

# Verteiltes Genom Browsing

## Projektspezifikation

23. Dezember 2015

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Aufgabenstellung und Anforderungen . . . . .	3
1.1.1	Aufgabenstellung . . . . .	3
1.1.2	Anforderungen . . . . .	3
1.2	Beispiele zur Nutzung: Use Cases . . . . .	5
1.2.1	Use Case 1 . . . . .	5
1.2.2	Use Case 2 . . . . .	5
1.2.3	Use Case 3 . . . . .	5
1.3	Umsetzung . . . . .	6
1.3.1	Vorgehensweise des Systems . . . . .	6
1.3.2	Arbeitsbereiche . . . . .	7
<b>2</b>	<b>Integration</b>	<b>8</b>
2.1	Integrationsprozess . . . . .	8
2.1.1	Ablauf der Integration . . . . .	8
2.1.2	Quellenauswahl . . . . .	8
2.1.3	Attributauswahl und -mapping . . . . .	8
2.1.4	Mengengerüst . . . . .	9
2.1.5	Inputfile-Format vom lokalen Parser erzeugt . . . . .	9
2.1.6	Inputfile vom globalen Parser erzeugt . . . . .	10
2.1.7	Sequenzdiagramm . . . . .	11
2.2	Datenbankentwurf . . . . .	12
2.3	Entwurf des Parsers . . . . .	13
2.4	Klassendiagramm . . . . .	14
2.5	Schnittstellenspezifikation . . . . .	14
2.5.1	Schnittstelle: Integration - Middleware . . . . .	14
2.5.2	Schnittstelle: Integration - Benutzer . . . . .	14
2.6	Tests . . . . .	14
2.6.1	Unit-Tests . . . . .	14
<b>3</b>	<b>Middleware</b>	<b>16</b>
3.1	Indexstruktur . . . . .	16
3.1.1	Anforderungen . . . . .	16
3.1.2	Funktionen und Datenstrukturen . . . . .	16
3.2	IndexController . . . . .	20
3.2.1	Anforderungen . . . . .	20
3.2.2	Funktionen . . . . .	20
3.3	Klassen-Diagramm . . . . .	21
3.4	Sequenzdiagramme . . . . .	22
3.4.1	Intervall-Suche . . . . .	22
3.4.2	Names-Suche . . . . .	23
3.4.3	Suche nach Gennamen . . . . .	24

3.5	Unit-Tests . . . . .	25
3.5.1	QueryReceiver . . . . .	25
3.5.2	GeneTranslator . . . . .	28
3.5.3	IndexController . . . . .	29
3.5.4	Intervallbaum . . . . .	30
3.6	Stresstests . . . . .	32
<b>4</b>	<b>Schnittstellenspezifikation: Middleware - Frontend</b>	<b>33</b>
4.1	Mögliche Anfragen . . . . .	33
4.1.1	Suchanfrage JSON . . . . .	33
4.1.2	Intervallanfrage JSON . . . . .	34
4.2	Beispiel der Kommunikation mit JSON . . . . .	35
<b>5</b>	<b>Frontend</b>	<b>38</b>
5.1	Use Cases . . . . .	38
5.2	Mock-Ups der Benutzerschnittstelle . . . . .	39
5.3	Technologie und Klassenstruktur . . . . .	41
5.4	Sequenzdiagramme . . . . .	43
5.5	Unit-Tests . . . . .	44
5.6	Integration-Tests . . . . .	45
5.6.1	Suchfunktion . . . . .	45
5.6.2	Quellen-Button . . . . .	45
5.6.3	Quellen-Scroller . . . . .	45
5.6.4	Zoom-slider Szenario . . . . .	45
5.6.5	Chromosom-Auswahl . . . . .	45
5.6.6	Allgemein . . . . .	46

# 1 Einleitung

Dieses Dokument ist die Projektspezifikation für das Semesterprojekt zur verteilten Echtzeitrecherche in Genomdaten. Hierbei umfasst der Begriff "Genomdaten":

**Chromosomen** enthalten das Erbgut eines Menschen. Sie enthalten somit die DNA, auf welcher die einzelnen Gene abgebildet sind. Verschiedene Spezies haben unterschiedlich viele Chromosomen und Gene.

Für dieses Projekt beziehen wir uns ausschließlich auf die 24 Chromosomen des Menschen. Hier sind die zwei Geschlechtschromosomen X und Y bereits enthalten.

**Gene** bestehen aus einer Sequenz von Basen, welche ein zu synthetisierendes Protein kodieren. Jedes Gen kodiert ein oder mehrere Proteine. Jedes Gen kann auf Grund seiner Genkoordinaten entweder identifiziert, oder seine Sequenz anhand der Koordinaten gefunden werden.

**Genkoordinaten** geben an, auf welchem Abschnitt von welchem Chromosom ein Gen liegt. Chromosom und Gen bilden somit die Koordinate. Hierbei werden die untere und die obere Grenze angegeben.

**Basenpaare** ergeben sich durch die Doppelhelix-Struktur der DNA.

Es gibt vier Basen:

1. Adenin (A)
2. Cytosin (C)
3. Guanin (G)
4. Thymin (T)

von denen jeweils zwei Gegenstücke zueinander sind. Diese Paare sind A - T und C - G. Es folgt also, dass nur diese Basenpaare in der Helix vorhanden sind.

Spricht man von Mbp, so sind dies Megabasenpaare, hierbei entspricht

1Mbp = 1000000 Bp (Basenpaare).

**Mutationen** sind Abweichungen in der Basensequenz eines Gens zum Referenzgenom.

Das heißt:

Unterscheidet sich an einer spezifischen Stelle des Chromosoms eines Probanden eine Base zu der im Referenzgenom verzeichneten, so kann dies eine Punktmutation sein. Wirkt sich die Veränderung jedoch nicht weiter aus, so handelt es sich nur um eine Variante des Basenpaares. Dies wird auch als Einzelnukleotid-Polymorphismus (SNP) bezeichnet.

Weiterhin existieren bestimmte Basensequenzen innerhalb der Gen, welche mehrfach hintereinander wiederholt werden. Die Anzahl der Wiederholungen kann von Mensch zu Mensch unterschiedlich sein. Diesen Sachverhalt bezeichnet man als Kopienzahlvariation (CNV).

Neben CNVs können auch fehlende oder zusätzliche Basenpaare in bestimmten Basensequenzen auftreten. Diese fehlenden bzw. zusätzlichen Basenpaare bezeichnet man als Indel.

**Das Referenzgenom** wird in unregelmäßigen Abständen neu ermittelt. Anhand dieses Genoms werden alle weiteren Sequenzierungen von Basenpaaren überprüft und Mutationen ermittelt. Das aktuelle Referenzgenom GRCh38.p5<sup>1</sup> ist am 25. September 2015 herausgegeben worden.

Im folgenden Dokument wird die Realisierung des Softwareprojektes erläutert.

---

<sup>1</sup><http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/>

## 1.1 Aufgabenstellung und Anforderungen

### 1.1.1 Aufgabenstellung

Es soll ein System entwickelt werden, welches in Echtzeit Genomdaten durchsucht und die grafische Auswertung übernimmt.

### 1.1.2 Anforderungen

- Es werden drei Datenbanken fest integriert:
  - dbSNP<sup>2</sup>
  - 1000 Genomes Project<sup>3</sup>
  - The Cancer Genome Atlas: Lung and Colorectal Cancer<sup>4</sup>

Weiterhin wird das System für weitere (eigene) Datenquellen erweiterbar sein.

- Folgende Informationen werden bereitgestellt:
  - Metadaten: Datenquelle, Zeitpunkt des Downloads
  - Genkoordinate der Mutation (ggf. Abschnitt)
  - Mutierte Basensequenz (auch Referenzgenom muss bekannt sein)
  - Relative Häufigkeit der Mutationen
  - Quellspezifische Sample-Attribute (Krankheit, Geschlecht, etc.)
- Eine Anfrage wird in <1 Sekunde durchlaufen.  
Zulässige Anfragen sind:
  - Intervallanfragen:
    - \* Anfrage mit: Chromosom, linke und rechte Grenze (Genkoordinaten)
    - \* Systemkorrektur: Intervall >10Mbp führt dazu, dass ein Fehler geworfen wird
    - \* Ergebnis: Mutationen in dem Abschnitt
  - Genanfragen:
    - \* Anfrage mit: Name eines humanen Gens
    - \* Systemkorrektur: Namensvorschläge, wenn Genname nicht vollständig bekannt
    - \* Ergebnis: Vom System bestimmter Genabschnitt, sowie die darauf befindlichen Mutationen

---

<sup>2</sup><http://www.ncbi.nlm.nih.gov/SNP/>

<sup>3</sup><http://www.1000genomes.org/>

<sup>4</sup><http://cancergenome.nih.gov/>

Folgende Filter werden zur Suchoptimierung angeboten:

- Einschränkung auf Quelle
- Einschränkung auf relative Häufigkeit
- Gleichheit quellspezifischer Attribute

- Die Darstellung erfolgt:

- je Datenquelle
- mit einem max. 10Mbp großen Ausschnitt des Chromosoms
- mit Referenzgenom zum Vergleich bei kleinen Abschnitten
- mit Anzeige der Basenpaare und Markierung von Mutationen in kleinen Abschnitten ( $\leq 200\text{Bp}$ )
- mit Andeutung der Verteilung der Häufigkeit von Mutationen in großen Abschnitten ( $> 200\text{Bp}$ )

Zudem bietet das Interface interaktive Möglichkeiten:

- Zoomfunktion mit fünf Stufen, um den dargestellten Bereich zu verkleinern oder zu vergrößern
  - Scrollen, um sich horizontal auf dem Chromosom fortzubewegen
  - Konfigurationsmöglichkeiten zur Darstellung und anderer Systembereiche
- Das System wird unter Mozilla Firefox laufen und auf bestehenden Webservern leicht einzurichten sein

## 1.2 Beispiele zur Nutzung: Use Cases

### 1.2.1 Use Case 1

Ein Benutzer möchte herausfinden, welche Mutationen im Bereich von 144MB – 154MB auf dem Chromosom 7 auftreten können. Hierfür wählt er die 1000 Genomes Projekt-Datenbank aus.

Das ganze Chromosom wird in einem eigenen Anzeigebereich (Lane) im Vergleich zum Referenzgenom angezeigt. Er erkennt durch eine Markierung, dass im Bereich von 150MB-152MB gehäuft Mutationen auftreten können. Da er diesen Bereich nicht detailliert einsehen kann, zoomt der Benutzer herein. Die Basenpaare in der 1000 Genomes-Lane sind immer detaillierter zu erkennen, bis er genau einsehen kann, welche Mutationen auftreten können. Bei Erreichen der maximale Zoomstufe, sind alle Basenpaare des Chromosoms 7 und des Referenzgenoms, das nun eingeblendet wird, zu erkennen und er kann die auftretenden Mutationen betrachten.

### 1.2.2 Use Case 2

Ein Benutzer möchte herausfinden, welche Mutationen bei einer bestimmten relativen Häufigkeit im Bereich eines Gens auftreten. Da er sich sehr für den colorectalen Bereich interessiert, wählt er The Cancer Genome Atlas (TCGA) aus. Da der Benutzer nicht den genauen Namen des Gens kennt und sich bei der Eingabe irrt, bekommt er keine Basenpaare angezeigt, sondern Vorschläge, welches Gen er gesucht haben könnte. Auf Grund der Vorschläge erinnert sich der Benutzer und wählt das korrekte Gen.

Es erscheint eine Lane, welche die Daten für den Bereich darstellt und der Benutzer erkennt die Bereiche, in welchen Mutationen auftreten. Zusätzlich werden ihm die relativen Häufigkeiten der Mutationen angezeigt.

