Entwurf

PSE Projekt SoSe 2013

Entwicklung mobiler Clients zur Lagedarstellung

14.06.2013

PSE Gruppe: Daniel Reichert, Kai Westerkamp, Pascal Burkhardt, Tim Reiter, Mischa Carl, Jacqueline Rutschke

Inhaltsverzeichnis

1 Einleitung

Beim Entwurf lag der Hauptfokus auf der einfachen Erweiterbarkeit und Portierbarkeit der Anwendung. Sämtliche graphischen Darstellungen werden durch Plugins gesteuert. Die Widerverwendbarkeit des Codes wurde duch mehrere Techniken gewährleistet. Es wird das Entwurfsmuster zur Trennung der Komponenten verwendet. MVP [53] steht für Modell-View-Presenter und ist eine Abwandlung des bekannten Entwurfsmusters. Im Gegensatz zu MVC [53] kommuniziert im MVP ausschließlich der Controller, hier Presenter genannt, mit der View (siehe hierzu auch Abbildung 1.1). Verwendet man das MVP Entwurfsmuster, kann man zwei Wege gehen: Den des "Supervising Controller" oder den der "Passive View". Bei ersterem aktualisiert sich die View immer selbst, bei letzterem wird die Aktualisierung durch den Presenter ausgelöst. Das Ziel möglichst viel Logik frameworkunabhängig zu halten und damit die Oberfläche so leichtgewichtig wie möglich zu gestalten ist am besten mit der "Passive View" zu erreichen. Die Datenhaltungsschicht, das Model, hat bei uns sowohl eine Clientseitige Ausprägung, die die Daten zwischenspeichert und deren Konsistens auch bei Verbindungsabrüchen gewährleistet, als auch eine Serverseitige Ausprägung welche die Anbindung an das Backend und Datensynchronisation ermöglicht. Bekommt der Presenter vom Model nun neue Daten zugeschickt, so übergibt er diese Daten einem DataPluginManager der passende DataPlugins sucht, welche nun über die View die Repräsentation der Daten ermöglichen.
Die View selbst ist eine Schnitstelle hinter der sich die konkrete Implementierung, hier in Gwt [53], verbirgt. Dies stellt den zweiten Punkt dar, der in unserem Entwurf die Widerverwendbarkeit des Codes sicherstellt: Objekte die eine graphische Repräsentation haben, werden immer hinter Schnittstellen verborgen. (siehe hierzu Abbildung 4.16)
Im Übersichtsdiagramm im Kapitel 2 sind die Zugehörigkeiten der einzelnen Komponenten zu Model, View oder Presenter nochmals verzeichnet. Des weiteren zeigt das Diagramm mehrere mit blau gestrichelten Linien, welche Arbeitspakete (Kürzel AP) repräsentieren und so in Entwurf und Implementierung die einzelnen Aufgabenbereiche der Projektmitglieder trennen. Auch hier kommen zur Entkoppelung der Arbeitspakete wieder Interfaces zum Einsatz.
In der View wurde nochmals besonderen Wert darauf gelegt, die für die Darstellung der Karte zuständige Komponenten durch Einführung eines Interfaces zu entkoppeln. Damit ist es möglich, das Framework welches die Kartendarstellung ermöglicht, unabhängig von der restlichen graphischen Implementierung zu tauschen.

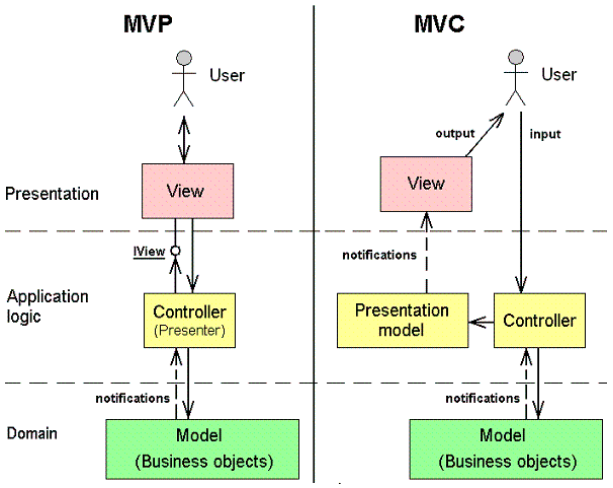
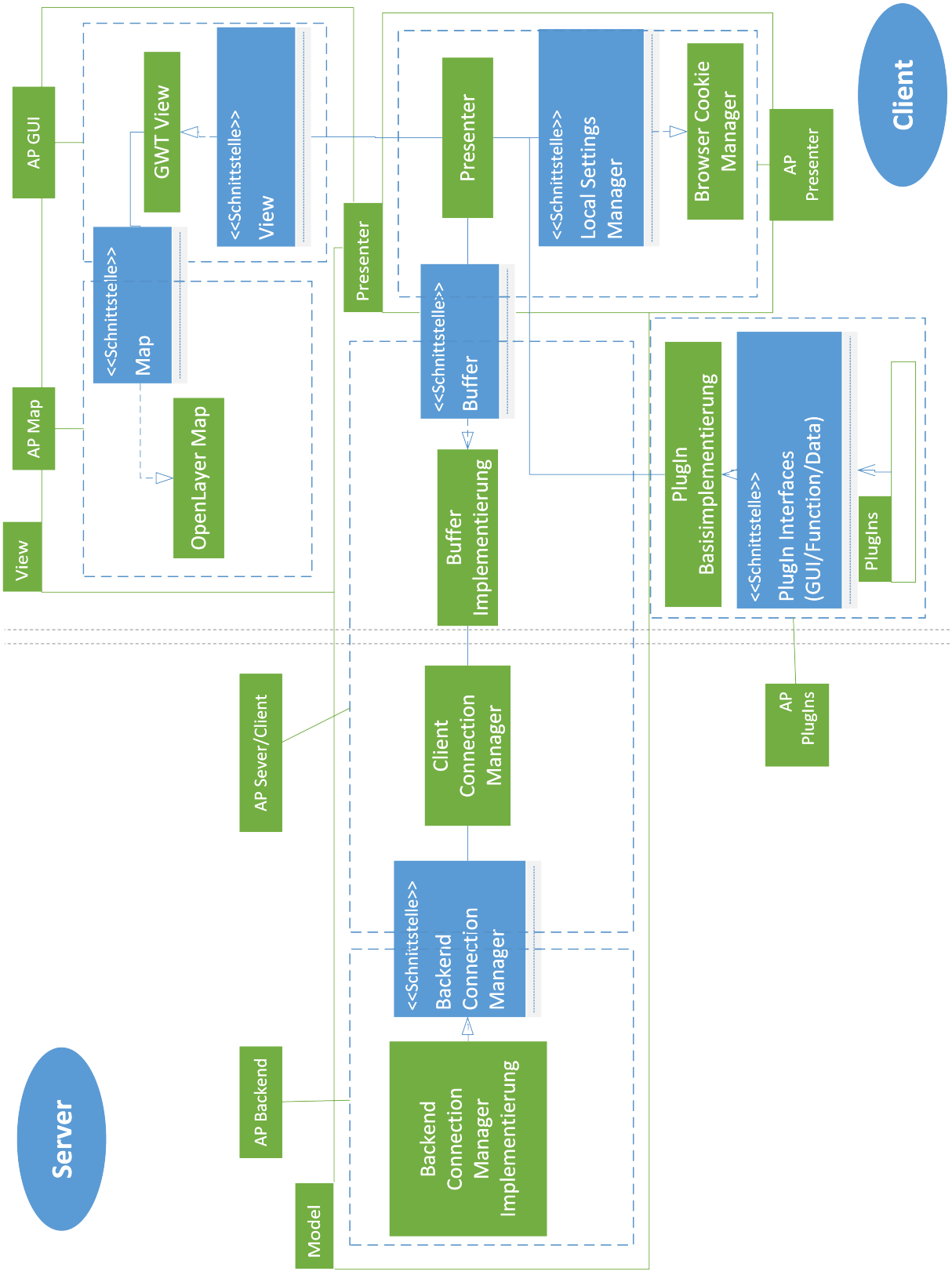


Abbildung 1.1: Unterschiede zwischen MVP und MVC ©http://www.c-sharpcorner.com/

2 Übersicht

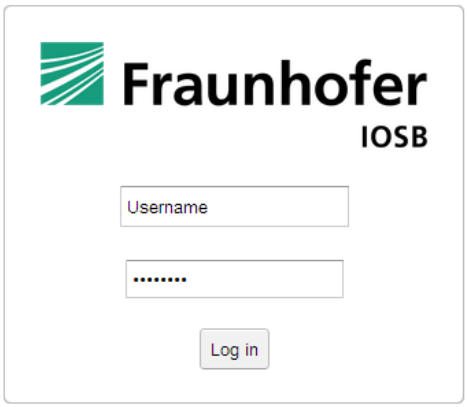

Client
Server
AP GUI
GWT View
<<Schnittstelle>>
View
Presenter
Presenter
<<Schnittstelle>>
Local Settings
Manager
Browser Cookie
Manager
AP
Presenter
<<Schnittstelle>>
Map
AP Map
OpenLayer Map
View
<<Schnittstelle>>
Buffer
Buffer
Implementierung
PlugIn
Basisimplementierung
<<Schnittstelle>>
PlugIn Interfaces
(GUI/Function/Data)
PlugIns
AP
PlugIns
AP Sever/Client
Client
Connection
Manager
<<Schnittstelle>>
Backend
Connection
Manager
AP Backend
Backend
Connection
Manager
Implementierung
Model

3 View

Die View kennt zwei Zustände: In der einen zeigt sie den Login-Bildschirm an, in der anderen wird das eigentliche Hauptprogramm dargestellt. Diese Hauptansicht besteht aus zwei Komponenten: der Hud [53]view und der eigentlichen Karte. Beide sind durch Interfaces untereinander und vom Rest des Codes abgekoppelt um Austauschbarkeit zu gewährleisten. Die gesamte View kann von außen über das Interface 'View' angesprochen werden. Die konkrete Kontrolle der mit -Komponenten implementierten View erfolgt über die Klasse 'GwtView', welche daher auch das Interface 'View' implementiert. Mit dem Interface 'Map', über welches die View mit der Karte kommuniziert, wird die Austauschbarkeit der Kartenimplementierung gewährleistet. Diese erfolgt in diesem Projekt mit OpenLayer, entsprechend heißt die Implementierung der Map auch 'OpenLayerMap'. Über der Karte liegt die sogenannte 'HudView' (= Head-up-Display), welche alle Werkzeuge bereitstellt mit der die Map bearbeitet werden kann.

3.1 Login

Der Login Bildschirm gibt dem User die Möglichkeit sich mit Username und Passwort anzumelden. Sollte die Anmeldung fehlschlagen wird ihm dies durch einen enstprechenden visuellen Effekt (z.B Verfärbung der Eingabefelder in Rot) deutlich gemacht.



3.2 Hud

3.2.1 Funktionale Bescheibung

Die folgende Beschreibung der funktionalen Aspekte der 'HudView' bezieht sich auf die standardmäßig initalisierten Plugins. Spätere Versionen mit anderen Plugins können anders aufgebaut sein und andere Funktionen bereitstellen. Jeder Button der auf der HudView angezeigt wird ist auf Tablets mindestens 1.5 bis 2cm groß. Die Icons werden dabei bei kleiner werdender Bildschirmgröße überproportional größer, so dass man sie auf kleinen Bildschirmen gut ansteuern kann und sie bei größeren nicht zu viel Platz verschwenden. Bei der Implementierung der Buttons muss im Besonderen beachtet werden, dass normale Buttons in einem mobilen, Touch-orientierten Browser (z.B Chrome mobile), mit einer Verzögerung von 300ms geöffnet werden. Solche und andere unerwünschten Standardverhalten von mobilen Browsern wie z.B Double-Tap-Zoom müssen in der Implementierung abgefangen werden. Das Head-up-Display zeigt im initialen Zustand drei Buttons. Einen Kompass oben links, der bei Click die Karte in Nord-Süd Richtung ausrichtet und den Winkel der Karte anzeigt, einen 'Navigations-Button' unten links und einen 'Tools-Button' unten rechts, welche die zugehörigen Menüs öffnen.
Beim Schließen eines Menüs werden alle Submenüs und Funktionen innerhalb des Menüs geschlossen bzw. deaktiviert. Die folgenden Mockups sind 1:1 Darstellungen der View auf einem iPad mini (Auflösung: 1024x768px, Bildschirmdiagonale 7.9 Zoll, 168 ppi). Alle Icons sind nur zur Demonstrationszwecken und nicht final.

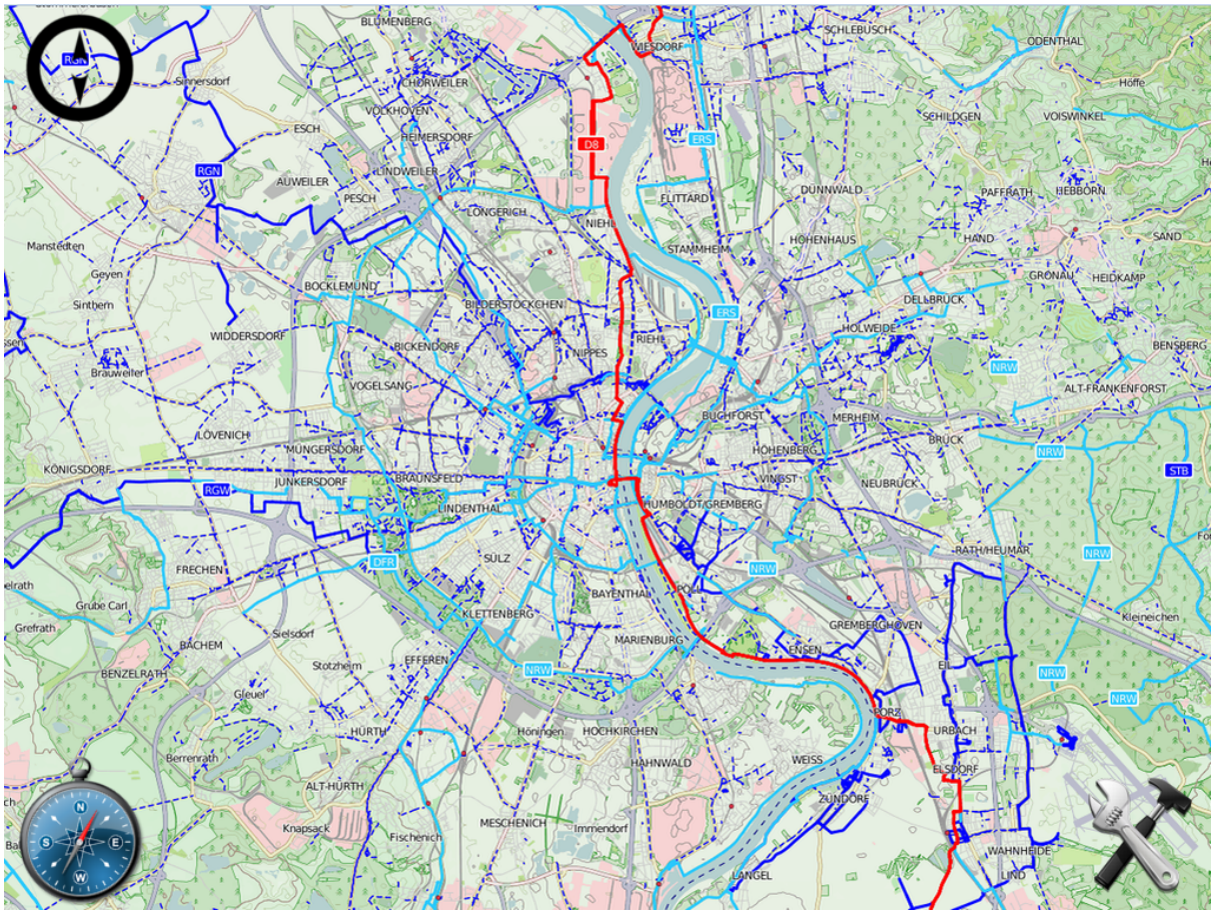


Abbildung 3.1: Default Benutzeroberfläche

Das Navigationsmenü beinhaltet:

- Einen Button um die Karte zu drehen.
- Einen Button um in die Karte hinein zu zoomen.
- Einen Button um aus der Karte heraus zu zoomen.

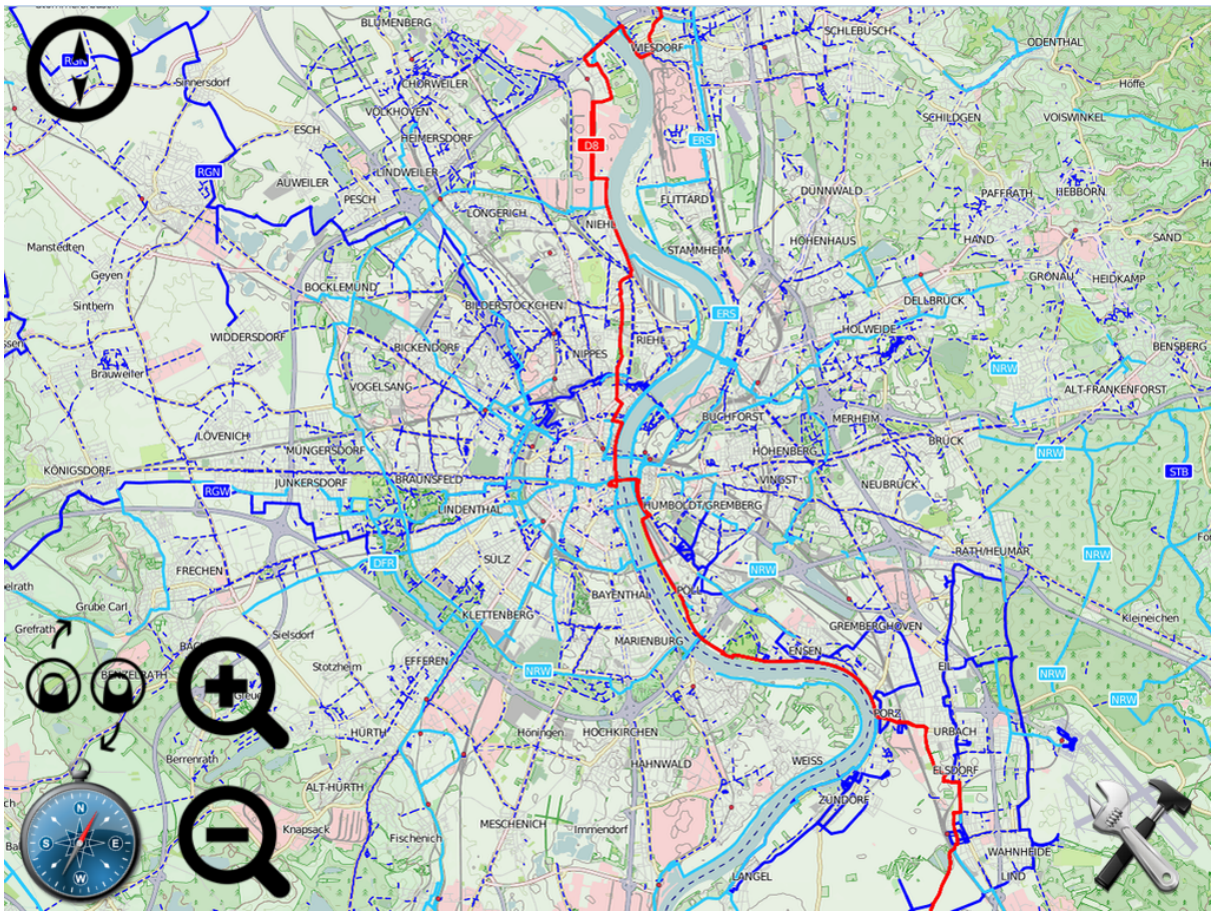


Abbildung 3.2: Navigationsmenü

Das Tools Menü beinhaltet:

- Einen Button um die Karte zu editieren.
- Einen Button um in der Karte zu zeichnen.
- Einen Button um Taktische Symbole hinzuzufügen.
- Einen Button um Einstellungen zu ändern.

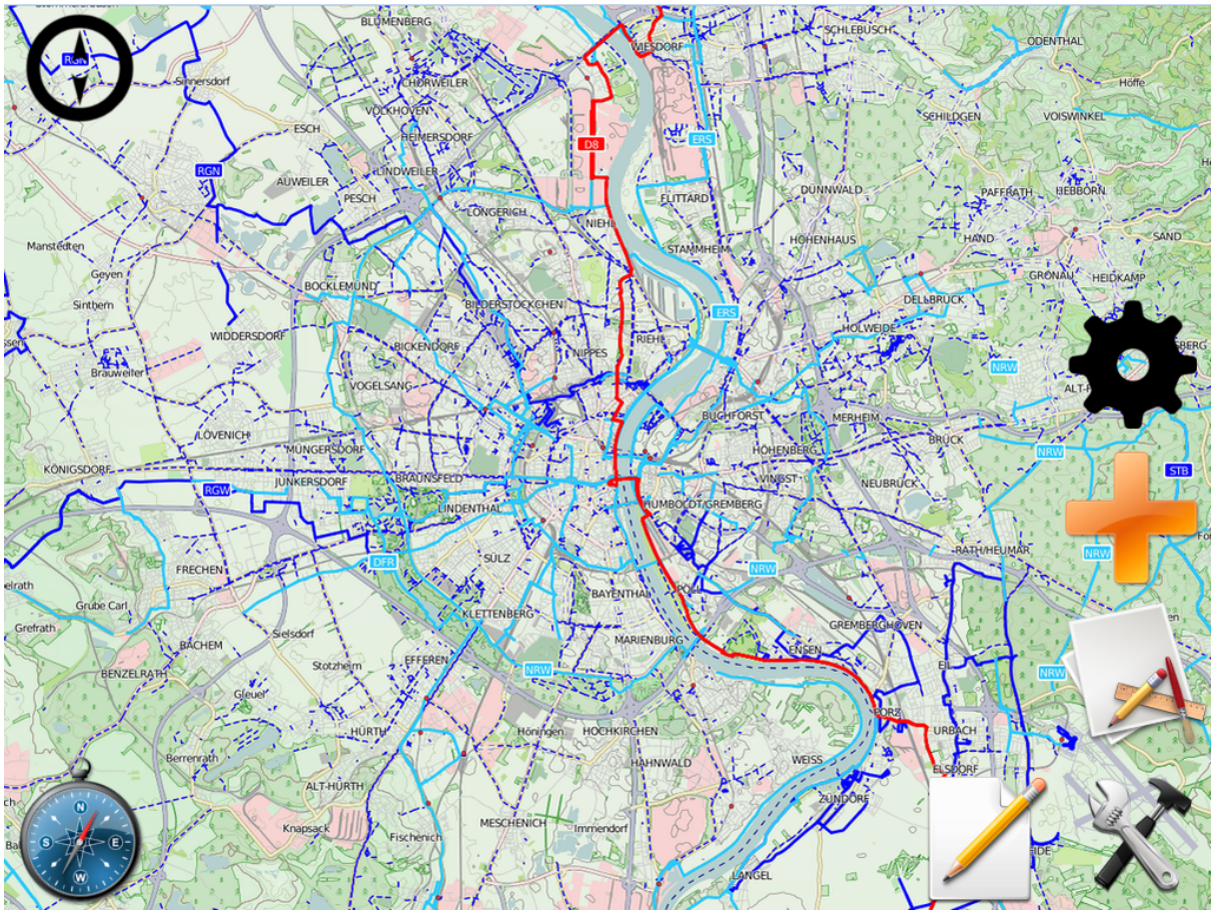


Abbildung 3.3: Toolmenü

Beim Klick auf den Zeichen-Button werden die verschiedenen Zeichenoptionen eingeblendet, wobei jeweils eine Option zum sofortigen Zeichnen vorselektiert ist. Die vorselektierte Option enspricht immer der zuletzt selektierten.

