen abgeleitet werden kann.

Eine weitere Aufgabe ist es, die Trainingsanwendung, sowie die Testanwendung und um drei Features ergänzt werden. Diese Features sind Single Target Tracking, die Bereitstellung des Dopplertons während STT und die Einbindung der Detektionswahrscheinlichkeit von Radarobjekten. Single Target Tracking ist die Funktion ein Radarziel über einen bestimmten Zeitraum zu verfolgen. Dabei wählt man ein bereits existierendes Ziel in der Venus aus und startet den STT Modus. Daraufhin schwenkt der Sensor mit einem kleinen Winkel über das Ziel und richtet sich bei Bewegung des Zieles neu aus. Diese Funktion wird vom Sensor bereitgestellt und nicht von der Venus. Deshalb muss der Simulator diese Funktion bereitstellen. Wenn das Ziel von STT erfasst wird soll nun der Dopplerton des verfolgten Objektes per Audio abgespielt werden. *Detektionswahrscheinlichkeit* Diese Features sollen eine realistischere Trainingsumgebung für neue Nutzer des Trainingssimulators bieten.

1.3 Aufbau der Thesis

Zu Beginn der Thesis werden die allgemeinen Grundlagen der Konzepte und Technologien, die in dieser Arbeit verwendet werden, erläutert. Als nächstes wird die alte Architektur des Radarsimulator analysiert und auf Schwachstellen überprüft. Zudem werden die funktionellen Anforderrungen der Trainingssimulator- und Testsimulatorandwendung aufgelistet und es wird untersucht an welchen Komponenten

die vorgegebenen Erweiterungen anknüpfen müssen. Nach der Analyse wird das Konzept der neuen Architektur vorgestellt. Ein besonderes Augenmerk wird dabei auf die Softwaremodule und deren Schnittstellen gelegt. Des Weiteren werden den verschiedenen Anwendungsausprägungen die Softwaremodule zugeordnet und genauer erklärt. Außerdem wird gezeigt, wie das Datenmodell verändert wurde und von den neuen Softwarekomponenten verwendet wird. Ergänzend wird darauf eingegangen, wie die Buildpipline die Testanwendung, die Traineranwendung und die Traineeanwendung baut. Im folgenden Kapitel wird erklärt, wie die Softwarekomponenten implementiert und zusammengesetzt wurden. Danach wird bewertet, wie der Status der Applikation am Ende des Projektes aussieht, bzw. ob alle wichtigen Funktionen enthalten sind und die Anwendung stabil läuft. Zuletzt gibt es noch eine Schlussbetrachtung der Thesis, in der Erkenntnisse der Arbeit beschrieben werden und wie es in der Zukunft mit den Technologien und der Anwendung weitergehen soll.

2 Analyse

2.1 Anforderungsanalyse der Ausprägungen/Anwendungen

2.1.1 Funktionale Anforderungen

In diesem Abschnitt werden die funktionalen Anforderungen der drei verschiedenen Anwendungen, die im Rahmen dieser Arbeit entstanden sind, vorgestellt. „Funktionale Anforderungen beschreiben gewünschte Funktionalitäten (was soll das System tun/können) eines Systems bzw. Produkts, dessen Daten oder Verhalten."

Die funktionalen Anforderungen des Trainersimulators sind:

- Der Trainersimulator muss dem Trainee ermöglichen sich mit dem Trainersimulator zu verbinden, wenn dieser sich im selben Netzwerk befindet und die korrekte IP-Adresse eingegeben wurde.
- Der Trainersimulator muss, bei Änderungen von PNU-Daten, dem hinzufügen von Bitfehlern oder dem Abspielen von Szenarien durch die graphische Oberfläche, die entsprechenden Informationen über das Netzwerk an alle verbundenen Trainees versenden.

Der Traineesimulator hat folgende funktionale Anforderungen:

- Der Traineesimulator soll sich mit dem Trainersimulator verbinden können.
- Der Trainingssimulator soll sich mit der Venus verbinden, nachdem dieser die nötigen Informationen vom Trainersimulator empfangen hat.
- Der Trainingssimulator muss, wenn er Simulations-Daten empfängt, diese verarbeiten und über das Asterix-Interface versenden
- Der Trainingssimulator muss, wenn er Model-Daten empfängt, die entsprechenden Informationen im Modell updaten und über das Asterix-Interface senden

Die funktionalen Anforderungen der Testanwendung sollen nicht von den funktionalen Anforderungen des Radar Simulators nicht abweichen. Diese lauten:

- Der Testsimulator soll eine Verbindung mit der Venus über das Asterix-Interface aufbauen.
- Der Testsimulator muss Informationen, die auf der graphischen Benutzeroberfläche verändert wurden über das Asterix-Interface versenden.

