Hochschule für Technik und Wirtschaft Dresden

Fakultät Informatik/Mathematik

Masterarbeit

im Studiengang Angewandte Informatik,

Informations- und Kommunikationstechnologien

Thema:	Von der Einzelnutzer- zur Mehrnutzeranwendung: Implementierung einer Client-Server-Lösung für kollaborative Projektarbeit mit Echtzeitupdates in Java
eingereicht von:	Michael Danzig
eingereicht am:	
1. Betreuer:	Prof. Dr.-Ing. Jörg Vogt
2. Betreuer:	Michael Küttner (DMOS GmbH)

Inhaltsverzeichnis

1. Verzeichnis verwendeter Abkürzungen

Abkürzung	Bedeutung auf englisch	Bedeutung auf deutsch
RT	Real-Time	Echtzeit
OT	operational transformation	operationale Transformation*
STR	Sample Test Report	Musterprüfbericht*
JS	JavaScript	JavaScript
IC	integrated circuit	Integrierter Schaltkreis
CRDT	Conflict-free Replicated Data Types	konfliktfrei replizierte Datentypen*
CRTE	collaborative real-time editing	Echtzeit-Zusammenarbeit*
CEP	Complex Event Processing	Verarbeitung komplexer Ereignisse
ODM	object-document mapper	Objekt-Dokument-Zuordnung*
EDF	Electronic Development Flow	Elektronischer Entwicklungsablauf
IO	input and output	Eingabe und Ausgabe

*eigene, nicht offizielle Übersetzung des Autors

2. Verzeichnis verwendeter Begriffe und deren Bedeutung (Glossar)

Begriff	Bedeutung
Echtzeit	Echtzeit bezeichnet eigentlich die Eigenschaft eines Systems, bestimmte Aufgaben innerhalb einer gewissen Zeitspanne zuverlässig erledigen zu können. In dieser Arbeit werden jedoch keine verbindlichen Aussagen über die maximale Zeit für die Verarbeitungen bestimmter Aufgaben gemacht. Das Wort Echtzeit wird innerhalb dieser Arbeit in seiner umgangssprachlichen Bedeutung verwendet. Es wird damit lediglich ausgedrückt, dass ein bestimmtes Verfahren automatisch und spontan ausgelöst und ohne unnötigen Verzug abgearbeitet wird.
operational transformation	Als operational transformation bezeichnet man das Erfassen und Anwenden von Ereignissen beim Editieren von Texten oder anderen Daten. Diese Ereignisse bezeichnen Einfügungen und Löschungen und werden als Änderungen in Bezug auf den vorherigen Stand beschrieben.
Differential Synchronization	Differential Synchronisation ist das von Neil Fraser beschriebene Verfahren zum Synchronisieren eines Dokumentes zwischen einem Server und mindestens einem Client. Es ist als Publikation unter der Nummer doi:10.1145/1600193.1600198, als Internetseite unter https://neil.fraser.name/writing/sync/ und als Video unter https://www.youtube.com/watch?v=S2Hp_1jqpY8 aufgeführt.
Diff	Diff ist die Kurzform für das englische Wort „difference“. In dieser Arbeit wird damit ein Algorithmus bezeichnet, der Unterschiede von einem ursprünglichen Datenobjekt zu einem geänderten Datenobjekt ermittelt.
Patch	Ein Patch ist ein Algorithmus zur Aktualisierung von Daten anhand einer Liste von Änderungen.
Merge	Das Verfahren, bei welchem zwei Listen von Änderungen an einem ursprünglichen Datenobjekt entweder ein geändertes Datenobjekt oder eine zusammengeführte Liste von Änderungen am ursprünglichen Datenobjekt erzeugen, heißt Merge.
diff-match-patch	Der Name diff-match-patch bezeichnet die von Neil Fraser entwickelte Programmbibliothek, die unter anderem für den Abgleich (Diff) von Texten verwendet werden kann. Sie ist für mehrere Programmiersprachen verfügbar und unter https://github.com/google/diff-match-patch veröffentlicht.
elmos	Die elmos ist ein international agierendes Unternehmen der Halbleiterindustrie. Ihre Kontaktdaten sind: Elmos Semiconductor SE Werkstättenstraße 18 51379 Leverkusen Telefon: +49 (0) 2171 / 40 183-0

	E-Mail: info@elmos.com Internet: www.elmos.com
DMOS	Die DMOS ist eine Firma, die ICs entwickelt. Sie hat diese Masterarbeit ausgeschrieben und betreut sie. Sie gehört zur elmos und kooperiert mit ihr. Da sich die Interessen der elmos und der DMOS überschneiden und sie sich einige ihrer Ressourcen teilen, wird in dieser Arbeit vereinfachend nur die DMOS erwähnt, auch wenn die elmos ebenfalls beteiligt ist. Die Kontaktdaten der DMOS sind: DMOS GmbH Bergstraße 4 01069 Dresden Telefon: +49 (0) 351 479 42 – 0 E-Mail: info@dmos2002.de Internet: www.dmos2002.de
STR-Tool	Das STR-Tool ist das von der DMOS entwickelte Programm zum Bearbeiten, Kommentieren und Auswerten von Testdatensätzen.
STR-Server	Der STR-Server wird ein Server sein, der als zentrale Stelle für die Synchronisierung von Projekten des STR-Tools sein. Er soll im praktischen Teil dieser Arbeit aufgesetzt werden.
String	String ist das englische Wort für Zeichenkette.
Map	Map ist im Kontext dieser Arbeit das englische Wort für Zuordnung und bezeichnet die Datenstruktur in Java, in welcher Daten als Schlüssel-Wert-Paare gespeichert werden.
Maven	Maven ist ein Programm zur Projekt- und Buildverwaltung, welches auch Abhängigkeiten auflösen kann und über Plugins konfigurierbar ist. Es wird auch vom STR-Tool verwendet. Weitere Informationen können unter https://maven.apache.org gefunden werden.

3. Abbildungsverzeichnis

4. Einleitung

Java ist aufgrund diverser Vorteile für viele Unternehmen die Programmiersprache für die firmeneigenen und internen Softwareprodukte. Das firmeneigene Java-Programm ist ursprünglich vielleicht nur darauf ausgelegt gewesen, einem Mitarbeiter viele oft zu wiederholende Aufgaben abzunehmen und die vorher verwendeten Excel-Tabellen zu ersetzen. Wenn Betriebe wachsen und ihre diese Software lange nutzen, kommen sie eventuell an einen Punkt, an welchem sie ihre Software weiterentwickeln müssen, um neueren Anforderungen gerecht werden zu können. Was geschieht, wenn die Software so nützlich und so beliebt ist, dass mehrere Mitarbeiter oder Nutzer sie gleichzeitig zur Bearbeitung der selben Daten verwenden wollen? Sie muss für den Mehrnutzerbetrieb umgerüstet werden, was oftmals mit sehr großen Aufwand verbunden ist. Diese Masterarbeit soll dazu beitragen, diesen Aufwand zu verringern.

Der Grund für diese Masterarbeit entspringt der DMOS GmbH in Dresden, die ihr STR-Tool, die interne Software für die Auswertung von Tests integrierter Schaltkreise, verbessern möchte. Das STR-Tool ermöglicht es Testingenieuren, mehrere Projekte zu bearbeiten, jedoch kann dabei aktuell ein Projekt immer nur von einem Nutzer gleichzeitig bearbeitet werden. Die Projektdateien liegen dabei in einem geteilten Ordner im Netzlaufwerk. Wenn ein Projekt mit dem STR-Tool zur Bearbeitung geöffnet wird, dann ist es für andere Nutzer gesperrt, sodass sie nur noch lesenden Zugriff haben. Ein wichtiger Arbeitsschritt der Testingenieure mit dem STR-Tool ist das Kommentieren der Tests. Dies ist insbesondere bei großen Projekten mit mehreren tausend Tests ein Flaschenhals, weil die Ingenieure in der aktuellen Version des STR-Tools aufeinander warten müssen, anstatt gleichzeitig verschiedene Tests kommentieren zu können.

5. Zielstellung

Ziel dieser Masterarbeit ist es, eine Programmbibliothek und Beispielimplementierungen bereitzustellen, mit deren Hilfe Java-Entwickler, eine vormals auf Einzelnutzung ausgelegte Software mit möglichst geringem Aufwand auf eine Client-Server-Architektur umstellen können. Die Idee dabei ist, dass der Server die Synchronisierung der Daten übernimmt und sie in geeigneter Weise in einer Datenbank speichert. Nach der Umstellung soll es mehreren Nutzern möglich sein, gleichzeitig dieselben Daten zu bearbeiten.

In dieser Arbeit soll auch ermittelt werden, ob und in wie weit es nötig ist, Dateien oder Datenbereiche für die Bearbeitung zu sperren, und wodurch sich bei der gleichzeitigen Bearbeitung Wartezeiten für die Bearbeiter ergeben.

Für die DMOS ist das Ziel dieser Arbeit, das STR-Tool mehrbenutzerfähig zu machen. Das STR-Tool ist in Java programmiert, sodass für die Clients eine Lösung in Java benötigt wird. Ebenfalls stellen sich für die DMOS im Zusammenhang mit der Umstellung des STR-Tools auch Fragen von denen in dieser Arbeit möglichst viele geklärt werden sollten. Es folgt eine Auflistung dieser Fragen, welche einer Seite des Intranets der elmos[STR-DB] entnommen und hier für die bessere Verständlichkeit abgewandelt worden ist:

Bereitstellung der Infrastruktur

- Welche Datenbank und welche Sichten (Views) werden benötigt?
- Wie läuft der Authentifizierungsvorgang ab? Wie werden die Rechte vergeben? Kann die Authentifizierung aus einem anderen firmeninternen Bereich wie z.B. der EDF-Datenbank wiederverwendet werden?
- Wie werden neue Versionen des STR-Tools installiert?

Wie bearbeiten mehrere User das gleiche Projekt?

- Wird ein Lock-Mechanismus benötigt? Falls ja, was genau wird dann gesperrt? Einzelne Bereiche des Tools, einzelne Kommentare, etc...?

Benötigen wir eine Import/Export/Sync-Funktionalität?

- Müssen bestehende STR-Projekte in die Datenbank importierbar sein?
- Müssen Datenbank-Projekte als Datei exportiert werden können?
- Wird eine Synchronisierung zwischen Datei- und Datenbank-basierten Projekten benötigt?
- Welche Nutzungsmöglichkeit ist für externe Test-Dienstleister geeignet, die keinen Zugriff auf die Datenbank haben?

Was muss alles an der STR-Tool GUI geändert werden?

- Sollen STR-Projekte aus der Datenbank erstellt oder geöffnet werden können?
- Sollen alle Nutzer alle STR-Projekte sehen können?
- Wie werden gesperrte Bereiche oder Änderungen durch andere Nutzer markiert?
- Wann und wie werden Änderungen gespeichert? Werden die Änderungen erst nach dem Speichern des ganzen Projekts für andere Sichtbar? Kann jeder Kommentar direkt nach der Bearbeitung gespeichert werden, um die Änderungen den anderen Bearbeitern des gleichen Projekts schnell anzuzeigen? Wird ein neuer Speicher-Button pro Kommentar gebraucht?

Die DMOS beschäftigt viele Mitarbeiter, die das STR-Tool nutzen und nicht alle haben das gleiche Nutzungsverhalten. Für einige Nutzer gibt es Fälle, in welchen sie das STR-Tool wie bisher auch offline mit ihren lokal gespeicherten Daten nutzen wollen, beispielsweise wenn sie es unterwegs am Laptop nutzen und keinen Zugriff auf das Firmennetzwerk haben. Nach der Umstellung des STR-Tools auf den Mehrbenutzerbetrieb soll die alte Nutzungsweise immer noch möglich sein.

Außerdem möchte die DMOS gerne bei Änderungen an einem Projekt wissen, welcher Mitarbeiter sie getätigt hat. Bei unerwünschten Änderungen, soll es auch möglich sein, einen alten Stand wiederherzustellen. Dafür ist auch eine Versionsverwaltung nötig.

