

# FirstSpirit™

*Unlock Your Content*

## FirstSpirit AppCenter FirstSpirit Version 5.0

<b>Version</b>	<b>0.46</b>
<b>Status</b>	<b>in process</b>
<b>Datum</b>	<b>2013-02-18</b>
Abteilung	Techn. Documentation
Autor/ Autoren	B.Ehle
Copyright	2013 <b>e-Spirit AG</b>
Dateiname	APPC50DE_FirstSpirit_Modules_AppCenter

### **e-Spirit AG**

Barcelonaweg 14  
44269 Dortmund | Germany

T +49 231 . 477 77-0  
F +49 231 . 477 77-499

[info@e-spirit.de](mailto:info@e-spirit.de)  
[www.e-spirit.de](http://www.e-spirit.de)

**e-Spirit**

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>4</b>
1.1	FirstSpirit-AppCenter – FirstSpirit als Integrationsplattform	4
1.2	Applikationsintegration im WebClient	5
1.3	Klassifikation	6
1.4	Allgemeine Hinweise	8
1.5	Einschränkungen	8
1.6	Lizenzmodell	9
1.7	Ausblick	10
1.8	Thema dieser Dokumentation	11
<b>2</b>	<b>Konzept: Integration einer Webapplikation in FirstSpirit</b>	<b>12</b>
2.1	Kommunikation: FirstSpirit JavaClient – Webapplikation	12
2.1.1	Steuerung des integrierten Browsers	12
2.1.2	Kommunikation zwischen Browser-Instanz und JavaClient	13
2.1.3	Konvertierung von Datentypen	16
2.1.4	Rückgabe per Callback-Funktion	17
2.2	DOM-Access: Zugriff auf die Daten des integrierten Browsers	18
<b>3</b>	<b>Standarterweiterungen</b>	<b>20</b>
3.1	Interface: ApplicationService	21
3.2	Interface: ApplicationTab	23
3.3	Interface: ApplicationTabAppearance	26
3.4	Interface: ApplicationTabConfiguration	30
3.5	Interface: TabListener	33
3.6	Abstract Class: ApplicationType	34



3.7	Interface: BrowserApplication.....	35
3.8	Interface: BrowserListener.....	38
3.9	Interface: BrowserApplicationConfiguration .....	39
3.10	Interface: ClientServiceRegistryAgent .....	41
<b>4</b>	<b>Beispiel: Integration von Google Maps in FirstSpirit .....</b>	<b>42</b>
4.1	Erste Schritte .....	43
4.1.1	Hinweis auf das FirstSpirit-Lizenzmodell .....	43
4.1.2	Hinweis auf rechtliche Implikation .....	43
4.1.3	Google Maps-API-Schlüssel generieren .....	43
4.1.4	Hinweis zur Konfiguration des FirstSpirit-Servers .....	44
4.1.5	Installation des Google Earth Plug-ins.....	45
4.1.6	Beispiel-Projekt .....	45
4.2	Anwendungsbereiche der Google Maps Integration .....	46
4.2.1	Adress-Suche mit Geolokalisierung.....	46
4.2.2	Ändern der Koordinate über die Google Maps Integration .....	47
4.2.3	Einblenden zusätzlicher Informationen (Google Balloons).....	48
4.2.4	3D-Darstellung über Google Earth.....	49
4.2.5	Anfahrtsbeschreibung .....	50
4.3	Implementierung: Applikationsintegration für Google Maps .....	52
4.3.1	(SwingGadget-) Eingabekomponente CUSTOM_GEOLOCATION... ..	54
4.3.2	MapsPlugin – Neue Instanz vom Typ MapsPlugin erzeugen .....	55
4.3.3	MapsPlugin – Öffnen der Applikation innerhalb eines Tabs .....	57
4.3.4	MapsPlugin – JavaScript ausführen (Java » JavaScript).....	61
4.3.5	MapsPlugin - GeolocationUpdater (Injection Java » JavaScript) .	65
4.3.6	Markierungen einblenden und einer Eingabekomponente zuordnen .....	69
4.3.7	Listener – Auf Änderungen reagieren .....	77
4.3.8	Geodaten der Eingabekomponente aktualisieren (JavaScript »	



- Java)..... 83
- 4.3.9 Auf Baum-Navigations-Events reagieren (Java » JavaScript)..... 91
- 4.3.10 Browser-Instanz aktualisieren (Java » JavaScript) ..... 93
- 4.3.11 MapsPlugin – Adress-Suche (Google-Geolocation) ..... 97
- 4.3.12 maps.html – Einführung..... 104
- 4.3.13 Exkurs: HTML und JavaScript ..... 105
- 4.3.14 maps.html - Laden der Google Maps-API..... 107
- 4.3.15 maps.html - Container für die Kartendarstellung initialisieren ..... 107
- 4.3.16 maps.html – Neues Kartenobjekt erstellen..... 109
- 4.3.17 maps.html - Karte zentrieren (Mittelpunkt bzw. Anzeigebereich) 110
- 4.3.18 maps.html – Kartentyp definieren..... 113
- 4.3.19 maps.html - Kartenobjekt über Ereignisse laden ..... 115
- 4.3.20 maps.html – Adressdaten konvertieren (Geocoding) ..... 116
- 4.3.21 maps.html – GeolocationUpdater (Injection Java / JavaScript) .. 117
  
- 5 Listings ..... 121**



# 1 Einleitung



*Die Dokumentation befindet sich zurzeit in Bearbeitung. Einige Aspekte und Schnittstellen sind noch nicht oder noch unvollständig dokumentiert.*

## 1.1 FirstSpirit-AppCenter – FirstSpirit als Integrationsplattform

FirstSpirit wurde von Anfang an als Integrationsplattform konzipiert und realisiert. Dazu gehört die konsequente Fokussierung aller Eigenimplementierungen auf die Kernbestandteile des Produktes "FirstSpirit" und die bewusste Verlagerung von spezifischen Funktionen in Fremdprodukte der jeweiligen Marktführer. Der Erfolg einer solchen Best-of-Breed-Strategie steht und fällt mit der Leistungsfähigkeit der Systemintegration. Die entscheidende Voraussetzung, um diesen populären Outsourcing-Gedanken in einem Softwareprodukt erfolgreich umsetzen zu können, ist dabei die "nahtlose Integration": es darf für den Endbenutzer keinen Bruch zwischen den eingesetzten Produkten geben. Die Benutzerführung muss für den Anwender voll integriert, nahtlos und optisch wie aus einem Guss erscheinen.

Dieser Gedanke der nahtlosen Integration von Fremdanwendungen in die FirstSpirit Redaktionsumgebung wird als "AppCenter" bezeichnet.

Das FirstSpirit AppCenter stellt einen definierten Bereich innerhalb des Redaktionssystems zur Verfügung, in dem eigenständige Anwendungen ablaufen können, die nicht Bestandteil von FirstSpirit sind (sogenannte "AppCenter-Anwendungen"). Beispiele für AppCenter-Anwendungen sind die Integration von Microsoft Office oder die Funktionen zur integrierten Bildbearbeitung. Auch bei den integrierten Webbrowsern Mozilla Firefox und Microsoft Internet Explorer handelt es sich um AppCenter-Anwendungen, sie werden als "Integrierte Vorschau" betitelt. All diese AppCenter-Anwendungen wurden von e-Spirit als Produktbestandteile realisiert. Es gibt aber auch eine Reihe von AppCenter-Anwendungen, die als FirstSpirit-Module realisiert werden. Diese AppCenter-Module können sowohl von e-Spirit selbst, aber auch von einem Partner entwickelt werden.

Folgende Beispiele, die im Rahmen des AppCenters bereits erfolgreich von e-Spirit integriert wurden, vermitteln einen Eindruck davon, welche Möglichkeiten das AppCenter über die aktuell als Produktbestandteile realisierten Anwendungen hinaus bietet:



- mittels der Integration von Google Maps bzw. Google Earth kann im FirstSpirit-JavaClient einfach und intuitiv mit Geo-Koordinaten gearbeitet werden (siehe Kapitel 4 Seite 42).
- mittels der Integration des Online-Video-Angebots von MovingImage24 können Videos ausgewählt und per Mausklick in FirstSpirit eingebunden werden.

Technisch gesehen besteht das AppCenter aus einer Menge von Schnittstellen, die von e-Spirit für die Nutzung durch Partner freigegeben wurden, damit diese im Rahmen des AppCenters individuelle Anwendungen realisieren und integrieren können, um so den Client an ihre speziellen Bedürfnisse anzupassen. Die Realisierung von AppCenter-Anwendungen und deren Integration in FirstSpirit wird allgemein als "Applikationsintegration" bezeichnet. Darunter versteht man die möglichst nahtlose Einbindung von Fremdsoftware, auch mit völlig unterschiedlicher Technologie-Basis, in die Redaktionsoberfläche des JavaClients.

#### Fazit:

- Der Redakteur findet eine vollintegrierte Arbeitsoberfläche vor, in der er mit den ihm vertrauten Werkzeugen (beispielsweise Office-Anwendungen) sofort arbeiten kann.
- Vorhandene, kundenspezifische Applikationen können ohne großen Aufwand integriert werden, auch wenn es sich um Nicht-Java-Anwendungen handelt.
- Für den Kunden können schnell und effektiv neue, individuell auf seinen Anwendungsfall zugeschnittene und auf vorhandener Technik basierende Mini-Apps entwickelt werden. Das können kleine Web-Applikationen auf .NET-Basis sein oder auch Flash-Anwendungen.

## 1.2 Applikationsintegration im WebClient

Auch im WebClient sind ab FirstSpirit Version 5.0 Applikationsintegrationen in Form von Reports verfügbar. Eine Erweiterung wird in FirstSpirit Version 5.1 mit dem Ziel erfolgen, Applikationsintegrationen clientübergreifend in JavaClient und WebClient verfügbar zu machen.



## 1.3 Klassifikation

Bei der Integration von Webapplikation wird zwischen folgenden Varianten unterschieden:

- a) Blackbox-Integration:  
Die zu integrierende Webapplikation bietet eine wie auch immer geartete Schnittstelle, die zur Integration genutzt wird. Diese Schnittstelle kann entweder in Form einer API, über HTTP-Parameter oder JavaScript definiert sein oder auch in Form von definierten HTML- oder URL-Konstrukten vorliegen. In allen Fällen gilt aber: der innere Aufbau der zu integrierenden Anwendung muss nicht bekannt sein, sondern die Interaktion erfolgt ausschließlich über die definierten API-Schnittstellen.
- b) Greybox-Integration:  
Die Greybox-Integration geht davon aus, dass die Schnittstelle zwischen der Webapplikation und dem FirstSpirit-Client der im Webbrowser befindliche HTML-Code ist. Dieser HTML-Code muss analysiert und ggf. auch manipuliert werden, um an die benötigten Informationen zu gelangen (vgl. Kapitel 2.2 Seite 18). Das bedeutet, es wird Wissen über den inneren Aufbau der zu integrierenden Webapplikation benötigt, was im Fall der Blackbox-Integration nicht erforderlich ist.
- c) Whitebox-Integration:  
Eine Whitebox-Integration liegt dann vor, wenn die Webapplikationen speziell für den Einsatz im Rahmen der FirstSpirit Applikationsintegration entwickelt bzw. modifiziert wurden. In einer whitebox-integrierten Webapplikation werden also gezielt FirstSpirit-Interfaces angesprochen bzw. für FirstSpirit geeignete Einsprungpunkte bereitgestellt, um die Applikationsintegration zu realisieren. Für diese Art der Integration wird natürlich nicht nur Zugriff auf den Quellcode der Webapplikation, sondern auch die Möglichkeit zur Veränderung der Anwendung benötigt.

Eine Blackbox-Integration eignet sich daher speziell für die Integration von Webapplikationen, die bereits existieren und nicht verändert werden können oder sollen (unternehmensspezifische Webapplikation) – zwingende Voraussetzung ist aber, dass die notwendigen Schnittstellen bereitstehen.

Wenn die zu integrierenden Webapplikation nicht die erforderlichen Schnittstellen bereitstellt oder der Quellcode der zu integrierenden Webapplikation nicht verändert werden soll oder kann, ist eine Greybox-Integration die richtige Lösung – wobei natürlich zu beachten ist, dass bei



einer Veränderung der Webapplikation (z. B. Relaunch) vermutlich Anpassungen an der Integration vorgenommen werden müssen. Eine Greybox-Integration eignet sich daher nur für die Integration von Webapplikationen, die nur wenigen (z. B. Wikipedia) oder gar keinen Veränderungen (Legacy-Unternehmens-Webapplikationen) unterliegen.

Die Integrationsform mit den meisten Möglichkeiten ist die Whitebox-Integration, die eine sehr tiefgehende Integration zwischen Webapplikation und FirstSpirit ermöglicht – hier kann zum einen aus dem JavaClient heraus eine Steuerung der Webapplikation vorgenommen werden oder aber auch umgekehrt aus der Webapplikation heraus eine Steuerung des JavaClients erfolgen. Eine Whitebox-Integration bietet speziell in Verbindung mit FirstSpirit-Modulen ein enormes Potenzial, da Teile der Benutzungsoberfläche des Moduls in Form einer Webapplikation realisiert werden können, was speziell dann Vorteile bringt, wenn bestimmte Schnittstellen nur für Webapplikationen zur Verfügung stehen.

**Fazit:** Es können im Prinzip alle Arten von Webapplikation im Rahmen der Applikationsintegration nahtlos in die Benutzungsoberfläche des FirstSpirit-JavaClients integriert werden – das gilt speziell im Rahmen der Greybox-Integration auch für Webapplikationen, die "eigentlich" nicht für eine Integration vorgesehen wurden. Allerdings erkaufte man sich (prinzipbedingt) eine Integration ohne explizite API mit einer höheren Abhängigkeit von der konkreten Implementierung der integrierten Anwendung, d. h. Änderungen in der zu integrierenden Anwendung führen potenziell zur Notwendigkeit, den Integrations-Code anzupassen. Die Techniken und Verfahren der FirstSpirit Greybox-Integration reduzieren diese Abhängigkeiten zwar so weit wie möglich, können die Änderungsabhängigkeiten aber prinzipbedingt nicht vollständig auflösen.



## 1.4 Allgemeine Hinweise

Bei der Integration und Verwendung von kundenindividuellen AppCenter-Anwendungen ist grundsätzlich zu beachten, dass FirstSpirit für die Applikationsintegration die erforderlichen Schnittstellen bereitstellt, aber in der Regel keinen Einfluss auf die integrierten Anwendungen selbst hat.

Integrierte Fremdanwendungen sind kein Produktbestandteil von FirstSpirit. Das bedeutet u.a., dass die Verantwortung für die Funktionalität der integrierten Anwendungen beim Hersteller bzw. beim Kunden oder Partner liegt, der die Anwendung realisiert.

Probleme können im Rahmen des FirstSpirit-Produktsupports gemeldet werden und werden (nach Möglichkeit) beseitigt, wenn sie auf der Ebene der Integrations-Schnittstelle liegen. Ein Anspruch auf Fehlerbeseitigungen innerhalb der integrierten Fremdanwendungen gegenüber e-Spirit besteht aber nicht.

Die Verwendung eigener Anwendungen im AppCenter erfordert eine Lizenz. Nähere Informationen dazu siehe Kapitel 1.6 Seite 9.

## 1.5 Einschränkungen

Die Applikationsintegration baut auf der bestehenden Webbrowser-Integration, der Browser Microsoft Internet Explorer und Mozilla Firefox, im FirstSpirit JavaClient auf (siehe Kapitel 2.1 Seite 12). Bei der Verwendung der Webbrowser-Integrationen im JavaClient kann es prinzipbedingt zu Einschränkungen kommen, z. B. weil einige der integrierten Anwendungen nicht vollständig mit allen Plattformen oder Bittigkeiten (32 oder 64 Bit) zusammenarbeiten.

Es wird empfohlen, einen Internet Explorer ab Version 8 zu verwenden. Internet Explorer bis Version 8 unterstützen keine Base64-Dekodierung. Dies kann zu Problemen bei der Injektion von Bild-Elementen im Rahmen der Applikationsintegration führen (z. B. bei der Anzeige der Komponente FS\_BUTTON in der integrierten Vorschau oder der Integration einer Bilddatenbank).

Zu Voraussetzungen und Einschränkungen der Applikationsintegration siehe FirstSpirit™ Release Notes zur Version 4.2.R4.



## 1.6 Lizenzmodell

Der Lizenzparameter `license.APPTAB_SLOTS` gibt an, wie viele verschiedene Applikationsintegrationen verwendet werden können. Dazu zählen sowohl Anwendungen, die im AppCenter des JavaClient, als auch solche, die im WebClient verwendet werden können, z. B. selbstimplementierte Reports. Mit `license.APPTAB_SLOTS=5` können z. B. fünf verschiedene Anwendungen verwendet werden. Welche Anwendungen das sind, ist dabei unerheblich. Denn im Unterschied zur Lizenzierung von FirstSpirit-(Modul-)Erweiterungen wird hier nicht die Funktionalität lizenziert, sondern die Anzahl der integrierten Anwendungen.

Jede über diesen Parameter lizenzierte Applikationsintegration kann in beliebig vielen Java- oder WebClients geöffnet werden, im JavaClient darüber hinaus in beliebig vielen Tabs. Der erste Client, in dem eine Applikationsintegration geöffnet wird, belegt eine Lizenz ("AppTab-Slot") für diese Anwendung (es erfolgt eine "Registrierung" der Anwendung) und erhöht den Zähler des Lizenzparameters um 1.



*Dabei gilt: Ein AppTab-Slot wird für jeweils eine aufrufende Stelle, z. B. ein Skript, und das Öffnen einer URL vergeben. Werden beispielsweise innerhalb eines Skripts mehrere URLs geöffnet, handelt es sich um einen Verstoß gegen die FirstSpirit-AppCenter-Lizenzbedingungen.*

Die Registrierung bleibt auch nach dem Beenden des jeweiligen Clients weiterhin bestehen. Ist der Wert des `license.APPTAB_SLOTS`-Parameters erreicht, kann zu Test- und Demozwecken noch eine weitere Applikationsintegration im jeweiligen Client gestartet werden. In den Clients bzw. für selbstimplementierte WebClient-Reports in den Projekteigenschaften (Anwendung zur "Server- und Projektkonfiguration", "WebEdit-Einstellungen" / "Report Plug-ins") wird eine entsprechende Warnung angezeigt und eine Warn-Meldung in der Datei `fs-server.log` protokolliert. Darüber hinaus können keine weiteren Anwendungen gestartet werden.

Einige Anwendungen, die zwar im AppCenter des JavaClients dargestellt, aber standardmäßig mit dem FirstSpirit-Kernprodukt ausgeliefert werden oder die über einen gesonderten Parameter lizenziert werden (z. B. die Office-Integration für FirstSpirit), fallen nicht unter den Lizenzierungsparameter `license.APPTAB_SLOTS` und werden nicht als Applikationsintegration gezählt, zurzeit sind das:



- Integrierte WYSIWYG-Vorschau (über Mozilla Firefox bzw. Microsoft Internet Explorer)
- Integrierte Vorschau von Medien
- Integrierte Anzeige der FirstSpirit Online-Hilfe
- Integrierte, erweiterte Bildbearbeitung im JavaClient

Die Eingabekomponente `FS_BUTTON` (siehe Online Dokumentation für FirstSpirit) ermöglicht die Integration von eigenen Anwendungen im AppCenter des JavaClients. Bei Verwendung von `FS_BUTTON` wird jedes Skript und jede Klasse, die von `FS_BUTTON` referenziert wird, als Anwendung gezählt, die über den Parameter `license.APPTAB_SLOTS` lizenzpflichtig ist.

Die Art und Anzahl der Anwendungen, die über den Parameter `license.APPTAB_SLOTS` aktuell registriert sind, können im Server-Monitoring im Untermenü "AppCenter Lizenzen" unterhalb des Menüs "FirstSpirit" / "Steuerung" geprüft werden.

Über den Button "Verwendungen zurücksetzen" kann bei Bedarf die Zahl der registrierten Anwendungen auf 0 zurückgesetzt werden. Registrierte Anwendungen, die aktuell in Clients geöffnet sind, können so lange weiter verwendet werden, bis die Anwendung bzw. das zugehörige Applikations-Tab geschlossen wird. Der Server muss nicht neu gestartet werden.

## 1.7 Ausblick

Die Entwicklung hinsichtlich der Web-Applikationsintegration ist in FirstSpirit derzeit noch nicht abgeschlossen: Erweiterungen in der Implementierung sollen eine tiefere Integration ermöglichen.

Nach der Applikationsintegration für den Redaktionsarbeitsplatz soll in einer weiteren Ausbaustufe der Entwicklerarbeitsplatz (z. B. Integration von Entwicklungsumgebungen) in den Fokus rücken. Die Integrationskomplexität wird dabei vermutlich deutlich höher liegen.



## 1.8 Thema dieser Dokumentation

Nachdem in diesem einleitenden Kapitel der Begriff der "Applikationsintegration" erläutert und die Begriffe "Blackbox-" vs. "Greybox-" vs. "Whitebox-"Integration gegenübergestellt wurden, werden nachfolgend die technischen Grundlagen für die kundenspezifische Integration von Webapplikationen beschrieben. Ebenso werden die zugrundeliegenden Interfaces, Packages und Klassen aufgelistet und erläutert. Alle Konzepte sowie die erforderlichen FirstSpirit-API-Schnittstellen werden anhand von Beispiel-Implementierungen vorgestellt.

Im Mittelpunkt dieses Dokuments steht die clientseitige Applikationsintegration (siehe Kapitel 4 Seite 42, Punkt 3) mit FirstSpirit. Dabei werden Konzepte sowie die erforderlichen FirstSpirit-API-Schnittstellen anhand von Beispiel-Implementierungen vorgestellt.



*Die Dokumentation befindet sich zurzeit in Bearbeitung. Einige Aspekte und Schnittstellen sind noch nicht oder noch unvollständig dokumentiert.*

**Kapitel 2:** In diesem Kapitel werden zunächst die hinter der Applikationsintegration liegenden Konzepte erläutert. Dabei geht es insbesondere um die Kommunikation zwischen der integrierten Webapplikation und dem FirstSpirit JavaClient (ab Seite 12).

**Kapitel 3:** Das Kapitel stellt die Standarderweiterungen der FirstSpirit-Access-API für die Applikationsintegration vor. Hier werden unter anderem die Schnittstellen für die Steuerung der Applikations-Tabs und die Integration von Webapplikationen vorgestellt (ab Seite 20).

**Kapitel 4:** Dieses Kapitel beschreibt die Beispiel-Implementierung zur Integration von Google Maps in den FirstSpirit JavaClient. Die vorgestellte Implementierung schafft eine einfache und intuitive Möglichkeit, um innerhalb der FirstSpirit-Redaktionsumgebung mit geographischen Koordinaten zu arbeiten. Dazu wird eine SwingGadget-Eingabekomponente entwickelt, die eng mit der Webapplikation Google Maps verknüpft ist (ab Seite 42).



## 2 Konzept: Integration einer Webapplikation in FirstSpirit

### 2.1 Kommunikation: FirstSpirit JavaClient – Webapplikation

Der FirstSpirit-JavaClient verfügt über einen nahtlos integrierten Webbrowser, der nicht nur eine direkte Vorschau der redaktionellen Inhalte im JavaClient anzeigt, sondern auch den Zusammenhang zwischen den im Client eingegebenen Inhalten und ihrer Auswirkung bzw. Darstellung auf der Webseite visualisiert. Dazu stehen wahlweise die Webbrowser Mozilla Firefox und Microsoft Internet Explorer zur Verfügung.

Auf diese Browser-Integration wird auch bei der Integration von Webapplikationen zurückgegriffen. Für die Integration einer Webapplikation in den FirstSpirit-JavaClient müssen zunächst folgende Aspekte abgedeckt werden:

- Steuerung des integrierten Browsers (siehe Kapitel 2.1.1 Seite 12)
- Kommunikation zwischen Browser-Instanz und JavaClient (siehe Kapitel 2.1.2 Seite 13)
- Konvertierung von Datentypen (siehe Kapitel 2.1.3 Seite 16)
- Rückgabe per Callback-Funktion (siehe Kapitel 2.1.4 Seite 17)

#### 2.1.1 Steuerung des integrierten Browsers

Um eine Webapplikation in den FirstSpirit-JavaClient zu integrieren, ist es notwendig, den in FirstSpirit integrierten Browser zu steuern. Die FirstSpirit-AppCenter-API stellt die benötigten Schnittstellen bereit, um beispielsweise ein neues Tab (bzw. eine neue Browser-Instanz) innerhalb des Applikationsbereichs zu öffnen, in dem die gewünschte Webapplikation aufgerufen werden kann (Beschreibung der Schnittstellen siehe Kapitel 3, Seite 20ff.).

Einstiegspunkt für die Steuerung eines Tabs ist der `ApplicationService` (siehe Kapitel 3.1 Seite 20). Über diesen Service kann eine neue Anwendung eines bestimmten Typs innerhalb des Applikationsbereichs geöffnet werden. Der gewünschte `ApplicationType` wird beim Öffnen der Anwendung übergeben (Abstract Class: `ApplicationType` siehe Kapitel 3.6 Seite 34). Für die Integration einer Webapplikation wird der `ApplicationType` `BrowserApplication` benötigt, der eine Schnittstelle zum Öffnen und Steuern einer neuen Browser-Instanz im Applikationsbereich anbietet (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite



35).

Die Methode `openApplication(...)` des Interfaces `ApplicationService` liefert eine Instanz vom Typ `ApplicationTab` zurück. Das Interface `ApplicationTab` bietet allgemeine Methoden zur Steuerung des Tabs an, beispielsweise kann der Tab über die entsprechenden Methodenaufrufe in den Vordergrund geholt oder geschlossen werden (Interface: `ApplicationTab` siehe Kapitel 3.2 Seite 23). Darüber hinaus kann über die Instanz vom Typ `ApplicationTab` die (Browser-)Applikation geholt werden, die innerhalb des Applikations-Tabs geöffnet wurde. Diese Instanz bietet dann den Zugriff auf weitere spezifische Möglichkeiten zur Steuerung der integrierten Anwendung (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35). Zum Verfolgen von Änderungen können außerdem geeignete Listener registriert werden, eine Instanz vom Typ `BrowserListener`, die auf Änderungen innerhalb der Webanwendung reagiert (Interface: `BrowserListener` siehe Kapitel 3.8 Seite 38) und eine Instanz vom Typ `TabListener`, die auf Änderungen innerhalb des Tabs reagiert (z. B. Selektion oder Deselektion durch den Benutzer) (Interface: `TabListener` siehe Kapitel 3.5 Seite 33).

### 2.1.2 Kommunikation zwischen Browser-Instanz und JavaClient

Die integrierten Browser-Engines sind natürlich nicht in Java, sondern nativ für das jeweilige Client-Betriebssystem realisiert. Der wichtigste Punkt für die Integration einer Webapplikation in FirstSpirit ist damit die Kommunikation zwischen der Java-Ebene des FirstSpirit-JavaClients und der nativen Browser-Ebene der Webapplikation. Dabei müssen zwei Kommunikationswege berücksichtigt werden:

- 1) **JavaClient » Webapplikation:** Änderungen oder Ereignisse, die über den FirstSpirit-JavaClient angestoßen werden, müssen der Webapplikation bekannt gegeben werden. Wird beispielsweise innerhalb der Geolocation-Eingabekomponente nach einer bestimmten Adresse gesucht (Eingabe eines Adress-Strings und Klick auf den Search-Button), muss eine Anfrage zur Geokodierung dieses Adress-Strings an die Webapplikation (Google Maps) gesendet und der Kartenausschnitt, innerhalb des integrierten Browsers angepasst werden (vgl. Beispiel Adress-Suche mit Geolokalisierung in Kapitel 4.2.1 Seite 46).
- 2) **Webapplikation » JavaClient:** Der umgekehrte Weg, d.h. die Übernahme einer Änderung bzw. eines Ereignisses innerhalb der Webapplikation in den JavaClient muss ebenfalls möglich sein. So soll beispielsweise die von Google Maps ermittelte Koordinate und die vollständige Adressinformation auch in der Geolocation-Eingabekomponente aktualisiert werden (vgl. Beispiel Adress-Suche mit Geolokalisierung in Kapitel 4.2.1 Seite 46).



Die FirstSpirit-Client-API (Java) kommuniziert mit der integrierten Browser-Engine über JavaScript. Die Anforderung lautet also, eine bidirektionale Kommunikation zwischen der Java- und der JavaScript-Ebene zu ermöglichen. Konkret wurden drei Möglichkeiten geschaffen, um eine bidirektionale Kommunikation einzurichten:

- 1) **JavaScript ausführen:** eine zielgerichtete, unidirektionale Kommunikation in Richtung Java » JavaScript.
- 2) **JavaScript ausführen und Rückgabewert evaluieren:** siehe oben, jedoch mit der Möglichkeit den Rückgabewert zu evaluieren.
- 3) **Java-Objekt in der JavaScript-Umgebung bereitstellen:** dient prinzipiell einer unidirektionalen Kommunikation, jedoch in Richtung JavaScript » Java.

Zu 1) Für die erste **Kommunikationsrichtung Java » JavaScript** wird eine Schnittstelle bereitgestellt, um eine JavaScript-Methode aus Java heraus aufzurufen. Dabei handelt es sich im Grunde nur um die Ausführung von JavaScript-Code in Form eines Strings. Das Interface `BrowserApplication` wurde dazu um die Methode `void executeScript(String script)` erweitert, die den übergebenen JavaScript-Code im aktuell geöffneten Browser-Dokument ausführt (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35).

Zu 2) Etwas komplizierter ist die zweite Möglichkeit. Diese beinhaltet zwar auch die Ausführung von JavaScript-Code, versucht aber den Rückgabewert auszuwerten und in geeignete Java-Objekte umzuwandeln. Das Interface `BrowserApplication` wurde dazu um die Methode `Object evaluateScript(String script)` erweitert, die den übergebenen JavaScript-Code im aktuell geöffneten Browser-Dokument ausführt und einen Rückgabewert zurückliefert. (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35). Dabei werden eine Reihe von Konvertierungsregeln auf den Rückgabewert angewendet, da JavaScript im Gegensatz zu Java nur eine eingeschränkte Menge an einfachen Datentypen unterstützt (siehe Kapitel 2.1.3 Seite 16).

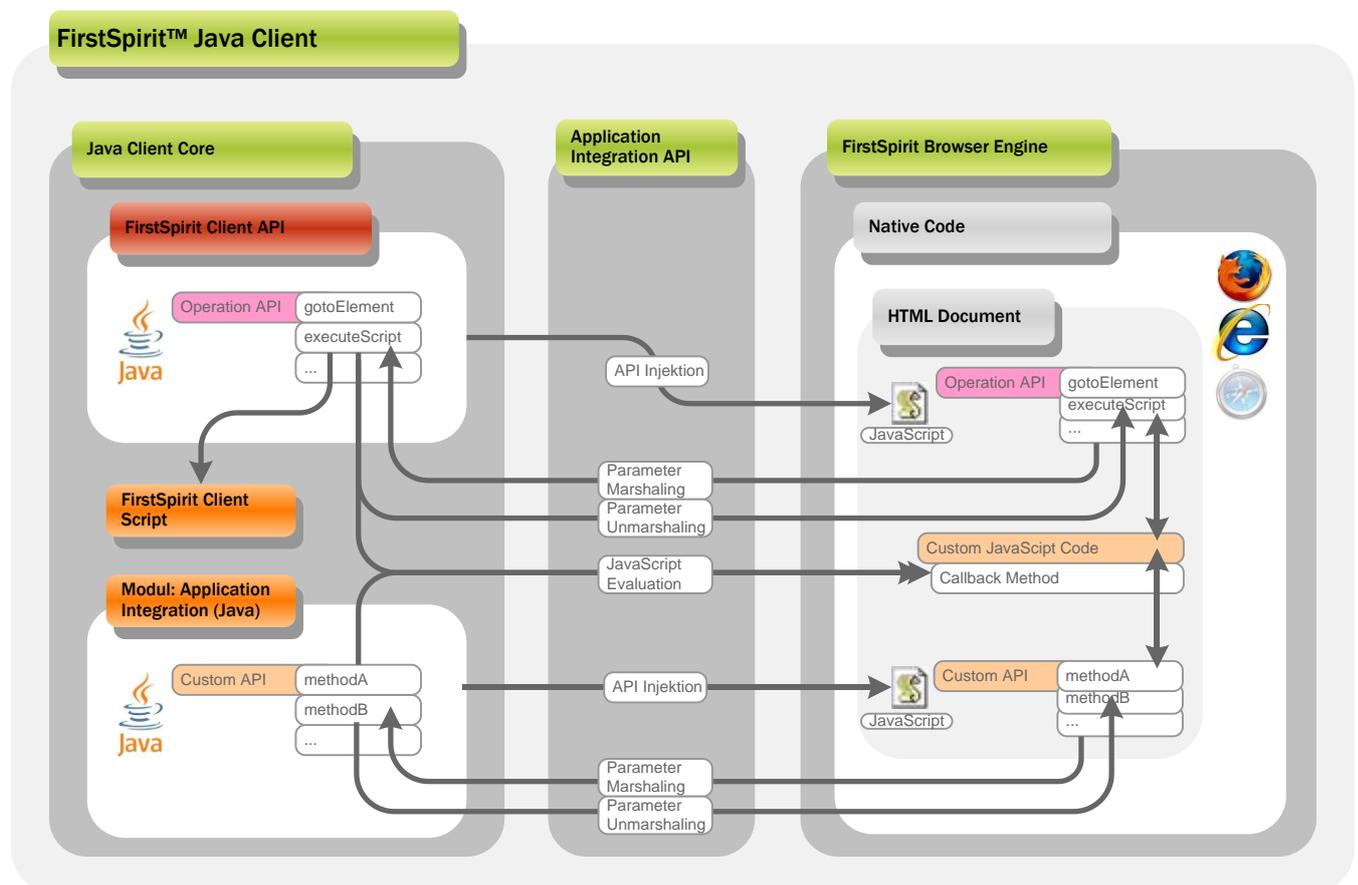
Zu 3) Für die zweite **Kommunikationsrichtung JavaScript » Java** wird eine weitere Schnittstelle bereitgestellt, um Methoden eines Java-Objekts in einem JavaScript-Umfeld zugreifbar zu machen. Konkret wird über die Injektion eines Java-Objekts in die JavaScript-Umgebung (Webbrowser) ein Stellvertreter-Objekt (Proxy) in Form eines JavaScript-Objekts erzeugt, dessen (JavaScript)-Methoden, denen des Java-Objekts entsprechen. Nach der Injektion können die entsprechenden Methoden aus dem JavaScript heraus aufgerufen werden. Die interne FirstSpirit-Implementierung erzeugt dabei für jede Methode der Java-Objekt-Instanzen eine entsprechende JavaScript-Methode. Diese Methode sendet beim Aufruf aus der JavaScript-Umgebung ein Event, welches auf der Java-Seite ausgewertet wird. Die



passende Java-Methode wird anhand der Methodensignatur und der übergebenen Parameter ermittelt und entsprechend aufgerufen.

