CHEMNITZ UNIVERSITY OF TECHNOLOGY

Department of Computer Science

Distributed and Self-organizing Systems Group

Master´s Thesis

Supporting Semantic Interoperability in Inter-Widget-Communication-enabled User Interface Mashups

Sebastian Heil

Chemnitz, 27 August 2012

Examiner: Prof. Dr.-Ing Martin Gaedke
Supervisor: Dipl.-Inf. Olexiy Chudnovskyy

Sebastian Heil
Supporting Semantic Interoperability in Inter-Widget-Communication-enabled User Interface Mashups
Master´s Thesis, Department of Computer Science
Chemnitz University of Technology, 27 August 2012

Abstract

User Interface Mashups are a promising approach for the development of Web applications based on re-usable components, called widgets, focusing End-User Development. One of the major challenges of UI mashups is to enable communication between widgets, so called *Inter-Widget Communication (IWC)* in order to enable data flows and state synchronization. An efficient IWC solution requires an approach which enables independent development of widgets and at the same time increases compatibility between widgets in order to allow communication regardless of data representations.

This thesis elaborates an IWC approach which, on the one hand, provides means of increasing semantic interoperability between independently developed widgets and, on the other hand features a simple and easy-to-use concept for widget developers and avoids mitigating usability for end-users. In addition to the theoretical IWC approach, this thesis presents the implementation of a client-side framework which supports the proposed approach and can be easily employed in several mashup set-ups by script inclusion.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
DOM	Document Object Model
FOAF	Friend of a Friend
GUI	Graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IWC	Inter-Widget Communication
JSON	JavaScript Object Notation
OSGi	Open Services Gateway initiative
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFa	RDF in attributes
RDFS	RDF Schema
REST	Representational State Transfer
SMCDL	Semantic Mashup Component Description Language
SOAP	Simple Object Access Protocol
SOP	Same Origin Policy
TSA	Thin Server Architecture

UI	User interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformation
XPATH	XML Path Language

1 Introduction

Within the last few years the development of Web applications in the form of *user interface mashups* (UI mashups) has become increasingly important [1]. Their asynchronous and event-driven nature [2] and their ability to spontaneously create custom views on specific issues aggregating data and tools from different sources by composing independently developed components, called *widgets*, have spawned a wide variety of user interface mashup applications and frameworks [3]. UI mashups form a promising approach for the development of Web applications based on re-usable components, i.e. widgets, which focuses *End-User Development*[1] [4].

1.1 Motivation

One of the major challenges faced by them is to enable communication between widgets, so called *Inter-Widget Communication (IWC)*. Without IWC, each widget in a mashup works in isolation. There may be other widgets which perform similar tasks, provide services the widget could use or which could extend its functionality by cooperation, however, without IWC the widget cannot know of nor interact with them. This isolation can induce redundancy as widgets are not enabled to co-operate and share common functionality and information thus causing them to become more extensive and less maintainable.

What is more, IWC, or the lack of it, has a strong impact on usability [3], [5]. Imagine, for instance, a mashup application concerned with travel which could feature a social travel advisory widget listing hotels highly recommended by the user's friends, a map widget for displaying possible destinations, a translator widget to translate foreign-language hotel descriptions, a calendar widget displaying the personal calendars of the user and a booking widget to book accommodation and flights. The advantage a potential user gains by using this mashup is the availability of several services relevant

[1] cf. http://www.interaction-design.org/encyclopedia/end-user_development.html

for the traveling domain interfaced by and accessible through the widgets in one single, aggregated view.

However, without IWC, the usage of this mashup requires several manual actions: The customer browses the list of recommended hotels and checks their locations by manually copying the address from the hotel description into the location prompt of the map widget. Once he has found a hotel located in a quiet area at some interesting destination, he reviews the hotel description in the advisory widget. Unfortunately, this description is written in Swahili so he has to copy and paste the text into the translator widget. After checking the availability of rooms and flights in the booking widget, the customer wants to cross-check those dates against his calendars. In order to do so, he has to manually leaf through the calendar widget to get to the corresponding dates. Ultimately, he books his flights and the hotel and manually adds his planned vacations to his calendar by creating an event in the calendar widget.

As illustrated by this example, without interactivity between widgets, users have to perform multiple manual actions in order to synchronise widgets and simulate communication mitigating the usability of the mashup application. (In 3.1 the leveraging of this scenario in an IWC-enabled environment is presented.)

Recent UI mashup projects like Geppeto[2] or Scrapplet[3] are aware of the necessity of IWC and have developed a wide range of different approaches [6]. In 2011, Zuzak et al. conducted a thorough analysis of cross-context communication methods covering more than 30 systems many of which can and have been used for Inter-Widget Communication [7].

In the field of widget specifications, there are two major competitors as of today, in 2012: OpenSocial Gadgets and W3C Widgets. These specifications detail packaging, configuration, metadata, communication and security aspects. While specified for OpenSocial Gadgets [8], the W3C Widgets specifications lack a description of IWC [9–11]. With an increasing number of widget engines which are supporting W3C Widgets coming into existence, the lack of a normative description of Inter-Widget Communication has encouraged the occurrence of a wide range of different approaches (cf. 2.2.1).

[2] http://www.geppeto.fer.hr/
[3] http://www.scrapplet.com/

1.2 Problem

Though many of these systems may work well in *Orchestrated UI mashups* [1], i.e. pre-composed mashups, where experts, called *mashup developers* [1] or mashup creators [2], build mashup applications by composing widgets with knowledge of the various data formats used and ensuring interoperability by either thoroughly selecting widgets or manually defining the required transformations a priori [2], there are not many solutions supporting interoperability in non-pre-composed, i.e. *Choreographed UI mashups* [1]. Moreover, there is a surprising lack of usage of IWC in current widget implementations which recent approaches are trying to tackle [4] [12].

However, the necessity of Inter-Widget Communication in these mashups, where end-users compose custom user interfaces by adding widgets to the UI mashup at runtime, is evident [1], especially if the resulting solutions shall overcome being mere portal sites without any interactivity between their components [13]. The travel booking scenario previously described could refer to a choreographed UI mashup as well. For instance, the user may change the default widget combination provided by the travel mashup application in that he substitutes some of the existing widgets with his favourite translator, map and calendar widgets.

In this thesis the lack of IWC usage is partially attributed to the lack of suitable IWC frameworks with sufficient interoperability support for ad-hoc built, choreographed UI mashups. In order to enable Inter-Widget Communication between arbitrary widgets employing different data formats, an IWC framework has to provide means of communication not explicitly defined, i.e. rather emerging from the current participants, while, by strictly separating semantic concepts from syntactical concerns, supporting the required transformation of data formats.

This raises the question as to what a theoretically ideal solution for Inter-Widget Communication in ad-hoc built, choreographed user interface mashups should be like, which requirements it should fulfil. There are several issues to solve, including the selection of the most appropriate communication model, the support of data format transformation and granting a high degree of independence in widget development.

Recent implementations of Inter-Widget Communication as for instance the OpenAjax[4] Hub rely exclusively on compatibility of the data formats used in their messages [7] which is considered an obstacle for the independent development of widgets.

Obviously, transforming different data formats is not an entirely new issue. In the Web Services domain, this is a well-researched sub-problem of Service Mediation [14–16] with several solutions existing [17] [18]. However, as yet, no appropriate solutions in the field of Inter-Widget Communication in choreographed user interface mashups could be found. As a consequence, there is more research required on how to combine data transformation patterns proven successful in Service Mediation with Inter-Widget Communication.

1.3 Objective

As illustrated above, an IWC approach which enables independent development of widgets and increases widget compatibility is required. Therefore, this thesis aims at elaborating an IWC approach which, on the one hand, takes into account experience of the IWC solutions currently existing in order to enable easy usage and configuration and, on the other hand, attempts to increase the number of widgets which are able to communicate with each other in a UI mashup modified by end-users at runtime by providing a basis for increasing *semantic interoperability* between widgets independently developed by different widget developers.

The term semantic interoperability in the context of UI mashups refers to the possibility of arbitrary widgets to communicate with each other, even without explicit definition by the developers, and being able to interpret the information they exchange "in accordance with the intentions of the sender." [5]

In order to increase semantic interoperability, the approach shall support establishing syntactic compatibility between semantically compatible widgets, i.e. between widgets which can communicate based on shared semantic concepts, on shared communication subjects but "talking" different "languages". Furthermore, the approach shall enable the re-use of the functionality of achieving this syntactic compatibility of IWC messages in future widgets.

[4] http://www.openajax.org/

Alongside with the elaboration of a theoretical approach, a prototype framework implementation shall be developed as a proof of concept and feasibility and its vital implementatory aspects outlined in this thesis.

1.4 Subsequent procedure and structure of this thesis

In the remainder of this thesis, an analysis of the state of the art of IWC will be conducted, a draft for a new IWC approach elaborated, implementatory aspects outlined and the presented approach evaluated.

The state of the art analysis conducted in chapter 2 "State of the Art" will systemize requirements and existing approaches, and analyse them with regard to semantic interoperability. Drawing conclusions from this analysis and taking into account procedures and concepts of different domains, in chapter 3 "Concept" a derived approach which supports communication between semantically compatible widgets will be outlined as well as problems and proposals upon their solution will be discussed. The subsequent chapters, 4 "Publish/Subscribe", 5 "Mediation" and 6 "Plugins" will explain successive development steps and crucial aspects of the prototype implementation starting with a simple communication solution and evolving it towards the desired scope presented in 3. Eventually, in chapter 7 "Evaluation", the resulting approach will be evaluated against the requirements previously presented and the findings of this thesis will be concluded in chapter 8 "Conclusion and future research".

2 State of the Art

This chapter aims at specifying requirements for an ideal solution of Inter-Widget Communication in choreographed user interface mashups. Subsequently, an analysis of several existing approaches in Inter-Widget Communication with regard to these requirements is conducted.

In order to systemize the state of the art analysis, the roles involved and their specific requirements will be identified, currently existing approaches considered and categorized and eventually compared employing the requirements previously stated.

2.1 Requirements

Considering user interface mashups, there are at least three distinct stakeholder roles involved: Mashup Platform Providers, Widget Developers and End-Users. Each of them has its own requirements. As a consequence, the requirements detailed in this section will be organized accordingly. In order to avoid confusion, these three roles are defined below. Subsequently, the requirements specific to these roles will be detailed.

2.1.1 Roles involved

The process of creating a new mashup application is initiated by the *Mashup Platform Provider*. There are several decisions to be taken by this role; functional decisions include which services to provide to which target group of users, which pricing model to select and whether to provide pre-composed mashups or to allow users to compose their own mashups within a certain set of restrictions.

Considering technical aspects, this role is responsible to decide upon which mashup engine to use or whether to develop a new one, which widgets to provide, which platforms, i.e. browsers and frameworks, to support as well as upon security, authentication and authorization aspects. The requirements of this role are described in subsection 2.1.3.

Widget Developers create the basic components, the building blocks as it were, of UI mashups, which are called widgets. They focus on providing a certain self-contained functionality using HTML, JavaScript and Web Services. There are many formal definitions of the term widget, for an in-depth definition one can refer e.g. to the W3C widget landscape specification [10].

The role of a Widget Developer may coincide with Mashup Platform Provider if it has been decided in favour of exclusively or partially providing widgets developed by the Mashup Platform Provider himself. However the recent situation shows a tendency towards allowing widgets foreignly developed to be used in mashups thus exploiting the advantages of *Crowdsourcing* [19].

Perhaps the most important role with regard to UI mashups is the role of the *End-User*. He uses the mashup application in order to achieve a certain goal based on several action steps or to satisfy a certain need for information. Depending on the decisions of the Mashup Platform Provider, End-Users use pre-defined sets of widgets or select and compose widgets according to their preferences from a pool of widgets provided or may even add their own favourite widgets from sources such as Google Gadgets directory[5] or from commercial providers like widgetbox[6]. The widgets usually can be arranged in the mashup by the End-Users. Usability is a quintessential issue for End-Users [3]. In particular, as in contrast to the more general role of a Mashup Developer [1], the term End-User emphasizes the potential lack of particular technical skills such as programming experience [20] which makes reference to the idea of "End-User Development" [21].

Although the mashup development process starts with the Mashup Platform Provider, in the following the requirements will be listed starting with End-User requirements as these induce Widget Developer and Mashup Platform Provider requirements.

2.1.2 End-User requirements

From an End-User's perspective, there are three important requirements which have to be met by UI mashup applications:

1. Communication,

[5] http://www.google.com/ig/directory?synd=open
[6] http://www.widgetbox.com/

2. Rich set of provided interactive components and
3. Simplicity.

The following paragraphs will detail and reason these three requirements.

Communication

Communication is the quintessential issue of this thesis. As it has been illustrated in the scenario described in section 1.1, End-Users require Inter-Widget Communication for improved usability as its impact on usability is substantial [3]. Without IWC, End-Users have to perform multiple manual actions in order to synchronise widgets and simulate communication mitigating the usability of the mashup application. This demand forms a functional requirement.

Furthermore, it forces Mashup Platform Providers to provide communication support and Widget Developers to employ the provided communication approaches in the widgets they create. The requirements of these two roles, as detailed in sections 2.1.3 and 2.1.4, are derived from this situation.