### 1.2.3 Use Case 3

Ein Benutzer möchte sich darüber informieren, mit welcher Häufigkeit in einem bestimmten Bereich eines bestimmten Gens Mutationen auftreten können. Hierbei wählt er die HGMD mit einem Bereich von 140MB – 155MB. Da der Bereich jedoch zu groß ist, wird ihm nur der Bereich von 140MB-150MB dargestellt.

Er erkennt in diesem Bereich, dass sehr wenig Mutationen auftreten. Ihn interessiert jedoch ebenfalls, ob bei Krebspatienten höhere Mutationsraten existieren. Dazu führt er die gleiche Suche noch einmal aus, wählt jedoch zusätzlich TCGA für Lungenkrebs und TCGA für Colorectalkrebs.

Als Ergebnis werden ihm die drei Lanes untereinander angezeigt und er kann die Häufigkeit von Mutationen in den Bereichen vergleichen.



## 1.3 Umsetzung

### 1.3.1 Vorgehensweise des Systems

Damit dem System Daten zur Verfügung stehen, müssen die drei Hauptdatenbanken zuerst aus dem Internet heruntergeladen werden.

Nachdem der Download erfolgreich war, werden die Datenbanken in ein einheitliches Format überführt. Ist die Vereinheitlichung vollzogen, können nun die humanen Datensätze herausgefiltert und in die eigens entworfene Datenbank (DWH) eingespeichert werden. Sind alle Daten verarbeitet und somit das DWH vollständig, stehen die Daten bei Programmstart zum Aufbau des verteilten Index zur Verfügung.

Wurde der verteilte Index fertig gebaut, kann der Benutzer nun Anfragen an das System absenden. Die Anfragen werden von der Benutzeroberfläche (GUI) an die Middleware geschickt. Hier werden die Anfragen validiert und gegebenenfalls korrigiert. Sind die Anfragen korrekt, werden diese über den verteilten Index ausgeführt und die Ergebnisse an das GUI zurückgegeben. Hat das GUI die Ergebnisdaten empfangen, werden diese grafisch aufbereitet.

Abbildung 1 verdeutlicht diesen Ablauf:

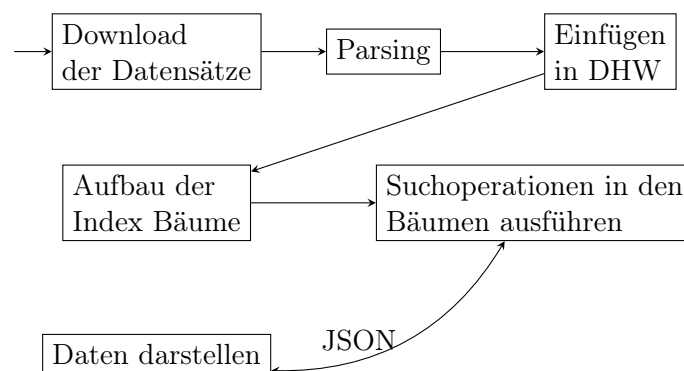


Abbildung 1: Kommunikationsdiagramm

### 1.3.2 Arbeitsbereiche

Zur Umsetzung des Systems müssen, wie aus 1.3 hervorgeht, drei Hauptbereiche abgedeckt werden. Diese sind:

**Integration:** Zuständigkeitsbereiche der Integration sind die Aufbereitung der Datenquellen und die Bereitstellung der aufbereiteten Datensätze in einer eigens entworfenen Datenbank.

**Middleware:** Die Middleware ist für die Kommunikation zwischen Datenbank und Frontend zuständig. Um die vom Frontend an die Middleware gesendeten Anfragen effizient bearbeiten zu können, befasst sich die Middleware mit dem Entwurf eines verteilten Index. Dieser ermöglicht die zeiteffiziente Suche in den Daten der Datenbank und somit eine zeiteffiziente Rückgabe der Ergebnisse an das GUI.

**Frontend:** Das Frontend ist für die grafische Aufbereitung der Ergebnisdaten, sowie für das GUI zur Bedienung des Systems zuständig.

Im folgenden werden die Teilbereiche, die Realisierung, sowie die Verwendung findenden Technologien erläutert.

## 2 Integration

### 2.1 Integrationsprozess

#### 2.1.1 Ablauf der Integration

Die Software wird standardmäßig zwei Quellen integriert haben: die dbSNP und das 1000GenomeProject. Diese werden, über ein Skript gesteuert, heruntergeladen. Wenn dieser Part abgeschlossen ist, wird ein lokaler Parser (später mehr zu den Parsern und ihrer Arbeitsweise) alle relevanten Daten, d.h. nur menschliche Daten, und aus diesen die, für das Projekt, relevanten Daten, aus den heruntergeladenen Dateien extrahieren und sie in einem, für den globalen Parser akzeptablen Format bereitstellen. Dies ist wichtig, da die Quellen nicht nur menschliche Daten bereitstellen, und auch viele weitere Daten vorhanden sind, die für das Projekt von keiner Relevanz sind. Der globale Parser wird diese Dateien dann in Datensätze staffeln und diese dann in das Data Ware House einfügen um sie dann so der Middleware für die Abfragen bereitzustellen.

Der lokale Parser wird benötigt, da es kein allgemeingültiges Standardformat gibt, in dem die Dateien abgespeichert werden. Die Formate sind auch keine, von dem DWH anerkannten, Dateiformate, die integriert werden können. Hinzu kommt, dass Sehr viele weitere Daten, die nicht notwendig sind, in den Dateien vorhanden sind und somit nur die benötigten Daten ausgelesen werden müssen.

Wenn der User weitere Quellen integrieren möchte, benötigt er zwei Dinge dafür: Ein Downloadskript, welches von der gewünschten Quelle die Dateien herunterlädt, und ein lokaler Parser, der die heruntergeladenen Dateien danach für den globalen Parser bereitstellt.

#### 2.1.2 Quellenauswahl

Die Quellen, die von uns integriert sein werden, werden die dbSNP und das 1000GenomeProject sein, da diese alle benötigten Daten frei zugänglich bereitstellen, ohne jegliche Gegenleistung zu verlangen. 1000GenomeProject stellt ausschließlich menschliche Daten mit Metadaten und Referenzgenom zur Verfügung, dbSNP stellt nur Mutationsdaten und das Referenzgenom bereit, ohne Metadaten, jedoch auch tierische Daten, wobei für uns nur die menschlichen von Relevanz sind. HGMD hingegen können wir nicht integrieren, da für die Benutzung ihrer Daten eine Lizenz erworben werden muss, die den finanziellen Rahmen des gesamten Projektes sprengt. TCGA bezieht sich auf ein weiteres Referenzgenom, weshalb wir die Integration dieser Quelle auslassen werden. Wir werden uns vorerst auf die frei zugänglichen Quellen konzentrieren, um eine Basis an Daten bereitstellen zu können. Weitere Quellen werden im Nachhinein integrierbar sein, weshalb die Option offen steht, TCGA im Nachhinein zu integrieren.

#### 2.1.3 Attributauswahl und -mapping

Beide Basis-Datenbanken werden von dem gleichen Anbieter gehostet, was beiden Quellen ein relativ uniformes Format gibt. So stellen beide Quellen die Daten in BAM-Dateien

und vcf-Dateien bereit, wobei wir uns auf die vcf-Dateien berufen, da diese die Daten beinhalten, die wichtig sind für die Abfragen, und auch gleichzeitig einfacher zu entschlüsseln sind, als die BAM-Dateien.

Wir entschieden uns, dass das Referenzgenom in einer Extradatei abgespeichert wird, um übermäßigen Traffic im DWH zu vermeiden. Des weiteren werden im DWH folgende Daten der Middleware bereitgestellt: Eine einzigartige ID für jeden Eintrag, die Mutationssequenz selber, die Koordinaten, wo im Referenzgenom die Mutation auftritt, der Name der Quelle, um eine schnelle Einordnung nach Quellen zu gewährleisten, der Name des Referenzgenoms, eine Angabe, in welchem Chromosom die Mutation auftritt, und einen Verweis auf den entsprechenden Metadatensatz der Mutation. Metadaten bedeuten in unserem Falle, zusätzliche Informationen zur Person, bei der diese Mutation gefunden wurde.

Der Metadatensatz besteht aus einer eindeutigen ID, der Quelle, woher die Daten stammen, dem Geschlecht des Testsubjekts, dem Herkunftsland und der Downloadzeit. Wir entschieden uns für diese Metadaten, da sie in dem 1000GenomeProject bereitgestellt wird. Die dbSNP stellt keine Metadaten bereit.

Im folgenden ein Ausschnitt aus einem Datensatz einer Mutation beider Quelldateien: *17 52 rs556541063 C A*. Danach kommt ein String mit Datenmüll - für das Projekt unrelevante Zusatzdaten - und dann hat die 1000GenomeProject-Quelle dahinter eine Liste mit Angaben, bei welchen Personen diese Mutation festgestellt wurde (somit also die Metadatenverweise). Der Ausschnitt hat folgende Werte: Das Chromosom, in dem die Mutation auftritt, die Position der Mutation, eine Quelldatenbankinterne ID für die Mutation, die wir jedoch nicht benötigen, referenzierte Stelle im Genom, und abschließend zu was sie mutierte. Im Falle einer Deletion wird dort nur ein eingetragenes sein, bei einer Insertion die entsprechend längere Basenfolge, eine Translokation zeigt die entsprechende mutierte Base an.

#### **2.1.4 Mengengerüst**

Eine Schätzung des Mengengerüsts ist für uns nicht möglich, da die Dateien der Quellen keine Schätzung zulassen. Die Dateien reichen in der Größe von einigen KB bis zu mehreren GB, die sehr stark anwachsen, wenn man sie entpackt (mindestens 15-fache Vergrößerung). In diesen Dateien kann man jedoch nicht approximieren, wie groß der Anteil an wichtigen Daten für das DWH ist, und wie viel Datenmüll enthalten ist, da das von Datei zu Datei variiert.

Der Speicherplatz, den die Daten im DWH einnehmen werden, wird sich nach dem Einfügen um das 1,5-fache erhöhen.

#### **2.1.5 Inputfile-Format vom lokalen Parser erzeugt**

Referenzgenomname: „Name des Referenzgenoms Bsp: GRCh38“  
Quelle: „hier die Quelle angeben“

```

$$
SampleID:,,SampleID aus der DB“
Genkoordinaten:,,Angabe der Koordinaten“
Mutationssequenz:,,Sequenz“
$$
SampleID:,,SampleID aus der DB“
Gender:,,m oder f“
Population:,,drei Buchstaben bsp: GBR“
EOF

```

Der lokale Parser wird für jeden Datensatz einen Eintrag in dieser Art in sein Inputfile schreiben, so dass er, nach erfolgreichem Parsen eine Textdatei mit entsprechend vielen Einträgen dieser Art hat. Eine EBNF liegt im Anhang bereit.

### 2.1.6 Inputfile vom globalen Parser erzeugt

Es werden zwei Textdateien erstellt werden, eine für die Metadaten-Datensätze und eine für die Mutations-Datensätze. Diese werden danach per COPY-Befehl vom DWH eingelesen. Für diesen Befehl benötigen beide Dateien einen bestimmten Aufbau: Jedes Attribut ist voneinander per Leerzeichen getrennt und jede Zeile beherbergt einen Datensatz.