2.1.2 Nicht funktionale Anforderungen

„Nichtfunktionale Anforderungen sind Anforderungen, an die "Qualitätïn welcher die geforderte Funktionalität zu erbringen ist“. Die nicht funktionalen Anforderungen lassen sich für die drei Applikationen zusammenfassen, da diese fast vollständig übereinstimmen. Lediglich durch die Netzwerkverbindung der Trainer- und Traineeanwendung entstehen Unterschiede. Die nichtfunktionalen Anforderungen lassen sich in mehrere Bereiche unterteilen. Leistungsanforderungen, Qualitätsanforderungen und Randbedingungen.

„Unter Leistungsanforderungen versteht man im Allgemeinen, Anforderungen an die empirisch messbaren nicht-funktionalen Anforderungen eines Systems, also Anforderungen, deren zu Grunde liegendes Bedürfnis ein Leistungsmerkmal ist.“ [Braunausarbeitung]. Eine Leistungsanforderung jeder Anwendung ist, dass diese die Simulation in Echtzeit ausführt und eine Verzögerung um 10 ms im akzeptablen Bereich liegt. Zudem sollen diese nicht zu viel Speicher besetzen und die Prozessorbelastung sollte einen realistischen Wert betragen. Um dafür zu garantieren werden Belastungstests, mit der entsprechenden Hardware, auf der die Anwendung später laufen soll, durchgeführt.

Der „ISO/IEC 25000 Software engeneering Software product Qualitiy Requirements and Evaluation (SQUuarRE) ist ein aktueller Standard für die Qualitätskriterien und -bewertungen von Softwareprodukten. Der Software Product Quality Model aus dem Standard ISO/IEC 25010 beschreibt acht Kriterien, um die Qualität eines Produktes zu bewerten. Die acht Kriterien sind:

- Funktionalität
- Leistungseffizienz
- Kompatibilität
- Benutzbarkeit
- Zuverlässigkeit
- Sicherheit
- Wartbarkeit
- Übertragbarkeit

Die Qualitätsanforderung, die in dieser Arbeit im Mittelpunkt steht, ist die Wartbarkeit der Anwendungen. Wie bereits erwähnt soll die Arbeit zeigen, wie eine Anwendung modularisiert und erweitert werden kann. Diese Kriterien fallen unter den Oberbegriff Wartbarkeit. Vor allem die Wiederverwendbarkeit von Komponenten ist ein wichtiges Kriterium beim Erstellen der Anwendungen.

2.2 Softwarearchitektur des Radarsimulators

Im folgenden Abschnitt werden die einzelnen Softwarekomponenten der Radarsimulator Anwendung beschrieben und auf Wiederverwendbarkeit und Modularität überprüft. Dabei wird sich auf die Kernkomponenten der Anwendung beschränkt. Diese sind die SimulatorInstance, die View-Komponente und das Datenmodell. Der Radarsimulator wird von Thales zur Verfügung gestellt und wurde, vor und während dem Schreiben des Kapitels, nicht verändert. Der Radarsimulator mit neuer Architektur wird als Testsimulator bezeichnet.

Der Radarsimulator ist eine Eclipse RCP Anwendung und setzt sich aus mehreren Equinox Plug-In Modulen zusammen. Er wird durch Basis- und weiteren Plug-Ins der Venus unterstütz. Diese stellen spezifische Funktionen, wie Anwendungsstruktur und Helferklassen zur Verfügung.Der Startpunkt der Anwendung ist die Klasse SimulatorMain, welche sich in dem Modul RadarSimulator.app befindet. Dieses Modul stellt die Benutzeroberfläche des Simulators zur Verfügung. Im Modul werden die Startklasse der Anwendung und weitere Fenstereinstellungen für diese Anwendungen in einer Applikation.e4xmi-Datei angegeben. Diese Datei ist die Basis jeder Eclipse RCP Applikation. Wenn die Eclipse Applikation die Startklasse in Java aufruft, startet der RestartService. Diese ist im folgenden Codeabschnitt zu sehen.

```
public double getAzimuthStepSize() {
   return azimuthStepSize;
}

public boolean isClockwise() {
   return clockwise;
}
```

Die gesamte Funktionalität der Sensorsimulation stellt die SimulatorInstance zur Verfügung. Wenn der RestartService die SimulatorInstance erfolgreich geladen hat, initialisiert dieser die Benutzeroberfläche der Anwendung. Außerdem ermöglicht es der RestartService, die Simulation durch einen Mausklick in ein Fenster Menü item neu zu starten. Dabei werden die Benutzeroberfläche, sowie die SimulatorInstance neu geladen. Der RestartService soll zudem auch im zukünftigen Testsimulator sowie im Trainer- und Traineesimulator enthalten sein. Die SimulatorInstance und der SimulatorMainContentProvider werden in der SimulatorMain als Attribute gespeichert. Wenn die Benutzeroberfläche geladen wird, bekommt diese die SimulatorInstance übergeben, damit diese die Simulation steuern kann. Wie es bei einer Eclipse Applikation üblich ist sind Funktionen der Anwendung in Softwaremodule

eingeteilt. Diese Module heißen bei Eclipse Plug-Ins. In der folgenden Abbildung erkennt man die Aufteilung der Hauptkomponenten in zwei Plug-Ins.