6. Lösungsweg

6.1 Recherche zu bereits vorhandenen, ähnlichen Lösungen

Zunächst soll geprüft werden, in wie fern es bereits bestehende Programme oder Programmbibliotheken gibt, welche für die Echtzeit-Übertragung zwischen Client und Server, die Synchronisierung oder für die OT genutzt werden können, sodass kein unnötiger Entwicklungsaufwand betrieben werden muss.

6.2 Entwicklung einer Programmbibliothek für die Erleichterung der kollaborativen Echtzeit-Synchronisierung in Java

In diesem Teil soll eine Programmbibliothek erstellt werden, in welcher so viel wiederverwendbare Funktionalitäten zur Echtzeit-Synchronisierung von Clients mit einem Server wie möglich enthalten sein sollen.

6.3 Integration der Programmbibliothek ins STR-Tool

Die im vorherigen Schritt entwickelte Programmbibliothek soll im STR-Tool genutzt werden. Ein neuer STR-Server soll für die Kommunikation mit den Clients aufgesetzt werden. Die Nutzeroberfläche des STR-Tools soll um geeignete Oberflächenelemente erweitert werden, mit welcher der Nutzer die Echtzeit-Synchronisierung mit dem STR-Server konfigurieren, starten und unterbrechen kann. Die Benutzung dieser Oberflächenelemente soll entsprechende Aufrufe an die Programmbibliothek auslösen, sodass mehrere Nutzer das STR-Tool auch tatsächlich zur gleichzeitigen Synchronisierung der von ihnen bearbeiteten Projekte nutzen können. Zudem sollen weitere Oberflächenelemente hinzugefügt oder geändert werden, mit denen Nutzer erkennen können, welche Projekte und welche Stellen darin gerade von anderen Nutzern bearbeitet werden.

7. Hauptteil

7.1 Recherche

7.1.1 Aufgaben für vorgefertigte Software

Bei der Recherche zu bereits bestehender Software für die Echtzeit-Synchronisierung sind viele Programmbibliotheken und Werkzeuge auf die Nützlichkeit für die Interessen der DMOS untersucht worden.

Im Zusammenhang mit der Synchronisierung der Projekte des STR-Tools gibt es einige Aufgaben, die voraussichtlich gut durch bereits existierende Software gelöst werden kann. Diese Aufgaben müssen zumindest clientseitig in der Programmiersprache Java bewältigt werden, weil das STR-Tool ebenfalls in Java programmiert ist. Für den Server bietet sich wegen der Wiederverwendung von Code ebenfalls Java als Programmiersprache an, jedoch sind auch andere Programmiersprachen für den Server denkbar. Die sich für diese Arbeit mit Hilfe bestehender Programmbibliotheken und Werkzeuge zu bewältigenden Aufgaben sind die folgenden:

1. Echtzeit-Datenübertragung zwischen Client und Server.
2. Erfassung von Änderungen (Diff) an einem Datenobjekt
3. Anwendung von Änderungen (Patch) für ein Datenobjekt
4. Zusammenführen unterschiedlicher Änderungen (Merge)
5. Versionierung, inklusive des ändernden Nutzers
6. Persistenz durch eine Datenbank

7.1.2 Suchwortvorschläge

Während der Recherche haben sich bestimmte Wortgruppen ergeben, welche sich für die Suche nach Werkzeugen, die die Synchronisierung zwischen Client und Server erleichtern können, anbieten. Diese sind:

- real-time synchronization
- real-time collaborative editing
- Conflict-free Replicated Data Types
- operational transformation
- Differential Synchronisation

Im englischen Wikipedia-Artikel zum Thema OT gibt es auch eine tabellarische Übersicht über Algorithmen zur Integration/Kontrolle von OT[OT Wiki].

7.1.3 Rechercheergebnis

In der folgenden Tabelle wird Software aufgeführt, welche bei der Recherche in Betracht gezogen worden ist. Die Spalte Möglichkeiten / Bemerkungen enthält eine kurze Beschreibung der Software, in wie fern sie für die Anforderungen der DMOS für die Entwicklung des STR-Tools nützlich ist.

Name	URLs	Möglichkeiten / Bemerkungen	Programmier-sprache / Technologie
FreeFileSync	https://freefilesync.org/	Keine Bauanleitung und es lässt sich nicht ohne Schwierigkeiten bauen	C++
SignalR	https://learn.microsoft.com/de-de/aspnet/core/signalr/introduction?view=aspnetcore-8.0	Bietet nur Echtzeit-Übertragung, aber keine Synchronisierung und keine OT.	ASP.NET
SyncedStore	https://syncedstore.org/docs/	Ermöglicht CRDT und bietet die Integration mehrere weiterer Software-Technologien	node.js, Yjs, optional: Vue, React, Svelte, Matrix-CRDT
realtime-sync	https://github.com/benotter/realtime-sync/	Keine Beschreibung oder Anleitung. 6 Jahre lang keine Änderung mehr.	TypeScript, node.js
ChronoSync	https://github.com/named-data/ChronoSync	Veraltet. Empfiehlt Psync oder StateVectorSync.	ndn-cxx
Psync	https://github.com/named-data/PSync	Implementiert das Psync Protokoll. Es erscheint zu umfangreich, um schnell evaluiert zu werden und zu wenige Vorteile gegenüber anderen hier aufgeführten Technologien zu bieten.	ndn-cxx
rsync	https://fracpete.github.io/rsync4j/	Bietet nur eine einfache Synchronisation von Dateien. Es gibt keine automatische Erkennung von Änderungen und somit auch keine Echtzeit-Übertragung	Verfügbar in Java, oder als Programm zur Ausführung über die Kommandozeile
SubEthaEdit	https://subethaedit.net/	Aufgrund der Ansprüchen an das Betriebssystems wurde SubEthaEdit nicht genauer untersucht.	XCode (MAC)

CodeMirror	https://codemirror.net/examples/collab/	Auf kollaborative Zusammenarbeit (CRTE) beim Bearbeiten von Quellcodes ausgelegt.	node.js
SwellRT	https://swellrt.org/	Seit 6 Jahren kein Update.	Erzeugt mit gradle aus Java-Dateien JS-Dateien
Apache Kafka Streams	https://kafka.apache.org/	Nur für Client-Server-Kommunikation; auf Stream-Handling ausgerichtet.	Java
Apache Storm	https://storm.apache.org/	Auf Server-Cluster ausgelegt und für diese Arbeit wegen unnötiger Komplexität ungeeignet.	Jede Programmier-sprache (https://storm.apache.org/about/multi-language.html)
Esper	https://www.espertech.com/	Esper ist auf CEP ausgelegt und für diese Arbeit wegen lukrativerer anderer Technologien nicht genauer untersucht worden.	Java oder .NET
Fluid	https://fluidframework.com/	Fluid ist auf JS ausgelegt und ist laut dem FAQ unter https://fluidframework.com/docs/faq#can-i-use-aspnet-aspnet-core-and-c unpraktisch auf andere Programmiersprachen zu portieren	node.js / JS / TypeScript
5day.io	https://5day.io/	Bietet nur fertige Business-Lösungen.	unbekannt
Confluence	https://www.atlassian.com/de/software/confluence	Source Code ist nur für Lizenzinhaber und darf nicht veröffentlicht werden.	Java (, HTML)
Kolab	https://kolab.org/	Bietet diverse Werkzeuge zur Zusammenarbeit, ist jedoch nicht für diese Arbeit geeignet.	C++
MixedInk		Man findet nur noch Verweise auf MixedInk, aber keine offizielle Webseite.	unbekannt
Nextcloud	https://	Bietet diverse Werkzeuge	C++

	nextcloud.com/de/ und https://github.com/nextcloud	zur Synchronisation. Für diese Arbeit wäre nur die Dateisynchronisation relevant.	
PlayCanvas Editor	https://github.com/playcanvas/editor	Auf das Editieren von WebGL/WebGPU/WebXR Anwendungen ausgelegt.	JS
zulip	https://zulip.com/	Auf Teamchat ausgelegt, augenscheinlich ohne für diese Arbeit nützliche Funktionen und zu umfangreich für eine genauere Evaluation.	Python, nginx
etherpad	https://etherpad.org/	Bietet RTCE.	node.js, auch für Docker verfügbar
diff-match-patch	https://github.com/google/diff-match-patch	Bietet nur Diff und Patch.	in verschiedenen Programmiersprachen verfügbar
mobwrite	https://code.google.com/archive/p/google-mobwrite/	Bietet RTCE, mit Differential Synchronization und nutzt diff-match-patch. Seit 14 Jahren kein Update.	Java oder JS. Alternativ können einige Tools auch per Python-Script ausgeführt werden.
Operational Transformation (slevental)	https://github.com/slevental/operational-transformation	Nur OT. Seit 7 Jahren kein Update.	Java
otter-java und otter-js	https://github.com/LevelFourAB/otter-java und https://github.com/LevelFourAB/otter-js	Bietet RTCE. Für Java gibt es zwar ein Interface, aber keine Implementierung für die Synchronisation zwischen Client und Server. Seit 3 Jahren kein Update.	Java oder JS

Die Recherche hat ergeben, dass es für JS bereits gute Werkzeuge gibt, welche in dieser Arbeit für die Server-Implentierung gut nutzbar wären. Namentlich ist dies SyncedStore, wobei eventuell auch etherpad in Frage käme.

Die Entscheidung ist zugunsten von otter-java getroffen worden. Die Gründe dafür sind, dass es in Java programmiert ist, sodass Clients und der Server eine große gemeinsame Codebasis nutzen können, dass es gut als Programmbibliothek eingebunden werden kann und dass es leicht erweitert werden kann.

7.2 Beschreibung von otter-java

Die Programmbibliothek otter-java ist eine Implementierung der OT und, wie ihr Name bereits sagt, in Java programmiert. Da sie eine wichtige Funktion für diese Masterarbeit hat, wird sie hier genauer erläutert.

Die wichtigsten Bestandteile von otter-java sind die Operationen, die Engine und das Modell. Jeder dieser Bestandteile hat ein eigenes Unterprojekt in otter-java und wird in den folgenden Unterkapiteln dieser Masterarbeit beschrieben.

7.2.1 Operationen

Operationen bilden die Basis vieler Funktionen von otter-java. Sie werden an zu vielen Stellen referenziert, als dass deren Aufzählung zweckhaft wäre. Zudem beinhalten sie so viele Hilfsklassen, dass sich die Beschreibung in dieser Arbeit auf die wichtigsten Klassen und Schnittstellen beschränkt. Das folgende Klassendiagramm stellt diese dar, wobei die Beziehungen der Klassen nach außen absichtlich weggelassen worden sind, um die Übersichtlichkeit zu wahren.