### 2.1.7 Sequenzdiagramm

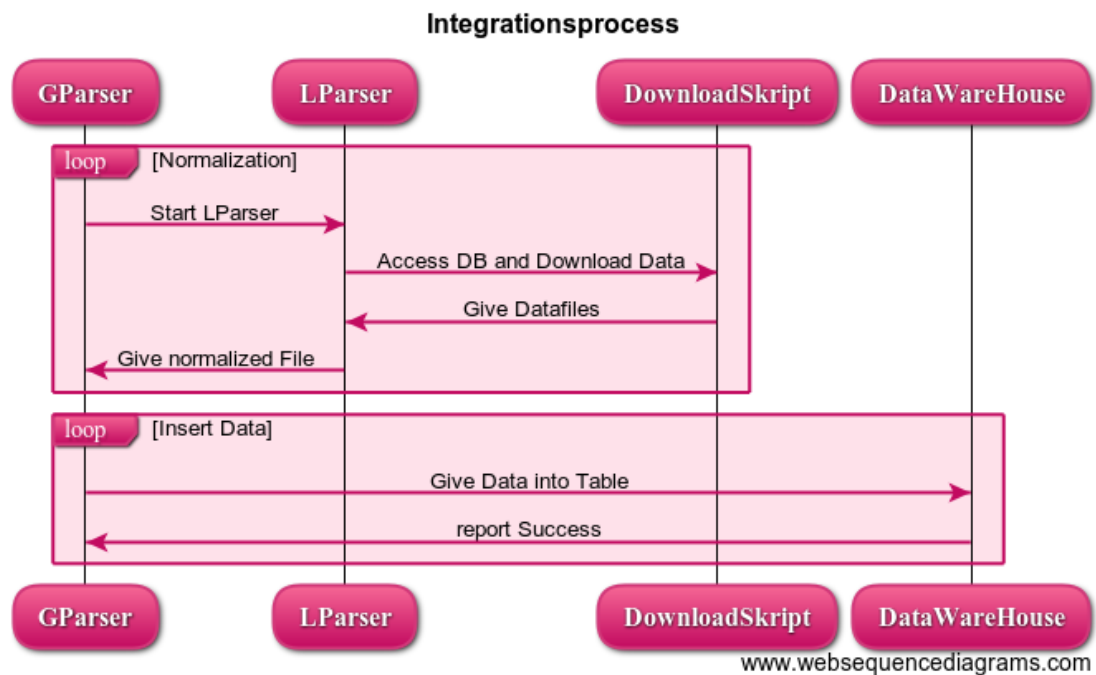


Abbildung 2: Sequenzdiagramm des Integrationsprozesses

Der Ablauf besteht aus zwei Schleifen: Zuerst wird der gesamte Prozess von hinten gestartet, um zu gewährleisten, dass vor Beginn des Integrationsprozesses alle Teilprozesse laufen um Probleme während des Prozesses zu vermeiden. Der globale Parser wird gestartet, startet von sich aus die  $n$  lokalen Parser, welche dann die Daten ihrer  $n$  Quellen herunterladen. Jede Quelle hat einen, eigens für sich geschriebenen, lokalen Parser. Nach dem erfolgreichen Herunterladen parsen die  $n$  lokalen Parser ihre Daten in jeweils ein Inputfile. Somit haben wir  $n$  Inputfiles. Wenn diese Schleife beendet ist, wird der globale Parser alle  $n$  Inputfiles einlesen und aus diesen  $n$  Dateien zwei Textdateien erstellen - eine Metadaten-Datei und eine Mutations-Datei. Diese zwei Dateien werden dann vom DWH per COPY-Befehl eingelesen, die Metadaten zuerst, um Probleme mit der Relation zu verhindern, danach die Mutationen.

## 2.2 Datenbankentwurf

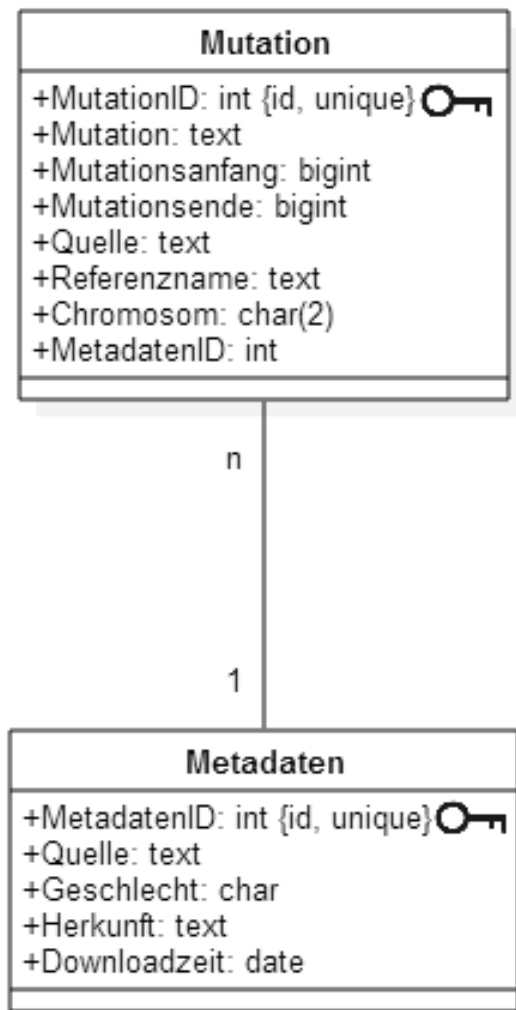


Abbildung 3: Entwurf der Datenbank

Die, bereits bei dem Attributmapping genannten, Attribute werden in zwei Tabellen gestaffelt, um eine effizientere und weniger fehleranfällige Struktur zu haben. Man kann sich möglicherweise sogar Einträge sparen, da es vorkommen kann, dass mehrere Mutationen von der gleichen Person stammen. Außerdem können die Anfragen der Middleware somit auch besser durchgeführt werden.

Erklärungen der Attribute der "MutationTabelle:

Name	Typ	Format	Wertebereich	Beispiel	Notwendig
MutationID	integer	i	0 - 2.147.483.647	1, 2, 3, ...	nein
Mutation	text	String	beliebig lang	'ATTCGATTAGCAGT'	nein
Mutationsanfang	bigint	i	0 - 9.223.372.036.854.775.807	60000	nein
Mutationsende	bigint	i	0 - 9.223.372.036.854.775.807	80000	nein
Quelle	text	String	beliebig lang	'dbSNP'	nein
Referenzname	text	String	beliebig lang	'GRCh38'	nein
Chromosom	char(2)	cc	X, Y oder 0 - 99	'X', 'Y', '64'	nicht notwendig
MetadatenID	int	i	0 - 2.147.483.647	1, 2, 3, ...	nein

Erklärungen der Attribute der "MetadatenTabelle":

Name	Typ	Format	Wertebereich	Beispiel
MetadatenID	int	i	0 - 2.147.483.647	1, 2, 3, ...
Quelle	text	String	beliebig lang	'dbSNP', '1000GenomeProject'
Geschlecht	char	c	m oder w	'm', 'w'
Herkunft	text	Länderkürzel	Länderkürzel (meist dreistellig)	'GBR'
Downloadzeit	date	dd/mm/yyyy	Bis zum Jahr 5874897 n.Chr.	'24/12/2015'

Wir entschieden uns für diesen Aufbau, da die Abfragen des Users sich in zwei Bereichen unterteilt: Die Mutation selber und die dazugehörigen Metadaten. Eine Separierung dieser beiden Teilgruppen gibt der Middleware mehr Möglichkeiten schneller und effizienter das gewünschte Ergebnis zu finden. Dabei muss aber beachtet werden, dass es noch eine Relation zwischen den beiden Tabellen geben muss, damit jede Mutation genau einen Metadaten-Datensatz zugewiesen bekommt. Dies war der Grund, weshalb wir uns für das relationale Datenbankschema entschieden. Dieses Datenbankschema gibt uns die Möglichkeit, eine Relation zwischen den Mutationen und den Metadaten zu erstellen.

Nach dieser Entscheidung benötigten wir ein Datenbankmanagementsystem, was den Ansprüchen des Projektes gerecht werden musste, das heißt, dass es effizient mit großen Datenmengen umgehen können muss. Unsere Entscheidung fiel auf PostgreSQL. Dieses DBMS kann problemlos viele Daten und vor allem große Datensätze speichern und verwalten, was ausschlaggebend für das gesamte Projekt ist.

## 2.3 Entwurf des Parsers

Jede Quelle wird einen eigenen Parser haben, der die nötigen Daten aus den heruntergeladenen Quelldateien herausliest und in ein entsprechendes Inputfile für den globalen Parser schreiben wird, den lokalen Parser. Es wird so viele lokale Parser geben, wie es Quellen gibt. Der lokale Parser wird aus jeder Datei, die ihm von seiner Quelle bereitgestellt wird, so viele Datensätze wie möglich parsen und in einem Inputfile für den globalen Parser abspeichern.

Der globale Parser wird danach sämtliche, von den lokalen Parsern erstellte, Inputfiles parsen und in zwei Textdateien abspeichern, eine mit den Mutations-Datensätzen, eine mit den Metadaten-Datensätzen. Das DWH wird diese zwei Dateien dann per COPY-Befehl einlesen.



## 2.4 Klassendiagramm

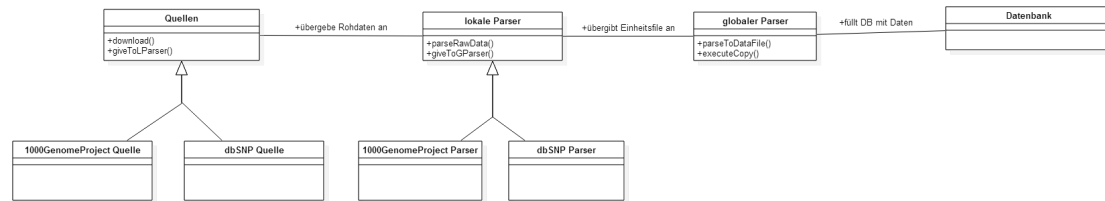


Abbildung 4: Klassendiagramm der Integration

Das Klassendiagramm verdeutlicht den linearen Ablauf des Integrationsprozesses. Zuerst werden die Quellen per Skript heruntergeladen und dann an die lokalen Parser übergeben. Diese parsen die Dateien dann in Inputfiles und übergeben diese an den globalen Parser übergeben. Dieser parset diese Dateien in Datenfiles, die danach mit dem COPY-Befehl des DWH in selbiges integriert werden.

## 2.5 Schnittstellenspezifikation

### 2.5.1 Schnittstelle: Integration - Middleware

Die Schnittstelle zwischen der Integration und der Middleware besteht darin, dass sich die Middleware per SQL Anfragen die, vom Nutzer gewünschten Daten, aus dem DWH lädt. Außerdem dient die Middleware als Bindeglied zwischen dem User und der Integration, da die Middleware die Einschränkungen der Anfrage des Users an die Middleware weitergibt und sich etwaige Auswahlmöglichkeiten an den User weitergibt.

### 2.5.2 Schnittstelle: Integration - Benutzer

Der User kann seine Anfragen nach speziellen Gesichtspunkten einschränken, welche er von der Integration gestellt kriegen muss. Diese Punkte wird die Middleware ihm übergeben.

Außerdem kann ein User weitere Quelldatenbanken integrieren, indem er ein Downloadskript und einen lokalen Parser in der Ordnerstruktur bereitstellt und sie in die Liste der Quellen einträgt.

## 2.6 Tests

### 2.6.1 Unit-Tests

Es gibt viele verschiedene Dinge, die getestet werden müssen. Bezüglich des lokalen Parsers, muss getestet werden, wie er mit einem fehlerhaften Input verfährt, und ob er korrekten Input auch auf korrekte Art und Weise parsed. Sollte er fehlerhaften Input

kriegen, soll er den Fehler in einer Extra-Fehlerdatei abspeichern.