Grundsätzlich kann ein beliebiges Java-Objekt in die JavaScript Umgebung injiziert werden, es müssen jedoch Restriktionen beachtet werden (siehe Kapitel 2.1.3, Seite 16). Die entsprechende Methode `void inject(Object object, String name)` wird über das Interface `BrowserApplikation` der FirstSpirit-AppCenter-API zur Verfügung gestellt (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35).

Eine weitere, kleine Einschränkung betrifft die Parameterübergabe. Da bei der Event-Erzeugung/Auswertung keine Synchronität gewährleistet werden kann, werden Rückgabewert der Java-Methoden per Callback zurückgegeben (siehe Kapitel 2.1.4 Seite 17).



**Abbildung 1: Kommunikation Java - JavaScript (Java-JavaScript-Bridge)**

Ein Prototyp für diese Integration wird im bereits erwähnten Beispiel zur Geolocation-Eingabekomponente vorgestellt. Dabei wird eine Aufrufchnittstelle zwischen der FirstSpirit-Client-API und einer Webapplikation, die in einer Instanz des integrierten Browsers läuft, implementiert (Implementierung: Applikationsintegration



für Google Maps siehe Kapitel 4.3 Seite 52).

### 2.1.3 Konvertierung von Datentypen

Auch in diesem Fall müssen zwei Kommunikationsrichtungen betrachtet werden:

- 1) Injektion eines Java-Objekts in die JavaScript-Umgebung: Über die API-Injektions-Schnittstelle kann grundsätzlich jedes beliebige Java-Objekt in eine JavaScript-Umgebung injiziert werden. Dabei werden alle Methoden des Java-Objektes, aber keine Attribute (z. B. name-Attribut) übernommen. Da JavaScript im Gegensatz zu Java nur eine eingeschränkte Menge an Datentypen unterstützt, gelten für die Methoden dieses Objektes außerdem gewisse Restriktionen. Es kann nur eine Reihe von einfachen, atomaren Datentypen, sowie Listen und Maps konvertiert werden. Komplexe, in JavaScript unbekannte Objekt-Typen werden dagegen nicht unterstützt. Außerdem ist die Methodenübernahme und die Abbildung der Java-Datentypen auf JavaScript-Datentypen nur auf der obersten Ebene möglich, das bedeutet:

```
MyJavaObject {  
    void helloWorld(String message);  
}
```

kann nach dem Aufruf `inject(MyJavaObject, "myObject")` auf JavaScript-Seite folgendermaßen aufgerufen werden:

```
window.myObject.helloWorld("Hallo!");
```

Eine verschachtelte Struktur ist aber nicht möglich, das bedeutet:

```
MyJavaObject {  
    MyComplexObject getMyComplexObject();  
}  
MyComplexObject {  
    void helloWorld(String message);  
}
```

kann nach dem Aufruf `inject(MyJavaObject, "myObject")` auf JavaScript-Seite nicht folgendermaßen aufgerufen werden:

```
window.myObject.getMyComplexObject().helloWorld("Hallo!");
```



- 2) Parameterübergabe aus der JavaScript- in die Java-Umgebung: Neben der Injektion eines Java-Objektes in die JavaScript-Umgebung, ist auch eine Parameterübergabe aus der JavaScript-Umgebung in die Java-Umgebung möglich, beispielsweise zur Auswertung eines Rückgabewertes. Alle aus der JavaScript-Umgebung übergebenen Parameter werden dabei analog zur bisherigen Objekt-Konvertierung in die Java-Umgebung gehoben (siehe Abbildung 1).

Außerdem gilt: Die konvertierten Java-Objekte sind lediglich Kopien der JavaScript-Objekte. Eine Änderung des Java-Objekts hat also keine Auswirkung auf die JavaScript-Umgebung.

Eine Übersicht über die mögliche Transformation der Datentypen (bzw. der Parameter) beim Wechsel zwischen der Java- und der JavaScript-Umgebung, bietet die `convertToScript(...)`-Methode des Interfaces `BrowserApplication` (siehe Kapitel 3.7 Seite 35), beispielsweise:

- `js:number` «» `Double`
- `js:boolean` «» `Boolean`
- `js:string` «» `String`
- `js:object` «» `Map<String, Object>`

#### 2.1.4 Rückgabe per Callback-Funktion

Da bei der Event-Erzeugung bzw. -Auswertung keine Synchronität gewährleistet werden kann, müssen die Rückgabewerte der Java-Methoden per Callback-Funktion zurückgegeben werden. Aus einer Java-Methode `String getName()`, wird also in der JavaScript-Umgebung die Methode `void getName(function:callback)`. Der letzte Übergabeparameter bezeichnet dabei immer die entsprechende Callback-Funktion, die aufgerufen werden soll, wenn die aufrufende Funktion abgearbeitet wurde (siehe Abbildung 1).



## 2.2 DOM-Access: Zugriff auf die Daten des integrierten Browsers

Wie bereits im Kapitel 2.1 erläutert, handelt es sich bei den integrierten Browser-Engines um eine native Implementierung, die aus der Java-Umgebung nicht ohne Weiteres zu erreichen ist. Das heißt, ein Zugriff auf die Daten des integrierten Browsers bzw. die Daten der Webapplikation (das HTML oder auch das Browser-Dokument) ist in Java zunächst einmal nicht möglich. Speziell im Rahmen einer nahtlosen Greybox-Integration ergibt sich aber die Notwendigkeit, aus der Java-Implementierung heraus auf die inneren Strukturen der Webapplikation zuzugreifen, da hier entweder gar keine API oder nur ein eingeschränkter API-Zugriff (JavaScript) bereitgestellt wird.

Die FirstSpirit-AppCenter-API wurde daher um eine Schnittstelle erweitert, die einen Zugriff auf den DOM-Baum des integrierten Browsers (Mozilla Firefox oder Internet Explorer) ermöglicht. Die Aufgabe dieser Schnittstelle ist es, einem Java-Programm (dem FirstSpirit-JavaClient oder auch einem Modul) einen lesenden und schreibenden Zugriff auf genau die Daten zu ermöglichen, die im integrierten Webbrowser aktuell dargestellt werden. Die Methode `Document getCurrentDocument()` des Interfaces `BrowserApplication` liefert das aktuelle Browser-Dokument als w3c-DOM<sup>1</sup> zurück (siehe Kapitel 3.7 Seite 35). Damit wird innerhalb der Java-Umgebung der komplette Inhalt des Webbrowsers als Dokumentenmodell zur Verfügung gestellt. Auf diesem Dokument können anschließend die HTML-Strukturen der integrierten Webapplikation durchlaufen und analysiert werden. Das der Java-Anwendung zur Verfügung gestellte Dokumentenmodell ist aber nicht auf lesenden Zugriff beschränkt, sondern kann auch manipuliert werden, wobei alle Änderungen am Dokument sofort im integrierten Browser sichtbar werden.

Die technischen Abläufe bei der Nutzung der DOM-Fassade verdeutlicht die folgende Abbildung:

---

<sup>1</sup> [http://de.wikipedia.org/wiki/Document\\_Object\\_Model](http://de.wikipedia.org/wiki/Document_Object_Model)



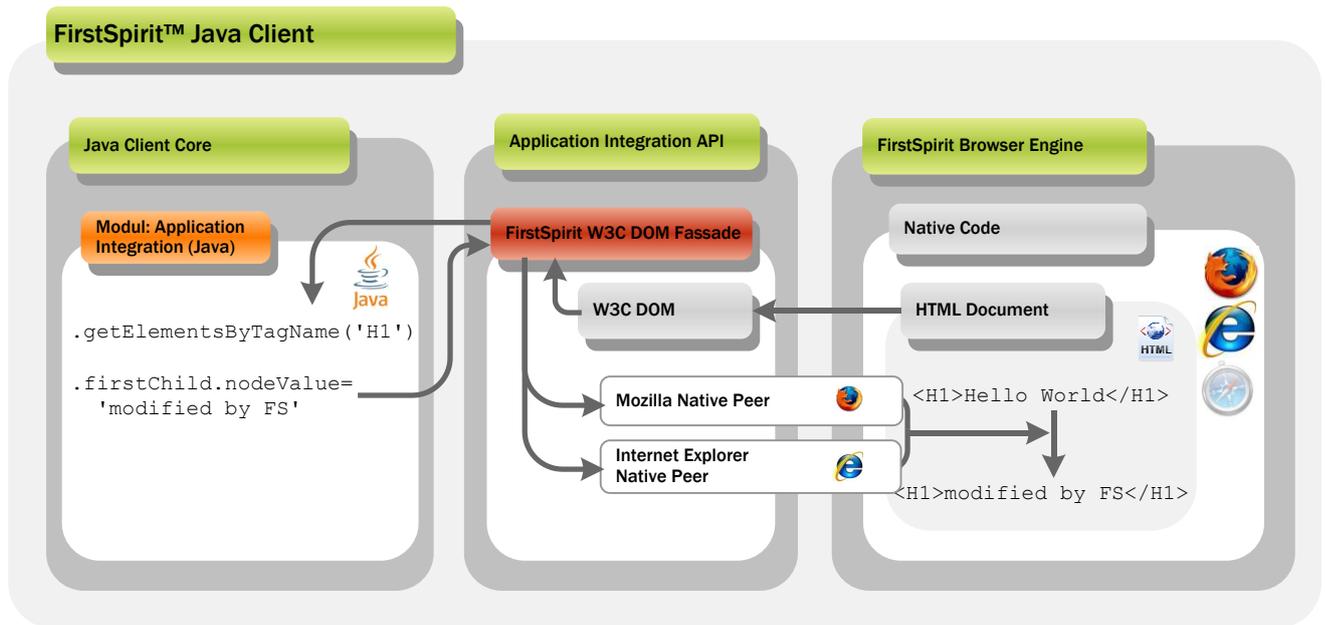


Abbildung 2: DOM-Access

Der Zugriff auf die im Webbrowser dargestellten Daten und die Manipulation dieser Daten ist über die Schnittstelle relativ einfach möglich, es gibt aber auch in diesem Fall prinzipbedingte Einschränkungen.

Das Dokumentenmodell ändert sich dynamisch. Eine Änderung innerhalb der Webapplikation wird also beispielsweise sofort im DOM nachgefahren. In der Praxis kann die Verwendung des DOM-Objekts zu vielen "nicht reproduzierbaren" Fehlern der Klasse NPE oder AIOOB führen, wenn innerhalb der Implementierung beispielsweise eine Kind-Iteration erfolgt, das betreffende Kind-Objekt aber "kurz darauf" schon gar nicht mehr existiert.



*Bei der Implementierung einer (Greybox-)Applikationsintegration für den FirstSpirit-JavaClient ist daher eine geeignete Fehlerbehandlung von zentraler Bedeutung für die Stabilität der Implementierung.*

Bei der Implementierung dieser DOM-Schnittstelle wurde besonderes Augenmerk auf den Aspekt der voll transparenten Synchronisation aller Nebenläufigkeiten gelegt, da sonst in der Verbindung nativ laufender Prozessen und verändernder Operationen Deadlock-Situationen unvermeidlich gewesen wären.

**Anmerkung:** Der DOM-Zugriff auf Flash- oder Silverlight-Applikationen ist prinzipiell NICHT möglich, da der (zugreifbare) HTML-Code nicht alle relevanten Parameter enthält und ein Einblick in die Flash-Anwendung selbst nicht möglich ist!



### 3 Standarderweiterungen

Technisch gesehen besteht das AppCenter aus einer Menge von Schnittstellen ("FirstSpirit-AppCenter-API"), die von e-Spirit für die Nutzung durch Partner freigegeben wurden, damit diese im Rahmen des AppCenters spezifische Anwendungen realisieren oder integrieren können.

Gegenwärtig beschränkt sich die FirstSpirit-AppCenter-API auf die Infrastruktur, die für die Integration von Web-Anwendungen benötigt wird. Bereits realisiert, aber (noch) nicht öffentlich verfügbar sind entsprechende Schnittstellen für die Integration von nativen Anwendungen (vgl. Integration von Microsoft Office) oder auch Java-Anwendungen (vgl. Integration zur Java-Bildbearbeitung, Java Image Editor).

Interfaces zur Konfiguration und Steuerung des Applikations-Tabs:

- Interface: ApplicationService (siehe Kapitel 3.1 Seite 21)
- Interface: ApplicationTab (siehe Kapitel 3.2 Seite 23)
- Interface: ApplicationTabAppearance (siehe Kapitel 3.3 Seite 26)
- Interface: ApplicationTabConfiguration (siehe Kapitel 3.4 Seite 30)
- Interface: TabListener (siehe Kapitel 3.5 Seite 33)

Interfaces zur Integration einer Webapplikation bzw. eines Browsers:

- Abstract Class: ApplicationType (siehe Kapitel 3.6 Seite 34)
- Interface: BrowserApplication (siehe Kapitel 3.7 Seite 35)
- Interface: BrowserListener (siehe Kapitel 3.8 Seite 38)
- Interface: BrowserApplicationConfiguration (siehe Kapitel 3.9 Seite 39)



*Alle nachfolgend vorgestellten Interfaces sind Bestandteil der FirstSpirit DEV-API. Im Gegensatz zur Access-API unterliegt die Developer-API geringeren Stabilitätsauflagen: Die Developer-API ist innerhalb einer Minor-Versionslinie stabil, d.h. dass Änderungen bei einem Minor-Versionswechsel durchgeführt werden dürfen.*



### 3.1 Interface: ApplicationService

**Package:** de.espirit.firstspirit.client.gui.applications

Einstiegspunkt für die Steuerung eines Tabs ist immer der `ApplicationService`. Dieser Service kann über unterschiedliche Broker vom FirstSpirit-Framework angefordert werden. Dazu muss zunächst über den typisierten `SwingGadgetContext` (vgl. Entwicklerhandbuch für Komponenten) eine Instanz vom Typ `SpecialistsBroker` mithilfe der Methode `SpecialistsBroker.getBroker()` angefordert werden.

```
SwingGadgetContext<...> _context;  
final SpecialistsBroker _specialistsBroker = _context.getBroker();
```

Auf dem `SpecialistsBroker` kann anschließend mithilfe der Methode `<S> S requireSpecialist(SpecialistType<S> type)` ein Spezialist vom Typ `ServicesBroker` angefordert werden. Dieser Broker liefert über den Aufruf der Methode `<T> T getService(Class<T> serviceClass)` eine Instanz vom Typ `ApplicationService` zurück (vgl. Kapitel 4.3.2).

```
final ServicesBroker servicesBroker =  
    _specialistsBroker.requireSpecialist(ServicesBroker.TYPE);  
  
final ApplicationService service =  
    servicesBroker.getService(ApplicationService.class);  
  
final BrowserApplication app =  
    service.openApplication(BrowserApplication.TYPE,  
        configuration).getApplication();
```

Über den `ApplicationService` können neue Anwendungen eines bestimmten Typs innerhalb des Applikationsbereichs geöffnet (vgl. Interface: `ApplicationTab`) oder die Anwendungen aus bereits bestehenden Browser-Instanzen geholt werden (vgl. Interface: `BrowserApplication`). Der `ApplicationService` kann nur innerhalb des FirstSpirit-JavaClients verwendet werden.

Der `ApplicationService` bietet Zugriff auf folgende Methoden:

- `ApplicationTab<T> openApplication(final ApplicationType type, final C configuration)`: Die Methode öffnet eine Anwendung eines bestimmten Typs (`ApplicationType`) in einem neuen Tab im



Applikationsbereich des FirstSpirit-JavaClients. Der Methode werden der Typ (Abstract Class: `ApplicationType`) der gewünschten Anwendung sowie die Konfiguration für den integrierten Browser übergeben (vgl. Interface: `ApplicationTabConfiguration` und Abstract Class: `ApplicationType`). Die Methode liefert eine typisierte Instanz vom Typ `ApplicationTab` zurück (Beispiel-Implementierung siehe Kapitel 4.3.3 Seite 57).

Hinweis: Für den Übergabe-Parameter `ApplicationType` gilt: FirstSpirit stellt zurzeit nur eine Schnittstelle vom Typ `BrowserApplication` bereit, über die neue Browser-Instanzen im Applikationsbereich des JavaClients geöffnet werden können (vgl. Abstract Class: `ApplicationType`).

Hinweis: Für den Übergabe-Parameter `ApplicationTabConfiguration` gilt: Soll ein `ApplicationTab` innerhalb der Implementierung wiederverwendet werden, sollte über die `ApplicationTabConfiguration` ein Identifier definiert werden (vgl. Interface: `ApplicationTabConfiguration`). Der Tab kann dann im weiteren Verlauf über die Methode `ApplicationTab<T> getApplication(final ApplicationType<T, C> type, final Object tabIdentifier)` geholt werden (s.u.).

- `ApplicationTab<T> getApplication(final ApplicationType<T, C> type, final Object tabIdentifier)`: Diese Methode liefert eine Instanz eines `ApplicationsTabs` zurück, das im Applikationsbereich des JavaClients geöffnet wurde, oder `null` falls kein `ApplicationTab` gefunden wird, das den Übergabe-Parametern entspricht. Der Methode werden der Typ (Abstract Class: `ApplicationType`) der Anwendung sowie der Identifier für das `ApplicationTab` übergeben (vgl. Interface: `ApplicationTabConfiguration`). Die Methode liefert eine typisierte Instanz vom Typ `ApplicationTab` zurück (siehe Kapitel 3.6 Seite 34).
- `boolean isVisible()`: Die Methode ermittelt, ob der Applikationsbereich im JavaClient sichtbar ist (`true`) oder nicht (`false`).
- `void setVisible(final boolean visible)`: Die Methode öffnet (`true`) oder schließt (`false`) den Applikationsbereich im JavaClient.



Folgendes Beispiel-Beanshell-Skript demonstriert den Zugriff auf den `ApplicationService` und das Öffnen eines `ApplicationTabs`.

```
import de.espirit.firstspirit.client.gui.applications.*;
import de.espirit.firstspirit.client.gui.applications.browser.*;

apps = context.connection.getService(ApplicationService.class);
tab = apps.openApplication(BrowserApplication.TYPE, "Browser");
browser = tab.getApplication();
browser.openUrl("www.e-spirit.com");
tab.setTitle("e-Spirit AG");
tab.close();
```

Beispiel zur Verwendung des `ApplicationService` siehe Kapitel 4.3.3 Seite 57.

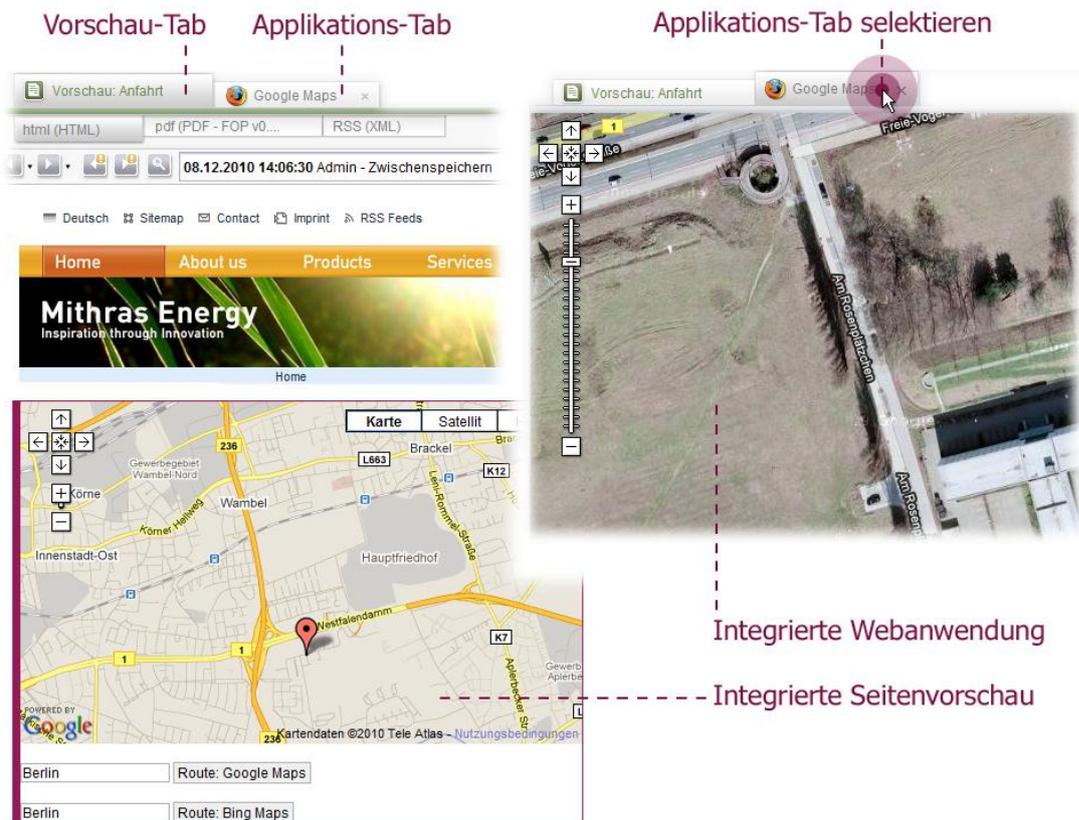
## 3.2 Interface: ApplicationTab

Package: `de.espirit.firstspirit.client.gui.applications`

Jede Seiten- bzw. Medienvorschau wird im Applikationsbereich des FirstSpirit-JavaClients in einem Vorschau-Tab dargestellt. Beim Anfordern einer neuen Vorschau (beispielsweise einer neuen Seitenansicht) wird die Ansicht des Vorschau-Tabs vom FirstSpirit-Framework automatisch aktualisiert.

Für die Integration einer Webapplikation muss zunächst ein neuer Tab im Applikationsbereich des JavaClients geöffnet werden. Dieser Applikations-Tab ist unabhängig vom eigentlichen Vorschau-Tab. Das bedeutet, wird die Webapplikation beispielsweise über eine Eingabekomponente in FirstSpirit integriert (vgl. Geolocation-Eingabekomponente), kann innerhalb des Applikationsbereichs neben der eigentlichen Seitenvorschau ein weiteres Tab mit der integrierten Webapplikation geöffnet werden.





**Abbildung 3: Vorschau-Tab und Applikations-Tab**

Den Zugriff auf das Applikations-Tab (und der darin enthaltenen Anwendungen) regelt das Interface `ApplicationTab`. Anders als das Vorschau-Tab, das durch das FirstSpirit-Framework gesteuert wird, muss die Steuerung des Applikations-Tabs vom Entwickler vorgenommen werden. Der Entwickler muss beim Anfordern der Webapplikation durch den Benutzer zunächst eine neue Browser-Instanz öffnen. Dazu wird die Methode `openApplication(...)` auf dem `ApplicationService` aufgerufen (siehe Interface: `ApplicationService`). Die Methode liefert eine Instanz vom Typ `ApplicationTab` zurück, über die die neue Instanz des integrierten Browsers gesteuert werden kann (z. B. Schließen des Tabs).

Zum Verfolgen von Änderungen innerhalb des Tabs (z. B. Selektion oder Deselektion durch den Benutzer) muss eine Instanz vom Typ `TabListener` hinzugefügt werden (vgl. Interface: `TabListener`).

Das Interface `ApplicationTab` stellt die folgenden Methoden zur Verfügung:

- `void addTabListener(@NotNull final TabListener listener):`  
Die Methode fügt einer Instanz vom Typ `ApplicationTab` einen `TabListener` hinzu. Ein `TabListener` reagiert auf Ereignisse innerhalb des Applikations-Tabs, beispielsweise das Schließen oder Deselektieren eines Tabs



durch den Benutzer (siehe dazu `Interface: TabListener` in Kapitel 3.5 Seite 33) (Beispiel siehe Kapitel 4.3.3 Seite 57).

- `void removeTabListener(@NotNull final TabListener listener)`: Die Methode entfernt einen bereits vorhandenen `TabListener`.
- `void close()`: Die Methode schließt die Instanz des `ApplicationTabs` auf der sie aufgerufen wurde.
- `boolean isClosed()`: Die Methode prüft, ob die Instanz des `ApplicationTabs` geschlossen wurde. Die Methode ist eng verbunden mit der Methode `void tabClosed()` aus dem `Interface TabListener` (siehe Kapitel 3.5 Seite 33). Beim Schließen eines Applikations-Tabs werden diese Methoden aufgerufen, um den Status "Tab wurde geschlossen" korrekt beantworten zu können. Benötigt wird diese Information beispielsweise, wenn ein bestehendes Applikations-Tab wiederverwendet werden soll. In diesem Fall muss der Entwickler entscheiden können, ob ein einmal geöffnetes `ApplicationTab` für eine neue Anfrage verfügbar ist (Tab wurde bereits geöffnet und kann für die neue Anfrage verwendet werden) oder nicht (Tab wurde geschlossen – es muss ein neuer Tab für die Anfrage geöffnet werden).
- `void setAppearance(ApplicationTabAppearance appearance)`: Die Methode beeinflusst die Darstellung des `ApplicationTabs` im Applikationsbereich. Der Methode wird eine Instanz vom Typ `ApplicationTabAppearance` übergeben, die eine Konfiguration des äußeren Erscheinungsbilds ermöglicht, beispielsweise das Hinzufügen eines Icons zum Tab (siehe Kapitel 3.3 Seite 26).
- `T getApplication()`: Diese Methode liefert die Instanz der Anwendung zurück, die in den Applikationsbereich integriert wurde. Der Rückgabewert ist typisiert (siehe Kapitel 3.6 Seite 34).
- `boolean isSelected()`: Die Methode liefert zurück, ob das `ApplicationTab` aktiv im Vordergrund erscheint (`true`) oder nur im Hintergrund geöffnet ist (`false`). Diese Information ist beispielsweise dann relevant, wenn eine Änderung an der zugehörigen `SwingGadget`-Eingabekomponente vorgenommen wird, die sich auf die integrierte Webapplikation auswirkt. Solange sich der entsprechende Tab im Hintergrund befindet, soll sich die Änderung nicht innerhalb der Anwendung auswirken.
- `void setSelected()`: Die Methode markiert die Instanz des Applikations-Tabs auf der sie aufgerufen wurde als aktiv. Das bedeutet, das entsprechende Tab wird im Applikationsbereich des `JavaClients` im Vordergrund angezeigt.

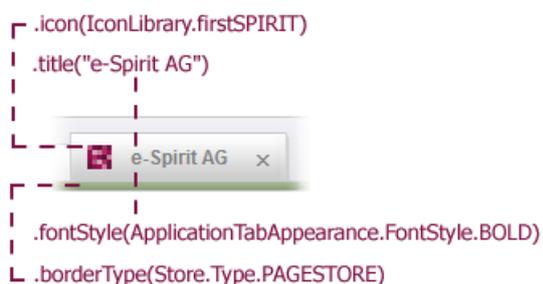
Beispiel zur Verwendung des `ApplicationService` siehe Kapitel 4.3.3 Seite 57.



### 3.3 Interface: ApplicationTabAppearance

**Package:** `de.espirit.firstspirit.client.gui.applications`

Die Darstellung eines Tabs im Applikationsbereich des FirstSpirit JavaClients wird über die Interfaces `ApplicationTabConfiguration` (siehe Kapitel 3.4 Seite 30) und `ApplicationTabAppearance` beeinflusst. Über diese Schnittstellen können beispielsweise ein Text oder ein bestimmtes Icon definiert werden, die innerhalb des Tabs angezeigt werden sollen. Darüber hinaus, bietet das Interface `ApplicationTabAppearance` weitergehende Konfigurationsmöglichkeiten, beispielsweise die Möglichkeit die Schriftstärke (plain/bold) für die Titelbeschriftung des Tabs zu ändern (siehe Abbildung 4).



**Abbildung 4: Beispiel zur ApplicationTabAppearance**

Die Methoden der beiden Interfaces überschneiden sich teilweise. So kann einem `ApplicationTab` ein Icon sowohl über die Methode `ApplicationTabConfiguration.icon(...)` als auch über die Methode `ApplicationTabAppearance.Builder icon(...)` hinzugefügt werden.

Um die Konfiguration zu vereinfachen, stellt das Interface `ApplicationTabAppearance` eine Builder-Implementierung (Interface `ApplicationTabAppearance.Builder`) mit den folgenden Methoden zur Verfügung:

- `ApplicationTabAppearance.Builder borderType(final Store.Type borderType)`: Die Leiste, die ein übergeordnetes `ApplicationTab` von seinen untergeordneten Tabs trennt, kann farbig dargestellt werden (siehe Abbildung 4). Die Darstellung ist abhängig vom Verwaltungstyp. Das Standardverhalten im JavaClient stellt die Leiste für eine Seitenvorschau (im Pagestore) beispielsweise grün dar, während sie bei einer Vorschau auf einer Seitenreferenz (im Sitestore) blau abgebildet wird. Über diese Methode kann ein Verwaltungstyp (`Store.Type`) definiert werden, um die Farbe der Leiste anzupassen. Wird kein Verwaltungstyp definiert wird eine



farblose Leiste angezeigt.

- `ApplicationTabAppearance.Builder fontStyle(final FontStyle fontStyle)`: Über diese Methode kann die Schriftstärke, die auf die Beschriftung innerhalb eines `ApplicationTabs` angewendet wird, beeinflusst werden. Dazu kann der Methode der gewünschte `ApplicationTabAppearance.FontStyle` übergeben werden. Aktuell werden die Typen `FontStyle.PLAIN` (Standardwert) und `FontStyle.BOLD` unterstützt (siehe Abbildung 4).
- `ApplicationTabAppearance.Builder icon(Icon icon)`: Über diese Methode kann ein Icon (bevorzugte Größe: 20x20 Pixel) übergeben werden, das innerhalb des `ApplicationTabs` eingeblendet wird (siehe Abbildung 4).
- `ApplicationTabAppearance.Builder title(String title)`: Über diese Methode kann ein Text übergeben werden, der als Beschriftung innerhalb des `ApplicationTabs` angezeigt wird (siehe Abbildung 4).
- `ApplicationTabAppearance get()`: Diese Methode liefert die Instanz von Typ `ApplicationTabAppearance` zurück, die auf diesem Builder aufbaut.