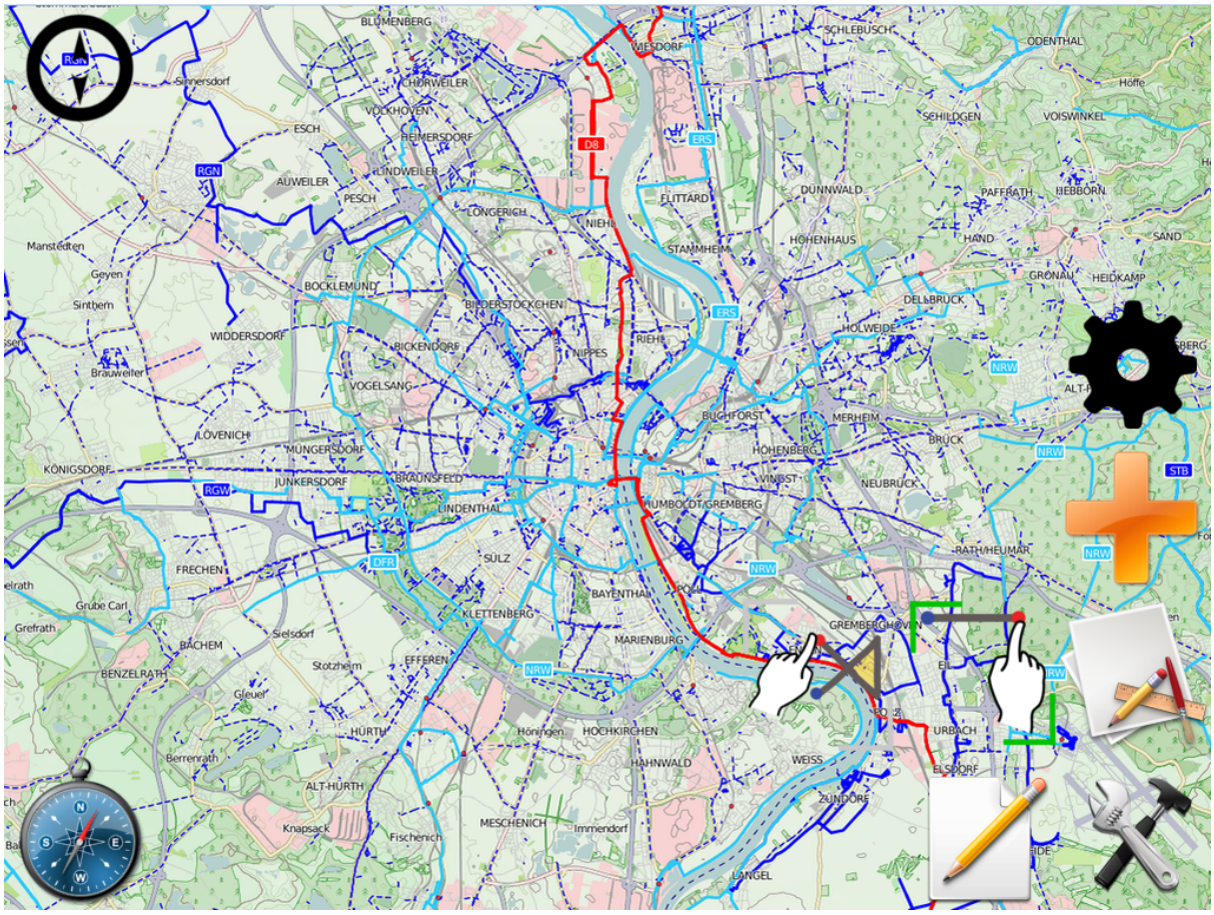


Abbildung 3.4: Zeichenmenü

Mit einem Klick auf den Bleistift wird der Edit-Mode aktiviert. Hier kann der Benutzer Einzeichnungen, wie Linien, Punkte oder Polygone, sowie taktische Zeichen auswählen und dann bestimmte Eigenschaften verändern. Man kann über zwei verschiedene Wege Einzeichnungen auswählen: Entweder klickt man die Einzeichnung direkt an (was bei kleinen Einzeichnungen wie Punkten oder Linien schwierig ist), oder man benutzt das Box-Select Tool. Das Box-Select Tool aktiviert man mithilfe des Icons, das über dem Edit-Mode Icon erscheint sobald der Edit-Mode aktiviert ist. Wenn das Box-Select Tool aktiviert wurde kann der Benutzer ein Rechteck auf der Karte malen. Nach dem Einzeichnen des Rechtecks werden alle Einzeichnungen, die (ganz oder teilweise) vom Rechteck überdeckt werden, ausgewählt und das Box-Select Tool wird automatisch deaktiviert. Mithilfe des Box-Select Tools lassen sich also kleine Einzeichnungen, wie Punkte und Linien, gut auswählen. Außerdem lassen sich so mehrere Einzeichnungen gleichzeitig anwählen.

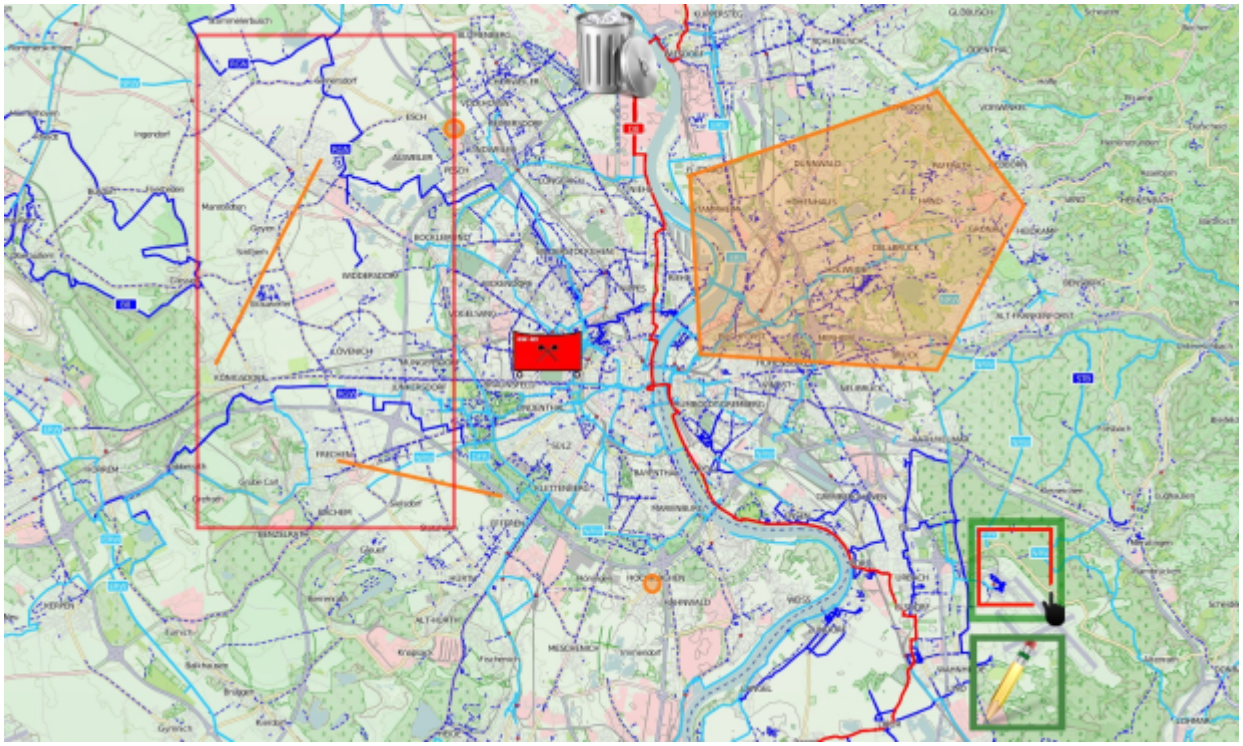


Abbildung 3.5: Box-Select-Tool

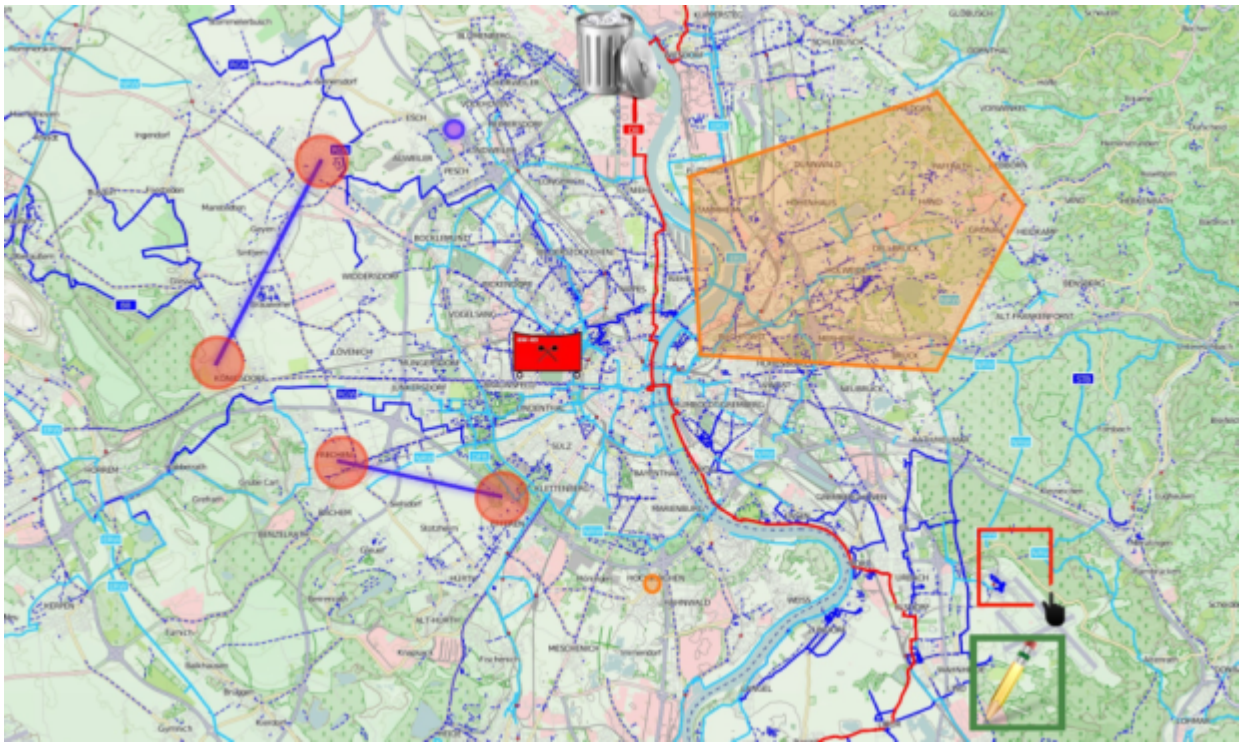


Abbildung 3.6: Durch das Box-Select-Tool ausgewählte Einzeichnungen

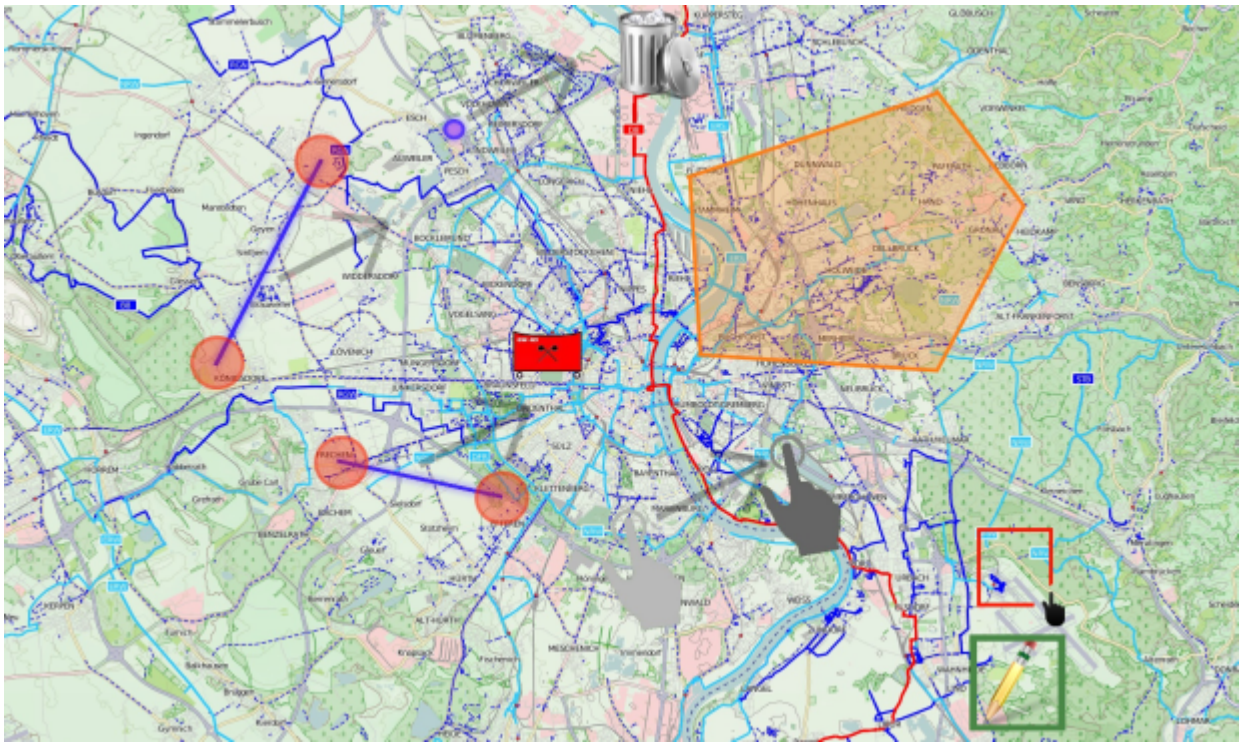


Abbildung 3.7: Durch eine Zieh-Bewegung irgendwo auf der Karte wird die Auswahl relativ zur Fingerposition verschoben

Im Edit-Mode ist ein Bewegen der Karte mit Zieh-Bewegungen nicht möglich. Die Karte lässt sich dennoch verschieben: Falls der Benutzer beim Verschieben eines Objekts oder Vertex mit dem Finger an den Rand des Bildschirms kommt wird die Karte in die passende Himmelsrichtung verschoben.

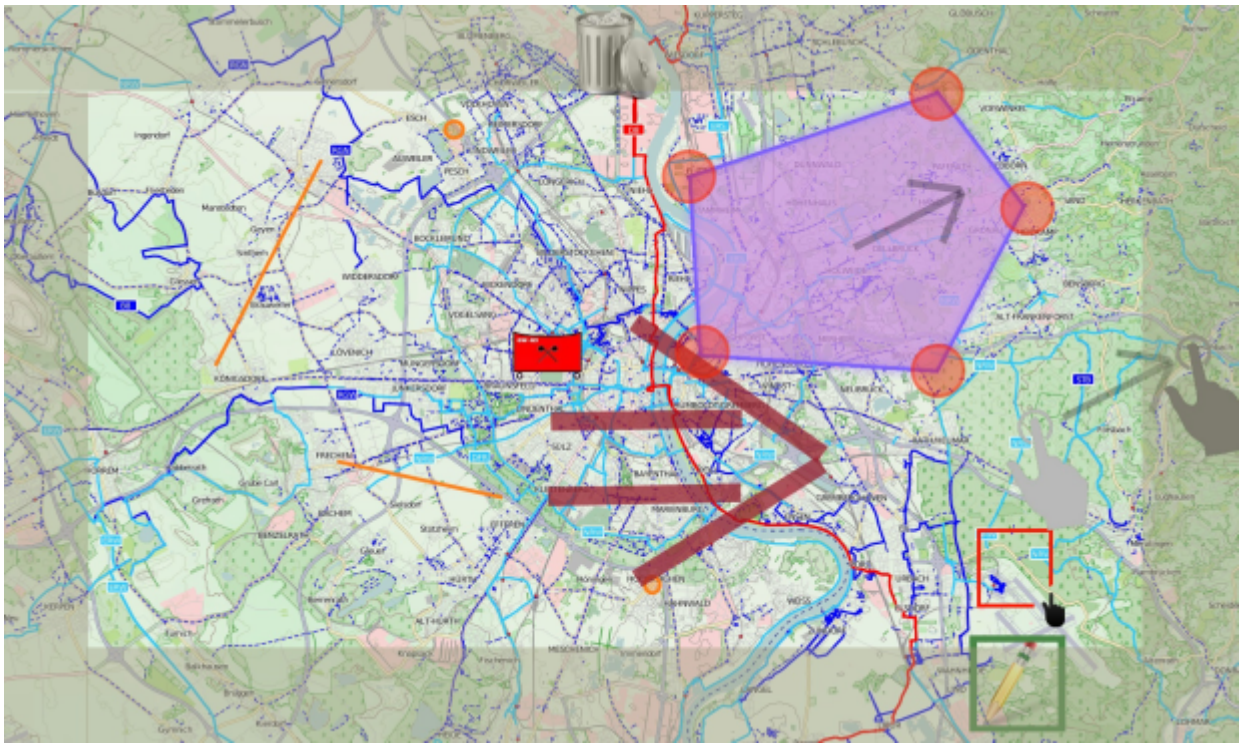


Abbildung 3.8: Bewegung der Karte bei ausgewähltem Objekt im Edit-Modus

Nachdem eine Einzeichnung ausgewählt wurde, lassen sich folgende Eigenschaften verändern:

- Position
- Positionen der einzelnen Vertices (bei Linien und Polygonen)
- Löschen des ganzen Objekts
- Löschen und Hinzufügen von Vertices

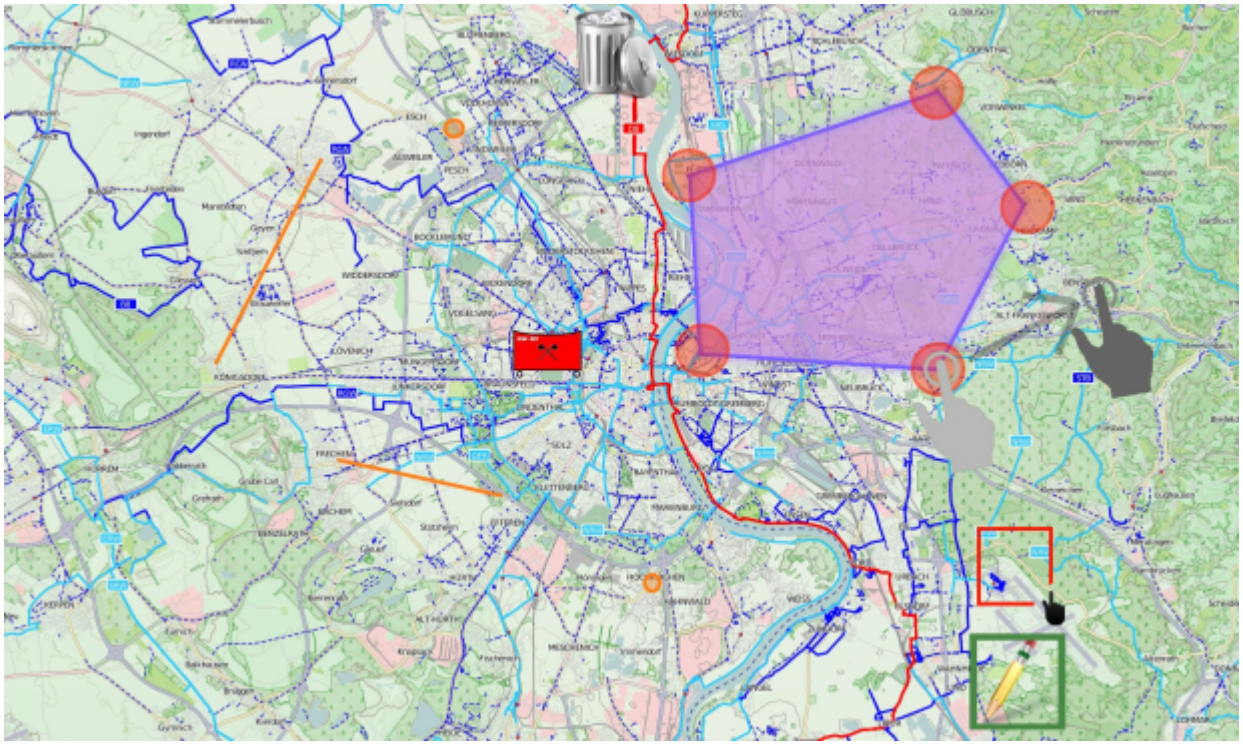


Abbildung 3.9: Verändern der Position eines Vertex

Beim Klick auf den 'Taktisches Zeichen hinzufügen' Button, im MockUp durch ein + repräsentiert, öffnet sich ein Menü das in Gruppen sortierte Taktische Symbole anbietet. Das Menü ist dabei im Stile einer SplitView aufgebaut. Dieser Modus hat im Vergleich zu einem Menü in dem die Gruppierungen im Menükopf sind den Vorteil, dass man mehr Platz auf der im kleineren Y Achse hat. Beide 'Views' der SplitView, also die Gruppierungen und der rechte Anzeigebereich sind individuell auf der Y-Achse scrollbar.