Das gleiche muss auch für den globalen Parser getestet werden. Dieser soll sich wie der lokale Parser verhalten, sollte er einen fehlerhaften Input bekommen, muss er diesen Datensatz in einer separaten Fehlerdatei abspeichern, damit der Nutzer sich später diese fehlerhaften Datensätze selber überprüfen kann.

Nach diesem doppelten Fehlertest könnte angenommen werden, dass der Input für das DWH nun korrekt ist, sollte dies nicht der Fall sein, so besitzt das DBMS eine Funktion, die bei einem fehlerhaften Datensatz, das gesamte restliche Einlesen stoppt, jedoch alle bisher eingelesenen Datensätze gespeichert lässt.

Beide Parser werden auch dafür benutzt, um etwaige doppelte Einträge in den Inputfiles herauszufiltern, sodass keine in dem DWH abgespeichert werden.

Im Anhang finden sich einige Testfiles um solche Fälle zu überprüfen.

## **3 Middleware**

### **3.1 Indexstruktur**

#### **3.1.1 Anforderungen**

Der Index soll eine effiziente Suche nach Mutationen für das Frontend ermöglichen. Während des Programmstarts wird der Index aus der vorhandenen Datenbank aufgebaut und steht dann so lange zur Verfügung, bis das Programm beendet wird. Im Index wird mit Intervallgrenzen gesucht und der Index gibt alle Intervalle zurück, die komplett innerhalb des angegebenen Intervalls liegen.

Der Index wird verteilt aufgebaut und liegt auf 4 virtuellen Maschinen verteilt. Die für den Index spezifizierten Funktionen sprechen immer einen Teilindex an. Der Aufruf der Funktionen wird über den IndexController erfolgen, der immer alle 4 Teilindizes ansprechen wird.

#### **3.1.2 Funktionen und Datenstrukturen**

Die Funktionen des Indexes variieren in ihrem Ablauf je nach gewählter Indexstruktur. Momentan existieren 3 Varianten, die getestet werden. Es ist nicht ausgeschlossen, dass weitere Strukturen im Laufe der Entwicklung getestet werden. Da die Dauer des Indexaufbaus für den Endnutzer nicht relevant ist hängt die Auswahl der letztendlich genutzten Struktur lediglich von der Geschwindigkeit der Suchanfragen ab. Im folgenden werden die Such-und Einfüge-Operationen basierend auf den jeweiligen Indexstrukturen beschrieben

## IntervallTree

In diesem Fall basiert die Indexstruktur auf einem Intervallbaum.

Hierfür wird die frei zugängliche Bibliothek `IntervallST.java` der Universität Princeton genutzt. Beide Funktionen haben hier lediglich die Aufgabe als Interface zu den zugehörigen Bibliotheksfunktionen zu dienen: `contains()` zur Suche und `put()` zum Einfügen.

Das Suchergebnis wird nach möglichen Filtern gefiltert. Bei  $n$  Intervallen und einer Such-Ergebnisliste der Größe  $m$  ergibt sich eine Komplexität von  $O(\log n + m)$ .

Die Bibliothek muss noch angepasst werden, damit das Einfügen von gleichen Intervallen möglich ist ohne, dass das zuerst eingefügte Intervall gelöscht wird.

```
search()
found intervalls = contains(intervall);
forall the elements in found intervalls do
    | if element corresponds to specified filters then
    | | add element to answer list;
    | end
    | return answer list;
end
```

```
addMutation()
if mutation is already in index then
    | return "index already contains mutation";
end
put();
return "added mutation";
```

### Suchbaumbasierter Index

Da ein Großteil der Intervalle einstellig bzw sehr kurz sind bietet sich ein einfacher binärer Suchbaum als Datenstruktur an.

Dieser wird um einen Iterator erweitert, damit effizient eine Menge an Knoten ausgewählt werden kann.

Hierfür wird die Java-Klasse TreeMap verwendet, die eine Suche in logarithmischer Zeit ermöglicht. Die Ergebnismenge wird einmal durchlaufen, Mutationen deren Endpunkt außerhalb des gesuchten Intervalls liegen werden dabei entfernt und jeder Knoten wird nach den angegebenen Filtern gefiltert. Bei  $n$  Intervallen und einer Such-Ergebnismenge von  $m$  Intervallen ergibt sich eine Laufzeit von  $O(\log n + m)$ .

Auch in dieser Implementation dienen die Funktionen als Interface zu den jeweiligen Funktionen der genutzten Klasse: `submap()` zur Suche und `put()` zum Einfügen von Objekten).

```
search()
found intervalls = submap(intervall);
forall the elements in found intervalls do
|   if element corresponds to specified filters then
|   |   add element to answer list;
|   end
|   return answer list;
end
```

```
addMutation()
if mutation is already in index then
|   return „index already contains mutation“;
end
put();
return „added mutation“;
```

### Arraybasierter Index

Sollte die Anzahl der Mutationen groß genug sein, dass sie mit Integer-Variablen darstellbar ist, so bietet sich unter Umständen auch ein arraybasierter Index an.

Dieser speichert alle Mutationen aufsteigend sortiert nach ihrem Anfangspunkt. Wird nun nach einem Intervall gesucht, so iteriert er über das Array beginnend bei der Mutation, deren Startwert noch im gesuchten Intervall liegt. Dabei wird für jede Mutation überprüft, ob ihr Endwert noch im gesuchten Intervall liegt. Ist dem so wird sie zur Ergebnismenge hinzugefügt. Hierbei können auch direkt die Filter überprüft werden.

Es so lange iteriert, bis der Startwert aller folgenden Mutationen größer, als der vom Nutzer angegebene Endwert ist.

Falls Mutationen an der gleichen Stelle beginnen, so verschieben sich alle folgenden Mutationen in der Liste, da in aufeinanderfolgenden Zellen gleiche Startintervalle gespeichert werden müssen. Es muss also ermittelt werden, wo sich die erste im Intervall liegende Mutation befindet. Ein Verfahren hierfür wird noch ermittelt

Bei  $m$  Mutationen, deren Startwert sich im gesuchten Intervall befinden, liegt die Laufzeit bei  $O(m + \epsilon)$ , wobei  $\epsilon$  davon abhängt, wie die erste Mutation ermittelt wird. Es kann aber davon ausgegangen werden, dass  $\epsilon$  einen geringen Anteil an der Laufzeit ausmachen wird.

```
search()
find index x of first mutation that lies in intervall;
while starting point of mutation at index x lies in search intervall do
    if endpoint of mutation at index x lies in search intervall and mutation
        corresponds to specified filters then
        | add mutation to answer list;
    end
    x=x+1;
end
return answer list;
```

```
addMutation()
if mutation is already in index then
    | return "index already contains mutation";
end
insert mutation at corresponding index and adjust array properly;
return „added mutation“;
```

## 3.2 IndexController

### 3.2.1 Anforderungen

Der IndexController nimmt Suchanfragen entgegen, leitet diese an die 4 Teilindizes weiter und fügt die Teilergebnisse wieder zusammen.

Falls eine Anfrage intern in mehrere Teilanfragen aufgeteilt werden sollte, weil z.B. ein Gen, nach dem gesucht wird an mehreren Stellen auftreten kann, leitet der IndexController alle Teilanfragen sequentiell an die Indizes weiter, fügt die Teilergebnisse zusammen und schickt die Ergebnismenge an den QueryReceiver zurück.

### 3.2.2 Funktionen

**answerQuery(int[] intervals,String[] Sources,int[]filter)** Die Funktion erhält mehrere Listen als Parameter, die die nötigen Informationen für die einzelnen Anfragen beinhalten. Jeweils 2 aufeinanderfolgende Einträge in der Intervall-Liste beschreiben den Start- und Endpunkt der gesuchten Intervalle. Die Einträge in den anderen Listen werden für alle Anfragen genutzt

Die Anfragen werden sequentiell an die 4 Teilindizes weitergeleitet und einzelnen Ergebnisse konkateniert und in einer Liste zurückgegeben.

```
forall the queries in parameter list do
    forall the sub indices do
        | answer list = index.search();
    end
    concatenate all answer lists;
end
concatenate answer lists of each sub query;
return answer list;
```

**buildIndex()** Die Funktion wird bei Programmstart ausgeführt und baut auf Basis der Datenbank die 4 Teilindizes auf.

Für jede Mutation wird zufällig entschieden in welchen Index sie eingefügt wird.

```
forall the elements in database do
    if element is already in index then
        | skip element;
    end
    choose which sub index to insert into;
    index.addMutation();
end
return index built";
```