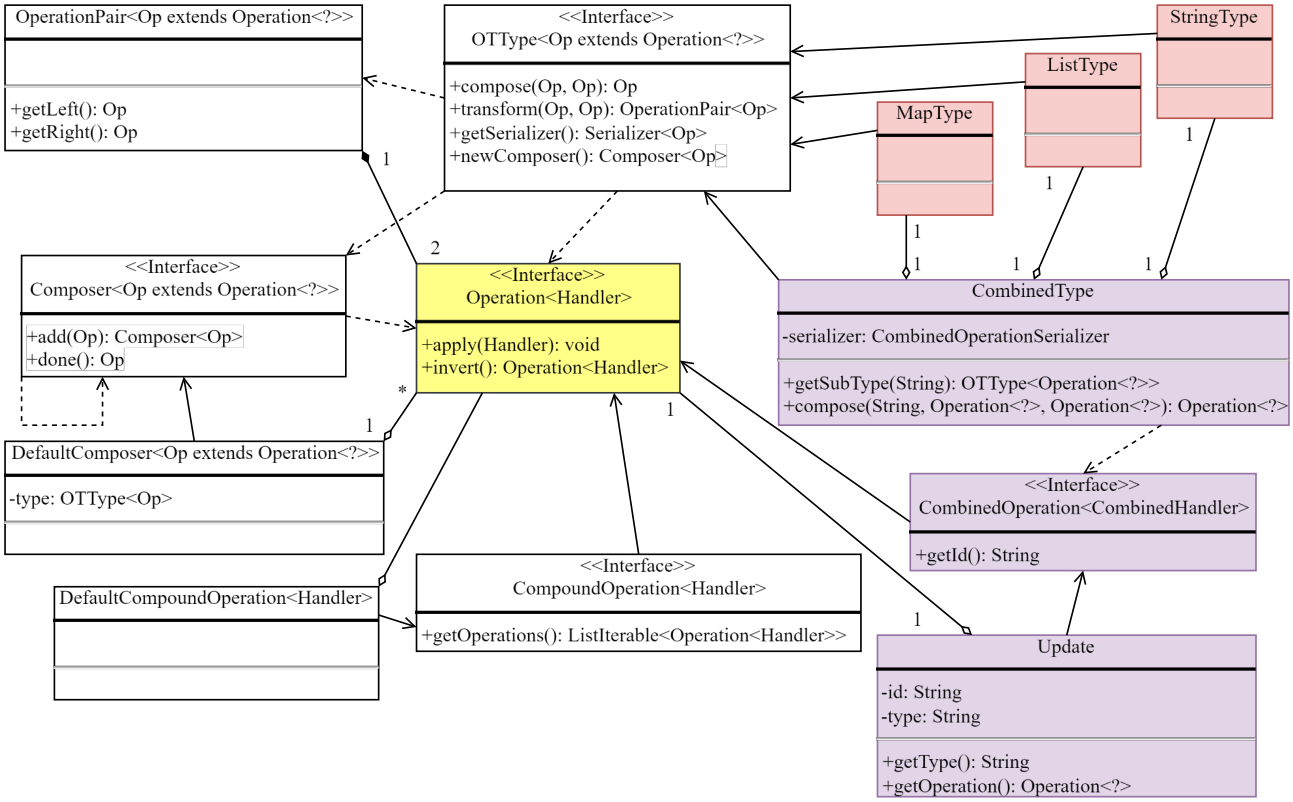

Abbildung 1: kurzes Klassendiagramm zu Operationen (gelb) mit Grundoperationstypen (rot) und zusammenfassenden Typen (lila)

7.2.1.1 Grundoperationen

Otter-java hat Implementierungen für die OT und RTCE von Strings, Listen und Maps. Zu jedem dieser Typen gibt es einen **OTType**. Otter-java unterstützt auch die Verwendung eigener Operationen. Die bereitgestellten Implementierungen nutzen jedoch nur die drei obigen Grundtypen. Die Anwendung der OT bewirkt das Änderungen an geteilten Objekten in einer platzsparenden Weise erfasst, gespeichert und serialisiert werden können.

SharedObject-Typ	Implementierung	Intern genutzter Typ	Operation	OTType
SharedString	SharedStringImpl	StringBuilder	Operation<StringHandler>	StringType
SharedList<T>	SharedListImpl<T>	List<T>	Operation<ListHandler>	ListType
SharedMap	SharedMapImpl	Map<String, Object>	Operation<MapHandler>	MapType

Tabelle TAB: Grundtypen von Operationen in otter-java

Modifikationen an einem SharedString sind durch folgende Methoden möglich: **set** zum überschreiben des alten Wertes, **append** und **insert** zum An- und Einfügen eines Strings und **remove** um Textteile in einem gegebenen Bereich herauszulöschen. Mit **get** wird der aktuelle String zurückgegeben.

Eine SharedList bietet die folgenden Methoden für Modifikationen: **clear** zum Leeren; **set** zum Ersetzen des Elementes am gegebenen Index, **add**, **addAll**, **insert** und **insertAll** zum An- und Einfügen von Elementen; **remove** und **removeRange** zum Entfernen eines oder mehrerer Elemente in einem gegebenen Bereich. Mit **length** kann die Länge der Liste ermittelt und mit **get** das Element am angegebenen Index zurückgegeben werden.

Bei einer SharedMap sind Modifikationen nur durch die Methoden **remove** zum Löschen eines Eintrages und **set** zum Setzen eines Eintrages. Mit **containsKey** kann überprüft werden, ob ein bestimmter Wert enthalten ist und mit **get** kann der Wert zurückgegeben werden. Die SharedMap hat im Vergleich zu den anderen beiden Grundtypen für geteilte Objekte zwei Besonderheiten. Zum einen bieten weder ihr Interface, noch ihre Implementierung eine Möglichkeit, von außen eine vollständige Liste ihrer Einträge oder ihrer Schlüssel zu bekommen. Zum Anderen definiert sie das Interface Listener, mit welchem man sich bei der SharedMap für Änderungsereignisse registrieren kann.

Die Werte, welche in einer SharedList oder einer SharedMap enthalten sind, werden von der Klasse **DataValues** als ein eigenes Datenobjekt behandelt. Beim Einfügen in und ggf. auch beim Löschen eines Objektes aus einer SharedList oder SharedMap wird dieses über die öffentliche statische Methode toData in ein solches Datenobjekt überführt. Diese Datenobjekte werden intern als Liste verwaltet. Wenn der an toData übergebene Parameter ein SharedObject ist, wird eine Referenz als Datenobjekt zurückgegeben, die intern als Liste mit den drei Einträgen „ref“, der ID und dem Typ verwaltet wird. Bei allen anderen Eingaben wird ein Wert als Datenobjekt zurückgegeben, der intern als Liste mit den zwei Einträgen „value“ und dem Eingabeobjekt verwaltet wird. Die Umkehroperation wird durch die ebenfalls öffentliche und statische Methode fromData bewirkt. Sie sucht bei Referenzen im Model nach dem geteilten Objekt mit der entsprechenden ID und dem Typ und gibt den Fund zurück. Bei Werten gibt sie einfach den zweiten Listeneinträg zurück.

7.2.1.2 CompoundOperation

Alle Operationen der Grundtypen für geteilte Objekte werden für gewöhnlich als Listen gruppiert übergeben und behandelt. Es gibt eine eigens dafür geschaffene Schnittstelle namens **CompoundOperation**. Bei Änderungen an geteilten Strings und geteilten Listen erfüllt diese Gruppierung den Zweck, mittels Einfüge-, Beibehalte-, und Löschoperationen die eindeutigen Position von Einfüge- und Löschoperation angeben zu können, wobei die Einhaltung der Reihenfolge wichtig ist. Die untere Tabelle verdeutlicht dies anhand eines Beispiels, bei dem zuerst der neue String „Hallo Welt“ angelegt wird, dem dann ein Ausrufezeichen angefügt wird und bei dem zum Schluss die drei Buchstaben „elt“ durch „ald“ ersetzt werden. Für jeden der drei Vorgänge

gibt es eine CompoundOperation. In der Serialisierung wird die Liste der CompoundOperation in eckige Klammern überführt.

String vorher	**String nachher**	**Elemente der CompoundOperation**	**serialisierte CompoundOperation**
	Hallo Welt	StringInsert(„Hallo Welt“)	[+Hallo Welt]
Hallo Welt	Hallo Welt!	StringRetain(10), StringInsert(„!“)	[_10, +!]
Hallo Welt!	Hallo Wald!	StringRetain(7), StringDelete(„elt“), StringInsert(„ald“), StringRetain(1)	[_7, -elt, +ald, _1]

Tabelle TAB: Beispiel für CompoundOperation anhand eines SharedString

7.2.1.3 OTType

Der OTType ist eine Klasse, die einem bestimmten Typ eines geteilten Objektes entspricht und welche , der die Methoden **transform** und **compose** enthält

Die Methode transform des OTType wird ausgeführt, um aus zwei konkurrierende Änderungslisten für einen gemeinsamen Ausgangsstand eines geteilten Objektes ein Paar von Änderungslisten zu ermitteln. Dies gleicht einem Merge, da jede der beiden neuen Listen an den ihnen entsprechenden Verursacher der Änderung übergeben werden kann, womit dieser durch Anwendung der Methode compose den gemeinsamen neuen Stand bekommt. Das gelingt nur, wenn Einigkeit über den Ausgangsstand herrscht und wirft andernfalls eine **TransformException**.

Die Methode compose des OTType wird verwendet, um eine Liste von Änderungen mit der Liste der Operationen, die den Ausgangsstand formen, zu vereinigen, um daraus einen neuen Stand zu ermitteln. Dies wird bei jeder lokalen Änderung an einem geteilten Objekt ausgeführt. Wenn compose bei geteilten Listen oder Strings genutzt wird, und die Gesamtlänge des durch die Operationen beschriebenen Inhaltes des Ausgangsstandes nicht mit der der Änderung übereinstimmt, wird eine **ComposeException** geworfen.

7.2.1.4 CombinedOperation

Die höheren Verarbeitungsebenen fassen die Grundtypen von Operationen in der Schnittstelle namens **CombinedOperation** zusammen, dessen einzige Implementierung die Klasse **Update** ist. Jedes Update erfasst die ID und den Typen des geteilten Objektes und die ausgeführte Operation. Der entsprechende OTType ist die Klasse **CombinedType**. Die Zusammenfassung der Grundtypen erleichtert die weitere Verarbeitung, da dieser eine gemeinsame Typ von geteilten Objekten anstatt der anderen drei verwendet werden kann. Die CombinedOperation leitet die Methodenaufrufe compose und transform an den passenden OTType weiter.

7.2.2 Engine

Die Engine wird bereits kurz im Repositorium von otter-java auf github[otter-java] beschrieben. Für diese Arbeit ist es nötig gewesen, sie detaillierter zu beleuchten, um geeignete Erweiterungen zu erstellen. Die Engine ist eine Gruppe von Schnittstellen, für die otter-java je eine Implementierung bietet.

Erwähnenswert ist, dass diese Implementierungen alle Daten im Arbeitsspeicher halten und dass es in otter-java weder eine Implementierung, noch einen Testfall gibt, mit denen das Synchronisieren über das Netzwerk demonstriert wird.

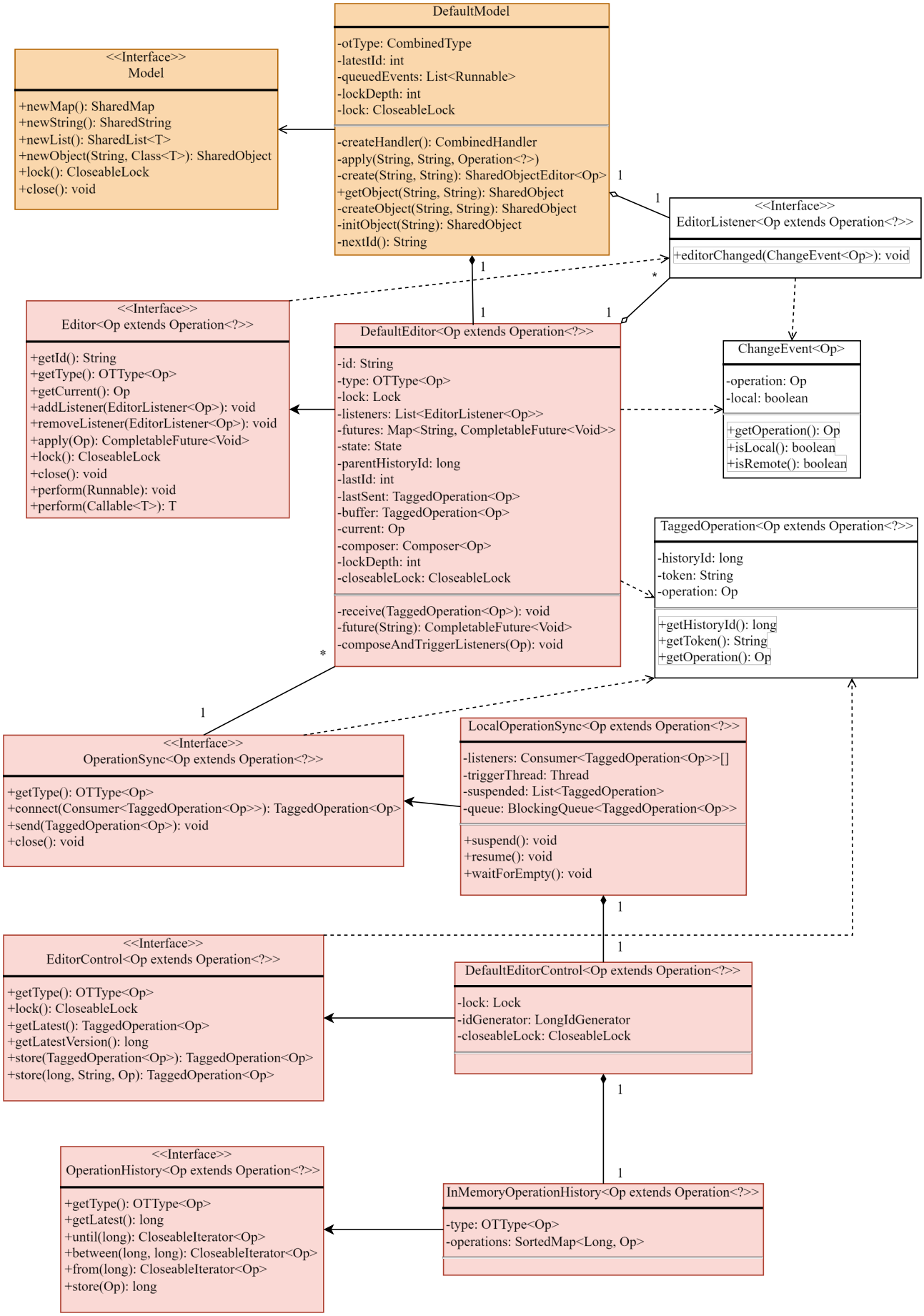

Abbildung 2: Klassendiagramm der Engine mit ihren Haupttypen (rosa) und deren Verbindung zum Modell (hellbraun)