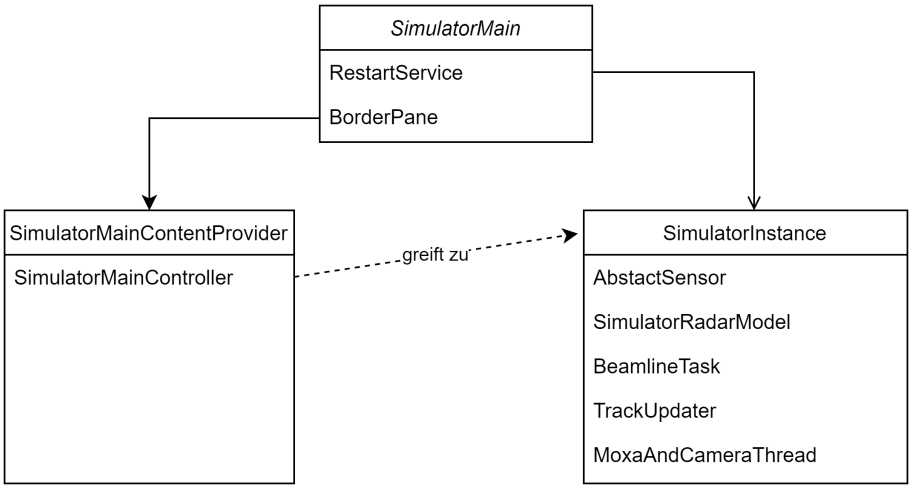

Abbildung 2.1: UML-Diagramm des Radarsimulators

2.2.1 Benutzeroberfläche

Das Frontend des RadarSimulators verwendet das Framework JavaFX und wird über den SimulatorMainContentProvider bereitgestellt. Der SimulatorMainContentProvider enthält verschiedene Einstellungsmöglichkeiten und stellt Klassen, die die Funktionalität des Benutzerinterface implementieren zur Verfügung. Diese Klassen heißen FXML Controller. Jeder FXML Controller ist mit einer FXML-Datei verknüpft, die die Anordnung der FXML Komponenten definiert. Der oberste FXML Controller des Radarsimulators ist der SimulatorMainController.

Die Benutzeroberfläche besteht aus mehreren einzelnen Tabs, in denen die Funktionen des Simulators in unterschiedliche Bereiche eingeteilt werden. Jeder Tab hat einen Haupt FXML Controller. Wenn der Controller einen zu großen Funktionsumfang hat, kann Funktionalität auf weitere FXML Controller aufgeteilt werden.

Da der SimulatorMainController der oberster FXML Controller ist, werden in ihm alle möglichen Tabs des Simulators vordefiniert und je nach Bedarf initialisiert. Welche Tabs geladen werden hängt davon ab, welcher Sensortyp und welche Asterixversion ausgewählt ist. Für jeden einzelnen Tab gibt es einen weiteren JavaFx Controller, in dem die Funktionalität der Oberfläche implementiert ist. Zwischen Backend und Frontend gibt es keine definierte Schnittstelle. Die Simulatorinstanz wird beim Erzeugen des Controllers an das Frontend übergeben. Das heißt die FXML

Controller haben immer vollen Zugriff auf die Simulatorinstanz. Das ist nicht optimal, da Abhängigkeiten schnell unübersichtlich werden können und die Wartbarkeit des Systems erhöhen.

Um das genauer zu erläutern wird das Beipiel aus dem Sequenzdiagramm ABBL. Bit genauer betrachtet. Auf der GUI-Oberfläche wird nun ein Bitfehler hinzugefügt. Daraufhin reagiert der entsprechende actionListener. Er fügt den Bitfehler dem Model hinzu und sendet diesen über das Asterix-Interface an die Venus. Wenn die GUI Komponente das Model und den Sensor kontrolliert hat diese keine klare Aufgabe definiert. Bei einer Modularen Architektur ist es wichtig, dass jede Komponente „eine klar abgegrenzte Aufgabe“[Quelle 2] erfüllt. Somit ist diese Architektur fehlerhaft. Inwiefern die Funktion der View-Komponente verändert werden muss damit diese die Anforderungen einer modularen Architektur erfüllt wird in Kapitel 4.X geschildert.