### 3.3 Klassen-Diagramm

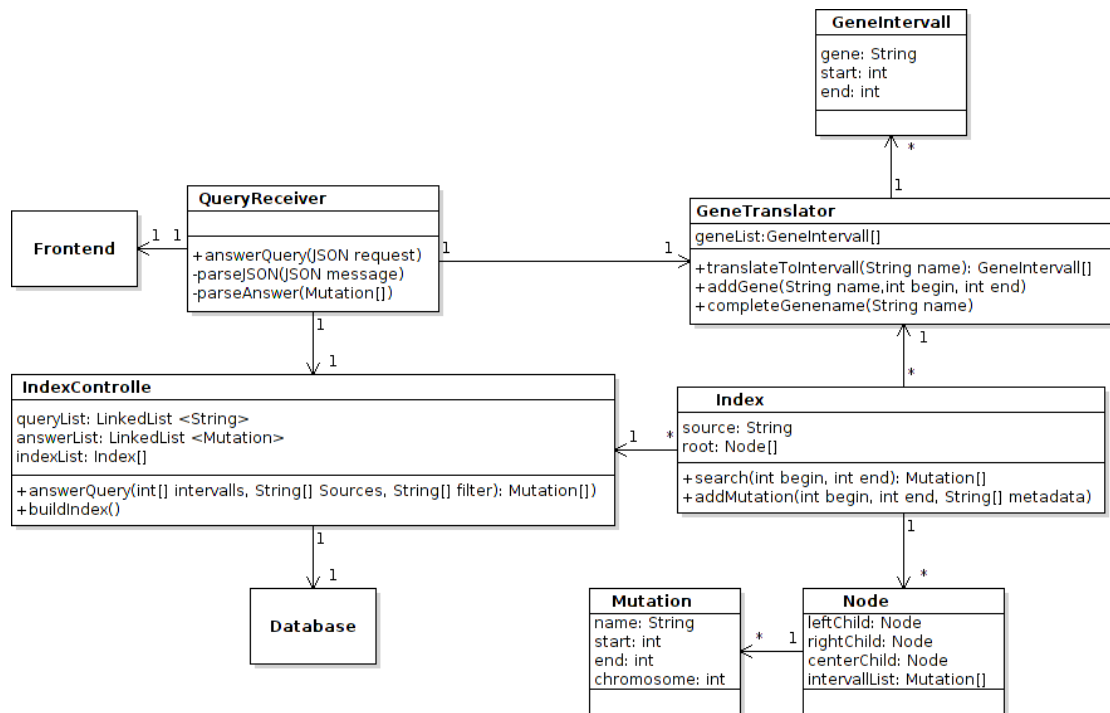


Abbildung 5: TODO



### 3.4 Sequenzdiagramme

#### 3.4.1 Intervall-Suche

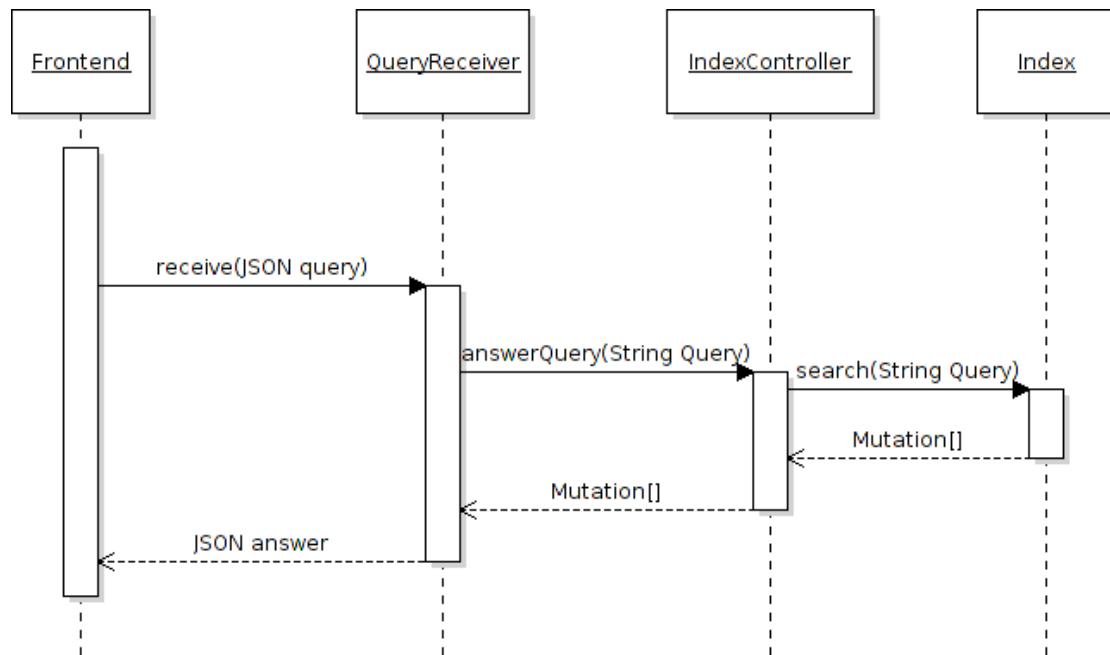


Abbildung 6: TODO

### 3.4.2 Names-Suche

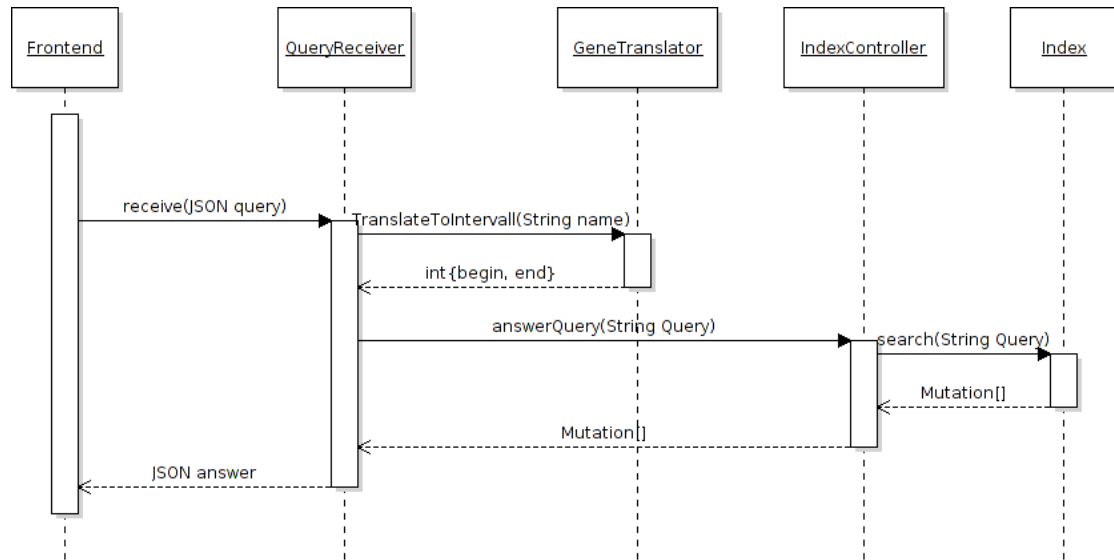


Abbildung 7: TODO

### 3.4.3 Suche nach Gennamen

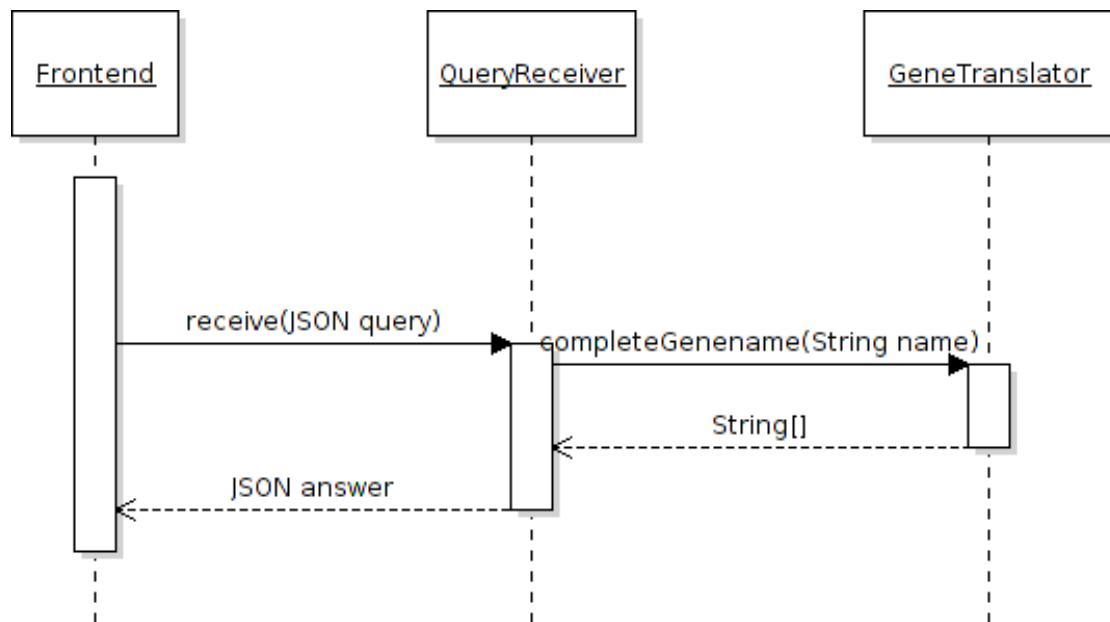


Abbildung 8: TODO

## 3.5 Unit-Tests

### 3.5.1 QueryReceiver

#### Testfall 1 - erfolgreiche Intervallanfrage ohne Metadaten

```
Eingabe
{
  "source": The Cancer Atlas,\usepackage[utf8]{inputenc}
  "chromosome": 3,
  "position": {"from": 100, "to": 200},
  "zoom": 1,
  "subindex": ,
  "hasDetail": (false)
}

Ausgabe
{
  {
    "source:" The Cancer Atlas,
    "chromosome": 3,
    "position": {"from": 100, "to": 200},
    "details": { "refseq": ,
                  "mutations": [{ "name": ,
                                   "position": {"from": , "to": },
                                   "metadata":
                                   },]},
    "graph": {
      { "subintervall": Anzahl der Subintervalle,
        "counts": Anzahl der Ergebnisse
      }
    }
  }
}
```

#### Testfall 2 - erfolgreiche Intervallanfrage mit Metadaten

```
Eingabe
{
  "source": The Cancer Atlas,
  "chromosome": 3,
  "position": {"from": 100, "to": 200},
  "zoom": 1,
  "subindex": ,
  "hasDetail": (true)
}
```

```

Ausgabe
{
  {
    "source:" The Cancer Atlas,
    "chromosome": 3,
    "position": {"from": 100, "to": 200},
    "details": { "refseq": Referenzsequenz,
                  "mutations": [{Mutation1},{Mutation2},...]},
    "graph": {
      { "subintervall": ,
        "counts":
      }
    }
  }
}

```

### Testfall 3 - erfolglose Intervallanfrage

```

Eingabe
{
  "source": The Cancer Atlas,
  "chromosome": 3,
  "position": {"from": 100, "to": 200},
  "zoom": 1,
  "subindex": ,
  "hasDetail": (true)
}

```

```

Ausgabe
{
  {
    "source:" The Cancer Atlas,
    "chromosome": 3,
    "position": {"from": 100, "to": 200},
    "details": { "refseq": ,
                  "mutations": ...},
    "graph": {
      { "subintervall": ,
        "counts":
      }
    }
  }
}

```

#### Testfall 4 - erfolgreiche Namensanfrage

```
Eingabe
{
  "source": The Cancer Atlas,
  "chromosome": 3,
  "search": "Gen im Cancer Atlas"
}

Ausgabe
{
  "source": The Cancer Atlas,
  "chromosome": 3,
  "search": "Gen im Cancer Atlas",
  "position": {"from": Startposition, "to": Endposition}
}
```

#### Testfall 5 - erfolglose Namensanfrage

```
Eingabe
{
  "source": The Cancer Atlas,
  "chromosome": 3,
  "search": "Gen, dass nicht im Cancer Atlas ist"
}

Ausgabe
{
  "source": The Cancer Atlas,
  "chromosome": 3,
  "search": "Gen im Cancer Atlas",
  "position": " "
}
```

#### Testfall 6 - fehlerhafte Anfrage

```
Eingabe
{
  "source": ""
}

Ausgabe
{
  "answer": "unknown format"
}
```

### 3.5.2 GeneTranslator

Der GeneTranslator hat 2 Aufgaben. Zum einen soll er während der Indexerstellung mit Inhalt (also Gennamen und zugehörigen Intervallen) befüllt werden, zum anderen soll eine Suche nach Gennamen in ihm möglich sein.

#### **Testfall 1 - addGene - Einfügen in Datenstruktur**

Eingabe: Testgen, 350, 500

Ausgabe: Dass Gen sollte in den Baum eingefügt sein und per searchForGene() findbar sein

#### **Testfall 2 - addGene - Doppeltes Einfügen in Datenstruktur**

Eingabe: Testgen, 350, 500

Testgen, 350, 500

Ausgabe: Dass Gen sollte nur einmal in die Datenstruktur eingefügt werden. Ausgabe, dass das Gen bereits in der Struktur vorhanden ist.

#### **Testfall 3 - addGene - Aufruf ohne Parameter**

Eingabe: [...]

Ausgabe: Fehlermeldung über parameterlosen Aufruf

#### **Testfall 4 - tranlateToIntervall - erfolgreiche Suche**

Eingabe: Testgen (befindet sich bereits in Datenstruktur)

Ausgabe: 350, 500

#### **Testfall 5 - tranlateToIntervall - erfolglose Suche**

Eingabe: Testgen2 (befindet sich nicht in Datenstruktur)

Ausgabe: Fehlermeldung über erfolglose Suche

#### **Testfall 6 - tranlateToIntervall - Aufruf ohne Parameter**

Eingabe: [...]

Ausgabe: Fehlermeldung über Parameterlosen Aufruf

#### **Testfall 7 - completeGeneName - erfolgreiche Suche mit einem Ergebnis**

Eingabe: Test (Testgen1 befindet sich bereits in Datenstruktur)

Ausgabe: Testgen1

#### **Testfall 8 - completeGeneName - erfolgreiche Suche mit mehreren Ergebnissen**

Eingabe: Test (Testgen1 und Testgen2 befinden sich bereits in Datenstruktur)

Ausgabe: Testgen1, Testgen2

**Testfall 9 - completeGeneName - erfolglose Suche**

Eingabe: Test (Es befindet sich kein Gen mit Präfix Test in der Datenstruktur)

Ausgabe: Fehlermeldung über erfolglose Suche

**3.5.3 IndexController****Testfall 1 - answerQuery - einfache Intervallanfrage**

Eingabe: 1,3,100,200

Ausgabe: zum Intervall gehörende Mutationsobjekte

**Testfall 2 - answerQuery - komplexere Intervallanfrage**

Eingabe: 1,3,100,200 ; 1,3,150,350

Ausgabe: eine Mutationsliste mit den Ergebnissen beider Anfragen

**Testfall 3 - answerQuery - unvollständige Intervallanfrage**

Eingabe: 1,100,200

Ausgabe: Fehlermeldung über unvollständige Anfrage

**Testfall 4 - answerQuery - leere Anfrage**

Eingabe: [...]