Eine neue Instanz vom Typ `ApplicationTabAppearance` kann über den Aufruf `ApplicationTabAppearance.GENERATOR.invoke()` erzeugt werden. Die Konfiguration erfolgt anschließend über das vereinfachte Builder-Pattern. Damit sich die geänderten Parameter hinterher auch auf die Darstellung des `ApplicationTabs` auswirken, muss die `ApplicationTabAppearance` dem `ApplicationTab` über die Methode `ApplicationTab.setAppearance(ApplicationTab Appearance appearance)` (vgl. Kapitel 3.2) übergeben werden. Beispiel:

```
...
private ApplicationTab<BrowserApplication> _tab;

final ApplicationTabAppearance appearance =
ApplicationTabAppearance.GENERATOR.invoke()
    .title("e-Spirit AG")
    .borderType(Store.Type.PAGESTORE)
    .fontStyle(ApplicationTabAppearance.FontStyle.BOLD)
    .icon(IconLibrary.firstSPIRIT)
    .get();

_tab = service.openApplication(BrowserApplication.TYPE,
(BrowserApplicationConfiguration) null);
_tab.setAppearance(appearance);
```



```
...
```

Eine Instanz vom Typ `ApplicationTabAppearance` kann auch direkt aus einer Instanz vom Typ `ApplicationTabConfiguration` über den Aufruf der Methoden `ApplicationTabConfiguration.appearance()` oder `ApplicationTabConfiguration.getAppearance()` geholt werden (siehe Kapitel 3.4). In diesem Fall ist die `ApplicationTabAppearance` ein Bestandteil der `ApplicationTabConfiguration` und kann beispielsweise einfach beim Öffnen eines neuen `ApplicationTabs` übergeben werden. Beispiel:

```
...
private ApplicationTab<BrowserApplication> _tab;

final BrowserApplicationConfiguration configuration =
    BrowserApplicationConfiguration.GENERATOR.invoke();

final ApplicationTabAppearance appearance = configuration.appearance()
    .title("e-Spirit AG")
    .borderType(Store.Type.PAGESTORE)
    .fontStyle(ApplicationTabAppearance.FontStyle.BOLD)
    .icon(IconLibrary.firstSPIRIT)
    .get();

_tab = service.openApplication(BrowserApplication.TYPE, configuration);
...
```

Zur Abfrage der Werte, die für die Darstellung eines `ApplicationTabs` konfiguriert wurden, stellt das Interface `ApplicationTabAppearance` die folgenden Methoden zur Verfügung:

- `Store.Type getBorderType()`: Diese Methode liefert den Verwaltungstyp (`Store.Type`) zurück, der zuvor über die Methode `ApplicationTabAppearance.Builder borderType(Store.Type borderType)` der Builder-Implementierung gesetzt wurde. Wurde keine `Store.Type` definiert, liefert die Methode `null` zurück.
- `FontStyle getFontStyle()`: Diese Methode liefert den Schriftschnitt zurück, der für die Beschriftung innerhalb des `ApplicationTabs` definiert wurde. Der Schriftschnitt kann über die Methode `ApplicationTabAppearance.Builder fontStyle(ApplicationTabAppearance.FontStyle fontStyle)` beeinflusst werden. Wurde kein spezieller Schriftschnitt für die Beschriftung definiert, wird eine normale



Schriftstärke verwendet (Standardwert `FontStyle.PLAIN`).

- `Icon getIcon()`: Diese Methode liefert das Icon zurück, das zuvor über die Methode `ApplicationTabAppearance.Builder icon(Icon icon)` der Builder-Implementierung oder über die Methode `ApplicationTabConfiguration.icon(Icon icon)` für die Darstellung innerhalb des `ApplicationTabs` gesetzt wurde.
- `String getTitle()`: Diese Methode liefert den Text zurück, der als Beschriftung innerhalb des `ApplicationTabs` angezeigt wird. Der Text wird über die Methode `ApplicationTabAppearance.Builder title(String title)` der Builder-Implementierung definiert.



### 3.4 Interface: ApplicationTabConfiguration

**Package:** `de.espirit.firstspirit.client.gui.applications`

Die Darstellung eines Tabs im Applikationsbereich des FirstSpirit JavaClients wird über die Interfaces `ApplicationTabConfiguration` und `ApplicationTabAppearance` (siehe Kapitel 3.3 Seite 26) beeinflusst. Über diese Schnittstellen können beispielsweise ein Text oder ein bestimmtes Icon definiert werden, die innerhalb des Tabs angezeigt werden sollen (siehe Abbildung 4).

Abhängig von der Art der Anwendung, die innerhalb eines `ApplicationTab`s geöffnet wird, bestehen weitere Konfigurationsmöglichkeiten. Das Interface `BrowserApplicationConfiguration` erweitert beispielsweise das Interface `ApplicationTabConfiguration`, um Methoden zur Konfiguration von Webapplikation. So kann innerhalb des Applikationsbereichs eine Adresszeile eingeblendet werden oder eine bestimmte Browser-Engine zum Öffnen der Webapplikation vorgegeben werden (vgl. Kapitel 3.9 Seite 39).

**Es gilt:** Das Interface `ApplicationTabConfiguration` bildet lediglich die Basisklasse. Diese Basisklasse wird durch weitere Konfigurationsschnittstellen erweitert, die genau auf den jeweiligen Anwendungstyp zugeschnitten sind. Beim Erzeugen eines neuen `ApplicationTab`s (Aufruf der Methode `ApplicationTab<T> openApplication(final ApplicationType type, final C configuration)`) wird der gewünschte Anwendungstyp übergeben. Für die Übergabe der Konfiguration wird ebenfalls eine anwendungsspezifische Instanz erwartet. Wird innerhalb des `ApplicationTab`s also eine Anwendung vom Typ `BrowserApplication` geöffnet, so muss die übergebene Konfiguration eine Instanz vom Typ `BrowserApplicationConfiguration` sein.

FirstSpirit bietet aktuell nur eine sichere Schnittstelle für den Anwendungstyp Webapplikationen (`BrowserApplication`) und die zugehörige Konfigurations-Schnittstelle `BrowserApplicationConfiguration` an. Weitere Anwendungstypen befinden sich bereits in der Entwicklung und werden voraussichtlich mit FirstSpirit Version 5 freigegeben (vgl. Kapitel 3.6)

Das Interface `ApplicationTabConfiguration` stellt die folgenden Methoden zur Verfügung:

- `ApplicationTabAppearance.Builder appearance()`: Diese Methode liefert eine Builder-Instanz vom Typ `ApplicationTabAppearance.Builder` zurück, die weitere Möglichkeiten zur Konfiguration der Tab-Darstellung bietet



(siehe Kapitel 3.3 Seite 26).

- `ApplicationTabAppearance` `getAppearance()`: Diese Methode liefert eine Instanz vom Typ `ApplicationTabAppearance` zurück, die weitere Möglichkeiten zur Konfiguration der Tab-Darstellung bietet (siehe Kapitel 3.3 Seite 26). Es wird empfohlen, an dieser Stelle die vereinfachte Builder-Implementierung des Interfaces `ApplicationTabAppearance` zu verwenden (s.o.).
- `Object` `getIdentifizier()`: Diese Methode liefert den Identifizier zurück, der über die `ApplicationTabConfiguration` mithilfe der Methode `public T identifizier(final Object tabIdentifizier)` für eine Instanz vom Typ `ApplicationTab` definiert wurde (s.u.). Wurde kein konkreter Identifizier festgelegt, liefert die Methode einen String zurück, der aus dem Präfix "BrowserApplication\_" und der Systemzeit gebildet wird. Ist der Identifizier bekannt, kann der zugehörige `ApplicationTab` im weiteren Verlauf über den `ApplicationService` mithilfe der Methode `ApplicationTab<T> getApplication(final ApplicationType<T, C> type, final Object tabIdentifizier)` geholt werden (Interface `ApplicationService` siehe Kapitel 3.1 Seite 21).
- `public T identifizier(final Object tabIdentifizier)`: Soll ein `ApplicationTab` innerhalb der Implementierung wiederverwendet werden, kann über diese Methode ein Identifizier vergeben werden. Mithilfe des Identifiziers kann das zugehörige `ApplicationTab` im weiteren Verlauf über den `ApplicationService` mithilfe der Methode `ApplicationTab<T> getApplication(final ApplicationType<T, C> type, final Object tabIdentifizier)` geholt werden (Interface `ApplicationService` siehe Kapitel 3.1 Seite 21).
- `public T icon(final Icon icon)`: Über diese Methode kann ein Icon (bevorzugte Größe: 20x20 Pixel) übergeben werden, das innerhalb des `ApplicationTabs` eingeblendet wird (siehe Abbildung 4) (siehe auch Kapitel 3.3 Seite 26).
- `public T openInBackground(boolean openInBackground)`: Über diese Methode kann beeinflusst werden, ob das `ApplicationTab`, das auf dieser Konfiguration basiert, im Hintergrund geöffnet wird (`true`) oder nicht (`false`).
- `boolean openInBackground()`: Die Methode liefert zurück, ob das `ApplicationTab`, das auf dieser Konfiguration basiert, aktiv im Vordergrund



(`false`) oder nur im Hintergrund geöffnet werden soll (`true`).

- `public T title(final String title)`: Über diese Methode kann ein Text übergeben werden, der als Beschriftung innerhalb des `ApplicationTabs` angezeigt wird (siehe Abbildung 4) (siehe auch Kapitel 3.3 Seite 26).

Beispiel siehe Kapitel 3.9 Seite 39.



### 3.5 Interface: TabListener

Package: `de.espirit.firstspirit.client.gui.applications`

Für die Integration einer Webapplikation in FirstSpirit wird ein neuer Tab im Applikationsbereich des JavaClients geöffnet. Die Steuerung dieses Applikations-Tabs muss (anders als beim herkömmlichen Vorschau-Tab) vom Entwickler übernommen werden. Die dazu erforderlichen Methoden sind im Interface `ApplicationTab` beschrieben (siehe Kapitel 3.2 Seite 23).

Die Steuerung des Tabs kann über Ereignisse erfolgen. Um auf interne oder externe Ereignisse reagieren zu können, muss eine Instanz vom Typ `TabListener` auf dem Applikations-Tab registriert werden. Dieser Listener beinhaltet Methoden, die den Entwickler über Ereignisse informieren, die auf dem Tabs im Applikationsbereich stattfinden. Wird beispielsweise ein Applikations-Tab von Benutzer geschlossen, reagiert das FirstSpirit-Framework mit dem Aufruf der Methode `void tabClosed()`.

Um eine neue Instanz vom Typ `TabListener` zu erzeugen, kann das Interface (und alle im Interface enthaltenen Methoden) implementiert werden. Es wird aber empfohlen, stattdessen die interne, abstrakte Adapter-Implementierung (Abstract Adapter Class) zu verwenden, die vom Interface `TabListener` zur Verfügung gestellt wird. Diese vordefinierte Klasse implementiert alle Methoden des Interfaces. Der Entwickler muss nur noch die für ihn relevanten Methoden implementieren (siehe Listener – Auf Änderungen reagieren, Kapitel 4.3.7, Seite 77) und kann ansonsten auf die vorhandene Standard-Implementierung zurückgreifen. Auch für eine spätere Erweiterung der Schnittstelle ist die Adapter-Klasse vorteilhaft. Wird das Interface beispielsweise um eine neue Methode ergänzt, bleiben bestehende Implementierungen kompatibel. Die neue Methode muss vom Entwickler nur bei Bedarf implementiert werden.

Eine Instanz vom Typ `TabListener` muss anschließend über den Aufruf der Methode `addTabListener(...)` auf der Ereignisquelle (hier dem Applikations-Tab) registriert werden.

Das Interface `TabListener` (bzw. die zugehörige Adapter-Implementierung) stellt die folgenden Methoden zur Verfügung:

- `void tabSelected()`: Diese Methode wird aufgerufen, wenn der zugehörige `ApplicationTab` selektiert ist, d.h. sichtbar im Vordergrund angezeigt wird. Die Methode ist eng verknüpft mit der Methode `void setSelected()` aus dem



Interface `ApplicationTab` (siehe Kapitel 3.2 Seite 23).

- `void tabDeselected()`: Diese Methode wird aufgerufen, wenn ein neues Tab (Applikations- oder Vorschau-Tab) in den Vordergrund geholt wird. In diesem Fall rückt das zugehörige `ApplicationTab` in den Hintergrund. Soll beispielsweise die Änderung innerhalb einer Eingabekomponente nur nachgefahren werden, wenn das betreffende `ApplicationTab` aktiv im Vordergrund angezeigt wird, kann das über die Methode `void tabDeselected()` realisiert werden.
- `void tabClosed()`: Diese Methode wird vom FirstSpirit-Framework aufgerufen, wenn die Instanz eines `ApplicationTabs` geschlossen wird. Ob ein Tab bereits geschlossen wurde, lässt sich durch den Aufruf der Methode `boolean isClosed()` des Interfaces `ApplicationTab` (siehe Kapitel 3.5 Seite 33) ermitteln.

### 3.6 Abstract Class: `ApplicationType`

Package: `de.espirit.firstspirit.client.gui.applications`

Beim Öffnen einer neuen Anwendung über den `ApplicationService` wird ein bestimmter Anwendungstyp übergeben (vgl. Kapitel 3.1 Seite 21) für Webapplikationen beispielsweise:

- `BrowserApplication`: Schnittstelle zum Öffnen und Steuern einer neuen Browser-Instanz im Applikationsbereich des FirstSpirit JavaClients (siehe Kapitel 3.7 Seite 35).

Weitere Anwendungstypen, beispielsweise eine Schnittstelle zur Integration von Swingapplikationen, siehe FirstSpirit-AppCenter-API.

Die abstrakte Klasse stellt folgende Methoden zur Verfügung:

- `String name()`: Die Methode liefert den vollständigen Namen des jeweiligen `ApplicationTypes` zurück.



### 3.7 Interface: BrowserApplication

**Package:** `de.espirit.firstspirit.client.gui.applications.browser`

Um Webapplikationen in den Applikationsbereichs des JavaClients zu integrieren, wird ein Zugriff auf den integrierten Browser von FirstSpirit benötigt. Das Interface `BrowserApplication` bietet Methoden an, um eine neue Instanz des jeweiligen Browsers zu erzeugen und zu kontrollieren. Viele der enthaltenen Methoden werden asynchron ausgeführt. Damit ein geordneter Zugriff auf die Inhalte der Webapplikation möglich ist, sollte eine Instanz vom Typ `BrowserListener` auf der `BrowserApplication` registriert werden.

Das Interface bietet den Zugriff auf folgende Methoden:

- `EngineType getEngineType()`: Diese Methode liefert den aktuellen `EngineType` des Browsers zurück. FirstSpirit integriert aktuell zwei Webbrowser-Engines, die wahlweise verwendet werden können - Mozilla Firefox und Microsoft Internet Explorer. Die gewünschte Brower-Engine kann über die Konfiguration (siehe Kapitel 3.9 Seite 39) ausgewählt werden. Neben einer starren Definition kann hier durch Angabe von `BrowserApplicationConfiguration.GENERATOR.invoke().engineType(EngineType.DEFAULT)` auch die Standard-Browser-Engine angegeben werden, die vom jeweiligen Benutzer im FirstSpirit JavaClient hinterlegt wurden (JavaClient Menüleiste: Menüpunkt Ansicht – Browser Engine). Die Methode `getEngineType()` liefert in diesem Fall den entsprechenden, konkreten Typ zurück.
- `BrowserApplication.getEngineVersion()`: Diese Methode liefert die aktuelle Version des Browsers als String zurück. Anmerkung: Dies ist im "Mozilla Firefox"-Fall nicht die Firefox-Version, sondern die Xulrunner-Version. Eine Zuordnung zur Firefox-Version muss hier eigenständig erfolgen (falls notwendig).
- `void openUrl(final String url)`: Die Methode öffnet die übergebene URL innerhalb einer Browser-Instanz im Applikationsbereich des JavaClients. Dabei kann es sich entweder um die URL einer externen Webapplikation handeln oder um eine kundenspezifische Implementierung, die zunächst global auf dem FirstSpirit-Server installiert werden muss und anschließend über die Methode `openUrl(...)` im Applikationsbereich geöffnet werden kann (vgl. Kapitel 4.3.3 Seite 57). Diese Methode wird asynchron ausgeführt. Um festzustellen zu welchem Zeitpunkt die Methode ausgeführt wird, muss ein `BrowserListener` registriert werden. Dieser informiert den Benutzer zum Ausführungszeitpunkt. Darüber, dass sich die Location geändert hat.



- `void openUrl(final Location location):`
- `String getUrl():`
- `void addBrowserListener(@NotNull final BrowserListener listener):` Die Methode registriert einen `BrowserListener` auf einer Instanz vom Typ `BrowserApplication`. Ein `BrowserListener` reagiert auf Änderungen oder Ereignisse innerhalb der Webapplikation (Interface: `BrowserListener` siehe Kapitel 3.8 Seite 38, Beispiel-Implementierung siehe Kapitel 4.3.7.2 Seite 80).
- `String convertToScript(Object object):` Die Methode konvertiert das übergebene Java-Objekt in JavaScript-Code und liefert diesen als String zurück. Da JavaScript im Gegensatz zu Java nur eine eingeschränkte Menge an Datentypen unterstützt, gelten für die Methoden dieses Objektes außerdem gewissen Restriktionen. Es kann nur eine Reihe von einfachen, atomaren Datentypen, sowie Listen und Maps konvertiert werden. Komplexe, in JavaScript unbekannte Objekt-Typen werden dagegen nicht unterstützt. Ist ein übergebenes Java-Objekt `null` oder wird nicht unterstützt, liefert die Methode `null` zurück :

Java	JavaScript
▪ <code>Number, Boolean</code>	▪ <i>(related toString mechanism)</i>
▪ <code>String</code>	▪ <code>"stringcontent"</code> ( <i>escapes newline and "</i> )
▪ <code>List&lt;Object&gt;</code>	▪ <code>[entry0,entry1,entry2,...]</code>
▪ <code>Map&lt;String,Object&gt;</code>	▪ <code>{'key0':value0,'key1':value1,...}</code>

Weitere Informationen zur Konvertierung von Datentypen siehe Kapitel 2.1.3 Seite 16).

- `<T> BrowserNodeHandlerBuilder<T> createNodeHandlerBuilder():`
- `void executeScript(String script):` Diese Methode führt den übergebenen JavaScript-Code im aktuell geöffneten Browser-Dokument aus und ermöglicht so eine zielgerichtete, unidirektionale Kommunikation in Richtung Java » JavaScript. Dabei wird der auszuführende JavaScript-Code einfach als String übergeben (Beispiel siehe Kapitel 4.3.4 Seite 61).
- `Object evaluateScript(String script):` Diese Methode führt den übergebenen JavaScript-Code im aktuell geöffneten Browser-Dokument aus, ermöglicht also ebenfalls eine zielgerichtete, unidirektionale Kommunikation in Richtung Java » JavaScript, liefert aber außerdem noch einen Rückgabewert zurück. Die aus der JavaScript-Umgebung übergebenen Rückgabewerte werden dabei nach bestimmten Konvertierungsregeln in Java-Objekte umgewandelt, so wird beispielsweise aus einem `js:number`-Objekt ein Objekt vom Typ `Double` (Konvertierung von Datentypen siehe Kapitel 2.1.3 Seite 16).



- `void removeBrowserListener(@NotNull final BrowserListener listener)`
- `void focus()`: Durch den Aufruf dieser Methode bekommt die aktuelle Browser-Instanz den Fokus. Nützlich ist dies beispielsweise, wenn die integrierte Webanwendung ein Formularelement enthält, das direkt einen Eingabe-Cursor erhalten soll oder um, wie im Beispiel der Google Maps-Integration, ein direktes Zoomen per Mausrad zu ermöglichen, wenn das Applikations-Tab vom Redakteur selektiert wird (vgl. Beispiel in Kapitel 4.3.7.1).
- `void setHtmlContent(String html)`
- `Document getCurrentDocument()`
- `void inject(Object object, String name)`: Die Methode wird für die Kommunikation zwischen der Java-Ebene des FirstSpirit-JavaClients und der JavaScript-Ebene der Webapplikation benötigt. Die Methode injiziert das übergebene Java-Objekt als Attribut des Window-Objekts in die Browser-Instanz, auf der es aufgerufen wurde (Informationen zum Objekt `window` siehe Kapitel 4.3.13). Durch die Injektion wird ein Stellvertreter-Objekt (Proxy) in Form eines JavaScript-Objektes erzeugt und unter dem übergebenen Namen registriert (Beispiel siehe Kapitel 4.3.5 Seite 65).

Ein Zugriff auf den DOM-Baum der Browser-Instanz ist nicht zu jedem Zeitpunkt möglich. Die Registrierung kann also nur erfolgen, wenn das Dokument vollständig geladen wurde. Um dies sicherzustellen, muss eine Instanz vom Typ `BrowserListener` verwendet werden (vgl. Interface: `BrowserListener`)(Konzept DOM-Zugriff siehe Kapitel 2.2 Seite 18).

Nach der Registrierung kann das JavaScript-Objekt über den Aufruf `window.{name}` in der JavaScript-Umgebung verwendet werden. Alle Methoden des Java-Objekts können anschließend ebenfalls aus der JavaScript-Umgebung des integrierten Browsers heraus aufgerufen werden.

**Hintergrund:** Das FirstSpirit-Framework erzeugt bei der Injektion für jede Methode der Java-Objekt-Instanzen eine entsprechende JavaScript-Methode mit (annähernd) identischer Methoden-Signatur. Diese JavaScript-Methode sendet beim Aufruf aus der JavaScript-Umgebung ein Event, welches auf der Java-Seite ausgewertet wird und dort die Ausführung der zugehörigen Java-Methode auslöst. Die passende Java-Methode wird anhand der Methodensignatur und der übergebenen Parameter ermittelt und aufgerufen.

Beispiel (Erzeugen einer Instanz vom Typ `BrowserApplication`):

```
ApplicationService appService =
servicesBroker.getService(ApplicationService.class);
```



```
BrowserApplication browser =
    appService.openApplication(BrowserApplication.TYPE,
        null).getApplication();
browser.openUrl("www.e-spirit.de");
```

### 3.8 Interface: BrowserListener

**Package:** `de.espirit.firstspirit.client.gui.applications.browser`

Mithilfe eines `BrowserListeners` kann der Zugriff auf die Inhalte der `Browser`-Instanz über Ereignisse gesteuert werden. Dazu muss eine Instanz vom Typ `BrowserListener` auf der `Browser`-Instanz (`BrowserApplication`) registriert werden. Eine Instanz vom Typ `BrowserListener` beinhaltet Methoden, die den Entwickler über Änderungen der `Browser`-Instanz im Applikationsbereich informieren. Wird beispielsweise der URL der `Browser`-Instanz geändert, reagiert das `FirstSpirit`-Framework mit dem Aufruf der Methode `void onLocationChange(@NotNull String url)`.

Um eine neue Instanz vom Typ `BrowserListener` zu erzeugen kann das Interface (und alle im Interface enthaltenen Methoden) implementiert werden. Es wird aber empfohlen, stattdessen die interne, abstrakte Adapter-Implementierung (`Abstract Adapter Class`) zu verwenden, die vom Interface `BrowserListener` zur Verfügung gestellt wird. Diese vordefinierte Klasse implementiert alle Methoden des Interfaces. Der Entwickler muss dann nur noch die für ihn relevanten Methoden implementieren (siehe `Listener – Auf Änderungen reagieren`, Kapitel 4.3.7, Seite 77) und kann ansonsten auf die vorhandene Standard-Implementierung zurückgreifen. Auch für eine spätere Erweiterung der Schnittstelle ist die Adapter-Klasse vorteilhaft. Wird das Interface beispielsweise um eine neue Methode ergänzt, bleiben bestehende Implementierungen kompatibel. Die neue Methode muss vom Entwickler nur bei Bedarf implementiert werden.

Eine Instanz vom Typ `BrowserListener` muss anschließend über den Aufruf der Methode `addBrowserListener(...)` auf der Ereignisquelle (hier der `BrowserApplication`) registriert werden.

Das Interface `BrowserListener` (bzw. die zugehörige Adapter-Implementierung) stellt die folgenden Methoden zur Verfügung:

- `void onLocationChange(@NotNull String url)`: Die Methode wird aufgerufen, wenn der `BrowserListener` meldet, dass sich die URL der `Browser`-Instanz (Instanz vom Typ `BrowserApplication`) geändert hat.

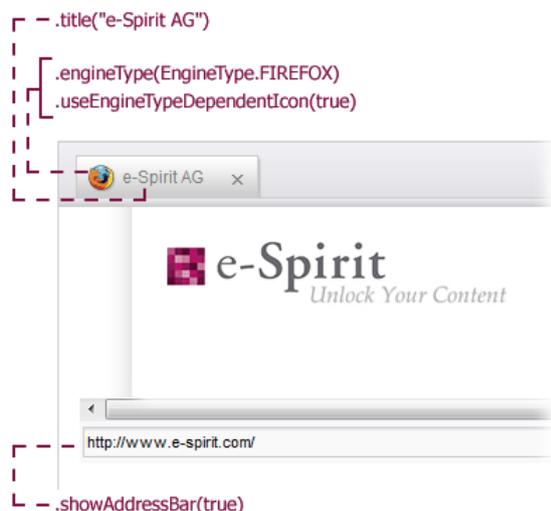


- `void onDocumentComplete(String url)`: Die Methode `void onDocumentComplete(...)` wird aufgerufen, wenn der `BrowserListener` der `Browser-Instanz` (Instanz vom Typ `BrowserApplication`) meldet, dass das Dokument (einschließlich aller Bilder) vollständig geladen wurde (Konzept DOM-Zugriff siehe Kapitel 2.2 Seite 18).

### 3.9 Interface: `BrowserApplicationConfiguration`

**Package:** `de.espirit.firstspirit.client.gui.applications`

Das Interface `BrowserApplicationConfiguration` erweitert die Basisklasse `ApplicationTabConfiguration` (siehe Kapitel 3.4 Seite 30) um Konfigurationsmöglichkeiten für die Darstellung von `BrowserApplications` innerhalb eines `ApplicationTabs`. So kann beispielsweise eine bestimmte `Browser-Engine` für das Öffnen der Webapplikation definiert oder eine Adresszeile zur Anzeige der aufgerufenen URL im Applikationsbereich eingeblendet werden.



**Abbildung 5: Beispiel zur `BrowserApplicationConfiguration`**

Das Interface `BrowserApplicationConfiguration` leitet von der Basisklasse `ApplicationTabConfiguration` ab und stellt damit alle in Kapitel 3.4 beschriebene Methoden zur Verfügung. Des Weiteren beinhaltet das Interface `BrowserApplicationConfiguration` die folgenden Methoden:

- `BrowserApplicationConfiguration useEngineTypeDependentIcon(boolean useEngineTypeDependentIcon)`: Wenn kein spezifisches Icon für die Tab-Darstellung definiert wurde, kann über diese Methode, das Icon der `BrowserEngine` eingeblendet werden.



- `BrowserApplicationConfiguration` `showAddressBar`(final boolean showAddressBar): Über diese Methode kann definiert werden, ob ein Adressfeld mit der aufgerufenen URL im unteren Bereich des Applikationsbereichs angezeigt werden soll (Standardwert: `true`) oder nicht (`false`). Diese Methode kann beispielsweise eingesetzt werden, um die URL einer externen oder auf dem FirstSpirit-Server installierten Webapplikation einzublenden, die unter Verwendung der Methode `openUrl(...)` im Applikationsbereich geöffnet wurde (vgl. Kapitel 3.7 Seite 35). Wird hingegen ein HTML-Code initiiert (unter Verwendung der Methode `setHtmlContent(...)`, vgl. Beispiel Google Maps) kann die Anzeige eines Adressfelds unterdrückt werden.
- `boolean showAddressBar()`: Diese Methode liefert zurück, ob für das `ApplicationTab`, das basierend auf dieser Konfiguration geöffnet wurde, ein Adressfeld eingeblendet wird (`true`) oder nicht (`false`) (vgl. Methode `BrowserApplicationConfiguration` `showAddressBar`(final boolean showAddressBar)).
- `public BrowserApplicationConfiguration engineType(@NotNull final EngineType type)`: Über diese Methode kann eine Browser-Engine definiert werden, die für das Öffnen der Webapplikation im Applikationsbereich verwendet werden soll. FirstSpirit integriert aktuell zwei Webbrowser-Engines, die wahlweise verwendet werden können - Mozilla Firefox und Microsoft Internet Explorer. Neben einer starren Definition kann hier durch Angabe von `BrowserApplicationConfiguration.GENERATOR.invoke().engineType(EngineType.DEFAULT)` auch die Standard-Browser-Engine angegeben werden, die vom jeweiligen Benutzer im FirstSpirit JavaClient hinterlegt wurden (JavaClient Menüleiste: Menüpunkt Ansicht – Browser Engine).
- `public EngineType getEngineType()`: Die Methode liefert den `EngineType` zurück, der zuvor über die Methode `BrowserApplicationConfiguration.engineType(@NotNull final EngineType type)` für das Öffnen der Webapplikation im Applikationsbereich definiert wurde. Anmerkung: Wurde statt eines konkreten Typs eine Standard-Browser-Engine definiert (`EngineType.DEFAULT`), liefert diese Methode den `EngineType DEFAULT` zurück. Um den jeweiligen konkreten Typ zu erhalten, kann die Methode `BrowserApplication.getEngineType()` eingesetzt werden (siehe Kapitel 3.7 Seite 35).

Eine neue Instanz vom Typ `BrowserApplicationConfiguration` kann über den Aufruf `BrowserApplicationConfiguration.GENERATOR.invoke()` erzeugt werden. Die Konfiguration erfolgt anschließend über das vereinfachte



Builder-Pattern. Die anwendungsspezifische Konfiguration wird beim Aufruf der Methode `ApplicationTab<T> openApplication(final ApplicationType type, final C configuration)` übergeben und muss zum übergebenen Anwendungstyp (hier: `BrowserApplication`) passen.

Beispiel:

```
final ApplicationService service =
    _servicesBroker.getService(ApplicationService.class);

final BrowserApplicationConfiguration configuration =
    BrowserApplicationConfiguration.GENERATOR.invoke()
        .icon(IconLibrary.firstSPIRIT)
        .title("e-Spirit AG")
        .identifier("test")
        .engineType(EngineType.FIREFOX)
        .showAddressBar(true)
        .useEngineTypeDependentIcon(true)
        .openInBackground(false);
```

### 3.10 Interface: ClientServiceRegistryAgent

Package: `de.espirit.firstspirit.agency`



## 4 Beispiel: Integration von Google Maps in FirstSpirit

Im Bereich der Presentation-Layer-Integration haben sich Anwendungen für Anfahrtsbeschreibungen, die mit geographischen Koordinaten (geographische Breite und geographische Länge) arbeiten, wie Lagepläne und Routenplaner, auf den Webseiten der meisten Unternehmen fest etabliert. In der Regel haben diese Anwendungen das Problem, dass zunächst nur die Adresse, nicht aber die durchaus relevante geographische Koordinate bekannt ist.

Die in diesem Kapitel vorgestellte Beispiel-Implementierung soll eine einfache und intuitive Möglichkeit schaffen, um innerhalb der FirstSpirit-Redaktionsumgebung mit geographischen Koordinaten zu arbeiten. Dazu wird eine Geolocation-Eingabekomponente entwickelt, die zu einer Adresse, alle relevanten, geographischen Informationen ermittelt und diese zur weiteren Bearbeitung speichert. Die Komponente verwendet dazu die Webapplikation Google Maps, die nahtlos in den Applikationsbereich der FirstSpirit-Redaktionsumgebung integriert wird. Suchanfragen können direkt von der Eingabekomponente an die Google Maps-API weitergeleitet werden, es ist aber auch eine Suche über die Google Maps Kartendarstellung im Applikationsbereich möglich. Die ermittelten geographischen Informationen werden anschließend zur Weiterverarbeitung innerhalb der Eingabekomponente gespeichert.

Im Beispiel soll eine enge Integration der Webapplikation mit dem FirstSpirit-JavaClient realisiert werden, die beispielsweise auch ein Drag & Drop von Google Maps Objekten in die Geolocation-Eingabekomponente erlaubt.

- **Ziel:** Einführung einer einfach zu handhabenden Lösung für geographische Koordinaten.
- **Technik:** Webapplikationsintegration, Greybox-Technik auf HTML-Basis (vgl. Kapitel 1.3 Seite 6).



## 4.1 Erste Schritte

### 4.1.1 Hinweis auf das FirstSpirit-Lizenzmodell

Die Verwendung des FirstSpirit AppCenters unterliegt einem neuen Lizenzmodell. Im Unterschied zur bisherigen Lizenzierung einer FirstSpirit-(Modul)-Erweiterung, wird nicht die Funktionalität lizenziert, sondern die Anzahl der integrierten Applikationen (siehe Kapitel 1.6 Seite 9). Eine Beispiel-Implementierung kann zu Test- und Demozwecken und jedoch ohne gültige Lizenz auf dem FirstSpirit-Server installiert werden.