7.2.2.1 TaggedOperation

Die Klasse **TaggedOperation** hat drei Datenfelder historyId, token und operation. Mit ihnen erfasst sie einen Stand oder eine Änderung eines Bearbeiters. Die historyId ist eine fortlaufende Nummer, die der Anzahl der Änderungen am geteilten Objekt entspricht, wobei das Erzeugen des geteilten Objektes ohne Inhalt mitzählt. Sie ist das Äquivalent zu einer Revisionsnummer. Das Token ist eine Zusammensetzung aus einem zufällig generiertem String, welcher den Editor identifiziert, und einer fortlaufenden Nummer. Mit der historyId und dem Token wird sozusagen ein Tag für die Operation gebildet, anhand der sie eindeutig im Netzwerk identifizierbar gemacht wird. TaggedOperation ist die Klasse, die von mehreren Klassen und Schnittstellen der Engine zum Synchronisieren verwendet wird.

7.2.2.2 OperationHistory

Die **OperationHistory** dient dazu, eine Historie aller Operationen zu pflegen. Die Methode **store** speichert eine gegebene Operation. Hierzu sei angemerkt, dass otter-java die Klasse **InMemoryOperationHistory** für die Implementierung der OperationHistory bereitstellt. Wie der Name der Klasse bereits sagt, hält sie die Operationen nur im Arbeitsspeicher und nutzt keine Form von Persistenz. Sie fügt die zu speichernden Operationen in eine sortierte Map ein, von wo sie später durch andere Operationen abgerufen werden kann.

Der Abruf ist durch mehrere Methoden möglich. Um die Nummer des neuesten Eintrages zu bekommen, verwendet man **getLatest.** Mit den Methoden **from**, **until** und **between** lassen sich die Operationen ab einer, bis zu einer oder im Intervall zwischen zwei Nummern zurückgeben. Für die Rückgabe der Operationen wird ein **CloseableIterator** verwendet. Dabei handelt es sich um einen Iterator, der auch die Schnittstelle AutoCloseable unterstützt.

In den in otter-java enthaltenen Implementierungen wird die OperationHistory nur von der **DefaultEditorControl** verwendet.

7.2.2.3 EditorControl

Die **EditorControl** ist eine Schnittstelle, die für das Einfügen von neuen und für das Abrufen von bisherigen Operationen verwendet wird, wofür sie die Methoden **store** und **getLatest** bietet. Die Nummer des neuesten Standes aller Operationen kann mit **getLatestVersion** abgerufen werden. Um den lokalen Stand aller Operationen konsistent zu halten, bietet sie die Methode **lock**, die ein CloseableLock zurückgibt, mit dem alle Schreibenden Zugriffe, sowie längere Abrufe umschlossen werden sollten. Die Klasse DefaultEditorControl ist die zugehörige Implementierung, welche die einfügenden und lesenden Aufrufe an die OperationHistory weiterleitet. Die EditorControl wird nur vom **LocalOperationSync** verwendet und bildet somit das Bindeglied zwischen ihr und der OperationHistory.

7.2.2.4 OperationSync

Die Schnittstelle **OperationSync** dient dazu, alle Operationen aller am Austausch der Operationen Beteiligten zu synchronisieren und bildet somit eine zentrale Verbindungsstelle für den Informationsaustausch über das Netzwerk. In otter-java gibt es dafür allerdings nur die Implementierung namens LocalOperationSync, welche genau diesen Zweck nicht erfüllt und deshalb nur zur Demonstration der OT auf einem einzigen System geeignet ist. Der Programmierer von otter-java gibt absichtlich keine Details darüber an, wie eine eigene Implementierung der OperationSync über das Netzwerk geschaffen werden könnte. Im Repositorium von otter-java wird über die Funktion der OperationSync nur sehr wenig ausgesagt:

„Editors require a synchronization helper for sending and receiving operations from a server. There is intentionally no default implementation of such a sync as different applications will have different requirements here.“[otter-java].

Daraus folgt, dass für die Schaffung eigener Anwendungen, welche otter-java nutzen wollen an dieser Stelle viel eigene Arbeit nötig ist.

Die wesentlichen Methoden des OperationSync sind **connect**, welche der Aufrufer nutzen kann, um den aktuellen Stand abzurufen und über alle Änderungen mit einer TaggedOperation benachrichtigt zu werden, sowie **send**, um eine TaggedOperation zu senden. Der LocalOperationSync kann als Mock-Objekt zur Simulation der Synchronisierung mit anderen Netwerkteilnehmern aufgefasst werden. Das Senden und Empfangen geschieht über einen eigenen Thread. Zur Unterbrechung und zur Wiederaufnahme des Sendens können die Methoden **suspend** und **resume** genutzt werden. Mit **waitForEmpty** kann der Aufrufer synchron auf die Abarbeitung aller zu sendenden TaggedOperations warten**.**

7.2.2.5 Editor

Der **Editor** ist die Schnittstelle, die für das Erfassen und ggf. für das Zusammenführen von Operationen verwendet wird und aus all diesen Operationen einen lokalen konsistenten Stand ermittelt. Der Editor ist anhand einer ID identifizierbar, die mit **getId** abgerufen werden kann. Operationen können dem Editor mit **apply** übergeben werden, wodurch man ein CompletableFuture bekommt, das als Callback ausgelöst wird, wenn die Operation vom Netzwerk akzeptiert worden ist. Der aktuelle Stand der Operationen lässt sich durch **getCurrent** zurückgeben. Die Methode **lock** gibt ein CloseableLock zurück, das genutzt werden kann, um Änderungen durch den Editor auszusetzen. Zur Bequemlichkeit bei der Nutzung der Methode lock können auch die beiden Überladungen der Methoden **perform** verwendet werden, um die Ausführung einer Aktion mit der Sperre zu umschließen. Mit **addListener** und **removeListener** können Aufrufer sich für Änderungen, die am aktuellen Stand der Operationen auftreten, an- oder abmelden. Der Editor wird sinnvollerweise als Mittler zwischen dem OperationSync und dem Modell verwendet, um Operationen von beiden zu erfassen, zu verarbeiten und an lokale oder entfernte Stellen weiterzugeben.

Die Implementierung DefaultEditor ruft sich im Konstruktor den aktuellen Stand mit dem OperationSync über die EditorControl von der OperationHistory ab und bekommt dadurch von der EditorControl auch eine zufällige ID. Dies ist auch das einzige Mal, dass der DefaultEditor einen ganzen Stand abruft. Danach bietet er nur noch die Möglichkeit, Änderungen am Stand zu erfassen. Intern nutzt er zur Synchronisierung aller lokal erfassten Änderungen ein ReentrantLock, welches auch für das CloseableLock, das über die öffentlich Methode lock zurückgegeben wird, zum Einsatz kommt. Die private Methode **receive** wird aufgerufen, wenn der DefaultEditor eine Operation vom OperationSync bekommt. Der DefaultEditor hat einen internen Zustand **state**, welcher beeinflusst, wie genau sich die mit receive und apply erfassten Operationen auswirken. Das Zusammenspiel von beiden ist so konzipiert, dass lokal mit apply angewendeten Operationen als TaggedOperation an den OperationSync gesendet werden. Für jede gesendete TaggedOperation wird ein CompletableFuture angelegt und eine ihm entsprechende Anwort vom OperationSync erwartet. Bis zur Antwort werden eventuelle weitere Änderungen intern gepuffert und dann, falls der lokale Stand nicht dem des Netzwerks entspricht, gruppiert gesendet. Der DefaultEditor notiert sich die aktuellste vom Netzwerk empfangene Version, um anhand dieses Standes mittels transform und compose die noch ans Netzwerk zu sendenden Operationen zu ermitteln.

Diese müssen nur dem DefaultModel (s. Fehler: Verweis nicht gefunden Fehler: Verweis nicht gefunden) im Konstruktor übergeben werden.

7.2.3 Modell

Das Modell bildet die hochrangige Ebene in der Hierarchie von otter-java. Es liegt im Paket model und wird durch die Schnittstelle **Model** und dessen Implementierung **DefaultModel** beschrieben.

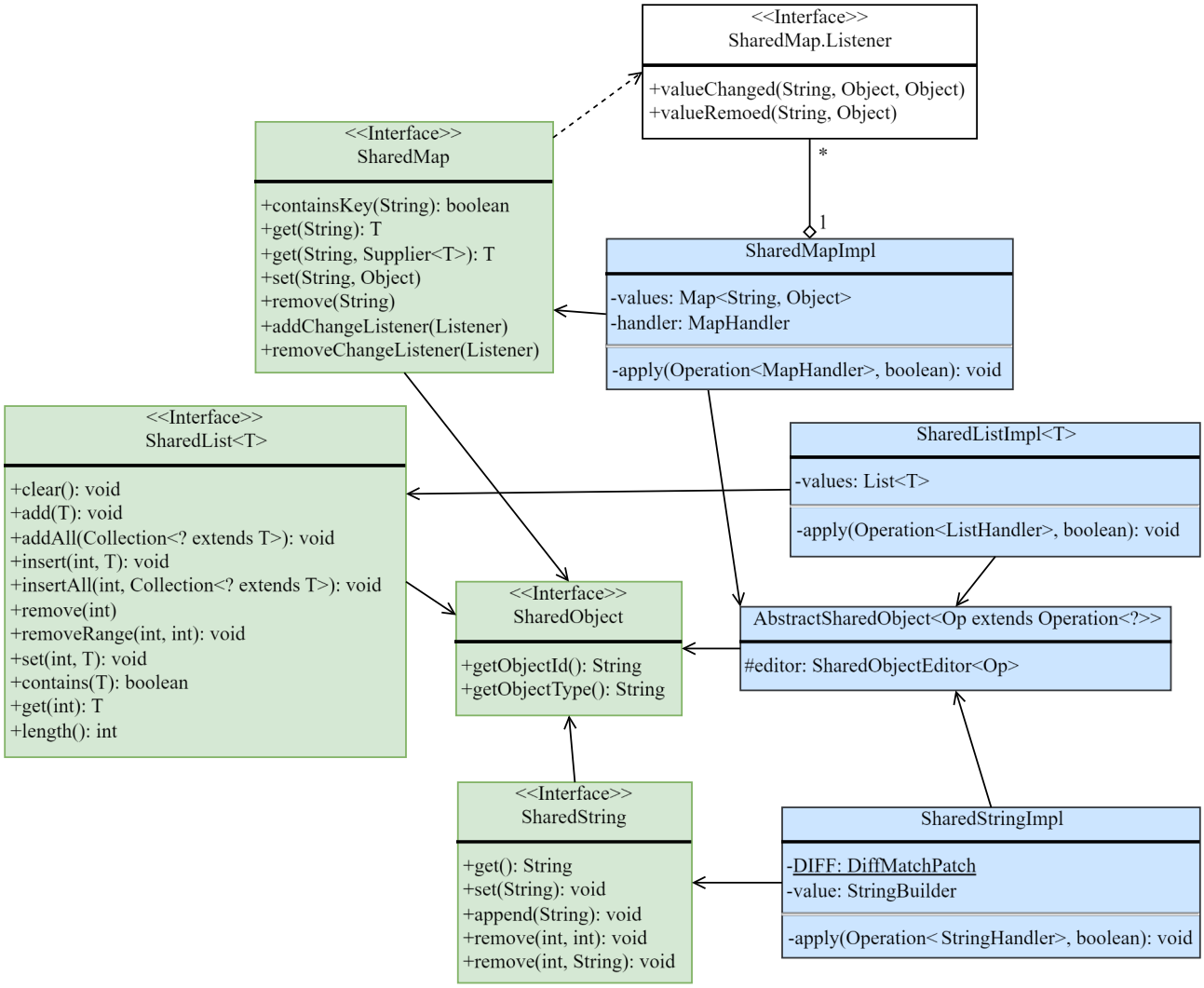

Abbildung 3: Klassendiagramm der Schnittstellen geteilter Objekte (grün) mit ihren Implementierungen (blau)