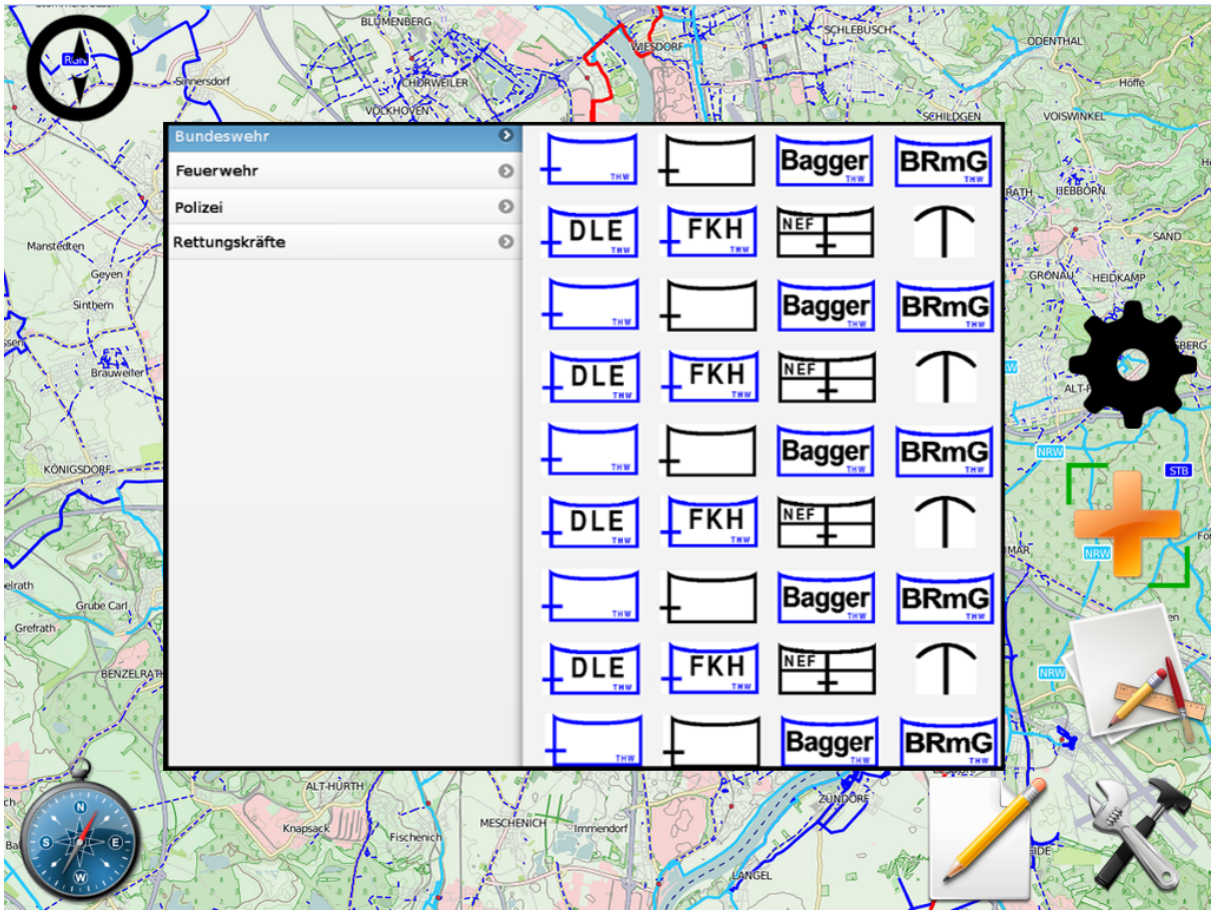


Abbildung 3.10: Menü mit Taktischen Symbolen

Der User kann nun ein Taktisches Symbol zum Einfügen anwählen. Dies ist auf zwei Wegen möglich: Entweder er zieht das Objekt auf die Karte (im folgenden Bild 4.11 mit einem Handsymbol und rotem Pfeil dargestellt), wobei das Menü sobald er seinen Finger auf das Symbol senkt verschwindet, oder aber er klickt einmal das gewünschte Symbol an und klickt dann an der einzufügenden Stelle auf die Karte. In beiden Fällen verschwindet das Menü und ein bestätigen Button erscheint neben dem 'Taktisches Symbol hinzufügen' Button. Beim Klick darauf werden alle eingefügten Symbole übernommen und das Menü zum eventuellen weiteren einfügen von Objekten wieder geöffnet. Um das Mehrfacheinfügen gleicher Symbole zu erleichtern erscheint, nach dem man ein Symbol zum Einfügen im Menü gewählt hat, ein Icon oben links ('Symbol Stack Icon') welches das momentan angewählte Taktische Symbol darstellt. Durch ein weiteres Klicken auf die Karte an einer bestimmten Position oder ein ziehen des Symbols vom dem 'Symbol Stack Icon' oben links an eine Position fügt erneut das enstprechende Symbol der Karte hinzu. Damit ist sichergestellt das der User nicht jedesmal das Menü öffnen muss wenn er dasselbe Symbol erneut hinzufügen will. Um den User auf diese Funktionalität aufmerksam zu machen fliegt das im Menü gewählte Objekt in einer Animation nach oben links. Die Animation ist im folgenden Bild mit dem goldenen Pfeil angedeutet.
Sollte sich der User beim Einfügen des Objekts geirrt haben, kann er dieses in diesem Modus jederzeit

löschen in dem er es über den Mülleimer, welcher sich mittig zentriert am oberen Bildschirmrand befindet, schiebt. Während der User sich im 'Taktisches Symbol einfügen' Modus befindet kann er nicht mit der Karte interagieren um eventuelle Doppelbelegung von Gesten zu verhindern. Er kann die Karte allerdings bewegen in dem er das Objekt an den Rand der Karte bewegt und diese so in eine 'Bildschirmrichtung' verschiebt.

Ein erneuter Klick auf den 'Taktisches Symbol hinzufügen' Button führt im 'Taktisches Symbol einfügen' Modus zum selben Ergebnis wie ein Klick auf den bestätigen Button. Ein Klick auf jeden andern Button führt dazu das alle einegefügten Symbole zwar übernommen werden, allerdings sich nicht erneut das 'Taktisches Symbol hinzufügen' Auswahlmenü öffnet.

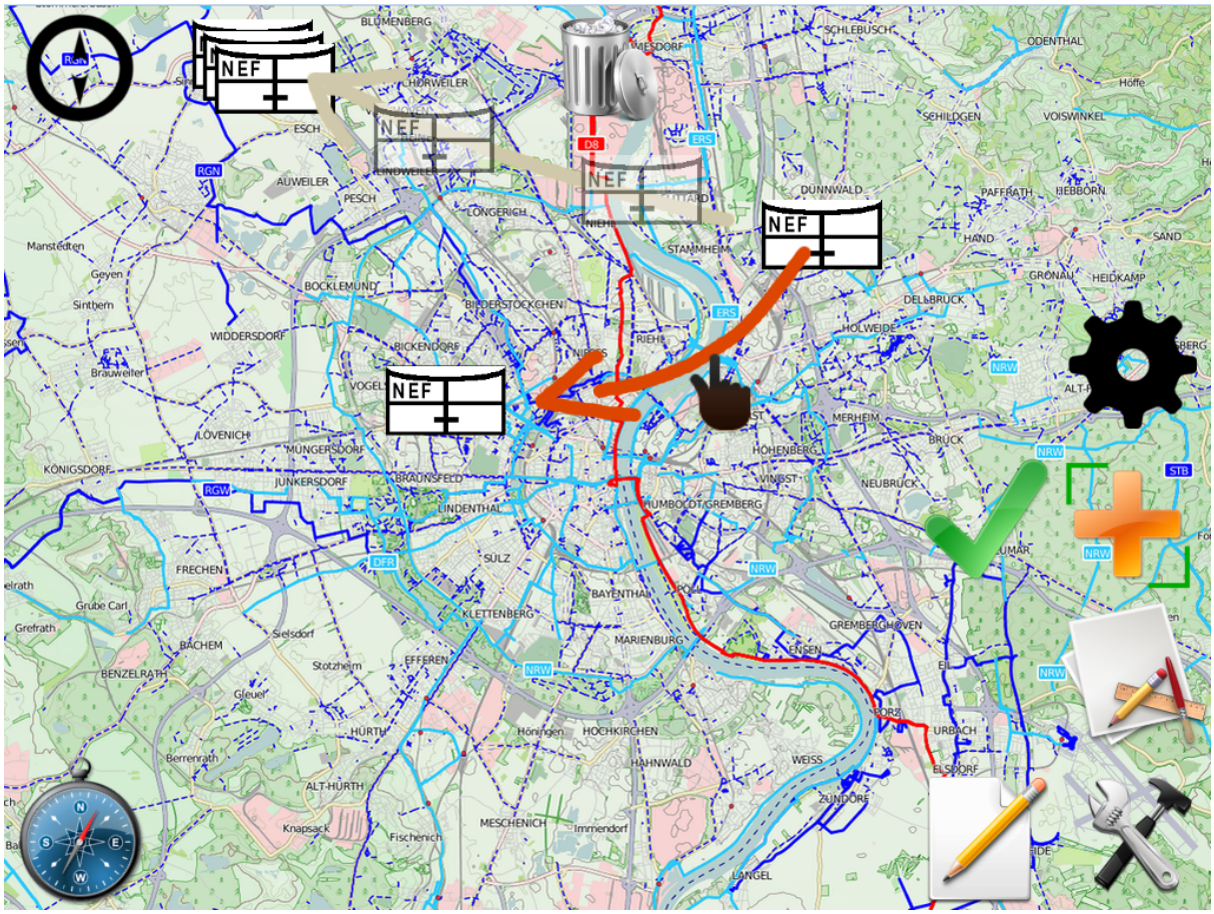


Abbildung 3.11: Taktisches Symbol zeichnen

Über einen Klick auf den Settings-Button öffnet sich das Settingsmenü, das die Möglichkeit bereitstellt, die Auswahl der angezeigten Layer zu verändern, das Szenario zu ändern und allgemeine Einstellungen vorzunehmen. Ändert man die Einstellung von Rechts- auf Linkshänderbedienung wandern alle an der linken Seite eingezeichneten Buttons nach rechts und visa-vers.

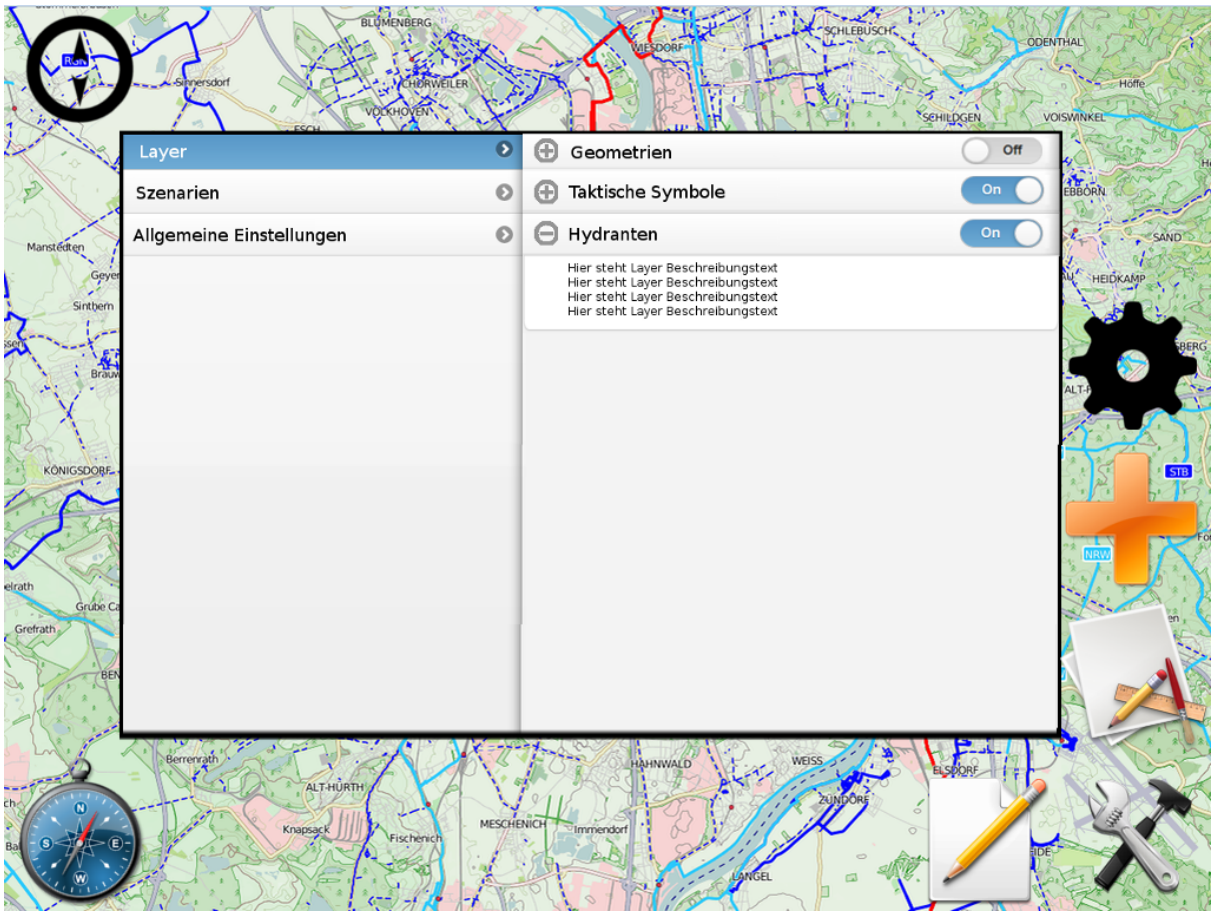


Abbildung 3.12: Layerauswahl

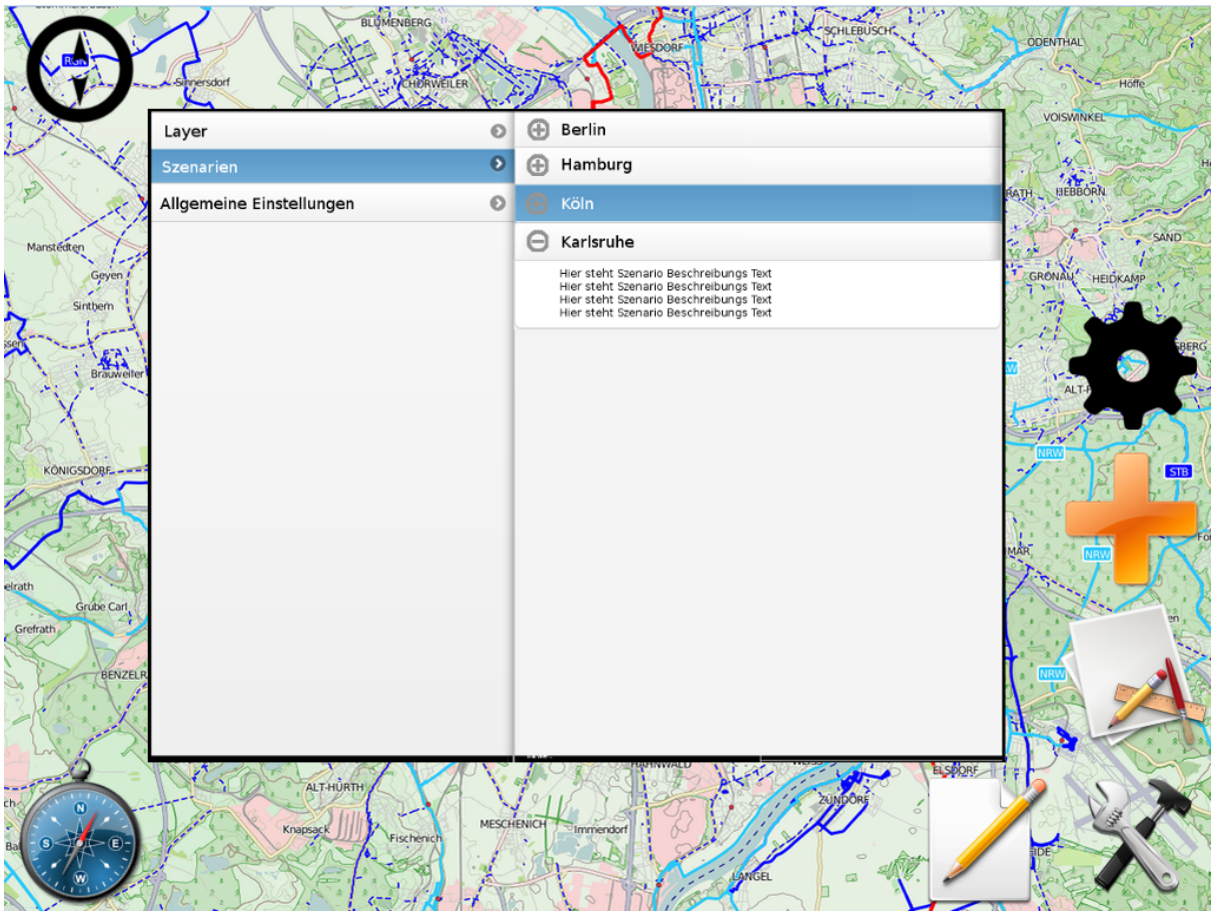


Abbildung 3.13: Szenen Auswahl Menü

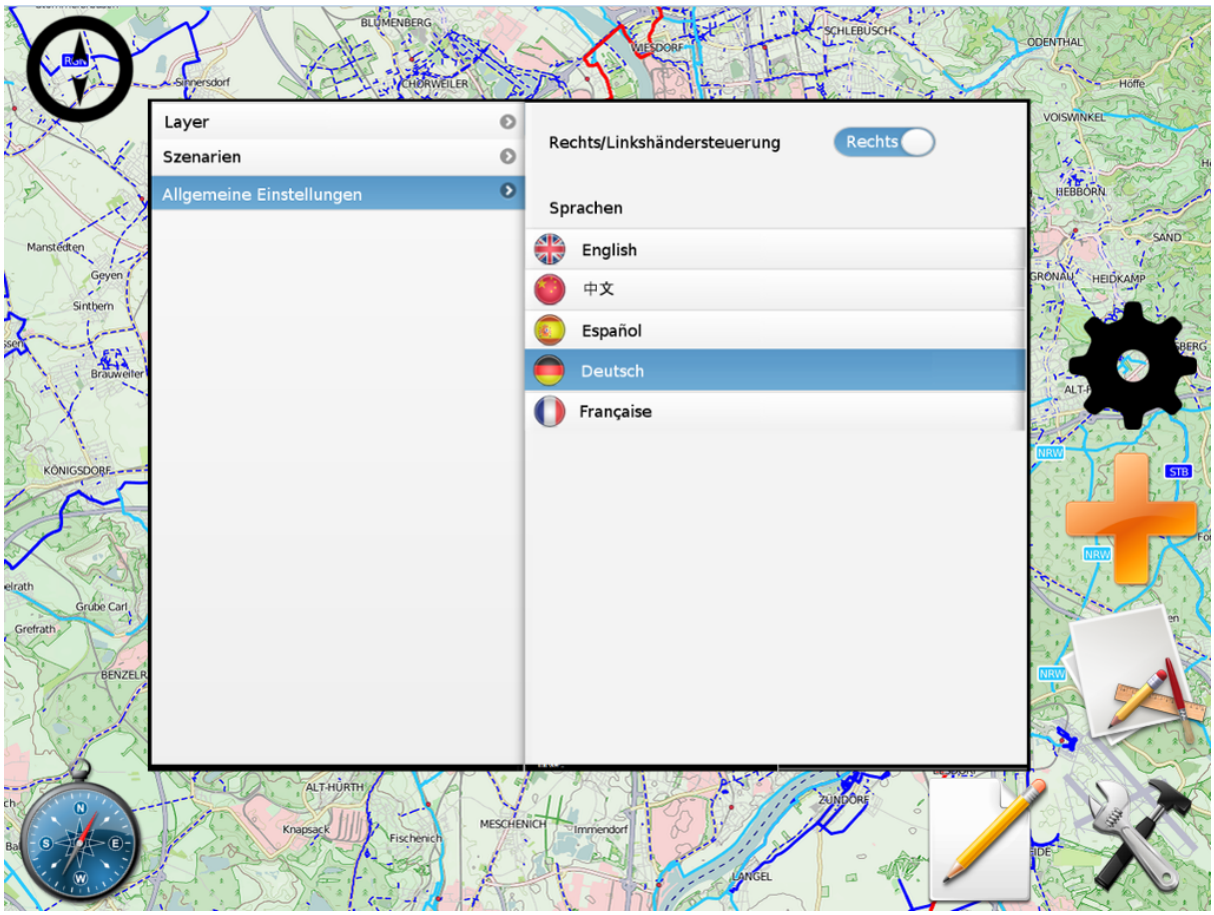


Abbildung 3.14: Einstellungs Menü

Jedes anklickbare Symbol, also z.B alle Buttons, aber auch alle Icons in Menüs, wie taktische Symbole, können in eine Schnellzugriffsleiste gezogen werden. Sobald der User eine “Zuggeste” ausführt erscheinen mittig zentriert unten drei Kästen in welche man die Symbole ablegen kann. Die dort abgelegten Symbole dienen als “Verknüpfung” zu der Funktionalität der normalen Symbole. D.h wenn man z.B den Zeichenbutton in die Schnellzugriffsleiste schiebt, so öffnet dieser bei Klick wie gewohnt das Zeichenmenü aber auch das Toolsmenü, in welchem der Zeichenbutton ein Element ist. Zieht man ein Element über eine Schnellzugriffsleistenbox welche bereits besetzt ist, so wird das letzte Element verdrängt.

