Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Bachelorarbeit
Im Studiengang Informatik

Entwicklung und Integration einer Lucene-basierten Suchfunktion zur interaktiven Selektion von Echtzeitdaten im maritimen Lagebildsystem iLEXX

von
Ammer Mechlaoui

Hochschule Bonn-Rhein-Sieg
Fachbereich Informatik
Grantham-Allee 20
53757 Sankt Augustin

Inhaltsverzeichnis

Abkürzungsverzeichnis

ASF ..... Apache Software Foundation

API ..... Application Programming Interfaces

BwDemo ..... Bundeswehrdemonstrationsraum

Fraunhofer-IAIS ..... Fraunhofer-Institut für intelligente Analyse- und Informationssysteme

GUI ..... Graphical User Interface

iLEXX-System ..... Informationsmanagementsystem zu LEXXWAR Seeraumüberwachungssystem

LEXXWAR ..... Long-term Experimental Setup for Asymmetric Warfare

RCP ..... Rich Client Platform

WTD71 ..... Wehrtechnische Dienststelle 71

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Unterschrift Sankt Augustin, den

Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Bachelorarbeit beigetragen haben.

Weiterhin danke ich Herrn Kai Pervölz, der mich mit all seinen Mitteln unterstützte und ohne dessen Hilfe und Bemühungen diese Arbeit nicht zustande gekommen wäre. Ich möchte außerdem den Mitarbeitern des IAIS Fraunhofer Instituts für ihre Unterstützung bedanken.

Mein besonderer Dank gilt meiner Familie, insbesondere meinen Eltern, die mir mein Studium ermöglicht und mich in all meinen Entscheidungen unterstützt haben.

Schließlich danke ich meinen Freunden die, mir während der Studienzeit geholfen haben.

1 Einführung

1.1 Motivation

Das Fraunhofer-Institut für Intelligente Analyse- und Informationssysteme (IAIS) betreibt seit mehreren Jahren auf dem Campus Schloss Birlinghoven in Sankt Augustin angewandte Forschung in den Bereichen Multisensordatenanalyse und Datenvisualisierung.

Im Rahmen einer mehrjährigen Kooperation zwischen dem Fraunhofer-IAIS und der Wehrtechnischen Dienststelle 71 (WTD71) wurde das Seeraumüberwachungssystem iLEXX entwickelt. Es soll den Benutzer auf auffällige Situationen hinweisen und ihm kontextabhängig alle notwendigen Handlungsoptionen zur weiteren Aufklärung der Situation oder der Abwehr einer Bedrohung aufzeigen.
Das iLEXX-System verarbeitet eine Vielzahl von Sensordaten und Ereignissen. Abhängig vom Szenario kommen hier mehrere tausend Updates pro Sekunde zusammen, die in Echtzeit vorverarbeitet und visualisiert werden müssen.

Die Visualisierung und Operatorinteraktion erfolgt mithilfe einer grafischen Benutzeroberfläche (iLEXX-GUI), welche als Desktopanwendung auf unterschiedlichen Betriebssystemen betrieben wird. Die iLEXX GUI stellt mittels verschiedener Daten eine detaillierte Übersicht des aktuellen maritimen Lagebildes dar. Die Informationen über Sperrzonen, Schiffsverkehr und schriftliche Anmerkungen der Operatoren, stellen den wichtigsten Anteil der visuellen Elemente dar (Abbildung 1).

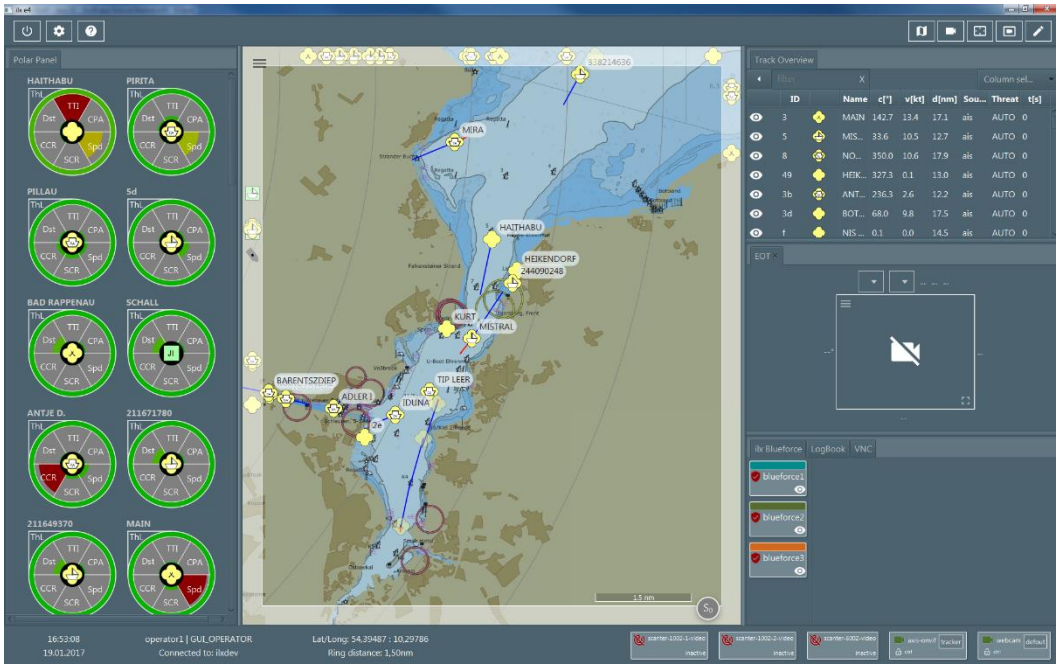

Abbildung 1: Ein maritimes Lagebildbeispiel der iLEXX-GUI

Trotz der umfassenden Vorverarbeitung der Daten wird auf der iLEXX-GUI eine Vielzahl von Informationen dargestellt, wodurch dem Operator die Beurteilung des Lagebildes erschwert wird. Aus diesem Grund wird die iLEXX-GUI im Rahmen dieser Arbeit um eine innovative Suchfunktion erweitert. Sie soll den Benutzer bei seiner Entscheidungsfindung unterstützen und ihm gleichzeitig eine übersichtlichere Datendarstellung anbieten.

Um die Suchfunktion zu realisieren, wird eine Suchbibliothek (search framework) verwendet. Bei der Auswahl einer geeigneten Suchbibliothek hebt sich Lucene vom

Wettbewerb deutlich ab. Lucene ist ein Produkt der Apache Software Foundation (ASF). Es birgt Vorteile, die für diese Arbeit relevant sind (Hardt / Theis 2004, S. 28):

- Java als gemeinsame Programmiersprache mit iLEXX
- Gebührenfreie Nutzung und langfristige Unterstützung von ASF
- Ausführliche Dokumentation
- Threadsichere Indexierung und Suche
- Schnelle Indexierung
- Geringer Speicherbedarf

1.2 Problemstellung

Die Realisierung einer Lucene-basierten Suchfunktion stellt die Hauptaufgabe dieser Arbeit dar. Dies wird erbracht, indem zwei Prozesse implementiert werden. Diese sind die Indexierung und die Durchsuchung. Folgende Abbildung veranschaulicht beide Prozesse:

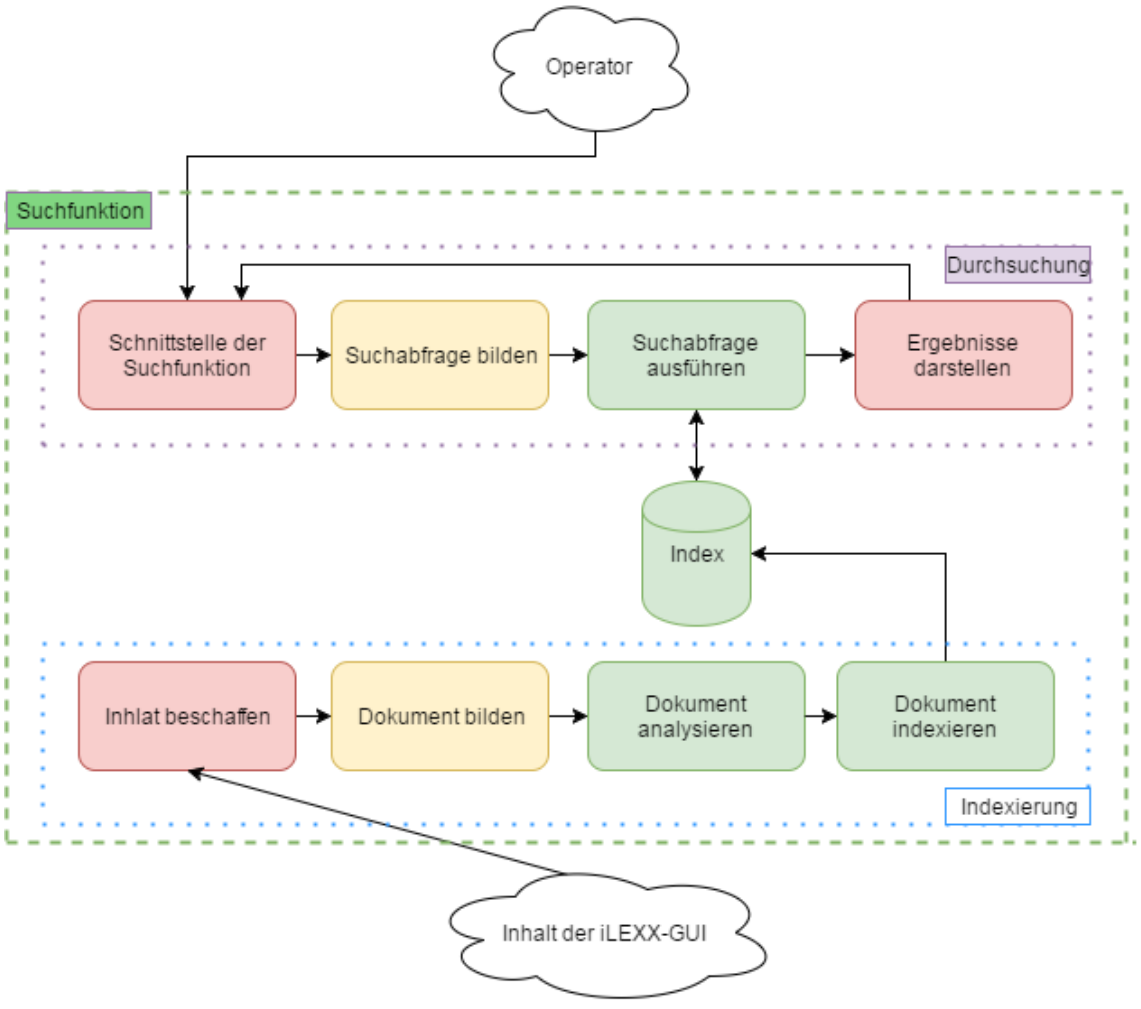

Abbildung 2: Die Struktur der geplanten Lucene-basierten Suchfunktion (nach: Gospodnetic / Hatcher / McCandless 2010, S. 10).

Die abgebildeten Ablaufschritte präsentieren einzelne Aktivitäten der oben genannten Prozesse. Die rot hinterlegten Ablaufschritte müssen von der Suchfunktion realisiert werden. Bei den Gelb hinterlegten, stellt Lucene APIs zur Verfügung. Die grün hinterlegten Ablaufschritte werden vollständig von Lucene erbracht.

Die iLEXX-GUI besteht aus mehreren Bereichen (Views). Jeder Bereich stellt Informationen auf unterschiedliche Weise dar. Während der Operator die GUI nutzt, wechselt ständig seine Aufmerksamkeit zwischen den geöffneten Bereichen. Durch die Integration und Nutzung der Suchfunktion, wird der Fokus des Operators auf die Suchfunktion gerichtet. Dieser Sachverhalt kann eine Veränderungsblindheit auslösen. Außerdem erfordert die Darstellung von Echtzeitdaten, sich wiederholende schnelle visuelle Änderungen. Dadurch besteht die Wahrscheinlichkeit, dass beim Operator ein Aufmerksamkeitsblinzeln vorkommt. Die Veränderungsblindheit und das Aufmerksamkeitsblinzeln können den Operator daran hindern, seine Aufgabenstellung zufriedenstellend zu erledigen. Andererseits ist der Operator, nach mehrjähriger Anwendung des Systems, mit den Arbeitsschritten und der Anordnung der iLEXX-GUI vertraut geworden. Er hat einen gewissen *Arbeitsfluss* entwickelt. Die Anforderung liegt also darin die Suchergebnisse in der iLEXX-GUI interaktiv zu selektieren, ohne die Handlung und den Arbeitsfluss des Operators zu stören.

Das angemessene Verhalten der Suchfunktion stellt eine weitere Herausforderung dar. Die zu durchsuchenden Echtzeitdaten besitzen eine hohe Änderungsfrequenz. Diese Daten werden permanent erzeugt, verändert und gelöscht. Die Suchergebnisse müssen stets aktuell gehalten werden. Suchergebnisse, die während der Selektion ungültig geworden sind, müssen in der Ergebnisdarstellung adäquat angepasst werden. Dies erfordert eine entsprechende Vielzahl an Operationen auf dem Index, die wiederum die Suchfunktion verlangsamt (Gospodnetic / Hatcher / McCandless 2010, S. 54).

1.3 Rahmenbedingung

Die Suchfunktion wird in der späteren Verwendung als eine Filterfunktion verwendet. Für die Filterfunktion werden die Mechanismen zur Indexierung und Durchsuchung von Informationen, ohne dabei eine Nutzerschnittstelle zu implementieren, realisiert. Unter einem Filter soll eine detaillierte Suchfunktion verstanden werden. Sie soll die Suche zukünftig nur auf eine begrenzte Informationsmenge anwenden und nicht auf den gesamten Index.

Eine weitere Bedingung an die Suchfunktion ist die Latenzzeit. Ab 100 Millisekunden fängt der Operator an, das Gefühl für die direkte Interaktion mit der Anwendung zu verlieren (Grigorik 2013, S. 174). Daher darf die Latenzzeit eines Suchvorganges (auf einem Rechner unter Windows 7 mit einem Leistungsindex von 7) die Zeitspanne von 100 Millisekunden nicht überschreiten.

Die iLEXX-GUI ist eine Rich Client Anwendung, die auf der Anwendungsplattform Eclipse 4 entwickelt wird. Die GUI besteht aus mehreren und untereinander unabhängigen Komponenten (Plug-Ins). Dies hat den Vorteil, dass jedes Plug-In entfernt oder durch ein anderes ersetzt werden kann, ohne die restlichen Funktionalitäten (Features) einzuschränken. Um dies zu erfüllen, muss die Suchfunktion als ein separates Plug-In in die iLEXX-GUI integriert werden.

1.4 Ziel der Arbeit

Wie in der Problemstellung erwähnt, soll die Suchfunktion dem Operator bei seiner Entscheidungsfindung unterstützen, sowie eine übersichtlichere Datendarstellung anbieten. Daher ist das primäre Ziel der Arbeit, diese Suchfunktion zu entwickeln.

Die Suchfunktion nahtlos zu integrieren stellt ein weiteres Ziel dar. Die Suchergebnisse sollen interaktiv präsentiert werden, sodass die Handlung des Operators ungestört bleibt. Das Ziel ist es dann, die Ergebnisse zu präsentieren, ohne die Handlung des Operators bei seiner Aufgabe zu beeinträchtigen.

Das Team, in welchem der Verfasser der vorliegenden Arbeit mitgearbeitet hat, setzt sich aus drei Experten zusammen. Der Projektleiter und zwei weitere Entwickler. Während einer der beiden Entwickler ein Backend-Experte ist, ist der andere auf Nutzeroberflächen (Frontend) spezialisiert. Diese Arbeit wurde unter Betreuung dieses Teams aufgeführt. Mindestens zweimal die Woche wird die Entwicklung der Suchfunktion in einem kurzen Meeting besprochen. Die Zusammensetzung des Teams und die wiederholten Besprechungen sorgen dafür, dass die Suchfunktion nach Wünschen des Kunden entwickelt wird. Zusätzlich wird die Funktion dadurch fachgerecht realisiert und dementsprechend auch die Ergebnisse auf der GUI dargestellt werden können.

2 Grundlagen

Die Entwicklung der Suchfunktion setzt ein tiefes Verständnis der Grundlagen von Lucene und iLEXX voraus. In diesem Abschnitt wird die Funktionsweise der Indexierung und der Durchsuchung mit Lucene erläutert. Abschließend wird die Struktur der iLEXX-GUI veranschaulicht.

2.1 Die Indexierung mit Lucene

Das Indexieren ist die Umwandlung der zu durchsuchenden Informationen, sodass sie schneller durchsucht werden können. Das Ergebnis der Indexierung ist der Index. Es ähnelt dem Index eines Buches, mit Hilfe dessen der Leser schneller auf die gewünschten Informationen zugreifen kann. Dieser Vorgang ist wesentlich schneller, als wenn der Leser das Buch sequenziell durchliest.

Die Indexierung ist sowohl für die Verarbeitung der Sucheingabe als auch für das Suchergebnis entscheidend, daher wird sie hier zuerst erläutert. Wie oben in der Abbildung 2 dargestellt ist, wird die Indexierung in Lucene in vier Arbeitsschritten realisiert. Im Folgenden werden diese Schritte erläutert.

2.1.1 Informationsbeschaffung

Die Informationsbeschaffung ist das Aufspüren von Informationen, damit sie im nächsten Schritt in Dokumenten extrahiert werden können. Lucene stellt keine Features zur Verfügung, die diesen Schritt implementieren (Gospodnetic / Hatcher / McCandless 2010, S. 67-68). Es ist dem Entwickler vorbehalten, wie die Informationen beschaffen werden sollen. Dabei kann ein sogenannter Crawler eingesetzt werden, alternativ kann dieses Feature individuell implementiert werden.

2.1.2 Erstellung der Dokumente

Aus den gewonnenen Informationen im vorherigen Schritt werden Dokumente erstellt. Dokumente sind die Eingangs- und Ausgangseinheit des Index. Lucene stellt keine fertigen Mechanismen zur Verfügung, welche die Informationen in Dokumente umwandeln können (Gospodnetic / Hatcher / McCandless 2010, S. 13). Lucene bietet andererseits die Möglichkeit Dokumente mit der Klasse „`Document`“ zu erstellen.

Jedes erstellte Dokument kann eine beliebige, voneinander unabhängige, Anzahl von Feldern besitzen. In jedem Feld ist wiederum ein *Feldname* und der dazugehörige *Wert* enthalten. Ein *Feldname* kann, im Falle einer Email-Indexierung, bspw. „Absender“ oder „Datum“ sein. Der *Wert* kann dann bspw. der Absendername oder das Sendedatum sein. Folgender Quellcode veranschaulicht, wie ein Dokument aufgebaut wird:

```
Document doc = new Document();
doc.add(new TextField("absender", „Johnny“, Store.YES));
doc.add(new StringField("datum", „06/07/16“, Store.YES));
```

Lucene erlaubt die Nutzung verschiedener Typen von Feldern. Folgendes UML-Klassendiagramm zeigt den Zusammenhang zwischen einem Dokument und den verschiedenen Feldtypen:

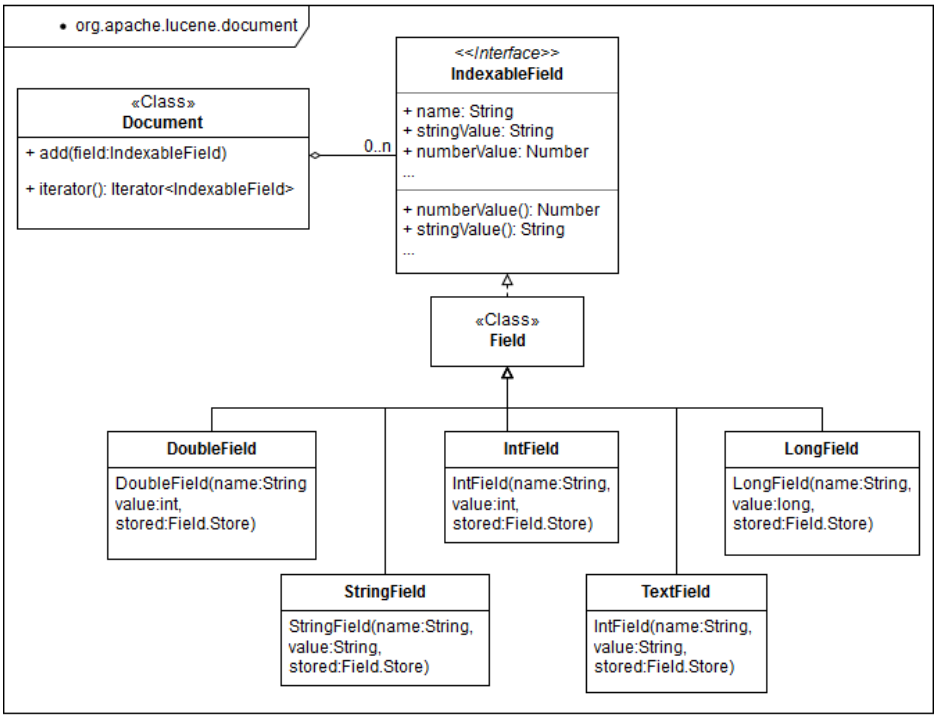

Abbildung 3: Ein UML Diagramm zur Veranschaulichung der Beziehung zwischen den Dokumenten und deren Feldern.

Die Felder werden genutzt um textuelle Informationen zu speichern. Diese können aus Zahlen, Buchstaben und/oder Zeichen bestehen. Beim Programmieren gibt es verschiedene Zahlentypen (z.B. `int`, `double`, etc.). Für jeden Zahlentyp ist ein Feldtyp vorgesehen. In der folgenden Abbildung sind die Feldtypen für die Zahlen „`DoubleField`“, „`IntField`“, „`LongField`“. Für Zeichen und/oder Buchstaben sind es die Feldtypen „`StringField`“ und „`TextField`“. Die Feldtypen unterscheiden sich dabei, auf welche Art und Weise ihre Werte bei der Analyse verarbeitet werden.

2.1.3 Analyse der Dokumente

Sobald die Dokumente erstellt und mit Feldern gefüllt wurden, müssen sie analysiert werden. Die Analyse der Dokumente wandelt den in den Feldern enthaltenen Text zu Termen (Token) um. Abhängig vom Nutzungskontext erfolgt die Analyse durch die Anwendung von Operationen auf den Text. Aus dem Text entstehen dann die Terme, indem Satzzeichen eliminiert, Leerzeichen wegfallen, Wörter zum Stammwort umgewandelt oder ganze Wörter entfernt werden. Diese Operationen reduzieren den Text, sodass Speicherplatz beim Indexieren und Zeit beim Durchsuchen gespart werden.