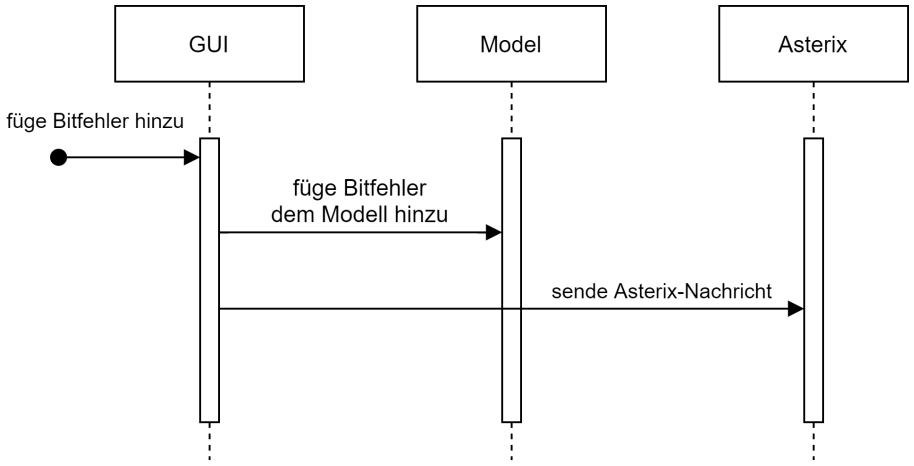

Abbildung 2.2: UI - Alter Simulator

Weitere Optimierung bedarf es bei der Initialisierung der Tabs. Die späteren Anwendungen haben unterschiedlich viele Tabs. Die genaue Übersicht gibt es in ABBL. X

Deshalb ist es entweder notwendig für jede Anwendung einen separaten Simulator-MainController zu konfigurieren. Daraus würde redundanter Code entstehen, den man so gut es geht, vermeiden möchte. Die Alternative wäre eine allgemeinen SimulatorMainController, welcher alle verfügbaren Tabs im OSGi-Kontext lädt. Um das zu implementieren können Extension Points verwendet werden.

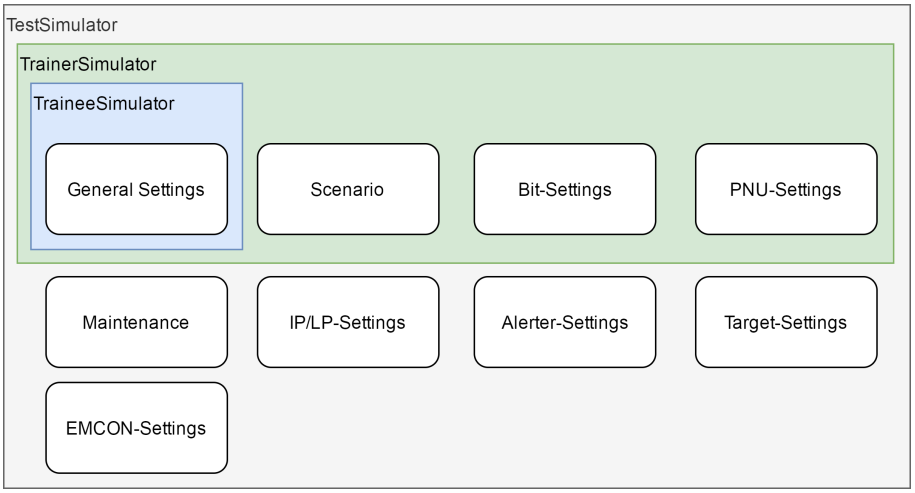

Abbildung 2.3: UI - Alter Simulator

2.2.2 SimulatorInstance

Die SimulatorInstance hat folgende Aufgaben:

- Erstellen des Datenmodells des Simulators. Aufgrund von unterschiedlichen Werten und Zubehör zwischen den Sensortypen, wird das Datenmodell je nach Sensortyp und Asterix-version entsprechend befüllt. Dafür wird eine Helferklasse verwendet.

- Erstellen eines Sensors, je nach Sensortyp wird ein passender Sensor initialisiert. Der AbstractSensor ist die Vaterklasse der jeweiligen Sensortypen (`SensorBORA`, SensorGO12, SensorGO80 und SensorGA10). Über den Sensor können Asterix-Kommandos an die Venus versendet werden

- Erstellen und starten des BeamlineTasks. Der BeamlineTask ist dafür zuständig, dass sich die Beamline des Sensors korrekt bewegt und die entsprechenden Werte im Modell geupdatet werden.

- Erstellen und starten des Trackupdater. Er ist dafür zuständig die Tracks korrekt upgedatet werden und über das Asterixinterface versendet werden.

- Erstellen und starten des MoaxaAndCameraServerTask. Dieser simuliert optionales Zubehör und die Kamera, falls dieser vorhanden ist.

Diese Tasks haben eine klar definierte Aufgabe und lassen sich gut wiederverwenden.