Ausgabe: Fehlermeldung über leere Anfrage

**Testfall 5 - answerQuery - überspezifizierte Intervallanfrage**

Eingabe: 1,3,100,200,300,400

Ausgabe: Fehlermeldung über überspezifizierte Anfrage

**Testfall 6 - buildIndex - erfolgreicher Indexaufbau**

Eingabe: [...] (Datenbank ist erreichbar)

Ausgabe: Erfolgreich gebauter Index

**Testfall 7 - buildIndex - erfolgloser Indexaufbau**

Eingabe: [...] (Datenbank ist nicht erreichbar)

Ausgabe: Fehlermeldung über erfolglosen Indexaufbau



### 3.5.4 Intervallbaum

#### 1. Intervalle einfügen:

Das Intervall muss im Baum an der richtigen Stelle eingefügt werden und der Baum muss gegebenenfalls neu balanciert werden (z.B. wie ein AVL-Baum).

Bsp.: Einfügen des Intervalls [12,15]

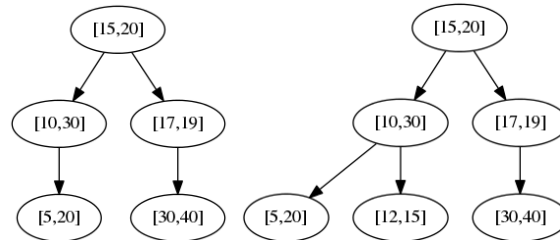


Abbildung 9: TODO

#### 2. Intervall mit Startpunkt > Endpunkt einfügen:

Wenn ein Intervall (S,E) mit  $S > E$  eingefügt wird, dann sollte unser Programm eine Fehlermeldung ausgeben und darauf hinweisen, dass die Grenzen für das Intervall nicht korrekt sind.

Bsp.: Einfügen des Intervalls [20,10] in einen beliebigen Baum.

#### 3. Intervall mit Start- bzw. Endpunkt außerhalb des betrachteten Zahlenbereichs:

Wenn ein Intervall in dem Baum eingefügt werden soll, das teilweise oder vollständig außerhalb unseres Zahlenbereichs liegt (Länge des Genoms), dann muss es eine Fehlermeldung geben, die dem Nutzer mitteilt, dass der gültige Zahlenbereich überschritten wurde.

Bsp.: Einfügen des Intervalls [-5,7] in einen beliebigen Baum.

#### 4. Schon vorhandenes Intervall einfügen:

Duplikate sollen von unserem Baum nicht gespeichert werden, d.h. es wird kein neuer Knoten hinzugefügt, sondern die Informationen (bei uns also Pointer auf Dateien) des neuen Knotens müssen im bereits vorhandenen Knoten mitgespeichert werden.

Bsp.: Einfügen des Intervalls  $[15,20]$  in den folgenden Baum

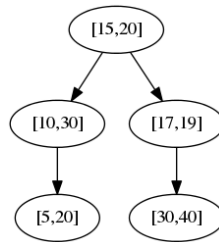


Abbildung 10: TODO

#### 5. Suche nach vorhandenem Intervall:

Bei der Suche sollen alle Intervalle ausgegeben werden, die das gesuchte Intervall in irgendeinem Punkt überlappen.

Bsp.: Suche im folgenden Baum

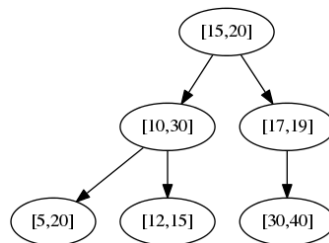


Abbildung 11: TODO

Suche  $[4,5] \Rightarrow$  gib  $[5,20]$  aus

Suche  $[25,35] \Rightarrow$  gib  $[10,30]$  und  $[30,40]$  aus

Suche  $[20,20] \Rightarrow$  gib  $[15,20]$  und  $[5,20]$  aus

### 3.6 Stresstests

Der Stresstest hat zum Ziel herauszufinden, wie lange die durchschnittliche Query-Laufzeit ist.

Ziel des Systems ist es eine Laufzeit von unter einer Sekunde zu erreichen.

Um dies zu überprüfen werden mehrere Anfragen der gleichen Art (Intervallsuche, Gennamensuche, Präfixsuche) sequentiell gestellt und die Antwortzeit gemessen. Alle Anfragen sollen in unter einer Sekunde eine Antwort erzielen.

## 4 Schnittstellenspezifikation: Middleware - Frontend

Bei der Kommunikation zwischen Middleware und Frontend ist ein Austausch von JSON über die HTTP-Methode GET vorgesehen. An die Middleware werden folgende Informationen geschickt:

*Quellen; Chromosom; Position; Suche; Präfix-Flag; Zoomstufe; Anzahl der Subintervalle; HasDetail-Flag.*

Von der Middleware werden bei bestimmten Flags, folgende Daten erwartet:

Wenn *IsPräfix-Flag* = **true**: Array aller Präfixe

Wenn *IsPräfix-Flag* = **false**: Name und Position der Suche

Wenn *HasDetail-Flag* = **true**: RefSeq + Mutationen mit MetaDaten

Wenn *HasDetail-Flag* = **false**: aggregierte Mutationen (durch Zoomstufe+Anzahl der Subintervalle)

### 4.1 Mögliche Anfragen

#### 4.1.1 Suchanfrage JSON

GUI  $\Rightarrow$  Middleware

```
{
  "source": ["src1", "src2", ...],
  "chromosome": int x,
  "search": String a
  "isPrefix": (true|false)
}
```

GUI  $\Leftarrow$  Middleware

```
{
  "source": ["src1", "src2", ...],
  "chromosome": int x,
  "search": String a,
  "prefix": ["String a", "String b", ...],
  "position": {"from": int x, "to": int y}
}
```

**Bemerkung** Wenn das *isPrefix* den Wert 'true' besitzt, wird "prefix" befüllt mit einem Array aller möglichen Präfixe.

Wenn das *isPrefix* den Wert 'false' besitzt, wird "prefix" nicht befüllt, aber "search" und "position" werden dann befüllt.

#### 4.1.2 Intervallanfrage JSON

GUI  $\Rightarrow$  Middleware

```
{
  "source": String s,
  "chromosome": int x,
  "position": {"from": int x, "to": int y},
  "zoom": int y,
  "subindex": int z,
  "hasDetail": (true|false)
}
```

GUI  $\Leftarrow$  Middleware (pro Quelle)

```
{
  {
    "source:" String s,
    "chromosome": int x,
    "position": {"from": int x, "to": int y},
    "details": { "refseq": String b,
                  "mutations": [{ "name": String s,
                                   "position": {"from": int x, "to": int y},
                                   "metadata": {...}
                                },{...},...]},
    "graph": {
      { "subintervall": int x,
        "counts": int y
      },
      {...}
    }
  }
}
```

**Bemerkung** Wenn das *HasDetail-Flag* 'true' ist, dann wird "detail" befüllt und "graph" bleibt leer.

Wenn das *HasDetail-Flag* 'false' ist, dann wird "graph" befüllt und "detail" bleibt leer.

Mit "subindex" ist die Anzahl der *Subintervalle* gemeint. Als *Subintervalle* werden kleine Bereiche des Intervalls bezeichnet. Diese werden dazu verwendet um einen Graphen detaillierter darzustellen.

## 4.2 Beispiel der Kommunikation mit JSON

**Suchanfrage** Es wird eine Suchanfrage des Users getätigt. Die *Suche* wird mit dem Wert *"FOXP2"* befüllt.

GUI  $\Rightarrow$  Middleware

```
{
  "source": ,
  "chromosome": ,
  "search": "FOXP2"
  "isPrefix": false
}
```

GUI  $\Leftarrow$  Middleware

```
{
  "chromosome": 7,
  "search": "FOXP2",
  "prefix": [],
  "position": {"from": 114086310, "to": 114693772}
}
```

Darauf folgt eine Intervallanfrage mit der *Zoomstufe* von '1,000,000bp' und der Anzahl der Subintervalle von '200'.

GUI  $\Rightarrow$  Middleware

```
{
  "source": "Source 1",
  "chromosome": 7,
  "position": {"from": 114086310, "to": 114693772},
  "zoom": 1,000,000,
  "subindex": 200,
  "hasDetail": false
}
```

GUI  $\Leftarrow$  Middleware

```
{
  {
    "source:" "Source 1",
    "chromosome": 7,
    "position": {"from": 114086310, "to": 114693772},
    "details": {},
    "graph": {
      { "subintervall": 114091310,
        "counts": 233
      },
      {
        "subintervall": 114096310,
        "counts": 122
      }
    }
  }
}
```

```

    },
    {
      "subintervall": 114101310,
      "counts": 89
    },
    {
      "subintervall": 114106310,
      "counts": 234
    },
    {...},
    ...
  }
}
}

```

**Intervallsuche** Der User gibt ein *Intervall* an und möchte sich in der niedrigsten *Zoomstufe* die Details anschauen.

GUI  $\Rightarrow$  Middleware

```

{
  "source": "Source_1",
  "chromosome": 14,
  "position": {"from": 5465, "to": 5665},
  "zoom": 200,
  "subindex": 0,
  "hasDetail": true
}

```

GUI  $\Leftarrow$  Middleware

```

{
  {
    "source:" Source 1,
    "chromosome": 14,
    "position": {"from": 5465, "to": 5665},
    "details": { "refseq": "CCCTGAGAAAGACGCCCTGAGCGGGTAACGTCAACTGTCTG
      TGGGATTGGAAGAACTTGGTGGGGTTCCACATATCATGAT
      CACGTAGAACACAATAAAGAAAATCTCCGGAGCGTGAATT
      AAAGTGAAGTCAATGATGCCCTAATCTGGATAGCTTTGTCAT",
    "mutations":
      [{ "name": "RS12334",
        "position": {"from": 5480, "to": 5498},
        "metadata": {"sequence": "TCGGTATTCTACGTCTGC"}
      },
      { "name": "RBDDFG23",
        "position": {"from": 5560, "to": 5567},
        "metadata": {"sequence": "GGGCACT"}
      }
    ]
  }
}

```

```

    { "name": "RS12334",
      "position": {"from": 5599, "to": 5610},
      "metadata": {"sequence": "TGACGAGATCG"}
    }
    { "name": "RS12334",
      "position": {"from": 5650, "to": 5655},
      "metadata": {"sequence": "CGACC"}
    }
  ],
  "graph": {}
}