### 4.1.2 Hinweis auf rechtliche Implikation

Die hier vorgestellten Beispiel-Implementierungen für die Integration von Google Maps (bzw. für die Integration einer Bilddatenbank) stellen keine FirstSpirit-Standardfunktionalität dar (siehe Kapitel 1.4 Seite 8). Die Implementierung soll lediglich exemplarisch aufzeigen, welche Möglichkeiten die Applikations-Integration in FirstSpirit bietet und wie diese realisiert werden können.



*Soll eine Applikations-Integration (beispielsweise für Google Maps) innerhalb eines Projekts realisiert werden, müssen die erforderlichen Lizenzen für die Verwendung direkt beim Hersteller der integrierten Anwendung angefordert werden. Insbesondere die Verwendung der Google-Technologie unterliegt hier starken Einschränkungen (siehe Nutzungsbestimmungen bei der Erstellung eines Google-Kontos).*

### 4.1.3 Google Maps-API-Schlüssel generieren

Um Google Maps auf Ihrer Website nutzen zu können, wird die Google Maps-API benötigt. Um diese verwenden zu können, ist wiederum ein Google Maps-API-Schlüssel erforderlich. Dieser muss direkt bei Google angefordert werden. Ein Google Maps-API-Schlüssel ist dann für ein "Verzeichnis" oder eine Domain gültig. D.h. ein Schlüssel kann für folgende URLs eingesetzt werden:

- <http://www.meinserver.com/maps/index.php>
- <http://www.meinserver.com/maps/map.html>

nicht aber z. B. für

- <http://subdomain.meinserver.com/index.html>



In der Regel empfiehlt sich die Registrierung des Domainnamens. Der Schlüssel ist dann für diese Domain, ihre Sub-Domains, alle URLs von Hosts in diesen Domains sowie sämtliche Ports auf diesen Hosts gültig<sup>2</sup>.

Hinweis: Sofern Google Maps auch in der Vorschau des Redaktionssystems verwendet werden soll, muss der FirstSpirit-Server in der gleichen Domain wie die hier registrierte Domain betrieben werden. Die Startseite des FirstSpirit-Servers muss also in der Domain des registrierten API-Schlüssels liegen (Hinweis zur Konfiguration des FirstSpirit-Servers siehe Kapitel 4.1.4).

Zunächst muss ein Konto (Account) bei Google vorhanden sein. Der API-Schlüssel, der im nächsten Schritt angefordert wird, ist an dieses Google-Konto gekoppelt. Falls noch kein Google-Konto vorhanden ist, kann es über folgende URL erstellt werden:

<https://www.google.com/accounts/NewAccount>

Für ein Konto können mehrere API-Schlüssel angefordert werden.

Der API-Schlüssel wird über folgende URL angefordert:

<http://www.google.com/apis/maps/signup.html>

Dazu wird die URL, für die der Google Maps-Dienst gelten soll, in das Feld "URL meiner Website" eingetragen. Wenn den Nutzungsbestimmungen zugestimmt und der Button "API-Schlüssel generieren" angeklickt wird, wird im nächsten Fenster der Schlüssel angezeigt.

#### 4.1.4 Hinweis zur Konfiguration des FirstSpirit-Servers

Der Google Maps API Schlüssel wird nur für eine Domain (nicht für einen Hostnamen) registriert (vgl. Kapitel 4.1.3).

Um zu vermeiden, dass FirstSpirit über URLs verwendet wird, die nicht in der registrierten Domain liegen, also beispielsweise über `http://fs4server` statt `http://fs4server.meinserver.com`, sollte der externe Apache httpd, der ergänzend zum integrierten Jetty-Webserver für FirstSpirit verwendet werden kann, folgendermaßen konfiguriert werden:

```
RewriteCond %{HTTP_HOST} !^hostname\.domain$ [nocase]
RewriteRule ^/(.*) http://hostname.domain/$1 [redirect=permanent,noescape,last]
```

Alle Aufrufe der FirstSpirit-Startseite werden anschließend auf eine definierte URL (mit Domain) umgeleitet.

---

<sup>2</sup> Für detaillierte Informationen zur Gültigkeit eines Google Maps-API-Schlüssels siehe auch

<http://code.google.com/intl/de/apis/maps/faq.html#keyssystem>



#### 4.1.5 Installation des Google Earth Plug-ins

Zur Verwendung der 3D-Darstellung von Google Earth im integrierten Applikationsbereich von FirstSpirit (vgl. Kapitel 4.2.4) muss das Google Earth Plug-in installiert werden. Fehlt das Plug-in, erscheint innerhalb des Applikationsbereichs im FirstSpirit-JavaClient eine Seite, die zum Download des Plug-ins auffordert. Klicken Sie auf den Button "Google Earth Plug-in herunterladen", wird das Plug-in direkt auf Ihrem Arbeitsplatzrechner abgespeichert. Zum Installieren öffnen Sie die betreffende Datei `GoogleEarthPluginSetup.exe` durch Doppelklick. Folgen Sie den Anweisungen des Installations-Programms<sup>3</sup>.

Nach der Installation ist die Google Earth Ansicht innerhalb des Applikationsbereichs sichtbar.

#### 4.1.6 Beispiel-Projekt

Ein Beispiel-Projekt, das die hier beschriebene Geolocation-Eingabekomponente verwendet, kann auf Anfrage vom FirstSpirit-Helpdesk zur Verfügung gestellt werden. Wenden sie sich bitte an <https://helpdesk.e-spirit.de>.

Die Eingabekomponente kann aber in beliebigen FirstSpirit-Projekten eingebunden werden. Dazu müssen lediglich die betreffenden Seiten- oder Absatzvorlagen erweitert werden.

*Weitere Informationen zur Vorlagenentwicklung siehe Online Dokumentation zu FirstSpirit (ODFS).*

*Informationen zur Entwicklung von SwingGadget-Eingabekomponenten siehe Entwicklerhandbuch für Komponenten.*

---

<sup>3</sup> Für weitere Informationen zum Installieren und Deinstallieren des Google Earth Plug-ins siehe auch <http://maps.google.com/support/bin/answer.py?hl=en&answer=178389>



## 4.2 Anwendungsbereiche der Google Maps Integration

### 4.2.1 Adress-Suche mit Geolokalisierung

In dieser Beispiel-Implementierung wird eine Geolocation-Eingabekomponente für FirstSpirit entwickelt, die zu einer eingegebenen Adresse bzw. einem Adressteil (beispielsweise einem Straßennamen), die korrekten geographischen Positionsdaten ermittelt und zur Weiterverarbeitung speichert. Die Eingabekomponente hat ein Eingabefeld zur Aufnahme eines Adress-Strings, das durch den Redakteur bearbeitet werden kann. Mit einem Klick auf den Button "Search Geolocation" kann der Redakteur eine textbasierte Suche über die Google Maps-API starten. Dazu wird die Webapplikation Google Maps in den Applikationsbereich der FirstSpirit-Redaktionsumgebung integriert. Die hinter der Eingabekomponente liegende Implementierung (siehe Kapitel 4.3 Seite 52) öffnet neben dem Vorschau-Tab, zunächst ein weiteres Tab im Applikationsbereich des JavaClients, das die Webapplikation Google Maps enthält. Die über die Suche ermittelten geographischen Koordinaten werden innerhalb der Karte mit einer Markierung  versehen, in die SwingGadget-Eingabekomponente übernommen und dort sowohl als vollständige Adressinformation als auch anhand des geographischen Längen- und Breitengrads dargestellt.

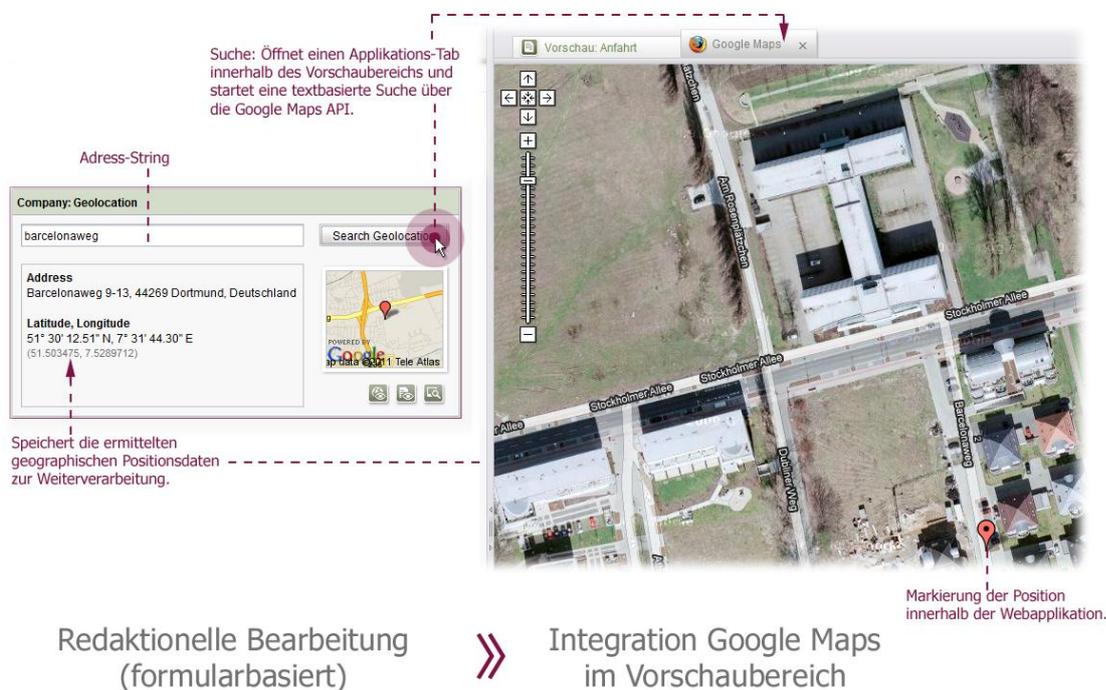


Abbildung 6: Anwendungsfall - Adress-Suche mit Geolokalisierung



### 4.2.2 Ändern der Koordinate über die Google Maps Integration

Die Markierung der Koordinate innerhalb der integrierten Google Maps Anwendung kann durch den Redakteur verändert werden. Die Änderung innerhalb der Kartendarstellung wirkt sich anschließend auf die in der SwingGadget-Eingabekomponente gespeicherte Koordinate aus.



**Abbildung 7: Anwendungsfall – Verschieben der Markierung in der Kartendarstellung**

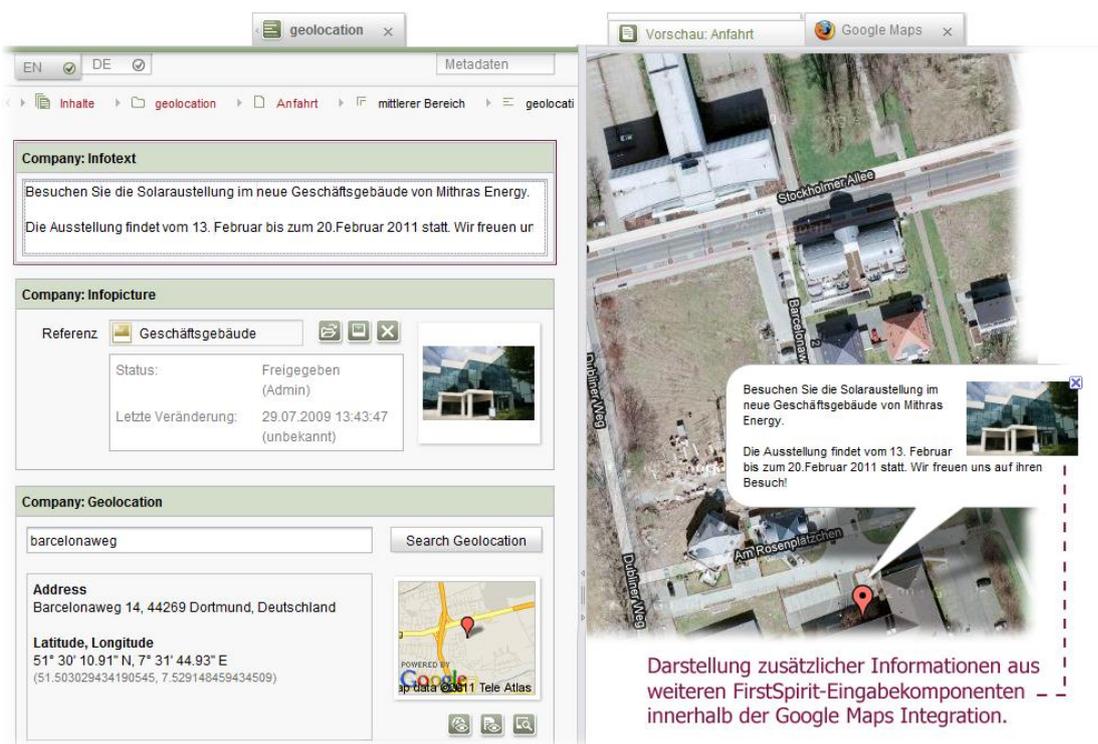
Der Redakteur muss den betreffenden Absatz zunächst zum Bearbeiten sperren. Mit einem Klick auf den Button  wird ein (Applikations-)Tab mit der Webapplikation im Applikationsbereich des JavaClients geöffnet. Im Bearbeitungsmodus wird die gesetzte Markierung von der Beispiel-Implementierung immer innerhalb einer Hybrid-Map dargestellt. Das gilt auch, wenn die Webapplikation bereits im Applikationsbereich geöffnet wurde und der Redakteur zuvor eine andere Darstellungsform ausgewählt hat.



Innerhalb der Hybrid-Map kann der Redakteur die Markierung  einfach per Drag&Drop oder über das Google Kontext-Menü ("Was ist hier?") verschieben. Auf diese Weise kann beispielsweise nach der Suche einer geografischen Position über die Google Maps-API, die Markierung innerhalb der Karte korrigiert und auf das gewünschte Gebäude verschoben werden. Die korrigierten Positionsdaten werden von der Beispiel-Implementierung anschließend in die Geolocation-Eingabekomponente übernommen.

### 4.2.3 Einblenden zusätzlicher Informationen (Google Balloons)

Neben den geographischen Daten sollen noch weitere Informationen zu einer Markierung innerhalb der Karte dargestellt werden. Der Formularbereich bietet dem Redakteur zu diesem Zweck weitere Eingabefelder (Company: Infotext und Company: Infopicture) an. Jeder Markierung, die innerhalb einer Geolocation-Eingabekomponente gespeichert ist, kann damit ein kurzer Informationstext und ein Bild aus der FirstSpirit-Medien-Verwaltung zugewiesen werden.



Redaktionelle Bearbeitung  
(formularbasiert)



Integration Google Maps  
im Vorschaubereich

**Abbildung 8: Anwendungsfall – Einblenden zusätzlicher Informationen**

Diese vom Redakteur gepflegten Informationen werden bei einem Klick auf die Markierung sowohl innerhalb der integrierten Vorschau (Vorschau-Tab) als auch in



der integrierten Webapplikation (Applikations-Tab) als Google Balloons eingeblendet. Das sind kleine Informationsfenster, die HTML, CSS oder JavaScript-Code enthalten können.

#### 4.2.4 3D-Darstellung über Google Earth

Neben der herkömmlichen Darstellung der Koordinate innerhalb einer Hybrid-Map mit geringer Flughöhe, bietet die Eingabekomponente über die Google Maps Integration auch eine 3D-Ansicht (Google Earth) mit großer Flughöhe an.



*Diese Darstellung erfordert die Installation eines Google Earth Plug-ins (siehe Kapitel 4.1.5 Seite 45).*

Sofern Google Maps nicht bereits im integrierten Applikationsbereich von FirstSpirit geöffnet ist, wird mit einem Klick auf den Button  zunächst ein Applikations-Tab geöffnet. Die hinter der Eingabekomponente liegende Implementierung stellt nun statt der herkömmlichen Hybrid-Map eine Karte vom Typ G\_SATELLITE\_3D\_MAP dar. Dieser Kartentyp zeigt ein interaktives 3D-Modell der Erde mit Satellitenbildern. (War die Anwendung bereits im Applikationsbereich geöffnet, wird einfach die Kartendarstellung im bestehenden Applikations-Tab umgeschaltet.)

Innerhalb der 3D-Ansicht kann komfortabel zwischen unterschiedliche Koordinaten (die in mehreren Geolocation-Eingabekomponenten gespeichert wurden) umgeschaltet werden. Beim Wechsel zwischen den Koordinaten (beispielsweise bei der Auswahl eines neuen Geolocation-Absatzes über den FirstSpirit-Navigationsbaum) erfolgt im Applikations-Tab eine unmittelbare 3D-Überblendung von einer Position zur anderen, die mit dem klassischen "Zoom-to-Location"-Effekt von Google Earth ausgeführt wird<sup>4</sup>. Auch alle zusätzlichen Informationen, die zu einer Markierung gespeichert wurden, können innerhalb dieser Darstellung eingeblendet werden (Injektion zusätzlicher Informationen als Google Earth Balloons) (vgl. Kapitel 4.2.3). Die Implementierung stellt damit eine interaktive Live-Vorschau für die Darstellung geographischer Informationen zur Verfügung.

---

<sup>4</sup> Beispiel siehe <http://earth-api-samples.googlecode.com/svn/trunk/examples/balloon-change-content.html>



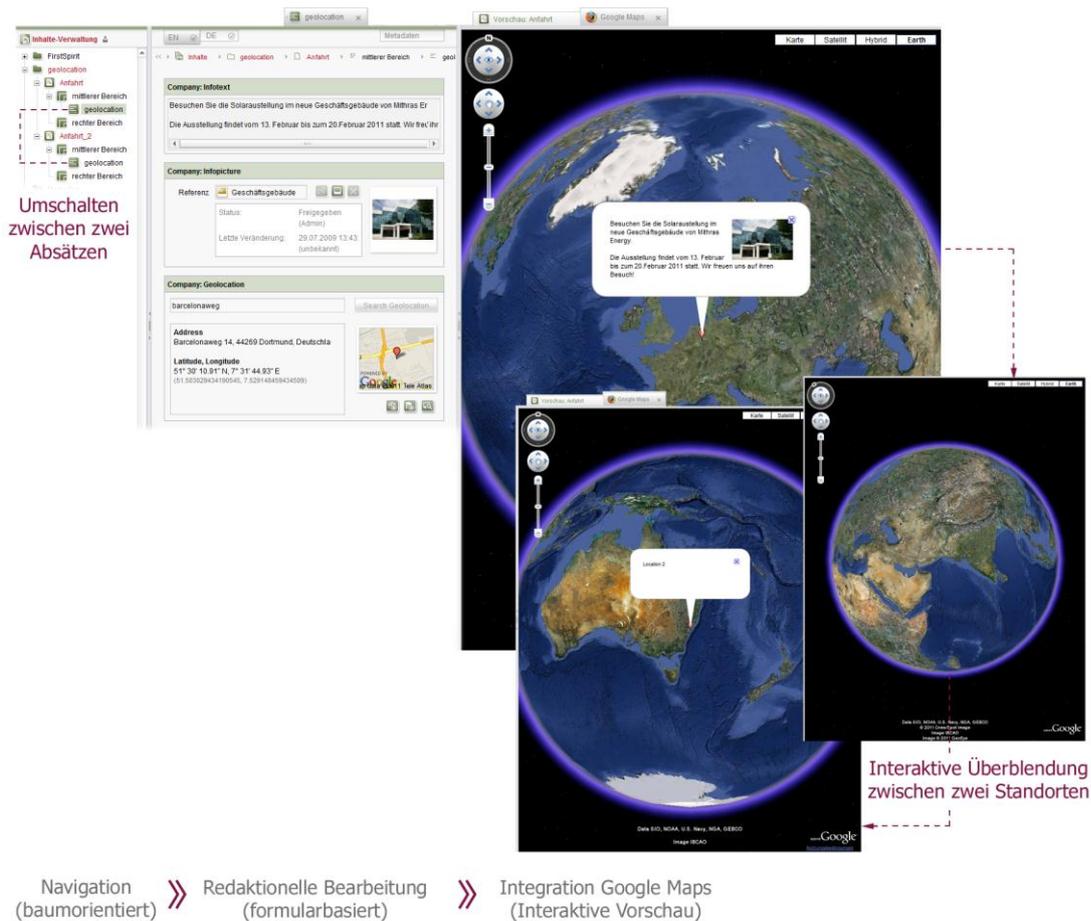


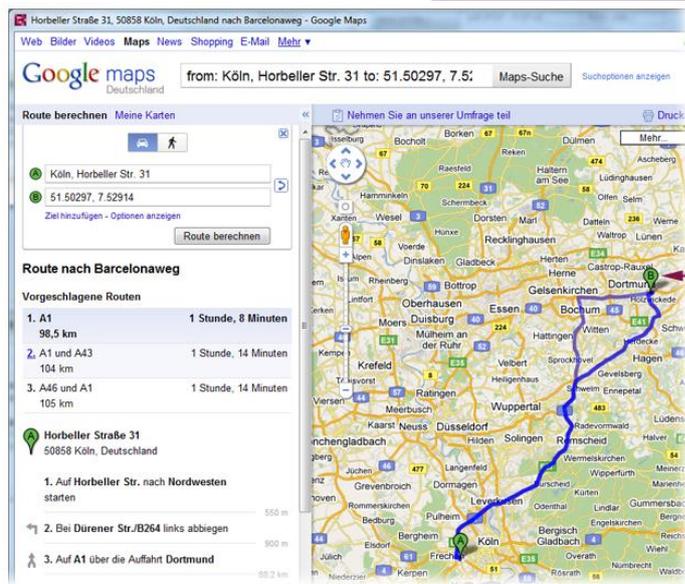
Abbildung 9: Anwendungsfall – 3D-Überblendung zwischen zwei Positionen

#### 4.2.5 Anfahrtsbeschreibung

Die Geolocation-Eingabekomponente speichert die über die Google Maps-API ermittelte geographische Koordinate zur weiteren Verarbeitung (vgl. Kapitel 4.2.1 Seite 46). Diese Daten werden innerhalb der Website für eine Routenplanung bzw. eine Anfahrtsbeschreibung verwendet. Sobald über die Google Maps Integration (beispielsweise über die Suche) eine geographische Koordinate ermittelt und in der Geolocation-Eingabekomponente gespeichert wurde, wird innerhalb der Website (bzw. im Vorschau-Tab) ein Formular auf Basis der gespeicherten Koordinaten erzeugt. Die aktualisierte Website zeigt einen Kartenausschnitt mit einer Markierung der gespeicherten Koordinaten. Zusätzlich werden Eingabefelder für zwei integrierte Routenplaner (Google Maps und Bing Maps) mit einem vorselektierten Startort angezeigt. Als Zielort werden die innerhalb der Komponente gespeicherten Koordinaten verwendet. Der Startort kann vom Besucher der Website (oder vom Redakteur innerhalb des Applikationsbereichs) verändert werden. Beim Klick auf den entsprechenden Button wird ein Fenster von Google Maps bzw. Bing Maps mit den



jeweiligen Parametern aufgerufen.



Routenplanung starten:

A: Adresse, die innerhalb der Website vom Benutzer eingetragen wird.

B: Geokoordinaten des Firmenstandorts, die innerhalb der Eingabekomponente in FirstSpirit gespeichert wurden.

Redaktionelle Bearbeitung  
(formularbasiert)



Vorschau  
(Live-Rendern)

Abbildung 10: Anwendungsfall - Anfahrtsbeschreibung



### 4.3 Implementierung: Applikationsintegration für Google Maps

In den vorangehenden Kapiteln wurde das Konzept der nahtlosen Integration einer Webapplikation in den FirstSpirit JavaClient (siehe Kapitel 2 Seite 12) sowie die dazu erforderlichen Erweiterungen der FirstSpirit-Client-API vorgestellt (siehe Kapitel 3 Seite 20). Dieses Kapitel zeigt nun exemplarisch eine konkrete Implementierung einer Applikationsintegration für den FirstSpirit-JavaClient.

Die Webapplikation Google Maps soll in den Applikationsbereich des JavaClients integriert werden. Gesteuert wird die Integration über eine SwingGadget-Eingabekomponente (siehe Kapitel 4.3.1 Seite 54), die eng mit der Applikation verknüpft ist. Die Verbindung zwischen der Java-Ebene des FirstSpirit-JavaClients und der nativen Browser-Ebene der Webapplikation erfolgt dabei mittels der neuen FirstSpirit-AppCenter-API.

Um die Implementierung einer eigenständigen Webapplikation zu vermeiden, wird innerhalb der Beispiel-Implementierung ein HTML-Code initiiert, der beim Laden einen Google-Maps-Container initialisiert (siehe maps.html - Container für die Kartendarstellung initialisieren Kapitel 4.3.15 Seite 107). Innerhalb dieser HTML-Seite werden JavaScript-Methoden definiert, die Kernfunktionalitäten wie "Eintrag hinzufügen", "Eintrag entfernen" und "Viewport modifizieren" bereitstellen. Diese JavaScript-Methoden werden auf Java-Seite entsprechend aufgerufen, um eine geographische Position anzuzeigen oder zu verändern.

Für die Identifikation der einzelnen Map-Einträge werden IDs erzeugt, durch die eine direkte Zuordnung und Steuerung erfolgt (Markierungen einblenden und einer Eingabekomponente zuordnen siehe Kapitel 4.3.6 Seite 69). Für die Modifikation der geographischen Positionsdaten und die entsprechende Änderungsnotifizierung ist hier ein Rückweg nötig, der mittels einer Objekt-Injektion bereitgestellt wird (MapsPlugin - GeolocationUpdater (Injection Java » JavaScript) siehe Kapitel 4.3.5 Seite 65). Dieses Objekt hat eine Methode für die Aktualisierung der exakten geographischen Position sowie eine Methode mittels der der Adress-String aktualisiert werden kann.

Die nachfolgenden Kapitel beschreiben ausgesuchte Codestellen der Beispiel-Implementierung und stellen eine Orientierung für die Entwicklung eigener Integrationslösungen dar. Dabei wird insbesondere die Verwendung der FirstSpirit-AppCenter-API erläutert. Die ebenfalls verwendeten Google-Maps-API wird nur in dem Umfang erklärt, der für das Verständnis des Beispiels erforderlich ist



(Implementierungsdetails können der Dokumentation zur Google-Maps-API<sup>5</sup> bzw. zur Google-Earth-API<sup>6</sup> entnommen werden.) Die Entwicklung der verwendeten SwingGadget-Eingabekomponente (siehe Kapitel 4.3.1) wird hier ebenfalls nicht beschrieben. Alle benötigten Informationen und Schnittstellen zur Entwicklung von SwingGadget-Eingabekomponenten können dem Entwicklerhandbuch für Komponenten entnommen werden.

- (SwingGadget-) Eingabekomponente `CUSTOM_GEOLOCATION` (siehe Kapitel 4.3.1 Seite 54)
- MapsPlugin – Neue Instanz vom Typ MapsPlugin erzeugen (siehe Kapitel 4.3.2 Seite 55)
- MapsPlugin – Öffnen der Applikation innerhalb eines Tabs (siehe Kapitel 4.3.3 Seite 57)
- MapsPlugin – JavaScript ausführen (Java » JavaScript) (siehe Kapitel 4.3.4 Seite 61)
- MapsPlugin - GeolocationUpdater (Injection Java » JavaScript) (siehe Kapitel 4.3.5 Seite 65)
- Markierungen einblenden und einer Eingabekomponente zuordnen (siehe Kapitel 4.3.6 Seite 69)
- Listener – Auf Änderungen reagieren (siehe Kapitel 4.3.7 Seite 77)
- Geodaten der Eingabekomponente aktualisieren (JavaScript » Java) (siehe Kapitel 4.3.8 Seite 83)

Der vollständige Source-Code, der in diesem Beispiel beschriebenen Applikations-Integrations-Implementierung für Google Maps, befindet sich im Zip-Archiv zum Entwicklerhandbuch. Die Archivdatei kann über die Online-Dokumentation von FirstSpirit heruntergeladen werden (Bereich: Dokumentation für Entwickler – Beispiel-Implementierungen).



*Bei Verwendung der Google-Maps-Integration müssen die Lizenzvorgaben des Herstellers beachtet werden (siehe Kapitel 4.1.2 Seite 43).*

---

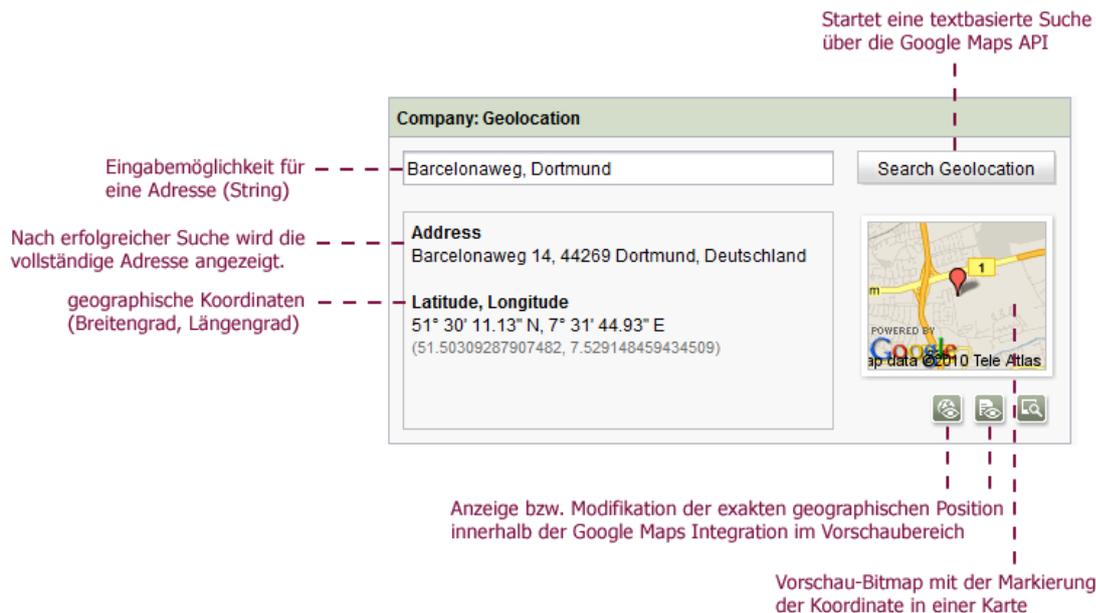
<sup>5</sup> <http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/reference.html>

<sup>6</sup> <http://code.google.com/intl/de/apis/earth/documentation/reference/index.html>



### 4.3.1 (SwingGadget-) Eingabekomponente CUSTOM\_GEOLOCATION

Für die in Kapitel 4.2 (Seite 46 ff.) vorgestellten Anwendungsbereiche wurde eine neue Eingabekomponente `CUSTOM_GEOLOCATION` entwickelt. Diese Eingabekomponente speichert eine geographische Koordinate, bestehend aus zwei dezimalen Werten (geographischer Längen- und Breitengrad) und zeigt sowohl die Koordinate als auch die zu dieser Koordinate gehörige, vollständige Adressinformation (Strasse, Stadt, Land) an. Des Weiteren steht dem Redakteur ein einfaches Textfeld für eine (unformatierte) Adresseingabe zur Verfügung, um die Suche zu vereinfachen. Der dort eingetragenen Adress-String wird für eine textbasierte Suche mittels der Google Maps-API verwendet. Da Google Maps für diese Suche den Ort des Internetzugangspunktes, von dem die Anfrage gestellt wird, mit einbezieht (soweit möglich/bekannt), kann es bei standortnahen Zielen bereits ausreichend sein, einen Straßennamen einzutragen.



**Abbildung 11: Geolocation-Eingabekomponente**

Die Besonderheit der Geolocation-Eingabekomponente liegt in der engen Integration mit der Webapplikation Google Maps. Die Komponente beinhaltet einen Button zur Anzeige/Modifikation der exakten geographischen Position sowie einen Button der auf Basis des Adress-Strings eine Suche initiiert. Bei beiden Aktionen wird - wenn noch nicht geschehen oder geschlossen - ein Applikations-Tab im integrierten Browser des JavaClients geöffnet (MapsPlugin – Öffnen der Applikation innerhalb eines Tabs siehe Kapitel 4.3.3 Seite 57). In diesem Tab wird die Webapplikation zur Anzeige der Geo-Koordinate geöffnet. Der Benutzer kann die angezeigte Koordinate dort ändern (beispielsweise durch Drag&Drop der Markierung innerhalb der Kartendarstellung). Die neue Koordinate (inkl. Adressinformationen) wird dann



dynamisch in der Eingabekomponente aktualisiert. Außerdem zeigt die Eingabekomponente ein Vorschau-Bitmap mit einer verkleinerten Kartendarstellung und einer Markierung der aktuell ausgewählten Position aus der Webapplikation an. Dieses Vorschaubild stammt nicht aus der FirstSpirit-Medien-Verwaltung, sondern wird über die integrierte Webapplikation bereitgestellt und bei einer Änderung der Position innerhalb der integrierten Anwendung (Google Maps) ebenfalls dynamisch aktualisiert. Das Öffnen des Applikations-Tabs und die Live-Aktualisierung der Kartendarstellung übernimmt die hinter der Eingabekomponente liegende Implementierung (siehe Kapitel 4.3.2 ff.).