Rich set of provided interactive components

When using an ad-hoc built UI mashup application, End-Users choose widgets for certain tasks according to their personal preferences. If they need a map widget, for example, they would like to use the widget of their favourite provider, be it "Google Maps", "Bing Maps" or "Open Street Map", regardless of interoperability concerns. The freedom of choice of widgets of a certain domain performing the same set of tasks is crucial for End-Users. This requirement influences widget discovery in that, instead of providing a group of widgets constrained by being syntactically compatible, compliance with this criterion shall enable discovering widgets for particular tasks. The key issue is to ensure that those widgets are able to communicate nonetheless. With regard to Requirements Engineering, this can be treated as another functional requirement.

Simplicity

This criterion emphasizes the End-User's demand for usability. As already described in this sub-section, the existence of IWC positively influences usability of the entire mashup. However, the IWC solution itself may not diminish this advantage. In order

to maintain the ease of use of the mashup, an IWC solution shall not require extensive End-User action or configuration for the purpose of enabling IWC. As the mashup composition is not necessarily predefined by a mashup developer different from the End-User, End-Users may add or remove widgets at runtime. The behaviour desired by End-Users is that IWC just works "out-of-the-box". Any widget newly added to the running mashup shall be able to participate in the communication immediately, without requiring further user interaction. This requirement partially overlaps Ivan Zuzak's IWC dimension "*User Involvement*" [22] in that it advocates a very low level of user involvement. Simplicity forms a non-functional requirement.

2.1.3 Mashup Platform Provider requirements

The End-User's demand for IWC causes Mashup Platform Providers to identify suitable IWC solutions. The vital requirement from a Mashup Platform Provider's point of view is

1. Rich set of compatible components with minimum effort

Rich set of compatible components with minimum effort

Driven by the End-User's requirement "Rich set of provided interactive components", Mashup Platform Providers attempt to achieve this by defining the mashup environment for the widgets, in particular the IWC solution, accordingly, in order to enable a rich set of compatible components, i.e. widgets which are able to communicate with each other.

However, from the perspective of a Mashup Platform Provider, it is important that the IWC solution he decides to employ requires no or just very few manual action, neither for configuration nor for the functionality itself. Developing an own IWC solution would grant high grade customization however require also high development effort. Apart from that, there exists the possibility to use an existing framework.

If End-Users request that new widgets be added to the mashup's widget repository, due to requiring either a completely new type of widget or just alternative implementations to existing widgets because of personal preferences, Mashup Platform Providers have to support these widgets and ensure their ability to participate in the commu-

nication in order to satisfy their user's requirements. This can require adaption of the underlying IWC framework.

Continuous adaption could off course be provided by the IWC framework provider by means of regular updates. However, these updates would require the Mashup Platform Provider to retrieve and install them, which is an additional effort he aims at avoiding. Alternatively, a framework could be adapted to support new widgets using configuration by means of domain specific languages, XSLT [23] etc. Still the Mashup Platform Provider has to perform this configuration tasks manually, so this also would mean additional effort to him.

Rather, when using an existing IWC framework, adapting it shall be possible with little effort for the Mashup Platform Provider for his aim would then be to avoid having to adapt the framework manually. In order to achieve this, there are two possibilities: One would be to find a framework complex enough to cover any functionality needed; the other one would be to use a framework which, beyond its own functionality, provides for extensibility.

Basically, this implies that either, if the IWC framework can adapt itself to new widgets, the effort is shifted towards the framework developers or otherwise towards the Widget Developers. For the latter, it is in the Mashup Platform Provider's interest, that this effort does not cause Widget Developers to cease developing widgets which can be used on his platform. Consequently, a good solution has to find a compromise balancing both the Mashup Platform Provider's and the Widget Developer's intensions. This requirement combines the functional aspect of the similar End-User requirement "Rich set of provided interactive components" with the non-functional aspect of minimum effort on the Mashup Platform Provider's side.

2.1.4 Widget Developer requirements

Caused by the End-User's requiring IWC, Widget Developers have to create widgets capable of communicating. Their requirements can be subsumed under the following two criteria:

1. Independence of the development process
2. Lean Communication Interface

Independence of the development process

A good IWC solution shall enable Widget Developers to create their widgets without any a-priori knowledge of other widgets which may or may not be present in the parent context of their own widget at runtime. This implies, inter alia, that widgets shall not communicate by exchanging messages explicitly directed to one or several specific recipients nor by reading from and writing to any shared memory dedicated to but one widget or a specific group of widgets. Instead, widgets shall rather communicate in a way which allows for the participation of any interested party. However, this does not affect the possibility to restrict or inhibit communication for reasonable security concerns. Independence of the development process is a non-functional requirement.

The main concept behind this criterion is to obey to the principle of loose coupling which has proven successful in Software Engineering in general and Web Engineering in particular. The less dependency induced by applying knowledge of foreign components to the design of one's own component, the higher its evolvability. [24], [25]

Lean communication interface

When using IWC solutions, Widget Developers prefer to employ solutions which do not require much effort for using them. This refers to both the complexity of the communication interface and the complexity of the concepts used by the IWC solution. In order to measure the leanness of the communication interface, one has to take into account that communication always involves at least two roles: a sender and a receiver [26].

Thus any communication interface will have to define at least two functionalities – one for sending, one for receiving – regardless of the concrete communication paradigm. For example, socket-based communication has at least "send" for the sending and "listen" for the receiving side, in Publish/Subscribe "publish" is used to send and "subscribe" is the concept for receiving parties, shared memory approaches use "in" and "out" or rather "write" and "read" [27]. Regarding the complexity of the concepts employed, Widget Developers are inclined towards preferring simple ones such as parameter value pairs over more complex ones like additional descriptive languages, meta data etc.

As a consequence, measuring this criterion, any communication interface which offers only the minimum of two different functionalities and only basic concepts will be con-

sidered as lean. Therefore, in this thesis leanness of the communication interface is defined as inversely proportional to the amount and complexity of the concepts employed for communication and interoperability in an IWC solution. The more additional complexity is in the interface, the less lean it is considered to be. “Lean communication interface“ is another non-functional requirement.

2.2 Analysis

This section will analyse existing implementations of IWC both in IWC frameworks and as parts of entire mashup systems which can be employed in or applied to UI mashups. In order to enable future solutions to be judged accordingly, categories will be derived from the solutions currently in existence. Finally, the categories will be analysed against the requirements elaborated in section 2.1.

2.2.1 Categories

Considering existing IWC solutions, one can identify common properties leading to the creation of categories. The main criterion in the context of this thesis is the existence or lack of interoperability support. Within IWC solutions with interoperability support, this thesis further more distinguishes two different sub-categories depending on which role is involved in defining the interoperability. As this thesis also focuses on usability aspects, User-defined interoperability solutions are further sub-divided into internal and GUI-based. The resulting categories are

1. IWC Solutions without interoperability support
2. IWC Solutions with interoperability support
    a. Widget-developer-defined interoperability
    b. User-defined interoperability
        i. Internal
        ii. GUI-based

and will be described in the following. For each category, one instance will be presented in detail.

IWC Solutions without interoperability support

The majority of IWC solutions offer basic cross-context communication capabilities without any dedicated interoperability support. Normally, they contain a monolithic architecture: they are provided "as is" with a certain degree of functionality and complexity interfaced through an API. Some of them are more general ones, such as pmrpc[7], providing different messaging patterns [28], whereas others, as for instance the OpenAjax Hub, are focused on Publish/Subscribe mechanisms [8]. Another implementation belonging to this category is the IWC solution of the LTFLL project, which uses a shared memory approach [29].

Interoperability aspects are not explicitly supported by these solutions. Widget Developers are responsible for both deciding which senders to listen to or which messages to receive and understanding these messages.

Now, an instance of this category will be presented: the Open Ajax Hub. The term OpenAjax Hub describes any implementation complying with the OpenAjax Hub Specification which was published by the OpenAjax Alliance in version 2.0 in 2009 [8]. There is a reference implementation available from the OpenAjax Alliance on sourceforge.[8] Moreover, it has been used in several mashup products such as Scrapplet, JackBe Presto[9] and the TIBCO Page Bus[10]. Further IWC approaches such as the OpenSocial[11] Inter-Gadget-Eventing have been developed "based on the OpenAjax Hub" [30].

It features a Publish/Subscribe paradigm and exists in two variations: the *Unmanaged Hub* and the *Managed Hub*. The latter differing from the former in that it „includes the ability to isolate components into secure sandboxes". As stated in the specification, the Unmanaged Hub centres on "Programmer-built mashups" whereas the Managed Hub supports both "Programmer-built mashups" and "End-user built mashups". The communication is based on publishing messages consisting of a topic and data to the corresponding subscribers. In addition to "anonymous broadcast" messaging as devised by Publish/Subscribe, it also offers advanced features such as "point-to-point messaging" or "remote procedure calls". [8]

[7] https://github.com/izuzak/pmrpc/
[8] cf. http://openajaxallianc.sourceforge.net/
[9] http://www.jackbe.com/products/
[10] https://docs.tibco.com/products/tibco-pagebus-2-0/
[11] http://opensocial.org/

In chapter 8, the recommended pattern for topic names is detailed. These topic names consist of tokens separated by dots. In order to avoid "potential topic name conflicts", the specification strongly recommends namespacing topic names by having them start "with a registered internet domain" in "reverse order". Due to the ability to distinguish tokens in topic names and employ wildcard characters, partial matching and consequently hierarchical subscriptions, e.g. "com.example.iwc.*", are supported. [8]

As there is no explicit support for interoperability, such as additional parameters for declaring the format of the data payload, the topic names are employed to transport syntax information. This happens either implicitly, i.e. by associating certain topic names with certain formats – e.g. topic "com.example.iwc.location" is always employing ISO6709 format – or explicitly, i.e. by adding a format identifying string as extra token to the topic names, e.g. topic "com.example.iwc.location.GeoRDF" is using GeoRDF[12].

In addition to the mere communication frameworks described above, there are also solutions which incorporate explicit interoperability support. Depending on whether the solution supports interoperability to be defined by the Widget Developer or by the user it is distinguished between

1. Widget-developer-defined interoperability
2. User-defined interoperability

Widget-developer-defined interoperability

In this category, the interoperability is defined by Widget Developers by means of providing additional information on their widgets. Solutions belonging to this category support interoperability by processing this Widget Developer-provided metadata and conducting the communication accordingly.

There are two different degrees of interoperability support: one possibility is to define data transformations in the widget metadata using transformation languages such as XSLT, JSONT[13] etc. Another, even more elaborate possibility is to specify data type information in the widget description in order to enable the platform to derive and perform the necessary conversions automatically. An implementation of the Widget-

[12] http://www.w3.org/wiki/GeoRDF/
[13] http://goessner.net/articles/jsont/

developer-defined interoperability approach is the CRUISe project [31] which features both of these possibilities. Another implementation has been presented in the Widget-based personal learning environment (WIPLE) [32].

The CRUISe[14] project, developed by Pietschmann et al., presented a holistic approach referred to as "universal composition". Any component of a mashup including UI components is abstracted by a set of "properties", "events" and "operations". These components are described using the "Semantic Mashup Component Description Language (SMCDL)". A "generic composition model" specifying components and defining data and control flow aspects is built by a domain expert supported by visual tools and interpreted by the CRUISe runtime environment. [31]

As this approach extends "the service-oriented paradigm to the presentation layer", the communication between widgets is handled in the same way in which communication between data services or business logic components is. Communication is considered a mapping of operations, events and the corresponding parameters. In order to cope with the problem of syntactic heterogeneity, the CRUISe platform features powerful transformation abilities. Properties and parameters are typed "referencing semantic concepts […] from ontologies". The data exchanged between components is transformed into XML[15]-"groundings" predefined using XML Schema[16]. Necessary transformations can be defined for example using XSLT or derived by the semantic descriptions. [31]

In CRUISe, mediation between components is conducted on two levels. On the syntactical level, transformations are preferably executed using XML transformations inside the browser due to performance, whereas on the semantic level, the reasoning is performed on the server. The former is achieved using a wrapper which is capable of "renaming, reordering and filtering […] operations, events and their parameters with regard to the composition model" [31]. The latter process includes two steps: lifting and lowering. Lifting takes place upcasting data from its XML-grounding utilizing XSLT into RDF[17]/XML and processing it with a reasoner, lowering consists in a SPARQL[18]

[14] http://mmt.inf.tu-dresden.de/Forschung/Projekte/CRUISe/docs/poster_en.pdf
[15] http://www.w3.org/TR/xml11/
[16] http://www.w3.org/XML/Schema
[17] http://www.w3.org/standards/techs/rdf#w3c_all
[18] http://www.w3.org/TR/rdf-sparql-query/

query and transforming the result back into the corresponding grounding with XSLT. [31]

To conclude, in CRUISe, IWC is an instance of the more general communication between components. These components have to be described using SMDCL. The communication aspects such as data transformations are either automatically derived from component descriptions or explicitly stated in them.

User defined interoperability

All frameworks belonging to this category have the common feature of user defined interoperability, i.e. in contrast to the previous category the communication is not defined by Widget Developers, but by the users of the mashup. The term user instead of End-User is used to contrast the target group of potentially technically more experienced users of some instances of this category from End-Users without technical experience. This concept can be supported by the frameworks in different ways including recently researched approaches of offering *observer widgets* which are listening on events and providing the user with means to define actions [4][12] or even using the graphical metaphor of *wires*, well-known in data mashups such as Yahoo Pipes[19], automatically drawing connections between widgets while allowing users to inhibit connections between the widgets not intended to communicate visually[20]. In order to belong to this category, a framework does not necessarily need to provide means of defining transformations for the data flows, instead defining interoperability aspects like event to event or event to method mappings are sufficient.