Die Klasse, die für die Analyse zuständig ist, wird „*`Analyzer`*“ genannt. Lucene stellt mehrere fertig implementierte „*`Analyzer`*“-Klassen zur Verfügung. Außerdem erlaubt Lucene die Implementierung eigener „*`Analyzer`*“-Klasse. Somit haben Entwickler die Möglichkeit, individuelle Anforderungen jeder Anwendung zu erfüllen.

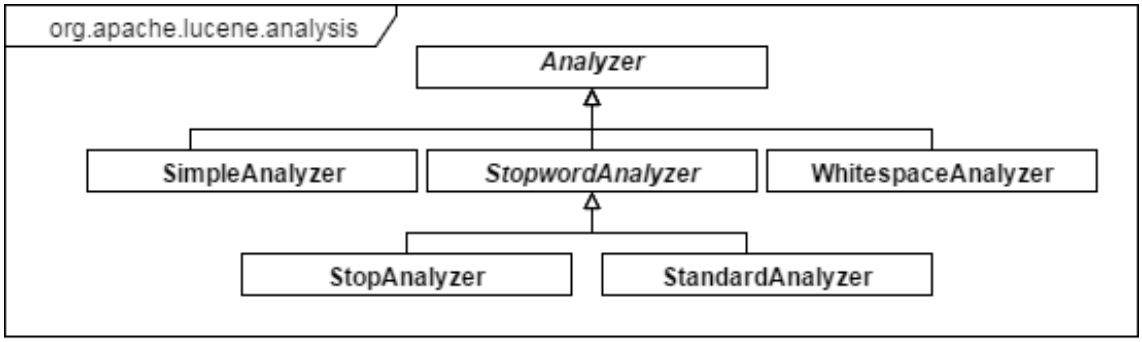

Abbildung 4: Von Lucene bereitgestellte Klassen zur Analyse der Dokumenten.

Mehrere Klassennamen in der Abbildung vermitteln welche Operationen mit der entsprechenden Klasse ausgeführt werden. Zur Veranschaulichung dieser Operationen und um den Unterschied zwischen den Klassen zu zeigen, wird ein Satz von unterschiedlichen Analyzern analysiert. Mittels eines Beispielprogramms wird der Satz „`This is a bad situation!, bedeutet keine guten Neuigkeiten.`" von den oben gebildeten Klassen analysiert. Nach der Analyse ergeben sich folgende Terme:

SimpleAnalyzer: `[this] [is] [a] [bad] [situation] [bedeutet] [keine] [guten] [neuigkeiten]`

StopAnalyzer: `[bad] [situation] [bedeutet] [keine] [guten] [neuigkeiten]`

StandardAnalyzer: `[bad] [situation] [bedeutet] [keine] [guten] [neuigkeiten]`

WhitespaceAnalyzer: `[This] [is] [a] [bad] [situation!,] [bedeutet] [keine] [guten] [Neuigkeiten.]`

Die Ausgaben des Programms zeigen eine große Veränderung des Satzes. Die deutschsprachigen Wörter wurden, abgesehen von der Eliminierung von Sonderzeichen, nicht verändert.

2.1.4 Indexierung der Dokumente

In diesem Schritt werden die Dokumente in einer Datenstruktur (Index) gesichert. Der Index dient dazu, Dokumente effektiv zu speichern, verarbeiten und durchsuchen zu können.

Der Index kann im Arbeitsspeicher oder im Dateisystem angelegt werden. Um Dokumente in den Index schreiben zu können, muss ein Objekt aus der Klasse „*IndexWriter*“ erstellt werden. Beim Erstellen dieses Objektes muss das Verzeichnis und der Analyzer an den Konstruktor übergeben werden. Danach können Dokumente mit „*addDocument*“ über das „*IndexWriter*“-Objekt in den Index eingetragen werden.

```
StandardAnalyzer analyzer = new StandardAnalyzer();
Directory index = new RAMDirectory();
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter writer = new IndexWriter(index, config);
Writer.addDocument(new Document());
```

Beim Indexieren werden Änderungen nicht sofort in den Index eingetragen, sondern zuerst gestapelt und in Segmente aufgeteilt.

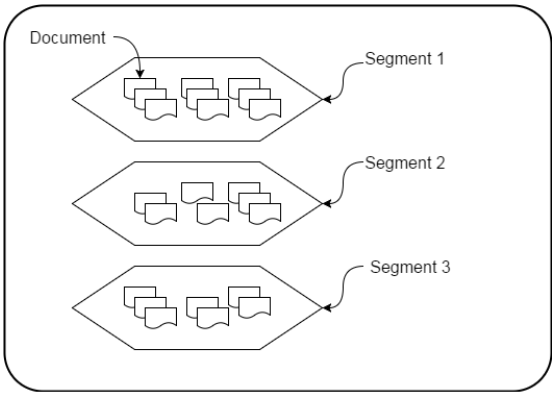

Abbildung 5: Die Struktur eines Lucene-Index (nach Gospodnetic / Hatcher / McCandless 2010, S. 436)

Die Segmente werden inkrementell erzeugt und müssen nicht zwingend eine identische Struktur aufweisen. Mehr über die Struktur der Segmente wird im Abschnitt 2.2.3 erläutert. Mehrere Segmente können zu einem einzigen Segment zusammengeführt werden. Dies geschieht mit dem Aufruf von „*commit*“ über das „*IndexWriter*“-Objekt. Dieser Aufruf kann explizit und implizit ausgeführt werden. Implizite Ausführungen können von anderen Methoden des „*IndexWriter*“-Objekts ausgelöst werden, wie beispielsweise „*flush*“.

2.2 Die Durchsuchung mit Lucene

Bei der Durchsuchung handelt es sich um das Aufspüren einer beliebigen Anzahl von Zeichenketten im Index unter bestimmten *Bedingungen*. Hierbei können die Bedingungen sehr unterschiedlich sein. Manchmal wird nur die hundertprozentige Übereinstimmung angefordert. In anderen Nutzungsszenarien wird zum Beispiel zwischen Groß- und Kleinschreibung nicht unterschieden.

Die Durchsuchung ist in der Regel ein Zyklusdurchlauf von vier Schritten, beginnend und endend bei der Nutzerschnittstelle (siehe Abbildung 2).

2.2.1 Nutzerschnittstelle

Die Suchanfrage fängt bei der Ermittlung der Nutzereingabe an. Es gibt viele Möglichkeiten die Nutzereingabe entgegenzunehmen. Manche Anwendungen bieten permanent die Möglichkeit die Sucheingabe entgegenzunehmen, andere erfordern eine Tastenkombination oder ein paar Mausklicks. Manche bearbeiten die Eingabe unmittelbar nach jeder Änderung und andere setzen eine Bestätigung voraus. In einigen Anwendungen stellt ein einziges Textfeld die Schnittstelle der Suche dar, während bei anderen Anwendungen ein komplettes Formular angeboten wird.

Lucene stellt keine Nutzerschnittstellen zur Verfügung (Gospodnetic / Hatcher / McCandless 2010, S. 14). Dieser Schritt ist der Anwendung überlassen. Wie die Sucheingabe in der iLEXX-GUI ermittelt wird, bestimmen die Anforderungen und die Rahmenbedingungen der Suchfunktion.

2.2.2 Erstellen der Suchabfrage

Sobald die Sucheingabe ermittelt wurde, muss diese in ein Abfrageobjekt („*Query*“-Objekt) formuliert werden. Die Bedingung (siehe 2.2) an die Suchfunktion und die Struktur des Index spielen hierbei eine große Rolle: Welche Dokumente müssen durchsucht werden? Müssen bestimmte Zeichen ignoriert, oder sollen reguläre Ausdrücke berücksichtigt werden?

Lucene bildet keine Mechanismen zur Umwandlung der Nutzereingaben in „*Query*“-Objekte. Dies ist anwendungsspezifisch und muss von den Entwicklern realisiert werden (Gospodnetic / Hatcher / McCandless 2010, S. 15). Lucene bietet aber bei diesem Schritt eine umfangreiche Auswahl von „*Query*“-Klassen, wobei jeder Typ Besonderheiten besitzt. Manche dienen der Suche nach Zahlen oder Texten. Andere verwenden die Nutzereingabe unbehandelt oder übersetzen die regulären Ausdrücke. Auch für die Suche nach Treffern in bestimmten Bereichen existieren passende „*Query*“-Klassen. Folgendes UML-Klassendiagramm spiegelt eine Übersicht über die meistverwendeten „*Query*“-Klassen wider:

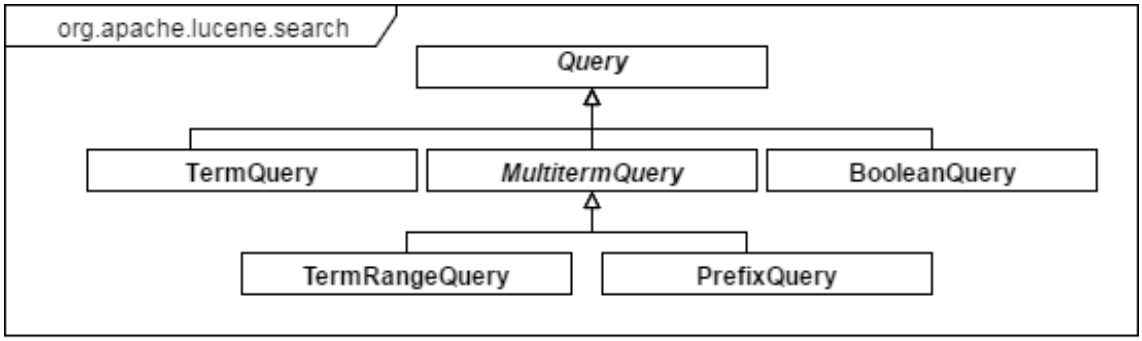

Abbildung 6: Klassen zur Erstellung von Abfrageobjekten

Es gibt mehrere Möglichkeiten eine Suchanfrage zu erstellen. Die einfachste Variante ist, eine der oben gebildeten „*Query*“-Klassen mit „*new*“ zu instanziieren und dem entstandenen „*Query*“-Objekt den gesuchten Text zu übergeben.

Die instanziierten „*Query*“-Objekte können auch kombiniert werden. Die Kombination erzielt eine bestimmte Verknüpfung von Treffermengen. Mit der passenden Zusammenstellung aus Konjunktion, Negation, Disjunktion oder Exklusion, kann jede Art der Suchanfrage mit Lucene gebildet werden.

Eine weitere Variante für die Erstellung der Suchanfrage bietet der „*org.apache.lucene.queryparser.classic.QueryParser*“. Ein Objekt dieser Klasse besitzt eine hohe Konfigurationsanpassung. Außerdem ermöglicht es die Nutzung von regulären Ausdrücken. Die Erzeugung eines „*Query*“-Objekts erfolgt mit dem Aufruf von „*parse*“ über das „*QueryParser*“-Objekt. Das entstandene „*Query*“-Objekt kann mit weiteren „*Query*“-Objekten kombiniert werden.

Sobald ein „*Query*“-Objekt fertiggestellt wurde, welches der Suchanfrage des Nutzers entspricht, kann dieses ausgeführt werden. Bei der darauf folgenden Ausführung der Suchabfrage, werden Dokumente mit bestimmten Kriterien aus dem Index gesammelt.

2.2.3 Ausführung der Suchabfrage

In diesem Abschnitt wird die Struktur des Index näher erläutert. Ein tiefes Verständnis über die Indexstruktur ermöglicht eine effiziente Nutzung des Index. Für das Verständnis wird im Folgenden der Index als ein einziges Segment behandelt.

Der Index besteht aus mehreren Dateien, unabhängig davon ob er in einem Verzeichnis oder im Arbeitsspeicher abgelegt wurde. Diese Dateien unterscheiden sich in erster Linie in ihrem Namen. Zur Veranschaulichung der Indexstruktur wird jede Datei als eine einzelne Tabelle dargestellt. Die Vorgehensweise, wie die Dokumente aufgespürt werden, wird mit Pfeilen angeführt. Daraus ergibt sich folgende Abbildung:

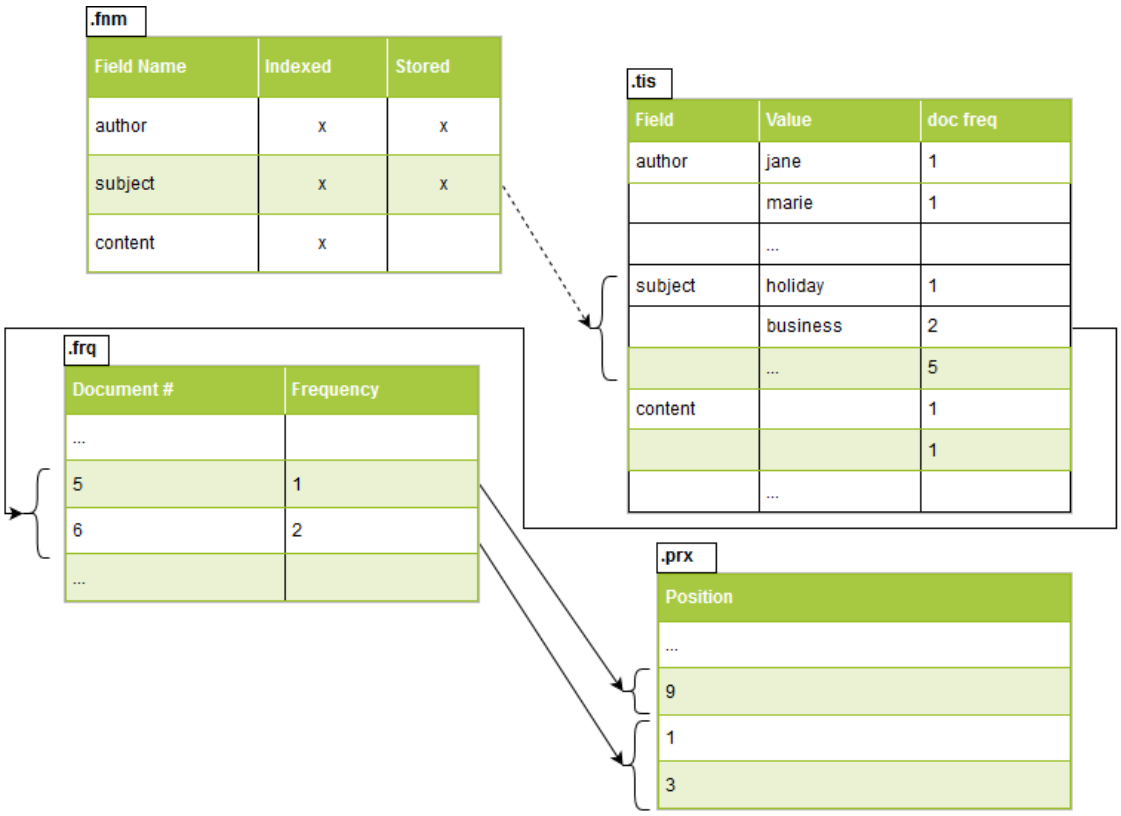

Abbildung 7: Dateien eines Lucene-Index (nach Gospodnetic / Hatcher / McCandless 2010, S. 440)

Die Abbildung zeigt eine mögliche Indexstruktur bei der Indexierung von E-Mails. In dem Beispiel wird ein Dokument für jede eingegangene E-Mail indexiert. Jedes Dokument besitzt Felder für den Namen des Autors, den Betreff und den Inhalt. Ein „*Query*“-Objekt durchsucht den Index nach E-Mails mit dem Betreff „*business*“. Die Pfeile zeigen die Vorgehensweise, wie der Index bei der Durchsuchung gelesen wird. Zuerst wird geprüft, ob in der Datei „*.fnm*“ das Feld „*subject*“ vorkommt. Dann werden alle Felder mit dem entsprechenden Namen nach dem Wort „*business*“ in der Datei „*.tis*“ durchsucht. In dem Beispiel enthalten zwei Dokumente das Wort „Business“. Die Datei „*.frq*“ gibt an, wie oft ein Wort in einem Dokument vorkommt. In welcher Position das Wort vorkommt, zeigt das dann die Datei „*.prx*“.

Die Ausführung einer Suchabfrage wird folgendermaßen durchgeführt. Ein Objekt aus der Klasse „*org.apache.lucene.index.DirectoryReader*“ wird erstellt bzw. geöffnet. Bei der Öffnung bzw. Erstellung wird intern die Methode „`commit`()“ aufgerufen und alle Segmente im Index werden zusammengeführt (siehe Abschnitt 2.1.4). Das Lesen des Index von mehreren Threads kann zu Race Conditions Probleme führen, daher greift das Reader-Objekt nicht direkt auf den Index zu. Der Reader erstellt stattdessen ein Bild, welches den Index zum Zeitpunkt, wo der Reader geöffnet wurde, widerspiegelt. Um ein „*Query*“-Objekt auf den Reader auszuführen wird eine Instanz aus der Klasse „*IndexSearcher*“ benötigt. Ein Methodenaufruf

„*search()*“ über diese Instanz sammelt alle Dokumente im Reader, die die Abfrage erfüllen. Das Ergebnis ist eine Liste von Dokumenten.

Spätere Änderungen nach dem Öffnen eines Readers kann der „*IndexSearcher*“ nicht mehr berücksichtigen. Damit Änderungen vom „*IndexSearcher*“ bei der Ausführung eines „*Query*“-Objekts mit einbezogen werden, muss ein neuer „*Reader*“ erstellt bzw. geöffnet werden. Das Öffnen eines *Readers* kann Ressourcenaufwändig sein, insbesondere bei Anwendungen, die große Änderungen an dem Index vornehmen und gleichzeitig diese Änderungen in den Suchergebnissen aufnehmen müssen. Um sparsamer mit den Ressourcen umzugehen, besteht die Möglichkeit mit „*DirectoryReader.reopen()*“ den gleichen *Reader* neu zu öffnen. Somit wird nur dann ein *Reader* neu geöffnet, wenn tatsächlich Änderungen an dem Index vorgenommen wurden.

2.2.4 Ergebnisdarstellung

Die Ausführung der Suchabfrage ergab eine Dokumentenauflistung. Aus dieser Auflistung müssen Informationen extrahiert und angemessen dargestellt werden. Da jede Anwendung eine individuelle Darstellung der Suchergebnisse verlangt, muss dieser letzte Schritt von der Suchfunktion implementiert werden.

Oftmals werden Ergebnisse durch Listen präsentiert, wobei die Ergebnisse nach bestimmten Kriterien sortiert werden. Diese Kriterien werden abhängig vom Zweck der Anwendung und dem Interesse des Nutzers bestimmt. Im Falle einer Ergebnisaktualisierung müssen je nachdem alle Schritte ab 2.2.1 oder ab 2.2.3 ausgeführt werden.

2.3 Einführung in die iLEXX-GUI

In diesem Unterkapitel werden, für diese Arbeit die relevanten Grundlagen der iLEXX-GUI erläutert. Zuerst erfolgt eine kurze Vorstellung mit ein paar Bedienungsinformationen. Anschließend werden OSGi-Services vorgestellt und einen vorteilhaften Aspekt von JavaFX näher betrachtet.

2.3.1 Vorstellung der iLEXX-GUI

In dieser Arbeit wird die entwickelte Suchfunktion in die iLEXX-GUI integriert. Daher erfolgt in diesem Abschnitt eine Vorstellung dieser Nutzeroberfläche. Die iLEXX-GUI besteht aus mehreren Bereichen, die von dem Operator beliebig angeordnet werden können. Die Struktur der Bereiche hängt auch von der ausgewählten Perspektive ab. In diesem Abschnitt wird die iLEXX-GUI so beschrieben, wie in der folgenden Abbildung zu sehen ist.

Abbildung 8: Die iLEXX-GUI in der Standard-Perspektive

In der Mitte befindet sich die ***Tactical Display Area*** (TDA) und nimmt den größten Teil der iLEXX-GUI ein. Darin wird eine Landkarte mit vielen wichtigen Daten eingeblendet. Die für diese Arbeit relevanten Daten sind *Sperrzonen* und *Tracks*.

Sperrzonen sind in der Regel die rot markierten Gebiete auf der TDA. Sie kennzeichnen bspw. militärische Sicherheitszonen, in denen Übungen oder Einsätze durchgeführt werden können. Diese Gebiete sind für fremde Objekte gesperrt. Falls solche Objekte diese Zonen kreuzen oder sich darin aufhalten, erhalten Operatoren eine Alarmierung auf der iLEXX-GUI.

Besagte Objekte können beispielsweise Schiffe, Boote oder Taucher sein und werden durch Tracks dargestellt. Tracks sind Aufzeichnungen über Positions- und Bewegungsdaten dieser Objekte.

Informationen über Daten, unteranderem Tracks, sind in eine Vielzahl von Eigenschaften unterteilt. Die Eigenschaften können statische und dynamische Werte beinhalten. Im Folgenden werden einige Eigenschaften eines Tracks erläutert:

- **Name** bzw. Alias Name: Der Name wird automatisch durch das iLEXX-System bzw. vom Operator vergeben.
- **Koordinaten**: Diese Angaben liegen in Längen und Breitengraden vor. Sie geben die Ortsposition an und werden in periodischen Zeitabständen dynamisch aktualisiert.
- **Threat Level**: Das Bedrohungspotenzial eines Tracks wird in Stufen kategorisiert. Mögliche Bedrohungsstufen sind: LOW, NEUTRAL, ELEVATED, HIGH.
- **Affiliation**: Bei der Affiliation handelt es sich um eine von der NATO standardisierte Kategorisierung von Zielen.
- **Course**: Der Kurs gibt die aktuelle Bewegungsrichtung über Grund eines Tracks an. Dieser Messwert wird in Grad angegeben.

- **Speed**: Diese Eigenschaft gibt die aktuelle Geschwindigkeit des Tracks an. Dieser Messwert wird in Knoten (kt) angegeben.
- **Bearing**: Gibt die aktuelle Peilung eines Tracks an. Dieser Messwert wird in Grad angegeben.

Rechts oben ist die ***Track-Tabelle*** angegliedert. Sie ist eine Auflistung aller Tracks, die im System vorhanden sind. Jede Zeile stellt ein Track dar, wobei jede Spalte den Wert einer bestimmten Eigenschaft anzeigt. Der Benutzer hat die Möglichkeit diese Spalten nach seinen Vorstellungen einzustellen. Mit dem Anklicken einer ausgewählten Spalte, können Tracks nach einer bestimmten Eigenschaft sortiert bzw. gruppiert werden.

Eine weitere Darstellung aller Tracks im System bietet das ***Polarpanel-View***. Dabei ist die Threat-Level-Eigenschaft der Fokus dieses Bereichs. Das Bedrohungspotenzial wird anhand verschiedener Faktoren mit einer von der WTD71 festgelegten Gleichung berechnet. Das Polarpanel sortiert die Tracks nach ihrem Bedrohungspotential und zeigt die Stärke einzelner Faktoren an.