2.2.3 Model

Das Datenmodell des Radarsimulators wird mithilfe des Frameworks EMF erstellt. Das Modell befindet sich in einem eigenen Plug-In Modul. Darin sind ein graphisches Entity-Relation-Diagramm des Modells und der daraus generierbare Modellcode enthalten. Das Objekt, welches die Informationen des Sensors zusammenfasst, heißt SimulatorModelRadar. Dieses enthält 19 Attribute, die die Eigenschaften des simulierten Sensors beschreiben. Zusätzlich gibt es noch Beziehungen zu zehn Klassen, die die Daten des Sensors speichern.

Die Klassen, die in dieser Arbeit verwendet werden, sind:

- SimulatorBitNode: Der SimulatorBitNode ist eine Baumdatenstruktur in der Bitfehler abgelegt werden. Das SimulatorModelRadar besitzt höchstens einen SimulatorBitNode.
- SimulatorModelEquipment: Der SimulatorModelRadar kann beliebig viele SimulatorModelEquipment$\bar{O}$bjekte des Sensors gespeichern. Weil die PNU ein Zubehör des Sensors ist, werden dessen Informationen als SimulatorModelEquipment gespeichert. Dafür gibt es eine Klasse, welche SimulatorModelEquipment erweitert. Diese heißt SimulatorModelPNU.
- SimulatorModelSector: Um Sektor Informationen des Sensors zu speichern, gibt es die Klasse SimulatorModelSector. Die Klasse SimulatorModelRadar hat höchstens einen aktuellen Sektor und beliebig viele weitere Sektoren.

Was bei der Betrachtung des grafischen Modells auffällt ist, dass es drei vom SimulatorModelRadar unabhängige Klassen gibt. Diese Klassen sind:

- `Scenario`: Die Klasse Scenario beinhaltet beliebig viele ScenarioTargets, welche beliebig viele Waypoints haben. Man erkennt daran gut wie das Scenario aufgebaut ist. Die ScenarioTargets sind mögliche Objekt, die der Sensor detektieren kann. Das sind zum Beispiel Fußgänger oder PKWs. Diese Objekte bewegen sich auf einem Pfad, welcher durch die Waypoints (auf Deutsch Wegpunkte) definiert wird.
- `TargetSimulation`: Die TargetSimulation beschreibt die Eigenschaften, wie z.B. Position, Klassifizierung und Radarquerschnitt, eines Radarziels zu einem bestimmten Zeitpunkt. Eine TargetSimulation entsteht, wenn im UI ein Scenario abgespielt wird. Dabei wird die aktuelle Position eines ScenarioTargets, anhand der Wegpunkte berechnet und daraus wird eine TargetSimulation erstellt.

- SimulatorModelTrack: Ein SimulatorModelTrack beschreibt ebenso, wie die TargetSimulation, die Eigenschaften eines Radarziels. Der entscheidende Unterschied ist, dass dieses Radarziel nun abhängig vom Sensor ist. Deswegen hat dieser weitere Attribute, welche die Venus benötigt um das Ziel darzustellen. In Tabelle 2.1 werden die entscheidenden Unterschiede der beiden Klassen aufgezeigt.

	`TargetSimulation`	`SimulatorModelTrack`
Koordinatensystem	Lat/Long	Polarkoordinaten
Doppler Speed Informationen	-	x

Tabelle 2.1: Vergleich zwischen`TargetSimulation` und `SimulatorModelTrack`

Wie sinnvoll ist es nun diese Klassen vom Modell zu trennen? Das Scenario wird von der UI Komponente verwendet, um es auf einer Karte darzustellen und um es als XML-Datei zu speichern. Wie man in Abbildung 2.4 erkennen kann gehört das Scenario zum ScenarioController und lediglich der ScenarioUpdater ist vo ihm abhängig. Deswegen ist es nicht notwendig das Scenario im Modell zu speichern und es müssen keine Änderungen vorgenommen werden.