Furthermore, the thesis distinguishes between internal and GUI-based user-defined interoperability solutions. In both sub-categories, the communication flows including or excluding parameter mappings or even transformation operations can be defined visually. However, the former allow allows the user to define flows which can trigger internal functions of the receiving widgets whereas the latter only supports invoking functionality which is accessible via the GUI of the receiving widget. EZWeb [33] and SAP Rooftop [34] belong to the category of internal user-defined interoperability solutions. GUI-based user-defined interoperability solutions include Geppeto [6].

[19] http://pipes.yahoo.com/pipes/

[20] Note: This approach is not yet published, however, an online demo is available at http://vsr-web.informatik.tu-chemnitz.de/workspace-editor/

In [33] the authors describe the combination of two mashup projects: EZWeb and FAST. Whereas FAST provides "a complex gadget development environment", EZWeb is "a reference architecture and implementation of an open Enterprise 2.0 Mashup Platform". As these projects originate in the Enterprise Mashup context, End-Users are regarded as "knowledge worker[s]" knowing "about business logic in her [sic] domain", however without "programming skills" [33]. The approach aims at supporting these knowledge workers in creating composite applications to help them with their work.

This is achieved by providing communication in the "gadget layer" called "wiring". As the term implies, wiring refers to defining "behaviour and data relationships" between existing gadgets" by "visually interconnecting their input and output parameters". The report illustrates this method using the example of a "form gadget" whose outputs can be connected to the input parameters of other gadgets [33]. By drawing lines between input and output parameters of the gadgets the wiring concept of the EZWeb API is interfaced visually to the End-Users. [33]

Gadgets declare their communication abilities in so called "Templates". There are two concepts in the "Wiring"-section of a template: Events and Slots. While Events are used to transmit data to other gadgets, Slots are used to receive notifications. Both of them feature identifying names, human readable names called "friendcode" and a type. However, type can only be text, which basically turns the data transmitted into atomic name-value pairs. [35]

To summarize, the EZWeb / FAST approach employs Templates to describe input (Slots) and output (Events) parameters of its gadgets and enables the End-User to define the mappings between them using a graphical wiring metaphor. The data is transmitted as name-value pairs.

Geppeto, abbreviation for "Gadget Parallel Programming Tool" is "a web automation framework based on consumer-programmable widgets for programming workflows of consumer applications" [6]. It employs "User-defined application logic based on GUI actions" [22] to enable consumers to build applications using widgets as "basic building blocks" [6]. IWC is considered a part of defining the workflow of an application. [6]

The workflow definition is taking place on GUI level by recording and replaying GUI actions such as clicks, selections and typing. The so called "programming-by-

demonstration method" is realized by providing three "special-purpose widgets": "TickMe", "TriggerMe" and "TouchMe". [6]

TickMe offers possibility to initiate a workflow by defining a certain start date and time. Moreover, an interval for repetition can be stated. With TriggerMe, event-initiated workflows can be defined. The events are provided in "event groups" by an "event repository". Users can choose from available events to subscribe the TriggerMe widget to, receiving the corresponding event at runtime will activate the defined workflow. The TouchMe widget can be used to define "consumer-initiated workflow". Workflow execution is triggered by user actions on the GUI like clicking a button. The GUI of the TouchMe widget is adaptable and can be used to create a visual starting point for workflows by for instance creating a form with all inputs required for the intended workflow. TouchMe workflows can also be initiated by TickMe and TriggerMe. [6]

The action to be initiated by all of the three widgets is defined switching into a recording mode and performing the action on the GUI. The TouchMe widget stores the user's interaction with the GUI into a 2D table employing a simple language to describe GUI actions. There are four action types: "when clicked, copy to, click, and select". The actions in this table can be executed sequentially or parallel depending on their positioning in the table. Forming semi-natural language phrases like

"copy Symbol at TouchMe to Symbol at Finance"

the actions are accessible to the user and can be edited, deleted and added after recording. [6]

To summarize the Geppeto approach, IWC is taking place on GUI level by defining workflows consisting of GUI actions which can be triggered by time, events or user actions. The definition of the data flows is based on copying values from textboxes of the sending widgets to the appropriate input fields of the receiving widgets and is yielded by recording user actions. These actions are transformed into a GUI scripting language which can subsequently be edited by the users.

2.2.2 Comparison

In order to support the reader in comparing the different categories presented above, this sub-section continues with an evaluation of these categories which is organized

according to the requirements introduced in 2.1. An overview of the evaluation results can be found in Table 2.1.

Communication

Analysis of the first criterion, Communication, reveals that all categories offer the ability to use communication between widgets. To contrast to a small degree, the category "No Interoperability Support" is evaluated slightly higher than the remaining categories as it comprises many general cross-context communication frameworks which tend to offer a wider range of communication abilities such as advanced messaging patterns like RPC or even cross-browser communication.

Rich set of provided interactive components

In contrast to this, solutions without interoperability support perform poorly with regard to the requirement "Rich set of provided interactive components" because they violate its very definition: with these IWC solutions, only a very narrow set of syntactically compatible, and hence from the End-User's perspective potentially interactive, widgets can be offered to the user, being the reason for the rating "inadequate".

When considering solutions with interoperability support, this set is substantially larger. The best results are yielded with developer-defined interoperability solutions as the set of provided components is very large due to their ability to not only define custom conversions, but also even have the platform infer semantic relations and transformations. This high compatibility is rewarded with "excellent".

Reflecting upon user-defined interoperability solutions, the set of interactive widgets is potentially smaller. For the group of internal solutions, it depends on the flexibility of parameter mappings offered; it gets larger if these mappings may include user-defined transformations, however this would be in conflict with the idea of *End-User Development* [21] and usability due to introducing too much complexity for non-technical users. For this reason, internal solutions are only rated with "sufficient".

The same disadvantages apply to GUI-based user-defined interoperability solutions with the additional constraint of limiting interoperability to functionality which is available via the graphical user interface. This restricts the set of widgets being able to communicate with each other even further thus yielding the rating "poor" for this category.

Simplicity

Evaluating the End-User requirement of simplicity, any solution which is not user-defined excels. The rationale is that from an End-User's perspective, in these solutions IWC simply works "out-of-the-box" without requiring him to take any actions. As a consequence, both IWC solutions without interoperability support and IWC solutions with widget-developer-defined interoperability support are rewarded with "excellent" in this criterion.

In contrast to this instant communication, IWC solutions with user-defined interoperability support require the user to take actions for the communication to work. One has to distinguish between internal and GUI-based solutions.

The former has two possible user involvements: one would be to simply connect ("wire") widgets – an action which still can be considered "good" to "sufficiently" simple – the other possibility is to additionally define parameter mappings. Understanding this concept and interpreting the potentially cryptic parameter names is considered "sufficient" to "poor" which leads to the overall rating of "sufficient" for internal solutions with regard to simplicity for End-Users.

The latter, GUI-based solutions, employ either additional communication widgets, extensions of the user interface of existing widgets, GUI scripting languages or combinations of these methods. GUI scripting languages are evaluated as an "inadequate" method with regard to simplicity as they require the user to learn their syntax and are not suitable for End-Users without technical experience.

Introducing additional communication widgets still means having to learn a new concept and using a widget, i.e. an intended "building block" of the mashup application, for communication purposes instead which may be unintuitive for some users. Moreover, defining communication by means of additional widgets requires a good amount of extra interaction and switching between widgets, i.e. mind contexts. Therefore, this idea is rated with "poor".

Extensions of the existing user interface integrate more smoothly into the mashup usage the user is accustomed to, still they require significantly more and more complex actions in comparison to the simple wiring method employed in the internal category. Off course, one could combine these two approaches, i.e. use graphical wires to define GUI-level interaction, this would, however, quickly lead to confusing graphical interfaces full of wires across – and thus hiding or being hidden by – the widgets. With this

in mind, the extensions method is rated with “sufficient”. The summarized simplicity rating for GUI-based solutions is the average of the three ratings of its partial concepts: “poor”.

Rich set of compatible components with minimum effort

This criterion hast two components: its first part is similar to the End-User requirement “Rich set of provided interactive components” because the Mashup Platform Provider wants to offer this to his users. However, in contrast to the End-Users, it is of importance to the Mashup Platform Provider to achieve this aim with as little effort as possible. Considering these two components and giving the reduction of effort a slightly higher influence, one can deduct the ratings of the different approaches for this requirement from the ratings assigned for “Rich set of provided interactive components” as follows.

All approaches do not require any specific configuration or updating by the Mashup Platform Providers. Thus the effort component will be evaluated as “excellent”. Combining this rating weighted slightly higher with the ratings of the corresponding End-User requirement, which reflects the amount of widgets provided, solutions without interoperability support are rated “sufficient”, developer-defined interoperability solutions “excellent” and user-defined solutions “good”.

Independence of the development process

When evaluating the Widget Developer requirement “independence of the development process”, one has to analyse the degree of independence which can be achieved when developing a widget using a certain framework of one of the categories presented above while attempting to maximize the interoperability regardless of whether or how much this is supported by the solution selected.

Thus, using solutions without interoperability support, this is rated as “inadequate” as maximizing interoperability without the support of the IWC solutions means integrating interoperability support into the widget. This implies severe dependency on a-priori knowledge about the widgets which are to be used along with the one the Widget Developer is currently creating. In particular, attempting to be able to understand any messages received from other widgets, Widget Developers need to know about the format which these messages will employ and have to implement conversions inside their widgets accordingly. Changes in the set of widgets used with one’s own widget

or in the formats they use will necessitate changes to the widget, indicating a high level of dependency.

In antithesis to this very low level of independence, solutions with Widget-developer-defined interoperability support excel with regard to independence of the development process. Using this approach, Widget Developers, who best know about their own widgets, merely have to provide metadata such as data type information about their widget. If the concept of "groundings" is employed, transformations to these groundings can be stated which could be considered a light dependency, in this case not on other widgets but rather on the framework itself. However this will not diminish the rating as, processing the provided metadata, the IWC solution is capable of ensuring interoperability. No further knowledge about foreign widgets is required in developing a new widget, neither in the widget itself, nor in the additional widget description metadata. Hence this category of solutions is rated with "excellent".

Although in user-defined interoperability support solutions the Widget Developer is not involved in interoperability per definitionem, upon attempting to maximise interoperability the action of the Widget Developer is required due to the limitations of the user-defined approach:

As users can only define connections or simple transformations such as parameter mappings in the internal sub-category, the task of understanding the incoming data is partially left to the Widget Developer. If there is, for instance, a widget whose "location" output parameter is mapped to the "address" input parameter of another widget, the task of deciding on semantic similarity has been solved by the user. However, if it is intended to maximise interoperability, the Widget Developer of the receiving widget still has to develop his widget with the ability to understand the syntax of the input data. This again introduces a certain level of dependency. As this is, however an extremely strict interpretation of the independence requirement, the resulting rating for internal IWC solutions with user-defined interoperability support is still "good".

On the other hand side, GUI-based solutions are rated with "excellent". Even though basically the same idea applies to them, this rating pays tribute to the fact that the input parameters in GUI-based solutions are developed for End-User input. Thus there is less complexity in the possible inputs compared to input parameters of internal solutions. End-user inputs via the GUI confine to simple textbox entries, i.e. strings without complex formal syntax, which are mainly brief search strings, and selections expressed by either using checkboxes, radiobuttons or clicking certain buttons. The selections

themselves are atomic; if End-Users define a mapping between an input and an output selection, there is no more interoperability issue to be solved. The same applies to input strings as they are human input without complex syntax as opposed to the possibilities of internal parameter mappings. As there is virtually no possibility for the Widget Developers to increase interoperability, the development process is considered to be independent.

Lean communication interface

Depending on the specific framework, IWC solutions without interoperability support have a varying degree of complexity in their communication interfaces. Very simple Publish/Subscribe solutions feature a two component interface, namely publish and subscribe, whereas others, such as pmrpc, offer more sophisticated concepts like distinction between synchronous and asynchronous communication, remote procedure calls and error handling [28]. Thus their communication interface is more complex, less lean. To rate this category, the range of leanness of communication interface is evaluated from "excellent" to "sufficient", the overall rating being the average of "good".

The high ratings of developer-defined interoperability solutions come at the cost of a comparatively complex interface. The additional concepts required employing these solutions, such as metadata widget descriptions, groundings and transformations, are not well suited for a lean communication interface. As a consequence, this category is rated with "poor".

User-defined interoperability support solutions are divided with regards to this criterion. For the internal category, Widget Developers create their widgets and define their basic communication abilities by means of configuration files. These descriptions are then processed in order to provide the End-Users with a set of options, e.g. methods to be called, input parameters etc., which he can use for defining interoperability of the widgets in a UI mashup. The leanness of the communication interface depends on the complexity of the configuration language which is employed by the solution.

Indeed, another option would be to inspect the widgets' code at runtime using reflection methods, however, no implementation of this approach has been encountered yet. The rating for this category will thus be "sufficient" with the possibility to improve towards "good" if future implementations feature the reflection approach.

The best category considering a lean communication interface is formed by GUI-based user-defined interoperability support solutions as there is no communication interface involved on the Widget Developer's side. One of the reasons behind these frameworks is to be able to support legacy widgets which do not make use of IWC by interacting with their GUI only [12]. The Widget Developer does not need to use any specific communication framework. Consequently, GUI-based solutions are evaluated with "excellent" regarding "Lean communication interface".