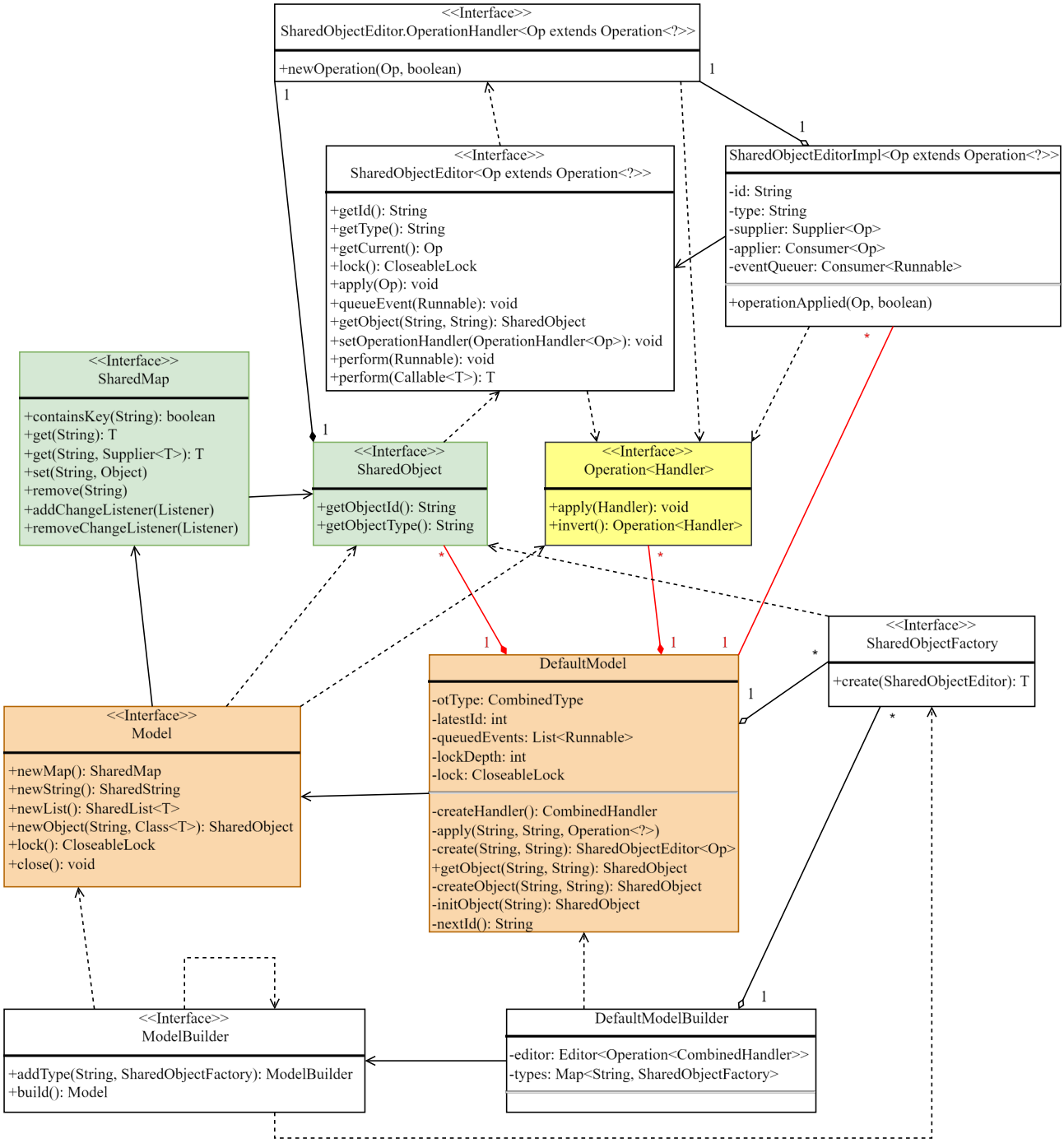

Abbildung 4: kurzes Klassendiagramm des hochrangigen Modells (hellbraun), der geteilten Objekte (grün), der Operationen (gelb) und der Kompositionen des DefaultModel zu einigen seiner verwalteten Daten und Objekten (rot).

7.3 Änderung an otter-java

pom.xml, Bugs

7.4Klärung der Zielstellung

Besprechung vom 10.3.2025

7.5Erweiterung von otter-java durch RTSync

Die Klassen von otter-java bieten alleine noch nicht die nötige Funktionalität, die für die Anforderungen, die sich aus der gewünschten Erweiterung des STR-Tools ergeben. In diesem Kapitel wird aufgezeigt, welche Funktionalität in otter-java dafür noch fehlt und wie sie durch neue Klassen, die im praktischen Teil dieser Masterarbeit erstellt worden sind, erreicht wird. Zur Demonstration der Synchronisation sind außerdem noch Oberflächenelemente geschaffen worden, die voraussichtlich nicht direkt ins STR-Tool übernommen werden. Diese neue Funktionalität wird durch das neu geschaffene Projekt RTSync bereitgestellt, welches sich als Programmbibliothek einbinden lässt.

7.5.1 Struktur und Unterprojekte von RTSync

7.5.2 Erweiterte Operationen

7.5.3 Erweiterte Engine

7.5.3.1 Übersicht (simple und projektbasierte Modelle)

7.5.3.2 OperationHistory

7.5.3.3 EditorControl

7.5.3.4 OperationSync

7.5.3.5 Editor

Bei Experimenten zur eigenen Entwicklung einer OperationSync hat sich ergeben, dass das Vorgehen des DefaultEditor nicht robust gegen unerwartete aus dem Netzwerk empfangene Nachrichten ist. Sollte innerhalb des Netzwerkes ein inkonsistenter Zustand entstehen, bei dem z.B. zwei Netzwerkteilnehmer unterschiedliche Operationen unter der selben historyId gesehen oder selbst erzeugt haben, löst dies eine TransformException aus. Versucht ein Netzwerkteilnehmer seinen inkonsistenten Zustand dem eines anderen Netzwerkteilnehmers anzupassen, ist dies mit dieser Implementierung des Editors auch nicht problemlos möglich, da dieser höchstens anhand seines als falsch angenommen Standes eine Änderung zum Stand des anderen Teilnehmers ermitteln und dann für sich umsetzen könnte. Der Versuch, einen neuen ganzen Stand an einen bestehenden DefaultEditor zu übergeben führt zu einer ComposeException, weil die Methode compose erwartet, dass der alte Stand zumindest mit der korrekten Länge seines Inhalts übergeben wird.

7.5.3.6 Model

7.5.4 Neue Oberflächenelemente

7.5.4.1 RTJTextPane

7.5.4.2 RTProjectList

7.5.4.3 RTProjectViewPanel

Das RTProjectViewPanel besteht aus zwei Komponenten, dem RTProjectViewPanel und RTModelActionPanel.

7.6 Integration von RTSync ins STR-Tool

Für die Integration von RTSync ins STR-Tools sind viele Veränderung nötig, welche nicht mehr im praktischen Teil dieser Masterarbeit umgesetzt werden konnten und deshalb als Entwürfe aufgezeigt werden. Einerseits ist das Integrieren von RTSync in eine bestehende Anwendung eine individuelle Angelegenheit und nicht jeder Leser wird das STR-Tool kennen, aber andererseits können die Prinzipien, welche in diesem Kapitel anhand des STR-Tools erläutert werden, auch für andere Programme gelten und die Planung bei ihrer Verwendung von RTSync vereinfachen.

7.6.1 Nutzungsschemata und Meilensteine

Dieses Unterkapitel soll die bisherige, sowie auch die zukünftige Struktur des STR-Tools beschreiben und dies anhand von schematischen Grafiken verdeutlichen.

Der Umbau des STR-Tools lässt sich in zwei Etappen unterteilen, deren Ergebnisse als Meilensteine bezeichnet werden. Schon beim ersten Meilenstein kann das STR-Tool für die Bearbeitung eines Projektes im Mehrbenutzer-Modus mit STR-Serveranbindung und Echtzeit-Updates verwendet werden. Der wesentliche Unterschied der beiden Meilensteine ist, dass der Server erst beim zweiten Daten persistent hält, wohingegen er im ersten nur als zentraler Vermittler für die Clients dient.

7.6.1.1 Ausgangslage (bisheriges Nutzungsschema):

Das STR-Tool verwendet mehrere Arten von Dateien für seine Arbeit, welche für gewöhnlich zusammen in einem Projektordner liegen. Die genauen Dateipfade können in der grafischen Benutzeroberfläche angegeben und geändert werden.

Die zentrale Datei ist die Projektdatei selbst, in welcher alle Daten des Projektes, so wie die Pfade zu den weiteren Dateien, die das Projekt verwendet, im XML-Format gespeichert werden. Die Serialisierung beim Speichern und die De-Serialisierung beim Laden geschehen über einen Xstream aus dem Paket com.thoughtworks.xstream. XStream nutzt dazu Reflections, sodass alle Datenfelder der Java-Objekte automatisch anhand ihres Namens entsprechenden XML-Elementen zugeordnet werden, ohne eine manuell erstellte ODM zu benötigen. Es gibt drei Arten von STR-Projekten, die durch eine bestimmte Dateiendung (.str, .dtr, .dpr) gekennzeichnet werden und eigene Datenstrukturen enthalten. Im Kontext dieser Masterarbeit sind ihre Unterschiede jedoch von zu geringer Bedeutung für eine detailliertere Beschreibung, da sie für das Thema dieser Arbeit nach den gleichen Prinzipien behandelt werden können.

Das STR-Tool liest diverse Dateien ein, welche die Tests eines Projektes beschreiben. Die Dateien mit der Endung .spt sind Textdateien, die in einer Tabellenstruktur für jeden Test Namen, Beschreibungen und Testgrenzen enthält. Die dadurch beschriebene Tabelle heißt SpecTable. Die tatsächliche Durchführung eines Tests erzeugt Testdaten, welche in verschiedenen Formaten vorliegen und durch ein externes Programm, das sogenannte stdftool, vorverarbeitet werden müssen.

Die EDF-Datenbank enthält weitere Daten zur Beschreibung der Projekte. Im STR-Tool werden diese Daten verwendet, um die tatsächlich verwendeten Testspezifikationen mit den offiziell beschriebenen abzugleichen.

Aus all den gesammelten Daten können umfangreiche Testberichte im .pdf- oder im .odt-Format exportiert werden.