```



## 5 Frontend

### 5.1 Use Cases

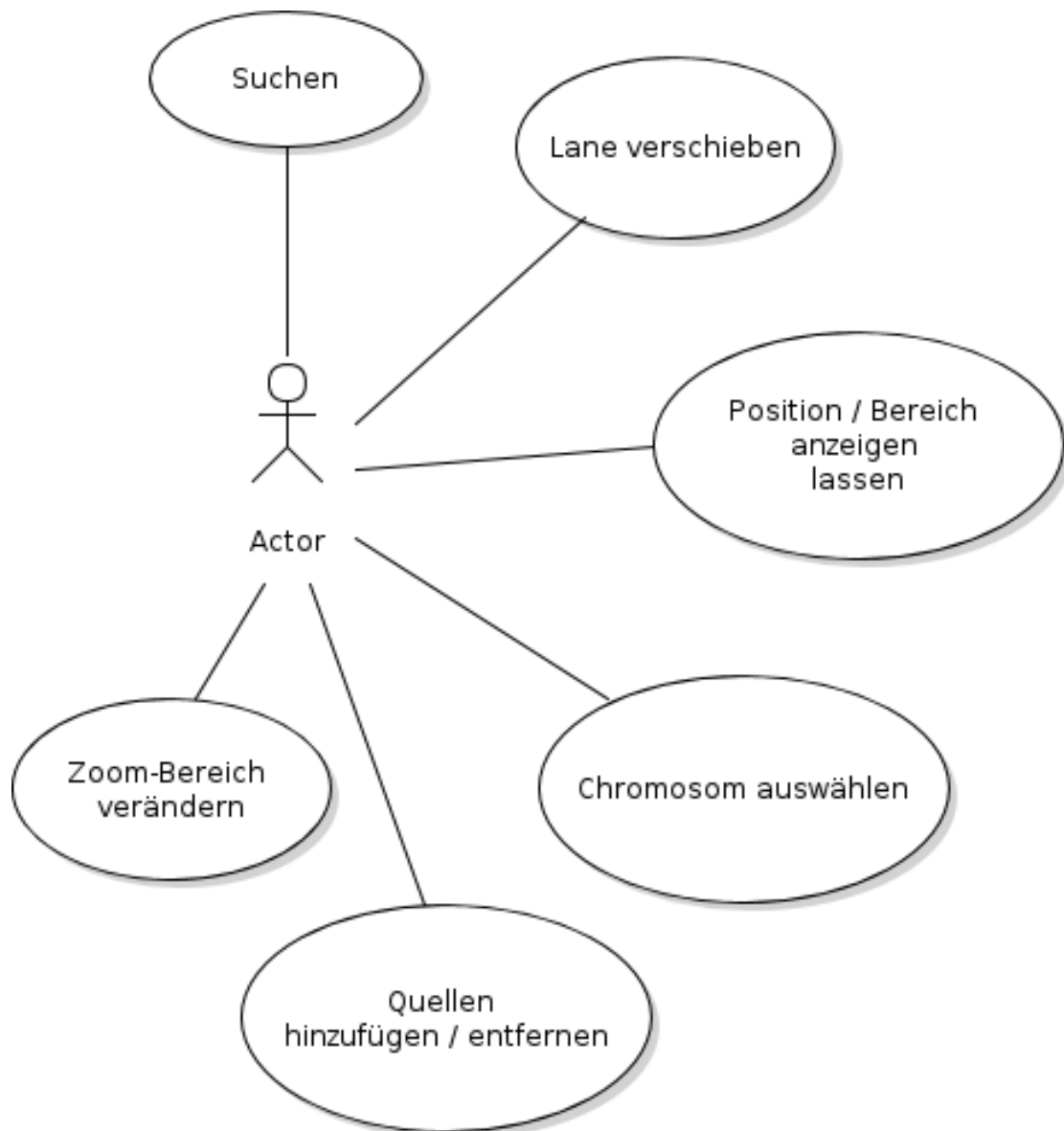


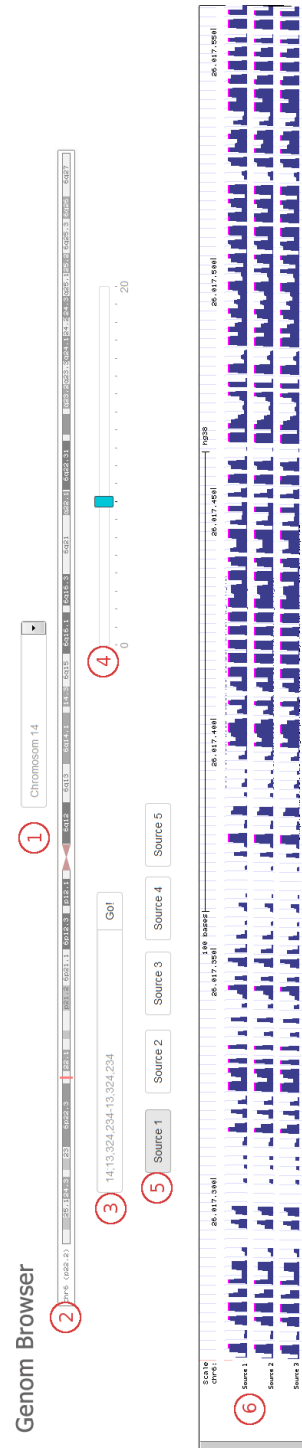
Abbildung 12: Das Use-Case-Diagramm listet die Möglichkeiten der Interaktionen mit der GUI auf.

**Erläuterung des Use-Case (mit Bezug auf das Sequenzdiagramm)** Der Benutzer hat die Möglichkeit mit der GUI zu interagieren. Durch das Suchfeld hat der Benutzer die Möglichkeit nach einem Gen zu **suchen** oder sich eine(n) **Position / Bereich anzeigen lassen**. Zur genaueren Auswahl kann dieser sich eine **Chromosom auswählen**. Durch das **hinzufügen / entfernen** einer oder mehrerer Quellen, kann der Benutzer das Angezeigte weiter eingrenzen. Durch das **Verändern des Zoom-Bereich** kann der Benutzer die Anzahl der angezeigten Basenpaare bestimmen. Durch einfaches **Verschieben der Lane** ist es dem Benutzer möglich, den genauen Bereich zu bestimmen, den dieser betrachten möchte.

## 5.2 Mock-Ups der Benutzerschnittstelle

Die Mock-Ups (Siehe Abbildung 13 und 14) werden durch folgende Elemente definiert:

- 1: Chromosom-Auswahl** Durch die Chromosom-Auswahl kann das zu betrachtende Chromosom ausgewählt werden. Auch als Information des aktuell angezeigten Chromosoms dient dieses Element.
- 2: Idiogramm** Dieses Element dient als Karte zur Orientierung auf dem Chromosom. Es kann auch verwendet werden um an eine bestimmte Stelle im Chromosom zu gelangen.
- 3: Suche/Position** Dieses Eingabefeld kann zur genauen Eingabe einer Position auf einem Chromosom oder zur Suche nach bestimmten Genen genutzt werden.
- 4: Zoomregler** Durch den Zoomregler kann die Zoomstufe angepasst werden. Dieses Element besitzt sieben Zoomstufen, diese gehen von den Basenpaaren bis hin zu 10MB.
- 5: Quellen-Auswahl** Durch das auswählen der Quellen kann eine Lane (siehe 6) hinzugefügt werden oder auch abgewählt werden.
- 6: Lanes** Diese Elemente dienen zur Darstellung der Mutationen. Pro aktiver Quelle wird eine Lane angezeigt. Durch die Zoomstufen werden die Lanes in zwei Darstellungen definiert:
  - a) detaillierte Darstellung** Auf den Lanes werden die Basenpaare angezeigt. Die Mutationen werden als farbige Stellen auf dem Referenzgenom markiert. Durch anklicken der Mutation-Stellen werden die Metadaten angezeigt.
  - b) aggregierte Darstellung** Ab der Zoomstufe 2 werden die Mutationen aggregiert dargestellt. Durch diese Darstellung wird angezeigt wie viele Mutationen in einem Bereich (Intervall) vorkommen.



### 5.3 Technologie und Klassenstruktur

Das Frontend ist eine Webapplikation, welche als Single Page App (SPA) implementiert ist. Die Darstellung der Inhalte und ihrer Interaktionsmöglichkeiten wird somit komplett im Benutzer-Client (Web Browser) durchgeführt. Daher basiert der Technologie-Stack auf JavaScript als Programmiersprache. Mit dem Backend wird über eine HTTP Schnittstelle kommuniziert, welche in Kapitel 4 näher erläutert wurde.

Eine grundlegende Technologie der Applikation ist das JavaScript-Framework **React.JS**, welches zum Rendern der verschiedenen Bestandteile des Frontendes verwendet wird. Dabei handelt es sich nicht um ein klassisches MVC-Framework, sondern um eine deklarative Art die Applikation in kleinere Komponenten zu modularisieren, den Zustand dieser Komponenten zu beschreiben und automatisch im Markup (HTML) darzustellen. Updates an den präsentierten Daten führen somit direkt zur Änderung des Frontendes. Bei einem Aufruf der Webseite werden zunächst alle benutzten JavaScript-Bibliotheken geladen, darunter **React.JS**. Als nächstes wird eine zentrale React-Komponente *GUIComponent* als Klasse initialisiert, welche für den globalen Zustand der Applikation verantwortlich ist. Diese lädt zunächst alle benötigten Daten vom *Backend* mittels des *DataProvider*-Services. Dieser ist ein Singleton-Objekt (ähnlich einer statischen Klasse in Java), welcher die eigentlichen HTTP Requests durchführt, die asynchron eintreffenden Responses aufbereitet und an die *GUIComponent* zurück leitet, wo sie gespeichert und dargestellt werden.

Die *GUIComponent* ist die Wurzel für einen Baum aus weiteren Komponenten ("View-Models"). Abhängig von den in der *GUIComponent* gespeicherten Daten, werden Kind-Komponenten gerenderet, beispielsweise die *Lane*-Klasse, welche eine Lane aus Mutationsdaten und dem Referenzgenom darstellt. Sobald in der *GUIComponent* beispielsweise aktualisierte Mutationsdaten vorliegen, werden diese an die jeweilige *Lane*-Komponente weitergegeben, diese darauf neu gerendert werden.

Die meisten Komponenten verfügen außerdem über einen Event-Listener, um Benutzer-Interaktionen zu verarbeiten. Diese werden an die übergeordnete *GUIComponent* weitergegeben und haben unter Umständen Einfluss auf den Gesamtzustand der Applikation. Gegebenenfalls ruft die *GUIComponent* den *DataProvider*-Service auf, um neue Daten aus dem *Backend* abzufragen.