### 4.3.2 MapsPlugin – Neue Instanz vom Typ MapsPlugin erzeugen

Innerhalb der SwingGadget-Implementierung (der Eingabekomponente `CUSTOM_GEOLOCATION`) muss zunächst eine neue Instanz vom Typ `MapsPlugin` erzeugt werden. Die Klasse `MapsPlugin` ist zuständig für die Integration von Google Maps in den Applikationsbereich des FirstSpirit-JavaClients.

Dazu ist ein sogenannter `SpecialistsBroker` notwendig. Eine Instanz vom Typ `SpecialistsBroker` bietet über unterschiedliche "Spezialisten" Zugriff auf bestimmte Dienste oder Informationen. Für das Arbeiten mit integrierten Webapplikationen, wird ein Spezialist vom Typ `ServicesBroker` benötigt. Dieser kann auf dem `SpecialistsBroker` mithilfe der Methode `<S> s requireSpecialist(SpecialistType<S> type)` angefordert werden. Zudem kann über den Aufruf der Methode `getService()` ein Dienst (`Service`) angefordert werden. Diese Methode wird beispielsweise im weiteren Verlauf benötigt, um den `ApplicationService` zu verwenden (siehe Kapitel 4.3.3 Seite 57).

Über den typisierten `SwingGadgetContext` (vgl. Entwicklerhandbuch für Komponenten) kann innerhalb der SwingGadget-Implementierung eine Instanz vom Typ `ServicesBroker` mithilfe der Methode `<S> s requireSpecialist(SpecialistType<S> type)` angefordert werden. Die neue Instanz vom Typ `ServicesBroker` wird abschließend beim Aufruf der Methode `MapsPlugin.getInstance(...)` an die Klasse `MapsPlugin` übergeben.

```
1. public class GeolocationSwingGadget...{
2.
3.     private final SwingGadgetContext<GomGeolocation> _context;
4.     private ServicesBroker _servicesBroker;
5.     ...
6. }
```



```
7.     public GeolocationSwingGadget(final
      SwingGadgetContext<GomGeolocation> context) {
8.         super(context);
9.         _context = context;
10.    }
11.    ...
12.    private MapsPlugin getMapsPlugin() {
13.        if (_servicesBroker == null) {
14.            _servicesBroker =
              _context.getBroker().requireSpecialist(ServicesBroker.TYPE);
15.        }
16.        return MapsPlugin.getInstance(_servicesBroker);
17.    }
18. }
```

Listing 1: Geolocation - Neue Instanz von MapsPlugin erzeugen (SwingGadget-Impl.)

Die Methode `MapsPlugin.getInstance(...)` erzeugt eine Singleton-Instanz der Klasse `MapsPlugin`. Diese Singleton-Instanz verfügt initial über einen `ServicesBroker` (der innerhalb der `SwingGadget`-Implementierung angefordert und an die Klasse `MapsPlugin` übergeben wird) und über den `MapType` `G_HYBRID_MAP`. Bei diesem `MapType` handelt es sich um einen Standard-Kartentyp der Google Maps-API, der eine Mischung aus Fotokacheln und zusätzlichen Informationen, wie beispielsweise Straßen oder Ortsnamen darstellt.

```
1.     public class MapsPlugin implements TabListener, BrowserListener {
2.
3.         @SuppressWarnings({"UnusedDeclaration"})
4.         public enum MapType {
5.             G_NORMAL_MAP, G_SATELLITE_MAP, G_HYBRID_MAP,
              G_SATELLITE_3D_MAP
6.         }
7.         private final ServicesBroker _servicesBroker;
8.         private static MapsPlugin INSTANCE;
9.
10.        ...
11.        private MapsPlugin(final ServicesBroker servicesBroker) {
12.            _servicesBroker = servicesBroker;
13.            _mapType = MapType.G_HYBRID_MAP;
14.        }
15.
16.        public static MapsPlugin getInstance(final ServicesBroker
              servicesBroker) {
17.            if (INSTANCE == null) {
```



```
18.         INSTANCE = new MapsPlugin(servicesBroker);
19.     }
20.     return INSTANCE;
21. }
22. ...
23. }
```

Listing 2: Geolocation – Konstruktor MapsPlugin (MapsPlugin-Impl.)

### 4.3.3 MapsPlugin – Öffnen der Applikation innerhalb eines Tabs

Um eine Webapplikation in den FirstSpirit-JavaClient zu integrieren, ist es notwendig, den in FirstSpirit integrierten Browser zu steuern (Steuerung des integrierten Browsers siehe Kapitel 2.1.1 Seite 12). Dazu muss zunächst eine neue Browser-Instanz (bzw. ein neues Tab) innerhalb des Applikationsbereichs erzeugt werden, in der die gewünschte Webapplikation geöffnet werden kann.

Beim Anfordern der integrierten Webapplikation, beispielsweise durch einen Klick auf den Button  der Geolocation-Komponente, wird zunächst überprüft, ob bereits eine Browser-Instanz der Webapplikation im Applikationsbereich des JavaClients existiert. Diese Bedingung wird über den Aufruf der Methode `ensureApplicationTab()` innerhalb der MapsPlugin-Implementierung überprüft:

```
1.  public class MapsPlugin implements TabListener, BrowserListener {
2.      private BrowserApplication _application;
3.      ...
4.      public void add(final GadgetIdentifier gadgetId, ...) {
5.          ensureApplicationTab();
6.          ...
7.      }
8.
9.      ...
10.     //inner method
11.     private void ensureApplicationTab() {
12.         if (_application == null) {
13.             getApplication(); // open application tab
14.         }
15.     }
16. }
```

Listing 3: Geolocation – BrowserApplication-Instanz vorhanden? (MapsPlugin-Impl.)

Existiert bisher keine Browser-Instanz für die Webapplikation wird die Methode `getApplication()` aufgerufen, die eine neue Instanz der Webapplikation vom



Typ `BrowserApplication` zurückliefert.

Die benötigten Schnittstellen der FirstSpirit-AppCenter-API werden nachfolgend beschrieben:

- 1) `ApplicationService`: Einstiegspunkt für die Steuerung eines Tabs ist immer der `ApplicationService`. Eine Instanz vom Typ `ApplicationService` wird durch den Aufruf der Methode `<T> T getService(Class<T> serviceClass)` auf dem übergeben `ServicesBroker` angefordert (vgl. Kapitel 4.3.2). Über diesen Dienst können neue Anwendungen eines bestimmten Typs innerhalb des Applikationsbereichs geöffnet (siehe Punkt 3) oder die Anwendungen aus bereits bestehenden Browser-Instanzen geholt werden (siehe Punkt 5) (Beschreibung des Interfaces `ApplicationService` siehe Kapitel 3.1 Seite 21).
- 2) `BrowserApplicationConfiguration`: Außerdem wird eine Konfiguration für die integrierte Browser-Instanz benötigt. Dazu wird eine Instanz vom Typ `BrowserApplicationConfiguration` über den Aufruf `BrowserApplicationConfiguration.GENERATOR.invoke()` erzeugt. Das Interface `BrowserApplicationConfiguration` stellt Methoden bereit, um beispielsweise eine bestimmte Browser-Engine für die Integration auszuwählen oder eine Adresszeile innerhalb der Browser-Instanz ein- bzw. auszublenden. Die Konfiguration wird beim Öffnen einer neuen Browser-Instanz über die Methode `openApplication(...)` als Parameter übergeben (siehe Punkt 3) (Beschreibung des Interfaces `BrowserApplicationConfiguration` siehe Kapitel 3.9 Seite 39).
- 3) `ApplicationTab`: Für die Integration einer Webapplikation muss zunächst ein neuer Tab (neben dem Vorschau-Tab) im Applikationsbereich des `JavaClients` geöffnet werden. Dazu wird die Methode `openApplication(...)` auf dem `ApplicationService` aufgerufen (siehe Punkt 1). Der Methode werden der Typ (Abstract Class: `ApplicationType` siehe Kapitel 3.6 Seite 34) der gewünschten Anwendung (hier: `BrowserApplication`, vgl. Punkt 5) sowie die Konfiguration für den integrierten Browser übergeben (vgl. Punkt 2). Die Methode `openApplication(...)` liefert eine Instanz vom Typ `ApplicationTab` zurück. Das Interface `ApplicationTab` bietet allgemeine Methoden zur Steuerung des Tabs an, beispielsweise kann der Tab über die entsprechenden Methodenaufrufe in den Vordergrund geholt oder geschlossen werden (Interface: `ApplicationTab` siehe Kapitel 3.2 Seite 23).
- 4) `TabListener`: Zum Verfolgen von Änderungen, beispielsweise das



Selektieren eines Tabs durch den Benutzer, muss dem Applikations-Tab eine Instanz vom Typ `TabListener` hinzugefügt werden (Interface: `TabListener` siehe Kapitel 3.5 Seite 33). Ein `TabListener` wird dem Applikations-Tab über den Aufruf der Methode `addTabListener(...)` hinzugefügt, in diesem Beispiel implementiert die Klasse `MapsPlugin` das Interface `TabListener` selbst. Es wird allerdings empfohlen, die zugehörige Adapter-Implementierung zu verwenden. Die Methoden des Interfaces zum Selektieren, Deselektieren und Schließen des Applikations-Tabs können dann bei Bedarf implementiert werden (siehe Listener – Auf Änderungen reagieren, Kapitel 4.3.7, Seite 77).

- 5) `BrowserApplication`: Durch den Aufruf der Methode `getApplication()` auf der Instanz vom Typ `ApplicationTab` wird die neue Instanz der integrierten Webapplikation vom Typ `BrowserApplication` zurückgeliefert (vgl. Punkt 3 – `openApplication(...)`). Das Interface `BrowserApplication` stellt Methoden zur Steuerung der neuen integrierten Browser-Instanz zur Verfügung, beispielsweise das Öffnen einer URL innerhalb der Browser-Instanz (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35). Abschließend muss die gewünschte Webapplikation (hier Google Maps) in der neu erzeugten Browser-Instanz geöffnet werden. Prinzipiell stehen für diese Integration zwei unterschiedliche Wege zur Verfügung:
  - a) Es kann zum einen eine globale Webapplikation implementiert und auf dem FirstSpirit-Server installiert werden. In diesem Fall kann im Applikationsbereich die URL der Webapplikation aufgerufen werden (unter Verwendung der Methode `openUrl(...)`). Diese Methode wird auch verwendet, um eine externe Webapplikation aufzurufen.
  - b) Soll eine eigenständige Webapplikation umgangen werden, kann aber auch einfach der gewünschte HTML-Code initiiert und aufgerufen werden (unter Verwendung der Methode `setHtmlContent(...)`). Dieser zweite Weg wird auch für die hier vorgestellte Google Maps Integration verwendet. Zur Initiierung des HTML-Codes wird die Datei `maps.html` verwendet, die eine Art Kapsel um die benötigten Google-API-Aufrufe bildet (siehe Kapitel 4.3.12 ff.).
- 6) `BrowserListener`: Ein Zugriff auf die im Webbrowser dargestellten Inhalte (DOM-Baum), beispielsweise zur Manipulation der Daten, ist nicht zu jedem Zeitpunkt möglich. Um einen gesteuerten Zugriff auf die Inhalte sicherzustellen, muss eine Instanz vom Typ `BrowserListener` verwendet werden (Interface: `BrowserListener` siehe Kapitel 3.8 Seite 38). Ein `BrowserListener` wird der `BrowserApplication` über den Aufruf der Methode `addBrowserListener(...)` hinzugefügt. in diesem Beispiel



implementiert die Klasse `MapsPlugin` das Interface `BrowserListener` selbst. Es wird allerdings empfohlen, die zugehörige Adapter-Implementierung zu verwenden. Die Methoden des Interfaces können dann bei Bedarf implementiert werden (siehe `Listener – Auf Änderungen reagieren`, Kapitel 4.3.7, Seite 77).

```
1. private BrowserApplication _application;
2. private ApplicationTab<BrowserApplication> _tab;
3. private final ServicesBroker _servicesBroker;
4.
5. /**
6.  * Get BrowserApplication instance and may initialize it.
7.  * @return BrowserApplication instance
8.  */
9. private BrowserApplication getApplication() {
10.     if (_application == null) {
11.         final ApplicationService service =
12.             _servicesBroker.getService(ApplicationService.class);
13.         final BrowserApplicationConfiguration configuration =
14.             BrowserApplicationConfiguration.GENERATOR.invoke();
15.         configuration.title("Google Maps");
16.         configuration.showAddressBar(false);
17.         configuration.engineType(EngineType.FIREFOX);
18.         _tab = service.openApplication(BrowserApplication.TYPE,
19.             configuration);
20.         _tab.addTabListener(this);
21.         _application = _tab.getApplication();
22.         _application.addBrowserListener(this);
23.         _application.setHtmlContent(getHtmlContent());
24.     }
25.     return _application;
26. }
```

Listing 4: Geolocation – Öffnen der Application innerhalb eines Tabs (`MapsPlugin-Impl.`)



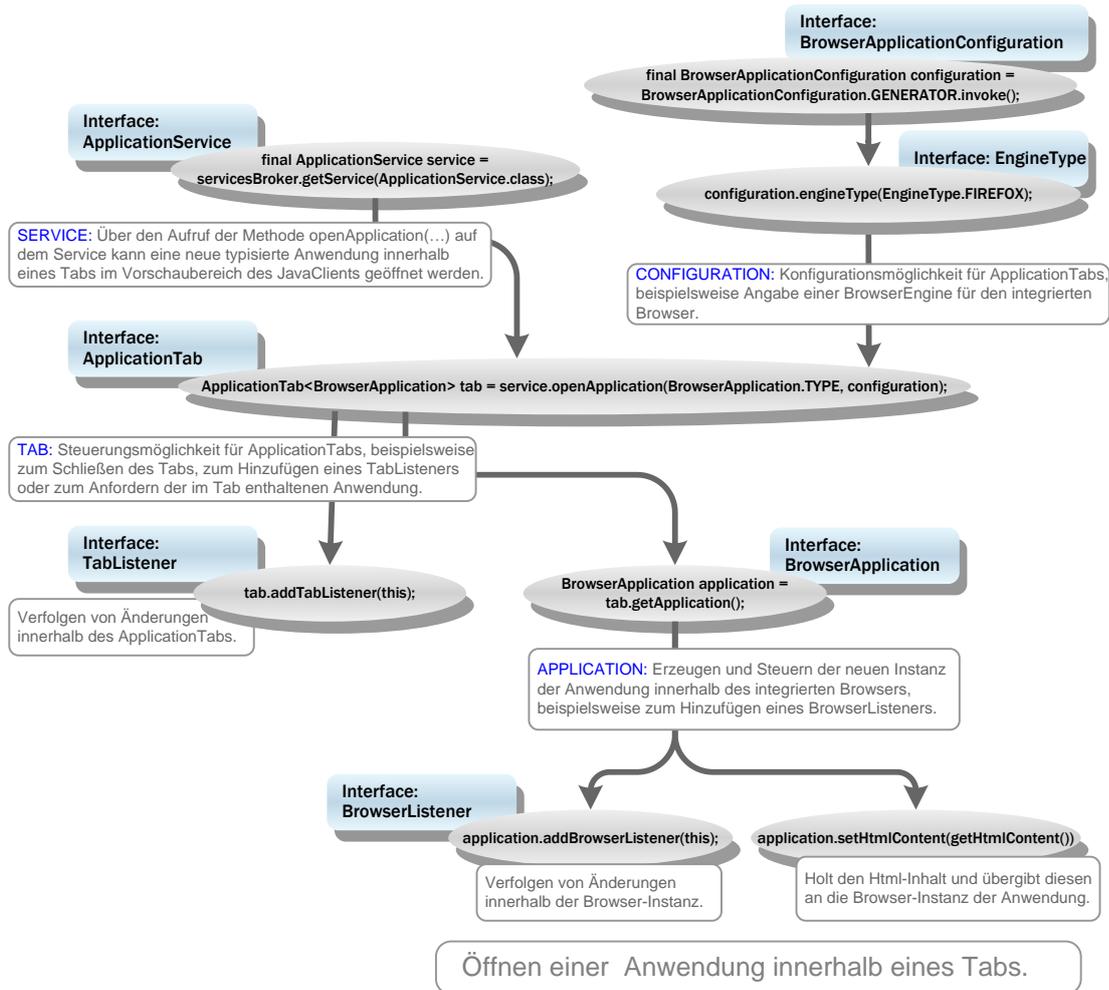


Abbildung 12: Öffnen einer Anwendung innerhalb eines Tabs

#### 4.3.4 MapsPlugin – JavaScript ausführen (Java » JavaScript)

Wie in Kapitel 2.1 bereits erläutert wurde, müssen Schnittstellen für die Kommunikation zwischen der Java-Ebene des FirstSpirit-JavaClients und der nativen Ebene des integrierten Browsers (bzw. der Google Maps-API) geschaffen werden (vgl. Konzept aus Kapitel 2.1.2 (Seite 13 ff.)).

Wird beispielsweise innerhalb der Geolocation-Eingabekomponente nach einer bestimmten Adresse gesucht (Eingabe eines Adress-Strings und Klick auf den Search-Button), muss eine Anfrage zur Geokodierung dieses Adress-Strings an die Webapplikation (Google Maps) gesendet und der Kartenausschnitt, innerhalb des integrierten Browsers angepasst werden (vgl. Anwendungsfall Kapitel 4.2.1 Seite 46). Voraussetzung dafür ist eine Schnittstelle, die es ermöglicht, aus der Java-Umgebung heraus eine JavaScript-Methode auszurufen (Kommunikation Java »



JavaScript).

Die Geolocation-Implementierung verwendet dazu die Methode `void executeScript(String script)`. Diese Methode führt den übergebenen JavaScript-Code im aktuell geöffneten Browser-Dokument aus und ermöglicht so eine zielgerichtete, unidirektionale Kommunikation in Richtung Java » JavaScript. Dabei wird der auszuführende JavaScript-Code einfach als String übergeben. Die Methode wird über das Interface `BrowserApplication` der FirstSpirit-AppCenter-API bereitgestellt (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35).

Zunächst muss jedoch der benötigte JavaScript-Code erstellt werden. Innerhalb der `MapsPlugin`-Implementierung werden dazu zahlreiche Methoden implementiert, die abhängig vom jeweiligen Anwendungsfall ein Skript über einen `StringBuilder` zusammenstellen. Die Methode `void updateBrowser()`, die zur Aktualisierung der Kartendarstellung im Browser verwendet wird, ruft beispielsweise unterschiedliche Methoden mit dem Präfix "getScript" im Methodennamen auf, die einzelne JavaScript-Fragmente zurückliefern. Für den Anwendungsfall "Adress-Suche" wird dort beispielsweise die Methode `getScriptFindAddress(...)` aufgerufen:

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     ...
3.     public void updateBrowser() {
4.         // build script code to update google map
5.         final StringBuilder buf = new StringBuilder();
6.         ...
7.         for (final GeolocationEntry location : ...) {
8.             if (location.isSearchMode()) {
9.                 buf.append(getScriptFindAddress(location));
10.            }
11.            ...
12.        }
13.        final String script = buf.toString();
14.        ...
15.    }
16.    ...
17.
18.    //--- inner methods ---//
19.
20.    /**
21.     * Build script code for an address search entry.
22.     *
```



```
23.     * @param location related geolocation entry
24.     * @return script code
25.     */
26.     private String getScriptFindAddress(final GeolocationEntry
        location) {
27.         final StringBuilder buf = new StringBuilder();
28.         final String pattern = location.getAddress().replaceAll("'",
            "\\\\"');
29.         buf.append("window.findAddress('")
30.             .append(location.getUUID()).append("'", "'")
31.             .append(pattern).append("'");
32.         return buf.toString();
33.     }
34.
35. }
```

Listing 5: Geolocation – Skript für die Geokodierung eines Adress-Strings erstellen (MapsPlugin-Impl.)

Der Methode wird ein sogenannter `GeolocationEntry` übergeben (siehe Kapitel 4.3.6 Seite 69). Dieser Eintrag enthält unter anderem den Adress-String, der im Suchfeld der Geolocation-Eingabekomponente eingetragen wurde. Die Methode `getScriptFindAddress(...)` baut aus diesen Informationen nun einen JavaScript-Code zusammen, der sowohl den Adress-String als auch die `uuid` der zugehörigen `SwingGadget`-Eingabekomponente enthält. Die `uuid` wird benötigt, um eine eindeutige Zuordnung des `GeolocationEntries` zu einer Eingabekomponente herzustellen (siehe Kapitel 4.3.6 Seite 69).

Das von der Methode `getScriptFindAddress(...)` zurückgelieferte Skript-Fragment sieht beispielsweise folgendermaßen aus (Informationen zum Objekt `window` siehe Kapitel 4.3.13):

```
window.findAddress('uuid:-1336359343', 'Barcelonaweg');
```

Innerhalb der Methode `void updateBrowser()` werden noch weitere Methoden aufgerufen. Die zurückgelieferten JavaScript-Fragmente werden abschließend zu einem Skript zusammengesetzt, beispielsweise:

```
window.clearOverlays();
window.setMapType(G_HYBRID_MAP);
window.findAddress('uuid:-1336359343', 'Barcelonaweg');
window.setEditable('uuid:-1336359343', true);
window.setInfoHtml('uuid:-1336359343', "<span ...' />...</span>");
```

Jeder dieser Aufrufe hat seine Entsprechung in einer JavaScript-Funktion der Datei `maps.html` (`maps.html` – Einführung siehe Kapitel 4.3.12 Seite 104). Der Aufruf



`window.findAddress('uuid:-1336359343', 'Barcelonaweg')` führt beispielsweise die folgende JavaScript-Funktion aus:

```
1.  /**
2.   * Use Geocoder to find the exact geolocation of the specified address
3.   * pattern
4.   *
5.   * @param uuid UUID of geolocation instance
6.   * @param pattern address string
7.   */
8.
9.  window.findAddress = function(uuid, pattern) {
10.     window.GoogleGeocoder.getLatLng(pattern, function(point) {
11.         if (point) {
12.             var latitude = point.lat();
13.             var longitude = point.lng();
14.             window.addOverlay(uuid, latitude, longitude);
15.             window.setViewPoint(latitude, longitude);
16.             window.GeolocationUpdater.update(uuid, latitude,
17.                 longitude);
18.             updateInfo(uuid, latitude, longitude);
19.         }
20.     });
21. }
```

Listing 6: Geolocation – JavaScript-Funktion findAddress (maps.html)

Diese Funktion startet eine Geokodierungs-Anfrage über die Google Maps-API. Die genaue Vorgehensweise wird in Kapitel 4.3.20 (Seite 116) beschrieben. Weitere Aufrufe entfernen beispielsweise die bisherigen Markierungen aus der Kartendarstellung (`window.clearOverlays()`) oder legen den `MapType` für die Kartendarstellung fest (`window.setMapType (G_HYBRID_MAP)`).

Das gesamte Skript mit allen enthaltenen Funktionsaufrufen, das innerhalb der Methode `void updateBrowser()` zusammengestellt wird, wird abschließend über den Aufruf der Methode `void executeScript(String script)` auf der Instanz vom Typ `BrowserApplication` ausgeführt (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35):

```
1.  public class MapsPlugin implements TabListener, BrowserListener {
2.     ...
3.     private BrowserApplication _application;
4.
5.     public void updateBrowser() {
```



```
6.         final StringBuilder buf = new StringBuilder();
7.             buf.append(getScriptClearOverlays());
8.             ...
9.             buf.append(getScriptFindAddress(location));
10.            ...
11.            final String script = buf.toString();
12.            ...
13.            _application.executeScript(script);
14.        }
15.    }
```

Listing 7: Geolocation – Ausführen der gesammelten JavaScript-Fragmente (MapsPlugin-Impl.)

Damit ist der erste Kommunikationsweg (Java » JavaScript) hinreichend beschrieben. Im zweiten Schritt müssen die Informationen aus der Webapplikation wieder in die FirstSpirit-Eingabekomponente zurückfließen (JavaScript » Java) (siehe dazu Kapitel 4.3.5 Seite 65).

### 4.3.5 MapsPlugin - GeolocationUpdater (Injection Java » JavaScript)

Im vorangehenden Kapitel wurde die Kommunikation ausgehend vom FirstSpirit-JavaClient in die nativen Ebene des integrierten Browsers über die Ausführung von JavaScript-Code beschrieben (siehe Kapitel 4.3.4 Seite 61). Dieses Kapitel beschäftigt sich mit dem umgekehrten Kommunikationsweg (JavaScript » Java). Ausgehend von der integrierten Webapplikation (bzw. der integrierten Browser-Instanz) sollen Änderungen oder Ereignisse, beispielsweise durch Verschieben der Markierung innerhalb der Kartendarstellung, auch in der Geolocation-Eingabekomponente aktualisiert werden. In diesem Fall muss die Java-Seite über die Änderung in der Webapplikation informiert werden und in der Lage sein, geeignet auf diese Änderung zu reagieren (vgl. Konzept Kommunikation zwischen Browser-Instanz und JavaClient in Kapitel 2.1.2, Seite 13)

Zur Aktualisierung der Java-Seite wird innerhalb der Beispiel-Implementierung das Java-Objekt `GeolocationUpdater` verwendet. Eine neue Instanz vom Typ `GeolocationUpdater` wird beim Ausführen der Methode `void onDocumentComplete(final String url)` erzeugt und unter dem Namen "GeolocationUpdater" in die Webapplikation injiziert.

**Hintergrund:** Ein Zugriff auf den DOM-Baum der Browser-Instanz ist nicht zu jedem Zeitpunkt möglich. Die Injektion kann also nur erfolgen, wenn das Dokument vollständig geladen wurde. Um dies sicherzustellen, muss eine Instanz vom Typ `BrowserListener` verwendet werden (Interface: `BrowserListener` siehe Kapitel 3.8 Seite 38). Die Methode `void onDocumentComplete(...)` wird immer aufgerufen,



wenn der `BrowserListener` der `Browser`-Instanz (Instanz vom Typ `BrowserApplication`) meldet, dass das Dokument (einschließlich aller Bilder) vollständig geladen wurde (Konzept DOM-Zugriff siehe Kapitel 2.2 Seite 18) (Listener – Auf Änderungen reagieren, siehe Kapitel 4.3.7, Seite 77).

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     ...
3.     private boolean _active;
4.     ...
5.
6.     //--- BrowserListener ---//
7.     ...
8.     public void onDocumentComplete(final String url) {
9.         final BrowserApplication application = getApplication();
10.
11.         // document is loaded; inject java/javascript bridge
12.         application.inject(new GeolocationUpdater(this),
13.             "GeolocationUpdater");
14.
15.         _active = true;
16.
17.         // initialize map viewport and marker
18.         updateBrowser();
19.     }
20. }
```

Listing 8: Geolocation – Injektion Java-Objekt `GeolocationUpdater` (`MapsPlugin-Impl.`)

Auf einer Instanz vom Typ `BrowserApplication` wird die Methode `void inject(Object object, String name)` aufgerufen. Die Methode wird über das Interface `BrowserApplikation` der FirstSpirit-AppCenter-API zur Verfügung gestellt (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35).

Die Methode injiziert das übergebene Java-Objekt `GeolocationUpdater` als Attribut des `Window`-Objekts in die `Browser`-Instanz, auf der es aufgerufen wurde (Informationen zum Objekt `window` siehe Kapitel 4.3.13). Durch die Injektion wird ein Stellvertreter-Objekt (Proxy) in Form eines JavaScript-Objektes erzeugt und unter dem übergebenen Namen (hier `"GeolocationUpdater"`) registriert. Nach der Registrierung kann das JavaScript-Objekt über den Aufruf `window.{name}` (hier: `window.GeolocationUpdater`) in der Datei `maps.html` verwendet werden (siehe Kapitel 4.3.21 Seite 117). Alle Methoden des Java-Objekts `GeolocationUpdater` können anschließend ebenfalls aus der JavaScript-Umgebung des integrierten



Browsers heraus aufgerufen werden.

**Hintergrund:** Das FirstSpirit-Framework erzeugt bei der Injektion für jede Methode der Java-Objekt-Instanzen eine entsprechende JavaScript-Methode mit (annähernd) identischer Methoden-Signatur. Diese JavaScript-Methode sendet beim Aufruf aus der JavaScript-Umgebung ein Event, welches auf der Java-Seite ausgewertet wird und dort die Ausführung der zugehörigen Java-Methode auslöst. Die passende Java-Methode wird anhand der Methodensignatur und der übergebenen Parameter ermittelt und aufgerufen.

Das Java-Objekt `GeolocationUpdater` verfügt beispielsweise über die Java-Methode `public void update(final String uuid, final double latitude, final double longitude)`, die einen `GeolocationEntry` auf die übergebene Koordinate aktualisiert und den zur Eingabekomponente gehörigen `ModificationListener` über die Aktualisierung informiert (Verwendung des `ModificationListeners` siehe Kapitel 4.3.7.3).

```
1.     @SuppressWarnings({"UnusedDeclaration"})
2.     public static class GeolocationUpdater {
3.
4.         private final MapsPlugin _mapsPlugin;
5.
6.
7.         public GeolocationUpdater(final MapsPlugin mapsPlugin) {
8.             _mapsPlugin = mapsPlugin;
9.         }
10.
11.
12.         public void update(final String uuid, final double latitude,
13.             final double longitude) {
14.             _mapsPlugin.update(uuid, latitude, longitude);
15.         }
16.
17.         public void info(final String uuid, final String address) {
18.             _mapsPlugin.info(uuid, address);
19.         }
20.     }
21. }
```

Listing 9: Geolocation – GeolocationUpdater (MapsPlugin-Impl.)

Nach der Injektion des Java-Objekts `GeolocationUpdater` kann die Update-Methode auf dem JavaScript-Stellvertreter-Objekt `window.GeolocationUpdater`



aufgerufen werden. Innerhalb der Beispiel-Implementierung wird die Aktualisierung durch ein Ereignis in der Webapplikation ausgelöst (beispielsweise durch das Verschieben der Markierung in der Kartendarstellung oder die Geokodierung eines Adress-Strings). Der entsprechende `EventListener` auf JavaScript-Seite ruft in diesem Fall die JavaScript-Funktion `window.GeolocationUpdater.update(uuid, newlat, newlng)` auf.

```
1.      /**
2.      * Add an marker to the GMap2 instance.
3.      *
4.      * @param uuid UUID of geolocation instance
5.      * @param latitude
6.      * @param longitude
7.      */
8.      window.addOverlay = function(uuid, latitude, longitude) {
9.          var marker = new GMarker(new GLatLng(latitude, longitude),
10.             {draggable: true});
11.          window.GoogleMap.addOverlay(marker);
12.          GEvent.addListener(marker, 'dragend', function() {
13.              var point = marker.getLatLng();
14.              var newlat = point.lat();
15.              var newlng = point.lng();
16.              window.GeolocationUpdater.update(uuid, newlat, newlng);
17.              updateInfo(uuid, newlat, newlng);
18.          });
19.      };
```

Listing 10: Geolocation – Aufrufen einer Java-Methode aus dem JavaScript-Code (maps.html-Impl.)

Dieser Aufruf wird auf der Java-Seite ausgewertet und löst dort die Ausführung der Java-Methode `public void update(final String uuid, final double latitude, final double longitude)` aus. Die Java-Methode empfängt die (neuen) Adressinformationen aus der integrierten Webanwendung, die beispielsweise bei der Geokodierung eines Adress-Strings von der Google-Maps-API ermittelt werden (JavaScript » Java). Die erneute Zuordnung des Eintrags zur Geolocation-Eingabekomponente in der Java-Umgebung erfolgt über den Parameter `uuid` (vgl. Methode `GeolocationEntry get(final String uuid)` in Kapitel 4.3.6, Seite 69).

Grundsätzlich kann ein beliebiges Java-Objekt in die JavaScript-Umgebung injiziert werden. Für die Methodenübernahme und die Abbildung der Java-Datentypen auf JavaScript-Datentypen, gelten für dieses Objekt jedoch gewisse Restriktionen (siehe



Kapitel 2.1.3 Seite 16).