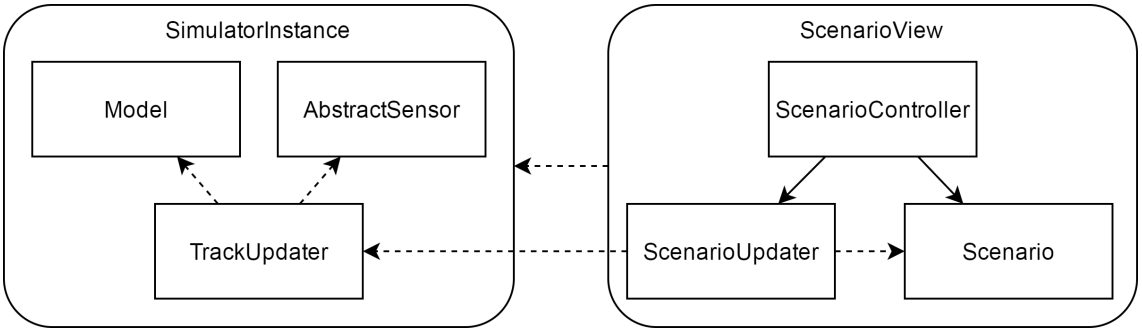

Abbildung 2.4: Beziehung zwischen SimulatorInstance und ScenarioView

TargetSimulation-Objekte werden nur vom ScenarioUpdater erzeugt. Dieser hat eine Liste mit ITargetUpdateListener, bei denen nach einem bestimmten Zeitintervall die Methode updateTarget(Targetsimulation) aufgerufen wird. Der TrackUpdater ist einer der ITargetUpdateListener. Die TargetSimulation-Objekte werden direkt durch den Methodenparameter an den TrackUpdater übergeben.

Eine modulare Architektur hat eine klare Schnittstelle. Das Problem ist aber, dass die SimulatorInstance keine geeignete Schnittstelle ist. Im Fall der neuen Anwendung, soll die Schnittstelle zwischen SimulatorInstance und ScenarioView nicht mehr die gesamte SimulatorInstance sein, sondern im Optimalfall nur noch das Modell. Wenn aber das Modell die einzige Schnittstelle zwischen SimulatorInstance und

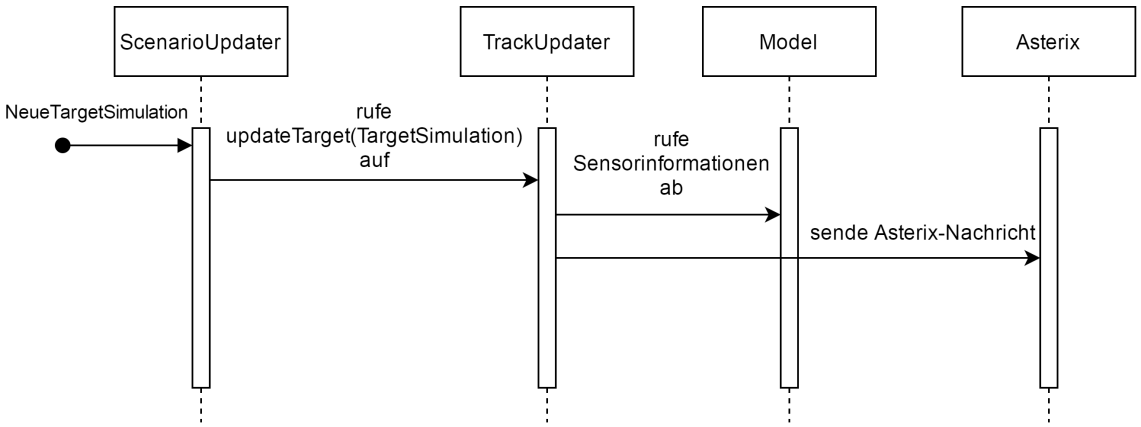

Abbildung 2.5: Beziehung zwischen SimulatorInstance und ScenarioView

ViewKomponente ist, hat der ScenarioUpdater keinen Zugriff mehr auf den TrackUpdater.

Wie ist es nun möglich, dass der ScenarioUpdater die Werte an den TrackUpdater übergeben kann, wenn er nur auf das Modell zugreifen kann? Er muss die TargetSimulation-Objekte im Modell speichern. Damit er diese im Modell speichern kann, muss die TargetSimulation Klasse eine Beziehung zum SimulatorModelRadar haben.

Im Falle, dass die Schnittstelle der ViewKomponente außer dem Modell noch einen ITargetUpdateListener beinhaltet, müssen die TargetSimulation-Objekte nicht im Modell gespeichert werden.

Der SimulatorModelTrack wird, wie das Scenario nur in einem Modul verwendet. In dem Fall ist das Modul die SimulatorInstance. Der SimulatorModelTrack wird im TrackUpdater mit Hilfe einer TargetSimulation erzeugt und in einer Liste gespeichert. In einem bestimmten Zeitintervall wird die Liste ausgewertet und Asterix-Nachrichten werden versendet. Es ist also auch nicht sinnvoll diese im Modell zu speichern.

3 Konzept

3.1 Neue Architektur

3.2 Aufbau der Ausprägungen

3.3 Features

3.3.1 Single Target Tracking

Um Single Target Tracking zu ermöglichen, benötigt die Anwendung eine Erweiterung um zwei Funktionen. Zum einen muss überprüft werden, ob sich ein Ziel im Suchbereich befindet, wenn dies zutrifft, soll nach jeder Bewegung des Ziels, das Audiogate auf dessen Position gesetzt werden. Des Weiteren muss sich die Beamline, je nach Zielverfolgung oder Zielsuche korrekt verhalten.