Unter der Track-Tabelle existieren mehrere Bereiche die als Registerkarten übereinander gegliedert sind. Die relevante Registerkarte für diese Arbeit ist die ***Operator-Log-View***. Dieser Reiter wird dazu genutzt, um Auffälligkeiten festzuhalten und an andere Operatoren zu verschicken. Operatoren können in den Nachrichten mit der Eingabe von „*#S_<SafeZone Id>*“ auf eine Sperrzone, oder mit „*#T_<TrackId>*“ auf einen Track verweisen. Die *SafeZoneId* und *TrackId* sind die entsprechenden Identifikationsnummern.

Eine weitere Möglichkeit bei der Erstellung von Verweisen bietet das ***Kontextmenü*** (context menu). Das Menü ist mit einem Rechtsklick auf das entsprechende Objekt auf der TDA aufrufbar. Der Befehl „*write Log*“ generiert automatisch einen Verweis auf das ausgewählte Objekt. Das Kontextmenü eines Objekts ist aus jedem Bereich, der Informationen zu diesem Objekt bereitstellt, aufrufbar.

Folgende Optionen im Kontextmenu sind für diese Arbeit relevant:

• Set Affiliation: zur Bestimmung der Zugehörigkeit (track affiliation).

• Set Threat Level: zur Kategorisierung des Bedrohungspotenzials(threat level).

• Center on map: setzt ein Track in die Mitte der TDA.

2.3.2 Entwicklung mit OSGi Services

Die iLEXX-GUI besteht aus mehreren und untereinander unabhängigen Plug-Ins. Jedes Plug-In kann entfernt werden, ohne die restlichen Features einzuschränken. Ein Feature kann realisiert werden, indem ein OSGi-Service selbst definiert wird.

Dies geschieht, indem ein Service aus zwei Teilen gebildet wird. Zum einen aus der Service-Schnittstelle, die den Service deklariert, zum anderen aus einem Service-Provider, der die Schnittstelle implementiert. Hierbei können beliebig viele Service-Provider implementiert werden (Teufel / Helming 2012, S.174). Mithilfe von Services kann die Umkehrung der Abhängigkeit (dependency injection) realisiert werden. Dies erspart die Verwendung von Singleton oder Factory Patterns in der eigenen Anwendung (Vogel 2015, S. 185). Weiterhin können die Schnittstelle und der Provider eines Services in verschiedene Plug-Ins aufgeteilt werden. Die Aufteilung birgt den Vorteil, dass jedes Plug-In entfernt oder durch ein anderes ersetzt werden kann, ohne die restlichen Features einzuschränken.

Nachdem die Service-Schnittstelle definiert und der Service-Provider implementiert wurde, werden beide Teile deklarativ mithilfe der Service Components zur Verfügung gestellt. Der Service-Provider wird dadurch für die anderen Module bereitgestellt und

kann von dort injiziert werden. Das Injizieren ist die indirekte Erzeugung einer Instanz zur Laufzeit, während die Module nur die Service-Schnittstelle kennen.

2.3.3 Entwicklung mit JavaFX

Die visuelle Darstellung der iLEXX-GUI wurde mit JavaFX realisiert. JavaFX ist eine Java-Spezifikation und liefert vielfältige Werkzeuge zur Gestaltung von Nutzeroberflächen. Eins von vielen Vorteilen, ist die Möglichkeit der Bildung von Nutzeroberflächen mittels XML. Hierfür wird eine FXML-Datei angelegt und darin alle UI-Komponenten deklariert. Die Datei wird anschließend mit einer Klasse veröffentlicht. Danach können die UI-Komponenten injizieren und anschließend verarbeiten.

3 Konzeption

In diesem Kapitel wird die Suchfunktion konzipiert. Zuerst werden die Anforderungen an die Suchfunktion analysiert. Anschließend wird ein Grobentwurf, gefolgt von einem Feinentwurf erstellt. Der Grobentwurf veranschaulicht die Struktur der Suchfunktion, der Feinentwurf konkretisiert die Struktur der Suchfunktion.

3.1 Anforderungsanalyse

> „Requirements Engineering im heutigen Sinne ist ein kooperativer, iterativer und inkrementeller Prozess mit folgenden drei Zielen:
>
> - Alle relevanten Anforderungen sollen bekannt und im erforderlichen Detaillierungsgrad verstanden werden.
> - Die involvierten Personen und Organisationen (Stakeholder) sollen ausreichende Übereinstimmung über die bekannten Anforderungen erzielen.
> - Die Anforderungen sollen konform zu den Dokumentationsvorschriften der Organisation spezifiziert sein.“ (Hruschka 2014, S. 8).

Um die Anforderungen zu kennen werden in diesem Unterkapitel *Nutzungsszenarien* formuliert. Mit den Szenarien kann eine beliebige Tiefe der Detaillierung erreicht werden. Sie sind auch für das zweite Ziel wichtig. Mit ihnen wird die Übereinstimmung der Stakeholder erlangt. Anschließend wird die Spezifikation von Anforderungen durch *Anwendungsfälle* (use cases) dokumentiert.

Zur Veranschaulichung werden alle wichtigen Anforderungen im Abschnitt 3.1.3 zusammengefasst. Zum Schluss werden die Anforderungen in die Filterfunktion vorgestellt.

3.1.1 Nutzungsszenarien

Nutzungsszenarien beschreiben einen erfolgreichen Ablauf eines Nutzungsbeispiels der Suchfunktion(Kahlbrandt 2013, S. 227). Dabei werden konkrete Informationen verwendet und eine realitätsnahe Nutzung der Suchfunktion simuliert.

Nach Tracks suchen

- **Szenario 1:**

Das Schiff „Madelene2000“ schweift von der geplanten Route ab. Ein Operator 1 bemerkt die Kursänderung und teilt sie den anderen Operatoren mit. Operator 2 fokussiert sich auf das Schiff und sucht nach weiteren Auffälligkeiten. Damit er zunächst das Schiff findet, gibt er den Namen „Madelene2000“ in die Suchfunktion ein. Das Schiff wird selektiert und Operator 2 kann es mit dem Kontextmenü in die Mitte der iLEXX-GUI setzten.

- **Szenario 2:**

Montag, der 15 Mai 2017, ist ein sonniger und ruhiger Tag. Um 6:00 Uhr kehren viele Fischereischiffe aus der Ostsee zurück. In diesem Moment wird der Operator 1 auf ein auffälliges Verhalten des Schiffes Sulzbach-Rosenberg aufmerksam. Der hohe Schiffsverkehr erschwert die Beobachtung des verdächtigen Schiffes. Der Operator 1 gibt zwei Namen in die Suche ein:

„Sulzbach“ und „Rosenberg“. Das Schiff wird in der iLEXX-GUI selektiert bzw. die restlichen Schiffe werden maskiert.

- **Szenario 3:**

Freitag der 13 Januar 2017 um Mitternacht herrschte ein Unwetter über die Küste Eckenförde. Ein unbekanntes Schiff konnte das schlechte Wetter nicht überstehen und fängt auseinander zu brechen. Der Kapitän sendet einen Hilferuf. Ein Aufklärungsboot der WTD71 befindet sich zufällig in der Nähe. Das Boot fährt los, ruft die Zentrale an und teilt die Koordinaten des gefährdeten Schiffes mit. Ein Operator soll überprüfen, wie die Lage in dem Umkreis ist und ob weitere Informationen in der GUI über dieses Schiff vorhanden sind. Der Operator gibt die erhaltenen Koordinaten in die Suche ein. Ein aufgezeichnetes Schiff besitzt die gleichen Koordinaten und wird in der iLEXX-GUI selektiert und die restlichen Schiffe werden maskiert. Der Operator kennt jetzt weitere Informationen über das Schiff und kann die Lage in der Nähe überprüfen.

Nach schriftlichen Anmerkungen suchen

- **Szenario 4:**

Operator 1 erhält eine wichtige schriftliche Anmerkung von Operator 3. Sofort teilt Operator 1 diese Anmerkung dem Administrator mit und liest sie vor. Auf Anordnung des Administrators muss die Anmerkung noch einmal vorgelesen werden. Inzwischen wurden weitere Anmerkungen von anderen Operatoren erhalten. Operator 1 kann die Anmerkungen nicht mehr zuordnen, weshalb er „Operator 3“ in die Suche eingibt. Alle Nachrichten des Operators 3 werden selektiert bzw. die restlichen Nachrichten werden maskiert.

- **Szenario 5:**

Montag, der 15. Mai 2017 entdeckt ein Operator im Hamburger Hafen Unregelmäßigkeiten bei den Schiffen „Princes“ und „Gigant“. Er möchte prüfen, ob zuvor weitere Auffälligkeiten von anderen Operatoren notiert wurden. Der Operator gibt die beiden Namen in die Suche ein. Alle Nachrichten, die diese Wörter enthalten, werden selektiert bzw. die restlichen Nachrichten werden maskiert.

Nach Sperrzonen Suchen

- **Szenario 6**

Am Mittwoch, den 15. Februar 2017 wurde die Sperrzone in Eckenförde aufgeteilt. Operator 1 trägt die Änderung in die iLEXX-GUI ein. Bei der Aktualisierung überlappen sich zwei Sperrzonen. Die Sperrzone „Eckenförde Nord“ und „Eckenförde Süd“ müssen noch überarbeitet werden. Um eine bessere Übersicht zu erhalten, gibt Operator 1 „Eckenförde“ in die Suche ein. Beide Sperrzonen werden selektiert bzw. andere Sperrzonen werden maskiert.

Während die Szenarien die Einblicke in eventuelle Anwendungssituationen bieten, wird eine weitere Methode benötigt um die Funktion der Suche genauer beschreiben zu können.

3.1.2 Use Cases

Mit den Nutzungsszenarien im vorherigen Abschnitt wurden reibungslose Nutzungsbeispiele vorgestellt. Die normalen Abläufe werden in diesem Abschnitt durch Variationen und Fehlsituationen mithilfe von Anwendungsfällen ergänzt.

Mithilfe der Anwendungsfälle kann sichergestellt werden welche Informationen indexiert werden müssen damit die Suchanfragen des Nutzers behandelt werden können. Die Anwendungsfälle dienen auch dazu, im späteren Kapitel (5.1.1), die implementierte Suchfunktion zu testen. Sie werden verwendet um sicherzustellen, dass die entstandene Suchfunktion die funktionalen Anforderungen erfüllt.

Die im Folgenden eingeführten Anwendungsfälle wurden nach der Empfehlung von Alias Cockburn erstellt (Cockburn 2003).

Tabelle 1: Use Case 1

<table>
<tr><td colspan="2">Use Case 1: Suche nach Tracks mittels einer alphanumerischen Eigenschaftswert</td></tr>
<tr><td colspan="2">Alphanumerische Eigenschaften:
Name, Alias-name, Display-name, Track-type, Source Typ. ID, Course, Bearing, APP6-type, Longitude, Latitude, Speed, Threat-level, Affiliation.</td></tr>
<tr><td colspan="2">Umfang: Suchfunktion</td></tr>
<tr><td colspan="2">Akteure: Operator</td></tr>
<tr><td colspan="2">Vorbedingung:
• Die Polarpanel-View, die Track-Tabelle und die TDA sind eingeblendet.
• Auf der iLEXX-GUI sind mehrere Tracks aufgezeichnet.
• Der Operator kennt einen alphanumerischen Eigenschaftswert eines bestimmten Tracks</td></tr>
<tr><td colspan="2">Nachbedingung:
Mindestens der gesuchte Track wird in der Polarpanel-View, der Track-Tabelle und der TDA selektiert. Die Distraktoren werden ausgeblendet.</td></tr>
<tr><td colspan="2">Trigger:
Der Operator möchte einen bestimmten Track finden.</td></tr>
<tr><td>Beschreibung: Schritt</td><td>Aktion</td></tr>
<tr><td>1</td><td>Der Operator gibt einen alphanumerischen Eigenschaftswert eines bestimmten Tracks in die Suche ein.</td></tr>
<tr><td>2</td><td>Nur jeder Track, bei dem die Sucheingabe mit einem seiner alphanumerischen Eigenschaftswerte (auch wenn teilweise) übereinstimmt, wird selektiert. Die Distraktoren werden ausgeblendet.</td></tr>
<tr><td>Erweiterung: Schritt</td><td>Verzweigende Aktion</td></tr>
<tr><td>1a</td><td>Der Operator hat bei der Eingabe auf Groß- und Kleinschreibung nicht geachtet.</td></tr>
<tr><td>1b</td><td>Der Operator gibt den alphanumerischen Eigenschaftswert nicht vollständig ein.</td></tr>
<tr><td>1c</td><td>Der Operator gibt mehrere alphanumerische Eigenschaftswerte eines bestimmten Tracks in die</td></tr>
</table>

	Suche ein.
2d	Es wurden keine Treffer gefunden. Die Distraktoren werden ausgeblendet.

Tabelle 2: Use Case 2

Use Case 2: Suche nach Tracks mit numerischen Eigenschaftswerte	
Numerische Eigenschaftswerte: Name, Alias-name, Display-name, Track-type, Source Typ. ID, Course, Bearing, APP6-type, Longitude, Latitude, Speed, Threat-level, Affiliation.	
Umfang: Suchfunktion	
Akteure: Operator	
Vorbedingung: • Die Polarpanel, die Track-Tabelle und die TDA sind eingeblendet. • Auf der iLEXX-GUI sind mehrere Tracks aufgezeichnet. • Der Operator kennt einen oder mehrere numerische Eigenschaftswerte eines bestimmten Tracks.	
Nachbedingung: Mindestens der gesuchte Track wird in der Polarpanel, der Track-Tabelle und der TDA selektiert. Die Distraktoren werden ausgeblendet.	
Trigger: Der Operator möchte einen bestimmten Track finden.	
Beschreibung: Schritt	Aktion
1	Der Operator gibt einen numerischen Eigenschaftswert eines bestimmten Tracks in die Suche ein.
2	Nur jeder Track, bei dem die Sucheingabe mit einem seiner numerischen Eigenschaftswerte (auch wenn teilweise) übereinstimmt, wird selektiert. Die Distraktoren werden ausgeblendet.
Erweiterung: Schritt	Verzweigende Aktion
1a	Der Operator gibt den numerischen Eigenschaftswert nicht vollständig ein.
1b	Der Operator gibt mehrere numerische Eigenschaftswerte eines bestimmten Tracks in die Suche ein.
2b	Nur jeder Track, bei dem alle eingegebenen Werte mit mindestens einem seiner numerischen Eigenschaftswerte (auch wenn teilweise)

	übereinstimmen, wird selektiert. Die Distraktoren werden ausgeblendet.
2c	Es wurden keine Treffer gefunden. Die Distraktoren werden ausgeblendet.

Tabelle 3: Use Case 3

Use Case 3: Suche nach schriftlichen Nachrichten mittels einer alphanumerischen Eingabe.	
Eigenschaften: Autor, Datum, Inhalt.	
Umfang: Suchfunktion	
Akteure: Operator	
Vorbedingung: • Die Operator-Log-View ist eingeblendet. • Im Verlauf sind mehrere Nachrichten angezeigt.	
Nachbedingung: Alle Treffer-Nachrichten werden selektiert. Die Distraktoren werden ausgeblendet.	
Trigger: Der Operator möchte eine Nachricht finden.	
Beschreibung: Schritt	Aktion
1	Der Operator gibt ein Wort in die Suche ein.
2	Nur jede Nachricht, bei der die Sucheingabe mit einem ihrer Eigenschaftswerte (auch wenn teilweise) übereinstimmt, wird selektiert. Die Distraktoren werden ausgeblendet.
Erweiterung: Schritt	Verzweigende Aktion
1a	Der Operator gibt mehrere Wörter in die Suche ein.
2a	Nur jede Nachricht, bei der alle eingegebenen Wörter mit mindestens einem ihrer Eigenschaftswerte (auch wenn teilweise) übereinstimmen, wird selektiert. Die Distraktoren werden ausgeblendet.
1b	Der Operator hat bei der Eingabe auf Groß- und Kleinschreibung nicht geachtet.
2c	Es wurden keine Treffer gefunden. Die Distraktoren werden ausgeblendet.

Tabelle 4: Use Case 4

Use Case 4: Suche nach Sperrzonen mittels einer alphanumerischen Eingabe.	
Eigenschaften: Autor, Datum, Beschreibung.	
Umfang: Suchfunktion	
Akteure: Operator	
Vorbedingung: • Die TDA ist eingeblendet. • Im Verlauf sind mehrere Nachrichten angezeigt.	
Nachbedingung: Alle Sperrzonen werden selektiert. Die Distraktoren werden ausgeblendet.	
Trigger: Der Operator möchte eine Sperrzone finden.	
Beschreibung: Schritt	Aktion
1	Der Operator gibt ein Wort in die Suche ein.
2	Nur jede Sperrzone, bei der die Sucheingabe mit einem ihrer Eigenschaftswerte (auch wenn teilweise) übereinstimmt, wird selektiert. Die Distraktoren werden ausgeblendet.
Erweiterung: Schritt	Verzweigende Aktion
1a	Der Operator gibt mehrere Wörter in die Suche ein.
2a	Nur jede Sperrzone, bei der alle eingegebenen Wörter mit mindestens einem ihrer Eigenschaftswerte (auch wenn teilweise) übereinstimmen, wird selektiert. Die Distraktoren werden ausgeblendet.
1b	Der Operator hat bei der Eingabe auf Groß- und Kleinschreibung nicht geachtet.
2c	Es wurden keine Treffer gefunden. Die Distraktoren werden ausgeblendet.

3.1.3 Funktionale Anforderungen und Designentscheidungen

Die Anwendungsfälle haben die funktionalen Anforderungen an die Suchfunktion spezifiziert. In diesem Abschnitt werden diese in Punkten zusammengefasst. Außerdem werden danach einige Designentscheidungen aus den Anwendungsfällen abgeleitet.

Folgende Anforderungen wurden bei der Sucheingabe festgestellt:

- Die Sucheingabe kann ein Text, eine Zahl oder eine Kombination aus den beiden sein.
- Bei einer Satzeingabe muss jedes einzelne Wort in den Treffern als Ganzes- oder als Teilwort mindestens einmal vorkommen.
- Bei der Eingabe wird ohne Bestätigung unverzüglich gesucht.

Die Suchfunktion verhält sich bei der Durchsuchung wie folgt:

- Die Suche unterscheidet nicht zwischen Groß- und Kleinschreibung.
- Alle Informationen über Tracks, Sperrzonen und Nachrichten können durchsucht werden (Id, Name usw.)
- Die Suche verhält sich bei einer Zahleneingabe ähnlich wie bei einer Texteingabe.

Der Nutzer nutzt die Suchfunktion über die Eingabe von Zeichenketten. An der iLEXX-GUI wird also eine Stelle benötigt um Sucheingabe entgegenzunehmen.

Die Suchfunktion selektiert Treffer auf der iLEXX-GUI. Die Selektion erfolgt mit der Maskierung der restlichen Informationen. Diese Art der Darstellung kann auf die bereits bestehenden Bereiche der iLEXX-GUI angewendet werden, es wird also kein zusätzlicher Bereich benötigt.

3.1.4 Anforderungen an die Filterfunktion

Die im vorherigen Abschnitt spezifizierten Anwendungsfälle beschreiben die funktionalen Anforderungen an die Suchfunktion. Die Anforderungen an der Filterfunktion bauen auf denselben Anforderungen auf, setzen allerdings weitere Bedingungen voraus.

- Die ausschließliche Suche nach Tracks, Sperrzonen oder Nachrichten
- Die ausschließliche Suche nach einer bestimmten Eigenschaft (z. B. Speed)
- Die Suche nach Werten in einem bestimmten Bereich (z. B. 13,5 bis 20)
- Die Suche nach Werten, die größer bzw. kleiner sind als die Sucheingabe
- Die Zeitangeben durchsuchen können

3.2 Die grobe Struktur der Suchfunktion

In diesem Abschnitt wird erläutert, wie die Suchfunktion als ein weiteres Plug-In „*ilx.e4.search*“ integriert wird. Die Integration ist auf die Vorgehensweise aus 2.3.2 basiert.

Zuerst wird eine Service-Schnittstelle „*ilx.e4.model.search.ISearchEngine*“ vordefiniert. Diese Schnittstelle wird von einem Service-Provider „*ilx.e4.search.engine.SearchEngine*“ implementiert. Eine Instanz aus dem Service-Provider wird dann, mithilfe der Service-Components, an allen Plug-Ins bereitgestellt. Folgendes verdeutlicht die beschriebene Struktur, mittels eines Komponentendiagramms.

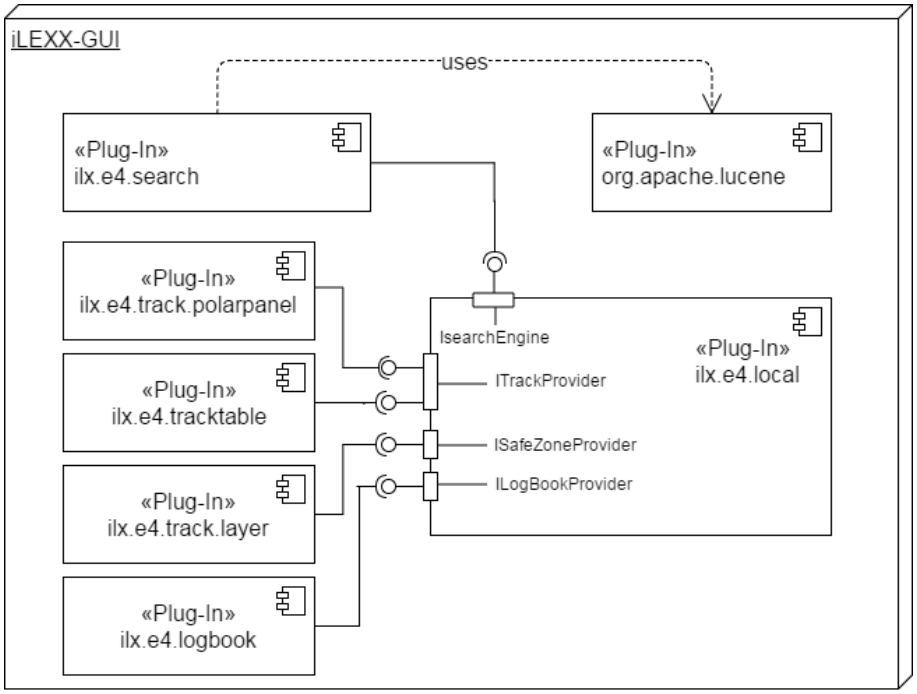

Abbildung 9: Die Integration der Suchfunktion als ein weiterer Plug-In in die iLEXX-GUI

Die iLEXX-GUI erhält Daten von unterschiedlichen Sensoren. Die Daten werden zentral von verschiedenen Daten-Providern (Track-, SafeZone- und Logbook-Provider) gespeichert. Die Abbildung zeigt, dass für jeden Datentyp einen eigenen Daten-Provider existiert, welcher die Daten in der ganzen Anwendung bereitstellt. Alle Bereiche, die Daten auf der GUI Darstellen, können dann den entsprechenden Daten-Provider injizieren.