Ein Projekt kann zwar gleichzeitig von mehreren Benutzern geöffnet werden, jedoch hat immer nur ein Nutzer schreibenden Zugriff und alle Oberflächenelemente des STR-Tools, die Änderungen am Projekt ermöglichen würden, sind während dessen für alle anderen Nutzer gesperrt.

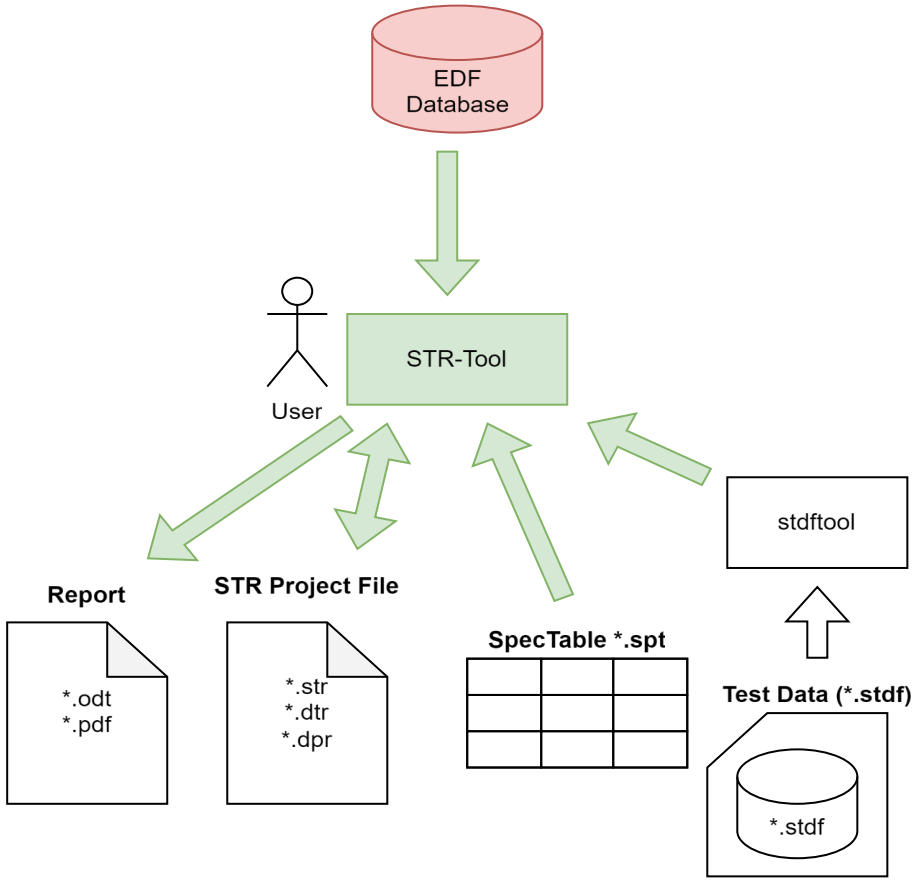

Abbildung 5: bisheriges IO-Schema des STR-Tools, verändert nach Michael Küttner (2023)

7.6.1.2 Meilenstein 1 - STR-Server ohne Datenbank

Bei Meilenstein 1 wird zwischen Online- und Offline-Projekten unterschieden. Die bisherigen Projekte des STR-Tools sind Offline-Projekte und können weiter wie bisher mit dem STR-Tool bearbeitet werden. Zudem wird auch der Import von Offline-Projekten zu Online-Projekten, sowie der Export von Online-Projekten zu Offline-Projekten ermöglicht. Der Nutzer hat zudem durch geeignete Oberflächenelemente die Möglichkeit, die Verbindung mit dem STR-Server zu konfigurieren. Online-Projekte werden automatisch in Echtzeit mit dem STR-Server und in Folge auch mit allen anderen Nutzern, die das gleiche Online-Projekt bearbeiten, synchronisiert.

Der neue STR-Server verwaltet drei Arten von Daten, welche bei Änderungen automatisch in Echtzeit an alle interessierten Clients weitergeleitet werden: Operationen an Echtzeit-Datenobjekten, Positionen von Einfügemarken und die Nutzerliste. Die Operationen bilden dabei die Echtzeit-Datenmodelle der Projekte ab. Sie können entweder als ganzer Stand oder als Änderung zum vorhergehenden Stand übermittelt werden. Der STR-Server hat im Wesentlichen die

gleiche Funktionalität wie der projektbasierte Server von RTSync. Er benötigt keine Kenntnis über das STR-Tool und könnte theoretisch auch für jede andere Art von Projekten verwendet werden. Es gibt serverseitig weder eine Datenbank, noch werden Dateien geschrieben oder gelesen. Alle Daten liegen lediglich im Arbeitsspeicher, sodass bei einem Neustart des Servers der erste bearbeitende Client des Projektes seinen Stand erneut dem Server übermitteln muss. Die in den Projekten benötigten Dateien werden im Echtzeit-Datenmodell nur als Pfade zu den Dateien behandelt.

Die Nutzer des STR-Tools müssen deshalb sicherstellen, dass sie wie bisher auf alle nötigen Dateien zugreifen können. Am vorteilhaftesten wäre es, alle benötigten Dateien zu kopieren und dann die Kopie zu bearbeiten. Für den Dateiaustausch kann eventuell das lokale Netzwerk genutzt werden. Bei Schwierigkeiten mit dem Dateizugriff ist eine individuelle Absprache mit den anderen Bearbeitern des Projektes nötig.

Wenn man von dem manuellen Aufwand des Dateiaustausches absieht, ist die Arbeit an Online-Projekten aus Sicht des Nutzers simpel. Das geplante Vorgehen besteht darin, das STR-Tool zu starten, sich mit dem STR-Server zu verbinden, das Online-Projekt von der Festplatte zu laden, um es dann kollaborativ in Echtzeit zu bearbeiten. Dazu gibt es an mehreren Stellen Oberflächenelemente, die Echtzeit-Änderungen senden und empfangen. Das Speichern des Projektes, der Abruf von Daten aus der EDF-Datenbank und das Exportieren von Berichten geschehen wie bisher auch. Es müssen keine Dateien gesperrt werden. Lediglich das Empfangen von Echtzeit-Änderungen muss bei langwierigen Aktionen, die atomisch auf einen konsistenten Projektstand ausgeführt werden sollen, pausiert werden. Während dieser Aktionen lässt die Nutzeroberfläche des STR-Tools, wie bisher auch, keine Änderungen zu.

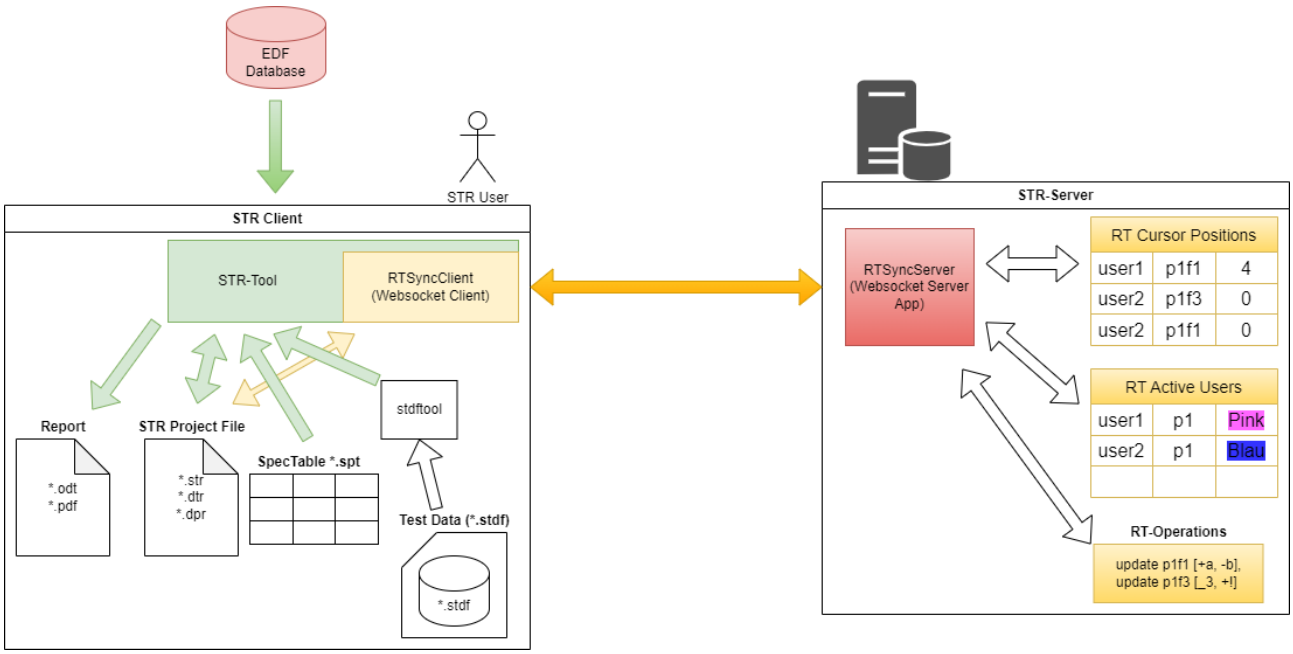

Abbildung 6: IO-Schema des STR-Tools bei Meilenstein 1

7.6.1.3 Meilenstein 2 - STR-Server mit Datenbank

Der Stand bei Meilenstein 2 basiert auf dem von Meilenstein 1 und erweitert diesen. Wie bei Meilenstein 1 auch sind von den hier beschriebenen Änderungen nur Online-Projekte betroffen. Die Stellen, an denen Änderungen nötig sind, involvieren Dateien für die SpecTable oder für die Testdaten. Jede dieser Dateien kann auf zweierlei Weise verwendet werden: entweder wie bisher oder als vom Server verwaltete Datei.

Der Server hat für die dauerhafte Datenhaltung eine Datenbank, in welcher sowohl die von ihm verwalteten Dateien, als auch die Operationen gespeichert werden. Der STR-Server bietet den Clients über neue Funktionen das Hoch- und Herunterladen von Dateien an.

Sofern alle Dateien eines Online-Projektes vom Server verwaltet werden, kann sich ein Nutzer des STR-Tools auch einen Projektstand aller Dateien vom Server herunterladen, wodurch für ihn das Anlegen einer Kopie des bisherigen Projekts von einem anderen Bearbeiter entfällt.