### 4.3.6 Markierungen einblenden und einer Eingabekomponente zuordnen

Über die Eingabekomponente `CUSTOM_GEOLOCATION` kann eine geographische Koordinate, bestehend aus zwei dezimalen Werten (geographischer Längen- und Breitengrad) innerhalb einer Google-Maps-Karte dargestellt werden (siehe (SwingGadget-) Eingabekomponente `CUSTOM_GEOLOCATION` in Kapitel 4.3.1, Seite 54). Die Google Maps API verwendet dazu ein Objekt der Klasse `GMarker`. Dieses Markierungsobjekt beinhaltet die Werte für die geografische Koordinate und stellt diese innerhalb einer Karte dar . Das Objekt wird der Kartendarstellung mithilfe der Google Maps-Methode `addOverlay()` hinzugefügt<sup>7</sup>.

```
1.   var marker = new GMarker(new GLatLng(latitude, longitude), {draggable:
      true});
2.   window.GoogleMap.addOverlay(marker);
```

Listing 11: Geolocation – Hinzufügen eines neuen `GMarker`-Objekts (maps.html)

Auf der Java-Seite wird das Java-Objekt `GeolocationEntry` verwendet. Eine Instanz vom Typ `GeolocationEntry` beinhaltet neben den Werten, die die Koordinate beschreiben (`double latitude`, `double longitude`) noch eine Instanz vom Typ `GadgetIdentifizier`.

**Hintergrund:** Damit eine eindeutige Zuordnung einer Markierung zu einer Eingabekomponente erfolgen kann, wird innerhalb der Java-Implementierung mit einem `GadgetIdentifizier` gearbeitet. Ein `GadgetIdentifizier` wird vom FirstSpirit-Framework bereitgestellt und dient dazu, ein benanntes Form-Element innerhalb des FirstSpirit-Komponenten-Modells eindeutig zu identifizieren. Der `GadgetIdentifizier` einer `SwingGadget`-Eingabekomponente kann über die Methode `GadgetIdentifizier getId()` der abstrakten Klasse `AbstractValueHoldingSwingGadget <T, F extends GomFormElement>` geholt werden.

Innerhalb der Beispiel-Implementierung kann damit jeder `GeolocationEntry` über einen `GadgetIdentifizier` eindeutig einer bestimmten `SwingGadget`-Eingabekomponente zugeordnet werden. Das bedeutet, dass bei einem Wechsel

---

<sup>7</sup> Weitere Informationen siehe Google Maps-API-Referenz:

<http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/reference.html#GMarker>



der Eingabekomponente, beispielsweise durch die Auswahl eines neuen Absatzes im FirstSpirit-Navigations-Baum, die bisherige Markierung innerhalb der Google-Maps-Integration entfernt und durch die neue Markierung (der aktuell ausgewählten Eingabekomponente) ersetzt wird.

Desweiteren besitzt ein `GeolocationEntry` einen `ModificationListener`, der auf Änderungen des Eintrags innerhalb der Eingabekomponente reagiert (siehe Listener – Auf Änderungen reagieren, Kapitel 4.3.7, Seite 77).

```
3. public class MapsPlugin implements TabListener, BrowserListener {
4.     ...
5.     private static class GeolocationEntry {
6.         private final String _uuid;
7.         private final GadgetIdentifier _gadgetId;
8.         private ModificationListener _listener;
9.         private double _latitude;
10.        private double _longitude;
11.        ...
12.
13.        GeolocationEntry(final GadgetIdentifier gadgetId, final double
            latitude, final double longitude, final ModificationListener `
            listener) {
14.            _uuid = "uuid:" + gadgetId.hashCode();
15.            _gadgetId = gadgetId;
16.            _latitude = latitude;
17.            _longitude = longitude;
18.            _listener = listener;
19.        }
20.        ...
21.    }
22. }
```

Listing 12: Geolocation – Konstruktor `GeolocationEntry` (`MapsPlugin-Impl.`)

Das Hinzufügen einer Markierung zur Google-Maps-Karte wird aus der Java-Umgebung der `SwingGadget`-Eingabekomponente heraus angestoßen. Die Eingabekomponente `GeolocationSwingGadget` ruft, ausgelöst durch einen Event, die Methode `addToMap(...)` auf. Diese Methode holt zunächst über den Aufruf der Methode `getGadgetId()` den `GadgetIdentifier` der Komponente. Anschließend wird die `MapsPlugin`-Implementierung aufgerufen, die prüft, ob bereits ein Eintrag für die `SwingGadget`-Eingabekomponente mit diesem `GadgetIdentifier` existiert. Ist kein Eintrag vorhanden, wird über den Aufruf der Methode `getMapsPlugin().add(...)` ein neuer Eintrag (vom Typ `GeolocationEntry`) hinzugefügt. Die Methode übergibt neben dem



geographischen Längen- und Breitengrad für den Eintrag auch den **GadgetIdentifizier** der Eingabekomponente an die MapsPlugin-Implementierung:

```
1. public class GeolocationSwingGadget extends
   AbstractValueHoldingSwingGadget<...> implements ...{
2.     ...
3.     public JComponent getComponent() {
4.         @Override
5.         public void actionPerformed(final ActionEvent e) {
6.             if (!isDefault()) {
7.                 addToMap(GeolocationSwingGadget.this.getValue());
8.                 ...
9.             }
10.        }
11.    }
12.    ...
13. private void addToMap(final Geolocation value) {
14.     final GadgetIdentifizier gadgetId = getGadgetId();
15.     if (!getMapsPlugin().contains(gadgetId)) {
16.         getMapsPlugin().add(gadgetId, value.getLatitude(),
17.                             value.getLongitude(), this);
18.     }
19.     ...
20.     updateMapsPlugin(gadgetId);
21. }
22. ...
23. }
```

Listing 13: Geolocation – Übergabe des GadgetIdentifiziers an MapsPlugin (SwingGadget-Impl.)

Innerhalb der MapsPlugin-Implementierung wird zunächst die innere Methode `GeolocationEntry get(final GadgetIdentifizier gadgetId)` aufgerufen. Diese liefert den `GeolocationEntry` zurück, der der Eingabekomponente mit dem übergebenen `GadgetIdentifizier` zugeordnet ist. Existiert noch kein `GeolocationEntry` für diese Eingabekomponente (`entry==null`), wird ein neuer Eintrag angelegt (`new GeolocationEntry(gadgetId, ...)`). Dem Konstruktor wird (unter anderem) der `GadgetIdentifizier` der Eingabekomponente übergeben.

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     ...
3.     /**
4.      * Checks if registered geolocation entries contains one with the
```



```
5.      * specified gadget id.
6.      *
7.      * @param gadgetId gadget id to search for
8.      * @return {@code true} if the given identifier exists in this maps
9.      * plugin, {@code false} otherwise
10.     */
11.     public boolean contains(final GadgetIdentifier gadgetId) {
12.         return _application != null && get(gadgetId) != null;
13.     }
14.
15.     /**
16.      * Register a geolocation instance with the specified
17.      * latitude/longitude position and modification listener.
18.      *
19.      * @param gadgetId gadget id of geolocation
20.      */
21.     public void add(final GadgetIdentifier gadgetId, ...) {
22.         ensureApplicationTab();
23.         GeolocationEntry entry = get(gadgetId);
24.         if (entry == null) {
25.             entry = new GeolocationEntry(gadgetId, ...);
26.             synchronized (_entries) {
27.                 _entries.add(entry);
28.             }
29.         } else {
30.             ...
31.         }
32.     }
33.
34.     ...
35.     private GeolocationEntry get(final GadgetIdentifier gadgetId) {
36.         synchronized (_entries) {
37.             for (final GeolocationEntry entry : new
38.                 ArrayList<GeolocationEntry>(_entries)) {
39.                 if (entry.getGadgetId().equals(gadgetId)) {
40.                     return entry;
41.                 }
42.             }
43.             return null;
44.         }
45.     }
46.     ...
```



```
47.     private static class GeolocationEntry {
48.         private final String _uuid;
49.         private final GadgetIdentifier _gadgetId;
50.
51.         GeolocationEntry(final GadgetIdentifier gadgetId, ...) {
52.             _uuid = "uuid:" + gadgetId.hashCode();
53.             _gadgetId = gadgetId;
54.             ...
55.         }
56.
57.         ...
58.     }
59. }
```

Listing 14: Geolocation – Anlegen eines neuen GeolocationEntry (MapsPlugin-Impl.)

Eine neue Instanz vom Typ `GeolocationEntry` enthält neben dem übergebenen `GadgetIdentifier`, für die eindeutige Zuordnung des Eintrags zu einer `SwingGadget`-Eingabekomponente auf der Java-Seite, noch eine Variable `uuid` (universal unique identifier), für die eindeutige Zuordnung des Eintrags auf der JavaScript-Seite. Dieser Variablen wird innerhalb der Beispiel-Implementierung ein eindeutiger Hashwert zugewiesen, der auf dem Hashwert des zugehörigen `GadgetIdentifier` basiert und über die Methode `int hashCode()` der Klasse `GadgetIdentifier` zurückgeliefert wird.

**Hintergrund:** Die Kommunikation zwischen der Java-Umgebung des FirstSpirit-JavaClients und der nativen Ebene des integrierten Browsers erfolgt über JavaScript. Dabei wird der auszuführende JavaScript-Code einfach als String übergeben. Die zusätzliche Variable ist notwendig, da eine Instanz vom Typ `GadgetIdentifier` nicht in ein Objekt vom Typ `String` umgewandelt werden kann. (MapsPlugin – JavaScript ausführen (Java » JavaScript) siehe Kapitel 4.3.4 Seite 61).

Zunächst existiert also nur ein Java-Objekt (vom Typ `GeolocationEntry`), das die benötigte Koordinate sowie Informationen zur Zuordnung dieser Koordinate zu einer Eingabekomponente enthält. Im nächsten Schritt müssen diese Informationen einem Google-GMarker-Objekt innerhalb der JavaScript-Umgebung zugeordnet werden. Das Einblenden einer Markierung (GMarker) in die Kartendarstellung wird über eine JavaScript-Funktion gesteuert. Der entsprechende Aufruf wird zunächst innerhalb der MapsPlugin-Implementierung zusammengestellt und abschließend über die Methode `void executeScript(String script)` ausgeführt (vgl. Kapitel 4.3.4).

```
1.     public class MapsPlugin implements TabListener, BrowserListener {
```



```
2.     ...
3.     /**
4.     * Updates browser and google map instance.
5.     * This method must be called to update geolocation
6.     * entries and viewport.
7.     */
8.     public void updateBrowser() {
9.         ...
10.        final GeolocationEntry location;
11.        buf.append(getScriptAddOverlay(location));
12.        ...
13.        final String script = buf.toString();
14.        ...
15.        _application.executeScript(script);
16.    }
17.
18.    /**
19.    * Build script code to adding a geolocation entry to map.
20.    *
21.    * @param location related geolocation entry
22.    * @return script code
23.    */
24.    private String getScriptAddOverlay(final GeolocationEntry location)
25.    {
26.        final StringBuilder buf = new StringBuilder();
27.        buf.append("window.addOverlay('")
28.            .append(location.getUUID()).append(",")
29.            .append(location.getLatitude()).append(',')
30.            .append(location.getLongitude()).append(");");
31.        return buf.toString();
32.    }
```

Listing 15: Geolocation – Informationen aus GeolocationEntry-Objekt übermitteln (MapsPlugin-Impl.)

Das von der Methode `getScriptAddOverlay(...)` zurückgelieferte Skript-Fragment sieht folgendermaßen aus (Informationen zum Objekt `window` siehe Kapitel 4.3.13):

```
window.addOverlay('uuid:-336359343',51.50297,7.52914);
```

Über diesen Aufruf wird sowohl der Latitude- und Longitude-Wert als auch der `uuid`-Wert des aktuellen `GeolocationEntry`s aus der Java-Implementierung an die JavaScript-Implementierung übergeben. Die zugehörige JavaScript-Funktion



`window.addOverlay = function(uuid, latitude, longitude)` aus der Datei `maps.html` nimmt diese Parameter entgegen und blendet, basierend auf diesen Informationen, ein neues GMarker-Objekt in die Kartendarstellung ein<sup>8</sup>.

```
1.     window.Mapping = {}; // UUID <> GMarker instance
2.     /**
3.     * Add an marker to the GMap2 instance.
4.     *
5.     * @param uuid UUID of geolocation instance
6.     * @param latitude
7.     * @param longitude
8.     */
9.     window.addOverlay = function(uuid, latitude, longitude) {
10.         var marker = new GMarker(new GLatLng(latitude, longitude),
11.             {draggable: true});
12.         window.GoogleMap.addOverlay(marker);
13.         GEvent.addListener(marker, 'dragend', function() {
14.             var point = marker.getLatLng();
15.             var newlat = point.lat();
16.             var newlng = point.lng();
17.             window.GeolocationUpdater.update(uuid, newlat, newlng);
18.             updateInfo(uuid, newlat, newlng);
19.         });
20.         if (window.HtmlCache[uuid]) {
21.             marker.bindInfoWindowHtml(window.HtmlCache[uuid],
22.                 {maxWidth:300});
23.         }
24.         if (window.EditableCache[uuid] === false) {
25.             marker.disableDragging();
26.         }
27.         window.EditableCache[uuid] = null;
28.         window.Mapping[uuid] = marker;
29.     };
```

Listing 16: Geolocation – GMarker-Objekt zur Kartendarstellung hinzufügen (`maps.html`)

Die Java-Umgebung (bzw. die Eingabekomponente `GeolocationSwingGadget`) muss über alle Änderung der Markierung (GMarker-Objekt) informiert werden. Wird

<sup>8</sup> Weitere Informationen siehe Google Maps-API-Referenz:

<http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/reference.html#GMarker>



beispielsweise eine Geokodierungs-Anfrage über die Google-Maps-API gesendet, so sollten die ermittelten Adress-Informationen in der SwingGadget-Eingabekomponente aktualisiert werden. Gleiches gilt für eine Änderung des Objekts `GMarker` innerhalb der Webapplikation (z. B. durch Verschieben der Markierung in der Kartendarstellung). Das bedeutet, die Informationen aus dem Objekt `GMarker` müssen wieder einem `GeolocationEntry` der Java-Umgebung zugeordnet werden. Dazu wird die Variable `uuid` verwendet.

Bei einer Aktualisierung über das Objekt `GeolocationUpdater` wird der `uuid`-Wert (sowie die geänderten Latitude- und Longitude-Werte) an die Update-Methode der `MapsPlugin`-Implementierung übergeben (vgl. Kapitel 4.3.5 Seite 65). Über den `uuid`-Wert kann eine erneute Zuordnung des Eintrags zu einer Eingabekomponente in der Java-Umgebung erfolgen. Verwendet wird dazu die interne Methode `GeolocationEntry get(final String uuid)` der `MapsPlugin`-Implementierung. Die Methode liefert den `GeolocationEntry` der Eingabekomponente mit dem übergebenen `uuid` zurück. Anschließend können die geänderten Latitude- und Longitude-Werte auf dem passenden `GeolocationEntry` innerhalb der Java-Umgebung aktualisiert werden. Der zur Eingabekomponente gehörigen `ModificationListener` wird über die Aktualisierung informiert.

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     private final List<GeolocationEntry> _entries = new
        ArrayList<GeolocationEntry>();
3.
4.     /**
5.      * Updates the geolocation of the specified geolocation entry and
6.      * may notify the related modification listener.
7.      *
8.      * @param uuid UUID of geolocation instance
9.      * @param latitude decimal latitude value
10.     * @param longitude decimal longitude value
11.     */
12.     public void update(final String uuid, final double latitude, final
        double longitude) {
13.         final GeolocationEntry location = get(uuid);
14.         if (location != null) {
15.             location.setLatitude(latitude);
16.             location.setLongitude(longitude);
17.             location.notifyPointModification();
18.         }
19.     }
20.     ...
```



```
21.      //--- inner methods ---//
22.      private GeolocationEntry get(final String uuid) {
23.          synchronized (_entries) {
24.              for (final GeolocationEntry entry : new
25.                  ArrayList<GeolocationEntry>(_entries)) {
26.                  if (entry.getUUID().equals(uuid)) {
27.                      return entry;
28.                  }
29.              }
30.              return null;
31.          }
32.      }
33.      ...
34.  }
```

Listing 17: Geolocation – Zuordnung des GeolocationEntries zurück in die Java-U. (MapsPlugin-Impl.)

#### 4.3.7 Listener – Auf Änderungen reagieren

Die FirstSpirit-AppCenter-API wurde um zwei Listener-Schnittstellen erweitert, die auf Ereignisse innerhalb der Browser-Instanz und innerhalb des Applikations-Tabs reagieren:

- Interface `TabListener` (Beschreibung der Schnittstelle siehe Kapitel 3.5, Seite 33, Beispiel siehe Kapitel 4.3.7.1, Seite 78).
- Interface `BrowserListener` (Beschreibung der Schnittstelle siehe Kapitel 3.8 Seite 38, Beispiel-Implementierung siehe Kapitel 4.3.7.2 Seite 80).

Neben diesen Standard-Schnittstellen der FirstSpirit-AppCenter-API wird für die Integration der Webanwendung Google Maps noch eine weitere Listener-Implementierung benötigt, die auf Ereignisse innerhalb der integrierten Anwendung reagiert, beispielsweise das Verschieben einer Markierung (GMarker) per Drag&Drop durch den Benutzer:

- Interface `ModificationListener` (Beschreibung der Schnittstelle siehe Kapitel 4.3.7.3 Seite 82, Beispiel siehe Kapitel 4.3.8 Seite 83).

Außerdem wird innerhalb der Beispiel-Implementierung noch ein Java-AWT-EventListener verwendet, der hier zumindest kurz erwähnt werden soll. Mithilfe eines `HierarchyListeners` kann die Eingabekomponente `GeolocationSwingGadget` in die Hierarchie der FirstSpirit-Eingabekomponenten eingehängt werden. Das ist notwendig, damit beim Wechsel der Arbeitsbereiche (bzw.



der Geolocation-Eingabekomponente) eine Aktualisierung des Applikationsbereichs stattfindet (siehe Kapitel 4.3.9 Seite 91).

#### 4.3.7.1 TabListener

Die Steuerung eines Applikations-Tabs kann über Ereignisse erfolgen. Um auf interne oder externe Ereignisse reagieren zu können, muss eine Instanz vom Typ `TabListener` auf dem Applikations-Tab registriert werden. Ein `TabListener` wird dem `ApplicationTab` über den Aufruf der Methode `addTabListener(...)` hinzugefügt. In diesem Beispiel implementiert die Klasse `MapsPlugin` das Interface `TabListener` selbst, daher müssen alle Methoden des Interfaces ebenfalls implementiert werden (Anmerkung: Ist das nicht gewünscht, sollte die zugehörige Adapter-Klasse verwendet werden (vgl. Kapitel 3.5)).

Beim Schließen des Applikations-Tabs durch den Benutzer, reagiert das FirstSpirit-Framework mit dem Aufruf der Methode `void tabClosed()`. Diese Methode wird innerhalb der Beispiel-Implementierung implementiert. Hintergrund: Die Implementierung sieht vor, dass jeder Eingabekomponente `CUSTOM_GEOLOCATION` eine bestimmte Markierung (einer geographischen Koordinate) innerhalb einer Google-Maps-Kartendarstellung zugeordnet wird (vgl. Kapitel 4.3.6). Verwaltet werden die `GeolocationEntries` in einer `ArrayList`. Schließt ein Redakteur das Applikations-Tab der Google Maps Integration, soll die Liste der `GeolocationEntries` verworfen werden.

Beim Selektieren des Applikations-Tabs durch den Benutzer, reagiert das FirstSpirit-Framework mit dem Aufruf der Methode `void tabSelected()`. Diese Methode wird innerhalb der Beispiel-Implementierung implementiert, um der aktuell selektierten Browser-Instanz den Fokus zu geben. Dazu wird der `Swing-EventQueue` über den Aufruf `SwingUtilities.invokeLater(new Runnable() {... die Methode BrowserApplication.focus() übergeben (vgl. Kapitel 3.7). Die Fokussierung wird damit erst am Ende aller Events asynchron ausgeführt.`

**Hintergrund:** Das Zoomen per Mausrad innerhalb der integrierten Webanwendung ist nur möglich, wenn die Anwendung auch den Fokus besitzt. Das kann entweder manuell durch den Benutzer erfolgen (Klick in die Anwendung) oder vom Entwickler implementiert werden, wie in diesem Beispiel.

```
1. public class MapsPlugin implements TabListener, ... {
2.     ...
3.     /**
4.         * Get BrowserApplication instance and may initialize it.
```



```
5.      *
6.      * @return BrowserApplication instance
7.      */
8.
9.      private BrowserApplication getApplication() {
10.         if (_application == null) {
11.             ...
12.             _tab = service.openApplication(...);
13.             _tab.addTabListener(this);
14.             ...
15.         }
16.         return _application;
17.     }
18.
19.     //--- TabListener ---//
20.
21.     public void tabSelected() {
22.         if (_focus) {
23.             SwingUtilities.invokeLater(new Runnable() {
24.                 public void run() {
25.                     _application.focus();
26.                 }
27.             });
28.             _focus = false;
29.         }
30.     }
31.
32.
33.     public void tabDeselected() {
34.     }
35.
36.
37.     public void tabClosed() {
38.         synchronized (_entries) {
39.             _entries.clear();
40.         }
41.         _active = false;
42.         _application = null;
43.     }
44. }
```

Listing 18: Geolocation – Registrierung und Verwendung des TabListeners (MapsPlugin-Impl.)



### 4.3.7.2 BrowserListener

Zusätzlich zum `TabListener` verwendet die Beispiel-Implementierung eine Instanz vom Typ `BrowserListener`. Mithilfe eines `BrowserListeners` kann der Zugriff auf die Inhalte der `Browser`-Instanz über Ereignisse gesteuert werden. Ein `BrowserListener` wird über den Aufruf der Methode `addBrowserListener(...)` auf der `BrowserApplication` registriert. In diesem Beispiel implementiert die Klasse `MapsPlugin` das Interface `BrowserListener` selbst, daher müssen alle Methoden des Interfaces ebenfalls implementiert werden (Anmerkung: Ist das nicht gewünscht, sollte die zugehörige Adapter-Klasse verwendet werden (vgl. Kapitel 3.8)).

Die Beispiel-Implementierung verwendet die Methode `void onDocumentComplete(...)`, für die Injektion des Java-Objekts `GeolocationUpdater` in die `Webbrowser`-Instanz (vgl. Kapitel 4.3.5 Seite 65). Die Methode wird aufgerufen, wenn der `BrowserListener` der `Browser`-Instanz (bzw. der Instanz vom Typ `BrowserApplication`) meldet, dass das `HTML`-Dokument (einschließlich aller Bilder) vollständig geladen wurde. Hintergrund: Dies ist notwendig, damit ein geordneter Zugriff auf den `Dom`-Baum (`w3c-DOM`) sichergestellt werden kann (Konzept `DOM`-Zugriff siehe Kapitel 2.2 Seite 18).

Daneben wird die Methode `void onLocationChange(@NotNull String url)` verwendet. Die Methode wird aufgerufen, wenn der `BrowserListener` meldet, dass sich die `URL` der `Browser`-Instanz (bzw. der Instanz vom Typ `BrowserApplication`) geändert hat. In diesem Fall wird der Boolean `_active` auf den Wert `false` gesetzt. Damit wird eine Aktualisierung der `Browser`-Instanz sowie eine Fokussierung unterbunden.

```
1. public class MapsPlugin implements ..., BrowserListener {
2.     ...
3.     private boolean _active;
4.     /**
5.      * Get BrowserApplication instance and may initialize it.
6.      *
7.      * @return BrowserApplication instance
8.      */
9.
10.    private BrowserApplication getApplication() {
11.        if (_application == null) {
12.            ...
13.            _tab = service.openApplication(...);
14.            ...
15.            _application = _tab.getApplication();
```



```
16.         _application.addBrowserListener(this);
17.         ...
18.     }
19.     return _application;
20. }
21.
22.
23. //--- BrowserListener ---//
24.
25.
26.     public void onLocationChange(@NotNull final String url) {
27.         _active = false;
28.     }
29.
30.
31.     public void onDocumentComplete(final String url) {
32.         final BrowserApplication application = getApplication();
33.
34.         // document is loaded; inject java/javascript bridge
35.         application.inject(new GeolocationUpdater(this),
36.             "GeolocationUpdater");
37.
38.         _active = true;
39.
40.         // initialize map viewport and marker
41.         updateBrowser();
42.
43.         if (_focus) {
44.             SwingUtilities.invokeLater(new Runnable() {
45.                 public void run() {
46.                     application.focus();
47.                 }
48.             });
49.             _focus = false;
50.         }
51.     }
52. }
```

Listing 19: Geolocation – Registrierung und Verwendung eines BrowserListeners (MapsPlugin-Impl.)



### 4.3.7.3 ModificationListener

Neben den beiden Standard-Listnern wird für die Beispiel-Implementierung noch eine weitere Schnittstelle benötigt – ein `ModificationListener`, der für die Aktualisierung der Informationen innerhalb der `SwingGadget`-Eingabekomponente verantwortlich ist.

*Das Interface `ModificationListener` ist kein Bestandteil der `FirstSpirit-AppCenter-API`, sondern ein Bestandteil der Beispiel-Implementierung zur `Google-Maps-Integration`.*

Die Eingabekomponente `GeolocationSwingGadget` speichert eine geographische Koordinate, bestehend aus zwei dezimalen Werten (geographischer Längen- und Breitengrad) und die zu dieser Koordinate gehörige, vollständige Adressinformation (Strasse, Stadt, Land)(vgl. Beschreibung der Eingabekomponente in Kapitel 4.3.1). Diese Werte werden von der Webanwendung Google Maps ermittelt und müssen aus der Browser-Instanz in die FirstSpirit-Java-Umgebung kommuniziert werden.

Die Klasse `GeolocationSwingGadget` implementiert dazu das Interface `ModificationListener`. Das Interface stellt die folgenden Methoden zur Verfügung:

```
1. package de.espirit.firstspirit.opt.geolocation.google;
2.
3. public interface ModificationListener {
4.
5.     void onModification(double latitude, double longitude);
6.
7.     void onModification(String address);
8. }
```

Listing 20: Geolocation – Interface `ModificationListener` (kein Bestandteil der `AppCenter-API`)

Die zugehörigen Methoden werden in der Beispiel-Implementierung implementiert und sind zuständig für:

- die Aktualisierung der Latitude-, Longitude-Werte
- die Aktualisierung der Adress-Informationen
- die Aktualisierung der Thumbnail-Darstellung

die innerhalb der Eingabekomponente angezeigt werden (siehe Kapitel 4.3.8 Seite 83). Die Methoden des Interfaces werden dann aufgerufen, wenn sich die Adressinformationen bzw. die Koordinate (Latitude- und Longitude-Werte) innerhalb der Browser-Instanz geändert haben.



#### 4.3.8 Geodaten der Eingabekomponente aktualisieren (JavaScript » Java)

Die Eingabekomponente `GeolocationSwingGadget` ist eng mit der Webanwendung Google Maps verknüpft. Das bedeutet, ändert sich die Geoinformation (bzw. die Position des `GMarker`-Objekts) innerhalb der integrierten Google-Maps-Anwendung, so müssen auch die Werte der zugehörigen `SwingGadget`-Eingabekomponente aktualisiert werden (Beschreibung des Anwendungsfalls siehe Kapitel 4.2.2, Seite 47).

Die Übernahme der Werte aus der Webanwendung in die FirstSpirit-Java-Umgebung wird über das Java-Objekt `GeolocationUpdater` angestoßen, das zu einem passenden Zeitpunkt in die Browser-Instanz injiziert wird (siehe Kapitel 4.3.5 Seite 65). Die Klasse `GeolocationUpdater` verfügt über die Methode `void update(String uuid, double latitude, double longitude)` und die Methode `void info(final String uuid, final String address)`, die die geänderten Werte an das Java-Objekt `GeolocationEntry` der `SwingGadget`-Eingabekomponente übermitteln und den zur Eingabekomponente gehörigen `ModificationListener` über die Aktualisierung informieren (Beschreibung des Interfaces `ModificationListener` siehe Kapitel 4.3.7.3).



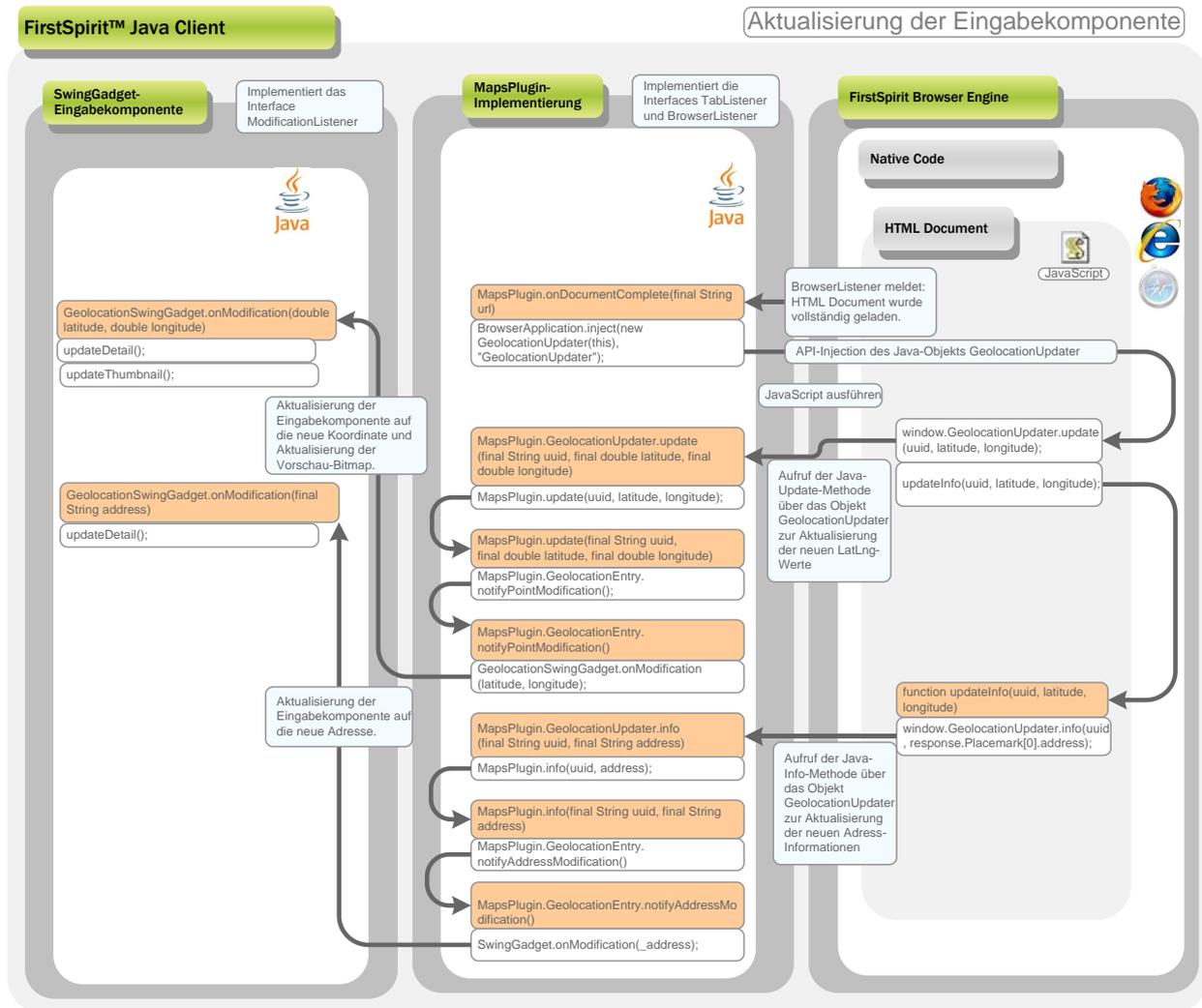


Abbildung 13: Aktualisierung der Eingabekomponente

Die für die Aktualisierung zuständigen Java-Methoden werden über JavaScript-Aufrufe aus dem HTML-Dokument heraus aufgerufen (vgl. Kapitel 4.3.5.). Das geschieht immer dann, wenn das GMarker-Objekt innerhalb der Google-Maps-Anwendung verändert wird, beispielsweise durch eine Drag&Drop-Aktion des Redakteurs innerhalb der Google-Maps-Kartendarstellung oder durch eine Such-Anfrage über das Adressfeld der SwingGadget-Eingabekomponente.



```
<html>
<head>
  <title>Google Maps</title>
  ...
  window.GeolocationUpdater = null;
  /**
   * Notify MapsPlugin about latitude/longitude modification.
   *
   * @param uuid UUID of geolocation instance
   * @param latitude
   * @param longitude
   */
  function updateInfo(uuid, latitude, longitude) {
    var request = new GetLocationsRequest(new GLatLng(latitude,
longitude));
    request.execute(function(response) {
      window.GeolocationUpdater.info(uuid,
response.Placemark[0].address);
    });
  }
  /**
   * Add an marker to the GMap2 instance.
   *
   * @param uuid UUID of geolocation instance
   * @param latitude
   * @param longitude
   */
  window.addOverlay = function(uuid, latitude, longitude) {
    var marker = new GMarker(new GLatLng(latitude, longitude),
{draggable: true});
    window.GoogleMap.addOverlay(marker);
    GEvent.addListener(marker, 'dragend', function() {
      var point = marker.getLatLng();
      var newlat = point.lat();
      var newlng = point.lng();
      window.GeolocationUpdater.update(uuid, newlat, newlng);
      updateInfo(uuid, newlat, newlng);
    });
    ...
  };
};
```

Listing 21: Geolocation – Aktualisierung über das Objekt GeolocationUpdater initiieren (maps.html)

Bei einer Änderung des GMarker-Objekts innerhalb der Webanwendung wird



zunächst `window.GeolocationUpdater.update(uuid, newlat, newlng)` aufgerufen (siehe Abbildung 13). Dieser Aufruf wird auf der Java-Seite ausgewertet und löst dort die Ausführung der Java-Methode `GeolocationUpdater.update(final String uuid, final double latitude, final double longitude)` aus (vgl. Kapitel 4.3.5 Seite 65). Der Methode werden die geänderten Latitude- und Longitude-Werte sowie ein `uuid`-Wert übergeben. Über den `uuid`-Wert wird der `GeolocationEntry` des `SwingGadgets` geholt. Verwendet wird dazu die interne Methode `GeolocationEntry.get(final String uuid)` der `MapsPlugin`-Implementierung (vgl. Kapitel 4.3.6). Anschließend werden die die geänderten Latitude- und Longitude-Werte auf dem `GeolocationEntry` gesetzt und der zugehörige `ModificationListener` über die Aktualisierung informiert. Dazu wird die Methode `GeolocationEntry.notifyPointModification()` aufgerufen. Diese Methode ruft wiederum die Methode `ModificationListener.onModification(double latitude, double longitude)` der inneren Klasse `GeolocationEntry` auf.

```
35. public class MapsPlugin implements TabListener, BrowserListener {
36.     private final List<GeolocationEntry> _entries = new
        ArrayList<GeolocationEntry>();
37.
38.     /**
39.      * Updates the geolocation of the specified geolocation entry and
40.      * may notify the related modification listener.
41.      *
42.      * @param uuid UUID of geolocation instance
43.      * @param latitude decimal latitude value
44.      * @param longitude decimal longitude value
45.      */
46.     public void update(final String uuid, final double latitude, final
        double longitude) {
47.         final GeolocationEntry location = get(uuid);
48.         if (location != null) {
49.             location.setLatitude(latitude);
50.             location.setLongitude(longitude);
51.             location.notifyPointModification();
52.         }
53.     }
54.     ...
55.     //--- inner methods ---//
56.     private GeolocationEntry get(final String uuid) {
57.         synchronized (_entries) {
```



```
58.         for (final GeolocationEntry entry : new
59.             ArrayList<GeolocationEntry>(_entries)) {
60.             if (entry.getUUID().equals(uuid)) {
61.                 return entry;
62.             }
63.         }
64.         return null;
65.     }
66.
67.     ...
68.     private static class GeolocationEntry {
69.         private ModificationListener _listener;
70.
71.         GeolocationEntry(..., final ModificationListener
72.             listener) {
73.             ...
74.             _listener = listener;
75.         }
76.         ...
77.         public void notifyPointModification() {
78.             if (_listener != null) {
79.                 _listener.onModification(_latitude, _longitude);
80.             }
81.         }
82.     }
83.
84.     public static class GeolocationUpdater {
85.
86.         private final MapsPlugin _mapsPlugin;
87.         ...
88.         public void update(final String uuid, final double
89.             latitude, final double longitude) {
90.             _mapsPlugin.update(uuid, latitude, longitude);
91.         }
92.     }
```

Listing 22: Geolocation – Aktualisierung des Latitude-, Longitude-Wertes (MapsPlugin-Impl.)