Abbildung 9 zeigt auch welche Struktur entsteht wenn die Suchfunktion als ein weiterer Plug-In in die iLEXX-GUI integriert wird. Außerdem ist es auch zu sehen, dass die Daten-Provider als Service-Provider (2.3.2) realisiert sind. Alle Service-Schnittstellen der Anwendung in dem Plug-In „*ilx.e4.local*“ definiert. Die einzelnen Bereiche, bspw. die Track Tabelle und die Polarpanel-View, injizieren dann die benötigten Service-Provider.

Die Daten aus den Providern werden ebenso von der Suchfunktion benötigt, um Informationen indexieren und durchsuchen zu können. Deswegen injizieren alle Daten-Provider den Service-Provider der Suchfunktion „*SearchEngine*“. Die Daten werden dann über das injizierte „*SearchEngine*“-Objekt indexiert. An dieser Stelle ist es also notwendig die Daten-Provider und die zu indexierenden Daten näher zu betrachten.

Einen genaueren Blick auf die Daten-Provider und deren Daten liefert folgendes UML-Klassendiagramm:

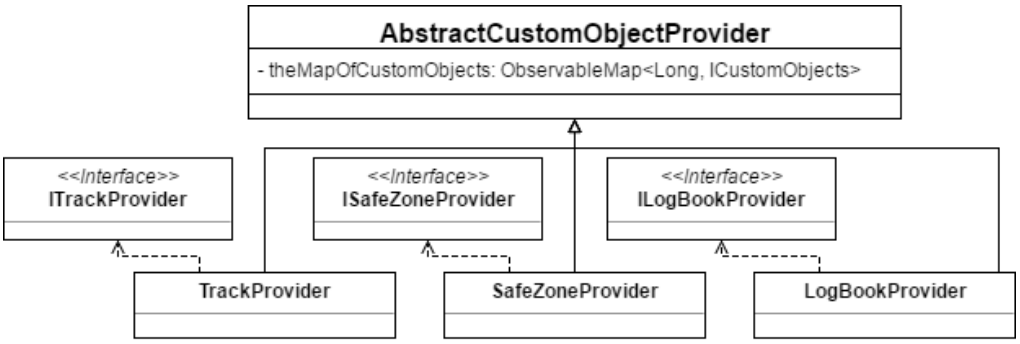

Abbildung 10: Struktur der Daten-Provider mittels UML-Klassendiagramm

Wie bereits erwähnt, werden Daten, wie Tracks, Sperrzonen und Nachrichten, von verschiedenen Daten-Providern zur Verfügung gestellt. Die einzelnen konkreten Daten-Provider besitzen eine gemeinsam ähnliche Struktur. Sie wurden alle aus einer gemeinsamen Klasse abgeleitet. Die abgebildeten Service-Schnittstellen sind in dem Komponentendiagramm (Siehe Abbildung 9) wieder zu finden.

Wie in Abbildung 10 zu sehen ist, sind die Daten aus einer gemeinsamen abstrakten Klasse „*AbstractCustomObject*“ abgeleitet. Die Suchfunktion gegen diese Klasse zu implementieren erzielt eine hohe Kompatibilität. Unabhängig davon, ob Informationen über einen Track, eine Sperrzone oder eine Nachricht indexiert werden sollen, wird für alle der gleiche Aufruf verwendet.

Die Suchfunktion wird die Ergebnisse weder selbst präsentieren noch selber zur Verfügung stellen. Die Suchfunktion markiert die Daten aller Daten-Provider als Suchergebnisse und die verschiedenen Bereiche stellen diese individuell dar.

3.3 Feinentwurf auf Klassenebene

Aus der Anforderungsspezifikation im Abschnitt 3.1.2 werden Entscheidungen über die Klassenbildung getroffen.

3.3.1 Die Service-Schnittstelle „ISearchEngine.java“

Zuerst wird eine Service-Schnittstelle benötigt, die den Methodennamen, die Parameter und den Rückgabewert festlegt. Ein UML-Klassendiagramm zeigt alle Methoden dieser Schnittstelle im Folgenden

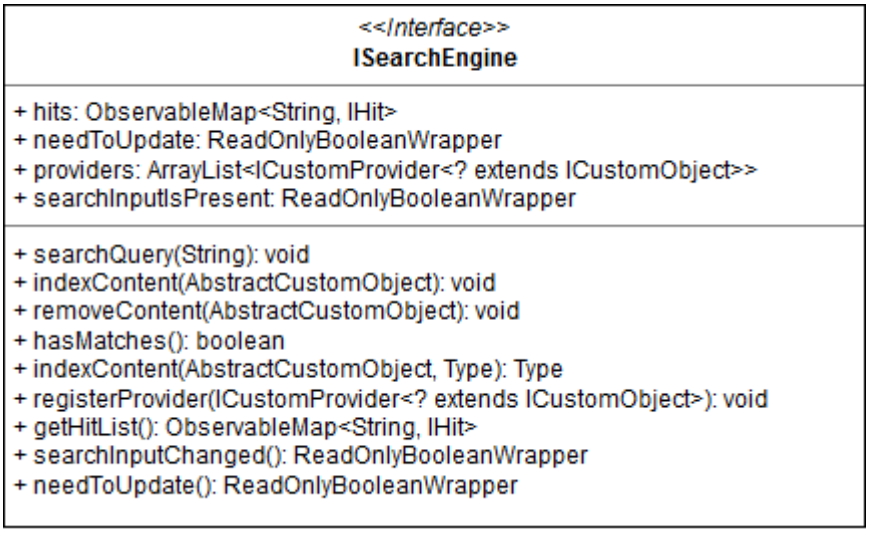

Abbildung 11: Das UML-Klassendiagramm der Service-Schnittstelle *ISearchEngine.java*

Zuerst werden Methoden erläutert, die der Verarbeitung von Informationen dienen. Diese besitzen keinen Rückgabewert oder liefern höchstens den Erfolg des Aufrufs zurück. In der Abbildung 10 zeigen die zu indexierenden Daten eine gemeinsame Schnittstelle. Daher erwarten die Methoden zur Verarbeitung des Indexes ein Objekt vom Typ „*ilx.e4.model.AbstractCustomObject*“. Um Informationen über ein Objekt zu indexieren wird „*indexContent*“ aufgerufen und „*removeContent*“ beim Entfernen. Eine Suchanfrage wird mittels „*searchQuery*“ ausgeführt. Die Eingabe ist eine beliebige Zeichenkette „*String*“. Die zuletzt ausgeführte Suchanfrage kann „*updateSearchQuery*“ aktualisieren. Jeder Daten-Provider kann mit „*registerProvider*“ registriert werden. Die Registration ermöglicht Daten des Providers als Treffer zu markieren.

Die restlichen Methoden erwarten keine Eingabeparameter aber liefern Werte zurück. Die Zustände der Suchfunktion können mit diesen Methoden abgefragt werden. Sie sind bei der Darstellung der Ergebnisse wichtig. Änderungen bei der Sucheingabe können mit „*searchInputChanged*“ verfolgt werden. Dies schließt das Eintragen, Löschen und Ersetzen der Eingaben. Die Erforderlichkeit der Ergebnisaktualisierung kann mittels „*needToUpdate*“ ermittelt werden. Um abzufragen ob die Suchfunktion bei der letzten Suchanfrage Treffer erlangt hat wird „*hasMatches*“ aufgerufen.

3.3.2 Der Service-Provider „SearchEngine.java“

Die Service-Schnittstelle legte fest über welche Methoden Kommuniziert wird. Als nächstes wird ein Service-Provider „*SearchEngine.java*“ implementiert. Der Service-Provider verfügt aber über weitere Methoden die nicht vordefiniert sind. Solche Methoden werden nur innerhalb des eigenen Plug-Ins aufgerufen. Sie ermöglichen eine gute Strukturierung und die Wiederverwendung des Quellcodes. Ein UML-Klassendiagramm zeigt alle Methoden der Service-Provider im Folgenden:

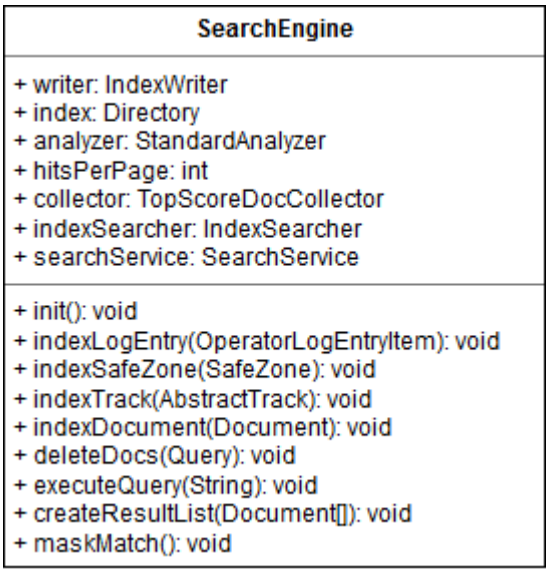

Abbildung 12: Das UML-Klassendiagramm des Service-Providers *SearchEngine.java*

Alle benötigten Ressourcen werden bei der Instanziierung der Klasse in der Methode „*init*“ geladen bzw. erstellt. Darunter die Objektvariablen dieser Klasse, wie „*analyzer*“, „*index*“ „*reader*“ „*searcher*“ und „*writer*“.

Methoden die zur Indexierung beitragen sind „*indexLogEntry*“, „*indexSafeZone*“ „*indexTrack*“, „*indexDocument*“ und „*deleteDocs*“. Je nachdem welches Objekt indexiert werden muss, erstellt die Methoden „*indexLogEntry*“, „*indexSafeZone*“ und „*indexTrack*“ das passende Dokument. Unabhängig vom Typ, wird mit „*indexDocument*“ ein Dokument in den Index geschrieben. „*deleteDocs*“ löscht bestimmte Dokumente aus dem Index.

Die restlichen Methoden dienen der Durchsuchung und der Bereitstellung der Ergebnisse. Nebenläufigen Threads die mit „*searchQuery*“ gestartet werden, nutzen „*executeQuery*“ um die Suchanfrage im gleichen Thread ausführen zu können. Nach der Ausführung können die resultierenden Treffer mittels „*createResultList*“ gespeichert werden.

Abhängig von den Ergebnissen, kann die Suchfunktion mit „*maskMatch*“ alle Daten der Daten-Provider markieren. Jedes Objekt kann geprüft werden ob es ein Treffer ist. Dazu wird das Objekt der Methode „*isHit*“ übergeben. Die Bereiche der iLEXX-GUI können dann die Ergebnisse entsprechend präsentieren.

3.3.3 Hilfsklassen

Der Service-Provider „*SearchEngine.java*“ bündelt alle Methoden, die eine Suchfunktion brauchen, in einer Klasse. Es sind selbstverständlich weitere Klassen notwendig um die Suchfunktion zu realisieren. Mit der folgenden Abbildung wird gezeigt in welcher Beziehung der Service-Provider „*SearchEngine.java*“ mit den Hilfsklassen und den Komponenten aus Lucene und JavaFX steht.

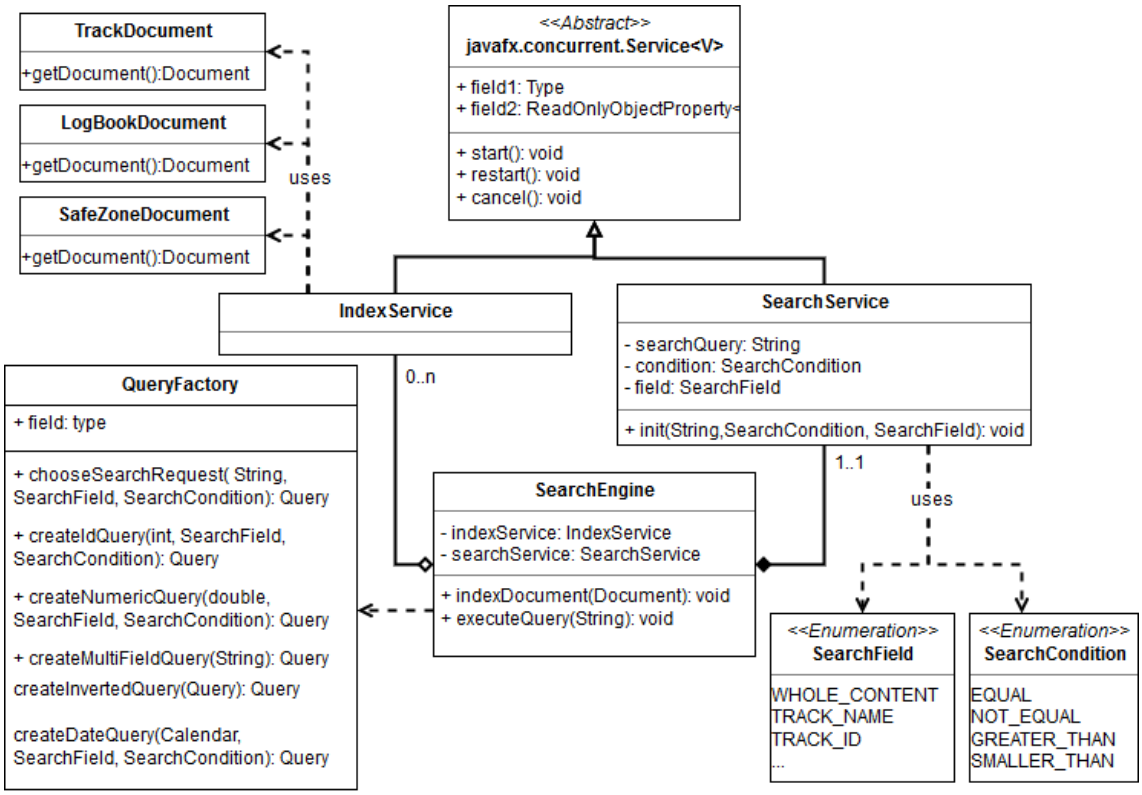

Abbildung 13: Die Beziehung zwischen dem Service-Provider und den Hilfsklassen

Wie in Abschnitt 2.1.2 erwähnt wurde, wird ein Dienst zur Indexierung bereitgestellt. JavaFX bietet mehrere Mechanismen zur Nebenläufigkeit an. *`javafx.concurrent.Service<V>`* bieten, neben der Erledigung von Aufgaben in einem Thread, die Möglichkeit abgebrochen oder neu gestartet zu werden. Weiterhin können eigene Variablen deklariert und bei der Erzeugung Eingebeparameter übergeben werden.

Informationen müssen aus den Daten, wie Sperrzonen, Tracks oder Nachrichten, extrahiert werden. Daher wird jedem Typ eine eigene Dokumentklasse zugeordnet. Jede Klasse besitzt die entsprechende Implementierung zum Extrahieren der Informationen.

Die Filterfunktion durchsucht ausschließlich bestimmte Informationstypen und Eigenschaften (3.1.4). Deswegen wird ein Enum-typ „`SearchField`“ benötigt um die Suche auf bestimmte Felder einzugrenzen. Außerdem kann die Filterfunktion, Werte die Größer oder kleiner als die Sucheingabe sind und in einem bestimmten Bereich liegen, durchsuchen. Dazu wird die Suchbedingung „`SearchCondition`“ benötigt.

4 Implementierung der Suchfunktion

Mithilfe der gewonnen Erkenntnisse aus Kapitel 2 und durch die Konzeption in Kapitel 3, kann mit der Implementierung der Suchfunktion begonnen werden. Die Herangehensweise in diesem Kapitel ähnelt dem von Kapitel 2. Zuerst wird ein Mechanismus zur Indexierung vorgestellt und dann der dazu passende Suchmechanismus erklärt.

4.1 Der Indexierungsmechanismus

Zuerst wird gezeigt, wie die Suchfunktion zum Indexieren der Daten genutzt wird. Dann wird erklärt, wie die Informationen aus den Daten extrahiert und in den Dokumenten angelegt werden. Zum Schluss wird erläutert, wie die entstandenen Dokumente in den Index geschrieben werden.

4.1.1 Informationsakquisition durch Anbindung an die Daten-Provider

Daten, wie Sperrzonen, Tracks oder schriftliche Nachrichten, müssen zuerst mit der Suchfunktion indexiert werden. Daher muss als erstes die Suchfunktion an die Daten-Provider gebunden werden.

In jedem Daten-Provider, wird eine Variable „*searchEngine*“ deklariert. Diese Variable ist vom Typ „*ISearchEngine*“ und wird nicht erzeugt sondern injiziert.

```
66
67      private ISearchEngine searchEngine;
68      @Inject
69      @Optional
70      public void onSearchEngineInjected(ISearchEngine searchEngine) {
71          this.searchEngine = searchEngine;
72          this.searchEngine.registerProvider(this);
73      }
```

Listing 1: Das Injizieren um die Suchfunktion zu nutzen

Nach dem erfolgreichen Injizieren kann die Variable „*searchEngine*“ benutzt werden um Daten zu indexieren. Dazu werden die von der Schnittstelle bekannten Methoden aufgerufen. An jeder Stelle, bei der neue Daten ankommen oder aktualisiert werden, wird „*indexContent()*“ aufgerufen.

```
194     @Override
195     public boolean addTrack(final AbstractTrack newTrack) {
196         if (addObject(newTrack)) {
197             if (searchEngine != null) {
198                 searchEngine.indexContent(newTrack);
199             }
200             return true;
```

Listing 2: Die Übergabe eines neuen Tracks zur Indexierung

An den Stellen wo Daten gelöscht werden, wird „*removeContent()*“ genutzt.

```java
    @Override
    public boolean removeTrack(final AbstractTrack trackToRemove) {
        if (removeObject(trackToRemove)) {
            if (searchEngine != null) {
                searchEngine.removeContent(trackToRemove);
            }
            return true;
```

Listing 3: Die Übergabe eines Tracks zum Entfernen aus dem Index

Das erfolgreiche Injizieren dieser Variable setzt voraus, dass das Plug-In der Suchfunktion „*ilx.e4.search*“, wie in 3.2, integriert wird. Das geschieht indem der Service-Provider „*SearchEngine.java*“ aus „ilx.e4.search“ die Service-Schnittstelle „*ISearchEngine.java*“ implementiert und deklarativ mit den Service-Components veröffentlicht wird.

```java
package ilx.e4.search.engine;

import org.osgi.service.component.annotations.Component;

@Component(
        name = "ilx.e4.search.engine.SearchEngineContextFunction",
        property = "service.context.key:String=ilx.e4.model.searchengine.ISearchEngine",
        service = org.eclipse.e4.core.contexts.IContextFunction.class)
public class SearchEngineContextFunction extends GenericContextFunction<ISearchEngine> {

    @Override
    public Class<? extends ISearchEngine> classToBeInstantiated() {
        return SearchEngine.class;
    }

}
```

Listing 4: Die deklarative Veröffentlichung des Service-Providers

Bei der Akquisition von Informationen in Echtzeit muss ein Dienst bereitgestellt werden, um neuankommenden Daten, kontinuierlich zu Indexieren(Gospodnetic / Hatcher / McCandless 2010, S. 11). Die Aufrufe in den Daten-Providern über das injizierte Objekt führen demnach die Indexierung nicht sofort aus, sondern starten dafür einen nebenläufigen Thread „*IndexService.java*“.

4.1.2 Erstellung von Objektspezifischen Dokumenten

Jeder Thread erstellt zuerst ein passendes Dokument. Das Erzeugen eines Dokumentes extrahiert, für die Suche relevante, Informationen aus den Daten.

```java
    public TrackDocument(final AbstractTrack track) {
        indexNewTextField(TYP, DOC_TYP);
        indexNewTextField(ID, track.idProperty().get());
        wholeContent = track.getAliasNameAttr() + " " + track.getName() + " "
        + track.getTrackSource() + " " + track.getHexId() + " "
        + track.getDisplayName() + " " +track.getAltitude() + " "
        + track.getCourse() + " " + track.getHeading() + " "
        + track.getLatitude() + " " + track.getLongitude() + " "
        + track.getSpeed();
        indexNewTextField(entireContentField, wholeContent);
    }
```

Listing 5: Extrahierung von Informationen aus einem Track

```
52    protected void indexNewTextField(String fieldName, Object value) {
53        if (value != null) {
54 document.add(new TextField(fieldName, value.toString(), Store.YES));
55
56        }
57    }
```

Listing 6: Ein neues Feld in das Dokument hinzufügen

Die Suchfunktion soll alle Eigenschaften durchsuchen können. Daher werden alle Eigenschaftswerte indexiert. Zusätzlich hat jedes Dokument, zur Identifikation, zwei Felder. Ein Feld repräsentiert die Identifikationsnummer. Weil Daten von verschiedenen Typen die gleiche Identifikationsnummer besitzen können, wird ein zusätzliches Feld benötigt um den Typ zu bestimmen.

Alle Informationen zu einem Objekt werden in einem einzigen Feld gespeichert. Dies beschleunigt später die Ausführung von Abfrageobjekten.

Die Filterfunktion kann ausschließlich bestimmte Datentypen und/oder Eigenschaften durchsuchen. Daher werden die Eigenschaftswerte zusätzlich in die entsprechenden getrennten Felder gespeichert.

```
79          document.add(new LongField(TRACK_ID, track.idProperty().getValue(), Store.YES));
80          document.add(new DoubleField(ALTITUDE, track.getAltitude(), Store.YES));
81          document.add(new DoubleField(COURSE, track.getCourse(), Store.YES));
82          document.add(new DoubleField(HEADING, track.getHeading(), Store.YES));
83          document.add(new DoubleField(LATITUDE, track.getLatitude(), Store.YES));
84          document.add(new DoubleField(LONGITUDE, track.getLongitude(), Store.YES));
85          document.add(new DoubleField(SPEED, track.getSpeed(), Store.YES));
86          document.add(new DoubleField(DISTANCE, distance, Store.YES));
```

Abbildung 14: Informationen in verschiedenen Feldern in das Dokument hinzufügen

4.1.3 Indexierung der Dokumente

Sobald das Dokument erstellt wurde, wird es in den Index getragen. Zuvor wird sichergestellt, dass kein anderes Dokument mit Informationen über das gleiche Objekt im Index vorhanden ist. Hierzu muss ein „*Query*“-Objekt erstellt und die Methode „*removeDoc*“ übergeben werden. Das „*Query*“-Objekt selektiert nun die Dokumente mit dem gleichen Typ und gleicher Identifikationsnummer, woraufhin die Methode diese Dokumente aus dem Index entfernt.