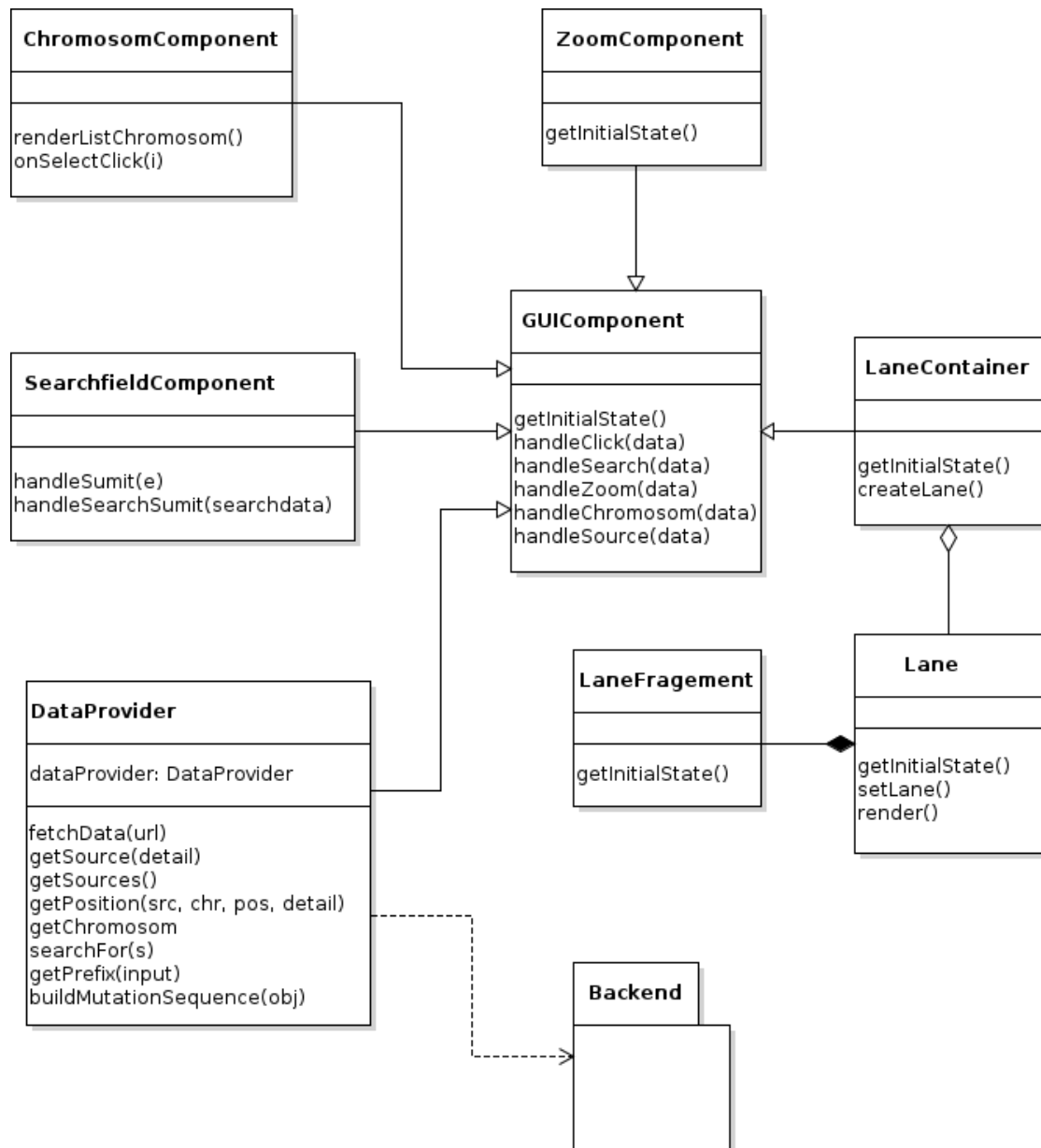


Abbildung 15: Das Klassen-Diagramm der GUI.

## 5.4 Sequenzdiagramme

In Abbildung 16 (CHECK\_LATER) werden die Interaktionsmöglichkeiten des Users auf die *Lane* beschrieben. Beim Bewegen der Lane und dem Zoomen der Lane findet jeweils eine Anfrage an das *Backend* statt. Beim Auswählen der Quelle findet ebenfalls eine Anfrage an das *Backend* statt und es wird eine neue *Lane* angezeigt, jedoch wird beim entfernen einer Quelle lediglich die Lane ausgeblendet.

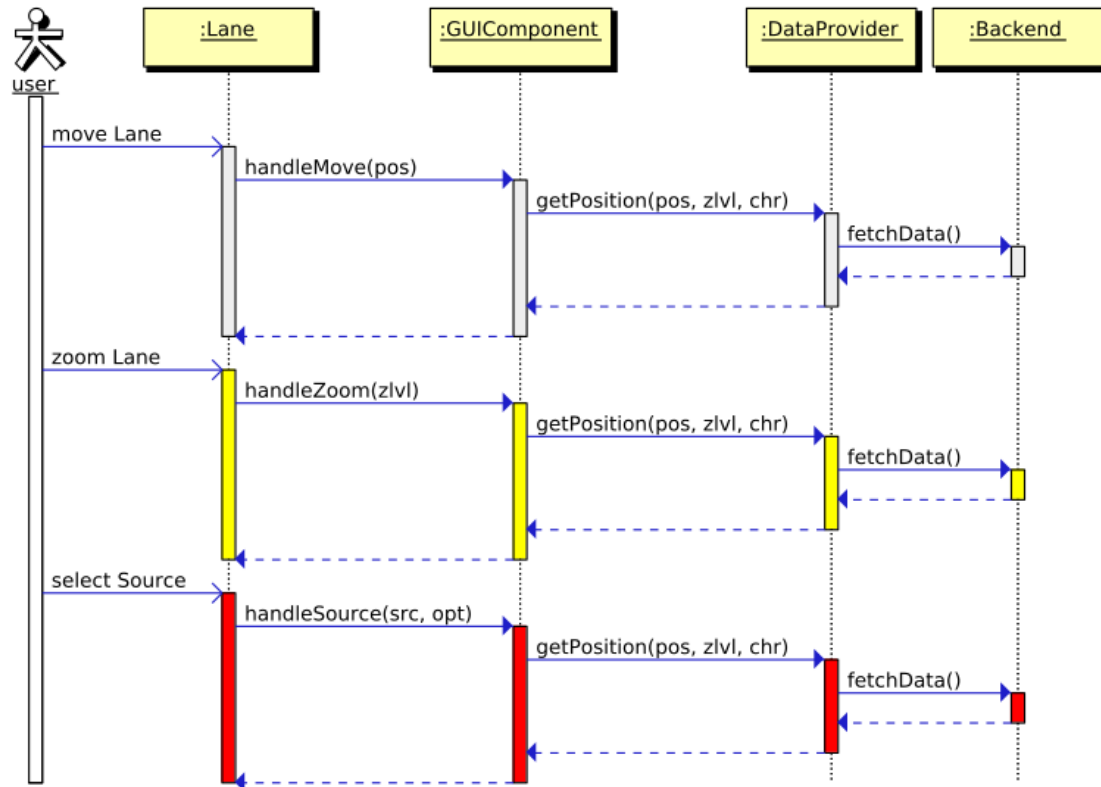


Abbildung 16: Das Sequenzdiagramm der Lane-Methoden und deren Verlauf.

In Abbildung 17 (CHECK\_LATER) wird die *GUIComponent* direkt angesprochen. Wird ein Chromosom ausgewählt, so werden die Mutationdaten für die letzte Zoomstufe geholt und werden aggregiert dargestellt. Beim Eingeben einer Position oder einer Suche werden nur so viele Mutationdaten geholt, wie für die aktuelle Zoomstufe konfiguriert ist.

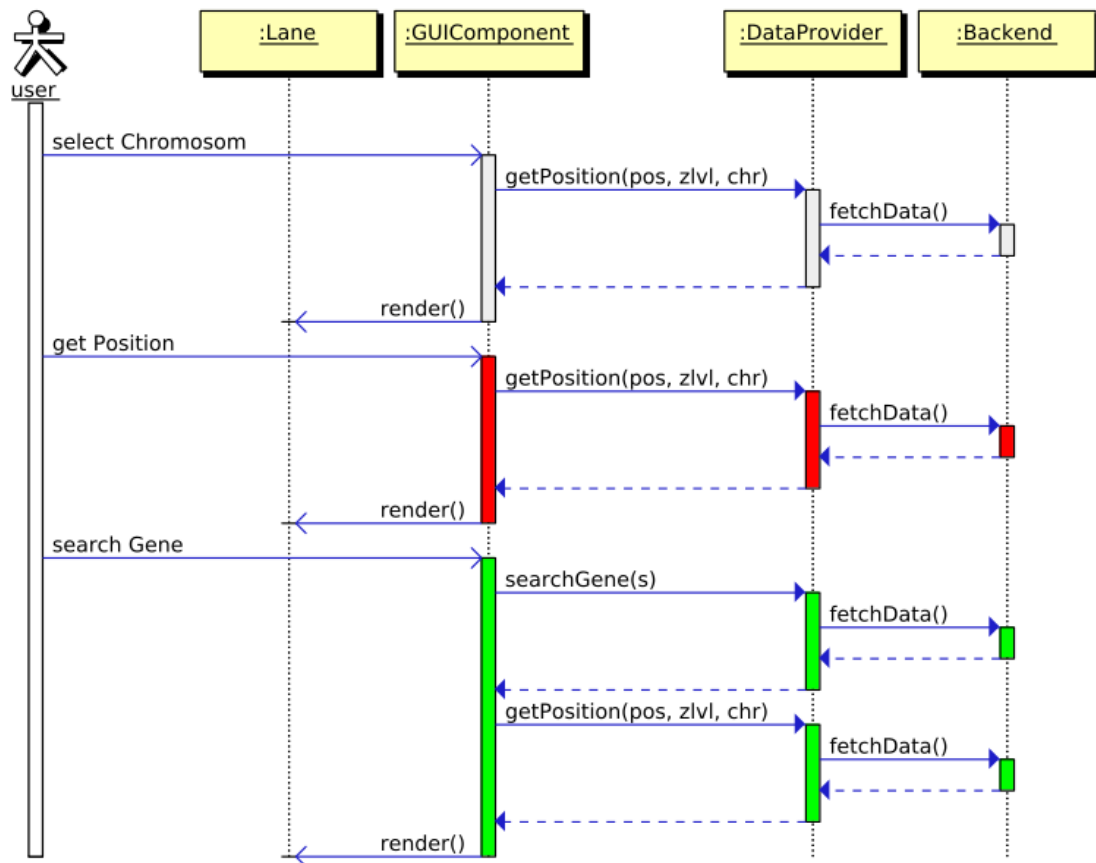


Abbildung 17: Das Sequenzdiagramm der GUIContainer-Methoden und deren Verlauf.

## 5.5 Unit-Tests

Wie für dynamische Sprachen üblich, werden alle öffentlichen JavaScript-Funktionen mit Unit-Tests abgedeckt. Dabei kommt als Testing-Framework `jasmine` zusammen mit dem Test-Runner `Karma` zum Einsatz. Die React-Komponenten werden mithilfe der `ReactTestUtils` getestet.

## 5.6 Integration-Tests

Die folgenden Test-Szenarien werden mit einem Selenium-basierten E2E-Testframework automatisiert durchgeführt.

### 5.6.1 Suchfunktion

1. Wenn der Benutzer eine leere Suche startet, dann soll er eine entsprechende Fehlermeldung angezeigt bekommen.
2. Wenn der Benutzer eine Suche mit falscher Eingabe startet, dann soll er eine entsprechende Fehlermeldung angezeigt bekommen.
3. Wenn der Benutzer nach einem gültigen Intervall sucht, dann wird automatisch die Zoomstufe auf dieses Intervall angepasst.
4. Wenn der Benutzer nach einem vorhandenem Gene sucht, dann wird automatisch die Zoomstufe auf dieses Intervall angepasst.

### 5.6.2 Quellen-Button

1. Wenn der Benutzer auf einen Quellen-Button drückt, dann wird ihm die entsprechende Quelle zusätzlich zu den bereits dargestellten Quellen, angezeigt.
2. Wenn der Benutzer auf den Quellen-Button einer bereits angezeigten Quelle drückt, wird die entsprechende Quelle nicht mehr angezeigt.

### 5.6.3 Quellen-Scroller

1. Als Benutzer kann man über horizontales Scrolling synchron durch die Quellen bewegen.

### 5.6.4 Zoom-slider Szenario

1. Wenn der Benutzer die Zoomstufe über den Slider ändert, dann werden die Quellen entsprechend der eingestellten Stufe dargestellt.
2. Wenn der Benutzer die feinste Zoomstufe einstellt, dann werden ihm die Basenpaare angezeigt.
3. Wenn der Benutzer eine andere Zoomstufe einstellt, dann werden ihm aggregierten Daten angezeigt.



### **5.6.5 Chromosom-Auswahl**

1. Der Benutzer kann über ein Dropdown aus einer Vorauswahl von Chromosomen auswählen.
2. Wenn der Benutzer ein Chromosom auswählt, dann wird die Quellen-Anzeige automatisch entsprechend des ausgewählten Chromosoms aktualisiert.
3. Wenn der Benutzer das bereits ausgewählte Chromosom erneut auswählt, dann passiert nichts.

### **5.6.6 Allgemein**

1. Wenn der Benutzer auf eine Anfrage warten muss, wird ihm dies durch einen Loading-Spinner signalisiert.