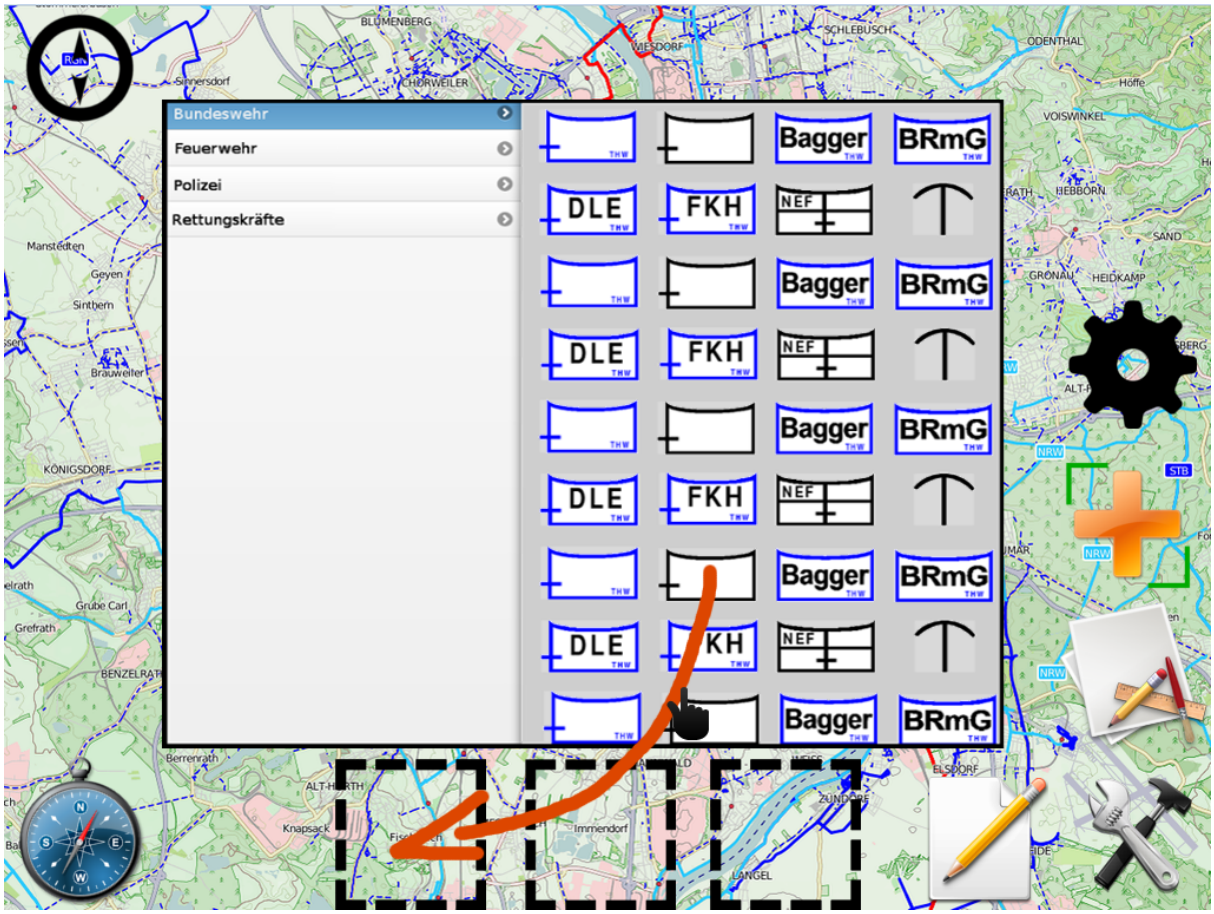


Abbildung 3.15: Menüanpassung

Sollte die Verbindung zum Server abbrechen wird dies über ein entsprechendes Icon im oberen rechten Eck angezeigt.

3.3 Umsetzung im Code

Die graphische Repräsentation des Programmes besteht wie bereits erwähnt aus zwei Komponenten, der Karte (Map), den darüber gelegten Buttons (HudView) und dem initialen Login Bildschirm. Diese drei sind unabhängige Komponenten. Wir wollen in diesem Kapitel speziell die Struktur der letzten beiden näher beleuchten. Alle Anfragen für die graphische Anzeige gehen über das Interface View. Gwt View steuert den Zusammenhang aller Komponenten speziell den Tausch der Login View durch die HudView (mit Map) und andersherum. Alle graphischen Elemente die auf dem Hud zu sehen sind werden von Plugins realisiert. Dafür zuständig ist das ButtonPlugin, welches es ermöglicht Buttons auf der Oberfläche hinzuzufügen ohne sich mit dem dahinter verwendeten Framework beschäftigen zu müssen. Mächtiger ist das GwtPlugin welches einen kompletten Zugriff auf alle Gestaltungsmöglichkeiten des Frameworks erlaubt, daher aber auch nur mit Kentniss des Frameworks und bei in kaufnahme von Portabilitätseinschränkungen genutzt werden kann. Während das GwtPlugin seine graphische Repräsentanz also direkt in Gwt Widgets findet, wird das ButtonPlugin über eine eigene, über ein Interface frameworkunabhängige, Klasse visualisiert. Mehrere Menübuttons, im Code durch die Klasse HudIcon mit der Implementation GwtHudIcon realisiert, werden über die Klasse MenuGroup zu logischen Gruppen zusammengefügt. Jeder Button und jedes Menü kann dabei über einen “Tag” indentifiziert werden. Dies erlaubt es einfach in weiteren ButtonPlugins auf zuvor instanzierte und in der View gehaltene Menü Gruppen oder Buttons zuzugreifen. Funktionen wie getPositionOfIconRelativToIcon erlauben es über eine Richtungsanagabe einfach Icons neben, über oder unter anderen Icons zu platzieren.

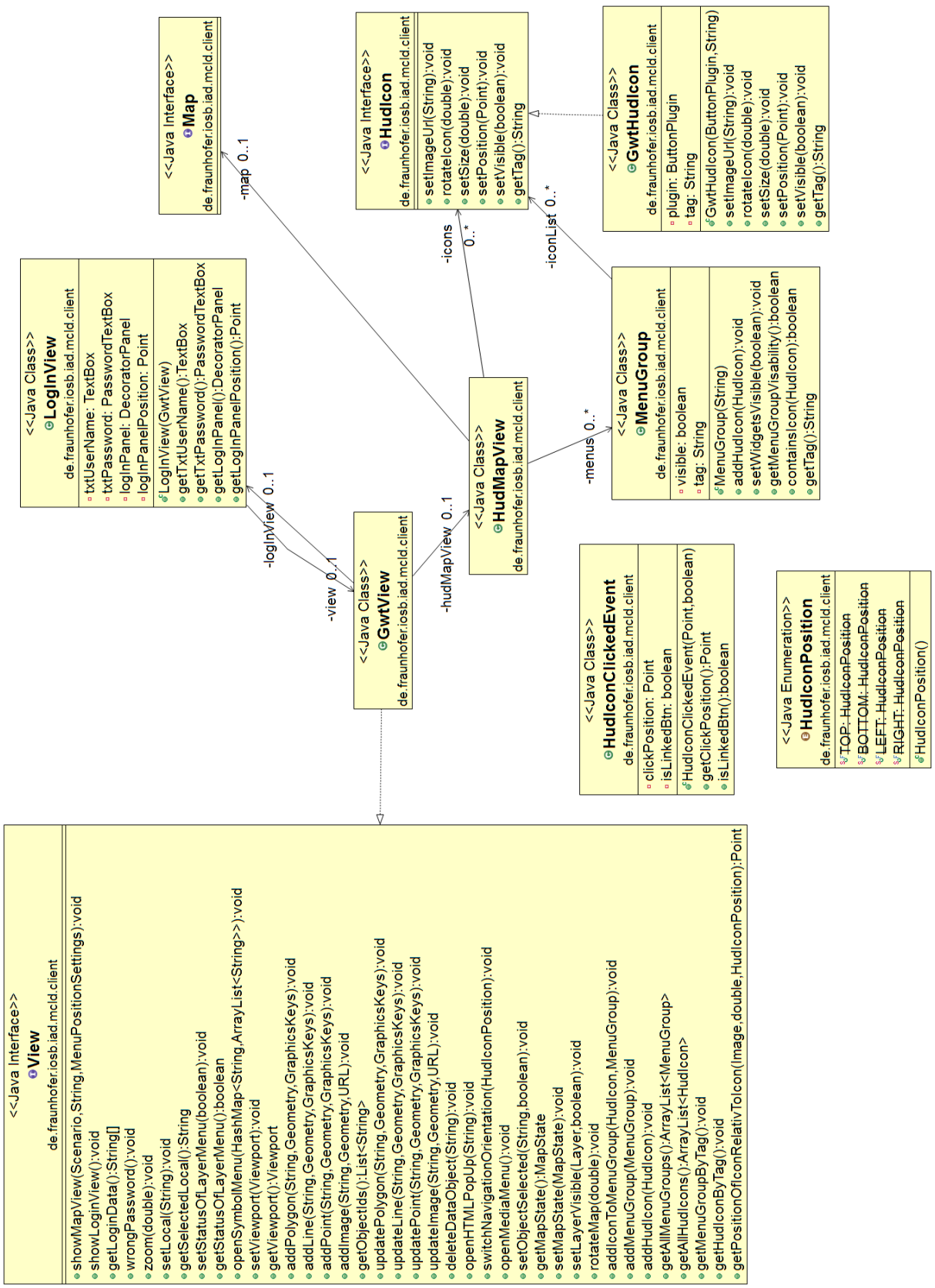


Abbildung 3.16: UML Darstellung der für die View (ohne Map) wichtigen Komponenten.

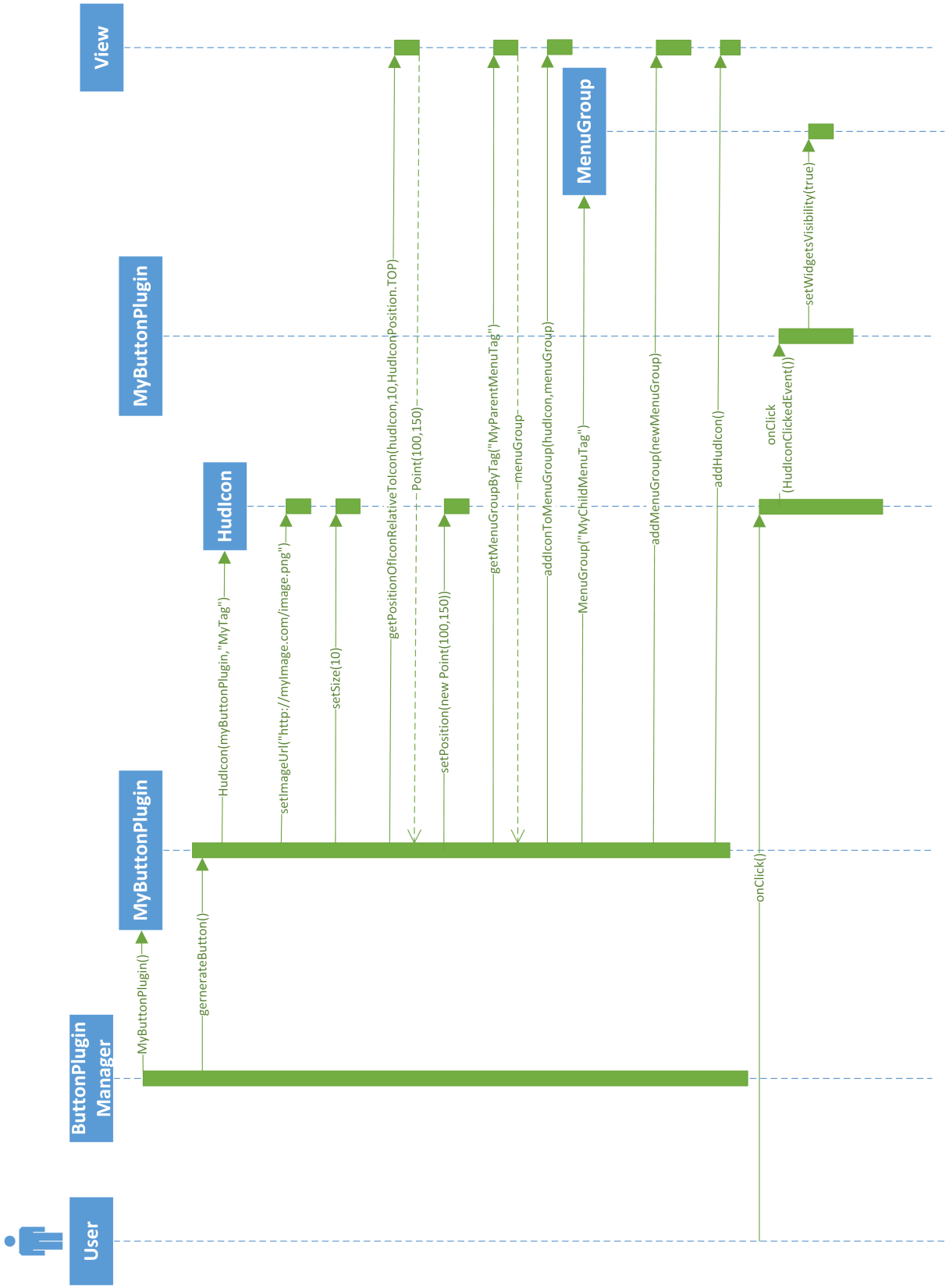
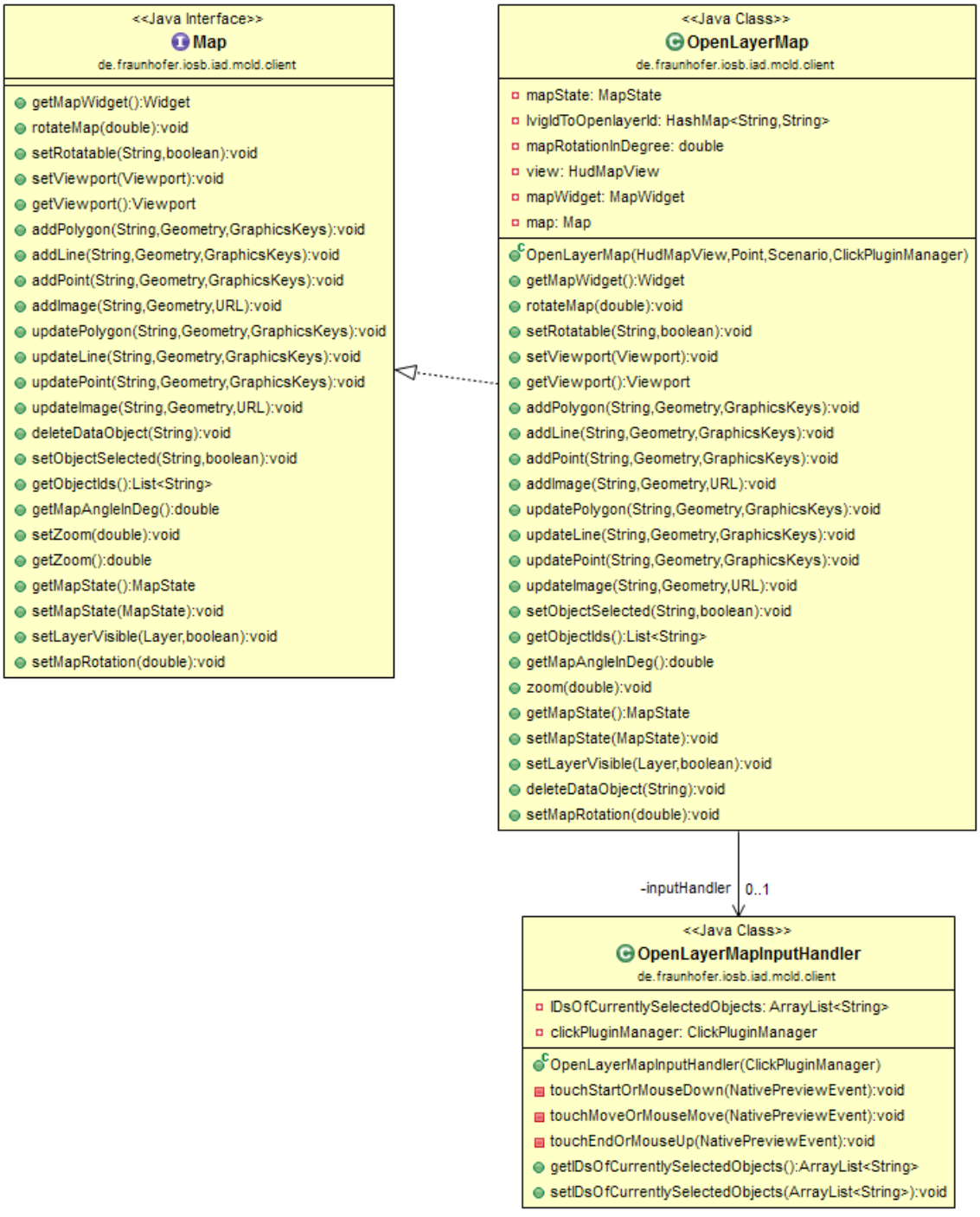


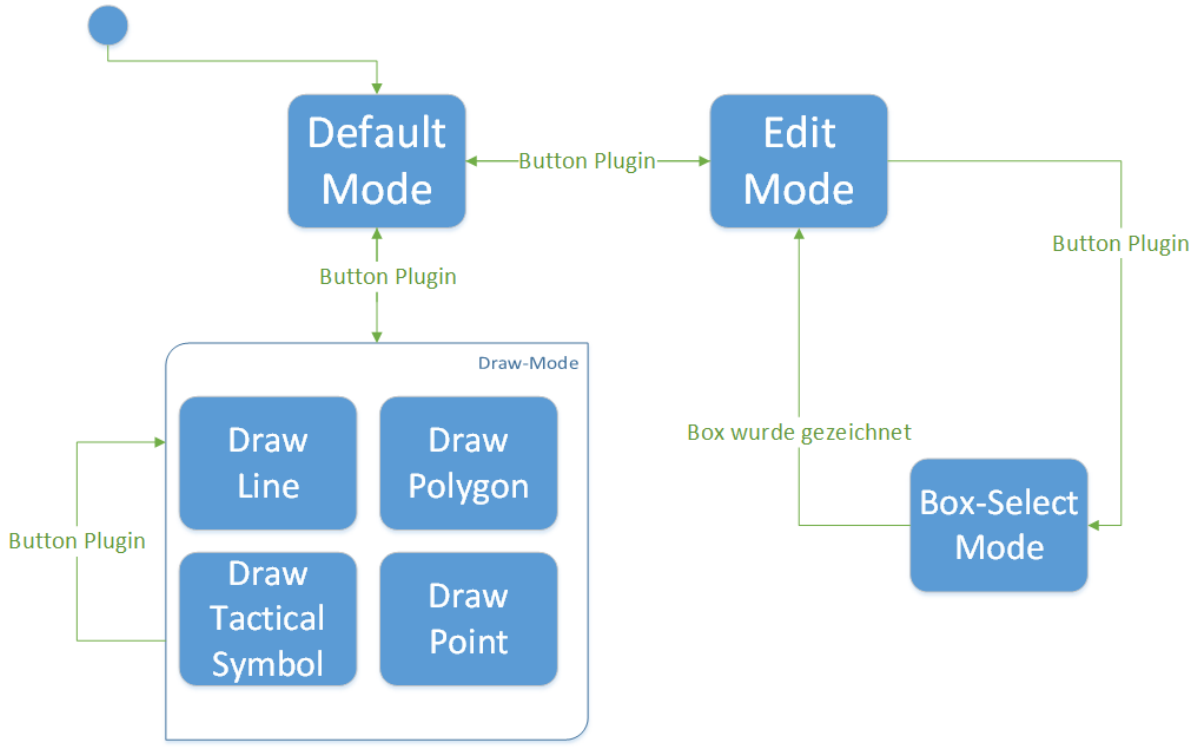
Abbildung 3.17: Beispielhafte Initialisierung und Nutzung eines ButtonPlugins. Das Plugin fügt seinen Button zu einer Menü Gruppe hinzu und legt eine Neue an welche er beim Click auf seinen Button anzeigt. Ein praktisches Beispiel hierfür wäre z.B der Zeichenbutton. Dieser gehört zu “Tools’menügruppe” und steuert die “Zeichenoptionenmenügruppe”. Der Button gehört also zu einer Menügruppe, zugleich stuert er auch eine Untermenügruppe

3.4 Map

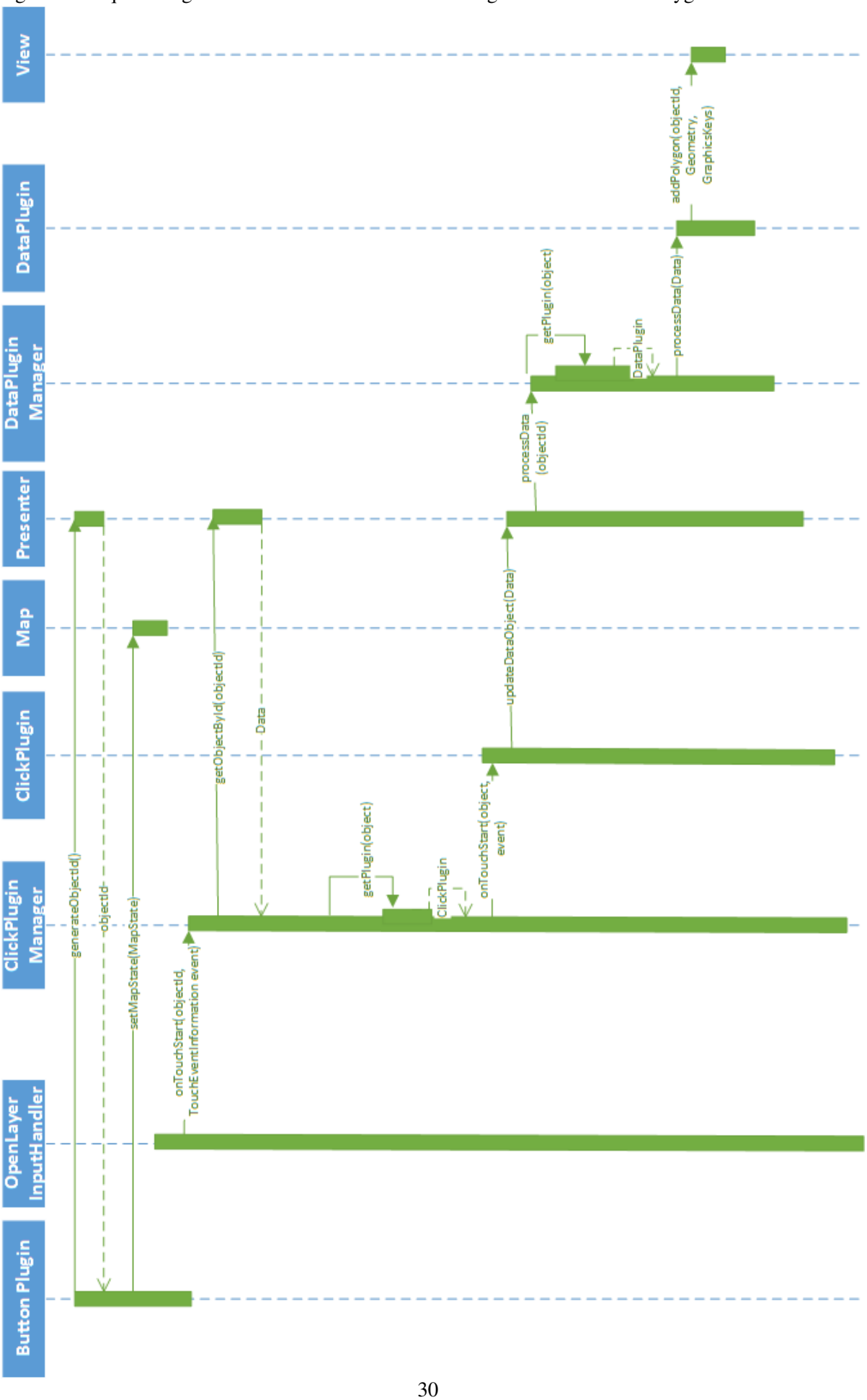
Im Folgenden ist das Klassendiagramm des Map-Interfaces zu sehen, sowie dessen OpenLayer-Implementation. Durch diesen Entwurf ist es auch möglich, eine andere Karten-Bibliothek zu implementieren und dadurch die OpenLayerMap auszutauschen. Im Konstruktor der Map wird das Szenario übergeben und dadurch auch die damit verknüpften Layer. Die Funktion 'setLayerVisible(Layer, boolean)' ist dafür zuständig einzelne Layer ein- oder auszublenden. Alle weiteren Methoden dürften selbsterklärend sein, die Methode 'setRotatable(String,boolean)' dient dazu, dass bestimmte Objekte (zum Beispiel Taktische Symbole) bei einer Kartenrotation nicht mitgedreht werden. Alle IDs bestehen aus einem Präfix die den Layer angibt oder aber bestimmten "recognition strings", die Daten auf der Map angeben die keine direkte Representation im Backend als Data Objekt haben z.B ein Symbol im Layer. Die Suffix ist dann eine eindeutige ID wie sie das Datum auch im Backend besitzt.