Das neue Dokument wird zum Schluss mit „*indexDocument()*“ indexiert und die Suche gegebenenfalls aktualisiert.

4.2 Der Suchmechanismus

Die Aktualisierung der Suche ist letztendlich die Wiederausführung der letzten Suchanfrage, falls dies bereits geschehen ist und die Sucheingabe noch vorhanden ist. Der Unterschied zwischen der Aktualisierung und eine vom Benutzer ausgelöste Suche, ist die Änderung in der Sucheingabe.

4.2.1 Die Sucheingabe

Ausgehend von den Anforderungen aus dem Abschnitt 3.1.3, wird für die Entgegennahme der Nutzereingabe ein Textfeld implementiert. Hierzu wird wie in 2.3.3 vorgegangen. Zuerst wird das Feld in einer bereits vorhandenen FXML-Datei deklariert und anschließend mit der Annotation „*@FXML*“ injiziert.

```
<HBox alignment="CENTER" HBox.hgrow="ALWAYS">
    <children>
        <TextField id="searchField" fx:id="searchField"
            promptText="Search...">
    </children>
</HBox>
```

Listing 7: Deklarierung des Suchfeldes in der FXML-Datei

Wie bei der Indexierung (Siehe 4.1.1), wird für die Durchsuchung die Instanz aus „*ISearchEngine.java*“ benötigt und daher injiziert. Nur wenn das Injizieren erfolgreich ist, wird das Suchfeld angezeigt.

```
140     @FXML
141     private TextField searchField;
142     @Inject
143     @Optional
144     private ISearchEngine searchEngine;
145     @Inject
146     @Override
147     public void initialize(final URL location, final ResourceBundle resources) {
148         if (searchEngine == null) {
149             searchField.setVisible(false);
150         } else {
151             searchField.setOnKeyReleased(e -> {
152                     searchEngine.searchQuery(searchField.getText());
153             });
154         }
```

Listing 8: Darstellung eines Suchfeldes nur wenn eine Suchfunktion verfügbar ist

Bei jeder Änderung der Eingabe des Suchfeldes wird eine Suchabfrage gebildet und ausgeführt.

4.2.2 Bildung und Ausführung der Suchabfrage

So wie bei der Indexierung (Siehe 4.1.1) wird bei jedem Suchvorgang ein separater Thread gestartet. Dieser Thread führt dann die Suchabfrage mit der Methode „*executeQuery*“ aus.

Bevor ein „*Query*“-Objekt (siehe 2.2.2) erstellt wird, müssen alle Sonderzeichen aus der Sucheingabe entfernt werden. Sonderzeichen können bei vielen Typen von „*Query*“-Objekten als reguläre Ausdrücke interpretiert werden. Das Schriftzeichen „*“ wird beispielsweise als eine beliebige Zeichenfolge ausgewertet. Kommt dieses Zeichen am Anfang der Sucheingabe vor, führt die Ausführung des Abfrageobjekts zu einer Fehlermeldung. Sonderzeichen können also unbestimmbares und unerwünschtes Verhalten der Suche verursachen und müssen eliminiert werden.

```
239             String searchInput = text.toLowerCase().replaceAll("\\p{Punct}", " ");
```

Listing 9: Sonderzeichen mit Leerzeichen ersetzen

Bei der Erstellung eines „*Query*“-Objekts wird die Klasse „*org.apache.lucene.queryparser.flexible.standard.StandardQueryParser*“ verwendet. Diese Klasse bietet neben der Kompilierung von regulären Ausdrücken, eine hohe Konfigurationsmöglichkeit (Gospodnetic / Hatcher / McCandless 2010, S. 320). Die für diesen Kontext relevante Konfiguration ist die Nutzungserlaubnis für das

Schriftzeichen „*“ am Anfang einer Sucheingabe. Somit können Teilwörter folgendermaßen abgefragt werden:

```
124         String field;
125
126         StandardQueryParser stdQP = new StandardQueryParser(analyzer);
127         stdQP.setAllowLeadingWildcard(true);
128         stdQP.parse("*" + querystr + "*", field);
```

Listing 10: Konfiguration und Bildung eines Anfrageobjektes

Die Sucheingabe kann aber aus mehreren Wörtern bestehen. Daher wird für jedes Wort, wie oben erläutert, ein „*Query*“-Objekt erstellt. Zum Schluss werden alle erstellten „*Query*“-Objekte zu einem einzigen Objekt mit „*BooleanQuery*“ zusammen gebündelt.

```
125         Builder bq = new BooleanQuery.Builder();
126         try {
127             for (String field : SearchFields.getEntireContentFields()) {
128                 Builder bq1 = new BooleanQuery.Builder();
129                 for (String word : querystr.split(" ")) {
130                     bq1.add(stdQP.parse("*" + word + "*", field), Occur.MUST);
131                 }
132                 bq.add(bq1.build(), Occur.SHOULD);
133             }
134             return bq.build();
135
136         } catch (QueryNodeException e) {
137             e.printStackTrace();
138         }
139
```

Listing 11: Bündeln von Abfrageobjekte mit BooleanQuery

Die Sucheingabe kann mit jedem Eigenschaftswert übereinstimmen. Daher müssen alle Informationen in jedem Dokument überprüft werden. Doch mit dem entstandenen „*Query*“-Objekt wird nur ein einziges Feld „*whole_content*“ durchsucht, das alle Eigenschaftswerte enthält. Dies ist wesentlich effizienter als in jedem Dokument alle Felder zu durchsuchen. Dies liegt an der benötigten Zugriffsanzahl auf die einzelnen Dateien in dem Index. Um dies zu beweisen wird für jede der beiden Fälle eine Formel zur Berechnung der benötigten Zugriffsanzahl erstellt und anschließend verglichen.

Die Zugriffe sind in Abbildung 7 mit Pfeilen zwischen den einzelnen Dateien visualisiert. Jeder Pfeil präsentiert eine gewisse Anzahl an Zugriffen auf die Dateien. Zur Erstellung der Formel stellt *n* die Anzahl der im System vorhandenen Tracks dar und *m* die Anzahl der Eigenschaften die jeder Track besitzt. Die Anzahl der Tracks multipliziert mit der Anzahl der Eigenschaften ergibt die Anzahl der Zugriffe zwischen den Dateien „*.fnm*“ und „*.tis*“. a_i gibt die Anzahl der Zugriffe von der Datei „*.tis*“ auf „*.frq*“ und b_j die Anzahl der Zugriffe von der Datei „*.frq*“ auf „*.prx*“.

Bei der Durchsuchung in einem einzigen Feld ergibt sich folgende Gleichung:

$(1_{feld} * n_{tracks}) + \sum^{k}_{1} a_i + \sum^{m}_{1} b_j$

Bei der Durchsuchung aller Felder vervielfacht sich die Zugriffsanzahl von der „.fnm“ auf „*.tis*“ um die Anzahl der Eigenschaften. In diesem Fall können aber Treffer nicht nur das gesuchte Wort mehrmals besitzen, sondern auch in verschiedenen Feldern. Dies verursacht redundante Verweise zwischen den Dateien „*.tis*“, „*.frq*“ und „*.prx*“. Dadurch erhöht sich die Anzahl der Zugriffe bei a_i und bi um r_a bzw. r_b. Diese Gegebenheit ergibt folgende Gleichung:

$(m_{felder} * n_{tracks}) + (\sum^{k}_{1} a_i) + r_a + (\sum^{m}_{1} b_i) + r_b$

Die Aussage, dass die Suche in einem einzigen Feld weniger Zugriffe erfordert als bei mehreren, wird durch den folgenden Vergleich beider Gleichungen bestätigt.

$(1_{feld} * n_{tracks}) + \sum^{k}_{1} a_i + \sum^{m}_{1} b_j < (m_{felder} * n_{tracks}) + (\sum^{k}_{1} a_i) + r_a + (\sum^{m}_{1} b_j) + r_b$

$n_{tracks} < (m_{felder} * n_{tracks}) + r_a + r_b$

$1 < m_{felder} + (r_a + r_b) / n_{tracks}$

$0 < m_{felder} - 1 + (r_a + r_b) / n_{tracks}$

Aus dieser Gegebenheit wird mit dem „*Query*“-Objekt ein einziges Feld mit allen Informationen durchsucht.

Das zum Schluss entstandene „*Query*“-Objekt kann benutzt werden, um nach Informationen zu suchen, wie in 3.1.3 beschrieben wurde. Anschließend wird das „*Query*“-Objekt wie in 2.2.3 ausgeführt. Das Ergebnis ist eine Sammlung von Dokumenten, die Bedingungen des „*Query*“-Objekts erfüllen.

4.2.3 Bildung der Suchabfrage für die Filterfunktion

Die Erstellung von „*Query*“-Objekte, die der Durchsuchung einer bestimmten *textuellen* Eigenschaft dienen, ähnelt stark der im vorherigen Abschnitt erläuterten Vorgehensweise.

Bei der Durchsuchung von *numerischen* Eigenschaften wird die Klasse „*TermRangeQuery*“ (Siehe Abbildung 6) verwendet. Mit dieser Klasse können Suchabfrageobjekte, zur Durchsuchung von Werten in einem bestimmten Bereich, erstellt werden. Dazu muss der Name des zu durchsuchenden Feldes, der minimale und maximale Wert eingegeben werden. Durch die Eingabe desselben minimalen und maximalen Wertes wird nach einem bestimmten Wert gesucht.
Bei der Filterfunktion sind mehrere Angaben erforderlich. Abgesehen von der Sucheingabe muss sowohl die Bedingung (größer, kleiner, etc.) als auch der Feldname angegeben werden. Folgender Quellcode zeigt, wie ein „*Query*“-Objekt, beispielsweise bei der Durchsuchung von Koordinaten, erstellt wird:

```
87    public static Query createNumericQuery(final double querystr, final String typ, final String con) {
88        double min = 0d;
89        double max = 0d;
90        switch (con) {
91        case "GreaterThan":
92            min = querystr;
93            max = DOUBLE_POSITIVE_INFINITY;
94            break;
95        case "LessThan":
96            min = DOUBLE_NEGATIVE_INFINITY;
97            max = querystr;
98            break;
99        default:
00            min = querystr;
01            max = querystr;
02
03        }
04
05        final Query q = NumericRangeQuery.newDoubleRange(typ, min, max, true, true);
06        return q;
07    }
08
```

Abbildung 15: Die Erzeugung eines „*Query*“-Objekts mit *NumericRangeQuery*

Ganz anders musste die Durchsuchung von Datierungen implementiert werden. In der Regel werden Datierungen mit Lucene durchsucht indem, beim Indexieren ein Datum

zu einem „String“ und bei der Durchsuchung wieder zu einem Datum umgewandelt wird (Gospodnetic / Hatcher / McCandless 2010, S. 218). Dieses Vorgehen stellt, im Rahmen dieser Arbeit, keine gute Alternative dar. Die Umwandlungen machen es unmöglich die vorausgesetzte Latenzzeit der Suchvorgänge einzuhalten (Siehe Rahmenbedingung). Stattdessen wurden Datierungen durch mehrere Felder nach Genauigkeitsgrad indexiert (Jahr- bis Millisekunden-Angabe). Zusätzlich wurde ein Mechanismus zum Erzeugen von Abfrageobjekten implementiert. Das „*Query*“-Objekt setzt sich aus mehreren Query-Objekten zusammen. Jedes „*Query*“-Objekt durchsucht die Felder mit einem bestimmten Genauigkeitsgrad. Anschließend werden sie so gebündelt, dass die Dokumente tiefgehend nach dem Genauigkeitsgrad abgefragt werden (genaue Implementierung siehe Listing 21).

4.2.4 Verarbeitung der gesammelten Dokumenten

Aus den Dokumenten wird eine Trefferliste erstellt. Treffer erhalten ihre eigene Klasse „IHit“. Mit dieser Klasse kann die Trefferliste gespeichert werden, ohne die realen entsprechenden Daten zu nutzen. Dies meidet unnötige Zugriffe auf die Daten. Die Treffer werden in eine „*HashMap*“ gespeichert. Dies bietet den Vorteil, nicht über alle Elemente der Liste iterieren zu müssen. Stattdessen wird das Vorhandensein des Schlüsselwerts geprüft. Der Schlüsselwert setzt sich aus der Identifikationsnummer und dem Typ des Treffers zusammen.

Die Suchfunktion soll die Treffer in der iLEXX-GUI nicht selbst präsentieren, da dies eine vermeidbare Abhängigkeit zu den anderen Projekten erzeugt. Stattdessen kennzeichnet die Suchfunktion alle Treffer. Die Bereiche der GUI können die Suchergebnisse dann eigenständig darstellen. Dazu wird die Schnittstelle „AbstractCustomObject“ angepasst. Eine „ReadOnlyBooleanWrapper“ variable mit den zugehörigen Getter- und Setter-Methoden wird deklariert.

```
15      ReadOnlyBooleanWrapper isMasked = new ReadOnlyBooleanWrapper(this, "isMasked", false);
16      public void mask() {
17          isMasked.set(true);
18      }
19      public void unmask() {
20          isMasked.set(false);
21      }
22      public boolean isMasked() {
23          return isMasked.get();
24      }
25
26      public AbstractCustomObject(final long id, final String name) {
27          this.id = new ReadOnlyLongWrapper(this, "id", id);
28          this.name = new ReadOnlyStringWrapper(this, "name", name);
29      }
30
```

Listing 12: Setzen eines Flags um Treffer kennzeichnen zu können

Die Selektion eines Treffers wird mit der Maskierung der restlichen Informationen umgesetzt. Alle Informationen sind nicht maskiert solange keine Suchabfrage ausgeführt wurden ist. Daher ist „*false*“ der Anfangswert der „*isMasked*“-Variable. Wenn eine Suchabfrage ausgeführt wird, erhält die Variable bei Treffern den Variablenwert „*false*“ während der Rest den Wert „*true*“ zugewiesen bekommt.

Folgendes Flussdiagramm zeigt die Semantik dieser Vorgehensweise:

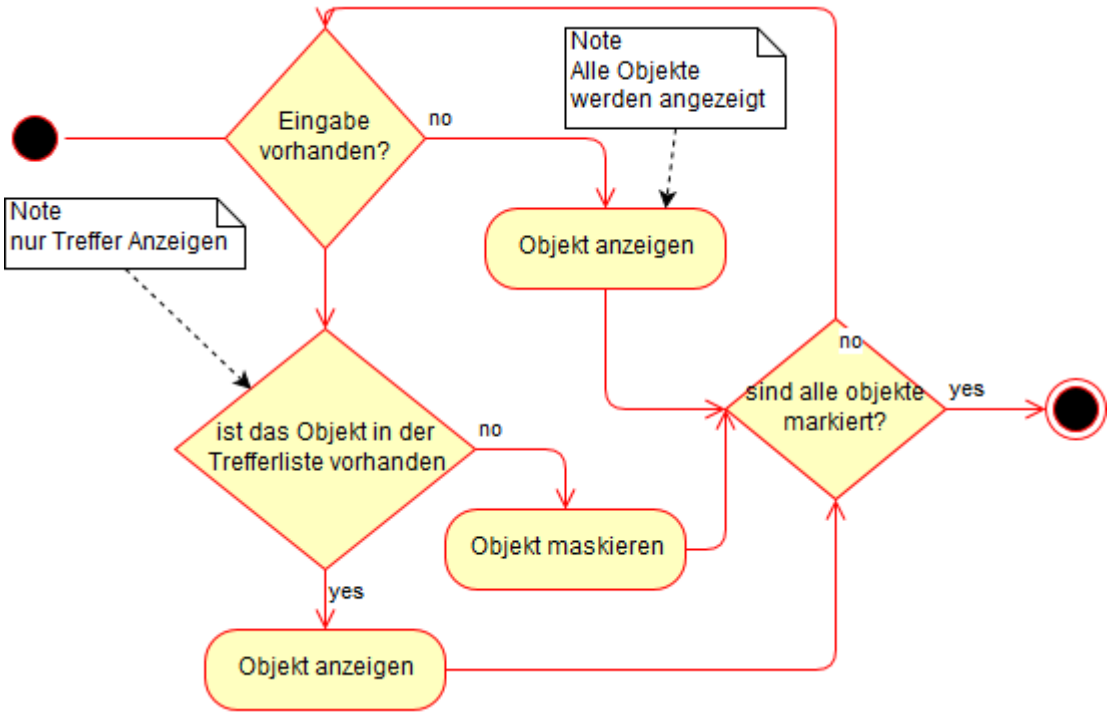

Abbildung 16: Flussdiagramm zur Kennzeichnung von Treffern

Mit dieser Vorgehensweise werden alle Daten der Daten-Provider gekennzeichnet.

```
185     public void maskMatch() {
186         for (ICustomProvider<? extends ICustomObject> p : providers) {
187             ContentType type = null;
188             if (p instanceof ISafeZoneProvider) {
189                 type = ContentType.safezone;
190             }
191             if (p instanceof ILogBookProvider) {
192                 type = ContentType.logentry;
193             }
194             if (p instanceof ITrackProvider) {
195                 type = ContentType.track;
196             }
197             if (type != null) {
198                 for (ICustomObject sel : p.getObjects()) {
199                     if (searchInputIsPresent.get()) {
200                         if (hits.containsKey(type.name().toLowerCase() + "_"
201                                 + sel.idProperty().getValue())) {
202                             ((AbstractCustomObject) sel).unmask();
203                         } else {
204                             ((AbstractCustomObject) sel).mask();
205                         }
206                     } else {
207                         ((AbstractCustomObject) sel).unmask();
208                     }
209                 }
210             }
211         }
212     }
```

Listing 13: Vorgehensweise der Kennzeichnung von Treffern

Beim Kennzeichnen muss jedes einzelne „`AbstractCustomObject`“ überprüft werden, ob es ein Treffer ist. Es ist ein Treffer, wenn in der Liste ein Element mit der gleichen ID und demselben Typ existiert. Es wird also aus der Identifikationsnummer des

Objekts und dem Typ ein Schlüsselwert generiert, dann nachgeschaut ob in der Trefferliste dieser Wert vorhanden ist.

Anschließend wird die Variable „*needToUpdate*“ auf „*true*“ gesetzt. Während „*isMasked*“-Property einen Treffer markiert, signalisiert „*needToUpdate*“ eine Änderung an den Suchergebnissen.

4.3 Präsentation der Suchergebnisse

Die Suchfunktion hat alle Treffer gekennzeichnet und die Änderung der Suchergebnisse signalisiert. Die Beobachter sind schließlich Bereiche der iLEXX-GUI. Diese können Ergebnisse unabhängig voneinander darstellen.

Jeder Bereich stellt Informationen, anders als die restlichen, dar. Aufgrund dieser Gegebenheit, muss für jeden Bereich eine eigene Implementierung der Ergebnispräsentation realisiert werden.

4.3.1 Ergebnispräsentation auf der TDA

Auf der TDA werden unter Anderem Tracks und Sperrzonen angezeigt. Beide werden mithilfe der Klasse „*Node*“ aus JavaFX visualisiert. „*Node*“-Objekte besitzen eine Property namens „*OpacityProperty*“ mit der die Durchsichtigkeit bestimmt wird. Gültige Werte für dieser Property sind zwischen 0 und 1, wobei 1 den „*Node*“-Objekt undurchsichtig und 0 vollständig durchsichtig macht.

Die Selektion von Treffern erfolgt durch die Maskierung der restlichen Informationen. In Abhängigkeit zu „`isMasked`“-Property muss die „`OpacityProperty`“ eingesetzt werden. Mit JavaFX können beide Properties aneinander gebunden werden. Hierfür werden die zuzuweisenden Werte wie folgt angegeben.

```
59
60          opacityProperty().bind(Bindings.createObjectBinding(() ->
61          (track.isMasked()) ? 0.4 : 1.0, track.trackStatusProperty(),
62                  track.isMaskedProperty()));
63
```

Listing 14: Einbindung der Durchsichtigkeitsparameter mit der „`isMasked`“-Property

Die folgende Bildschirmabbildung zeigt, wie die Suchergebnisse auf der TDA dargestellt werden:

Abbildung 17: Ergebnisdarstellung auf der TDA

4.3.2 Ergebnispräsentation auf der Track-Tabelle

Die Track-Tabelle präsentiert die Ergebnisse mit dem gleichen Prinzip, wie in der TDA. Bei diesem Bereich wird jeder Track durch eine ganze Zeile mit mehreren Zellen in der Tabelle präsentiert. Die Zeilen werden aber nicht durch die GUI-Komponente „*Node*“ dargestellt, weshalb auf die „*OpacityProperty*“ nicht zugegriffen werden kann, wie dies in der TDA der Fall war. Die Track-Tabelle wird mit der JavaFX-Komponente „*TableView*“ dargestellt. Mit dieser Komponente wird über „*RowFactory*“ auf eine ganze Zeile zugegriffen, damit die Durchsichtigkeit bestimmt werden kann.

```
219         trackTable.setRowFactory((TreeTableView<AbstractTrack> param) -> {
220             return new TreeTableRow<AbstractTrack>() {
221                 @Override
222                 protected void updateItem(AbstractTrack track, boolean empty) {
223                     super.updateItem(track, empty);
224                     if (track != null && !empty && track.isMasked()) {
225                         setStyle("-fx-opacity: 0.25");
226                     } else {
227                         setStyle("");
228                     }
229                 }
230             };
231         });
```

Listing 15: Implementierung eigener „RowFactory“ zur Darstellung der Suchergebnisse auf der Track-Tabelle

Die Treffer werden zusätzlich gruppiert und oben eingereiht. Hierzu wird das Sortieren in der Tabelle erweitert. Ein neuer „`comparator`“ wird implementiert. Der Vergleich erfolgt über die Variable „*isMasked*“. Besitzen zwei Tracks den gleichen Wert, so wird mit dem ursprünglichen „*Comparator*“ der Tabelle verglichen.

```
94 Comparator<TreeItem<AbstractTrack>> modifiedTrackTableComperator =
95   new Comparator<TreeItem<AbstractTrack>>() {
96    public int compare(TreeItem<AbstractTrack> o1, TreeItem<AbstractTrack> o2){
97             if (o1.getValue() != o2.getValue()) {
98                 if (o1.getValue() == null) {
99                     return 1;
00                 }
01                 if (o2.getValue() == null) {
02                     return -1;
03                 }
04                 if (!o1.getValue().isMasked() && o2.getValue().isMasked()) {
05                     return -1;
06                 }
07                 if (o1.getValue().isMasked() && !o2.getValue().isMasked()) {
08                     return 1;
09                 }
10             } else {
11                 return 0;
12             }
13             if(trackTable.getComparator() == null ){
14                 return 0;
15             }else{
16                 return trackTable.getComparator().compare(o1, o2);
17             }
18         }
19     };
20
```

Listing 16: Erweiterung des „*Comparator*“ der Track-Tabelle.