Die Klasse `GeolocationSwingGadget` implementiert das Interface `ModificationListener` (Beschreibung des Interfaces siehe Kapitel 4.3.7.3). Der Aufruf `onModification(double latitude, double longitude)` der



MapsPlugin-Implementierung wird damit direkt an die Methode `GeolocationSwingGadget.onModification(final double latitude, final double longitude)` weitergeleitet. Diese übernimmt die geänderten Latitude- und Longitude-Werte und sorgt, über den Aufruf der Methode `updateDetail()`, für die Aktualisierung der Werte in der Eingabekomponente. Das in der Eingabekomponente eingeblendete Vorschau-Bitmap wird, über den Aufruf der Methode `updateThumbnail()`, ebenfalls aktualisiert und zeigt anschließend die geänderte Position an.

```
1. public class GeolocationSwingGadget ... implements
   ModificationListener, ...{
2.     ...
3.     private double _valueLatitude;
4.     private double _valueLongitude;
5.     private String _valueAddress;
6.     ...
7.
8.     public void onModification(final double latitude, final double
       longitude) {
9.         _valueLatitude = latitude;
10.        _valueLongitude = longitude;
11.        updateDetail();
12.        updateThumbnail();
13.    }
14.
15.
16.    public void onModification(final String address) {
17.        _valueAddress = address;
18.        updateDetail();
19.    }
20. }
```

Listing 23: Geolocation – Verwendung des Interfaces `ModificationListener` (`SwingGadget-Impl.`)

Die Adress-Informationen, die zu den geänderten Koordinaten passen, müssen ebenfalls aktualisiert werden. Innerhalb der JavaScript-Umgebung wird dazu zunächst ein `Geolocation-Request` mit den geänderten Latitude-, Longitude-Werten an Google gesendet (`new GetLocationsRequest(new GLatLng(latitude, longitude));` vgl. Seite 118). Dieser Aufruf fordert eine detaillierte Adressinformation auf Grundlage der übergebenen Latitude- und Longitude-Werte an. Anschließend wird `window.GeolocationUpdater.info(uuid, response.Placemark[0]. address)` aufgerufen. Dieser Aufruf wird auf der Java-Seite ausgewertet und löst dort die Ausführung der Java-Methode



`GeolocationUpdater.info`(final String uuid, final String address) aus (vgl. Kapitel 4.3.5 Seite 65). Der Methode werden das Ergebnis der Geokodierungsanfrage sowie ein uuid-Wert übergeben. Über den uuid-Wert wird erneut der passende `GeolocationEntry` geholt. Verwendet wird dazu die interne Methode `GeolocationEntry.get`(final String uuid) der `MapsPlugin`-Implementierung (vgl. Kapitel 4.3.6).

Anschließend werden die Adress-Informationen auf dem `GeolocationEntry` gesetzt und der zuständige `ModificationListener` über die Aktualisierung informiert. Dazu wird die Methode

`GeolocationEntry.notifyAddressModification`() aufgerufen.

Diese Methode ruft wiederum die Methode

`ModificationListener.onModification`(final String address )  
der inneren Klasse `GeolocationEntry` auf (siehe Abbildung 13).

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     private final List<GeolocationEntry> _entries = new
           ArrayList<GeolocationEntry>();
3.
4.     /**
5.      * Updates the address information of the specified geolocation entry
6.      * and may notify the related modification listener.
7.      *
8.      * @param uuid UUID of geolocation instance
9.      * @param address address string of the related geolocation
10.     */
11.     public void info(final String uuid, final String address) {
12.         final GeolocationEntry location = get(uuid);
13.         if (location != null) {
14.             location.setAddress(address);
15.             location.notifyAddressModification();
16.         }
17.     }
18.     ...
19.     //--- inner methods ---//
20.     private GeolocationEntry get(final String uuid) {
21.         synchronized (_entries) {
22.             for (final GeolocationEntry entry : new
                   ArrayList<GeolocationEntry>(_entries)) {
23.                 if (entry.getUUID().equals(uuid)) {
24.                     return entry;
25.                 }

```



```
26.         }
27.         return null;
28.     }
29. }
30.
31. ...
32. private static class GeolocationEntry {
33.     private ModificationListener _listener;
34.
35.     GeolocationEntry(..., final ModificationListener
36.         listener) {
37.         ...
38.         _listener = listener;
39.     }
40.     ...
41.     public void notifyAddressModification() {
42.         if (_listener != null) {
43.             _listener.onModification(_address);
44.         }
45.     }
46. }
47.
48. public static class GeolocationUpdater {
49.
50.     private final MapsPlugin _mapsPlugin;
51.     ...
52.
53.     public void info(final String uuid, final String
54.         address) {
55.         _mapsPlugin.info(uuid, address);
56.     }
57. }
```

Listing 24: Geolocation – Aktualisierung der Adress-Informationen (MapsPlugin-Impl.)



### 4.3.9 Auf Baum-Navigations-Events reagieren (Java » JavaScript)

Die integrierte Webanwendung soll auf Ereignisse innerhalb des JavaClients reagieren. Das betrifft nicht nur Ereignisse, die über die Geolocation-Eingabekomponente angestoßen werden (beispielsweise die Adress-Suche über den Button "Koordinate suchen"), sondern auch Baum-Navigations-Events oder den Wechsel des aktiven Arbeitsbereichs durch den Redakteur. Bei der Auswahl einer neuen Geolocation-Eingabekomponente (beispielsweise über die Baum-Navigation) muss eine Aktualisierung der Kartendarstellung im Applikationsbereich erfolgen (vgl. Beschreibung zum Anwendungsfall in Kapitel 4.2.4, Überblenden der Kartendarstellung von einer Koordinate zur nächsten beim Wechsel zwischen unterschiedlichen Eingabekomponente (Seite 49 f.)). Hintergrund: Jeder Eingabekomponente wird ein GeolocationEntry zugeordnet. Beim Wechsel zwischen zwei Eingabekomponenten muss die aktuelle Kartendarstellung verworfen und durch eine neue Darstellung mit der neuen Koordinate ersetzt werden.

Mithilfe des Java-AWT-EventListeners `HierarchyListener` wird ein Mechanismus implementiert, der entscheidet, wann die Kartendarstellung im Applikationsbereich aktualisiert werden muss. Dazu wird die Eingabekomponente `GeolocationSwingGadget` in die Hierarchie der FirstSpirit-Eingabekomponenten eingehängt. Bei einer Änderung der Komponentenhierarchie, beispielsweise beim Wechsel des Tabs im mittleren Arbeitsbereichs, wird der `HierarchyListener` informiert und ruft die Methode `public void hierarchyChanged(HierarchyEvent e)` auf.

Die Klasse `GeolocationSwingGadget` implementiert diese Methode und ruft darin die Methode `void onShowing(boolean showing)` auf. Diese Methode regelt die Aktualisierung der Google-Maps-Anwendung im Applikationsbereich, abhängig von der Sichtbarkeit der Eingabekomponente im Arbeitsbereich des JavaClients. Die Aktualisierung der Browser-Instanz wird dabei über den Boolean `showing` gesteuert. Eine Aktualisierung soll nur erfolgen, wenn die Geolocation-Eingabekomponente auch im Arbeitsbereich des Redakteurs sichtbar ist. Beim Aufruf der Methode `void onShowing(boolean showing)` wird daher `e.getChanged().isShowing()` übergeben. Der `HierarchyEvent` liefert die Komponente zurück, die sich zum Zeitpunkt des Ereignisses an oberster Stelle der Komponentenhierarchie befindet, und prüft, ob diese Komponente sichtbar ist oder nicht.

Ist die Komponente sichtbar (`showing`) wird zunächst der initiale Kartentyp für die Darstellung im Applikationsbereich ausgewählt. Abhängig davon, ob die Eingabekomponente zum Bearbeiten gesperrt wurde, wird hier entweder der



Kartentyp Hybrid-Map gesetzt, oder (falls die Inhalte nicht geändert werden können), die 3D-Kartendarstellung (siehe Kapitel 4.3.18 Seite 113). Anschließend wird die Geolocation über den Aufruf der Methode `getValue()` ermittelt. Die Methode liefert eine Instanz der Klasse `GeolocationImpl` zurück. Der zurückgelieferte Wert wird an die Methode `addToMap(...)` übergeben, die den übergebenen Wert für eine Geolocation-Eingabekomponente registriert (vgl. Kapitel 4.3.6). Im nächsten Schritt wird die Methode `getMapsPlugin().updateBrowser()` aufgerufen, die die Aktualisierung der Browser-Instanz veranlasst (siehe Kapitel Seite).

```
1. public class GeolocationSwingGadget implements HierarchyListener
2.     ...
3.     {
4.     ...
5.     public JComponent getComponent() {
6.     ...
7.     _panel = new JPanel(layout);
8.     _panel.addHierarchyListener(this);
9.     ...
10.    }
11.    ...
12.    public void hierarchyChanged(final HierarchyEvent e) {
13.        if ((e.getChangeFlags() & HierarchyEvent.SHOWING_CHANGED) != 0)
14.            {
15.                onShowing(e.getChanged().isShowing());
16.            }
17.    }
18.
19.    private void onShowing(final boolean showing) {
20.        if (getMapsPlugin().isAccessible() && _valueSet) {
21.            GuiUtil.execute(new Runnable() {
22.                public void run() {
23.                    if (showing) {
24.                        final boolean isEditable = _editable;
25.                        if (isEditable) {
26.                            getMapsPlugin().setMapType(MapsPlugin.MapType.G_HYBRID_MAP);
27.                        } else {
28.                            getMapsPlugin().setMapType(MapsPlugin.MapType.G_SATELLITE_3D_MAP);
29.                        }
30.                        addToMap(getValue());
31.                        getMapsPlugin().updateBrowser();
```



```
32.         } else {
33.             getMapsPlugin().remove(getGadgetId());
34.         }
35.     }
36.     });
37. }
38. }
39. }
```

Listing 25: Geolocation – Verwendung des EventListeners `HierarchyListener` (SwingGadget-Impl.)

#### 4.3.10 Browser-Instanz aktualisieren (Java » JavaScript)

Die Browser-Instanz der integrierten Webanwendung muss abhängig von bestimmten Ereignissen aktualisiert werden.

Die Methode `updateBrowser()` aktualisiert die Browser-Instanz innerhalb der integrierten Vorschau des FirstSpirit-JavaClients. Die Methode wird nur ausgeführt, wenn bereits eine Google-Maps-Anwendung existiert und der Applikations-Tab im Vordergrund des Applikationsbereichs angezeigt wird.

Suche und Ermittlung der Adressdetails: Die Suche und die Ermittlung der Adress-Details erfolgt mittels der Google-Maps API. Die Geolocation und Adress-Detail Updates werden mittels des injizierten Objektes an den JavaClient weitergegeben und führen zu einer Aktualisierung der entsprechenden Eingabekomponente. Die Anfahrtsbeschreibung - bzw. der Aufruf der entsprechenden Google-Maps / Bing Seite - wird über ein Formular innerhalb des Absatztemplates realisiert. Google-Maps und Bing haben verständlicher Weise unterschiedliche URL-Parameter die jeder für sich berücksichtigt werden mussten.

```
1.  public class MapsPlugin implements TabListener, BrowserListener {
2.      ...
3.      private BrowserApplication _application;
4.      private boolean _active;
5.      ...
6.
7.      /**
8.       * Updates browser and google map instance.
9.       * This method must be called to update geolocation entries and
10.      * viewport.
11.     */
12.     public void updateBrowser() {
13.         if (_application != null && _active) {
```



```
14.         // build script code to update google map
15.         final StringBuilder buf = new StringBuilder();
16.         buf.append(getScriptClearOverlays());
17.         buf.append(getScriptSetMapType(_mapType));
18.         for (final GeolocationEntry location : new
19.             ArrayList<GeolocationEntry>(_entries)) {
20.             if (location.isSearchMode()) {
21.                 buf.append(getScriptFindAddress(location));
22.             } else {
23.                 buf.append(getScriptAddOverlay(location));
24.             }
25.             buf.append(getScriptSetEditable(location));
26.             buf.append(getScriptSetInfoHtml(location));
27.         }
28.         buf.append(getScriptSetViewport());
29.         final String script = buf.toString();
30.
31.         // prevent unnecessary rapidly script executions (may
32.         //interfere camera movement), caused by multiple showing
33.         //events
34.         if (_updateScript == null ||
35.             !_updateScript.equals(script) || (_updateTime +
36.                 EQUAL_UPDATE_TIMEOUT < System.currentTimeMillis())) {
37.             _application.executeScript(script);
38.             _updateScript = script;
39.             _updateTime = System.currentTimeMillis();
40.         }
41.     }
42. }
43. /**
44.     * Builds script code to clear all overlays from map.
45.     *
46.     * @return script code
47.     */
48. private String getScriptClearOverlays() {
49.     return "window.clearOverlays();";
50. }
51.
52. /**
53.     * Builds script code for map type switch.
54.     *
55.     * @param type map type to show
56.     * @return script code
```



```
*/
private String getScriptSetMapType(final MapType type) {
    final StringBuilder buf = new StringBuilder();
    buf.append("window.setMapType(").append(type).append(");");
    return buf.toString();
}

48. }
49. /**
50.     * Build script code for an address search entry.
51.     *
52.     * @param location related geolocation entry
53.     * @return script code
54.     */
55.     private String getScriptFindAddress(final GeolocationEntry
location) {
56.         final StringBuilder buf = new StringBuilder();
57.         final String pattern =
location.getAddress().replaceAll("'", "\\\'");
58.         buf.append("window.findAddress('")
59.             .append(location.getUUID()).append("'", "'")
60.             .append(pattern).append("'");");
61.         return buf.toString();
62.     }
63.     /**
64.     * Builds script code to modify editable-state of related
geolocation entry.
65.     *
66.     * @param location related geolocation entry.
67.     * @return script code
68.     */
69.     private String getScriptSetEditable(final GeolocationEntry
location) {
70.         final StringBuilder buf = new StringBuilder();
71.
        buf.append("window.setEditable('").append(location.getUUID()).append("
', ").append(location.isEditable()).append(");");
72.         return buf.toString();
73.     }
74.     /**
75.     * Build script code to update the info html balloon.
76.     *
77.     * @param location entry to build info html for.
```



```
78.         * @return script code
79.         */
80.         private String getScriptSetInfoHtml(final GeolocationEntry
location) {
81.             final StringBuilder buf = new StringBuilder();
82.
            buf.append("window.setInfoHtml('").append(location.getUUID()).append("
', ");
83.             String text = location.getInfoText();
84.             if (location.getInfoPicture() != null || text != null) {
85.                 buf.append("\"<span style='font-family: Arial,
Sans-Serif; font-size: 11px;'>");
86.                 if (location.getInfoPicture() != null) {
87.                     buf.append("<img align='right'
src='").append(location.getInfoPicture()).append("' />");
88.                 }
89.                 if (text != null) {
90.                     text = text.replaceAll("\n", "<br/>");
91.                     text = text.replaceAll("\"", "&quot;");
92.                     buf.append(text);
93.                 }
94.                 buf.append("</span>\"");
95.             } else {
96.                 buf.append("null");
97.             }
98.             buf.append(");");
99.             return buf.toString();
100.        }
101.    /**
102.        * Builds script code to modify viewport and show registered
geolocation entries on map.
103.        *
104.        * @return script code.
105.        */
106.        private String getScriptSetViewport() {
107.            synchronized (_entries) {
108.                final StringBuilder buf = new StringBuilder();
109.                final MapViewport viewBounds = new MapViewport();
110.                for (final GeolocationEntry entry : _entries) {
111.                    viewBounds.include(entry);
112.                }
113.                if (viewBounds.getPointCount() == 1) {
114.                    final double lat =
```



```
viewBounds.getMinLatitude();
115.         final double lng =
viewBounds.getMinLongitude();
116.         buf.append("window.setViewPoint(")
117.             .append(lat).append(',')
118.             .append(lng).append(")");
119.     } else if (viewBounds.getPointCount() > 1) {
120.         final double minlat =
viewBounds.getMinLatitude();
121.         final double minlng =
viewBounds.getMinLongitude();
122.         final double maxlat =
viewBounds.getMaxLatitude();
123.         final double maxlng =
viewBounds.getMaxLongitude();
124.         buf.append("window.setViewBounds(")
125.             .append(minlat).append(',')
126.             .append(minlng).append(',')
127.             .append(maxlat).append(',')
128.             .append(maxlng).append(")");
129.     }
130.     return buf.toString();
131. }
132. }
```

Listing 26: Geolocation –UpdateBrowser (MapsPlugin-Impl.)

### 4.3.11 MapsPlugin – Adress-Suche (Google-Geolocation)

Anwendungsfall (siehe Kapitel 4.2.1 Seite 46).

Einstiegspunkt ist beispielsweise ein Klick auf den Search-Button der Geolocation-Eingabekomponente (vgl. Kapitel 4.3.11 Seite 97). Diese Aktion löst den Aufruf der internen Search-Methode aus. Die search-Methode der SwingGadget-Implementierung übergibt den `GadgetIdentifizier` der Eingabekomponente, den Adress-String aus dem Textfeld und den `ModificationListener` (siehe Kapitel 4.3.7.3 Seite 82) an die search-Methode der MapsPlugin-Implementierung.

```
1.     public class GeolocationSwingGadget {
2.         ...
3.         public JComponent getComponent() {
4.             ...
5.             final AbstractAction searchAction = new AbstractAction() {
```



```
6.         public void actionPerformed(final ActionEvent e) {
7.             search();
8.             getMapsPlugin().updateBrowser();
9.             getMapsPlugin().focusBrowser();
10.        }
11.    };
12.    ...
13. }
14.
15. private void search() {
16.     final String address = _searchField.getText();
17.     ...
18.     getMapsPlugin().search(getGadgetId(), address, this);
19.     ...
20.     updateMapsPlugin(getGadgetId());
21. }
22. }
```

Dort wird zunächst der Suchmodus eingeschaltet, aber noch keine Suche über die Google Maps-API gestartet:

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     public void search(final GadgetIdentifier gadgetId,...) {
3.         entry1.setSearchMode(true);
4.         entry1.setAddress(search);
5.     }
6. }
```



Das Suchen einer neuen geographischen Position erfolgt über den Search-Button der SwingGadget-Eingabekomponente `CUSTOM_GEOLOCATION`. Der Redakteur kann eine Adresse bzw. einen Adressbestandteil (beispielsweise einen Straßennamen) in das dazu vorgesehene Textfeld eingeben und mit einem Klick auf den Button die Suche innerhalb der integrierten Webapplikation starten (siehe Kapitel 4.2.1 Seite 46).

```
1. public class GeolocationSwingGadget...{
2.     private JTextField _searchField;
3.     ...
4.     private void search() {
5.         final String address = _searchField.getText();
6.         if (!StringUtil.isEmpty(address)) {
7.             getMapsPlugin().search(getGadgetId(), address, this);
8.         }
9.         updateMapsPlugin(getGadgetId());
10.    }
11. }
```

Listing 27: Geolocation –Suche mittels eines Adress-Strings (SwingGadget-Impl.)

Die Methode `MapsPlugin` `getMapsPlugin()` erzeugt zunächst die Singleton-Instanz der Klasse `MapsPlugin` (vgl. Kapitel 4.3.2 Seite 56). Die Klasse `MapsPlugin` verfügt über eine eigene `search`-Methode. Die `search`-Methode der `SwingGadget`-Implementierung übergibt den `GadgetIdentifier` der Eingabekomponente (`GadgetIdentifier` siehe Kapitel Seite), den Adress-String aus dem Textfeld und den `ModificationListener` an die `search`-Methode der `MapsPlugin`-Implementierung:

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.     private final List<GeolocationEntry> _entries = new
3.         ArrayList<GeolocationEntry>();
4.
5.     /**
6.      * Register a geolocation instance with the specified address string
7.      * and modification listener
8.      *
9.      * @param gadgetId gadget id of geolocation instance
10.     * @param search address string of geolocation
11.     * @param listener modification listener, may be null
12.     */
13.     public void search(final GadgetIdentifier gadgetId, final String
14.         search, final ModificationListener listener) {
15.         ensureApplicationTab();
16.     }
17. }
```



```
14.     final GeolocationEntry entry = get(gadgetId);
15.     if (entry != null) {
16.         _entries.remove(entry);
17.     }
18.     GeolocationEntry entry1 = get(gadgetId);
19.     if (entry1 == null) {
20.         entry1 = new GeolocationEntry(gadgetId, 0, 0, listener);
21.         synchronized (_entries) {
22.             _entries.add(entry1);
23.         }
24.     }
25.     entry1.setModificationListener(listener);
26.     entry1.setSearchMode(true);
27.     entry1.setAddress(search);
28. }
29. }
```

Listing 28: Geolocation –Suche mittels eines Adress-Strings (MapsPlugin-Impl.)

Dort wird über die Methode `ensureApplicationTab()` zunächst geprüft, ob bereits eine Browser-Instanz der Webapplikation im Applikationsbereich des `JavaClients` existiert. Ist noch keine Browser-Instanz vorhanden, wird eine neue Instanz erzeugt (siehe Kapitel 4.3.3 Seite 57).

Um einen neuen Eintrag innerhalb der Google-Maps-Integration anzuzeigen, muss zuvor ermittelt werden, ob bereits alte Einträge existieren. Die Beispiel-Implementierung verwendet dazu die innere Methode `GeolocationEntry get(final GadgetIdentifizier gadgetId)`. Die Methode liefert einen `GeolocationEntry` zurück (sofern für die Komponente mit der übergebenen `Gadget-ID` ein Eintrag gespeichert wurde) oder `null` (wenn kein Eintrag vorhanden ist).

```
1.     public class MapsPlugin implements TabListener, BrowserListener {
2.         private final List<GeolocationEntry> _entries = new
           ArrayList<GeolocationEntry>();
3.
4.         private GeolocationEntry get(final GadgetIdentifizier gadgetId) {
5.             synchronized (_entries) {
6.                 for (final GeolocationEntry entry : new
7.                     ArrayList<GeolocationEntry>(_entries)) {
8.                     if (entry.getGadgetId().equals(gadgetId)) {
9.                         return entry;
10.                    }
11.                }
12.                return null;
            }
        }
```



```
13.         }
14.     }
15. }
```

Listing 29: Geolocation –Innere Methode: GeolocationEntry holen (MapsPlugin-Impl.)

Falls bereits ein GeolocationEntry existiert, wird der zurückgelieferte Eintrag anschließend mithilfe der remove-Methode aus der Liste der GeolocationEntries entfernt (siehe Beispiel auf Seite 100).

Anschließend wird von der search-Methode erneut die innere Methode GeolocationEntry get(final GadgetIdentifizier gadgetId) aufgerufen. Diesmal ist kein Eintrag vorhanden, die Methode liefert null zurück. Die search-Methode legt daraufhin eine neue Instanz vom Typ GeolocationEntry mit den Koordinaten 0.0/0.0 an und fügt diesen der Liste der GeolocationEntries hinzu (siehe Beispiel auf Seite 100).

Im nächsten Schritt wird die Methode setSearchMode(final boolean searchMode) aufgerufen, die der Such-Modus für den neuen Eintrag aktiviert. Über die Methode void setAddress(final String address) wird anschließend der Suchstring aus dem Eingabefeld gespeichert.

Die SwingGadget-Implementierung ruft nun die Methode void updateMapsPlugin(final GadgetIdentifizier gadgetId) auf (vgl. Beispiel auf Seite 99).

```
1. public class GeolocationSwingGadget...{
2.     private boolean _editable;
3.     ...
4.     private void updateMapsPlugin(final GadgetIdentifizier gadgetId) {
5.         getMapsPlugin().setEditable(gadgetId, _editable);
6.         getMapsPlugin().setInfoPicture(gadgetId, getInfoPicture(),
7.             "image/jpeg");
8.         getMapsPlugin().setInfoText(gadgetId, getInfoText());
9.     }
10. }
```

Listing 30: Geolocation –MapsPlugin aktualisieren (SwingGadget-Impl.)



Auf der Instanz vom Typ `MapsPlugin` werden hier weitere Methoden der `MapsPlugin`-Implementierung aufgerufen. Zunächst wird der Bearbeitungsmodus der Komponente eingeschaltet, anschließend ein Info-Bild und ein Info-Text hinzugefügt:

```
1. public class MapsPlugin implements TabListener, BrowserListener {
2.
3.     /**
4.      * Modify editable-state of an google map marker entry specified by the
5.      * gadget id.
6.      * @param gadgetId gadget id of the related geolocation instance
7.      * @param editable new editable-state
8.      */
9.     public void setEditable(final GadgetIdentifier gadgetId, final
10.        boolean editable) {
11.         final GeolocationEntry entry = get(gadgetId);
12.         if (entry != null) {
13.             entry.setEditable(editable);
14.         }
15.
16.     private static class GeolocationEntry {
17.         ...
18.         private boolean _editable;
19.
20.         public void setEditable(final boolean editable) {
21.             _editable = editable;
22.         }
23.     }
24. }
```

Listing 31: Geolocation –... (MapsPlugin-Impl.)



```
1. public class GeolocationSwingGadget...{
2.     ...
3.     public JComponent getComponent() {
4.         @SuppressWarnings({"serial"})
5.         final AbstractAction searchAction = new AbstractAction() {
6.             public void actionPerformed(final ActionEvent e) {
7.                 search();
8.                 getMapsPlugin().
9.                     setMapType(MapsPlugin.MapType.G_HYBRID_MAP);
10.                    getMapsPlugin().updateBrowser();
11.                    getMapsPlugin().focusBrowser();
12.            }
13.        };
14.    }
```

Listing 32: Geolocation –UpdateBrowser (SwingGadget-Impl.)

Für die Adress-Suche wird dieser Eintrag zunächst mit den Koordinaten 0.0/0.0 neu angelegt (siehe Kapitel 4.3.11 Seite 97). Anschließend wird der Adress-String aus der Eingabekomponente für den GeolocationEntry gesetzt (siehe Listing 7).



### 4.3.12 maps.html – Einführung

Prinzipiell stehen zwei unterschiedliche Wege zur Verfügung, um eine Webapplikation in FirstSpirit zu integrieren. Es kann zum einen eine globale Webapplikation implementiert und auf dem FirstSpirit-Server installiert werden. In diesem Fall kann im integrierten Browser die URL der Webapplikation aufgerufen werden. Soll eine eigenständige Webapplikation umgangen werden, kann aber auch einfach der gewünschte HTML-Code initiiert und aufgerufen werden. Dieser zweite Weg wird auch für die hier vorgestellte Google Maps Integration verwendet.

Zur Initiierung des HTML-Codes wird die Datei maps.html benötigt. Die Datei besteht zunächst aus einem einfachen HTML-Grundgerüst. Dieses Grundgerüst muss nun um die passenden JavaScript bzw. Google Maps-API-Ausdrücke erweitert werden:

- Laden der Google Maps-API
- Container für die Kartendarstellung initialisieren
- Neues Kartenobjekt erstellen
- Karte auf eine Koordinate zentrieren
- Kartentyp definieren
- Karte über Ereignisse laden
- Adressdaten konvertieren – Geocoding
- GeolocationUpdater (Injection Java / JavaScript)

Dazu muss dem Browser zunächst mitgeteilt werden, dass er eine externe JavaScript-Datei laden muss

Die Datei maps.html besitzt ein windows-Objekt.

Dargestellt wird die Karte in einem Google Maps Container:

Um eine eigenständige Webapplikation zu vermeiden, wird hier direkt ein HTML-Code initiiert, der beim Laden einen Google-Maps-Container initialisiert.

Die Verbindung zwischen Web und Java erfolgt mittels der mit [TS#87759](#) [AN FS42\_4-N3: Durchstichsprototypen für die Applikationsintegration | API-Erweiterungen] entwickelten Applikationsintegrations API. Um eine eigenständige Webapplikation zu vermeiden wird hier direkt ein HTML-Code initiiert der beim Laden einen Google-Maps Container initialisiert. Innerhalb dieser HTML-Seite werden JavaScript Methoden definiert die Kernfunktionalitäten wie "Eintrag Hinzufügen", "Eintrag Entfernen" und "Viewport modifizieren" bereitstellen. Diese JavaScript Methoden werden auf Java-Seite entsprechend aufgerufen um Geolocations anzuzeigen oder modifizierbar zu machen. Für die Identifikation der



einzelnen Map-Einträge werden IDs erzeugt mittels der eine direkte Zuordnung und Steuerung erfolgt. Für die Modifikation der Geolocation und die entsprechende Änderungsnotifizierung ist hier ein Rückweg nötig der mittels einer Objekinjektion bereitgestellt wird. Dieses Objekt hat eine Methode für die Aktualisierung der exakten Geolocation sowie eine Methode mittels der der Adress-String aktualisiert werden kann.

### 4.3.13 Exkurs: HTML und JavaScript

Für das Verständnis der nachfolgenden Kapitel sind Kenntnisse in HTML und JavaScript vorteilhaft, aber nicht zwingend notwendig. Dieses Kapitel bietet eine kurze Einarbeitung zur Erläuterung der wichtigsten HTML-Tags und JavaScript-Objektypen, die innerhalb der Beispiel-Implementierung verwendet werden.

Standard-HTML-Grundgerüst:

```
<html>
  <head>
</head>
  <body>
</body>
</html>
```

**JavaScript:** Über die Skriptsprache JavaScript können HTML-Seiten, die innerhalb eines Browsers angezeigt werden (also bereits vom Besucher der Seite geladen wurden) nachträglich verändert werden. JavaScript ist eine ereignisgesteuerte Sprache. Bestimmte Aktionen des Benutzers (z. B. ein Mausklick), können zu einer Änderung der Seite führen. JavaScript ist eng mit dem Browser verknüpft, der die HTML-Seite anzeigt und bietet Objekte und Methoden zur Steuerung des Browserfensters (siehe Objekt window) und zur Manipulation des DOM-Baums einer HTML-Seite (siehe DOM).

Innerhalb der HTML-Datei muss dem Browser mitgeteilt werden, das es sich beim nachfolgenden Code um ein JavaScript handelt. Dazu wird innerhalb des Script-Tags der MIME-Typ ("**text/javascript**") angegeben. Das öffnende bzw. schließende Script-Tag definieren, wo das JavaScript beginnt und wo es endet und bilden damit die Begrenzungen des Skripts.