Das Enum MapState ist dazu da, um zu speichern in welchem Zustand sich die OpenLayerMap gerade befindet. Dies dient dazu, dass bei einer Wisch-Geste unterschieden werden kann, ob die Karte oder ein ausgewähltes Objekt bewegt werden soll. Fast alle Zustände werden über Button-Plugins geändert, zum Beispiel in dem durch das Drücken des Zeichnen-Buttons der Draw-Line-Modus aktiviert wird.



Um das Zusammenspiel zwischen Map, Plugins, View und Presenter besser zu verdeutlichen, zeigt folgendes Sequenzdiagramm den Ablauf beim Hinzufügen eines neuen Polygons auf der Karte.

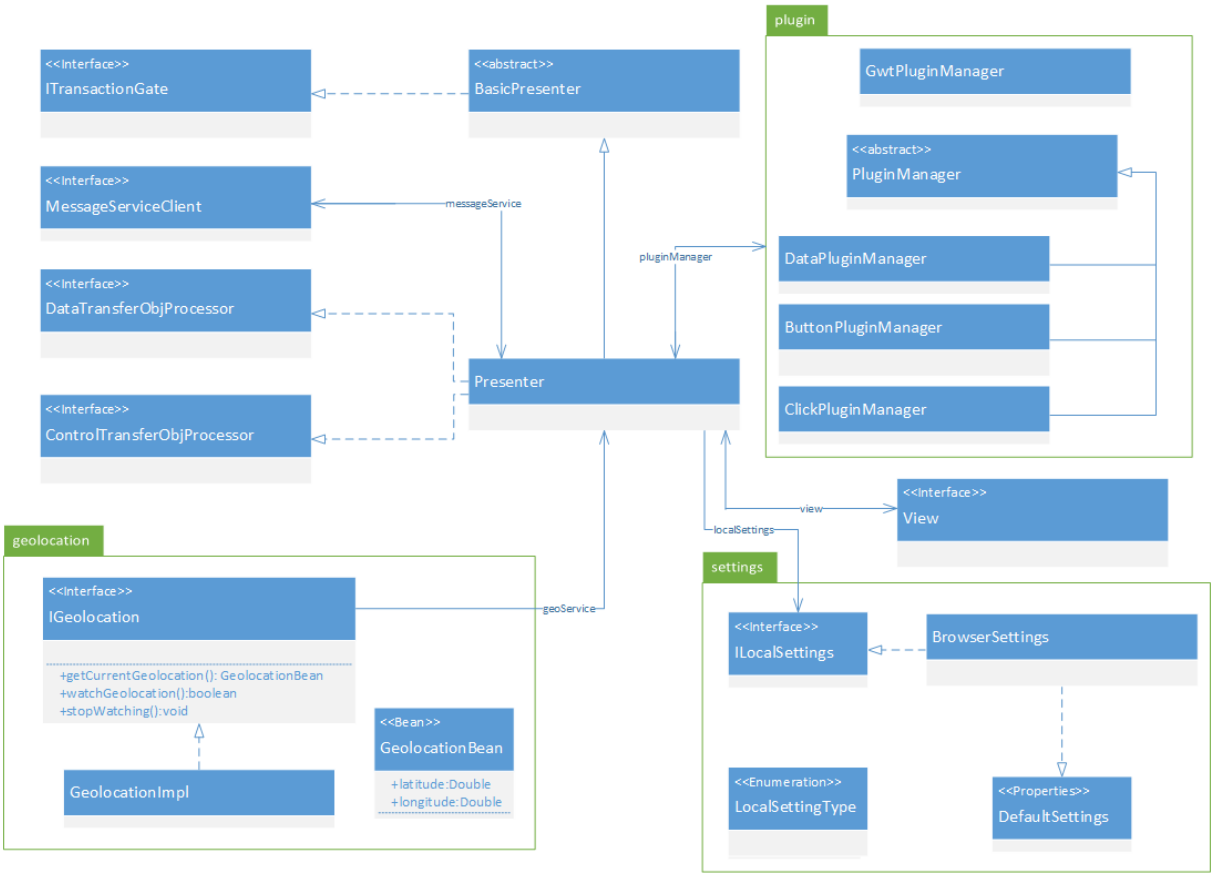


4 Presenter

Der Presenter ist die Verbindung zwischen Model und View. Die Hauptaufgabe des Presenters ist es, logische Abläufe zwischen Model und View zu steuern und die eigentliche Funktionalität der View zu implementieren.
Aufgrund des Plugin-Entwurfes wird viel Logik vom Presenter direkt in das Plugin übertragen, wodurch die Hauptaufgabe des Presenters dahingehend verändert wird, dass dieser den grundlegenden Ablauf steuert und Objekte sowie Funktionalität für die Plugins vorhält. So ist es für das Gesamtsystem ein leichtes neue DatenObjekte zu integrieren ohne am Presenter etwas ändern zu müssen.

Wird beispielsweise ein neues DatenObjekt angelegt, so ist später im Presenter nichts zu tun. Er wird die Steuerung nur grundlegend leiten und den Rest dem Plugin selbst überlassen. Klickt der Benutzer nun auf das neue DatenObjekt so wird der Presenter wie auch bei den anderen Objekten dieses Objekt dem ClickPluginManager übergeben, welcher wiederum versucht das Objekt mit dem richtigen ClickPlugin zu verarbeiten.

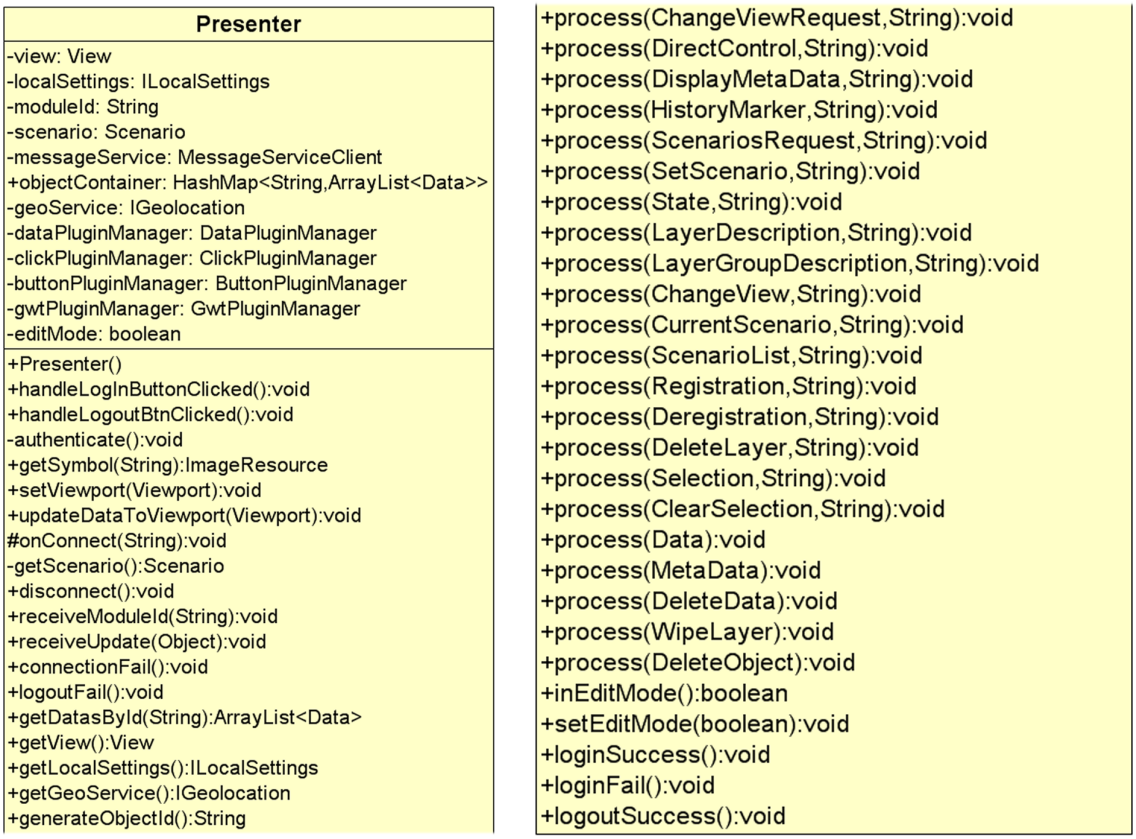
Das folgende Klassendiagramm gibt einen Eindruck von der Vernetzung des Presenters. Um Abläufe zu steuern ist der Zugriff auf einzelne Komponenten essentiell.



BasicPresenter
Da man nicht ausschließen kann, dass es irgendwann einmal noch weitere Presenter geben könnte, wurde eine abstrakte Oberklasse dazu entworfen um Grundfunktionalität, die jeder Presenter benötigt, wie die Kommunikation mit dem Backend, zu extrahieren und in eine Oberklasse auszulagern.
Der Presenter implementiert die Schnittstelle ITransactionGate worüber er vom Server durch ein Default-Callback Nachrichten über Updates oder aufgetretene Verbindungsprobleme erhält.

Über den MessageServiceClient kann der Presenter Objekte vom Client aus zum Backend übermitteln und erhält dann über das ITransactionGate wiederum die Updates. Um den Client genau zu identifizieren erhält dieser beim ersten Kontakt eine moduleId, die er bei der Kommunikation mitgeben muss.

Client Data Store
Um verwendete Objekte der View aufzubewaren und später wiederzufinden verwendet der Presenter eine Map worin er eine ID als String auf eine Liste von passenden Objekten mappt. Wichtig zu beachten ist hierbei, dass nicht nur ein Objekt genau identifiziert werden muss, sondern auch ein einzelner State. Ein State ist ebenfalls ein Objekt vom gleichen Type mit der gleichen ID hat jedoch einen anderen Index bzw. ein anderes Datum.
Ein Plugin hat somit die Möglichkeit anhand der ID eine Liste von Objekten zu bekommen und kann dann selbst entscheiden, ob es nur den aktuellsten Wert nutzt oder eine Liste von Werten.



Empfangen eines Data-Objektes vom Server
Wird etwas vom Server angefordert, so kann dieser über das ITransactionGate beim Presenter die Methode receiveUpdate(Object) aufrufen.

Im folgenden ist in einem Sequenzdiagramm dargestellt, wie ein Data-Objekt vom Server verarbeitet wird.

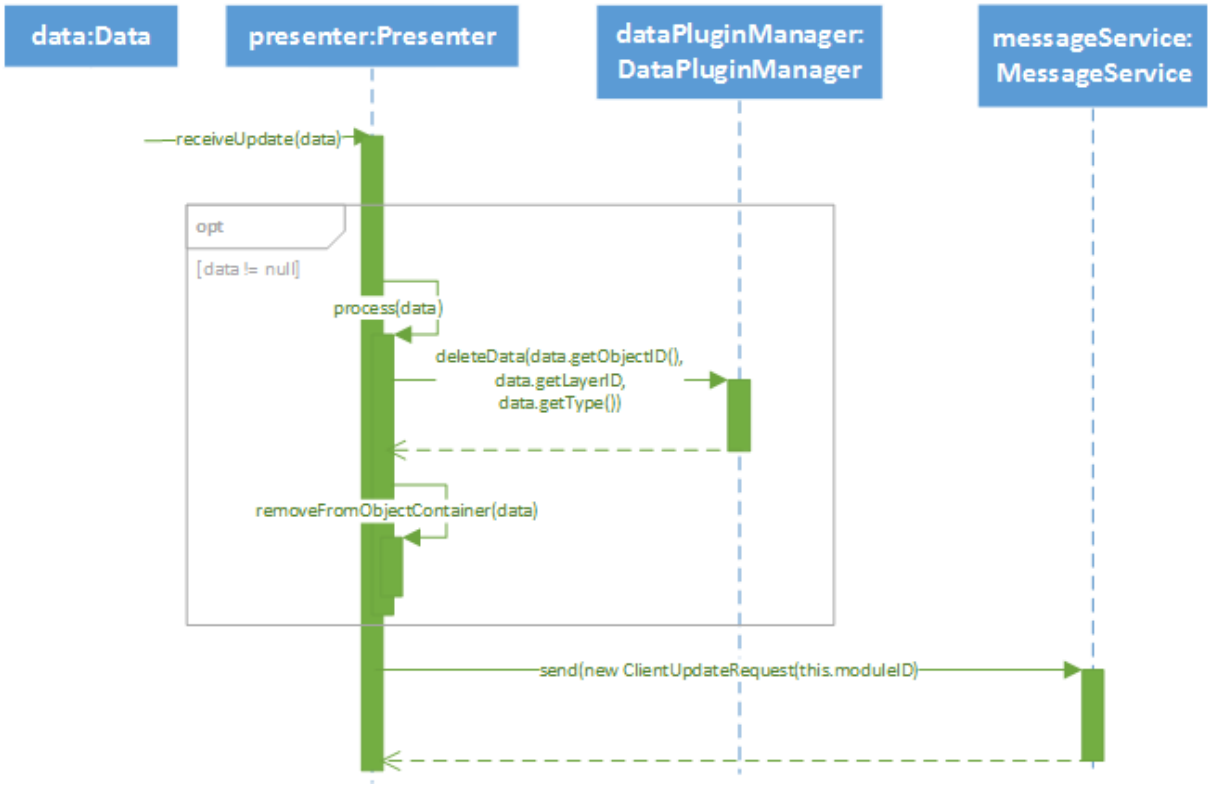


Abbildung 4.1: Sequenzdiagramm beim Empfangen eines Data Objektes

Empfangen eines DeleteData-Objektes vom Server
Kommt vom Server ein DeleteData-Objekt so kann aufgrund des Process-Pattern entsprechend darauf reagiert werden.

Beim Löschen ist es wichtig, dass der Presenter das DatenObjekt solange im Client Data Store vorhält, bis das DataPlugin das Objekt verarbeitet hat.

Der Presenter führt folgendes Sequenzdiagramm aus.

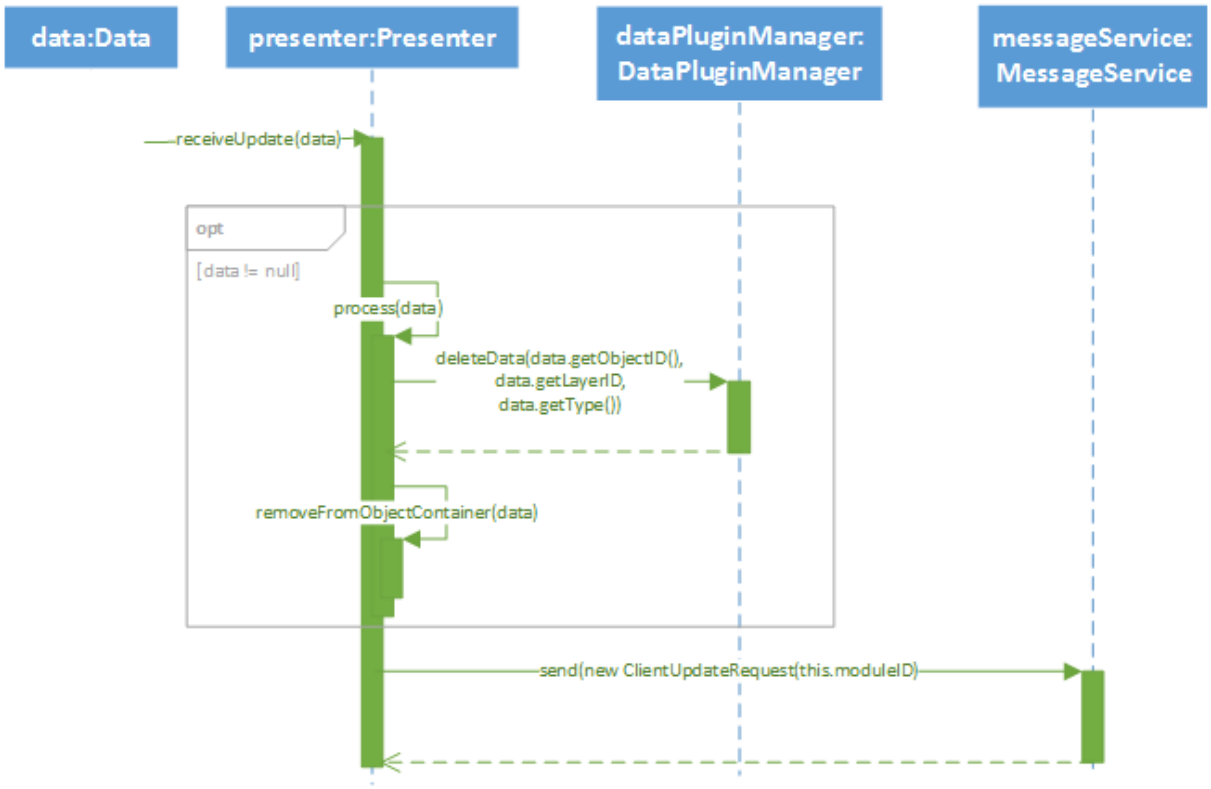


Abbildung 4.2: Sequenzdiagramm beim Empfangen eines DeleteData Objektes

Nachdem ein Objekt empfangen wurde muss der Presenter das nächste Update aktiv anfordern, um die Kommunikationskette nicht zu unterbrechen. Hierzu bietet der MessageServiceClient die send-Methode an, die der Presenter entsprechend aufrufen kann.

5 Plugins

Plugins sind ein essentieller Teil unseres Entwurfs, über diese wird das Anzeigen von Daten, Hinzufügen von Buttons und Ausführen von Touch- und Maus-Gesten auf der Karte koordiniert.
Hierzu gibt es drei grundlegende Plugin Typen für welche es jeweils ein Interface gibt.
Zusätzlich gibt es noch ein mächtigeren Plugin Typ, das GwtPlugin, welches alle Gwt Funktionalitäten unterstüzt.

Diese Initialisierung der Plugins ist aufgrund der Verwendung von Gwt und dessen Java zu Javascript Compiler nicht dynamisch zur Laufzeit möglich. Für jede dieser Plugin Typen gibt einen zugehörigen Plugin Manager, welcher die zu verwendente Plugins statisch initialisiert. Die Plugin Manager werden im Presenter initialisiert, die Manger initialisieren dann die Plugins.

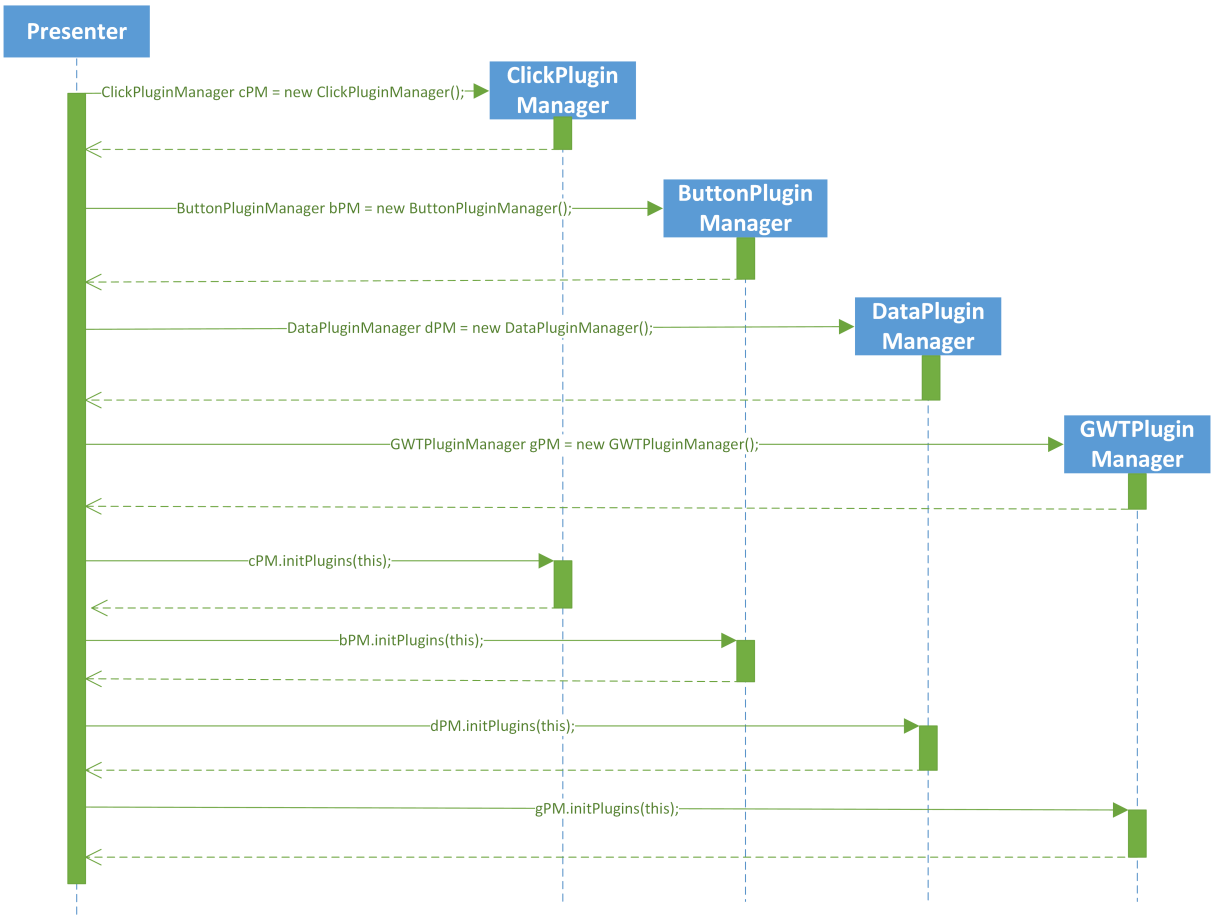


Abbildung 5.1: Sequenzdiagramm PluginManager- und Plugininitialisierung

5.1 ButtonPlugin

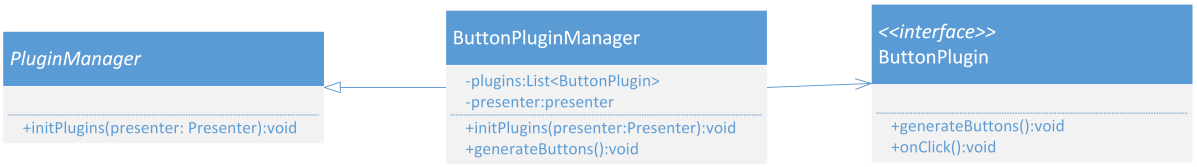


Abbildung 5.2: Klassendiagramm des ButtonPlugins

Das ButtonPlugin ermöglicht es Buttons mit Funktion zum Menü hinzuzufügen.
Im Konstruktor des implementierten ButtonPlugins muss eine Presenter Instanz übergeben werden können. Über diese kann dann direkt mit der View kommuniziert werden. Die Funktion *generateButton()* soll dann den Button über die durch den Presenter gegebene View-Schnittstelle generiert werden. Man kann dann Icon, Position und Menügruppe des zu erstellenden Buttons setzen.
In der Funktion *onClick()* ist nun die Funktion hinterlegt, welche aufgerufen werden soll wenn der Button angeklickt wird.
Über den ButtonPluginManager werden alle ButtonPlugins statisch initialisiert, indem die Liste der zu ladenen Plugins jeweils die *generateButton()* Funktion des Plugins aufruft.