Die folgende Bildschirmabbildung zeigt, wie die Suchergebnisse auf der Track-Tabelle dargestellt werden:

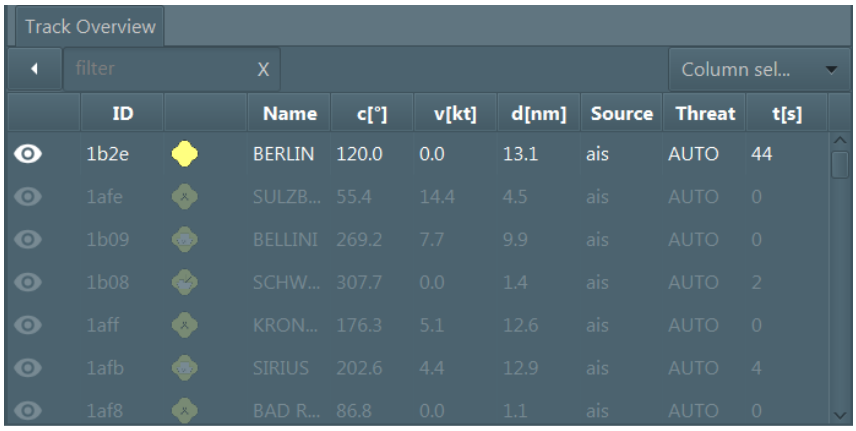

Abbildung 18: Ergebnisdarstellung auf der Tracktabelle

4.3.3 Ergebnispräsentation auf der Polarpanel-View

Die Präsentation auf der Polarpanel-View ist analog zu der Darstellung in der Track-Tabelle. Zuerst werden die einzelnen Darstellungen der Tracks angepasst und dann die Sortierbedingung erweitert.

```
14  private static final Comparator<Node> POLAR_PANEL_COMPERATOR =
15      new Comparator<Node>() {
16          @Override
17          public int compare(Node n1, Node n2) {
18            PolarPanel p1 = (PolarPanel) n1;
19            PolarPanel p2 = (PolarPanel) n2;
20            double threatLvl1 = p1.getTrack().getThreatLevel();
21            double threatLvl2 = p2.getTrack().getThreatLevel();
22            if (!p1.getTrack().isMasked() && p2.getTrack().isMasked()) {
23                return -1;
24             } else if (p1.getTrack().isMasked() && !p2.getTrack().isMasked()) {
25                return 1;
26              }
27             return threatLvl1 < threatLvl2 ? 1 : threatLvl1 == threatLvl2 ? 0 : -1;
28            }
29      };
30
```

Listing 17: Erweiterung des „*Comparator*“ der Polar-Panel-View

Die folgende Bildschirmabbildung zeigt, wie die Suchergebnisse auf der Polarpanel-View dargestellt werden:

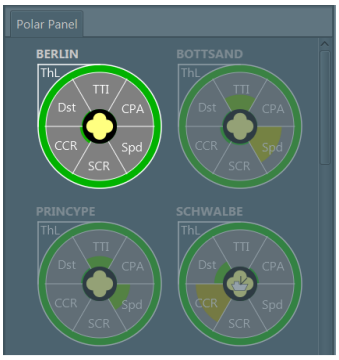

Abbildung 19: Ergebnisdarstellung auf der Polarpanel-View

4.3.4 Ergebnispräsentation der Schriftliche Nachrichten

Auf der Operator-Log-View werden schriftliche Nachrichten, wie bei anderen Bereichen der GUI, zuerst maskiert.

```
221         message.opacityProperty().bind(Bindings.createObjectBinding(() ->
222         (addedItem.isMasked() ? 0.4 : 1.0), addedItem.isMaskedProperty()));
223
```

Listing 18: Quellcode zeigt Treffer und maskiert die restlichen Nachrichten.

Bei diesem Bereich wandern bei jeder neuen Nachricht die vorherigen Nachrichten nach oben. Es ist also notwendig auf die, nach oben geschobenen, Nachrichten hinzuweisen, falls ein Treffer unter ihnen ist. Dazu werden rechts auf dem Rollbalken Rechtecke als Hinweis auf weitere Treffer angegeben.

Bei dieser Verwendung weist der Einsatz von „*isMasked*“-Property ein fehlerhaftes Verhalten auf. Folgendes Szenario zeigt den Fehler: Eine Suchabfrage wurde ausgeführt und die Treffer gekennzeichnet. Auf abgedeckte Nachrichten wurde mit Rechtecken hingewiesen. Der Nutzer löscht seine Sucheingabe. Alle Nachrichten sind wieder sichtbar, aber die Rechtecke bleiben, weil die „*isMasked*“ bei den Treffern nicht verändert wurde.

Für diesen Bereich der iLEXX-GUI reicht also die Einbindung der „`isMasked`“-Property mit dem „`OpacityProperty`“ nicht mehr aus. Die Operator-Log-View muss informiert

werden, wenn die Suchergebnisse sich ändern. Dies ist mit „*NeedToUpdate*“-Property möglich. In der Operator-Log-View wird die „*NeedToUpdate*“-Property und gegebenenfalls werden die Rechtecke neu gezeichnet.

```
166 if (searchEngine != null) {
167   searchEngine.needToUpdate().addListener(new ChangeListener<Boolean>() {
168       @Override
169       public void changed(
170         ObservableValue<? extends Boolean> observable,Boolean oldValue, Boolean newValue) {
171           if (newValue) {
172               Rectangle rec;
173               Iterator<LogbookMessage> iterator = logentries.values().iterator();
174               hitRectangles.clear();
175               while (iterator.hasNext()) {
176                LogbookMessage logbookmessage = iterator.next();
177                if (searchEngine.inputIsPresent().get()) {
178                 if (!logbookmessage.getLogbookItem().isMasked()) {
179                   double listPos = dataModel.indexOf(logbookmessage);
180                   rec = new Rectangle();
181                   double hitPos = listPos / (dataModel.size());
182                   rec.setLayoutY(entryListTrack.heightProperty().get() * hitPos);
183                   rec.setWidth(entryListTrack.getWidth() - (entryListTrack.getWidth() * .2));
184                   rec.setLayoutX((entryListTrack.getWidth() * .2) / 2);
185                   rec.setHeight(entryListTrack.heightProperty().get() / dataModel.size());
186                   rec.setFill(Color.WHITE);
187                   hitRectangles.add(rec);
188                   }
189                    } else {
190                                break;
191                           }
192                      }
193                      Platform.runLater(new Runnable() {
194                          @Override
195                          public void run() {
196                              hitsPane.getChildren().clear();
197                              for (Rectangle r : hitRectangles) {
198                                  hitsPane.getChildren().add(r);
199                              }
200                          }
201                      });
202                  }
203              }
204          });
205      }
206   }
```

Listing 19: Quellcode zum Kennzeichen von Treffer-Nachrichten auf dem Rollbalken

Die folgende Bildschirmabbildung zeigt, wie die Suchergebnisse auf der Operator-Log-View dargestellt werden:

Abbildung 20: Ergebnisdarstellung auf der Operator Log View

4.3.5 Alternative Realisierung der Suchfunktion

In den vorherigen Abschnitten wurde dokumentiert, wie die Suchfunktion implementiert wurde. In diesem Abschnitt werden relevante *Implementierungsmöglichkeiten* und *alternative Wege* demonstriert. Die Implementierungsmöglichkeiten veranschaulichen welche Optionen zur Verfügung standen, um dasselbe Ergebnis zu erzielen. Die Alternative Wege stellen Darstellungsmöglichkeiten vor, die während der Entwicklung zur Auswahl standen.

Implementierungsmöglichkeiten

Daten, wie Tracks, Sperrzonen und Nachrichten, werden für die Indexierung von den Daten-Providern an die Suchfunktion übergegeben. Die Suchfunktion extrahiert dann die relevanten Informationen mit objektspezifischen Dokumenten (4.1.2). Alternativ könnten Informationen aus den Daten in einer Datei (XML oder Yaml) geschrieben werden. Die entstandene Datei kann anschließend mit fertigimplementierten Lösungen, wie Tika, indexiert werden. Der Vorteil dieser Implementierungsmöglichkeit besteht darin, mehr Flexibilität im Indexierungsmechanismus zu haben. Der Nachteil bei der Umwandlung in die entsprechende Datei äußert sich in einer längeren Bearbeitungszeit und zusätzlich benötigtem Speicherplatz.

Beim Indexieren wurde der Feldtyp „*TextField*“ verwendet (Listing 6). Es besteht aber die Möglichkeit, andere Feldtypen wie „*StringField*“ oder „*TermField*“ zu benutzen. Die Nutzung eines anderen Feldtyps außer „*TextField*“ kann Änderungen bis auf die Verarbeitung der gesammelten Dokumente(4.2.4) hervorrufen. Der Grund hierfür ist der kleine Unterschied zwischen den genannten Feldtypen. Beispielsweise wird mit „*TextField*“ der Wert indexiert und gespeichert, während „*StringField*“ den Wert nur indexiert aber nicht speichert.

Im Abschnitt 4.2.2 lag die Vermutung vor, dass die Eingabe von Sonderzeichen bei der Suchfunktion zu einem Fehlverhalten führen kann. Daher wurden diese Zeichen bei der Bildung der Suchanfrage eliminiert (Listing 9). Die Behandlung von Sonderzeichen wäre mit einem selbstdefiniten Analyzer auch machbar. Der eigene Analyzer muss dann die Zeichen, sowohl beim Indexieren als auch beim Durchsuchen, durch Schlüsselwörter ersetzen. Die Schlüsselwörter lösen dann keine Fehlermeldungen aus und werden nicht als reguläre Ausdrücke von Lucene interpretiert.

Die Suchabfrage wurde mit „*StandardQueryParser*“ erstellt. Lucene bietet viele Möglichkeiten zur Erstellung von Suchabfragen. Die Erstellung derselben Suchabfrage mit einem anderen „*QueryParser*“, oder einer Zusammenstellung von Objekten aus den Suchabfrageklassen (Abbildung 6), wäre auch möglich.

Die Darstellung der Ergebnisse nach dem Kennzeichen durch die Suchfunktion kann mit dem Observer-Pattern realisiert werden. Die Verwendung von „*javafx.beans.binding.Bindings*“ und „*javafx.beans.value.ChangeListener*“ ersparte jedoch die Anwendung des Observer-Musters.

Mehrere Angaben („*SearchCondition*“ „*SearchField*“) waren bei der Filterfunktion erforderlich. Es besteht die Möglichkeit diese Angaben in selbstdefinierte reguläre ausdrücke zu formulieren. Dazu müssen Mechanismen zur Identifikation derartiger Ausdrücke und zur Umwandlung der Sucheingabe in die entsprechenden „*Query*“-Objekte implementiert werden. Hierfür kann wiederum derselbe implementierte Analyzer wiederverwendet werden, der die Sonderzeichen bei der Sucheingabe behandelt.

Alternative Wege

Die Ergebnisdarstellung erfolgt durch das Ausblenden der Distraktoren. Die Ergebnisse können aber auch auf eine andere Art und Weise präsentiert werden. Lucene stellt APIs für die Hervorhebung von Termen zur Verfügung. Die einzelnen Übereinstimmungen können beispielsweise auf der iLEXX-GUI markiert werden, ohne die Distraktoren auszublenden. Eine weitere Alternative besteht darin, die Distraktoren vollständig auszublenden. Hierbei wird allerdings riskiert, dass relevante Informationen zur Beurteilung des maritimen Lagebildes ausgeblendet werden.

5 Evaluation

Fehler, die nach der Freigabe bzw. Abnahme entdeckt werden, verursachen deutlich höhere Kosten als Fehler die während des Integrations- bzw. Systemtests entdeckt werden. Daher wird die entwickelte Suchfunktion in diesem Kapitel evaluiert und auf Fehler hin untersucht.

Die Evaluation erfolgt mittels eines automatisierten Tests und eines Usability-Tests. Beide Tests untersuchen sowohl die funktionalen als auch die qualitativen Anforderungen der Suchfunktion, welche bereits in Abschnitt 3.1.3 bereits definiert wurden.

5.1 Evaluation durch automatisiertes Testen

Das Verhalten der Suchfunktion muss bei verschiedenen Eingaben getestet und die daraus resultierenden Suchergebnisse geprüft werden. Dies muss gleichermaßen sowohl für Tracks, als auch für Sperrzonen und Nachrichten durchgeführt werden. Die Anzahl der potentiellen Sucheingaben und der produzierbaren Suchergebnisse sind dadurch unzählbar groß.

Die dafür benötigten Testfälle sind manuell mit unrealistisch großem Aufwand verbunden, welcher verhindert, dass zuverlässige Ergebnisse produziert werden. Bereits bei der manuellen Durchführung einer Aufgabe erhöht sich ab dem zehnten Mal die Anzahl der möglichen Fehler drastisch. Sie können durch Frust und/oder Konzentrationsmangel entstehen (Baumgartner et al. 2015, S. 6). Daher muss die Suchfunktion automatisch getestet werden. Die wesentlichen Vorteile beim automatisierten Testen sind die kurzen Laufzeiten bei den Testdurchläufen und die daraus resultierenden zuverlässigen Ergebnisse. Das automatisierte Testen ermöglicht außerdem eine genaue Messung der benötigten Laufzeit während der Durchsuchung. Somit kann die durchschnittliche Laufzeit ermittelt und die Anforderung aus den Rahmenbedingungen begutachtet werden.

Bei dem beabsichtigten Test wird die Funktion als Ganzes getestet und nicht die einzelnen Klassen bzw. Methoden. Diese Testform wird Komponenten- bzw. Integrationstest genannt und kann mit zahlreichen xUnits-frameworks durchgeführt werden, z.B. TestFX (Baumgartner et al. 2013, S.146). Jedoch sind die Auswahl und die Einsetzung solcher Tools mit hohem Zeitaufwand verbunden (Baumgartner et al. 2013, S. 21). Daher wird die Suchfunktion mittels eines vom Verfasser dieser Arbeit entwickelten Komponententests untersucht.

5.1.1 Automatisiertes Testen

Mit diesem Test wurden Suchanfragen, wie in den Anwendungsfällen beschrieben, simuliert. Für die Sucheingabe wird Text aus Tracks, Sperrzonen oder Nachrichten mit dem Zufallsprinzip generiert. Der generierte Text ist eine alphanumerische Zeichenkette. Für die Sucheingabe wird entweder der gesamte, oder nur ein Teil des Textes verwendet. Die Suchanfragen werden je nach festgestellten Zeitabständen periodisch ausgeführt. Nach jeder Ausführung werden die Ergebnisse auf Korrektheit geprüft und gegebenenfalls wird eine Fehlermeldung ausgegeben.

Für die periodische Ausführung wird die Klasse „*ScheduledExecutorService*“ genutzt. Ein Objekt aus dieser Klasse kann eine bestimmbare Anzahl an Threads nebenläufig ausführen. Für den Test wird ein einziger Thread wiederholt ausgeführt. Der Thread wählt zufällig einen Track aus und nutzt einen Wert aus seinen Eigenschaften als Sucheingabe. Das gleiche Verfahren gilt für die Sperrzonen und die Nachrichten.

Bei jeder simulierter Suchanfrage wird die benötigte Laufzeit während der Ausführung berechnet und ausgegeben. Diese Werte werden bei der Auswertung, wie in Abschnitt 5.1.2 beschrieben, verwendet.

5.1.2 Auswertung der Ergebnisse

Die Ausführung von Suchanfragen alle halbe Sekunde weist keine Fehler auf. Durchschnittlich dauerte die Bearbeitung der Suchanfrage 44 Millisekunden. Folgende Tabelle visualisiert die gemessenen Statistiken.

	Name	***ID***	***Breitengrad***	***Längengrad***	***Sperrzonen***	***Nachrichten***	***Total***
Laufzeit in ms	119	54	103	39	40	40	
	90	47	53	31	114	28	
	62	51	43	79	58	37	
	64	71	56	73	39	47	
	59	82	62	43	47	26	
	65	55	84	54	44	77	
	109	96	57	53	53	82	
	65	82	47	43	40	44	
	58	46	46	46	33	27	
	29	42	37	29	31	38	
	47	50	34	26	20	32	
	39	16	27	27	39	25	
	84	37	73	37	40	28	
	29	23	24	63	25	20	
	26	14	37	18	28	22	
	59	27	20	13	23	21	
	31	17	51	38	18	39	
	23	15	45	15	41	15	
	35	55	167	28	15	23	
	27	19	29	19	27	44	
Durchschnitt	*56*	*44,95*	*54,75*	*38,7*	*38,75*	*35,75*	**44,8166667**

Tabelle 5: Aufgezeichnete Zeitdauer einer Suchanfragen pro 500 Millisekunden

Bei der Ausführung von Suchanfragen, in einer periodischen Zeitspanne von einer Sekunde oder länger, wurden folgende Fehlermeldung ausgegeben:

```
org.apache.lucene.util.ThreadInterruptedException:
java.lang.InterruptedException at
org.apache.lucene.index.DocumentsWriterFlushControl.waitForFlush
(DocumentsWriterFlushControl.java:261)
```

Dieser Fehler entsteht, wenn ein Thread mit „*wait()*“ angehalten wird und beim Aufruf zur Fortsetzung mit „*notify()*“ nicht mehr erreichbar ist. Dieser Fall ist aufgetreten während ein Thread eine Suchanfrage ausgeführt hat und beim Lesen des Indexes angehalten wurde. Das Öffnen hat zu lange gedauert und währenddessen wurde der Thread unterbrochen bzw. neu gestartet. Dieses Problem ist an den hohen Änderungen im Index zurückzuführen und wurde im Kapitel 2.2.3 bereits besprochen.

5.1.3 Fehlerbeseitigung

Die Standardvorgehensweise, wie der Index gelesen wird, hat sich als ungeeignet erwiesen. Lucene bietet einen anderen Weg den Index in Echtzeit zu verwalten. Statt den „*Reader*“ manuell zu öffnen, kann die Klasse „*SearcherManager.java*“ den Index zuverlässig von mehreren Threads lesen lassen. Diese Klasse wird folgendermaßen eingesetzt.

Die Klasse wird zuerst instanziiert. Diese Instanz stellt den „*IndexSearcher*“ zur Verfügung. Jeder Thread der eine Suchanfrage ausführen will, öffnet den Index nicht mehr selbst, sondern nutzt den vom Manager zur Verfügung gestellten „*IndexSearcher*“. Threads die Änderungen an dem Index vornehmen, sorgen dafür, dass diese Instanz stets aktuell bleibt. Sie rufen nach jeder Indexverarbeitung die Methode „*SearcherManager.maybedorefresh()*“ auf. Diese Methode ist für Multithreading in Echtzeit ausgelegt. Sie wird ein einziges Mal ausgeführt, falls sie von mehreren Threads gleichzeitig aufgerufen wird.

Folgende Änderungen wurden an der Suchfunktion vorgenommen:

```
256         try {
257             s = manager.acquire();
258             try {
259                 try {
260                     hits = s.search(query, hitsPerPage).scoreDocs;
261                     documents = new Document[hits.length];
262                     for (int i = 0; i < hits.length; ++i) {
263 
264                         documents[i] = s.doc(hits[i].doc);
265                     }
266                 } catch (final IOException e) {
267                     e.printStackTrace();
268                 }
269             } finally {
270                 manager.release(s);
271             }
272 
273         } catch (IOException e1) {
274             e1.printStackTrace();
275         }
276         s = null;
```

Listing 20: Die Verwendung eines „*SearcherManager*“

Durch diese Änderung im Quellcode ist eine Verbesserung in Bezug auf die Laufzeit der Suchanfragen festgestellt worden. Durchschnittlich dauerte die Bearbeitung der Suchanfrage 10 Millisekunden.

	Name	***ID***	***Breitengrad***	***Längengrad***	***Sperrzonen***	***Logbook***	***Total***
Laufzeit in ms	223	6	6	5	5	6	
	24	7	8	5	5	7	
	22	54	7	5	5	6	
	20	5	5	10	4	10	
	10	8	5	5	15	9	
	13	7	6	6	6	10	
	9	25	5	5	5	10	
	11	5	16	8	5	13	
	10	6	5	5	8	11	
	16	6	7	7	20	5	
	11	6	5	7	23	10	
	7	6	24	6	5	7	

	8	6	8	5	5	5	
	17	4	6	5	4	13	
	11	6	7	8	5	11	
	11	5	5	8	5	5	
	12	14	5	8	5	5	
	9	4	5	7	5	17	
	8	7	6	8	5	9	
	5	5	5	5	17	5	
Durchschnitt	*22,85*	*9,6*	*7,3*	*6,4*	*7,85*	*8,7*	**10,45**

Tabelle 6:Tabelle mit aktuellen Laufzeiten der Suchanfragen

5.2 Evaluierung der Usability (Gebrauchstauglichkeit)

Nach der Implementierung wurden die Umstände der Suchfunktion nach der Integration untersucht.

Die iLEXX-GUI bietet mehrere Bereiche zur Darstellung von Information an. Der Operator kann auswählen, welche Bereiche eingeblendet werden sollen. Die Durchführung der Operatoraufgaben erfordert einen ständigen Blickwechsel zwischen den Bereichen. Die Nutzung der integrierten Suchfunktion erfordert einen zusätzlichen Wechsel. Dieser Sachverhalt kann, wie in der Problemstellung beschrieben wurde, selektive visuelle Aufmerksamkeitsdefizite beim Operator auslösen.

Andererseits ist der Operator, nach mehrjähriger Nutzung des Systems, mit den Arbeitsschritten und der Anordnung der iLEXX-GUI vertraut geworden. Er hat einen gewissen Arbeitsfluss entwickelt.

Die genannten Gegebenheiten können die Handlung des Operators und seinen Arbeitsfluss beeinträchtigen. Dieses Verhalten wird in diesem Kapitel mit einem Usability-Test begutachtet. Der Usability-Test besteht aus mehreren Phasen. Im Folgenden werden die Planungs-, Vorbereitungs- und Auswertungsphase dokumentiert. Alle Phasen sind an der Empfehlung von Carol M. Barnum angelehnt (Barnum 2011).

5.2.1 Planung

Bei der Planungsphase werden mehrere Entscheidungen getroffen, die bei der Organisation unerlässlich sind. Außerdem trägt diese Phase bei der Aufwandsschätzung des Usability-Tests bei. Deswegen wurde eine Besprechung mit dem Team durchgeführt, um die signifikanten Entscheidungen zu treffen. Diese betreffen die *Ziele*, den *Ort*, den *Zeitplan und die Szenarien für die Testdurchführung*.