```
<html>
  <head>
    <script type="text/javascript">
      ...
```



```
</script>
</head>
<body>
</body>
</html>
```

**window-Objekt:** Jedes Browser-Fenster bzw. jeder Frame besitzt ein window-Objekt. Das window-Objekt wird automatisch von der Browserinstanz zur Verfügung gestellt, muss also nicht zuvor über den new-Operator erzeugt werden. Das Objekt stellt Methoden zur Modifikation der Anzeige (z. B. Statusleiste einblenden) und zur Steuerung mithilfe von Ereignissen (Events) zur Verfügung.

**Document Object Modell (DOM):** Das DOM ist ein Standard für den Zugriff auf die Inhalte eines HTML-Dokuments. Dabei werden die einzelnen HTML-Elemente eines HTML-Dokuments über eine Baumstruktur in eine Beziehung zueinander gesetzt. Anhand dieser Struktur kann das Skript, ausgehend von einem Element, zu einem anderen Element der Seite navigieren und dieses ggf. verändern (beispielsweise ein Tag einfügen). Über die entsprechenden Methoden des DOM können Tags an bestimmte Stellen der HTML-Seite eingefügt, Attribute manipuliert oder Elemente umstrukturiert werden. Im nachfolgenden Beispiel wird beispielsweise die Google Maps Karte innerhalb eines DIV-Elements in die HTML-Seite eingefügt.

**document-Objekt:** Das document-Objekt liegt in der Objekthierarchie unterhalb des window-Objekts und ist zuständig für die Inhalte, die innerhalb einer Browserinstanz angezeigt werden. Das document-Objekt ist der Wurzelknoten der Baumstruktur innerhalb des DOMs. Der Zugriff auf die untergeordneten HTML-Elemente erfolgt über den Aufruf der vom DOM zur Verfügung gestellten Methoden, z. B. getElementById(...). In diesem Fall wird ein Element der HTML-Seite, das ein id-Attribut besitzt geladen. Im nachfolgenden Beispiel wird so das DIV-Element, das die Google Maps Karte anzeigt, referenziert. Analog zum window-Objekt wird auch das document-Objekt automatisch von der Browserinstanz zur Verfügung gestellt, muss also nicht zuvor über den new-Operator erzeugt werden.



#### 4.3.14 maps.html - Laden der Google Maps-API

Bevor die Google Maps-API verwendet werden kann, muss dem integrierten Browser zunächst mitgeteilt werden, dass er eine externe JavaScript-Datei laden muss. Dies wird innerhalb der Datei maps.html über den folgenden Aufruf erreicht:

```
1. <html>
2. <head>
3. <title>Google Maps</title>
4. <script type="text/javascript"
   src="http://maps.google.com/maps?file=api&v=2&key="></script>
5. <script type="text/javascript">
6. </head>
7. ...
8. </html>
```

Innerhalb der Script-Tags wird dem Browser hier mitgeteilt, dass er eine externe JavaScript-Datei vom URL `http://maps.google.com/maps?file=api&v=2&key=myKey` laden soll (weitere Informationen siehe Kapitel Exkurs: HTML und JavaScript Seite 105 f.). Diese JavaScript-Datei enthält alle erforderlichen Funktionen, Klassen und Bibliotheken, die für den Einsatz der Google Maps-API erforderlich sind. Der hier übergebene Schlüssel (`key=...`), muss dem zuvor generierten, individuellen Google Maps-API-Schlüssel entsprechen (siehe Kapitel 4.1.3 Seite 43). Soll die Beispiel-Implementierung verwendet werden, muss dazu innerhalb des Script-Tags der individuelle Schlüssel ergänzt werden (z. B. `SRC="http://maps.google.com/maps?file=api&v=2&key=myKey"`).

#### 4.3.15 maps.html - Container für die Kartendarstellung initialisieren

Ein zentrales Element der Google Maps Integration ist die Kartendarstellung. Diese Darstellung erfolgt extern über die Google Maps-API. Zur Anzeige der Karte im integrierten Browser muss jedoch ein HTML-Element bereitgestellt werden, das als eine Art Platzhalter bzw. Container für die Karte fungiert. In der Datei maps.html wird dazu ein DIV-Element mit einem Attribut `id ("container")` innerhalb der `<body>`-Tags definiert.

```
1. <body>
2. <div id='container'></div>
3. </body>
```

Alle HTML-Knoten der Seite sind untergeordnete Objekte des JavaScript-Objekts `document` (weitere Informationen siehe Kapitel Exkurs: HTML und JavaScript Seite



105 f.). Über das id-Attribut (hier: "container") kann das DIV-Element im Document Object Model (DOM) des Browsers referenziert werden (weitere Informationen siehe Kapitel Exkurs: HTML und JavaScript Seite 105 f.). Dazu wird die Methode `getElementById(...)` auf dem `document`-Objekt aufgerufen:

```
document.getElementById('container');
```

Die Google Maps Kartendarstellung passt sich automatisch an die Größe des DIV-Containers an, in dem sie angezeigt wird. Die Darstellung des DIV-Containers (z. B. Weite, Höhe) kann innerhalb der Style-Tags über die entsprechenden Stilattribute verändert werden.

```
1. <style>
2.   ...
3.   #container {
4.     position: absolute;
5.     visibility: hidden;
6.     width: 100%;
7.     height: 100%;
8.   }
9. </style>
```

Das Initialisieren des DIV-Containers (bzw. der Kartendarstellung) wird über Ereignisse gesteuert, die sicherstellen, dass alle erforderlichen Dateien, zum Ausführen der JavaScript-Funktionen bereits geladen wurden (siehe Kapitel 4.3.19 Seite 115). Tritt einer der Events ein, wird die JavaScript –Funktion `init()` aufgerufen, die zuerst ein neues Kartenobjekt erzeugt (siehe Kapitel 4.3.16 Seite 109).



### 4.3.16 maps.html – Neues Kartenobjekt erstellen

Mithilfe des JavaScript-Operators `new` erzeugt die Funktion `init()` eine neue Instanz vom Typ `GMap2` (zur Beschreibung der Klasse und des Konstruktors siehe Google Maps API Dokumentation<sup>9</sup>).

```
1.   window.GoogleMap = null; // GMap2 instance
2.   ...
3.
4.   function init() {
5.     window.GoogleMap = new GMap2(document.getElementById('container'), {
6.       mapTypes: [G_NORMAL_MAP, G_SATELLITE_MAP, G_HYBRID_MAP,
7.                 G_SATELLITE_3D_MAP]
8.     });
9.     ...
10.  }
```

`GMap2` ist die zentrale Klasse der Google Maps-API, die die Karte repräsentiert. Damit die Karte im integrierten Browser angezeigt werden kann, muss sie mit einem DOM-Knoten der HTML-Seite verbunden werden. Dazu wird dem Konstruktor das zuvor definierte `DIV`-Element übergeben. Außerdem werden dem Konstruktor die gewünschten Kartentypen übergeben. Die Beispiel-Implementierung verwendet die folgenden Kartentypen:

- `G_NORMAL_MAP`: Anzeige der Karte mit Standard-2D-Kacheln.
- `G_SATELLITE_MAP`: Anzeige der Karte mit Fotokacheln (Satellitenbildern).
- `G_HYBRID_MAP`: Anzeige der Karte mit Fotokacheln und Kacheln für die Anzeige von markanten Funktionen, wie Straßen und Ortsnamen.
- `G_SATELLITE_3D_MAP`: Anzeige der Karte als interaktives 3D-Modell der Erde mit Satellitenbildern.

Weitere Eigenschaften, beispielsweise für das Zoomen innerhalb der Karte, werden über die entsprechenden Methoden der Google Maps-API definiert<sup>10</sup>. Über die Methode `addControl(...)` werden der Karte Bedienelement hinzugefügt. Innerhalb der `init`-Funktion sind das ein Bedienelement zum Schwenken und Zoomen (`GLargeMapControl`), ein Bedienelement zur Änderung des Maßstabs (`GScaleControl`) und Schaltflächen zum Umschalten zwischen den definierten Kartentypen (z. B. Karte und Satellit) (`GMapTypeControl`).

---

<sup>9</sup> <http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/introduction.html>

<sup>10</sup> siehe <http://code.google.com/intl/de-DE/apis/maps/documentation/javascript/v2/reference.html#GMap2>



### 4.3.17 maps.html - Karte zentrieren (Mittelpunkt bzw. Anzeigebereich)

Im nächsten Schritt muss die Kartendarstellung initialisiert werden. Karten bzw. Objekte vom Typ GMap2 müssen zentriert werden, bevor sie in einer HTML-Seite angezeigt werden können. Diese Initialisierung wird durch den Aufruf der Methode `setCenter()` auf der Instanz vom Typ GMap2 erreicht (siehe Kapitel 4.3.16 Seite 109).

Dazu muss zunächst die Start-Koordinate definiert werden, die in der Karte anzeigen werden soll, nachdem sie geladen wurde. Die Google Maps-API stellt dazu die Klasse `GLatLng`<sup>11</sup> zur Verfügung. Ein Objekt vom Typ `GLatLng` ist ein geographischer Punkt, der über die zugehörigen Koordinaten, bestehend aus dem geographischen Breiten- und Längengrad, bzw. Latitude und Longitude definiert wird (abgekürzt: `LatLng`). Ein neues Objekt vom Typ `GLatLng` wird über den `new`-Operator erzeugt:

```
new GLatLng(latitude, longitude)
```

Die Koordinaten einer neuen Kartendarstellung werden initial auf 0.0/0.0 gesetzt (siehe Kapitel Seite).

Die `GLatLng`-Koordinate muss nun der Methode `setCenter()` übergeben werden. Optional können auch eine Zoomstufe und ein Kartentyp übergeben werden:

```
<html>
  <head>
    <script ...>
    ...
    window.GoogleMap = null; // GMap2 instance
    window.DefaultMapZoom = 18; // Zoom for ~200 meter

    function(latitude, longitude) {
      ...
      var zoomLevel = window.DefaultMapZoom;
      window.GoogleMap.setCenter(new GLatLng(latitude, longitude), zoomLevel);
      ...
    }

    </script>
  </head>
  <body>
```

<sup>11</sup> siehe <http://code.google.com/intl/de-DE/apis/maps/documentation/javascript/v2/reference.html#GLatLng>



```
<div id='container'></div>
</body>
</html>
```

Die Methode `setCenter()` setzt nun den Mittelpunkt der Karte auf die übergebene Koordinate (unter Verwendung der optional angegebenen Zoomstufe und des Kartentyps). Diese Methode muss immer initial ausgeführt werden, bevor andere Operationen auf der Karte stattfinden dürfen. Das gilt auch für die Definition weiterer Kartenattribute, beispielsweise dem Kartentyp (siehe Kapitel 4.3.18 Seite 113).

Innerhalb der Beispiel-Implementierung wird die Karte in zwei JavaScript-Funktionen verwendet. Die `function(latitude, longitude)` ermittelt einen Mittelpunkt der Kartenansicht anhand einer einzelnen Koordinate, die `function(minLatitude, minLongitude, maxLatitude, maxLongitude)` ermittelt einen sichtbaren, rechteckigen Auswahlbereich der Kartenansicht anhand von zwei Koordinaten, die die Grenzen des Auswahlbereichs definieren.

```
<html>
  <head>
    <script ...>
    ...
    window.GoogleMap = null; // GMap2 instance

    /**
     * Set viewport to specified latitude/longitude position.
     *
     * @param latitude
     * @param longitude
     */
    window.setViewPoint = function(latitude, longitude) {
      if (window.GoogleMap.getCurrentMapType() == G_SATELLITE_3D_MAP) {
        window.GoogleMap.getEarthInstance(function(ge) {
          if (ge) {
            ge.getOptions().setFlyToSpeed(window.DefaultEarthFlySpeed);
            var lookAt = createEarthLookAt(ge, latitude, longitude);
            ge.getView().setAbstractView(lookAt);
          }
        });
      } else {
        var zoomLevel = window.DefaultMapZoom;
        if (window.LazyMapType == G_SATELLITE_3D_MAP) {
          zoomLevel = window.DefaultEarthZoom;
        }
      }
    }
  }
</script>
</head>
</html>
```



```
        window.GoogleMap.setCenter(new GLatLng(latitude, longitude),
zoomLevel);
        if (window.LazyMapType) {
            _setMapType(window.LazyMapType);
            window.LazyMapType = null;
        }
    }
    // map is initially hidden; show map after viewport modification
    document.getElementById('container').style.visibility = "visible";
};
</script>
</head>
<body>
    <div id='container'></div>
</body>
</html>
```



### 4.3.18 maps.html – Kartentyp definieren

Zur Anzeige der Karte muss zuvor der Kartentyp bekannt sein. Innerhalb der Beispiel-Implementierung werden dem Objekt vom Typ GMap2 im Konstruktor die gewünschten Kartentypen übergeben (siehe Kapitel 4.3.16 Seite 109).

Die Definition eines Kartentyps wird in der Beispiel-Implementierung, nach der Initialisierung der Karte, über die JavaScript-Funktion `function _setMapType(type)` gewährleistet. Innerhalb der Funktion wird die Google-Maps-Methode `setMapType(type:GMapType)` auf der Instanz vom Typ GMap2 aufgerufen. Der Methode wird der gewünschte Kartentyp (GMapType<sup>12</sup>) übergeben. Der GMapType muss der Karte bekannt sein, also entweder direkt im Konstruktor angegeben (siehe Beispiel) oder zuvor über die Methode `addMapType(type:GMapType)` hinzugefügt worden sein.

Aufgerufen wird die Funktion erst nach dem Initialisieren der Karte (vgl. Kapitel 4.3.17) über die Methode `setCenter()`, innerhalb der JavaScript-Funktionen `function(latitude, longitude)` und `function(minLatitude, minLongitude, maxLatitude, maxLongitude)`, die die Kartenansicht auf eine bestimmte Koordinate bzw. einen bestimmten Auswahlbereich zentrieren.

```
1. <html>
2.   <head>
3.     <script ...>
4.     ...
5.     window.GoogleMap = null; // GMap2 instance
6.
7.     // earth plugin apply current location on first matype switch, no
       // matter what.
8.     // as workaround we change matype in that case after viewport
       // modification not before.
9.     // Matype to set after viewport modification
10.    window.LazyMapType = null;
11.
12.    function init() {
13.      window.GoogleMap = new GMap2(document.getElementById('container')),
        {
```

<sup>12</sup> <http://code.google.com/intl/de/apis/maps/documentation/javascript/v2/reference.html#GMapType>



```
14.         mapTypes: [G_NORMAL_MAP, G_SATELLITE_MAP, G_HYBRID_MAP,
15.                     G_SATELLITE_3D_MAP]
16.     });
17.     ...
18. }
19. /**
20.  * Set map type to GMap2 instance.
21.  *
22.  * @param type new maptype
23.  */
24. function _setMapType(type) {
25.     window.GoogleMap.setMapType(type);
26.     ...
27. }
28.
29. /**
30.  * Set viewport to specified latitude/longitude position.
31.  *
32.  * @param latitude
33.  * @param longitude
34.  */
35. window.setViewPoint = function(latitude, longitude) {
36.     ... window.GoogleMap.setCenter(new GLatLng(latitude, longitude),
37.         zoomLevel);
38.
39.     if (window.LazyMapType) {
40.         _setMapType(window.LazyMapType);
41.         window.LazyMapType = null;
42.     }
43.
44.     // map is initially hidden; show map after viewport modification
45.     document.getElementById('container').style.visibility = "visible";
46. };
47.
48. window.setViewBounds = function(minLatitude, minLongitude,
49.     maxLatitude, maxLongitude) {
50.     ...
51.     if (window.LazyMapType) {
52.         _setMapType(window.LazyMapType);
53.         window.LazyMapType = null;
54.     }
55.
56.     // map is initially hidden; show map after viewport modification
57.     document.getElementById('container').style.visibility = "visible";
```



```
52.     };
53.     ...
54.         </script>
55.     </head>
56.     <body>
57.         <div id='container'></div>
58.     </body>
59. </html>
```

### 4.3.19 maps.html - Kartenobjekt über Ereignisse laden

Die Initialisierung der Karte (bzw. des DIV-Containers in dem die Karte angezeigt wird) erfolgt ereignisgesteuert über onLoad- bzw. load-Events. Damit wird sichergestellt, dass alle erforderlichen Dateien, die beim Ausführen der JavaScript-Funktionen benötigt werden, bereits geladen sind. Dazu wird das Window-Objekt um den EventHandler "onLoad" bzw. um "load" erweitert (Ereignis ist abhängig vom verwendeten integrierten Browser) (weitere Informationen siehe Kapitel Exkurs: HTML und JavaScript Seite 105 f.):

```
1.     if (window.attachEvent) {
2.         window.attachEvent("onload", init);
3.     } else if (window.addEventListener) {
4.         window.addEventListener("load", init, false);
5.     }
```

Tritt einer der Events ein, wird die JavaScript-Funktion `init()` aufgerufen, die ein neues Kartenobjekt erstellt (siehe Kapitel 4.3.16 Seite 109). Diese Kartendarstellung wird anschließend initialisiert und mit Attributen versehen. Damit die Karte innerhalb des DIV-Containers angezeigt wird, muss das DIV-Element (und damit auch die Karte) abschließend eingeblendet werden. Dies wird durch den Aufruf von `document.getElementById('container').style.visibility = "visible"` erreicht. Aufgerufen wird diese Methode erst nach dem Initialisieren der Karte (vgl. Kapitel 4.3.17) über die Methode `setCenter()`, innerhalb der JavaScript-Funktionen `function(latitude, longitude)` und `function(minLatitude, minLongitude, maxLatitude, maxLongitude)`, die die Kartenansicht auf eine bestimmte Koordinate bzw. einen bestimmten Auswahlbereich zentrieren.



### 4.3.20 maps.html – Adressdaten konvertieren (Geocoding)

Solche geographischen Koordinaten bestehen aus zwei dezimalen Werten, die eine geographischer Längen- und Breitengrad beschreiben. Von Geokodierung spricht man immer dann, wenn eine Adresse (als Objekt vom Typ String) in eine geographische Koordinate konvertiert wird. Google stellt einen Dienst zur Geokodierung bereit, auf den über das Objekt GClientGeocoder() zugegriffen werden kann<sup>13</sup>.

In der Beispiel-Implementierung wird ein neues Objekt vom Typ GClientGeocoder() innerhalb der init-Funktion erzeugt:

```
1. <html>
2. <head>
3. <script type="text/javascript"
   src="http://maps.google.com/maps?file=api&v=2&key="></script>
4. <script type="text/javascript">
5.
6. window.GoogleGeocoder = null; // Geocoder, address <> lat/lng
7. ...
8. function init() {
9.   ...
10.  window.GoogleGeocoder = new GClientGeocoder();
11. }
```

Diese Funktion ermittelt zunächst aus dem übergebenen Adress-Parameter (ein Objekt vom Typ String), eine geographische Koordinate. Für diese Geokodierung stellt Google einen Dienst (GoogleGeocoder) bereit (siehe Kapitel 4.3.20 Seite 116). Dabei wird auch der Ort des Internetzugangspunktes, von dem die Anfrage gestellt wird, einbezogen. Wird also nur ein Straßename für die Adress-Suche angegeben, wird ein standortnahes Ziel gesucht. Nach erfolgreicher Google-Geokodierung erfolgt ein Callback in die JavaScript-Funktion. Dort wird der Kartendarstellung einer neue Markierung hinzugefügt (über `window.addOverlay(...)`) und ein neuer Mittelpunkt gesetzt (siehe Kapitel 4.3.17 Seite 110).

---

<sup>13</sup> [http://code.google.com/intl/de-DE/apis/maps/documentation/javascript/v2/services.html#Geocoding\\_Object](http://code.google.com/intl/de-DE/apis/maps/documentation/javascript/v2/services.html#Geocoding_Object)



#### 4.3.21 maps.html – GeolocationUpdater (Injection Java / JavaScript)

Für die Kommunikation zwischen der nativen Ebene des integrierten Browsers (bzw. der Google Maps-API) und der Java-Ebene des FirstSpirit-JavaClients muss ein Java-Objekt (JavaScript-Proxy –GeolocationUpdater) in der JavaScript-Umgebung bereitgestellt werden (siehe Kapitel 4.3.5 Seite 65). Dieses Objekt dient dazu, eine unidirektionale Kommunikation von der JavaScript-Umgebung des integrierten Browsers in die Java-Umgebung des FirstSpirit JavaClients bereitzustellen. Das gewünschte Java-Objekt muss von außen (aus der Java-Umgebung) in das HTML-Dokument injiziert werden (siehe Kapitel 4.3.5 Seite 65). Die FirstSpirit-AppCenter-API bietet die dazu notwendigen Schnittstellen und Methoden an (Interface: `BrowserApplication` siehe Kapitel 3.7 Seite 35).

Die Beispiel-Implementierung erzeugt zuerst ein Java-Objekt vom Typ `GeolocationUpdater` in der Java-Klasse `MapsPlugin` (siehe Kapitel 4.3.5 Seite 65). Die Klasse `GeolocationUpdater` verfügt über die Methoden `void update(String uuid, double latitude, double longitude)`, die einen `GeolocationEntry` auf die übergebene Koordinate aktualisiert und den zur Eingabekomponente gehörigen `ModificationListener` über die Aktualisierung informiert und die Methode `void info(String uuid, String address)`, die die Adressinformation des `GeolocationEntry`s aktualisiert.

```
@SuppressWarnings({"UnusedDeclaration"})
public static class GeolocationUpdater {

    private final MapsPlugin _mapsPlugin;

    public GeolocationUpdater(final MapsPlugin mapsPlugin) {
        _mapsPlugin = mapsPlugin;
    }

    public void update(final String uuid, final double latitude,
final double longitude) {
        _mapsPlugin.update(uuid, latitude, longitude);
    }

    public void info(final String uuid, final String address) {
```



```
        _mapsPlugin.info(uuid, address);
    }
}
```

Eine neue Instanz vom Typ `GeolocationUpdater` wird anschließend über den Aufruf der Methode `void inject(Object object, String name)` in das HTML-Document (`maps.html`) injiziert. Dabei wird ein Name (hier: "GeolocationUpdater") übergeben, über den dieses Objekt innerhalb der JavaScript-Umgebung erreichbar ist.

```
application.inject(new GeolocationUpdater(this), "GeolocationUpdater");
```

Nach der Injektion stehen in der JavaScript-Umgebung nicht nur die Objekt-Instanz selber, sondern auch die zugehörigen Methoden des Objekts `GeolocationUpdater` zur Verfügung. Das bedeutet, alle Methoden der Java-Klasse `GeolocationUpdater` können nach der Injizierung auch innerhalb der JavaScript-Umgebung aufgerufen werden. Konkret stehen also die Methoden `update` und `info`, die innerhalb von `MapsPlugin` implementiert werden, auch auf dem Proxy innerhalb der JavaScript-Umgebung zur Verfügung:

```
1. <html>
2.   <head>
3.     <title>Google Maps</title>
4.     <script type="text/javascript"
5.       src="http://maps.google.com/maps?file=api&v=2&key="></script>
6.     <script type="text/javascript">
7.       ...
8.       window.GeolocationUpdater = null;
9.       /**
10.        * Notify MapsPlugin about latitude/longitude modification.
11.        *
12.        * @param uuid UUID of geolocation instance
13.        * @param latitude
14.        * @param longitude
15.        */
16.       function updateInfo(uuid, latitude, longitude) {
17.         var request = new GetLocationsRequest(new
18.           GLatLng(latitude, longitude));
19.         request.execute(function(response) {
20.           window.GeolocationUpdater.info(uuid,
21.             response.Placemark[0].address);
22.         });
23.       }
24.       ...
```



```
22.      /**
23.         * Use Geocoder to find the exact geolocation of the
24.         * specified address pattern
25.         * @param uuid UUID of geolocation instance
26.         * @param pattern address string
27.         */
28.      window.findAddress = function(uuid, pattern) {
29.          window.GoogleGeocoder.getLatLng(pattern,
30.              function(point) {
31.                  if (point) {
32.                      var latitude = point.lat();
33.                      var longitude = point.lng();
34.                      window.addOverlay(uuid, latitude,
35.                          longitude);
36.                      window.setViewPoint(latitude, longitude);
37.                      window.GeolocationUpdater.update(uuid,
38.                          latitude, longitude);
39.                      updateInfo(uuid, latitude, longitude);
40.                  }
41.              });
42.      };
43.  </script>
```

Listing 33: Geolocation – GeolocationUpdater (maps.html)

Dazu wird vom FirstSpirit-Framework für jede Methode der Java-Objekt-Instanzen eine entsprechende JavaScript-Methode erzeugt. Diese Methode sendet beim Aufruf aus der JavaScript-Umgebung ein Event, welches wiederum auf der Java-Seite ausgewertet wird. Das bedeutet also der Aufruf von:

```
window.GeolocationUpdater.update(uuid, latitude, longitude);
```

in einer JavaScript-Funktion, führt direkt zu einer Ausführung der update()-Methode innerhalb der Java-MapsPlugin-Implementierung. Die passende Java-Methode wird anhand des Namens und der übergebenen Parameter ermittelt und entsprechend aufgerufen:

```
1.      /**
2.         * Updates the geolocation of the specified geolocation entry and may
3.         * notify the related modification listener.
4.         *
5.         * @param uuid UUID of geolocation instance
6.         * @param latitude decimal latitude value
7.         * @param longitude decimal longitude value
8.         */
```



```
9.     public void update(final String uuid, final double latitude, final
double longitude) {
10.         final GeolocationEntry location = get(uuid);
11.         if (location != null) {
12.             location.setLatitude(latitude);
13.             location.setLongitude(longitude);
14.             location.notifyPointModification();
15.         }
16.     }
```

Die innerhalb der JavaScript-Umgebung übergebenen Parameter werden analog zur bisherigen Objekt-Konvertierung in die Java-Umgebung gehoben. Da JavaScript im Gegensatz zu Java nur eine eingeschränkte Menge an einfachen Datentypen unterstützt, gelten bei der Konvertierung gewisse Restriktionen:

Eine kleine Einschränkung gibt es hier allerdings noch. Da bei der Event-Erzeugung/Auswertung keine Synchronität gewährleistet werden kann, werden Rückgabewert der Java-Methoden per Callback zurückgegeben. Gibt es also eine Methode `String getName()` so wird in JavaScript daraus eine Methode `void getName(function:callback)`. Hier ist also immer der letzte Parameter der entsprechende Callback.

Über die `inject`-Methode kann grundsätzlich jedes beliebige Java-Objekt in eine JavaScript-Umgebung injiziert werden. Für die Methoden dieses Objektes gelten aber gewissen Restriktionen. So werden beispielsweise nur eine Reihe von simplen Datentypen unterstützt. Komplexe, in JavaScript unbekannte Objekt-Typen werden nicht unterstützt.



## 5 Listings

LISTING 1: GEOLOCATION - NEUE INSTANZ VON MAPSPUGIN ERZEUGEN (SWINGGADGET-IMPL.) ..56	56
LISTING 2: GEOLOCATION – KONSTRUKTOR MAPSPUGIN (MAPSPUGIN-IMPL.) .....57	57
LISTING 3: GEOLOCATION –BROWSERAPPLICATION-INSTANZ VORHANDEN? (MAPSPUGIN-IMPL.) ..57	57
LISTING 4: GEOLOCATION – ÖFFNEN DER APPLICATION INNERHALB EINES TABS (MAPSPUGIN-IMPL.) .....60	60
LISTING 5: GEOLOCATION – SKRIPT FÜR DIE GEOKODIERUNG EINES ADRESS-STRINGS ERSTELLEN (MAPSPUGIN-IMPL.).....63	63
LISTING 6: GEOLOCATION – JAVASCRIPT-FUNKTION FINDADDRESS (MAPS.HTML).....64	64
LISTING 7: GEOLOCATION – AUSFÜHREN DER GESAMMELTEN JAVASCRIPT-FRAGMENTE (MAPSPUGIN-IMPL.).....65	65
LISTING 8: GEOLOCATION – INJEKTION JAVA-OBJEKT GEOLOCATIONUPDATER (MAPSPUGIN-IMPL.) .....66	66
LISTING 9: GEOLOCATION – GEOLOCATIONUPDATER (MAPSPUGIN-IMPL.) .....67	67
LISTING 10: GEOLOCATION – AUFRUFEN EINER JAVA-METHODE AUS DEM JAVASCRIPT-CODE (MAPS.HTML-IMPL.) .....68	68
LISTING 11: GEOLOCATION – HINZUFÜGEN EINES NEUEN GMARKER-OBJEKTS (MAPS.HTML).....69	69
LISTING 12: GEOLOCATION – KONSTRUKTOR GEOLOCATIONENTRY (MAPSPUGIN-IMPL.).....70	70
LISTING 13: GEOLOCATION – ÜBERGABE DES GADGETIDENTIFIERS AN MAPSPUGIN (SWINGGADGET-IMPL.).....71	71
LISTING 14: GEOLOCATION – ANLEGEN EINES NEUEN GEOLOCATIONENTRYS (MAPSPUGIN-IMPL.)73	73
LISTING 15: GEOLOCATION – INFORMATIONEN AUS GEOLOCATIONENTRY-OBJEKT ÜBERMITTELN (MAPSPUGIN-IMPL.).....74	74
LISTING 16: GEOLOCATION – GMARKER-OBJEKT ZUR KARTENDARSTELLUNG HINZUFÜGEN (MAPS.HTML) .....75	75
LISTING 17: GEOLOCATION – ZUORDNUNG DES GEOLOCATIONENTRIES ZURÜCK IN DIE JAVA-U. (MAPSPUGIN-IMPL.).....77	77
LISTING 18: GEOLOCATION – REGISTRIERUNG UND VERWENDUNG DES TABLISTENERS (MAPSPUGIN-IMPL.).....79	79
LISTING 19: GEOLOCATION – REGISTRIERUNG UND VERWENDUNG EINES BROWSERLISTENERS (MAPSPUGIN-IMPL.).....81	81
LISTING 20: GEOLOCATION – INTERFACE MODIFICATIONLISTENER (KEIN BESTANDTEIL DER APPCENTER-API) .....82	82
LISTING 21: GEOLOCATION – AKTUALISIERUNG ÜBER DAS OBJEKT GEOLOCATIONUPDATER INITIIEREN (MAPS.HTML).....85	85
LISTING 22: GEOLOCATION – AKTUALISIERUNG DES LATITUDE-, LONGITUDE-WERTES (MAPSPUGIN- IMPL.).....87	87
LISTING 23: GEOLOCATION – VERWENDUNG DES INTERFACES MODIFICATIONLISTENER (SWINGGADGET-IMPL.).....88	88
LISTING 24: GEOLOCATION – AKTUALISIERUNG DER ADRESS-INFORMATIONEN (MAPSPUGIN-IMPL.) .....90	90
LISTING 25: GEOLOCATION – VERWENDUNG DES EVENTLISTENERS HIERARCHYLISTENER (SWINGGADGET-IMPL.).....93	93



LISTING 26: GEOLOCATION –UPDATEBROWSER (MAPSPUGIN-IMPL.) .....97

LISTING 27: GEOLOCATION –SUCHE MITTELS EINES ADRESS-STRINGS (SWINGGADGET-IMPL.).....99

LISTING 28: GEOLOCATION –SUCHE MITTELS EINES ADRESS-STRINGS (MAPSPUGIN-IMPL.) ..... 100

LISTING 29: GEOLOCATION –INNERE METHODE: GEOLOCATIONENTRY HOLEN (MAPSPUGIN-IMPL.)  
..... 101

LISTING 30: GEOLOCATION –MAPSPUGIN AKTUALISIEREN (SWINGGADGET-IMPL.)..... 101

LISTING 31: GEOLOCATION –... (MAPSPUGIN-IMPL.)..... 102

LISTING 32: GEOLOCATION –UPDATEBROWSER (SWINGGADGET-IMPL.) ..... 103

LISTING 33: GEOLOCATION – GEOLOCATIONUPDATER (MAPS.HTML)..... 119