Während der STT Modus aktiviert ist, bewegt sich die Beamline innerhalb eines engen Sektors wiederholt über das Ziel. Wenn kein Ziel gefunden wird, sucht die Beamline in einem größeren Bereich nach einem neuen Ziel. Der Sektor hat das Audiogate als Mittelpunkt und ändert seinen Bereich synchron zum Audiogate. Wenn ein Ziel verfolgt wird beträgt die Breite des Sektors 100 mils. Falls der Sensor ein Ziel zur Verfolgung sucht, beträgt die Bewegungsweite 300 mils. Diese Funktion wird im BeamlineTask implementiert, da dieser Zugriff auf das Audiogate und die Beamlineinformationen hat. Der BeamlineTask hat jedoch keinen Zugriff auf die Daten der Tracks. Deshalb weiß er nicht, ob ein Ziel verfolgt wird und kann kein Audiogate setzen. Damit der Beamlinetask weiß, ob ein Ziel verfolgt wird, wird ein Booleanflag, welches jederzeit abgefragt werden kann, im Model hinzugefügt.

Das Flag, sowie das Audiogate sollen vom Trackupdater gestetzt werden. Dieser hat alle Informationen der verfügbaren Tracks, die in der Simulation existieren und kann berechnen, ob das Ziel derzeitig von der Beamline erfasst wird. Der TrackUpdater hat die Methode schedule(), welche wiederholt nach einem bestimmten Zeitintervall aufgerufen wird. Diese Methode aktualisiert die Tracks und sendet Track Informationen an den Sensor. Diese Methode wird nun erweitert, um das STTFlag und das Audiogate zu setzen. Ein verfolgter Track wird im TrackUpdater gespeichert.

Zum Anfang der Methode wird überprüft, ob bereits ein Track verfolgt wird, falls dies nicht der Fall ist wird die Distanz vom Audiogate zu allen vorhandenen Tracks berechnet. Die Werte werden in einer Liste mit dem jeweiligen Track gespeichert, wenn sie innerhalb des STT-Sektors liegen. Die STT-Funktion ist so definiert, dass sie den nächsten Track zum Audiogate innerhalb des Sektors so lange verfolgt, bis dieser verschwindet oder man das Audiogate manuell ändert. Deshalb wird die Liste so sortiert, damit der Track mit dem geringsten Abstand zum Audiogate an erster Stelle steht. Nachdem sortiert wurde, wird das erste Element gespeichert, das Flag wird auf true gesetzt und die Liste wird geleert. Wenn die Liste leer ist wird nichts getan.

Wird bereits ein Track verfolgt, wird geprüft, ob der zu verfolgende Track aktuell existiert. Das passiert indem man in der Liste, der gegenwärtigen Tracks nach einem

Track mit der identischen ID des gespeicherten Tracks ist sucht. Ist dieser vorhanden, wird dieser Track als neuer Track gespeichert. Als nächstes wird getestet, ob dieser Track vom Sensor detektiert wird. Trifft es zu, dass der Track vorhanden ist und detektiert wird setzt der TrackUpdater das Audiogate auf die aktuelle Position des Tracks. Wenn der Track nicht detektiert wird oder existiert, wird ein Zähler hochgezählt. Es kann passieren, dass ein Track in einem oder mehreren Durchläufen nicht im Bereich der Beamline liegt oder dass der Track vom Sensor nicht erfasst wird, da er z.B. zu weit entfernt ist. Durch den Zähler kann man einen Toleranzwert festlegen, wie oft der Track nicht erkannt werden muss bevor diesem nicht mehr gefolgt wird. Dem Track wird entfolgt, indem das Flag auf false gesetzt und der Track gelöscht wird.