Ziele

Die Modalität der Zeit und des Budgets schränken den Umfang ein, daher müssen Ziele definiert werden (Barnum 2011, S. 107). Die vereinbarten Ziele in diesem Test dienen dazu, die ausdrückliche Nutzererfahrung der Suchfunktion zu ermitteln und nicht die der gesamten iLEXX-GUI.

In erster Linie wird mit dem Usability-Test die Handlung des Nutzers untersucht. Im Allgemeinen tendieren die Nutzer dazu die Produkte falsch zu interpretieren und auf inkorrekter Weise zu benutzen (Nielson 1993, S.10). Diese Gegebenheit erfordert die Untersuchung zweier weiterer Punkte bei der Suchfunktion. Einerseits wird geprüft ob die Suchergebnisse für den Nutzer verständlich sind, andererseits inwiefern die Suchfunktion eine nützliche Hilfestellung darstellt.

- In der Problemstellung (1.2) wurde geschildert, dass der Operator bei seiner Aufgabenstellung nicht gestört werden soll und dies oberste Priorität hat. Daher muss dies als erstes getestet werden. Hierzu müssen gewisse Vorarbeiten durchgeführt werden. Die Testperson muss die Handlung des Operators imitieren. Die Imitierung wird mit der gleichen Aufgabenstellung erlangt. Diese Handlung wird mit den passenden Szenarien auf eine Beeinträchtigung hin untersucht. Dieses Ziel wird im Folgenden mit dem Wort „Beeinträchtigung“ referenziert.
- Die Nutzung der Suche soll für den Operator leicht verständlich und nachvollziehbar sein. Aber auch die Darstellung der Suchergebnisse soll intuitiv verstanden werden. Dieses Ziel wird im Folgenden mit dem Wort „Nachvollziehbarkeit“ referenziert.
- Das letzte definierte Ziel ist, herauszufinden inwiefern die Suchfunktion für den Operator eine Hilfestellung darstellt. Bringt die Suchfunktion einen bedeutsamen Vorteil mit oder erschwert sie die Bedienung. Dieses Ziel wird mit dem Schlüsselwort „Unterstützung“ referenziert.

Sobald die Ziele definiert wurden, muss der Ort für die Testdurchführung festgelegt werden (Barnum 2011, S 111).

Ort

Hierbei wurde die Auswahl auf den Bundeswehrdemonstrationsraum „BwDemo-Raum“ gelegt. Dieser Raum wurde für Test- und Demozwecke der Bundeswehr mit speziellen Hardware Komponenten ausgestattet. Mehrere Rechner und Monitore in unterschiedlichen Größen sind im Raum aufgebaut. Auf den Rechnern ist der aktuelle Prototyp des iLEXX-Systems installiert. Auf der iLEXX-GUI werden live Daten und aufgezeichnete Daten visualisiert bzw. abgespielt. Dies simuliert eine realitätsnahe Umgebung. Darüber hinaus ermöglicht die Räumlichkeit eine gleichzeitige Beobachtung der Probanden und ein besseres Moderieren des Usability Tests.

Zeitplan

Der Zeitplan liefert sowohl Informationen über das Zeitmanagement als auch Details über die Probanden und ihren Aufgaben während des Tests. Die Durchführung des Tests wurde auf einen Tag festgelegt und die Anzahl der Probanden auf drei Personen begrenzt. Ein Test dauert planmäßig dreißig Minuten. Vor der tatsächlichen Testdurchführung wird eine kurze Zeitspanne für die Vorbereitung der Geräte vorbehalten. Nach jedem Durchlauf mit einem Probanden wird eine kleine Pause eingeplant um den nächsten Testdurchlauf vorzubereiten. Am Ende der Testdurchführung werden die gewonnen Erkenntnisse gesichert und die Geräte ausgeschaltet.

Die Notizen des Usability-Tests wurden in der folgenden Schablone zusammengefasst:

Planungsnotizen für den Usability-Test	
Umfang	Suchfunktion der iLEXX-GUI
Ziele	*Beeinträchtigung*: In Erfahrung bringen, ob die Handlung des Operators gestört wurde.
	Nachvollziehbarkeit: Sicherstellen, dass die Suchergebnisse für den Operator Nachvollziehbar sind.
	Unterstützung: Feststellen inwiefern die Suchfunktion den Operator bei seiner Aufgabe unterstützt.
Testkriterien	Verständlichkeit, Nachvollziehbarkeit, Mehrwertigkeit, Unterstützungs- und Beeinträchtigungsgrad

Messvorgehen	Bearbeitungsdauer und Bewertung der Testpersonen
Ort der Testdurchführung	Bundeswehr Demo-raum im Fraunhofer IAIS
Zielgruppe der Testpersonen	Allgemeine Computerkenntnisse erforderlich.
	Kenntnisse über die iLEXX-GUI sind wünschenswert aber nicht erforderlich
Ansporn	Muffins
Datum der Testdurchführung	Donnerstag 30/03/2017
Testplan	9:30 - 10:30 Vorbereitung
	10:30 - 11:00 Testperson 1
	11:00 - 11:15 Pause
	11:15 - 11:45 Testperson 2
	11:45 - 12:00 Pause
	12:00 - 12:30 Testperson 3
	12:30 - 13:00 Abbau und Sicherung der Testergebnisse
Bearbeitungsdauer der Ergebnisse	Eine Woche nach der Durchführung

Tabelle 7: Planungsnotizen für den Usability-Test (nach Barnum 2011, S. 143-145).

Szenarien für die Testdurchführung

Der letzte Schritt der Planungsphase ist die *Ausformulierung von Szenarien*. Die Ausformulierung orientiert sich an den oben beschriebenen Zielen. Die Szenarien beinhalten vier Bestandteile.

Der erste Bestandteil ordnet jedes Szenario einem bestimmten Ziel zu. Dies dient dazu den Usability Test in dem geplanten Umfang zu halten und den entsprechenden Aspekt zu untersuchen.

Der zweite Bestandteil beinhaltet eine Frage, die aus dem Ziel abgeleitet ist und eines der Prüfkriterien aus den Planungsnotizen abdeckt.

Der dritte Bestandteil beinhaltet eine Aufgabe. Die Aufgabe beschreibt den Testablauf und legt fest welche Ergebnisse mit dem Szenario erzielt werden sollen. Sie bezieht sich immer auf einem konkreten Anwendungsfall aus dem Kapitel 3.1.2

Der letzte und vierte Bestandteil ist die Beschreibung des Szenarios. Im Folgenden werden fünf Szenarien beschrieben, die für die Testdurchführung verwendet wurden:

Ziel: Beeinträchtigung
Frage: Können Beeinträchtigungen in der Handlung und dem Arbeitsfluss des Operators beobachtet werden?
Aufgabe: Der Moderator nennt der Testperson eine Aufgabe. Die Testperson beginnt mit der Durchführung und darf diese Aufgabe nicht unterbrechen. Der Moderator wartet bis die Testperson die gewünschte Handlung entwickelt hat. Der Moderator teilt der

Testperson Suchaufträge zu und versucht die verursachte Beeinträchtigung zu beobachten. Das richtige Verhalten: Die Testperson unterbricht die Suchaufträge um diese Aufgabe zu erledigen.
Szenario: Der Moderator bittet die Testperson das maritime Lagebild mit der GUI zu überwachen. Die Testperson muss verdächtiges Verhalten kontinuierlich verarbeiten. Der Moderator teilt der Testperson den Vorrang der Aufgabe mit sobald sie den gewünschten Arbeitsfluss entwickelt hat. Der Moderator bittet die Testperson mehrmals einen bestimmten Track mit unterschiedlichen Merkmalen zu suchen.

Ziel: Nachvollziehbarkeit
Frage: Kann die Testperson die Suchergebnisse verstehen?
Aufgabe: Der Moderator nennt der Testperson ein Zugehörigkeitstyp (Affiliation Name). Die Testperson trägt die jeweilige Affiliation in das Suchfeld ein. Die Testperson wird gebeten die Suchergebnisse zu beschreiben. Die richtige Antwort: Alle Tracks mit der entsprechenden Affiliation wurden selektiert.
Szenario: Der Moderator nennt eine Affiliation bspw. „Combatant, Fisher oder Tanker“. Die Testperson gibt die Affiliation in das Suchfeld ein. Der Moderator bittet die Testperson das Ergebnis zu erläutern. Die Testperson drückt die Ergebnisse mit den eigenen Worten aus.

Ziel: Nachvollziehbarkeit
Frage: Kann die Testperson die Suchergebnisse verstehen?
Aufgabe: Der Moderator nennt der Testperson die Koordinaten eines bestimmten Schiffes. Die Testperson trägt die Koordinaten in das Suchfeld ein. Die Testperson wird gebeten die Suchergebnisse zu beschreiben. Die richtige Antwort: Das sich im eingegebenen Punkt befindende Schiff wurde selektiert.
Szenario: Der Moderator nennt die Koordinaten eines Schiffes. Die Testperson gibt die Koordinaten in das Suchfeld ein. Der Moderator bittet die Testperson das Ergebnis

zu erläutern. Die Testperson drückt das Ergebnis mit eigenen Worten aus.

Ziel: Unterstützung
Frage: Wie lange braucht die Testperson durchschnittlich **ohne** Nutzung der Suchfunktion, um ein Track zu finden?
Aufgabe: Die Testperson erhält einen Tracknamen und muss diesen Track **ohne** Suchfunktion orten. Die Ortung erfolgt, indem die Testperson den Track selektiert und mittig auf der Karte platziert.
Szenario: Der Moderator nennt drei Tracks. Die Testperson soll den Track mit dem entsprechenden Namen orten. Bei jedem Suchvorgang misst der Moderator die Dauer der Suche.
Notizen:

Ziel: Unterstützung
Frage: Wie lange braucht die Testperson durchschnittlich **mit** Nutzung der Suchfunktion, um ein Track zu finden?
Aufgabe: Die Testperson erhält einen Tracknamen und muss diesen Track **mit** Suchfunktion orten. Die Ortung erfolgt in dem die Testperson den Track selektiert und in die Mitte der Karte setzt.
Szenario: Der Moderator nennt drei Tracks. Die Testperson soll den Track mit dem entsprechenden Namen orten. Bei jedem Suchvorgang misst der Moderator die Suchdauer.

5.2.2 Vorbereitung

Die Vorbereitungsphase vervollständigt die Planung. In dieser Phase werden die erforderlichen Ressourcen und Medien vorbereitet und der geplante Usability-Test geprüft. Die Ressourcen sind die *Testpersonen* und die Medien sind die *Aufzeichnung* und die *Hilfsmittel*.

Testpersonen

Das Wissen über das Nutzerverhalten, während der Nutzung eines Produkts, erfordert menschliches Eingreifen (Nielson 1993, S.165).

Im Rahmen dieser Arbeit war es nicht gestattet, Externen den Zutritt zum Fraunhofer IAIS zu gewährleisten. Die Teilnahme von Testpersonen außerhalb der Fraunhofer Organisation, ist ausschließlich mit der Beantragung einer gesonderten Erlaubnis möglich. Um diesen Verwaltungsaufwand zu reduzieren, wurden stattdessen Testpersonen aus dem Fraunhofer IAIS (nur die interne Abteilung) für diesen Test zugelassen. Das Fraunhofer IAIS ist vertraut mit der Durchführung von Studien und Umfragen, welches den Vorteil hat, dass die potentiellen Testpersonen über die entsprechende Erfahrung verfügen. Beispielsweise war die Methodik des lauten Denkens (Think-Aloud) für sie bereits bekannt. Think-Aloud ist für die meisten Menschen ein unnatürliches Verhalten und es kann Ihnen schwer fallen dies umzusetzen (Barnum 2011, S.205).

Die potentiellen Testpersonen verfügten über die erforderlichen Vorkenntnisse, sodass diese Schwierigkeiten nicht aufkamen und der damit verbundene Aufklärungsaufwand ausgeschlossen werden konnte.

Nach der Planungsphase wurden die Details über den Usability-Test beschrieben. Die potentiellen Testpersonen wurden eingeladen und über das behandelte Thema, die Dauer des Tests und die wählbaren Termine informiert.

Am Ende konnten drei Testpersonen für den geplanten Testtag beschaffen werden.

Aufzeichnung:

Aufzeichnungen sind bei Usability Tests sehr vorteilhaft. Damit können Beobachtungen während der Testdurchführung abgespielt werden und die Usability besser untersucht werden. Aufzeichnungen können Ton-, aber auch Video-Aufnahmen sein. Beide Aufnahmearten müssen jedoch rechtlich geregelt sein. Im Rahmen dieser Arbeit durfte keine Aufzeichnung erfolgen.

Hilfsmittel

Während des Testdurchlaufs kann es hektisch werden. Um keine Aufgaben und Aktivitäten zu vergessen, ist eine *Checkliste* sehr nützlich. Diese Liste stellt sicher, dass alle Testdurchläufe (inklusive Reihenfolge) bei allen Testpersonen identisch sind.

Das Moderator-Skript sorgt dafür, dass die gleichen Informationen allen Testpersonen übermittelt werden. Eine implizite Vorbedingung für die Testdurchführung ist, dass alle Testpersonen über denselben Informationsstand verfügen.

Zur Dokumentation der Beobachtungen wird ein Fragebogen erstellt. Dies wird nach jeder Testsitzung von der entsprechenden Testperson ausgefüllt. Dabei befassen sich die Fragen mit den Kriterien der Dialoggestaltung (ISO 9241-110, 2008). Anschließend hat die Testperson die Möglichkeit ihr Feedback zu geben bzw. ihre Verbesserungs-vorschläge vorzutragen.

Testüberprüfung

Bevor der Test mit den Testpersonen durchgeführt wird, muss eine Test Review (Testüberprüfung) stattgefunden haben (Barnum 2011,S.188). Die Testüberprüfung untersucht den Test nach möglichen Problemen die bei der Durchführung auftreten können. Dies erfolgt frühzeitig vor dem geplanten Tag der Durchführung, wobei der Test anschließend mit einer einzelnen Testperson durchgeführt wird.

5.2.3 Auswertung

Nach der Vorbereitungsphase konnte der Usability-Test erfolgreich durchgeführt werden. Das Ergebnis ist eine Reihe von Dokumenten mit Notizen. Während der Auswertungsphase werden die Notizen in einer Tabelle *mit Beobachtungen* protokolliert. Anschließend werden die *positiven* und die *negativen Merkmale* hergeleitet und *mögliche Verbesserungen* vorgeschlagen werden (Barnum 2011, S270-275).

Tabelle mit Beobachtungen

Die notierten Anmerkungen werden nach Testpersonen gruppiert. Um die Lesbarkeit weiter zu erleichtern, werden die Anmerkungen nach Zielen aufgeteilt. Zu jedem Ziel wird die festgestellte Beobachtung eingetragen. Die eingetragene Beobachtung kann mit einem Kommentar erläutert werden.

Als Ergebnis dieser Auswertung und Protokollierung wurde folgendes Dokument erstellt:

Tabelle 8: Festgestellte Beobachtungen während der Testdurchführung

Testperson 1		
Kategorie	**Beobachtung**	**Kommentare**
Nachvollziehbarkeit	Die Testperson entdeckte die Treffer erst nach dem Hochscrollen der Track-Tabelle. Die Testperson wusste sofort, dass ein Track wegen Namenübereinstimmung selektiert wurde.	Okay! Alles ist durchsichtig geworden…
Unterstützung	Um einen Track **mithilfe** der Suchfunktion zu finden benötigte die Testperson im Durchschnitt **3,3** Sekunden.	Messzeiten von drei Suchvorgängen: 3, 4 und 4 Sekunden
	Um einen Track **ohne** Nutzung der Suchfunktion zu finden benötigte die Testperson im Durchschnitt **12,3** Sekunden.	Messzeiten von drei Suchvorgängen: 17, 10 und 10 Sekunden
Beeinträchtigung	Die Testperson konnte während der Suchanfrage ihre Hauptaufgabe makellos erledigen. Sie hat alle Alarmierungen gemeldet und konnte allen gefährlichen Tracks die entsprechende Affiliation zuordnen.	
Testperson 2		
Kategorie	**Beobachtung**	**Kommentare**

Nachvollziehbarkeit	Die Testperson wusste sofort, dass ein Track wegen Übereinstimmung seiner Affiliation mit der Sucheingabe, selektiert wurde.	
Unterstützung	Um einen Track **mithilfe** der Suchfunktion zu finden benötigte die Testperson im Durchschnitt **3,3** Sekunden.	Messzeiten von drei Suchvorgängen: 4, 3 und 3 Sekunden
	Um einen Track **ohne** Nutzung der Suchfunktion zu finden benötigte die Testperson im Durchschnitt **15,6** Sekunden.	Messzeiten von drei Suchvorgängen: 17,10 Sekunden und über 20 Sekunden
Beeinträchtigung	Die Testperson hat eine Alarmierung verpasst dann sehr spät entdeckt und gemeldet.	Die Alarmierung wurde ausgelöst während der Testperson eine vorherige Alarmierung gemeldet hat.
Testperson 3		
Kategorie	**Beobachtung**	**Kommentare**
Nachvollziehbarkeit	Die Testperson achtet mehr auf die maskierten Ergebnisse als auf die selektierten. Die Testperson hat die Trefferlosigkeit erkannt. Sie wusste, dass es bei den eingegebenen Zahlen um Koordinaten handelt und dass kein Track diese Koordinaten besitzt.	
Unterstützung	Um einen Track **mithilfe** der Suchfunktion zu finden benötigte die Testperson im Durchschnitt **4** Sekunden.	Messzeiten von drei Suchvorgängen: 5, 3 und 4 Sekunden
	Um einen Track **ohne** Nutzung der Suchfunktion zu finden benötigte die Testperson im Durchschnitt **12** Sekunden.	Messzeiten von drei Suchvorgängen: 17, 9 und 10 Sekunden
Beeinträchtigung	Die Testperson hat einen bedrohlichen Track nicht bemerkt. Die Affiliation dieses Tracks wurde erst sehr spät richtig angesetzt.	Dies lag an der hohen Anzahl Alarmierungen die behandelt werden mussten.

Die oben dokumentierten Beobachtungen wurden in einem Meeting besprochen. Darauf basierend konnte die folgende Einteilung beschlossen werden.

Positive Merkmale

Die Suchergebnisse waren für die Testpersonen nachvollziehbar. Sie alle konnten die Ergebnisse verstehen und korrekt beschreiben. Sie konnten aus den Ergebnissen herleiten, was die Eingabe bedeutete (zum Beispiel Koordinaten). Darüber hinaus wurden keine Missverständnisse bei Trefferlosigkeit beobachtet.

Die Testpersonen benötigten deutlich weniger Zeit wenn sie die Suchfunktion benutzt haben. Ohne die Nutzung der Suchfunktion haben die Testpersonen durchschnittlich drei bis fast fünf Mal länger gebraucht (siehe Tabelle 8).

Die Verwendung der Suchfunktion hat die Testpersonen bei der Hauptaufgabe nicht abgelenkt. Alle Alarmierungen und verdächtigen Tracks wurden während der Nutzung der Suchfunktion bemerkt und behandelt.

Negative Merkmale

Die Testpersonen wollten den aktuellen Stand der Suchergebnisse erfahren. Sie haben sich beispielsweise gefragt, wie viele Treffer gefunden wurden. Außerdem bevorzugten sie eine eindeutige Differenzierung zwischen einer erfolgreichen und einer erfolglosen Suchanfrage.

Beim Abbrechen der Suche bzw. beim Entfernen der Sucheingabe wünschten sich die Testpersonen die Möglichkeit dies mit einem einzigen Mausklick erledigen zu können.

Als die Testpersonen das Ergebnis beschreiben wollten, haben sie die Distraktoren vor den Treffern bemerkt. Die Treffer hatten also weniger Aufmerksamkeit erregt als die restlichen Informationen.

Die Testpersonen konnten nicht wissen welche Informationen können mit der Suchfunktion durchsucht werden. Es war nicht eindeutig, dass nur Tracks, Sperrzonen und Nachrichten durchsucht werden können. Deren Ansicht nach, hätten sie auch nach Bereichen der GUI oder Features suchen können.

Mögliche Verbesserungen

Der Verfasser dieser Arbeit konnte Lösungen für die Negativen Merkmale vorschlagen. Es gab folgende Verbesserungsvorschläge:

- Status der Suchergebnisse anzeigen lassen.
- Dem Operator die Möglichkeit anbieten, die Eingabe per Mausklick zu löschen.
- Den Fokus des Operators weniger auf die maskierten Ergebnisse und mehr auf die selektierten Ergebnisse lenken.
- Verdeutlichen, welche Informationen mit der Suchfunktion durchsucht werden können.

Im Meeting wurde auch entschieden, wie die Verbesserungen realisiert werden können.

5.2.4 Einarbeitung

Status der Suchergebnisse anzeigen

Zuerst muss zwischen drei Zuständen der Suchfunktion, die durch die Nutzereingabe bestimmt sind, unterschieden werden. Der Anfangszustand der Suche ist eine leere Sucheingabe. Mit dem Vorhandensein der Sucheingabe kann es zu einem Treffervorkommnis kommen oder eine Trefferlosigkeit ergeben.

Das Icon der Suchfunktion soll die Zustände der Suche mittels drei Farben visualisieren. Am Anfang ist die Lupe weiß. Nach Auftreten der Sucheingabe des Nutzers wird sie bei Treffern grün und falls keine Treffer erzielt wurden, rot. Um die Aussagekraft der Farben zu stärken, wird unter der Lupe die Anzahl der Treffer angezeigt. Diese Zahl kann 0 bis 99 sein. Im Falle einer größeren Anzahl an Ergebnissen wird dann „+99“ angezeigt.

Die Sucheingabe per Mausklick löschen

Zusätzlich wird dem Operator die Möglichkeit angeboten, die Sucheingabe per Mausklick zu löschen. Dazu wird ein Abbruch-Icon sichtbar, wenn eine Sucheingabe vorhanden ist. Das Anklicken des Icons löscht die Eingabe und bricht die Suche ab.

Abbildung 21: Verschiedene Zustände der Suchfunktion nach Implementierung der Verbesserungsvorschläge

6 Zusammenfassung

Die Entwicklung und die Integration einer Lucene-basierten Suchfunktion wurden im Rahmen dieser Arbeit erfolgreich abgeschlossen. Die Suchfunktion kann Daten indexieren, durchsuchen und in Echtzeit darstellen. Zusätzlich stellt sie die Ergebnisse interaktiv dar. Sie unterstützt die Entscheidungsfindung des Operators und bietet eine übersichtlichere Datendarstellung. Schlussendlich wurde die Handlung des Operators durch die Nutzung der Suchfunktion nicht beeinträchtigt.