	No Interopera-bility Support	**Interoperability Support**		
		Developer-defined[21]	**User-defined**	
			internal	**GUI-based**
Communication	+ +	+	+	+
Rich set of provided interactive components	— —	+ +	O	—
Simplicity	+ +	+ +	O	—
Rich set of comp. comps w/ minimum effort[22]	O	+ +	+	+
Independence of the development process	— —	+ +	+	+ +
Lean communication interface	+	—	O	+ +
+ + excellent, + good, O sufficient, — poor, — — inadequate				

Table 2.1 Evaluation

2.2.3 Conclusion

To summarize the above comparison, there is a wide range of different solutions each having their own advantages and shortcomings. The basic "communication" requirement is satisfied well by all categories. Also "rich set of compatible components with

[21] abbreviated for: Widget-developer-defined
[22] abbreviated for: Rich set of compatible components with minimum effort

minimum effort" is met by the solutions with only low variance. In spite of these good results with respect to the categories currently in existence, they maintain their significance as also upcoming solutions have to be measured against them and need to receive comparatively high results. The remaining requirements show a much more variant outcome; consequently these have to be considered individually.

Some of the solutions of the user-defined category are not specific for UI mashups but rather originate in the enterprise mashup domain focusing on a workflow-like interaction. Therefore, these solutions performed only "sufficient" to "good". It has to be noted, however, that in spite of their usability flaws when used by technically non-experienced users, they feature a "user in control" property, which is considered to be of benefit.

Solutions without interoperability are advantageous in respect to their simplicity for End-Users and Widget Developers. This benefit, however, comes at the cost of the lack of interoperability support which severely influences the vital End-User requirement "rich set of provided interactive components" as well as the "independence of the development process". As a consequence, this category cannot be considered a suitable approach overcoming the problems described in chapter 1 and received an overall rating of "sufficient" to "poor".

The analysis revealed that solutions with developer-defined interoperability support represent a potentially suitable candidate to overcome these problems. With the exception of "lean communication interface", this category performed very well in the comparison leading to an average rating of "good" to "excellent", which is the best result of the entire set of approaches evaluated. With regard to the usability concerns presented in [5] they comply well with "Just works" and "Automatic". Properties worth noting include a very low level of End-User involvement, the role of Widget Developers as domain experts for their own widgets knowing best which information to share and which to receive and the powerful interoperability support on the framework side by means of developer defined as well as automatically inferred transformations.

In spite of their power regarding interoperability support, current implementations of this category have the disadvantage of requiring the use of complex description languages. This is considered cumbersome for a high level of widget development activities. Moreover, considering the current endeavours of the W3C to standardize widget specifications, compatibility of this descriptive approach with the W3C widgets specification is questionable.

Concluding these findings, it becomes necessary to design a new solution which attempts to combine the advantages of the solutions previously evaluated while mitigating their shortcomings. In particular, this thesis intend to create an approach similar to the developer-defined interoperability category capable of a good level of interoperability support, however, being less complex and not requiring additional widget description languages.

3 Concept

This chapter will detail example usage scenarios to support identifying communication subjects commonly used in UI mashups, provide an overview of possible IWC solutions supporting semantic interoperability and discuss them in order to select an appropriate approach taking into account the knowledge gained in the state of the art analysis and finally elaborate a draft which combines the previous findings.

3.1 Example usage scenarios

This section exemplarily describes two usage scenarios which benefit from a suitable IWC solution in that the degree of user interaction required to synchronise the employed widgets is being limited. The subsequent concept will be elaborated with these examples in mind. Moreover, these usage scenarios will be used to identify potentially common IWC subjects to which will be paid tribute in 3.4.

Travel booking

In section 1.1 the necessity of IWC was briefly illustrated using a travel booking scenario which requires extensive manual synchronization by the user without any IWC solution. This sub-section will now elaborate on this scenario and demonstrate how the same situation, i.e. the same setup of widgets, can benefit from a proper IWC solution.

Assume Bob is on holiday and wishes to do a trip to some exciting location. For planning and booking his vacation he uses a specific mashup application concerned with travel, offering, among others, the following set of widgets:

1. A social travel advisory widget listing hotels highly recommended by his friends,
2. a map widget for displaying possible destinations,
3. a translator widget to translate foreign-language hotel descriptions,

4. a calendar widget displaying Bob's personal calendars and
5. a booking widget to book accommodation and flights.

In an IWC-enabled environment, Bob wants to get some inspiration for deciding which destination to travel to based on the experience of his friends, so he begins with browsing the list of recommended hotels in the social travel advisory widget. While he reviews these recommendations, the map widget will switch to display the locations of the hotels which he is viewing.

As Bob's friends love travelling in exotic regions, some of the descriptions of the hotels are written in foreign languages. However, when Bob accesses such foreing-language descriptions, the translator widget provides a translated text.

Also, the booking widget displays the hotel which Bob views in the recommendation widget so he can check the availability of rooms there. When he does so, the calendar widget switches its display to the corresponding dates so it becomes easy for Bob to check whether these dates fit his tight schedule. The same is possible for checking flight availability in the booking widget. Booking his vacation finally, the calendar widget proposes to create an appointment for the corresponding date.

Scheduling a meeting

Another scenario which can be enhanced with IWC is the scheduling of meetings. This is a task which can be supported by mashups in both leisure and business contexts. Assume Bob wants to organize a party. He plans to invite his friends as well as some of his colleagues. A mashup supporting this process, regardless whether it is pre-composed by some provider or built at runtime by Bob, could feature the following widgets:

1. A calendar widget in order to schedule a date and time,
2. two or more social networking widgets for selecting the guests to invite,
3. a map widget to find an appropriate location and provide "getting there" information,
4. an event widget to create the event and
5. a poll widget to enable the invitees to vote for the best date.

If there is an appropriate IWC framework and the widgets make use of it, Bob can first check his own calendar in the calendar widget to identify possible dates. As he has some really close friends he would like to be able to join his party, he consults his social network widgets and checks their statuses. Each time he views the profile of one of his friends, the social network widget publishes identity information. The calendar widget has subscribed to this type of information and, upon receiving it, toggles the display of the public or shared calendars of the corresponding users, so Bob can make sure his best friends are not on vacation.

Once he has found some suitable dates, Bob switches to the poll widget and creates a poll listing all the dates. He completes the poll setup and his social network widgets offer to share the link to the poll. The only manual configuration he has to perform is to restrict the group of recipients of the poll link to his planned guests.

Some days later, after the poll has expired, Bob uses the event widget to create an event at the date the poll yielded. He can use his social network widgets to choose invitees for the event and the map widget to specify an event location.

After finishing the event creation, his calendar widget can propose to create an appointment for the target date and again the event link can be shared to his friends using the social network widgets.

3.2 Possible Solutions

This section presents the first step towards the elaboration of a concept to improve semantic interoperability in IWC-enabled UI mashups. Three possible solutions were found in an initial brainstorming session at the early stage of the development process:

1. Intelligent framework
2. Thin architecture
3. Wiring + Filters

In this section, they will be presented and the preferences of the three stakeholder roles will be considered for each of them. One has to keep in mind, however, that the solutions presented here are mere abstract, theoretical solution concepts which have either not yet been implemented up to the full extent detailed in this section or even not yet

been implemented in the context of semantic interoperability, UI mashups and IWC at all. They do not refer to any particular implementations currently in existence.

The following descriptions will consider the three roles described in sub-section 2.1.1 and their intensions and requirements as detailed in sections 2.1.2, 2.1.3 and 2.1.4. In sub-section 3.2.4 these solution ideas will be discussed and the most suitable one will be selected to be elaborated upon in order to develop an appropriate draft.

3.2.1 Intelligent framework

Solutions of this category provide a monolithic IWC framework which is complex enough to provide communication and interoperability between arbitrary widgets. In order to achieve this aim, methods of artificial intelligence and logical reasoning could be employed. It would be possible to support the operation of the framework by providing additional information on the widgets. For example, existing widget specifications could be extended to contain metadata relevant for IWC and interoperability. Alternatively, Widget Developers could provide separate descriptive documents for their widgets such as RDFS[23] and OWL[24] files.

3.2.2 Thin architecture

Thin architecture solutions define a group of frameworks which feature a thin core that provides basic functionality and which additionally support extending them by adding further components. If new functionality beyond the capabilities of the framework itself is required, this functionality can be added in a modular fashion, i.e. by providing components obeying to a well-defined, thin interface and which can be re-used within the framework. With regards to interoperability, this means that Widget Developers employing some particular data formats can provide transformation modules to allow for their widgets to participate in the communication.

For Mashup Platform Providers, the thin architecture category works well because adapting the framework to new widgets does not cause additional effort for them; the same is true for End-Users. The acceptance of Widget Developers is dependent on the

[23] http://www.w3.org/TR/rdf-schema/
[24] http://www.w3.org/TR/owl2-overview/

probability to have to extend the framework, which is inversely proportional to the amount of functionality supported by the framework itself, and on the degree of complexity imposed upon them by providing additional modules.

3.2.3 Wiring and filters

Another possible approach is frameworks which provide means for the End-Users to define the communication flows between widgets. Usually, this process is supported by a graphical editor which allows users to draw connections – so called "*wires*" – to indicate communication sources and targets. With regard to interoperability, this process could be extended by introducing interposed components which perform the operations necessary for interoperability. Users could define the extended communication flows using the same graphical metaphor: The interoperability components – called "*filters*" – could be dragged and dropped either onto the wires or onto the "input sides" of the widgets.

For defining the filters' behaviour, there are several possibilities. One option would be to use a formal language. For instance, for processing text, regular expressions could be used defining matches, variables and replacements. If the data to be processed consists of numbers, simple mathematical operations such as addition and multiplication could be supported. Another option would be to use a graphical metaphor inside the filters itself. The GUI could, for example, provide a set of "building blocks" which represent certain transformations which the user can visually drag and drop into the filter to build chains in order to define the desired transformation.

3.2.4 Discourse

Comparing these three possible solutions, one can exploit the experiences gained in the state of the art analysis, in particular the considerations presented in the conclusion sub-section 2.2.3.

Intelligent framework

The category "Intelligent framework" meets the requirements of the Mashup Platform Provider quite well whereas the approval of Widget Developers highly depends on how much the complexity of the framework affects the complexity of its interface, of its

usage and thus the widget development process. For End-Users, these solutions meet all requirements and do not show any disadvantage as long as the framework complexity does not noticeably affect the runtime and thus the responsiveness of the mashup.

It bears some similarity to the complex monolithic architecture of the CRUISe platform (cf. 2.2.1) Likewise, in order to support the intelligence of a framework, i.e. its ability to interpret semantics of the communication contents and infer relations and transformations, additional description of the widgets becomes necessary.

One could argue, that a sufficiently intelligent framework would not need for instance type information but would rather be able to retrieve this information by looking at the names of functions and variables. However, this approach would require artificial intelligence methods similar to the ones employed in Natural Language Processing (NLP) as the naming conventions are not necessarily equally formalized across different Widget Developers. Moreover, these names would lack a surrounding context, such as the sentence surrounding a word in natural language, which is very likely to diminish the recognition rate below practicable levels [36].

Hence, semantic mark-up information would be needed to support the framework, which implies the necessity of Widget Developers to use some specific description language, e.g. a particular RDF-ontology, to describe their widgets and communication aspects. If one considers common usage scenarios such as the ones presented in 3.1, this approach has to be regarded excessively complex for IWC in UI mashups. It has to be feared that the required use of additional descriptive languages will prevent a significant amount of Widget Developers from using a complying IWC solution. Consequently, this approach is not regarded as being helpful to overcome the lack of IWC usage (cf. section 1.1) and will therefore not be pursued in this thesis.

Wiring and filters

Considering “Wiring and filters” from the Mashup Platform Provider’s perspective, this category of solutions complies with all requirements; the same applies to Widget Developers. For End-Users, the advantage is a high level of user exercised control over the communication itself and the possibility of defining fine-grained communication flows. However, this approach also implies extensive user involvement which may not be well-suited for the End-User requirement of “Simplicity” (cf. sub-section 2.1.2).

To be more precise, the same drawbacks identified for user-defined interoperability solutions in 2.2.3 apply to this idea. By adding the extra component of "filters" in order to incorporate extended interoperability support, this solution gets even further away from real End-User Development [21] as these filters require technical knowledge and understanding of the underlying communication logic. In particular, regular expressions have to be considered an option which cannot be intuitively understood and used by technically inexperienced End-Users. The second alternative presented, using drag-and-drop building blocks for defining transformations, could partially improve usability compared to formal languages, however, the range of transformations defined that way seems rather limited if mere graphical components without requiring further configuration by the users have to be used.

Regardless of the method employed to define the filters, the involvement of the End-User is considered as being too high like it has been for user-defined interoperability solutions in the state of the art analysis. As a consequence, the "Wiring and filters" approach will be left out of consideration in the following concept.

Thin architecture

Thin architectures are frequently seen in the category of IWC solutions without interoperability support. As noticed in sub-section 2.2.3, due to their limited complexity, their advantage is to offer a relatively lean communication interface which can be easily used by Widget Developers. Therefore, it seems a promising approach to extend these architectures towards the term of "Thin architecture" presented above.

Incorporating an extensible architecture, allows for the extension of the functionality of the framework beyond its own capabilities. This applies the idea of a plugin architecture, providing both extensibility and re-use, to IWC frameworks. These plugins can be used to conduct data transformations which would add interoperability support to the solution. Taking into account the overall good results with developer-defined interoperability support solutions summarized in sub-section 2.2.3, the most promising idea is to have Widget Developers provide the plugins as they are the persons who know best which information their widgets should communicate.

The following draft will elaborate on the details of an IWC approach with support for semantic interoperability exploiting the advantages of Thin architectures.