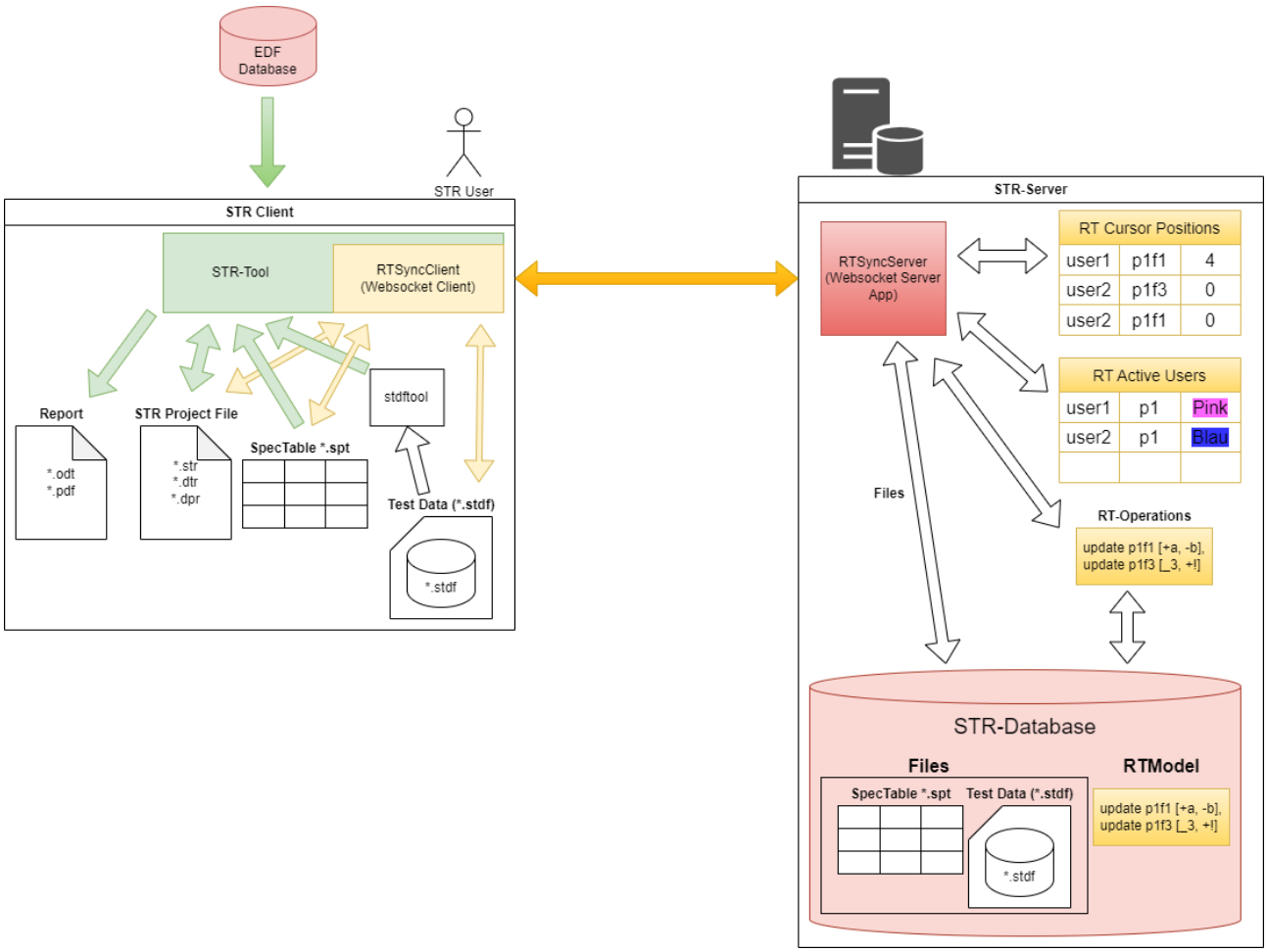

Abbildung 7: IO-Schema des STR-Tools bei Meilenstein 2

7.6.2 STR-Server

7.6.2.1 Meilenstein 1 - Basisfunktionalität

Das STR-Tool ist ein Maven-Projekt mit mehreren Unterprojekten, deren Beziehungen und Abhängigkeiten in Dateien namens pom.xml beschrieben werden. Für die Funktionalität des STR-Servers wird das neue Unterprojekt str-server mit einer eigenen pom.xml-Datei angelegt. Alle in diesem Kapitel genannten Quellcode-Dateien liegen in dem für Maven typischen Ordner src. Im Elternprojekt des STR-Tools wird str-server als neues Kindprojekt hinzugefügt. Das Unterprojekt str-server ist von rtsync-project-server, aber nicht von anderen Unterprojekten des STR-Tools abhängig.

Für die Funktionalität von Meilenstein 1 ist am Server kaum etwas zu tun. Aus dem Project rtsync-project-server müssen von den drei Java-Dateien RTSyncProjectServer, ProjectWebSocketMessageBrokerConfig und RTSyncProjectServerController Kopien für den str-server gemacht und ein wenig geändert werden.

Die Kopie von RTSyncProjectServer und die Klasse darin werden in **StrServer** umbenannt. Darin befindet sich die öffentliche statische Methode **main**, mit welcher der Server gestartet werden kann. Die anderen beiden Dateikopien müssen im selben Paket wie StrServer liegen, um beim Start des Servers automatisch gefunden werden zu können.

Der neue Name der kopierten Konfigurationsdatei und ihrer Klasse können frei gewählt werden. Die Klasse muss nur so geändert werden, dass sie ProjectWebSocketMessageBrokerConfig erweitert, aber sie braucht außer einem leeren Konstruktor, der den Elternkonstruktor aufruft, keinen Inhalt. Dies ist nötig, weil Spring Boot beim Starten einer Anwendung im Normalfall nur das Paket der Anwendung selbst auf Konfigurationen untersucht.

Die Kopie von RTSyncProjectServerController wird zusammen mit dem Klassennamen in **StrServerController** geändert. Die neue Klasse StrServerController ist eine Erweiterung von RTSyncProjectServerController und muss nur dessen Methode onUnsubscribeEvent überschreiben, benötigt aber sonst für Meilenstein 1 keinen Inhalt. Die überschriebene Methode soll zuerst die der Elternklasse aufrufen und anschließend prüfen, ob es für das betroffene Projekt noch Bearbeiter laut der vom Server verwalteten Nutzerliste gibt und wenn nicht, dieses Projekt schließen, um den Arbeitsspeicher einzusparen. Diese Funktionalität des automatischen Schließens könnte auch als optionale Funktion im RTSyncProjectServerController umgesetzt werden.

Der STR-Server läuft genau wie RTSyncProjectServer als Spring Boot Application und unterstützt das Protokoll STOMP über Websockets. Die Clients können sich mit dem Websocket verbinden, indem sie dessen URL mit ws als Protokoll und /socket als Pfad verwenden.

7.6.2.2 Meilenstein 2 - Datenbank

Bei Meilenstein 2 wird der Server eine neue Datenbank bekommen und kann darin bestimmte Dateien eines STR-Projektes speichern. Der Dateiinhalt wird dabei nur gespeichert und abgerufen, jedoch nicht überprüft oder ausgewertet. Dies wird den Clients überlassen. Die STR-Projekte selbst werden nur als Echtzeit-Datenmodell gespeichert, sodass dafür auf dem Server und in der DB weder eine Datei, noch ein StrProject angelegt werden.

Welche Art von Datenbank genau für den STR-Server genutzt werden wird, ist bisher noch nicht geklärt. Es böte sich die Nutzung der Technologie Liquibase an, über welche man sich unter https://docs.liquibase.com/concepts/introduction-to-liquibase.html belesen kann. Der Vorteil von Liquibase besteht darin, sich nicht von vornherein auf ein bestimmtes Datenbanksystem festlegen zu müssen. In den folgenden Absätzen dieses Unterkapitels wird zum Zweck der Vereinfachung davon ausgegangen, dass sowohl die Dateien, als auch die Echtzeit-Datenmodelle in einer relationalen Datenbank gespeichert werden. Die entsprechenden Tabellen heißen Files für die Dateien und RTModel für die Echtzeit-Datenmodelle.

Eine Anforderung an die Datenbank ist, dass Änderungen nachvollzogen werden können sollen. Die Versionierung geschieht, indem in beiden Tabellen Spalten für einen bestimmten Stand eines Eintrages angelegt werden. Diese Spalten enthalten das von der DB automatisch eingetragene Änderungsdatum (changeDate), eine aufsteigende Revisionsnummer (revision) und den Nutzer, der die Änderung verursacht hat (editedBy).

Bei letztem handelt es sich um die Sitzungsnummer, welcher der STR-Server für jeden neuen Client als UUID generiert. Die Zuordnung dieser Sitzungsnummer zu einem Nutzer geschieht über die SimpUserRegistry, die Spring Boot für Websocket-Clients automatisch pflegt. Diese Zuordnung wird nicht dauerhaft gespeichert, sodass derjenige, der eine Änderung verursacht, nur für die Dauer seiner Anmeldung am Server anhand der Sitzungsnummer identifiziert werden kann. Dieses Verhalten ist beabsichtigt und verhindert, dass Mitarbeiter oder ihre Arbeit unnötig überwacht oder ausgewertet werden. Falls doch eine längere Zuordnung gewünscht sein sollte, könnte man das Feld

editedBy auch zum Speichern des Nutzernamens verwenden und den STR-Server den Zugriff auf dieses Feld beschränken lassen, z.B. indem es vor der Weitergabe an den Client verändert wird.

Für die Dateien (**Files**) wäre die Speicherung durch das hier angeführte Schema möglich, wobei keine Spalte null sein darf. In einigen Projekten gibt es Dateien mit einer Größe von 56MB Dateien. Das Ablegen solch großer Dateien in der Datenbank anstatt auf dem Netzlaufwerk ist zwar unerwünscht, jedoch nicht unmöglich. Die genaue Obergrenze für die Dateigröße ist ungewiss, was beim dafür verwendeten Datentyp beachtet werden sollte.

Feld (Spaltenname)	**Typ**	**Bedeutung**
id	INT	automatisch eingetragener Primärschlüssel
revision	SMALLINT	aufsteigende Revisionsnummer für project und path
changeDate	DATETIME	automatisch eingetragenes Änderungsdatum
editedBy	VARCHAR	Sitzungsnummer des ändernden Nutzers
project	VARCHAR	Name des Projektes zu dem die Datei gehört
path	VARCHAR	Dateipfad
content	LONGBLOB	Dateiinhalt

Tabelle TAB: Tabelle Files

Das Tabelle für das Echtzeit-Datenmodell (**RTModel**) enthält zusätzlich zur Versionierung und der gespeicherten Operation (operation) die Spalte wholeState, die die Bedeutung der Operation angibt. Falls dieses Feld auf wahr gesetzt ist, enthält operation den ganzen Stand des Echtzeit-Datenmodells und andernfalls nur die Änderung zum vorherigen Stand. Um eine komplette Historie zu erstellen, muss man alle Zeilen des betrefflichen Projektes abfragen und nach der Revisionsnummer oder dem Änderungsdatum sortieren. Um den aktuellen Zustand eines Projektes zu bekommen, muss man den letzten ganzen Stand des Projektes und all dessen Änderungen danach abrufen, wofür ebenfalls die Revisionsnummer oder das Änderungsdatum genutzt werden kann. Auch in dieser Tabelle darf keine Spalte null sein.

Feld (Spaltenname)	**Typ**	**Bedeutung**
id	INT	automatisch eingetragener Primärschlüssel
revision	SMALLINT	aufsteigende Revisionsnummer für project
changeDate	DATETIME	automatisch eingetragenes Änderungsdatum
editedBy	VARCHAR	Sitzungsnummer des ändernden Nutzers
project	VARCHAR	Name des Projektes der Operation
wholeState	BOOL	ganzer Stand (wahr) oder Änderung (falsch)
operation	TEXT	ausgeführte Operation

Tabelle TAB: Tabelle RTModel

7.6.2.3 Meilenstein 2 – Erweiterungen des STR-Servers

Für Meilenstein 2 wird der STR-Server erweitert werden, um den Clients den Up- und Download von Dateien zu ermöglichen und um Änderungen an den Echtzeit-Datenmodellen zu speichern. In welchem Format die Dateien genau gespeichert werden, steht noch nicht fest. Es wird vereinfachend angenommen, dass die Dateiinhalte als Strings übertragen und in der DB in der Tabelle Files gespeichert und aus ihr abgerufen werden können und dass die Übertragung nicht durch den Inhalt gestört wird.

Der StrServerController braucht für den Up- und Download je eine neue Methode. Beide Methoden benötigen einen Parameter für den Projektnamen und einen für den Pfad, unter dem die Datei auffindbar sein soll. Diese Parameter werden aus dem Pfad des Methodenaufrufs bezogen. Beim Projektnamen wird diese Technik auch bereits anderen Methoden der Klasse RTSyncProjectServerController umgesetzt. Die Methoden werden mit der Annotation SubscribeMapping gekennzeichnet, um es den Clients zu erleichtern für jeden einzelnen Aufruf ein eigenes Callback einzurichten. Statt der Annotation als SubscribeMapping wäre auch die möglich, diese Methoden als MessageMapping zu kennzeichnen, was für andere Programme als das STR-Tool voraussichtlich die bessere Herangehensweise wäre.