5.2 ClickPlugin

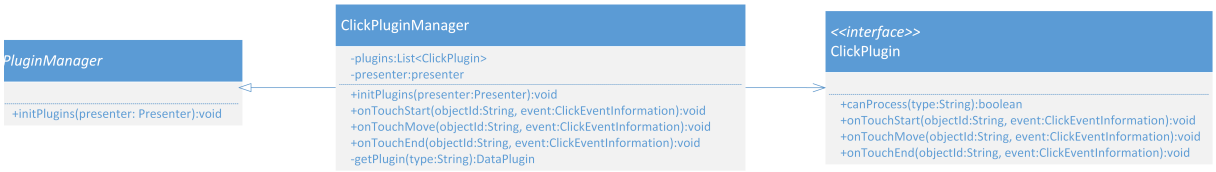


Abbildung 5.3: Klassendiagramm des ClickPlugins

Das ClickPlugin ist dafür zuständig zu entscheiden was bei einem Klick auf verschiedene Objekt Typen passieren soll. Bei Objekten auf dem WFS [54] Layer sollen zum Beispiel Metadaten angezeigt werden, während bei Klick auf ein Taktisches Symbol dieses z.B. verschoben oder gelöscht werden soll.

Um eine Schnittstelle zur View zu haben, muss auch im Konstruktor des implementierten ClickPlugins eine Presenter Instanz übergeben werden können. Um von verschiedenen Klickaktionen zu unterscheiden, besitzt das ClickPlugin drei verschiedene Funktionen. Die Funktion *onTouchStart()* legt fest was bei der Berührung des Objektes geschehen soll (dies entspricht auch dem Drücken der linken Maustaste). Die Funktion *onTouchMove()* legt fest was bei der Bewegung die nach dem Berühren geschieht geschehen soll (entspricht auch dem Bewegen eines Objektes bei gedrückter Maustaste). Und letztendlich noch die Funktion *onTouchEnd()* welche festlegt, was geschehen soll wenn man den Finger wieder vom Touchscreen nimmt (entspricht auch dem loslassen der linken Maustaste).

Die Funktion *canProcess()* wird vom ClickPluginManager benötigt und ist dafür da, die richtige Implementierung zum Plugin zu finden, da es für verschiedene Objekte unterschiedliche Objekttypen geben kann. Der ClickPluginManager entscheidet nun welche ClickPlugin Implementierung für die einzelnen Plugins zuständig ist und wie diese verarbeitet werden sollen.

5.3 DataPlugin

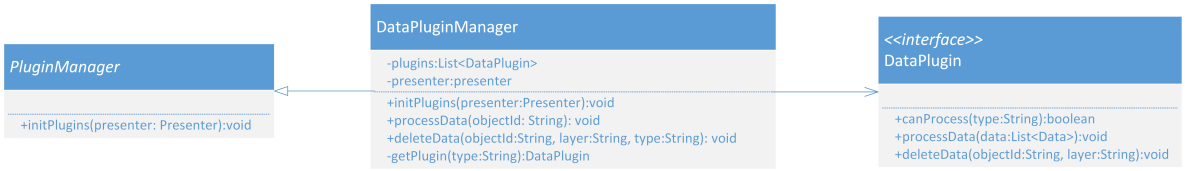


Abbildung 5.4: Klassendiagramm des DataPlugins

Das DataPlugin entscheidet wie welcher Datentyp verarbeitet werden soll.

Wie schon bei den anderen Plugintypen muss auch hier im Konstuktor des implementierten DataPlugins eine Presenter Instanz übergeben werden können. Wie genau ein Datentyp dann angezeigt werden soll, wird bei der Implementierung in der Funktion *processData()* definiert, während das Löschen eines Objektes in der Funktion *deleteData()* definiert ist.
Kommt beim Presenter ein UpdateData an so ruft er die Methode *processData()* im Manager auf. Mit der hierbei übergebenen ID kann der Manager das zugehörige Objekt über die Funktion *getObjectById())* beim Presenter erfragen. Existieren zu einer ID mehrere Daten für z.B verschiedene Zeitpunkte so werden diese in einer Liste gespeichert. Kommt nun ein DeleteObject vom Backend so wird ein Element aus der Liste gelöscht und diese neue verkleinerte Liste über *processData()* dargestellt. Das würde zum Beispiel bei einem GPS Track bedeuten das ein Wegpunkt verschwindet. Empfängt der Client DeleteData vom Backend wir das ganze Objekt über die Funktion *deleteData()* gelöscht.

Wie schon beim ClickPluginManger wird auch beim DataPluginManager die Funktion *canProcess()* benötigt um die richtige Implementierung zum Plugin zu finden.

5.4 GwtPlugin

Das GwtPlugin erlaubte es sämtliche Fähigkeiten von Gwt zu nutzen. Es initiallisiert seine Widgets bei der initialisierung des GwtPlugin. Bei aufruf von Render holt es sich das “RootPanel” der Gwt Applikation und zeichnet sich darauf worauf es im Browser dargestellt wird.

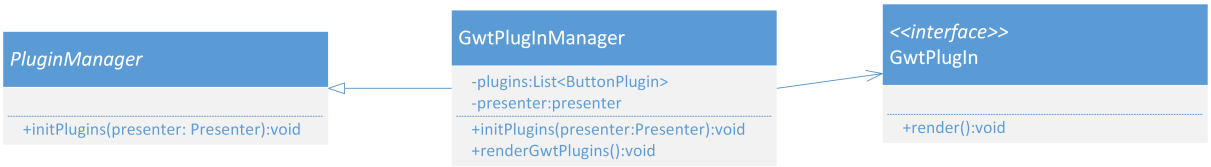


Abbildung 5.5: Klassendiagramm des GwtPlugins

6 Benutzereinstellungen

Im folgenden Abschnitt wird auf die Speicherung von Benutzereinstellungen eingegangen.

Interagiert ein Benutzer des öfteren oder über einen längeren Zeitraum mit dem System, wird er über kurz oder lang ein paar Einstellungen vornehemen, um die Oberfläche oder Abläufe zu optimieren.

Damit diese Benutzereinstellungen nicht jedesmal gemacht werden müssen, können diese über eine Schnittstelle persistiert werden.

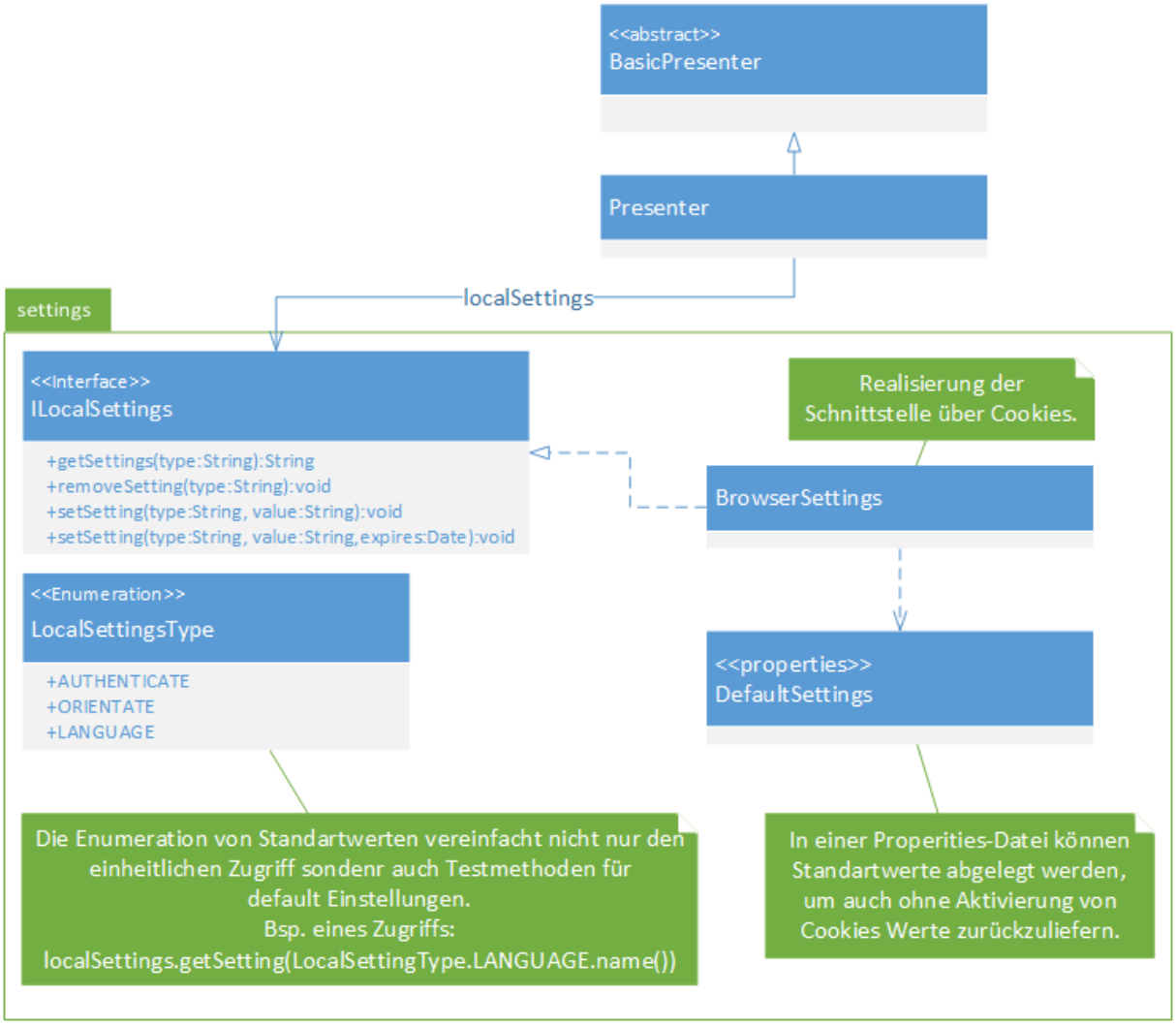
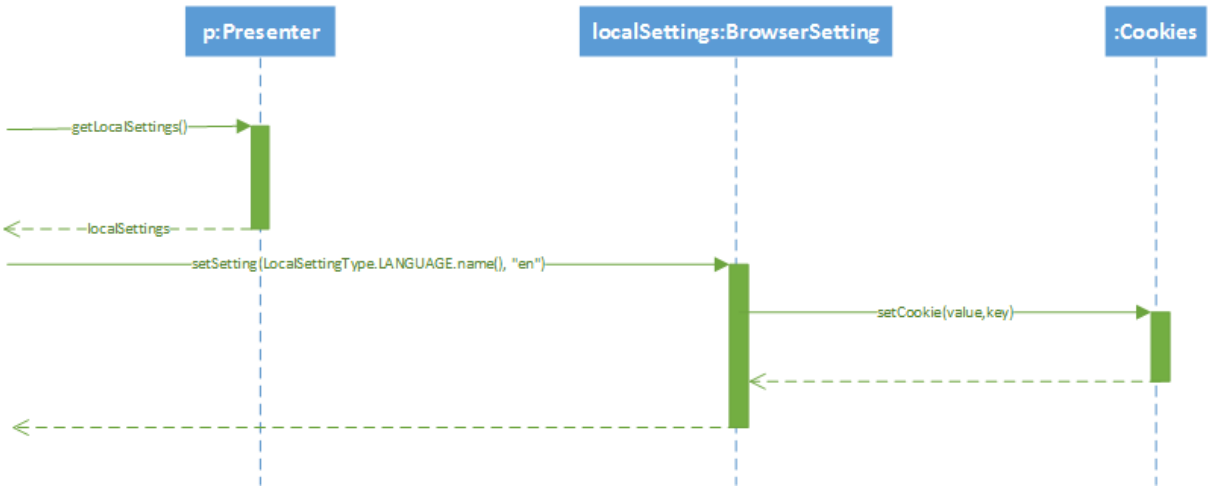
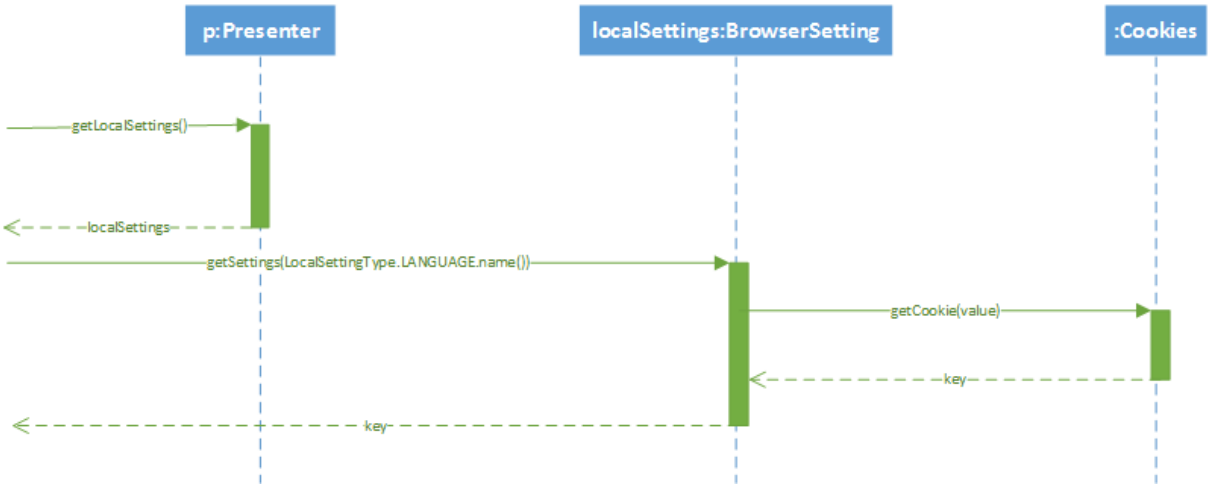


Abbildung 6.1: Klassendiagramm des setting-Packages

Benutzereinstellungen setzen



Benutzereinstellungen abfragen



7 Geolocation

Der Presenter bietet die Funktionalität die aktuelle Geoposition des mobilen Endgeräts herauszufinden, falls das mobile Endgerät dies unterstützt. Hierzu stellt der Presenter die Schnittstelle IGeolocation zur Verfügung die drei Möglichkeiten bereitstellt. Die Methode getCurrentGeolocation liefert ein GeolocationBean-Objekt womit man Breiten- und Längengrad auslesen kann. Um permanent Geokoordinaten abzufragen kann man watchGeolocation verwenden, womit Standortänderungen permanent wahrgenommen werden können. Die konkrete Implementierung der Schnittstelle GeolocationImpl greift hier auf die von Google bereitgestellte Geolocation-API zu. Im Falle der permanenten Verfolgung der Geoposition wird ein Callback generiert, der bei Veränderung der Geoposition die neue Position mitteilt und anschließend einen neuen Callback generiert. Somit können Standortveränderung permanent wahrgenommen werden. Die Methode stopWatching beendet diese Funktion.

Das folgende Klassendiagramm zeigt die Anbindung an den Presenter.

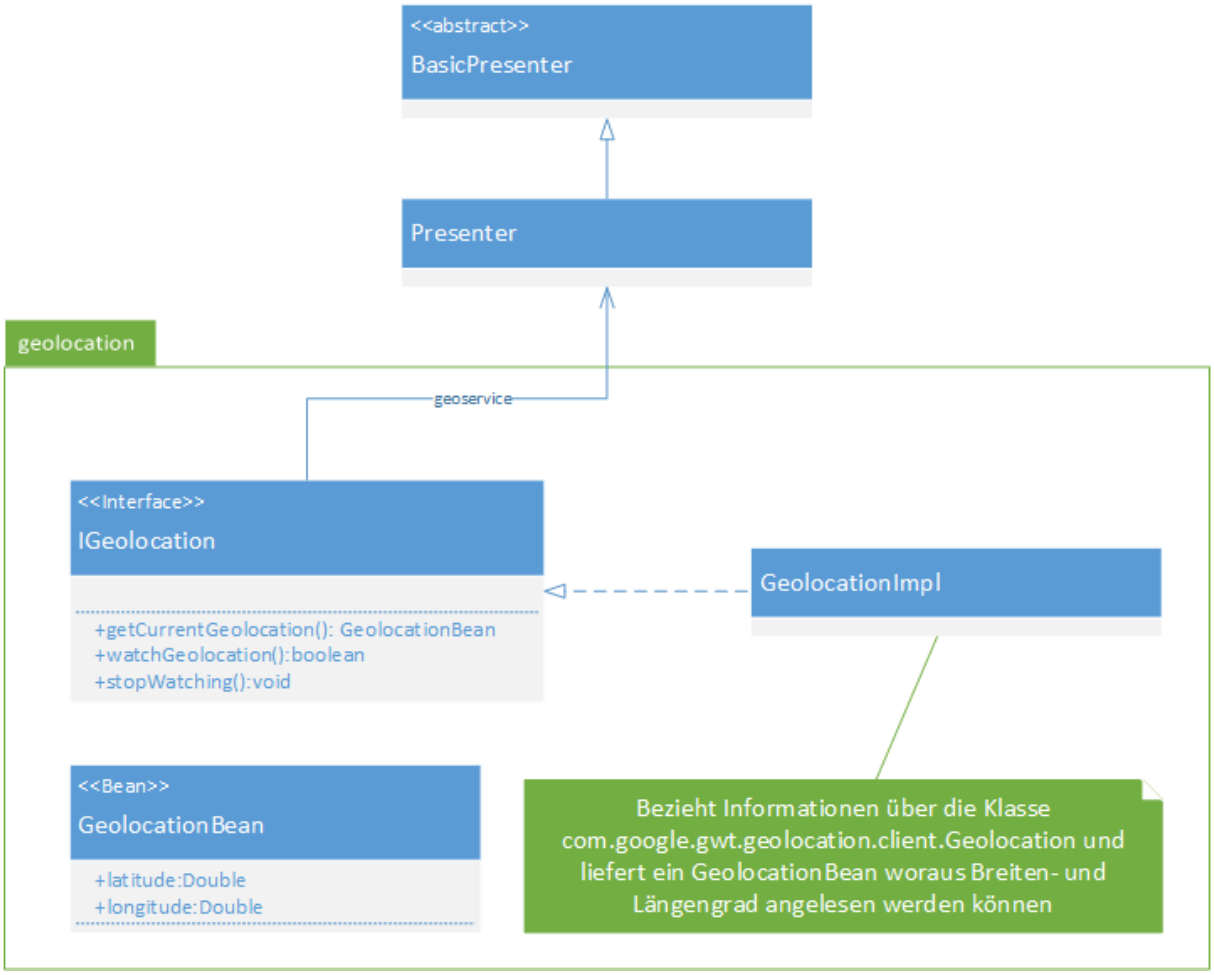
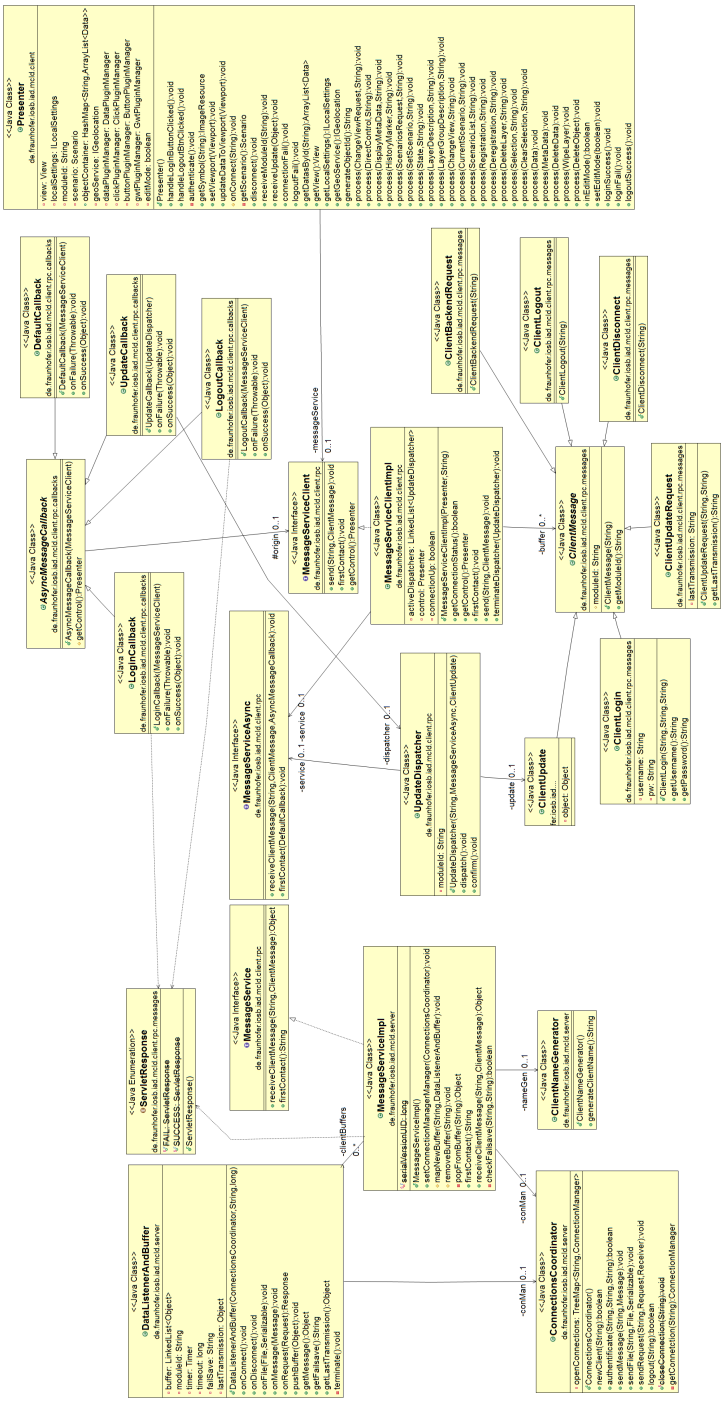


Abbildung 7.1: Klassendiagramm des geolocation-Packages

8 Servlet und Servlet-Client Kommunikation

8.1 Struktur



8.2 Anmeldung beim Backend

Zur Kommunikation zwischen Servlet und Client werden die RPC (remote procedure call) Funktionalitäten von Gwt genutzt. Die Anmeldung an das Backend erfolgt für jedes Gerät einzeln. Da ein RPC Webcall für jedes Gerät an das gleiche Servlet gesendet wird, werden an dieser Stelle die Geräte voneinader über die ModulID getrennt. Mit Starten der Anwendung, wird eine neue Instanz der clientseitigen Messageservice Implementation instanziiert. Diese versucht sofort eine Verbindung zum Servlet herzustellen. Schlägt dies fehl, entweder aufgrund eines Fehlers auf Serverseite, schlechter Verbindung oder andersweitiger Nichterreichbarkeit des Servers bzw. des Servlets, wird bis zum Erfolg oder Abbruch erneut versucht eine Verbindung herzustellen. Kommt eine Verbindung zustande, so generiert das Servlet einen eindeutigen Identifikator für das neue Gerät. Dieser wird an den ConnectionsCoordinator weitergereicht, der daraufhin versucht eine Verbindung zum Backend über einen neuen ConnectionManager der MiddlewarTools aufzubauen. Außerdem wird ein neuer DataListenerAndBuffer generiert um die eingehenden Objekte des Backends entgegenzunehmen und zu puffern bis sie an den zugehörigen Client weitergereicht werden. Der Identifikator wird an das Gerät zurückgeschickt, von diesen gespeichert und für zukünftige Anfragen an den Server genutzt.