3.3 Architectural Draft

In order to solve the problems previously described, the following IWC framework architecture is proposed.

Being a "common model for choreographed communication" this thesis joins Wilson in advocating "a *publish-subscribe* messaging approach" [3]. The Publish/Subscribe paradigm features a threefold decoupling: decoupling in space, time and synchronization [27]. The advantage lies in avoiding explicit communication by addressing and instead enabling communication within a group of participants unaware of each other. Hence, this decision addresses the Widget Developer requirement of a "Lean Communication Interface". This is well-suited for a choreographed UI mashup approach, where End-Users add and remove widgets at runtime and the resulting communication is rather emerging from the current situation than pre-definable.

According to Zuzak et al., there are three possible IWC framework types: client-side, server-side coordinated and server-side data transfer frameworks [7]. The prototype implementation will be developed as *client-side framework*, as this type provides sufficient functionality to achieve the requirements of 2.1 while keeping communication overhead, as introduced by including a server component, low. Moreover, this can be considered as complying with the *Thin Server Architecture (TSA)* recommended by Pietschmann et al. for "user-driven composite applications" [37].

In order to accommodate *semantic interoperability* support in a Publish/Subscribe solution, it is necessary to introduce a separation between the semantics and the syntax of IWC messages. As opposed to current frameworks, where the Publish/Subscribe topics or channels are used to transport both semantics and syntax information, this thesis proposes an extension of the topic-based Publish/Subscribe interface [27] which consists in an additional "format" parameter. In this way, the topic, which will be called IWC "*subject*" in order to contrast it from topic in topic-based Publish/Subscribe and emphasize its semantic function, can be used to transport semantics, i.e. identifying the abstract concept which the communication deals with, and the format can bear syntax information needed to interpret the contents of the message. If necessary because sender and receiver of a message do not understand the same data formats, the framework attempts to transform the message accordingly.

The intended communication flow which is enabled by this extension can be illustrated by the following example:

Widget A is added to the mashup and subscribes to the subjects it is interested in stating the formats it is capable of understanding, e.g. using a tuple like (“Location”, [“GML”, “GeoRDF”]).

Later on, the user adds widget B to the mashup and begins using it. At some point, let’s for example assume he enters the name of a city, widget B notifies subscribers on “Location” about the entry of a location stating the format of the data it transmits by sending a tuple like (“Location”, “KML”, <data>) to the framework.

The IWC runtime environment checks the list of all subscriptions to “Location”. For each entry equal to (“Location”, “KML”, <widget>) it delivers the message directly to the subscribing widgets as there is no data format transformation needed. Any other entry, like (“Location”, <format>, <widget>) will be handled by looking up a transformation from “KML” to <format>, if available performing the transformation and delivering the transformed result to the subscriber otherwise dropping the message.

To support this communication pattern, the framework architecture proposed by this draft features a central hub, which handles all communication. This follows the *Gang of Four* behavioural pattern called *Mediator* [38] (which should not be confused with the Mediator term used in the Mediation domain, which will be introduced in 5).

The hub provides a hybrid Publish/Subscribe interface – hybrid due to combining topic-based and type-based Publish/Subscribe [27] – which can be seen as a special case of the *Gang of Four Observer pattern* [38]. This interface stating the subjects is extended by additional format parameters:

publish(subject,messageFormat, message)

subscribe(subject,supportedFormats[])

The idea behind those format parameters is inspired by HTTP content negotiation [39]. All messages sent calling the publish() functionality declare the format they are employing. The parameter messageFormat can be considered as an equivalent of the Content-Type header field in an HTTP response. Likewise, widgets subscribing to a certain subject state the list of formats they are able to understand, corresponding to the Accept header field in an HTTP request. In contrast to HTTP, however, IWC is no Client-Server communication but rather Peer-to-Peer-like. The choice of the best data format – a resource representation in HTTP terms – if several are possible, as well as the transformation – the creation of a certain representation in HTTP terms – if necessary, is not

handled by the participants of the communication, i.e. not by the widgets; instead, the hub performs these two tasks transparently for the widgets. This addresses the Widget Developer requirement of "Independence of the development process".

The hub uses a Mediator component to transform data from source to target format. Obviously, the Mediator cannot be able to transform arbitrary formats. At this point, i.e. reaching the limits of the functionality provided by the framework itself, the idea of a "*Thin architecture*" previously discussed in 3.2.2 is applied. Following the argumentation of Wilson et al. stating that "data transformations" are "not intuitive enough" for End-Users and instead "can […] be developed by more skilled developers and plugged in" [1], Widget Developers can provide transformation plugins and register them with a Plugin-Registry in order to extend the framework towards supporting the specific data formats which they would like to use, enabling to transform data which the Mediator is not capable of transforming. Introducing mediation into IWC addresses the End-User requirement "Rich set of provided interactive components", the thin architecture using transformation plugins takes into account the Mashup Platform Provider requirement of a "Rich set of compatible components with minimum effort".

The Mediator can query the Plugin-Registry for available transformations and mediate between the original publication and its subscribers by determining a transformation plan, i.e. an ordered list of transformations. By successively performing the transformations forwarding the output of a transformation to the subsequent one in the order given by the transformation plan, the Mediator finally yields the data in the desired target format. The idea of chaining transformations adds support for transformations where there is no direct transformation available which explicitly transforms from source to target format, but which can be achieved by a sequence of transformations either built-in or with the corresponding plugins available. One mayor issue to be solved in this context is the concrete implementation of a suitable planning algorithm.

Unfortunately, the use of several transformations introduces a certain degree of complexity into the transformation process. To address this issue, the use of a central data format, a lingua franca as it were, for each subject is advocated so transformations merely have to transform a specific data format from and to this intermediate format consequently limiting the complexity by restricting actual chain lengths to two transformations. This idea is inspired by the *groundings* observed in current interoperability approaches and mentioned in 2.2.

An implementation of this approach should incorporate *caching* both for Mediator planning results and transformation results to avoid unnecessarily redundant execution of computationally intensive code, i.e. planning and transformations, and improve performance. That is, once the Mediator has determined a sequence of transformations for a certain subject, source format and target formats combination, it should be stored as well as during the transformation phase the preliminary results of each transformation performed, i.e. a representation of the original message in a different format, should be kept. The cache persistency for the plan should be restricted by the next change in the set of transformations available as this change could outdate the validity of the planning results. Transformation results cached by the Mediator can be discarded once all subscribers have been notified about one particular publish() call for although the results may be re-used within one publication process, it is unlikely that subsequent publish() calls will contain exactly the same data.

In Figure 3.1 a non-formal overview of the proposed IWC approach previously presented can be found. In order to go into detail, Figure 3.2 shows the UML[25] class diagram representing the proposed framework architecture draft. The Hub component has a Mediator component which uses the Plugin Registry to retrieve a list of instances of the Plugin interface in order to transform a message from source to target format. The UML sequence diagram in Figure 3.3 presents the control flow of a publication from a widget to subscribers 1 to N with plugin A to Z performing the necessary transformations.

The proposed IWC approach evidently addresses the End-User requirement "Communication". Additionally, its Widget-developer-defined interoperability nature pays tribute to the "Simplicity" requirement with little to no End-User involvement. Introducing mediation into IWC addresses the End-User's demand for a "Rich set of provided interactive components". The characteristics of the Mediator itself address Widget Developer and Mashup Platform Provider requirements. In particular, its focus on transparent mediation within the framework based on the Publish/Subscribe paradigm takes the "Independence of the development process" into consideration, its API attempts to achieve the "Lean Communication interface" and the proposed plugin architecture takes into account the "Rich set of compatible components with minimum effort" requirement in avoiding extensive configuration or updates by the Mashup Platform Provider.

[25] http://www.omg.org/spec/UML/

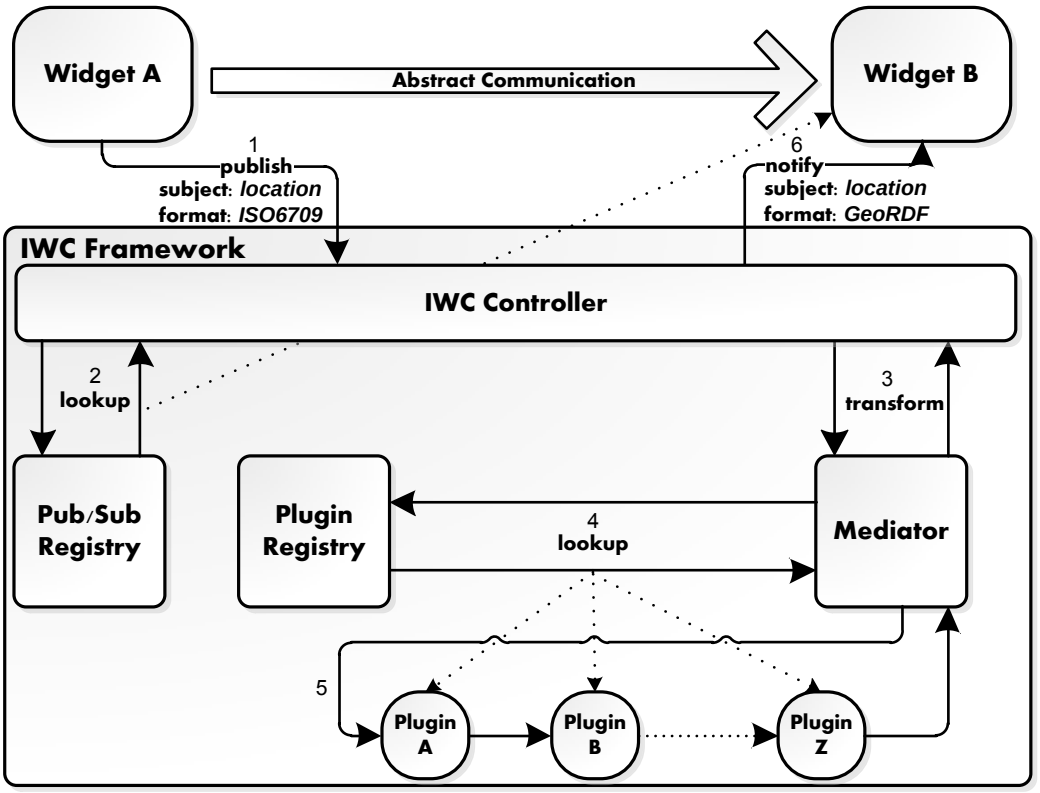

Figure 3.1 Overview of the proposed IWC approach

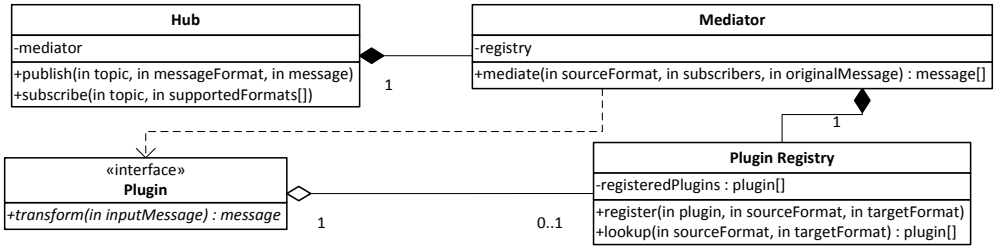

Figure 3.2 Proposed framework architecture draft

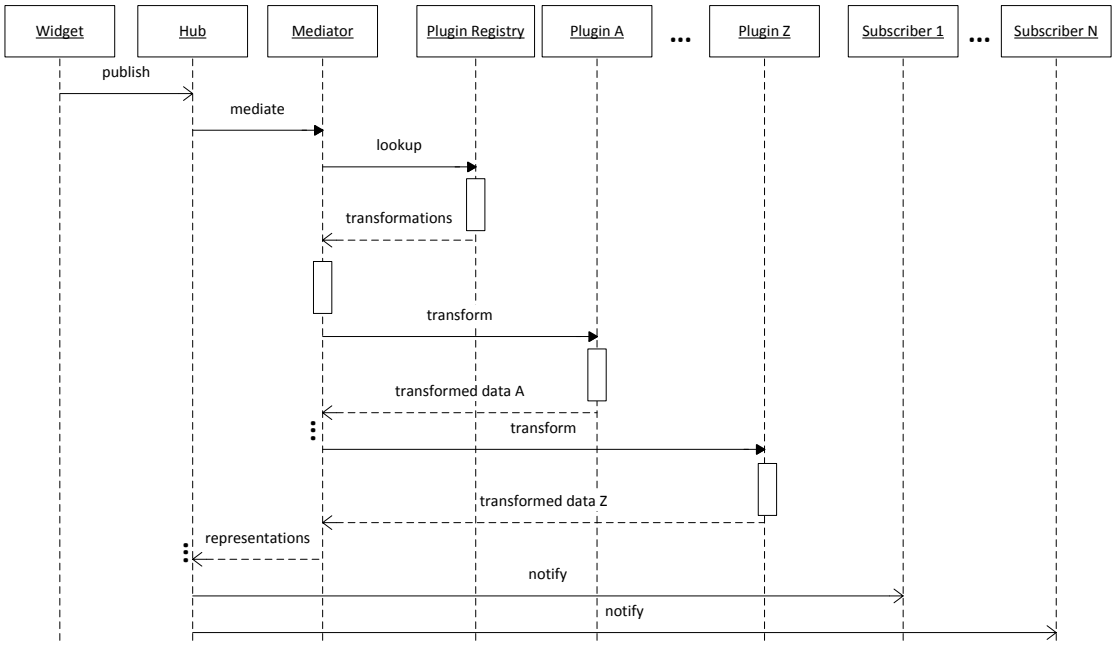

Figure 3.3 Proposed publication sequence

3.4 Widget Developer guideline

As previously described, the approach in this thesis also aims at identifying communication subjects commonly to be encountered in user interface mashups in order to cover enough scenarios to satisfy most use cases. This is inspired by the *default web intents* collated by the WebIntents task force [40]. By analysing several use cases, they identified 7 default intents, namely

1. Discover,
2. Share,
3. Edit,
4. View,
5. Pick,
6. Subscribe and
7. Save,

forming a group of services frequently required, however, independent of specific service providers.