Die Methode für den **Upload** benötigt zudem den Dateiinhalt, der in den Nutzdaten des Aufrufs als String übergeben wird. Zudem gibt es einen Parameter für Principal, aus dem mit der Methode getSubscriber der aufrufende Abonnent gefunden werden kann, um dann dessen ID zu bekommen. Die Upload-Methode speichert den übergebenen Dateiinhalt unter Angabe des Projektnamens, der ID und des Pfades in der DB. Falls es bereits einen Eintrag für diesen Dateipfad in diesem Projekt gibt, wird die letzte Revisionsnummer automatisch um 1 erhöht. Andernfalls ist sie 1. Die Rückgabe der Methode ist im Falle eines Fehlers ein aussagekräftiger Fehlertext oder im Erfolgsfall null. Für die Rückgaben eines Fehlers können die Methoden zur Fehlermeldung der Elternklasse als Referenz benutzt werden. Eine mögliche Methodensignatur für den Upload wäre die folgende:

```
@SubscribeMapping("/{project}/uploadFile/{path}")
public String uploadFile(
   @DestinationVariable("project") String project,
   @DestinationVariable("path") String path,
   @Payload String content,
   Principal principal)
```

Die Methode für den **Download** benötigt keinen weiteren Parameter. Sie ruft nur den neuesten Stand der Datei im angegebenen Projekt unter den angegebenen Pfad aus der DB ab und gibt den Dateiinhalt zurück. Zukünftig ließe sich diese Methode noch um einen optionalen Parameter für eine bestimmte zurückzugebende Revisionsnummer erweitern, die anstelle des neuesten Standes genutzt wird. Eine mögliche Methodensignatur für den Download wäre:

```
@SubscribeMapping("/{project}/downloadFile/{path}")
public String downloadFile(
   @DestinationVariable("project") String project,
   @DestinationVariable("path") String path,
   Principal principal)
```

In Meilenstein 2 wird die OperationHistory des Servers auf dauerhafte Datenspeicherung mit der DB ausgelegt. Dies wird über die Verwendung einer neuen Klasse namens **DBOperationHistory** erreicht. Sie implementiert das Interface TaggedUserOperationHistory und ist im Prinzip eine Kopie der CustomHistory mit bestimmten Änderungen, die dazu dienen alle Operationen mit der Tabelle RTModel in der DB synchron zu halten. Dazu wird ein zusätzlicher Konstruktorparameter für die Verbindung zur DB und für den Projektnamen übergeben und in geeigneten Datenfeldern abgelegt. Im Konstruktor muss die DBOperationHistory entweder die gesamte Historie des Projektes oder dessen aktuellen Stand aus der DB abrufen und in die Map _operations und die ganzen Stände ins Set _wholeStateOperationIds eingefügt werden. Wenn der Abruf fehlschlägt, muss dieser Fehler geloggt und schnellstmöglich von einem Entwickler behandelt werden, denn er sollte nicht auftreten. Es sollte aber beachtet werden, dass für neue Projekte kein Model vorhanden ist und dass dieser Fall kein Fehler ist. Die DBOperationHistory muss im Gegensatz zur CustomHistory jedes Anlegen eines Eintrages in der Map _operations erst nach einem Speichervorgang in der DB

ausführen. Bei jedem put-Aufruf in _operations kann dabei von einem Parameter für wholeState und einem eventuellen begleitenden put-Aufruf in _wholeStateOperationIds der Paramter für die Tabellenspalte wholeState genutzt werden. Die id für den put-Aufruf wird als Revisionsnummer für die Datenbanktabelle genutzt. Die CustomHistory hat bereits die von otter-java bereitgestellte DefaultHistory so erweitert, dass zusätzlich zur Operation auch der Nutzername übergeben wird, aus dem die Sitzungsnummer für die Tabellenspalte editedBy bezogen wird. Falls das Einfügen eines Tabelleneintrags fehlschlägt, muss eine Ausnahme geworfen werden, die über die EditorControl, den OperationSync und den StrServerController als Fehlernachricht an den Client weitergegeben wird, da jedes Speichern in der OperationHistory von einem Client ausgelöst worden sein sollte.

7.6.3 RTSync Client im STR-Tool

7.6.4 Integration des Echtzeit-Datenmodells

7.6.5 Änderungen an der grafischen Benutzeroberfläche

8. Ergebnisse und Bewertung

Es ist in dieser Arbeit gelungen, das Projekt namens RTSync für die Echtzeit-Synchronisierung zwischen Clients und einem Server für die Programmiersprache Java zu erstellen. Dies kann als Programmbibliothek verwendet werden. Es ermöglicht entweder die Nutzung eines simplen Datenmodells, welches komplett im Netzwerk geteilt wird, oder die Unterteilung in projektbezogene Daten, wobei Clients das Datenmodell eines oder mehrerer bestimmter Projekten miteinander und dem Server teilen.

Zur Anzeige der Daten in Echtzeit sind vier Oberflächenelemente geschaffen worden. Das erste zeigt die im Falle eines simplen Modells alle Nutzer und im Falle des projektbezogenen Modells alle Bearbeiter des Projektes mit ihrem Namen und ihrer Farbe an. Das zweite und dritte Oberflächenelement stellen den Zustand eines geteilten Objektes in Echtzeit dar und heben dabei kürzlich hinzugefügte Stellen in der Farbe des Autors hervor. Das zweite ist ein Textfeld für einen geteilten String, welches auch die Einfügemarken anderer Autoren in ihrer Farbe an ihre jeweils letzte editierte oder ausgewählte Stelle setzt. Das dritte ist eine Ansicht aller geteilten Objekte eines Datenmodells in einer Baumstruktur. Das vierte ist eine Liste aller verfügbaren Projekte, wobei dahingehend unterschieden wird, ob ein Projekt nur lokal, nur auf dem Server oder an beiden Orten existiert.

Für beide Modelltypen sind Beispielanwendungen für die Demonstration der Echtzeit-Synchronisierung mit einer grafischen Benutzeroberfläche für die Clients geschaffen worden. Die Beispielanwendung der Nutzung des simplen Datenmodells demonstriert den Austausch eines einzelnen Strings. Diejenige für das projektbezogene Datenmodell erlaubt es jedem Client, neue geteilte Listen, Maps und Strings hinzuzufügen, Strings zu ändern und diese Modellobjekte zu löschen.

Die Voraussetzung für die Nutzung des aktuellen Standes von RTSync ist jedoch, dass alle Clients eine ununterbrochene Verbindung zum Server haben. Bei jeder neuen Verbindung zu einem Server setzt der jeweilige Client den Zustand des abgerufenen Datenmodells auf den des Servers zurück und überschreibt damit eventuelle lokale Änderungen. Im aktuellen Stand wird beim Zurücksetzen jedoch eventuell noch die letzte Änderung des Clients gesendet.

Die Integration von RTSync ins STR-Tool ist bisher noch nicht geschehen und es gibt auch noch keinen STR-Server. Es ist jedoch eine textuelle Beschreibung der nötigen Änderungen am STR-Tool erstellt worden, anhand welcher der Leser dies schnell umsetzen kann.

Für den Server gibt es noch keine Datenbank und auch noch keine andere persistente Datenhaltung. Für die Anbindung einer Datenbank ist allerdings schon die Versionierung, inklusive des ändernden Nutzers vorbereitet worden. Dazu müsste die aktuell verwendete Historie in RTSync vom Verwalten der Stände und Änderungen im Arbeitsspeicher auf Datenbankoperationen geändert werden. Der Server bietet aktuell auch noch keine Möglichkeit, Dateien hoch- oder herunterzuladen, was jedoch für das STR-Tool benötigt wird.

Von den zahlreichen eingangs gestellten Fragen (s. 5 Zielstellung) sind nur wenige offen geblieben. Ungeklärt ist noch, ob bestimmte STR-Projekte für einige Nutzer unsichtbar bleiben sollen. Derzeit hat RTSync noch keine Nutzer-, und Rechteverwaltung, um diese umsetzen zu können und die Frage ist bisher noch von zu geringer Wichtigkeit für eine genaue Klärung gewesen. Für externe Test-Dienstleister ohne Datenbankzugriff bleibt nur die Möglichkeit, die dateibasierten Projekte zu verwenden, wie es bisher auch möglich war.

Besonders erwähnenswert ist, dass das Verfahren der Echtzeit-Synchronisation an sich gänzlich ohne Sperrung einer Datei oder eines Datenbereichs über längere Zeit verwendet werden kann. Die einzigen Sperrungen entstehen für kritische Abschnitte im Sinne des Multithreadings, welche sicherstellen, dass gleichzeitige Änderungen an geteilten Echtzeit-Datenobjekten nicht miteinander konkurrieren. Die aktuelle Version erfordert, wie oben bereits erwähnt, eine dauerhafte Verbindung jedes Clients zum Server. Wenn das STR-Tool mit dieser Version für Online-Projekte verwendet wird, sollten Online-Projekte solange gesperrt werden, bis das STR-Tool wieder mit dem STR-Server verbunden ist. Dafür kann die Funktionalität des Sperrmechanismusses für Offline-Projekte wiederverwendet werden.

9. Schlussbemerkungen und Ausblick

Möglicherweise wäre weniger Aufwand entstanden, wenn nicht otter-java als zugrunde liegende Programmbibliothek für diese Arbeit verwendet worden wäre, sondern google-mobwrite.

In zukünftigen Versionen von RTSync sollten sich die Clients in geeigneter Weise ihre noch nicht an den Server übermittelten Änderungen speichern und beim nächsten Verbinden den Nutzer entscheiden lassen, ob er sie verwerfen, immer noch anwenden oder, falls nötig, mit einer weiteren Änderung zusammenführen will.

Die Datensynchronisierung von RTSync läuft über zwei verschiedene Verfahren ab. Das erste synchronisiert alle Modelldaten, wobei so viel Funktionalität von otter-java genutzt wird, wie möglich. Das zweite wird für das Senden von Daten verwendet, welche nicht direkt zu Datenmodell gehören. Dies betrifft die Nutzerdaten, d.h. ihre Anzeigenamen und -Farben, sowie ihre aktuell in Bearbeitung befindlichen Projekte und auch die Liste aller Projektnamen. Dieses zweite Verfahren nutzt weder otter-java, noch eine andere Form von OT und sendet bei jedem Sendevorgang die vollen Daten. Die neuen Daten werden dann bei jedem Client mit seinen bisherigen Daten abgeglichen, um ggf. angezeigte Farben bei neu eingefügten geteilten Objekten zu ändern. Für eine neuere Version von RTSync böte es sich an, nur das erste Verfahren zu nutzen und die Daten des zweiten Verfahrens als besondere Bestandteile des Datenmodells zu senden. Diese Metadaten ließen sich dann mit OT als geteilte Objekte auffassen. Bei projektbasierten Datenmodellen könnte man dies auch über ein neues Projekt „Meta“ erreichen, welches die Nutzer- und Projektlisten enthält. Alle Clients melden sich dann zusätzlich für Änderungen am Meta-Projekt an und könnten so die Änderungen empfangen.

Die Beispielanwendung von RTSync für das simple Datenmodell könnte noch erneuert werden, um nicht nur einen einzelnen String ändern zu können, sondern das ganze Datenmodell – so wie es auch bei der Beispielanwendung für das projektbasierte Datenmodell möglich ist. Dazu müsste noch das RTProjectViewPanel angepasst und in ein geeigneteres Unterprojekt verschoben werden.

Eine Verbesserungsmöglichkeit bietet auch der RTSharedTextComponentAdapter, der vom RTJTextPane verwendet wird, da dieser derzeit nicht nur die Veränderungen am geteilten String aus den bei der Änderung generierten DokumentEvents erfasst, sondern stattdessen eine Änderung vom alten zum neuen String mit dem Diff aus diff-match-patch berechnet. Dies ist früher nötig gewesen, als dafür noch otter-javas SharedStringImpl verwendet worden ist, deren insert-Methode fehlerhaft war (s. 7.3Änderung an otter-java). Dieser Fehler ist nun behoben, sodass die unnötige Verzögerung durch den Diff entfallen kann.

10. Literaturverzeichnis

STR-DB: Michael Küttner, STR-Tool Datenbank, 2023

OT Wiki: , Operational transformation, 2025, https://en.wikipedia.org/w/index.php?title=Operational_transformation&oldid=1287538213

otter-java: Andreas Holstenson, Otter, , https://github.com/LevelFourAB/otter-java