Die Integration der Suchfunktion wurde in einem eigenen Plug-In realisiert. Dies ermöglicht das Entfernen und Ersetzen des Plug-Ins durch ein anderes, ohne die Features der iLEXX-GUI einzuschränken. Die integrierte Suchfunktion kann die Daten der iLEXX-GUI innerhalb von Millisekunden durchsuchen. Tatsächlich benötigen die Suchvorgänge durchschnittlich ein Zehntel der erlaubten Latenzzeit. Ebenfalls konnte eine schnellere Suchanfrage bei der Durchsuchung von Datierungen erstellt werden.

Aus Sicht des Entwicklerteams hat die Suchfunktion eine hohe Akzeptanz und eine gute Usability erwiesen. Aus den Ergebnissen des Usability-Tests geht hervor, dass die Suchfunktion von den Testpersonen intuitiv genutzt werden konnte. Es wurde kein einziges schwerwiegendes Problem entdeckt. Alle entdeckten Defizite sind als unerheblich klassifiziert worden, weil die Testpersonen dennoch die Aufgaben erfolgreich abschließen konnten.

Die Ziele der Arbeit wurden erfüllt. Durch den Usability-Test konnten jedoch Verbesserungsmöglichkeiten entdeckt werden. Diese wurden nur teilweise umgesetzt. Den Testpersonen war nicht klar, welche Informationen sie mit der Suchfunktion durchsuchen können. Der Nutzer sollte wissen ob er nur Tracks, Sperrzonen und/oder Nachrichten durchsuchen kann oder nach weiteren Daten. Darüber hinaus fielen den Testpersonen die Distraktoren eher auf, als die Treffer. Daher sollte die Aufmerksamkeit des Nutzers mehr auf die Treffer gelenkt werden.
Außerdem konnten die implementierten Grundlagen für die erweiterte Suchfunktion nicht getestet werden. Dazu müssten mehr Details über die Anforderungen vorhanden sein und eventuell eine Schnittstelle implementiert werden.

Es wurde für die iLEXX-GUI eine Schulung eingeplant, in der die Suchfunktion die oben genannten Defizite behoben werden kann.

7 Abbildungsverzeichnis

8 Tabellenverzeichnis

9 Listingsverzeichnis

10 Glossar

Aufmerksamkeitsblinzeln: ist das Übersehen von Änderungen, bei Aufgaben mit schnell aufeinanderfolgenden visuellen Darstellungen (Müller / Krummenacher / Schubert 2015, S. 71).

Autmatisiertes Testen: Testautomatisierung ist die Durchführung von ansonsten manuellen Testtätigkeiten durch Automaten(Bucsics et al. 2015, S. 7).

Crawler: Ein Crawler der sich die Seiten merkt und ausgewählte Informationen zu diesen Seiten in Datenbanken speichert(Fisher-Stabel / Gollmer 2016, S. 265).

Distraktor: Sind Fehl- oder Alternativinformationen, welche mit der Suchfunktion indexiert und nach einem Suchvorgang nicht als Treffer selektiert wurden.

Echtzeit: Einhaltung von vorgegebene Reaktionszeiten (Kanbach 2005, S. 210).

Operator: der Endnutzer der iLEXX-GUI.

OSGi-Services:

> „OSGi-Services sind einfache Java-Objekte (POJOs), die innerhalb eines Bundles erzeugt und an der zentralen OSGi Service Registry registriert werden. Durch die Verwendung von OSGi Services ist es damit möglich, Objekte über Bundle-Grenzen hinweg verfügbar zu machen.“(Wütherich et al. 2008,S. 93).

Peilung: Bestimmung einer Richtung bzw. eines Winkels bezgl. einer Bezugsrichtung für Flugzeuge oder Schiffe unterwegs. (Wolski S. 1084)

Race Condition: Wettlauf; Zustand, dass mehrere Prozesse im gleichen Speicherbereich ineinander verzahnt lesen und schreiben und das Ergebnis letztendlich durch Zufälle und Prioritäten und nicht allein durch den Algorithmus beeinflusst wird (Fisher / Hofer 2011, S.725).

Service Components: Eine Komponente aus der OSGi Plattform, die dafür zuständig ist Services bereitzustellen (Wütherich et al. 2008, S.201).

Think Aloud: Lautes Denken ist eine Methode zur Untersuchung des Verhaltens von Testpersonen, statt Rückschlüsse aus ihren Interaktionen zu ziehen, beschreiben die Testpersonen selbst was sie denken (Bernsen / Dybkjær 2009, S. 281).

Tika: ein Toolkit zur Extrahierung von Informationen aus Dateien (Gospodnetic / Hatcher / McCandless 2010, S. 236).

Usability: Gebrauchstauglichkeit; bestätigt, dass der Nutzer mit dem Design wie gewünscht interagiert (Cooper / Reimann 2003, S.156).

Usability-Testing: ist eine Sammlung von Techniken um die Nutzerinteraktionen nach bestimmten Kriterien zu messen (Cooper / Reimann 2003, S. 57).

Veränderungsblindheit: ist die Beeinträchtigung der Wahrnehmung von visuellen Änderungen durch bestimmte Ausrichtung des Nutzerfokus (Müller / Krummenacher / Schubert 2015, S. 71).

11 Anhang

Dokument 1: Checkliste

Checkliste für den Moderator

1. Vorbereitung:
    - ☐ Check-Liste und Skript des Moderators bereitstellen
    - ☐ Bewertungsbogen für den Proband bereitstellen
    - ☐ Anwendung starten bzw. zurücksetzen
    - ☐ Testdaten generieren (Koordinaten und Namen)
    - ☐ Stoppuhr zurücksetzten
    - ☐ Muffin bereitstellen
2. Einführung:
    - ☐ Willkommengruß und iLEXX vorstellen
    - ☐ Erläuterung der Testziele und des Think-Aloud-Tests
    - ☐ Einführung in die iLEXX-GUI
    - ☐ Testablauf Erkläutern
3. Durchführung der Testaufgaben :
    - ☐ Erste Aufgabe erledigen ggf. Notizen schreiben
    - ☐ Zweite Aufgabe erledigen ggf. Notizen schreiben
    - ☐ Dritte Aufgabe erledigen ggf. Notizen schreiben
    - ☐ Vierte Aufgabe erledigen ggf. Notizen schreiben
    - ☐ Fünfte Aufgabe erledigen ggf. Notizen schreiben
    - ☐ Sechste Aufgabe erledigen ggf. Notizen schreiben
4. Feedback
    - ☐ Fragebogen vom Proband ausfüllen lassen
5. Abschließen

Notizen:

Dokument 2: Moderator-Skript

Willkommen

Danke, dass Sie sich entschieden haben mir zu helfen und bei diesem Usability-Test teilzunehmen. iLEXX ist ein maritimes Überwachungssystem der Bundeswehr. Dieses System wird über mehrere Jahre weiterentwickelt. Nachdem die iLEXX-GUI jetzt um eine Suchfunktion erweitert wurde, möchte ich mit diesem Test ein paar Usability-Fragen beantworten.

Bei der iLEXX-GUI handelt es sich um mehrere Ansichten (auch **Views**) zur Darstellung detaillierter Lageinformationen. Die **Tactical Display Area (TDA)** nimmt den größten Teil der Benutzeroberfläche ein. Darin sind Daten über Ziele dargestellt. Beispiele für diese Ziele können Schiffe oder Taucher sein. Die aufgezeichneten Positions- und Bewegungsdaten dieser Ziele werden als Tracks genannt.

Ein **Track** verfügt über eine Vielzahl von Eigenschaften. Diese beinhalten statische und dynamische Informationen. Die folgenden Track-Eigenschaften sind für den Test relevant:

• *Name bzw. Alias Name:* Der Name wird automatisch durch das iLEXX-System vergeben. Der Alias Name hingegen stellt ein Pseudonym dar.

• *Coordinates* Gibt die aktuellen Koordinaten des Tracks an. Die Koordinaten liegen dabei in Längen und Breitengraden vor und werden in periodischen Zeitabständen dynamisch aktualisiert.

• *Threat Level:* kategorisiert das Bedrohungspotenzial eines Tracks. Mögliche Bedrohungsstufen sind: *LOW*, *NEUTRAL*, *ELEVATED*, *HIGH*.

• *Affiliation* Bei der Affiliation handelt es sich um eine Kategorisierung von Zielen. Der Track-Affiliation können verschiedene Zustände zugeordnet werden. Für diesen Test ist nur der Zustand „Suspect“ relevant.

Safezone: in der Regel sind rot-markierte Gebiete auf der TDA Sperrzonen (Safezones). Es sind bspw. militärische Sicherheitszonen, in denen Übungen oder Einsätze durchgeführt werden können. Es ist normalerweise anderen Schiffen nicht gestattet dieses Gebiet zu kreuzen oder sich darin aufzuhalten. Daher erhalten Sie eine Alarmierung, wenn sich ein Track innerhalb des definierten Gebiets einer Safezone befindet.

Die Track-Tabelle bezeichnen wir die Tabelle die rechts oben angegliedert ist. Sie ist eine Auflistung aller Tracks, die im System vorhanden sind. Sie finden alle Eigenschaften die, ich Ihnen genannt habe, in den verschiedenen Spalten. Durch das Anklicken des Spaltentitels wird die Tabelle nach der ausgewählten Spalte entsprechend sortiert.

PolarPanel: eine weitere Darstellung aller Tracks bietet der Polarpanel. Dabei ist die Threat-Level-Eigenschaft der Fokus dieser Ansicht.

Die Operator-Log-View wird dazu genutzt, um Auffälligkeiten festzuhalten und an anderen Operatoren zu verschicken. Um auf einen Track oder eine Sperrzone zu verweisen fügen Sie (#T_<TrackId>) bzw. (#S_<SafeZone Id>) in Ihrer Nachricht ein.

Kontextmenu eines Tracks
Durch einen Rechtsklick auf der TDA auf das entsprechende Objekt, wird mit dem Befehl *write Log* automatisch die Operator-Log-View geöffnet und auf das ausgewählte Objekt verwiesen. Das Kontextmenu eines Tracks ist aus jedem Bereich, die, Informationen zu Tracks bereitstellt, aktivierbar.

Folgende Optionen im Kontextmenu sind für diesen Test relevant:

• *Set Affiliation:* die Benutzer haben die Möglichkeit die Trackaffiliation zu setzen (siehe Eigenschaften eines Tracks).
• *Set Threat Level:* es besteht die Möglichkeit den Trackthreatlevel zu setzen (siehe Eigenschaften eines Tracks).

Der Test dauert maximal 30min und läuft folgendermaßen: Ich weise Ihnen mehrere Aufgabe zu. Sie bearbeiten die Aufgabe während ich Notizen mache. Lassen Sie sich nicht dadurch ablenken. Es handelt sich bei diesem Test **nur um die Untersuchung der Suchfunktion** nicht jedoch um Ihrer Kenntnisse.

Dokument 3: Hauptaufgabe

Ihre Hauptaufgabe: Sie überwachen mit dieser Anwendung die nordöstliche Küste Deutschlands. Sie müssen **Tracks** mit verdächtigem Verhalten kennzeichnen und **Ereignisse** melden.

Ein **Track** ist verdächtig wenn er den Threat-Level „High“ besitzt. In diesem Fall müssen Sie diesem Track die Affiliation „Suspect“ zuordnen.

Wenn ein Track in eine Sperrzone einfährt, spricht man von einem Ereignis. Das System blendet einen entsprechenden Warnhinweis ein. Sie müssen nun auf den roten Kreis klicken um festzustellen welcher Track das ist. Melden Sie dieses Ereignis im Operator-Log-View in dem Sie den folgenden Text eintippen: „#T_<TrackId> fuhr in eine Sperrzone ein.“

Während Sie die Küste überwachen werde ich Ihnen weitere Teilaufgaben zuweisen. Ihre Überwachungsaufgabe soll weiterhin Vorrang haben.

Sie können mit der Überwachung beginnen!

Tabelle 9: Fragebogen

Abschließender Fragebogen für den Proband					
	1	**2**	**3**	**4**	**5**
Wie oft haben Sie schätzungsweise den Sperrzonenalarm verpasst?	Oft	2	3	4	Nie
Wie oft konnten Sie schätzungsweise den bedrohlichen Track nicht melden?	Oft	2	3	4	Nie
Wie bewerten Sie das Suchfeld (Position, Größe, Label, Icon, etc.)	unangemessen	2	3	4	angemessen
Wie bewerten Sie die Darstellung der Suchergebnisse? (Maskieren und Selektieren)	unangemessen	2	3	4	Angemessen
Wie gut konnte das Suchfeld seine Funktion beschreiben?	unklar	2	3	4	Selbstbeschreibend
Wie würden Sie die Steuerbarkeit bewerten? (Sucheintrag eingeben, ändern oder löschen)	schlecht	2	3	4	gut
Inwiefern entspricht die Suchfunktion Ihrer Erwartungshaltung?	enttäuschend	2	3	4	Erwartungskonform
Wie hat sich die Suchunktion bei Tippfehlern verhalten? (Fehlertoleranz)	intolerant	2	3	4	tolerant
Wie schätzen Sie die Antwortzeit der Suchfunktion?	langsam	2	3	4	schnell
Wie schätzen Sie den Lernaufwand zur Bedienung der Suchfunktion? (Lernförderlichkeit)	schwer	2	3	4	leicht
Haben Sie Verbesserungsvorschläge oder Feedback?					

```java
    /**
     * for future use: create date query
     */
    public static Query createDateQuery(final Calendar requestedDate, final String typ, final String con) {

        Calendar min = null;
        Calendar max = null;
        switch (con) {
        case "GreaterThan":
            min = requestedDate;
            max = maxdate;
            break;
        case "LessThan":
            min = mindate;
            max = requestedDate;
            break;
        default:
            min = requestedDate;
            max = requestedDate;

        }

        final Query monthquery = NumericRangeQuery.newIntRange("month",
                min.get(Calendar.MONTH), max.get(Calendar.MONTH), false, false);
        final Query dayquery = NumericRangeQuery.newIntRange("day",
                min.get(Calendar.DAY_OF_MONTH), max.get(Calendar.DAY_OF_MONTH), false, false);
        final Query hourquery = NumericRangeQuery.newIntRange("hour",
                min.get(Calendar.HOUR_OF_DAY), max.get(Calendar.HOUR_OF_DAY), false, false);
        final Query minutequery = NumericRangeQuery.newIntRange("minute",
                min.get(Calendar.MINUTE), max.get(Calendar.MINUTE), false, false);
        final Query secondquery = NumericRangeQuery.newIntRange("second",
                min.get(Calendar.SECOND), max.get(Calendar.SECOND), false, false);

        final Query yearconditionquery = NumericRangeQuery.newIntRange("year",
                requestedDate.get(Calendar.YEAR), requestedDate.get(Calendar.YEAR), true, true);
        final Query monthconditionquery = NumericRangeQuery.newIntRange("month",
                requestedDate.get(Calendar.MONTH), requestedDate.get(Calendar.MONTH), true, true);
        final Query dayconditionquery = NumericRangeQuery.newIntRange("day",
                requestedDate.get(Calendar.DAY_OF_MONTH), requestedDate.get(Calendar.DAY_OF_MONTH), true, true);
        final Query hourconditionquery = NumericRangeQuery.newIntRange("hour",
                requestedDate.get(Calendar.HOUR_OF_DAY), requestedDate.get(Calendar.HOUR_OF_DAY), true, true);
        final Query minuteconditionquery = NumericRangeQuery.newIntRange("minute",
                requestedDate.get(Calendar.MINUTE), requestedDate.get(Calendar.MINUTE), true, true);

        // query using year comparing
        final Query yearquery = NumericRangeQuery.newIntRange("year", min.get(Calendar.YEAR),
                max.get(Calendar.YEAR), false, false);

        // query using month comparing. only on document with value of the field
        // "year" equals the year of requestedDate
        final Builder bqSameYear = new BooleanQuery.Builder();
        bqSameYear.add(yearconditionquery, Occur.MUST);
        bqSameYear.add(monthquery, Occur.MUST);
        final Query querySameYear = bqSameYear.build();

        // query on documents that have the same date as the requestedDate but
        // starts to differ with(as of) the day
        final Builder bqSameYearMonth = new BooleanQuery.Builder();
        bqSameYearMonth.add(yearconditionquery, Occur.MUST);
        bqSameYearMonth.add(monthconditionquery, Occur.MUST);
        bqSameYearMonth.add(dayquery, Occur.MUST);
        final Query querySameYearMonth = bqSameYearMonth.build();

        // query on documents that have the same date as the requestedDate but
        // starts to differ with(as of) the hour
        final Builder bqSameYearMonthDay = new BooleanQuery.Builder();
        bqSameYearMonthDay.add(yearconditionquery, Occur.MUST);
        bqSameYearMonthDay.add(monthconditionquery, Occur.MUST);
        bqSameYearMonthDay.add(dayconditionquery, Occur.MUST);
        bqSameYearMonthDay.add(hourquery, Occur.MUST);
        final Query querySameYearMonthDay = bqSameYearMonthDay.build();

        // query on documents that have the same date as the requestedDate but
        // starts to differ with(as of) the minute
        final Builder bqSameYearMonthDayHour = new BooleanQuery.Builder();
        bqSameYearMonthDayHour.add(yearconditionquery, Occur.MUST);
        bqSameYearMonthDayHour.add(monthconditionquery, Occur.MUST);
        bqSameYearMonthDayHour.add(dayconditionquery, Occur.MUST);
        bqSameYearMonthDayHour.add(hourconditionquery, Occur.MUST);
        bqSameYearMonthDayHour.add(minutequery, Occur.MUST);
        final Query querySameYearMonthDayHour = bqSameYearMonthDayHour.build();

        // query on documents that have the same date as the requestedDate but
        // starts to differ with(as of) the second
        final Builder bqSameYearMonthDayHourMinute = new BooleanQuery.Builder();
        bqSameYearMonthDayHourMinute.add(yearconditionquery, Occur.MUST);
        bqSameYearMonthDayHourMinute.add(monthconditionquery, Occur.MUST);
        bqSameYearMonthDayHourMinute.add(dayconditionquery, Occur.MUST);
        bqSameYearMonthDayHourMinute.add(hourconditionquery, Occur.MUST);
        bqSameYearMonthDayHourMinute.add(minuteconditionquery, Occur.MUST);
        bqSameYearMonthDayHourMinute.add(secondquery, Occur.MUST);
        final Query querySameYearMonthDayHourMinute = bqSameYearMonthDayHourMinute.build();

        // combining queries using OR condition.
        // hit document must satisfies at least one query
        final Builder bqDateQuery = new BooleanQuery.Builder();
        bqDateQuery.add(yearquery, Occur.SHOULD);
        bqDateQuery.add(querySameYear, Occur.SHOULD);
        bqDateQuery.add(querySameYearMonth, Occur.SHOULD);
        bqDateQuery.add(querySameYearMonthDay, Occur.SHOULD);
        bqDateQuery.add(querySameYearMonthDayHour, Occur.SHOULD);
        bqDateQuery.add(querySameYearMonthDayHourMinute, Occur.SHOULD);

        return bqDateQuery.build();
    }
```

Listing 21: Die Erstellung eines Abfrageobjekts zur Durchsuchung von Datierungen

12 Literatur

Apache Lucene TM (Hrsg.):	„Apache Lucene Core. “. https://lucene.apache.org/core/ Stand: 15.11.2016.
Barnum, C.;	„Usability Testing Essentials.“ 1.Aufl.,Elsevier Inc., Burlington 2011.
Baumgartner, M.; Klonk, M; Pichler, H; Seidl, R; Tanczos, S;	„Agile Testing.“ 1.Aufl.,Carl Hanser Verlag, München 2013.
Bernsen, N.; Dybkjær, L;	„Multimodal Usability. “ 1.Aufl.,Springer Verlag, London 2009.
Bucsics, T; Baumgartner, M; Seidl, R; Gwihs, S;	„Basiswissen Testautomatisierung : Konzepte, Methoden und Techniken. “ 1.Aufl., dpunkt Verlag, Heidelberg 2015.
Cockburn, A;	„USE CASES: effektiv erstellen. “ 1.Aufl.,mitp Verlag, Heidelberg 2003.
Cooper, A.; Reimann, R;	„About face 2.0 : the essentials of interaction design.“ 2.Aufl., Indianapolis Ind., Wiley 2003.
Fischer, P.; Hofer, P;	„Lexikon der Informatik. “ 15.Aufl.,Springer Verlag, Berlin 2011.
Fisher-Stabel, P.; Gollmer, K.;	„Informatik für Ingenieure : fit für das Internet der Dinge.“ UVK Verlagsgesellschaft mbH, Konstanz 2016.
Gospodnetic, O.; Hatcher, E.; McCandless, M.;	„Lucene In Action. “ 2. Aufl., Manning Publications Co., Stamford 2010.
Grigorik, I;	„High Performance Browser Networking. “ 1. Aufl., O'Reilly, United States of America 2013.
Hardt, M.; Theis, F.;	„Suchmaschinen entwickeln mit Apache Lucene.“ 1. Aufl., Software & Support Verlag, 2004.
Hruschka, P.;	„Business Analysis und Requirements Engineering. “ 1.Aufl.,Carl Hanser Verlag, München 2014.
ISO 9241-110	„Ergonomie der Mensch-System-Interaktion. Teil 110: Grundsätze der Dialoggestaltung“. Beuth Verlag, Berlin 2008.

Kahlbrandt, B.;	„Software Engineering: Objektorientierte Software-Entwicklung mit der Unified Modeling Language. “ 2. Aufl., Springer Verlag, Berlin 2013.
Kanbach, A;	„SIP - die Technik : Grundlagen und Realisierung der Internet-Technik. “ 1. Aufl., Vieweg Verlag, Wiesbaden 2005.
Müller, H; Krummenacher, J; Schubert, T;	„Aufmerksamkeit und Handlungssteuerung: Grundlagen für die Anwendung. “ 1. Aufl., Springer Verlag, Berlin 2015.
Nielson, J;	„Usability engineering“ 1. Aufl., Morgan Kaufmann Verlag, United States of America 1993.
Teufel, M.; Helming, J;	„Eclipse 4: Rich Clients mit dem Eclipse SDK 4.2“ 1.Aufl.,entwickler.press Verlag, Frankfurt 2012.
Vogel, L;	„Eclipse Rich Client Platform.“ 3.Aufl, Vogella Verlag, o.O. 2015.
Wolski, W;	„PONS Großwörterbuch Deutsch als Fremdsprache.“ 1.Aufl.,PONS GmbH, Stuttgart 2015.
Wütherich, G; Hartmann, N; Kolb, B; Lübken, M;	„Die OSGi Service Platform.“ 1.Aufl.,dpunkt Verlag, Heidelberg 2008.