Considering usage scenarios such as those presented in section 3.1, the following *default IWC subjects* have been identified which the proposed framework should be able to support:

1. Location (e.g. coordinates, addresses, names of landmarks)
2. Date (e.g. dates, points in time, timespans)
3. Search (e.g. user entered search terms, possibly containing Boolean operators like AND, OR, NOT or parameters like filetype, inurl etc.)
4. Message (e.g. Email, SMS, Chat)
5. Content Created (e.g. Image uploaded, Tweet posted)
6. Identity (e.g. URL[26] of a FOAF[27] profile)
7. Selection (e.g. URIs[28] of selected options)
8. Description (e.g. a descriptive text shown to End-Users in one widget to be enriched by other widgets by translation, suitable images or encyclopaedic articles etc.)

This list of default IWC subjects should be regarded as a guideline for Widget Developers which can be used to understand the IWC approach for improving semantic interoperability presented. Whenever possible, a Widget Developer should use these subjects instead of arbitrarily using self-defined subjects in order to concentrate IWC on a small set of subjects which attempts to unify communication and increase participation by partially limiting the variety of subjects. In relation to the research of choreographed UI mashups, they can be considered a part of the "*reference topic ontology*" in that "Interoperability is achieved in that each widget complies with the reference topic ontology T"[1].

[26] http://www.ietf.org/rfc/rfc1738.txt
[27] http://xmlns.com/foaf/spec/
[28] http://www.ietf.org/rfc/rfc2396.txt

Although Widget Developers are strongly advised to stick with the above list of IWC subjects, it is not necessarily complete. Therefore, additionally any subject apart from those listed as default IWC subjects above is supported in the same way WebIntents do:

The WebIntents specification enables to register and request any service by defining the "action" parameter as "[a]n opaque string indicating the action type of the intent" [41]. Moreover, the "type" parameter which is "[a] string indicating the type of the data payload" ensures the description of the syntax employed in the intent data.

Likewise, by allowing publishing data on any subject and of any type if required, communication beyond the default subjects is enabled. Widget Developers, however, are encouraged to stick to the default subjects as long as practicable. The limited number of well-defined subjects will potentially increase the number of widgets being able to communicate with each other.

When deciding to necessitate communicating on a subject which cannot be represented by the default subjects, Widget Developers shall avoid namespacing practices such as prefixing with inverse domain names as seen in recent OpenAjax Hub implementations and advocated in chapter 8 "Topic Names" of the OpenAjax Hub 2.0 specification [8]. Namespacing subjects should only be employed if a specific communication between widgets of the same provider, for instance within a company, has to be established. However, this is not the focus of this approach as it rather aims at a wide range of interoperable widgets created by different providers. (cf. section 2.1)

An important issue to address in choreographed UI mashups is, due to the emergent nature of communication, to avoid "self-reinforcing loops" [1]. These loops are caused by a group widgets, with one widget initially notifying another which reacts on the notification by notifying yet another widget itself which could, for instance notify the initial widget again closing the circuit. This cyclic receiving and emitting of notifications can cause "serious problem[s]" when it causes "real-world consequences, such as SMS-sending" [1]. Hence this widget developer guideline advocates the use of the publish method only in reaction to direct user action, e.g. a click, instead of reacting upon notifications received due to subscriptions. This behavioural guideline is intended to avoid self-reinforcing loops by prohibiting a "forwarding" of notifications. It does not negatively influence the interactivity of the widgets in the UI mashup as, due to unified IWC subjects and mediation, the initial notification should reach all concerned

widgets in a representation they are able to understand, thus forwarding it is not necessary.

3.5 Summary

This chapter has presented a concept of an IWC approach which combines topic- and type-based Publish/Subscribe with mediation in order to increase semantic interoperability in UI mashups. Two exemplary usage scenarios were outlined and used to identify IWC subjects frequently used. Several possible solutions have been presented and their advantages and disadvantages discussed. A first architectural draft has been elaborated indicating important conceptual components of the proposed framework architecture, a draft for the publication flow sequence and the framework usage from a Widget Developer's perspective. Moreover, a guideline for Widget Developers has been compiled, applying the idea of default Web Intents to IWC yielding a set of default IWC subjects which Widget Developers are advised to comply to if possible. The guideline also governs avoidance of communication loops.

4 Publish/Subscribe

In section 3.3, an extended client-side Publish/Subscribe architecture was proposed as basis of the implementation of this IWC framework. In the following section, it will be discussed how the Publish/Subscribe basis of this architecture can be implemented. This chapter is the first in a series of several implementation specific chapters and will discuss aspects of building a basic IWC prototype featuring a Publish/Subscribe architecture. Following chapters elaborate on this basic Publish/Subscribe implementation and will describe its further extension towards achieving the intended properties as presented in 3.

4.1 Situation

The Publish/Subscribe paradigm is widely employed both in IWC contexts [7] and more general, non-IWC contexts [27] with protocols such as pubsubhubbub[29] or in message oriented middleware [42]. In the context of IWC, frameworks such as OpenAjax Hub and pmrpc are implementing this concept. Moreover, Publish/Subscribe can be considered a particular variation of the Gang of Four *observer pattern* [38] and allows obeying to the principle of *loose coupling* [27].

In Publish/Subscribe, there are two roles: subscribers and publishers. "Subscribers express their interest in an event […] and are subsequently notified of any event, generated by a publisher, which matches their registered interest." These events are asynchronously propagated to all subscribers that registered interest in that given event". [27]

The advantage of Publish/Subscribe is the loose coupling of the participating parties. Its "event-based interaction style" decouples publishers and subscribers in time, space and synchronization, i.e. "the interacting parties do not need to know each other", "do not need to be actively participating […] at the same time" and the interaction is asynchronously non-blocking so participants can perform "concurrent activities" mean-

[29] http://code.google.com/p/pubsubhubbub/

while [27]. For IWC, in particular space and synchronization decoupling are vital as widgets in ad-hoc built, choreographed UI mashups can hardly know about each other at runtime and widget program logic controls the user interface thus a blocking interaction would render the user interface irresponsive to user inputs.

There are different variations of Publish/Subscribe of which the approach presented in 3 can be considered as a combination of the "Topic-Based" and "Type-Based" Publish/Subscribe. [27]

4.2 Problem

This creates the question as to how to implement such Publish/Subscribe architecture for IWC capable of running on the client side. In particular, in the context of UI mashups the client side runs inside a Web browser so there are only limited possibilities to integrate an IWC framework into the client runtime environment.

These possibilities confine integration of the extended Publish/Subscribe API into the browser itself as native code offering browser primitives, extension of the browser as browser extension / browser plugin or providing a JavaScript Shim using the JavaScript engine of the browser. For the JavaScript variant, there is the option of building a framework on top of some existing communication framework as well as the option of creating a framework from scratch. [7]

Another issue to consider is the problem of cross-context communication. Recent mashup engines such as Apache Rave[30] and mashup applications like Scrapplet shield themselves from foreign code by encapsulating widgets into iframe containers with different origins making these widgets subject to the Same Origin Policy (SOP) of Web browsers [43].

4.3 Discourse

Obviously, the possibility of integrating IWC into the native code of a Web browser offers a certain degree of flexibility due to not being subject to cross-context concerns.

[30] http://rave.apache.org/

However, for a proof-of-concept implementation this method is considered inadequate for the following reasons.

IWC in UI mashups is a relatively specific issue, so changes to the source code of a browser are very unlikely to be merged into the main development branch. The scientific criterion of reproducibility would hence be jeopardized because reproducing the implementation presented in this thesis would have to be reliant on a certain – potentially deprecated – version of the browser. Taking into account the rapid development of recent Web browsers [44], this approach seems not suitable.

Additionally, extending the native code of a Web browser and providing new browser primitives always is highly security relevant and "can increase the attack surface for Web applications in unexpected ways" as it has been demonstrated for the postMessage and Web Storage API [45].

Moreover, there is a wide range of Web browsers currently in use [44] thus an implementation for one specific browser would be insufficient whereas integrating the IWC framework into several browsers would exceed the scope of this thesis. The same disadvantage applies to browser extensions and plugins with the additional drawback of being subject to the Same Origin Policy [43].

The remaining option is to implement the framework as JavaScript shim. This approach has been successfully used for instance for adding support of upcoming W3C drafts such as Web Workers[31] [46] or Web Intents[32] [40] to browsers which do not yet integrate them. Moreover, considering current server-side JavaScript implementations such as Node.js[33] using JavaScript is beneficial because, if the framework has to be extended in future, parts of the client-side JavaScript code can be easily migrated to or reused by JavaScript-based Web servers. If for example mediation concerns are getting too exhaustive to be run entirely on the client, this aspect could be outsourced onto the server.

One issue to address in client-side JavaScript solutions is the question as to how cross-context communication can be enabled in spite of SOP. Off course, one could build the framework on top of an existing communication framework such as OpenAjax Hub. In order to use the API of framework of this kind, particular system messages could be

[31] http://www.w3.org/TR/workers/
[32] http://www.w3.org/TR/web-intents/
[33] http://nodejs.org/

designed which are sent via the underlying communication framework and contain all additional information needed and published to dedicated system channels / topics.

This would be sensitive, if the proposed framework were on a higher layer of abstraction such as a GUI-based framework which transports information about UI level events over an underlying Publish/Subscribe Hub as proposed by Fischer et al. [4].

The approach presented in this thesis, however, is situated on a similarly low level as the communication frameworks themselves. Building two approaches of similar levels on top of each other bears unnecessary redundancy thus decreasing performance.

As a consequence, the proof of concept implementation of this thesis will be developed as JavaScript shim from scratch without using a communication framework.

4.4 Related work

The previous discourse section argued in favour of implementing the IWC framework as JavaScript shim from scratch. In order to benefit from the experiences of current cross-context communication frameworks, this section continues with an analysis of the reference implementation of OpenAjax Hub and pmrpc. The possibility to change and extend their existing source code is disregarded because both focus on offering several messaging possibilities unnecessary in this context and due to licensing concerns.

OpenAjax Hub

The following analysis is based on the OpenAjax Hub 2.0.7 reference implementation provided by the OpenAjax Alliance under an Apache 2.0 license [47] as well as on the OpenAjax Hub 2.0 Specification[8].

The OpenAjax Hub framework, capable of communication across iframe borders and hence being analysed, consists of three basic components: The ManagedHub, the Container and the HubClient.

The ManagedHub is the central Publish/Subscribe component which keeps track of subscriptions and marshals incoming messages to the corresponding subscribers performing security checks. It is usually instantiated once by the mashup application

(Manager Application in OpenAjax terminology), although there may be more than one instance of it, by calling the OpenAjax.hub.ManagedHub constructor.

Thereafter, the mashup application is supposed to create instances of one of the two container implementations: OpenAjax.hub.InlineContainer or OpenAjax.hub.IframeContainer, and appending the widget code to the DOM[34] tree.

In the widget-side JavaScript, called ClientApp in OpenAjax terms, a HubClient instance, again existing in the two variations OpenAjax.hub.InlineHubClient and OpenAjax.hub.IframeHubClient, is created. The HubClient is connected to the ManagedHub by calling its connect function.

When messages are to be published, the widget uses a HubClient instance and calls its publish function. The HubClient forwards the message to its parent container, which forwards it further to the ManagedHub. The ManagedHub checks whether the communication is allowed using security manager callbacks and then reverses the message flow by sending the message to the container of the subscriber. This container invokes the registered callback function of the widget.

Having considered the basic components and the flow of messages in OpenAjax Hub the analysis continues with a look on how the messages are transferred across different origins and how the subscriptions are saved.

The transport layer is abstracted by OpenAjax.gadgets.rpc. It handles communication between widgets and their containers and vice versa as well as communications between widgets across container borders. If source and target are on the same origin, the target function will be called directly, otherwise several communication methods are provided. The choice of a method depends on which communication method is supported by the browser. The default is the so called *fragment-identifier messaging* which is implemented in OpenAjax.gadgets.rpctx.ifpc.

Fragment identifier messaging methods have been in use for IWC as early as in 2008 [48], [49]. They exploit the possibility of creating iframes situated in the destination origin which can be accessed by the communication target. Data is passed by adding the data encoded as strings to the fragment identifier part of the iframes source URL. Several variations based on this approach exist, in OpenAjax a source widget on domain A creates an invisible iframe on domain B of the target widget it wishes to send a

[34] http://www.w3.org/TR/DOM-Level-2-Core/

message to, pointing to a special file called relay file, the message is encoded in the fragment identifier. The relay then finds the target widget on domain B (which equals its own domain and therefore SOP does not interfere with this access) and calls a receipt function passing the message to it. Although this method seems rather inconvenient with current browsers supporting simpler cross-domain messaging methods, it still provides a stable fallback communication for legacy systems.

As the reference implementation was developed in 2009, when support of the HTML5 Web Messaging API [50] was only experimentally supported by some Web browsers [51], another transport layer implementation is OpenAjax.gadgets.rpctx.wpm using HTML5 postMessage.