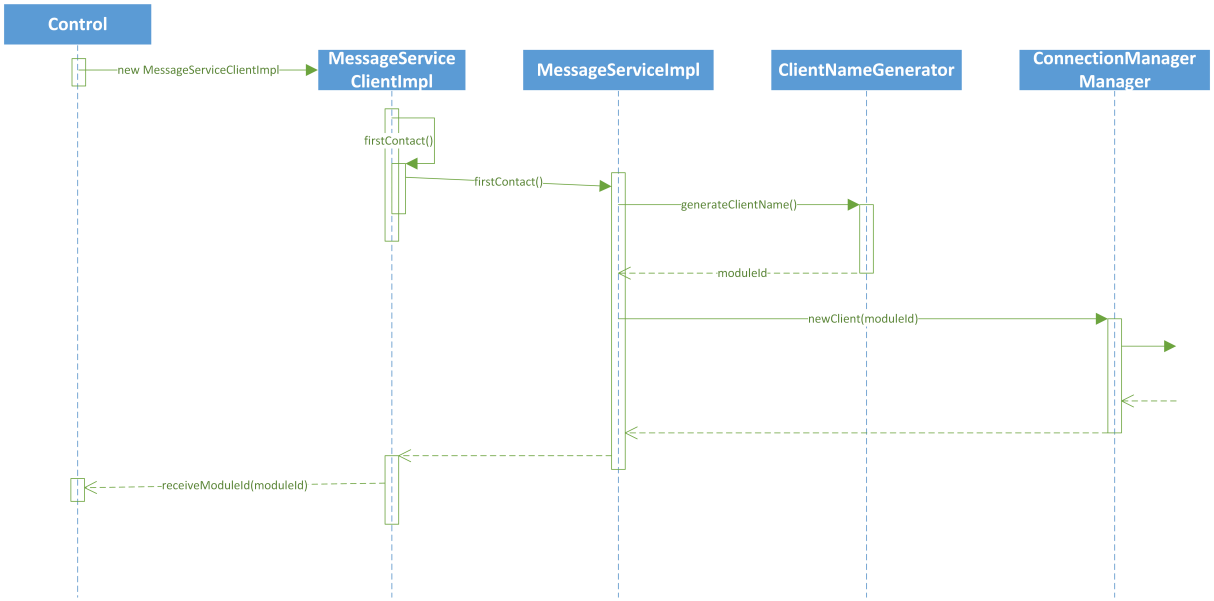


Abbildung 8.1: FirstContact von der Client Seite aus

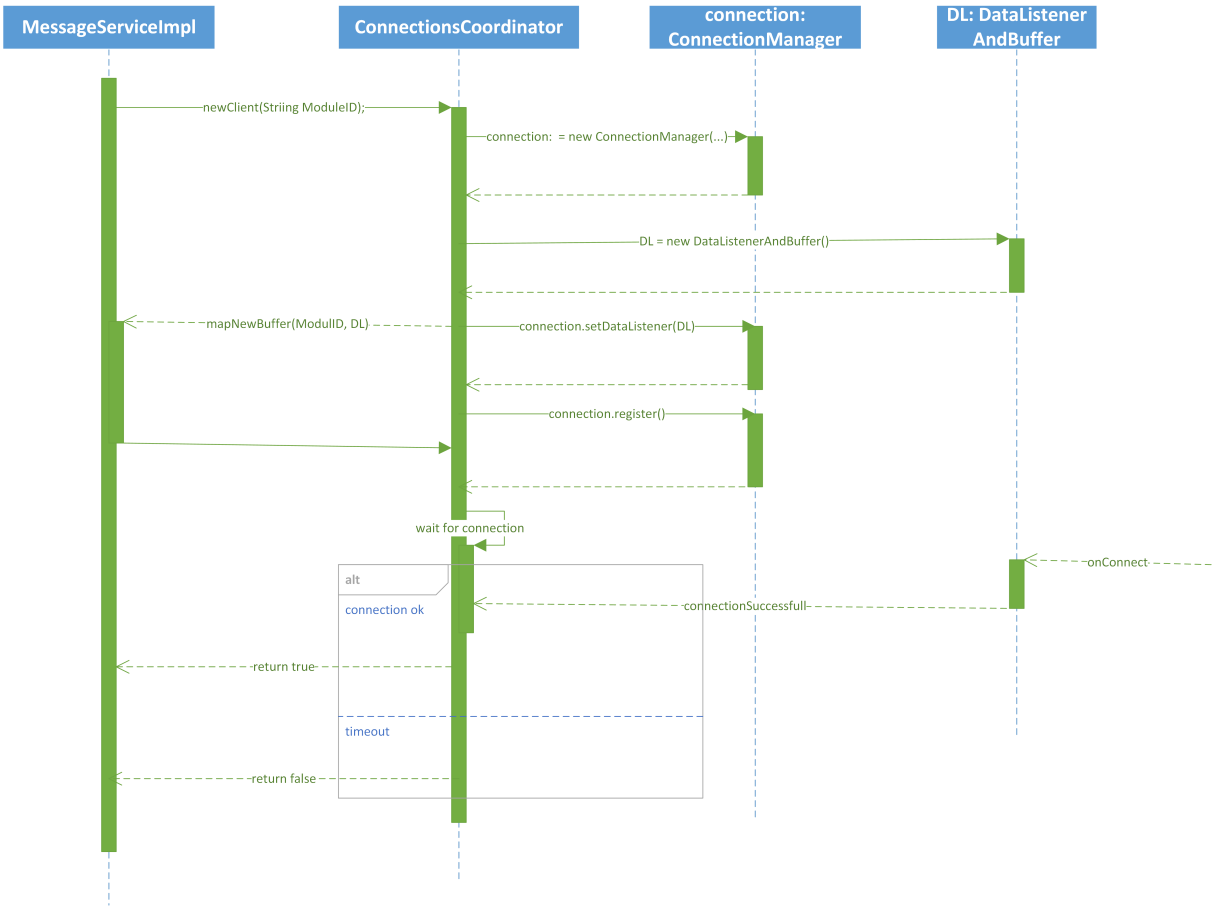


Abbildung 8.2: FirstContact von der Client Seite aus

8.3 Anmeldung des Users

Nach erfolgreichem Ablauf des ersten Kontaktes ist eine Anmeldung beim Server durch den Nutzer möglich. Hierzu werden Benutzername, Passwort und und zuvor erhaltener Identifikator an das Servlet geschickt. Im Falle eines falschen Passwortes und/oder Benutzernamens oder eines Verbindungsfehlers wird der Benutzer aufgefordert seine Daten erneut abzuschicken. Sind Benutzername und Passwort gültig, so stellt der Server das StandardSzenario im Buffer für den entsprechenden Clients zur Verfügung und dieser kann beginnen Updates von Server zu verlangen und zu erhalten.

8.4 Message vom Client

Sendet der Client ein Update an den Server, so stellt er ein entsprechendes Update-Objekt zur Verfügung. Für dieses wird von der Clientseitigen Implementation des Message Services ein Dispatcher erstellt. Dieser sendet das Object an den Server. Ist die Übertragung ein Erfolg, so beendet er sich, ist dies nicht der Fall, sendet er das Objekt erneut. Die Anzahl an gleichezeitig aktiven Dispatchern ist hierbei zur Compilezeit einstellbar, inaktive Dispatcher werden in einer Warteschlange gehalten. Das Servlet gibt das erhaltene Objekt dann nach entnehmen aus der RPC-Übertragunsklasse an den Server weiter.

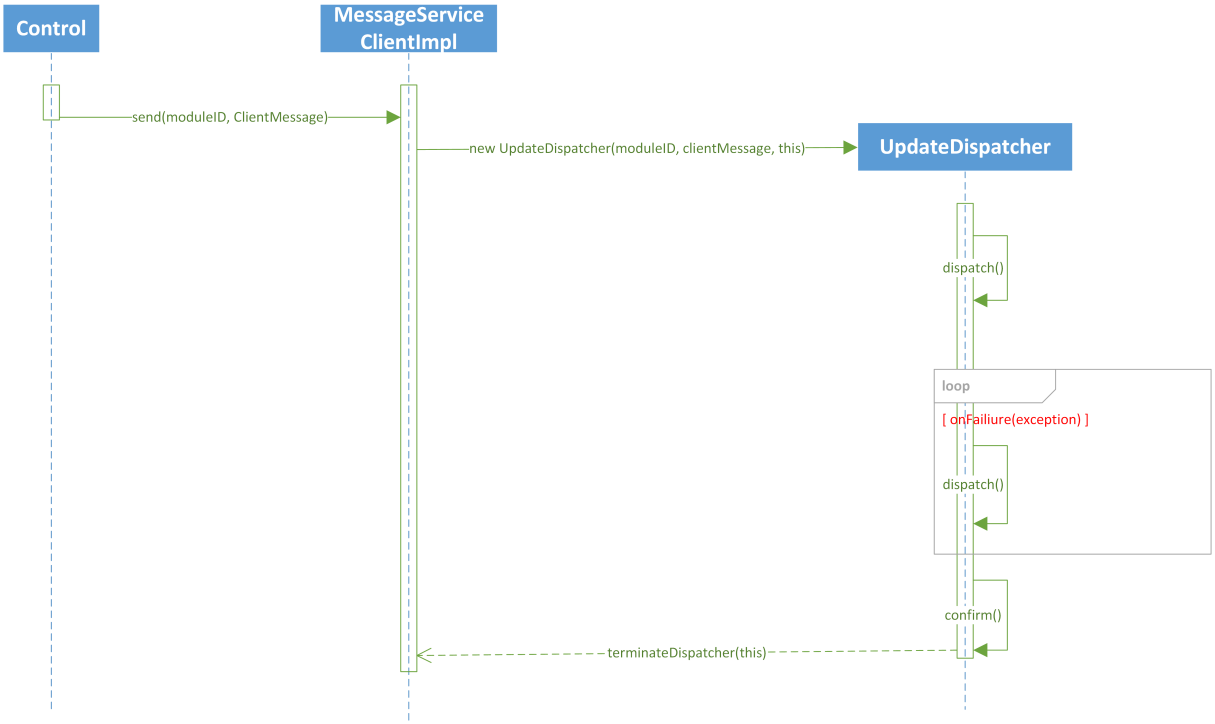


Abbildung 8.3: Ein Update vom Client aus

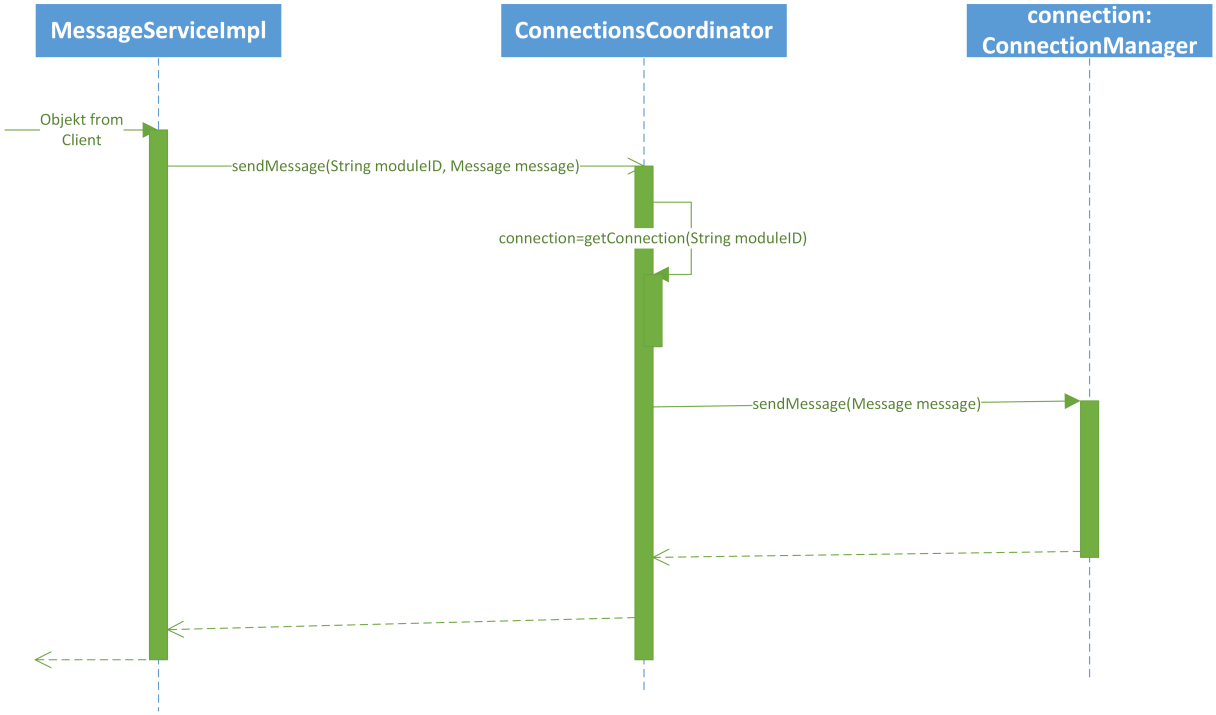


Abbildung 8.4: Ein Update richtung Server

8.5 Message vom Server

Bei einer Webanwendung, wie sie im Falle von Gwt vorliegen, steht kein Server Push zur Verfügung. Daher sendet der Client regelmäßig Anfragen den Server. Im Gegensatz zum Polling geschieht dies jedoch nicht im Sekundentakt, sondern im Normalfall zwei bis drei mal pro minute. Der Server hat dann die Möglichkeit über eine mitgegebene Rückrufadresse (den sog. Callback) das Update auch noch lange (30 sekunden) nach der Anfrage zu ßenden". Verlangt der Client ein Update vom Server, so sendet er ein ensprechendes Request-Objekt an das Servlet. Im Falle einer fehlgeschlagenen Verbindung sendet der Client nach einer kurzen Wartezeit eine erneute Anfrage. Kommt diese beim Sevlet an, so prüft das Servlet zuerst die in diesem Request-Objekt enthaltene Identifikationsnummer des letzten vom Client empfangenen Updates gegen die im dazugehörigen Buffer gespeicherte Identifikationsnummer der letzten Übertragung. Sind diese nicht identisch, so ist die letzte Übertragung des Servers zum Client nicht korrekt vollzogen worden. In diesem Falle sendet das Servlet erneut das zuletzt übertragene Object an den Client. Sind beide Identifikatoren hingegen identisch, so überprüft das Servlet den zum Client gehörenden Buffer auf noch nicht gesendete Inhalte. Sind diese vorhanden, so wird das erste Object in der Warteschlange (das älteste) an den Client übergeben. Sind keine zu sendenden Objecte vorhanden, so wartet das Servlet bis zu 30 sekunden auf ein neues Update. Wird in dieser Zeit kein neues Update bereitgestellt, so erfolgt keine Übertragung zum Client. Spätestens in diesem Moment findet ein Timeout im Asynchronen Callback des Clients statt und dieser sendet eine neue Anfrage. Wird in der entsprechenden Zeit ein Update bereit, so wird dieses an den Client übergeben. Sobald der Client ein Update oder ein Timeout erhält startet er eine erneute Anfrage.

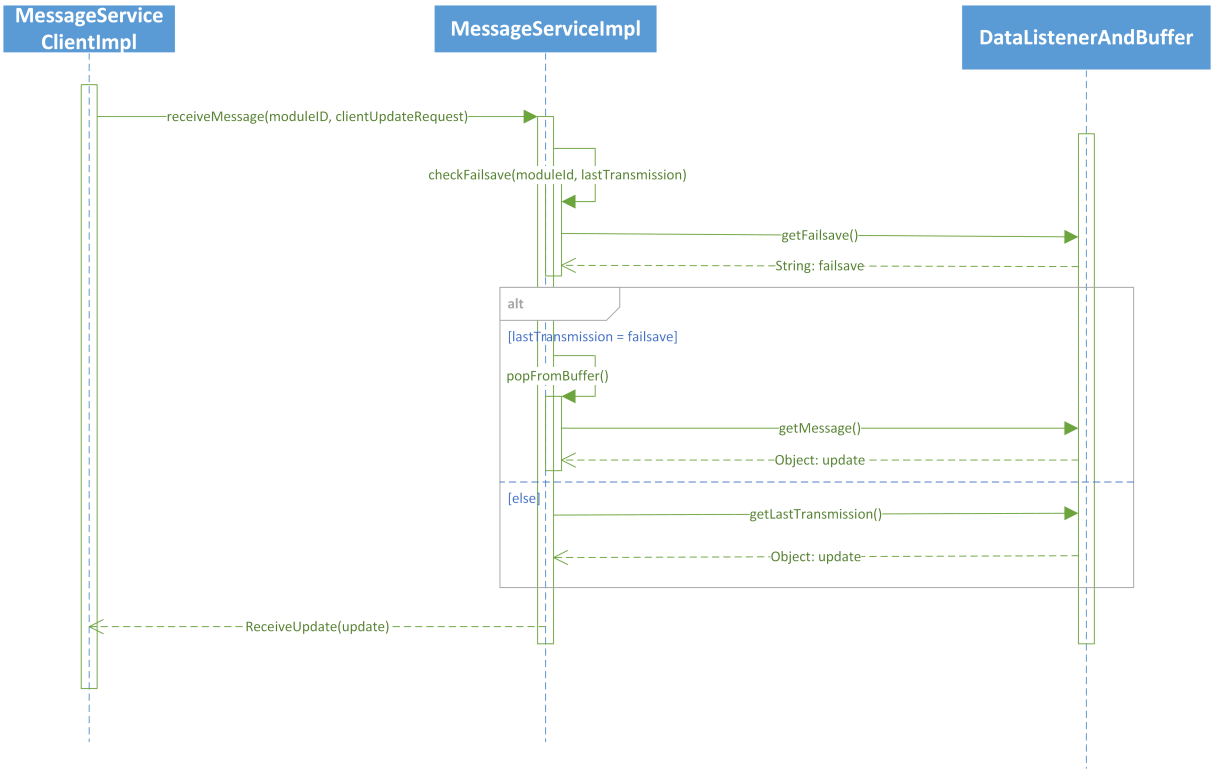


Abbildung 8.5: Der Client polled und fraget nach neuen Updates vom Server

8.6 Ausloggen und Disconnect

Versucht ein Client sich beim Server auszuloggen, beispielsweise um den Benutzer zu wechseln, so sendet er ein entsprechendes Request Objekt an das Servlet. Schlägt die Übertragung fehl, so wird der Benutzer darüber informiert und hat die Möglichkeit erneut zu versuchen sich auszuloggen. Erhält das Servlet die Anfrage, so wird der Benutzer abgemeldet und der Client erhält die Möglichkeit sich erneut einzuloggen.

Versucht ein Client sich vom Server abzumelden, beispielsweise beim Schließen der Applikation, so sendet er ein entsprechendes Request Objekt an das Servlet. Trifft dieses ein, so wird der Benutzer abgemeldet und die Verbindung geschlossen. Trifft dieses nicht ein, so besteht keine Möglichkeit dies dem Client mitzuteilen, da sein Dienst bereits beendet ist und der Server die Nachricht nie erhalten hat. In diesen Falle greift nach einer zur Compilezeit einstellbaren Zeitdauer (empfohlen 30 Minuten) ein Timeout, der den Client abmeldet und die Verbindung schließt. Dieser Timeout wird beim ersten Kontakt mit dem Client, beim Erstellen dessen Buffer gesetzt und bei jeder Nachricht vom Client zurückgesetzt, er greift also erst, wenn in der eingestellten Zeitdauer keine Requests vom Client kamen. Bei ausreichend großer Dauer (20+ Minuten) kann davon ausgegangen werde, dass der Client nicht mehr teilnimmt. Bei zu kurzer Dauer (5 Minuten) könnte ein Verlust der Verbindung zum ungewollten schließen der Verbindung führen.

8.7 Timeouts und Failsafes

Sowohl auf Client- als auch auf Servletseite existieren Failsafe-Strukturen um eine verlustfreie Kommunikation zu ermöglichen.

Client Seite: Hier stellt der Dispatcher das Failsafe dar. Es wird solange versucht das Objekt zu übertragen, bis eine positive Rückmeldung vom Servlet erhalten wird, ein Verlust von Updates vom Client an den Server ist daher unwahrscheinlich.