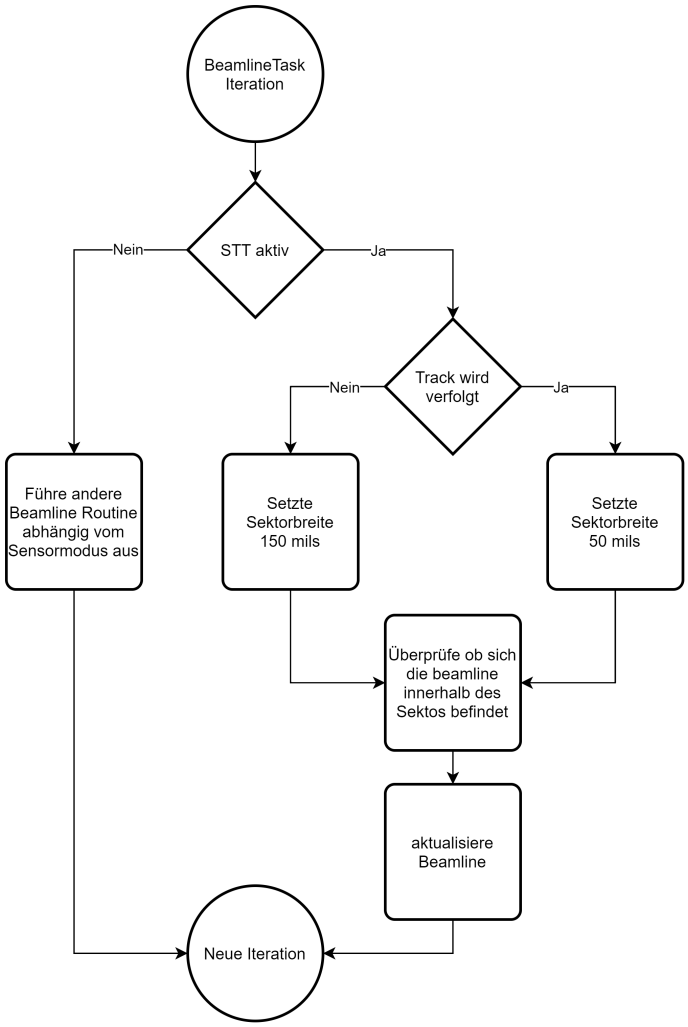

Abbildung 3.1: Flussdiagramm des Algorithmus des BeamlineTask

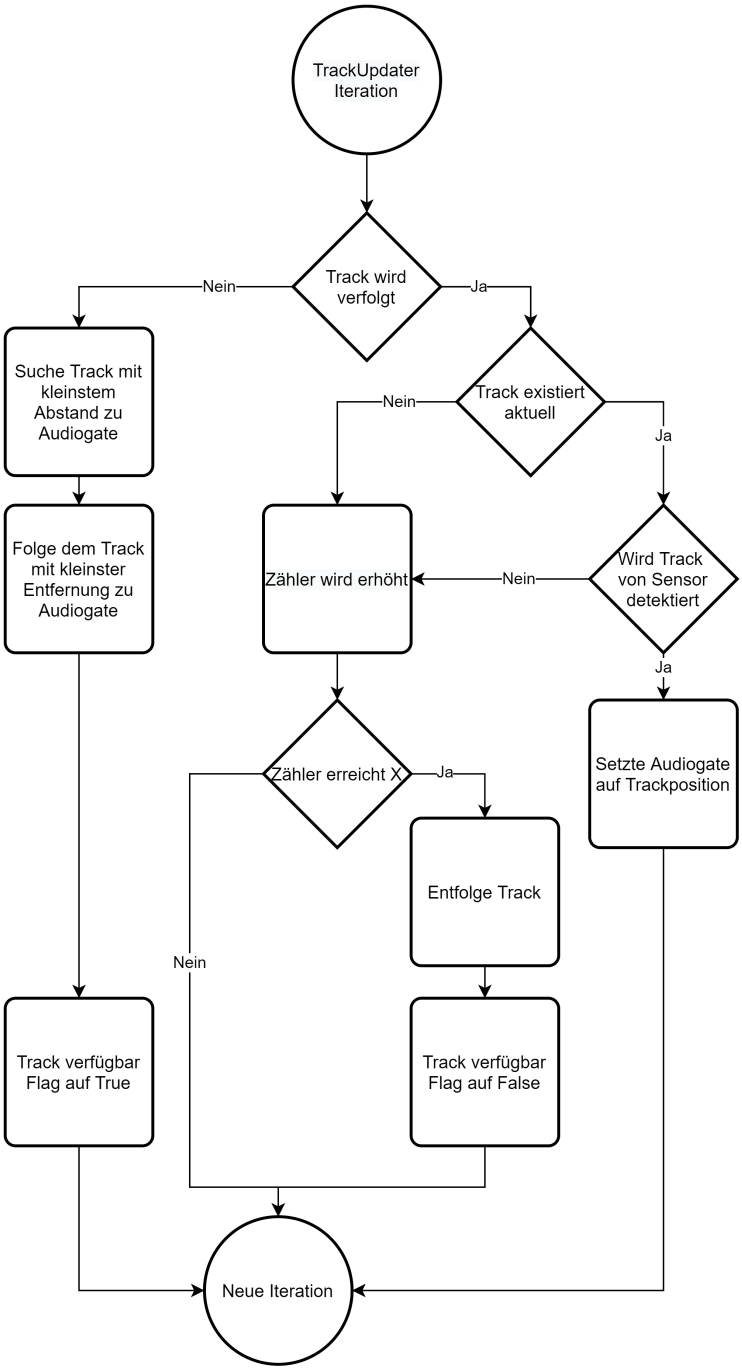

Abbildung 3.2: Flussdiagramm des Algorithmus des Trackupdaters