The wpm transport module provides an implementation of the RPC call function which uses postMessage. Firstly, it retrieves the target window, i.e. the window object in which the target widget is situated. It does so by calling getElementById on the document returning the container element, i.e. the iframe, with the receiver's id. This is possible, because the container implementations create the surrounding HTML elements, i.e. the div or iframe elements, themselves and add an id attribute to them. On the retrieved target container element, the contentWindow property yields the target window object. Another parameter needed for postMessage is the destination origin. This is saved in an associative array with the receiver's id as key.

In order to encode the RPC call, the data parameter of postMessage is assigned either a string, such as "__ack", for simple intra system messages or an object in the form which is depicted in Listing 4.1.

```
var rpc = {
    s: serviceName,
    f: from,
    c: callback ? callId : 0,
    a: Array.prototype.slice.call(arguments, 3),
    t: authToken[targetId],
    l: useLegacyProtocol[targetId]
};
```

Listing 4.1 Structure of the RPC message object sent via postMessage()

To conclude the findings in the analysis of the OpenAjax Hub reference implementation, an additional container component is considered unnecessary. Recent mashup

engines such as Apache Rave encapsulate widgets into iframes themselves; hence comparable JavaScript container implementations will not be implemented. Instead, a client component similar to the HubClient will be implemented which runs inside the iframes and provides a publish method for sending messages as well as a subscribe method to register callback functions. On the transport layer, postMessage can be used due to the improved availability of this API in current browsers [51]. Instead of retrieving the references to the widgets' content windows for each call, they will be stored. Another aspect which is worth noting is the modular fashion in which the OpenAjax Hub framework is built. The possibility to change the implementation of the transport technique depending on the browsers capabilities seems beneficial.

Pmrpc

In order to conduct a thorough analysis, another, more elaborate client-side communication framework called pmrpc is considered. It supports the three communication models "message-based, remote procedure call and publish-subscribe" and employs HTML5 postMessage as transport system and JSON-RPC [52] as communication protocol. This sub-section will present a brief analysis of its architecture and code, available from GitHub [53], with focus on the Publish/Subscribe-relevant aspects. [28][54]

The pmrpc programming interface defines a set of methods: Register, Unregister, Call and Discover, which provide a unified interface for all supported communication models and which are therefore not specific for Publish/Subscribe. As a consequence, this interface will not be considered for the design of the Publish/Subscribe interface. [28][54]

Support of Publish/Subscribe is built on top of a basic RPC-infrastructure. Pmrpc distinguishes between receiving and sending contexts. Within the receiving context, there is the "Receiver controller" which registers "Incoming message handlers" to the on-message event dispatcher of the browser. The "Registered procedures storage" object stores names, callback functions and access control rights of the registered procedures. Upon receiving messages, a "JSON-RPC serializer" is used to deserialize the incoming messages. The requested method is looked up in the "Registered procedures storage" object and invoked by the "Receiver Controller". Its result is serialized as JSON-RPC message and sent back to the requesting context. [28]

The sending context has a “Request Controller” which offers the “Call” method. When invoked, it uses the “UUID generator” to create a request identifier and stores the request in the “Active Requests storage”. If called with an array of destination contexts, JSON-RPC messages will be created and sent to these contexts, otherwise the “Discovery controller” is invoked in order to “dynamically discover which contexts implement the named remote procedure.” [28] This controller first discovers all reachable contexts and then calls a special system remote procedure called “Discover Registered Procedures” which is registered by pmrpc on initialization and returns the content of the “Registered procedures” object. The replies are then aggregated and filtered using regular expressions. [28]

In order to use pmrpc for Publish/Subscribe, the RPC interface is used with a changed meaning of its parameters. For a subscription, the Register method is called stating the name of the Publish/Subscribe topic as “publicProcedureName”. Publishing is done by using the call method with the string “publish” as destination, instead of providing an array of destination contexts and again assigning the intended topic to the “publicProcedureName”. As explained above, leaving out a target destination context in the call method will trigger a dynamic discovery which identifies all reachable contexts by “recursively visiting every iframe in the window.frames object starting from window.top”. [54]

Drawing conclusions from the analysis of pmrpc, it has to be underlined that pmrpc focuses on RPC. Publish/Subscribe features are made available through the RPC interface, i.e. using call and register, and are also implemented on top of the RPC infrastructure. This approach is not considered suitable for the implementation of a pure Publish/Subscribe solution as drafted in section 3.3. However, pmrpc also bears advantageous aspects. In particular, the consequent separation of concerns introduced into its architecture with separate Controllers for each concern such as the Receiver Controller, Send Controller, Request Controller and Response Controller shall be considered for the implementation. Generating a UUID to identify and keep track of each communication is an RPC-specific behaviour used for acknowledgements which could, however, be considered as a solution to avoid infinite loops in the communication flow.

Instead of the dynamic discovery which is performed by pmrpc to aggregate the information stored in the local Registered procedures objects, in the prototype implementation a rather centralized approach will be pursued. The dynamic discovery behaviour can be reasoned with the more general cross-context communication purpose of pmrpc which has been built for a sincerely distributed context landscape with mul-

tiple nested contexts, e.g. iframes containing other iframes containing others and so on. Therefore, each pmrpc instance is capable of working as a relay which forwards RPC messages received from child contexts to its parent context. In IWC, which can be seen as a specific subset of cross-context communication, the nesting hierarchies are usually flat with only one level of iframe containers containing the widgets nested in the parent, i.e. mashup application, context.

In the prototype implementation, the central framework instance running in the mashup application context will receive all necessary information from the IWC clients upon subscription and store it in a global storage. Especially, the content window references dynamically discovered by pmrpc will be stored instead alongside with the subscription. However, the information about callbacks for the subscription will be stored locally on the IWC client side and will not be transferred to the central framework instance as in pure Publish/Subscribe this information is only needed locally upon receipt of a publication to determine the callback function to be invoked.

4.5 Implementation

In this section several aspects of the implementation of an IWC prototype according to the draft elaborated in 3 will be presented. In addition to OpenAjax Hub and pmrpc which have been discussed previously, also other IWC approaches such as SMash[48] and OMOS[49] have influenced this prototype. In 4.5.1 the basic components and architecture of the IWC framework will be detailed, 4.5.2 presents the API from a Widget Developer's perspective focusing of the basic usage of the framework, in 4.5.3 the structure of the messages used for communication is shown and 4.5.4 details transport aspects of these messages.

As previously reasoned, the prototype implementation will be created as a client-side JavaScript framework. For improved flexibility and readability, this prototype is written in CoffeeScript[35] which transcompiles to JavaScript. The following listings show, unless explicitly stated, code in CoffeeScript syntax. However, CoffeeScript does not add any extended functionality compared to JavaScript, hence CoffeeScript code features the same capabilities and restrictions as JavaScript. In the following, JavaScript

[35] http://coffeescript.org/

specific issues will be detailed in reference to the ECMAScript specification [55], which forms the formal core of JavaScript.

4.5.1 Architecture

As observed in OpenAjax Hub above as well as in recent IWC approaches such as OMOS [49] and Smash [48], the IWC framework is internally structured according to a *layered communication stack*, which is depicted in Figure 4.1. The hub layer provides the basic public Publish/Subscribe interface which can be used by widgets. It uses the mediation and message building services of the communication layer situated below. The communication layer then uses the transport layer to send its messages to the recipient and the transport layer abstracts the low-level communication mechanisms provided by the Browser. This layered communication stack realizes the architectural pattern of a *multi-tier architecture*. The guiding principle behind employing this pattern is the paradigm of *loose Coupling*.

The advantage of a multi-tier architecture lies in the possibility to evolve the loosely coupled tiers independently of each other or even exchange the concrete implementation of a tier. As the interaction is limited to well-defined, usually thin interfaces between the tiers, this can be achieved without having to change the entire implementation. Another advantage is the *separation of concerns* with each concern being presented by one tier. From bottom to top, each tier provides a higher level service to the next tier adding specific functionality. Multi-tier architectures and loose coupling have been extensively discussed in the context of Web Services. [24][56]

In order to de-couple the framework components and have them communicate with the neighbouring layers only, a global object called „IWC“ is defined as property of the window object. The framework layers are declared as property members of this IWC object:

- IWC.Hub for the hub layer,
- IWC.Communication for the communication layer and
- IWC.Transport for the transport layer.

Upon initialization, a concrete implementation is constructed and assigned to each of these layers as shown in Listing 4.2. Instead of a monolithic Hub having “private”

members as in 3.3, the originally proposed architecture is refined in that the Hub itself becomes the top layer of a layered IWC stack. Each layer implementation interacts with its surrounding layers only via the global IWC stack, e.g. the communication layer implementation creates a message and sends it using IWC.Transport which can be seen in Listing 4.3. The advantage of this architecture is again loose coupling: In antithesis to a Hub class which implements the same functionality in member class instances with sub-members, the initialization of the stack is defined in one place and any layer can use the other layers centrally via the global IWC object. Exchanging an implementation of a certain layer only requires a different constructor call in the initialization section of the framework.

On the hub layer, there are two different implementations: HubClient for the client, i.e. widget side and IWCController for the server[36], i.e. mashup application side. They differ in their tasks and in the data they store. The HubClient employs a ClientRegistry component for storing the local subscriptions. Each local subscription consists of a subject and callback. When the message handler of the HubClient receives a message, it looks for a subscription corresponding in regard to its subject to the message received in the ClientRegistry and invokes the callback function which the widget registered, passing the message data. For publications, the HubClient uses the communication layer implementation ClientCommunication to create a message and send it to the IWCController.

There, the message handler will check for each received message whether it is a system message or a regular publish/subscribe message. For regular messages, the IWCController looks for the subscribed receivers using the ServerRegistry. This storage component keeps the global subscriptions consisting of subject and subscriber identifier. Additionally the transport layer saves a reference to the subscriber's contentWindow which is a technical aspect accelerating the notification of subscribers by avoiding to either discover them or broadcasting the message as seen in pmrpc [28]. Subscriptions are received as system messages and will be stored if there is not yet an equivalent subscription in the ServerRegistry.

Upon receipt of a message event, the registered message handler of the transport layer, which abstracts cross-context communication functionality, is invoked and extracts the

[36] In the following section server side refers to the mashup application JavaScript context, which technically runs on the client side, i.e. inside the browser, and should not be confused with the Web server, which is not involved in the communication solution presented.

cross-context communication relevant information, i.e. origin and content window reference, and passes the message up in the IWC stack to the communication layer. This layer extracts the messaging relevant information such as subject and payload data and forwards the message up to the hub layer. On this layer, top level functionality like registration of subscriptions or notification of subscribers is performed.

Likewise, sending a message always starts on top of the IWC stack, on the hub layer, which calls the communication layer to build the message structure. The communication layer then forwards the message to the transport layer which looks for origin and content window in its storage and sends the message via postMessage. The transfer of messages up and down in the IWC stack, with each layer extracting or adding information relevant for it, is inspired by the network stacks like ISO OSI[37] or TCP/IP[38].

```
if not IWC.Hub
    IWC.Hub = new HubClient
if not IWC.Communication
    IWC.Communication = new ClientCommunication IWC.Hub.clientId
if not IWC.Transport
    IWC.Transport = new PostMessageTransport true
```

Listing 4.2 Initialization of the IWC layers on the client side

```
subscribe: (subject) ->
  subscription = @messageBuilder.buildSubscription(subject)
  id= subscription.id
  message = @messageBuilder.buildSystemMessage(
@clientId,subscription)
  target = "IWCControler"
  IWC.Transport.send(target, message)
```

Listing 4.3 Communication layer using send service of transport layer

[37] http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip
[38] http://tools.ietf.org/html/rfc1122

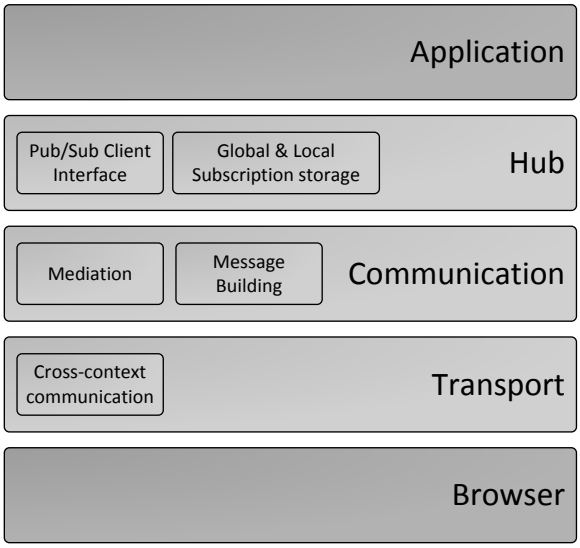

Figure 4.1 Layered communication stack and layer responsibilities

4.5.2 API

This sub-section will present the framework prototype from a Widget Developer's point of view. As argued in 3, it is attempted to keep the API as lean as possible. The layered communication stack presented in 4.5.1 implies that widgets, situated in the Application layer, interact with the hub layer only. Figure 4.2 shows the UML class diagram of the HubClient interface. On the client side, there are only two methods: publish and subscribe. Publish has a string parameter subject stating the subject of the publication. The data parameter can be any object holding the data of the message. For subscriptions, subscribe is used. The string parameter subject is used in the same way as for publish, the callback parameter receives a reference to the callback function which has to be called when a message suitable for the subscription is received. In Listing 4.4 an example of the usage of subscribe can be seen.