Server Seite: Jedem an den Client gesendeten Update wird ein Identifikator mitgegeben. Dieser wird vom Client gespeichert und bei der nächsten Updateanfrage wieder an das Servlet zurückgegeben. Stimmen bei einer Anfrage nach einem Update der erhaltene und die Servletseitig gespeicherte Identifikator nicht überein, so wird das zuletzt gesendete Update erneut gesendet, ein Verlust von Updates vom Server an den Client ist daher unwahrscheinlich.

Zu Unterschieden in erhaltenem und hinterlegtem Identifikator kann aus primär zwei Gründen kommen: I: Probleme der Verbindung, wie z.B. Abbruch während der Übertragung. II: Da es sich um einen Webservice handelt müssen Timeouts für alle Interaktionen gesetzt werden. Für diesen Fall relevant sind die Timeouts des Asynchronen Callbacks des Clients und der des Servlets beim Warten auf ein neues Update vom Server. Da die Verbindungsqualität eine große Rolle für die allgemeine Übertragungsgeschwindigkeit spielt, der Timeout des Asynchronen Callbacks aber fest gesetzt werden muss (in diesem Falle idealer Weise mit einem Wert zwischen 30 und 60 Sekunden) kann es zu folgendem Szenario kommen: Der Client stellt eine Anfrage an den Server, in diesem Moment startet der Timer des Asynchronen Callbacks. Aufgrund schlechter Verbindung kommt die Anfrage einige Sekunden verzögert beim Server an, wo kein aktuelles Update vorliegt. In diesem Moment startet der Timer des Servers. Wird nun in den letzten Sekunden dieses Zeitfensters ein Update zur Verfügung gestellt, so kann es aufgrund der verzögerungen in der Verbindung möglich sein, dass die Rückrufadresse (der Callback) nicht mehr gültig ist und die Transaktion zwar auf Servletseite registriert würde aber den Client niemals erreicht. Dies kann durch Verlängern des Timers auf Servletseite zwar beliebig unwahrscheinlich gemacht werden, erhöht aber die Serverlast und ist nie komplett auszuschließen. Der Servletseitige Failsafe bietet hier eine günstigerere und sicherere Variante als beispielsweise ein 5 Minuten langer Timer.

9 Zeitplan

ID	Aufgabenname	Anfang	Abschluss	Dauer	Jun 2013		Jul 2013			
					16.6	23.6	30.6	7.7	14.7	21.7
1	Aufbau der Projektstruktur	17.06.2013	21.06.2013	1w						
2	Intensivwoche: Einbau aller Klassen und Interfaces. Implementierung erster Plugins	24.06.2013	28.06.2013	1w						
3	Implementierung weiterer Plugins und Komponententests	01.07.2013	05.07.2013	1w						
4	Puffer	08.07.2013	12.07.2013	1w						
5	Code Besprechung, Integration-tests, Finishing und verfassen des Implementierungsberichts	15.07.2013	19.07.2013	1w						
6	Abschluss der Implementierungsphase. Evaluation des Implementierungsberichts weitere Codeevaluation.	22.07.2013	26.07.2013	1w						

Glossar

Gwt: **G**oogle **W**eb **T**oolkit ist ein Toolkit zu Entwicklung von Webandwendungen. Die Kerntechnologie besteht aus einem Java zu JavaScript Compiler. Er erlaubt es Webanwendungen zu schreiben bei denen JS/CSS und HTML genutzt wird. Anstatt aber den Haupteil aller dynamischen Komopnennten in JavaScript zu schreiben, kann man dies nun in der für große Softwareprojekte beliebteren Sprache Java tun. Desweiteren bringt Gwt diverse vorgefertigte graphische Komponeten mit, sogenannte Widgets. Es existieren auch bereits Schnittstellen zur asynchronen Serverkommunikation.

5, 9, 25, 35, 37, 45, 49, 53

Hud: **H**ead **U**p **D**isplay ursprünglich aus der Luftfahrt stammender Begriff, der ein über das Geschehen gelegtes Sichtfeld beschreibt

9, 25, 53

Landscape Modus: Dieser Modus ist dann aktiv wenn man das Gerät so hält das die X Achse größer als die Y Achse ist. Dies ist bei Tablets die normale Nutzerhaltung.

19, 53

MVC: Der englischsprachige Begriff **m**odel **v**iew **c**ontroller (MVC; deutsch: Modell-Präsentation-Steuerung) ist ein Muster zur Strukturierung von Software-Entwicklung in die drei Einheiten Datenmodell (engl. model), Präsentation (engl. view) und Programmsteuerung (engl. controller). Ziel des Musters ist ein flexibler Programmentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht.

http://de.wikipedia.org/wiki/Model_View_Controller Wikipedia: MVC

5, 53

MVP: **m**odel-**v**iew-**p**resenter (Abkürzung MVP; wörtlich etwa ‚Modell-Ansicht-Präsentator') ist ein Entwurfsmuster in der Softwareentwicklung, das aus dem model-view-controller (MVC) hervorgegangen ist. Es beschreibt einen neuartigen Ansatz, um das Modell (engl. model) und die Ansicht (engl. view) komplett voneinander zu trennen und über einen Präsentator (engl. presenter) zu verbinden. Dabei steht neben einer deutlich verbesserten Testbarkeit auch die strengere Trennung der einzelnen Komponenten im Gegensatz zu MVC im Vordergrund.

http://de.wikipedia.org/wiki/Model_View_Presenter Wikipedia: MVP

5, 53

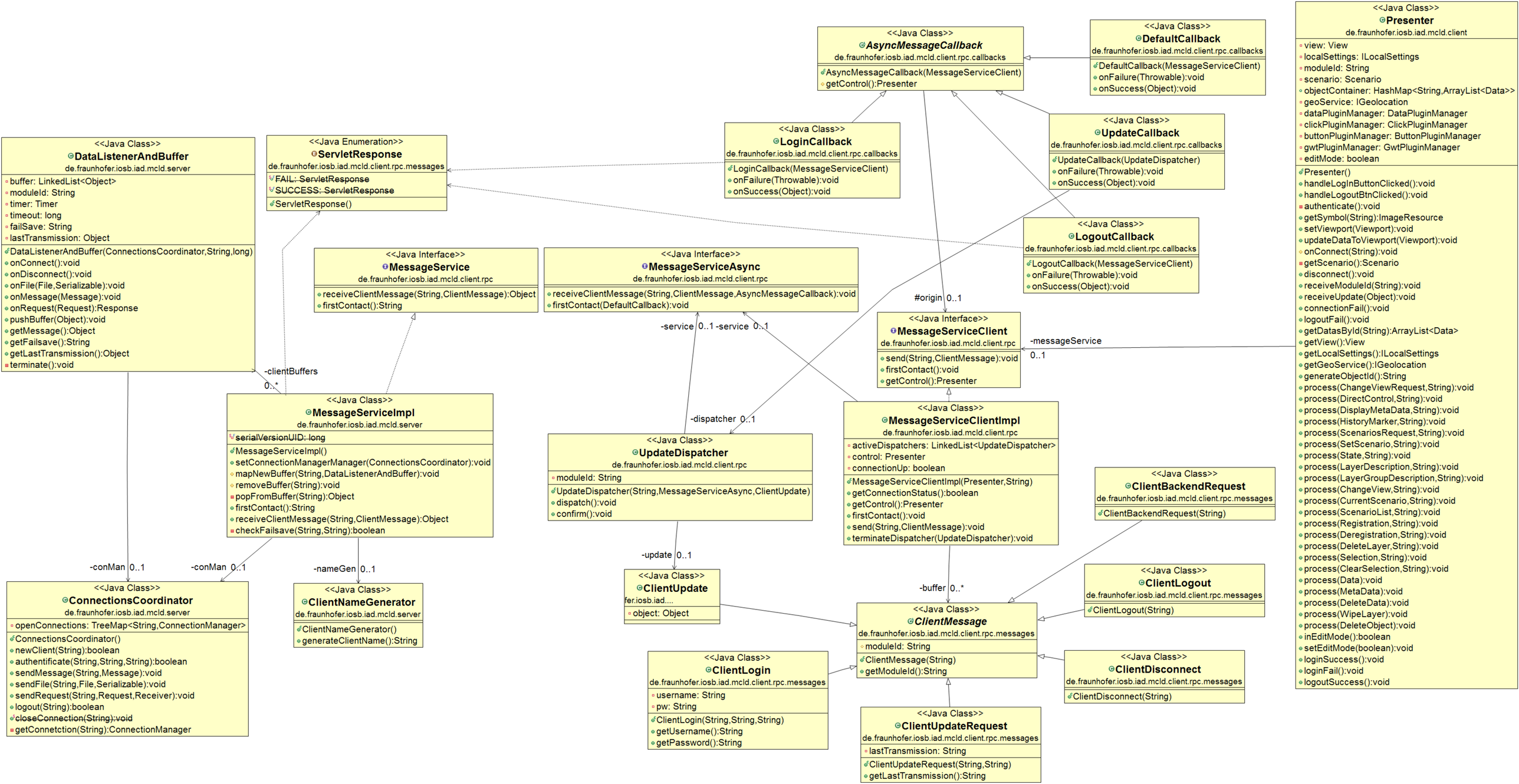
WFS: **W**eb **F**eature **S**ervice - eine Schnittstelle zum Abrufen von Vektordaten.

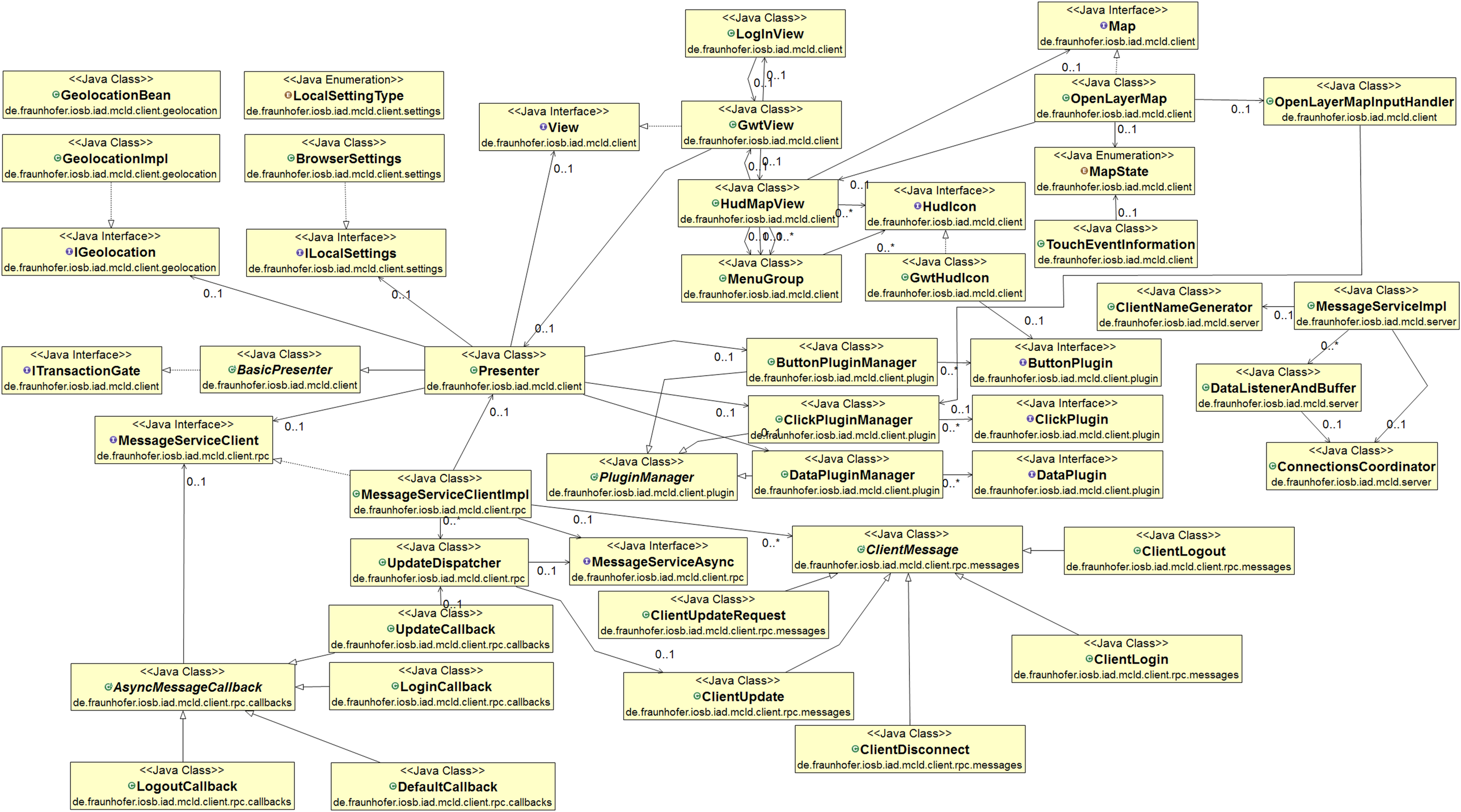
28, 36, 53

10 Anhang

10.1 Struktur aus Kapitel 8.1 in voller Größe

10.2 Gesamtübersicht aller Klassen


<<Java Class>>
DataListenerAndBuffer
de.fraunhofer.iosb.iad.mcld.server
buffer: LinkedList<Object>
moduleId: String
timer: Timer
timeout: long
failSave: String
lastTransmission: Object
DataListenerAndBuffer(ConnectionsCoordinator,String,long)
onConnect():void
onDisconnect():void
onFile(File,Serializable):void
onMessage(Message):void
onRequest(Request):Response
pushBuffer(Object):void
getMessage():Object
getFailsave():String
getLastTransmission():Object
terminate():void
<<Java Enumeration>>
ServletResponse
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
FAIL: ServletResponse
SUCCESS: ServletResponse
ServletResponse()
<<Java Class>>
AsyncMessageCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
AsyncMessageCallback(MessageServiceClient)
getControl():Presenter
<<Java Class>>
DefaultCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
DefaultCallback(MessageServiceClient)
onFailure(Throwable):void
onSuccess(Object):void
<<Java Class>>
LoginCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
LoginCallback(MessageServiceClient)
onFailure(Throwable):void
onSuccess(Object):void
<<Java Class>>
UpdateCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
UpdateCallback(UpdateDispatcher)
onFailure(Throwable):void
onSuccess(Object):void
<<Java Class>>
LogoutCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
LogoutCallback(MessageServiceClient)
onFailure(Throwable):void
onSuccess(Object):void
<<Java Class>>
Presenter
de.fraunhofer.iosb.iad.mcld.client
view: View
localSettings: ILocalSettings
moduleId: String
scenario: Scenario
objectContainer: HashMap<String,ArrayList<Data>>
geoService: IGeolocation
dataPluginManager: DataPluginManager
clickPluginManager: ClickPluginManager
buttonPluginManager: ButtonPluginManager
gwtPluginManager: GwtPluginManager
editMode: boolean
Presenter()
handleLogInButtonClicked():void
handleLogoutBtnClicked():void
authenticate():void
getSymbol(String):ImageResource
setViewport(Viewport):void
updateDataToViewport(Viewport):void
onConnect(String):void
getScenario():Scenario
disconnect():void
receiveModuleId(String):void
receiveUpdate(Object):void
connectionFail():void
logoutFail():void
getDatasById(String):ArrayList<Data>
getView():View
getLocalSettings():ILocalSettings
getGeoService():IGeolocation
generateObjectId():String
process(ChangeViewRequest,String):void
process(DirectControl,String):void
process(DisplayMetaData,String):void
process(HistoryMarker,String):void
process(ScenariosRequest,String):void
process(SetScenario,String):void
process(State,String):void
process(LayerDescription,String):void
process(LayerGroupDescription,String):void
process(ChangeView,String):void
process(CurrentScenario,String):void
process(ScenarioList,String):void
process(Registration,String):void
process(Deregistration,String):void
process(DeleteLayer,String):void
process(Selection,String):void
process(ClearSelection,String):void
process(Data):void
process(MetaData):void
process(DeleteData):void
process(WipeLayer):void
process(DeleteObject):void
inEditMode():boolean
setEditMode(boolean):void
loginSuccess():void
loginFail():void
logoutSuccess():void
<<Java Interface>>
MessageService
de.fraunhofer.iosb.iad.mcld.client.rpc
receiveClientMessage(String,ClientMessage):Object
firstContact():String
<<Java Interface>>
MessageServiceAsync
de.fraunhofer.iosb.iad.mcld.client.rpc
receiveClientMessage(String,ClientMessage,AsyncMessageCallback):void
firstContact(DefaultCallback):void
<<Java Interface>>
MessageServiceClient
de.fraunhofer.iosb.iad.mcld.client.rpc
send(String,ClientMessage):void
firstContact():void
getControl():Presenter
<<Java Class>>
MessageServiceImpl
de.fraunhofer.iosb.iad.mcld.server
serialVersionUID: long
MessageServiceImpl()
setConnectionManagerManager(ConnectionsCoordinator):void
mapNewBuffer(String,DataListenerAndBuffer):void
removeBuffer(String):void
popFromBuffer(String):Object
firstContact():String
receiveClientMessage(String,ClientMessage):Object
checkFailsave(String,String):boolean
<<Java Class>>
UpdateDispatcher
de.fraunhofer.iosb.iad.mcld.client.rpc
moduleId: String
UpdateDispatcher(String,MessageServiceAsync,ClientUpdate)
dispatch():void
confirm():void
<<Java Class>>
MessageServiceClientImpl
de.fraunhofer.iosb.iad.mcld.client.rpc
activeDispatchers: LinkedList<UpdateDispatcher>
control: Presenter
connectionUp: boolean
MessageServiceClientImpl(Presenter,String)
getConnectionStatus():boolean
getControl():Presenter
firstContact():void
send(String,ClientMessage):void
terminateDispatcher(UpdateDispatcher):void
<<Java Class>>
ConnectionsCoordinator
de.fraunhofer.iosb.iad.mcld.server
openConnections: TreeMap<String,ConnectionManager>
ConnectionsCoordinator()
newClient(String):boolean
authentificate(String,String,String):boolean
sendMessage(String,Message):void
sendFile(String,File,Serializable):void
sendRequest(String,Request,Receiver):void
logout(String):boolean
closeConnection(String):void
getConnetction(String):ConnectionManager
<<Java Class>>
ClientNameGenerator
de.fraunhofer.iosb.iad.mcld.server
ClientNameGenerator()
generateClientName():String
<<Java Class>>
ClientUpdate
fer.iosb.iad....
object: Object
<<Java Class>>
ClientMessage
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
moduleId: String
ClientMessage(String)
getModuleId():String
<<Java Class>>
ClientBackendRequest
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
ClientBackendRequest(String)
<<Java Class>>
ClientLogout
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
ClientLogout(String)
<<Java Class>>
ClientLogin
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
username: String
pw: String
ClientLogin(String,String,String)
getUsername():String
getPassword():String
<<Java Class>>
ClientDisconnect
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
ClientDisconnect(String)
<<Java Class>>
ClientUpdateRequest
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
lastTransmission: String
ClientUpdateRequest(String,String)
getLastTransmission():String
-clientBuffers
0..*
-service 0..1
-service 0..1
#origin 0..1
-messageService
0..1
-dispatcher 0..1
-update 0..1
-buffer 0..*
-conMan 0..1
-conMan 0..1
-nameGen 0..1


<<Java Class>>
LogInView
de.fraunhofer.iosb.iad.mcld.client
<<Java Interface>>
Map
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
GeolocationBean
de.fraunhofer.iosb.iad.mcld.client.geolocation
<<Java Enumeration>>
LocalSettingType
de.fraunhofer.iosb.iad.mcld.client.settings
<<Java Class>>
OpenLayerMap
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
OpenLayerMapInputHandler
de.fraunhofer.iosb.iad.mcld.client
<<Java Interface>>
View
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
GwtView
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
GeolocationImpl
de.fraunhofer.iosb.iad.mcld.client.geolocation
<<Java Class>>
BrowserSettings
de.fraunhofer.iosb.iad.mcld.client.settings
<<Java Enumeration>>
MapState
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
HudMapView
de.fraunhofer.iosb.iad.mcld.client
<<Java Interface>>
HudIcon
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
TouchEventInformation
de.fraunhofer.iosb.iad.mcld.client
<<Java Interface>>
IGeolocation
de.fraunhofer.iosb.iad.mcld.client.geolocation
<<Java Interface>>
ILocalSettings
de.fraunhofer.iosb.iad.mcld.client.settings
<<Java Class>>
MenuGroup
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
GwtHudIcon
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
ClientNameGenerator
de.fraunhofer.iosb.iad.mcld.server
<<Java Class>>
MessageServiceImpl
de.fraunhofer.iosb.iad.mcld.server
<<Java Interface>>
ITransactionGate
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
BasicPresenter
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
Presenter
de.fraunhofer.iosb.iad.mcld.client
<<Java Class>>
ButtonPluginManager
de.fraunhofer.iosb.iad.mcld.client.plugin
<<Java Interface>>
ButtonPlugin
de.fraunhofer.iosb.iad.mcld.client.plugin
<<Java Class>>
DataListenerAndBuffer
de.fraunhofer.iosb.iad.mcld.server
<<Java Class>>
ClickPluginManager
de.fraunhofer.iosb.iad.mcld.client.plugin
<<Java Interface>>
ClickPlugin
de.fraunhofer.iosb.iad.mcld.client.plugin
<<Java Interface>>
MessageServiceClient
de.fraunhofer.iosb.iad.mcld.client.rpc
<<Java Class>>
PluginManager
de.fraunhofer.iosb.iad.mcld.client.plugin
<<Java Class>>
DataPluginManager
de.fraunhofer.iosb.iad.mcld.client.plugin
<<Java Interface>>
DataPlugin
de.fraunhofer.iosb.iad.mcld.client.plugin
<<Java Class>>
ConnectionsCoordinator
de.fraunhofer.iosb.iad.mcld.server
<<Java Class>>
MessageServiceClientImpl
de.fraunhofer.iosb.iad.mcld.client.rpc
<<Java Class>>
UpdateDispatcher
de.fraunhofer.iosb.iad.mcld.client.rpc
<<Java Interface>>
MessageServiceAsync
de.fraunhofer.iosb.iad.mcld.client.rpc
<<Java Class>>
ClientMessage
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
<<Java Class>>
ClientLogout
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
<<Java Class>>
ClientUpdateRequest
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
<<Java Class>>
UpdateCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
<<Java Class>>
ClientLogin
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
<<Java Class>>
AsyncMessageCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
<<Java Class>>
LoginCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
<<Java Class>>
ClientUpdate
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
<<Java Class>>
ClientDisconnect
de.fraunhofer.iosb.iad.mcld.client.rpc.messages
<<Java Class>>
LogoutCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
<<Java Class>>
DefaultCallback
de.fraunhofer.iosb.iad.mcld.client.rpc.callbacks
0..1
0..*