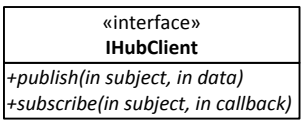

Figure 4.2 HubClient interface

```
onSubject = (message) ->
[…]

subject = “subject”
IWC.Hub.subscribe(subject, onSubject)
```

Listing 4.4 HubClient usage example: subscription

4.5.3 Messages

The structure of the messages created by the communication layer consists of four basic parts:

- Sender
- Subject
- Format
- Data

These messages are represented by anonymous JavaScript objects with corresponding properties. Sender is either a UUID[39] representing the hub layer client instance which sent the message or the string “IWCController” to identify the IWC server-side Hub. The subject can be an arbitrary string which should, however, comply with the guidelines stated in 3.4. The format part is a slight anticipation of the following mediation chapter introduced already in order to avoid having to re-define the message structure. As this part of the message is relevant for mediation it will be detailed in chapter 5. The data part of a message can be any JavaScript object, e.g. a string or an XML DOM Document, which transports the message data. Following the HTTP terminology, subject and format form the header of the message whereas data is its body. The basic message object structure can be seen in Listing 4.5.

```
message={"sender":id,"subject":subject,"format":format,"data":data}
```

Listing 4.5 General structure of a message object

[39] http://www.itu.int/ITU-T/studygroups/com17/oid.html

In addition to these regular Publish/Subscribe messages which transport the communication payload, the framework employs system messages. These messages can be sent using the same structure. The subject and format of a system message is the string "SYSTEM" written in capital letters which is therefore a reserved subject.

Subscriptions are enveloped into the data part of a system message. They have three properties: type containing the string "SUBSCRIBE", subject to represent the intended subject string of the subscription and formats containing an array of format identifier strings (a mediation aspect introduced later, cf. 5.5.1).

4.5.4 Transport

As described in the conclusions of section 4.4, the HTML5 Web Messaging API is made use of for implementing the transport layer. This specification defines "a messaging system that allows documents to communicate with each other regardless of their source domain, in a way designed to not enable cross-site scripting attacks." [50] In order to keep communication across different origins secure, only messages with a destination origin matching the origin of the recipient will be delivered. Therefore, sending a message not only requires a reference to the parent window of the destination, but also the destination origin. [50]

Retrieving the destination window can be implemented easily. A widget, encapsulated into an iframe, announces its subscriptions by calling the "postMessage" function of its topmost parent window, accessible via the "window.top" property. The transport layer receives the corresponding message event and gains a reference to the window containing the widget from the "event.source" property. Likewise, looking at the "event.origin" property, the origin of the widget's iframe can be obtained and used as destination origin for messages from the IWCController to the widget. This behaviour also ensures that parent windows do not commence posting Web messages to their children before they announce readiness as demanded in section 4.3 of the specification. [50]

Storing both source window and origin of the received message events along with the sender's id on the transport layer enables the framework to send messages to the destination's window and stating the correct destination origin without having to look for it like OpenAjax Hub or broadcasting like pmrpc.

However, there remains one problem with using the Web Messaging API: On initial calls to the parent window – in this implementation these calls are subscriptions – only the reference to the parent window can be retrieved. The parent's destination origin, however, is not accessible. Attempting to read the "location" property of "window.top" fails because of violating the same origin policy [43]. As there is no way to get the origin the IWCController runs in directly, one possibility would be to implement a configuration property which can be set by the Widget Developers. This would however reduce the independence of the widget development process, which is a requirement of this thesis stated in 2.1.4, as these widgets would then be configured to run in but one mashup and could not be used in other mashups with different origins.

Changing the initial call to be triggered by the IWCController reveals the hen-egg nature of the problem: Even though the widget can get the parent origin from the message event, the IWCController has to find a destination origin for its initial call. Moreover, this approach seems not very feasible as it would require polling the DOM for newly added iframes.

Analysis of OpenAjax Hub shows that the origin of the parent is retrieved from the "oahParent" parameter which is stored URL-encoded as query parameter of the iframe "src" attribute. This is possible, however, only because OpenAjax Hub creates the iframe containers of its widgets on its own and can hence set the necessary attributes. This solution cannot be considered suitable here because recent mashup engines such as Apache Rave [57] create their own widget-surrounding iframes, so IWC solutions integrated as JavaScript libraries cannot influence the iframe attributes. Pmrpc uses the wildcard destination origin "*"for initial calls, e.g. for its discovery mechanisms. According to the specification, this should be avoided [50]. Nonetheless, this seems the only solution feasible and will therefore be adopted.

Even though security is not a main aspect of this implementation, handling of the destination origin of initial calls remains an issue for all postMessage-based communication frameworks and this thesis is unfortunately unable to solve this. Only changes or extensions to the Web Messaging specification could help overcoming this particular problem.

4.6 Summary

In this chapter, it was demonstrated how to build a basic Publish/Subscribe solution using client-side JavaScript. Several general design issues, such as browser integration versus JavaScript framework or building on top of existing communication solutions versus developing from scratch, have been outlined and the according decisions in the prototype implementation have been reasoned. Furthermore, experience from existing IWC solutions has been detailed and taken into account. In the implementation section several technical aspects of the implementation have been considered, the advantage and realization of a layered architecture underlined and general issues in implementing cross-context communication in an environment which is subject to the Same Origin Policy utilizing the Web Message API identified. The resulting framework structure can be seen in the UML class diagram Figure A.1.

5 Mediation

This chapter will research how to integrate basic mediation capabilities into a Publish/Subscribe IWC framework. In order to demonstrate this, this chapter elaborates on the implementation presented in 4. Within the layered architecture presented in 4.5.1, mediation is a task of the communication layer.

5.1 Situation

Early Inter-widget communication solutions such as OpenAjax Hub, OpenSocial Inter-Gadget Eventing etc. lack the support of data mediation [8], [30]. For orchestrated UI mashups, there are approaches with pre-defined transformations such as the "Distributed UI synchronization" defined by BPEL4UI and using "technologies such as XPath, XQuery, XSLT, or Java" [13]. Recently however, mainly in the context of complex mashup platforms such as CRUISE and WIPLE, data mediation has been implemented [31], [32]. Although differing in technical details, the approaches employed to enable mediation are similar. Widget Developers use semantic markup, e.g. RDF, to describe the data formats utilized in their widgets. While CRUISE uses separate widget descriptions in SMCDL, WIPLE prefers embedded semantics technologies such as RDFa in the widgets' HTML markup. When widgets communicate with each other, mediation rules are inferred by performing reasoning on these descriptions. In CRUISE, the reasoning is performed on server side, in WIPLE there is both a client-side reasoner and a server-side reasoner [31], [32].

5.2 Problem

As it has already been argued in chapter 2, existing approaches may work well in these systems, where an entire mashup platform has been purpose-built e.g. for a learning environment, but in more general settings, with heterogeneous platforms consisting in different parts such as widget repositories, a mashup editor and an IWC framework

which have not been jointly developed, less elaborate and less interlinked solutions are necessary.

This raises the question how to provide support for mediation in IWC without requiring Widget Developers to employ additional semantic markup languages. As of now, a comparable solution which would provide experience one could benefit from could not be found.

5.3 Related Work

As there is no related work available for mediation in IWC without additional markup, the issue of mediation has to be considered in a more general context. Research in mediation dates back into the early 1980s when middleware systems enabling to link newly developed with legacy systems became increasingly important. In the context of Service Oriented Architectures, Service Mediation has been researched [16]. Earlier research of Mediation revealed several functions of Mediation, such as data transformation, integration of data from heterogeneous sources and abstraction or generalization of data [14] [15]. In Service Mediation, there are three levels: data mediation, message mediation and process mediation. [16]

Mediation solutions like the MOMIS-STASIS approach employ a semantic markup and reasoning approach [18] similar to CRUISE. Others such as the Apache Synapse Enterprise Service Bus use XML rules to define mediation, in particular XSLT for data transformations [16].

5.4 Discourse

In the context of IWC-enabled UI mashups, not consider all levels of mediation are considered necessary. As the main purpose of IWC in UI mashups is widget synchronization, the aim of mediation is to support communication about common semantic concepts between syntactically incompatible participants [13]. Therefore, in the following mediation is defined as “manipulating messages in-flight on the ‘bus’ […] [so that] messages initially sent [by a widget] are transformed into messages understood by […] incompatible [receiving widgets].”[16] The focus, in terms of Service Mediation, lies on data transformation.

Widgets in UI mashups are “small web applications” interacting with “background processes” [6], usually interfacing some Web Service accessible through an API [3]. When the business logic of the widget interacts with the underlying API, it employs certain data representations. Hence, it is considered advantageous if the same representations, or at least those parts which the widget is willing to share, could be used in Inter-Widget Communication directly, as they are available without additional effort due to the interaction with the API.

Considering the data representations employed by Web Service APIs based on the statistics of programmableweb[40] accessed on 19 July 2012 XML-based (4175) and JSON[41]-based (2931) representations were identified to be the most common ones. In addition to those, plain text is considered a simple yet powerful representation option.

The decision upon how to implement the data format transformations can be reduced on to how Widget Developers transform data. Assume, for instance, using an IWC framework without interoperability support a widget is configured to receive messages on a certain channel. The Widget Developer knows, for example due to a certain agreement on the format used on this channel, the syntax of the data received and has to convert it to a representation which can be further used. The most obvious way to implement this conversion is to simply program it. For example, if receiving an XML fragment, the Widget Developer can have a DOM parser parse the input and then use its functions to traverse the XML tree and extract the information needed.

If the Widget Developer has reasonable experience with XSL Transformations (XSLT) he could also define the transformations using XSLT to be applied to the received data in order to create an XML representation which is processable [23]. If JSON has to be transformed, there are several approaches such as JsonT [58], JXL [59] and json2json [60]. For cross-conversions from XML to JSON XSLT extensions like XSLTJSON [61] could be employed. The significant drawback of transformations involving JSON compared to XML and XSLT is that there is not yet any approach which is widely accepted and used nor any standard or specification. Processing plain text, i.e. a string, Regular Expressions as represented by the RegExp object defined in ECMA Script [55] can be used to match particular patterns and replace these matches.

[40] http://www.programmableweb.com/apis/directory/1
[41] http://www.json.org/

5.5 Implementation

This section will detail how to progress from the Publish/Subscribe implementation presented in 4.5 towards a solution incorporating Mediation features. The following 4 sub-sections are to be considered subsequent steps necessary to achieve this objective.

5.5.1 Data format identifiers

Inspired by the two-part definition of Internet Media Types [62], two-part data format definitions are utilized also. Although stated in 3 that a data format can be described by any string, the following usage is advocated.

```
<representation>/<syntax>
```

Figure 5.1 Data format identifier structure

The first part of a data format identifier states the *representation* used. These representations can be either

- Text,
- XML or
- JSON

The advantage of separately defining the representation is that upon receiving a message it can be easily decided which tool to use: For plain text the string can be left as received, for XML a SAX or DOM parser can parse the received string as for JSON a JSON parser can be employed.

The representation part is separated by a single forward slash ("/") from the second part which defines the particular *syntax*. For XML, this is sometimes referred to as the XML dialect. If necessary, the syntax part can be used to distinguish particularly specific formats by utilizing reverse DNS names as proposed for the topic names in the OpenAjax Hub specification [8], e.g. "json/org.company.specific.syntax".

For example, "text/iso6709" can be used to identify the "Text string representation of point location" defined in Annex H of ISO6709 whereas "xml/iso6709" to identify the

XML representation as of Annex F [63]. For user input directly retrieved from the user interface, "text/user-input" is proposed as data format identifier. For instance, if the prompt of a map widget does not require any specific address syntax, the user input "Mt. Everest" can be published onto the "Location" subject and labelled as "text/user-input".

Taking into account an approach for data type declaration observed for instance in [4] and [5], "json/name-value-pairs" is recommended as data identifier for JSON objects with properties representing *name-value pairs*. These pairs are very flexible if the property names are employed to reference semantic concepts. For example in [5], the pair "'http://purl.org/dc/elements/1.1/title' : 'News story'" is used to type the title string referencing the corresponding Dublin Core[42] element, in [4] a similar idea applies to "'http://dbpedia.org/ontology/city' : 'aberdeen'" referencing the concept city using RDFS and OWL. The same usage is recommended for the data format identifier "json/name-value-pairs", which shall be utilized to mark JSON objects with self-describing properties referencing well known entities such as Dublin Core elements or vertices of the Linked Data[43] graph.

Data format identifiers are used to label outgoing messages, i.e. information to be communicated to other widgets using the publish mechanism, as well as to declare the list of accepted formats, i.e. the formats a widget can understand and thus requests to receive according messages, upon a subscription on a certain subject. Usually the number of formats a widget is capable of understanding is rather low. If, however, the service API underlying the widget is complex and able to handle a variety of formats the widget could receive any relevant message, regardless of its format, and pass the data to the service which probably can parse it.

Thus, on subscriptions it is possible to use the *wildcard data format identifier*, a single asterisk ("*"), to denote interest in receiving messages regardless of the data format. This coincides partially with the principle of "*partial semantic interoperability"* in that the efforts of understanding the messages sent are partially shifted from the framework to the recipients' side [5]. In contrast to Isaksson and Palmer, however, this thesis does not advocate implementing this in the widgets, instead one can argue that there are some well-established Web Services which incorporate support for different data for-

[42] http://dublincore.org/
[43] http://linkeddata.org/

mats already and hence neither additional effort developing a transformation nor redundancy by adding this support to the widgets is required.

5.5.2 Extending the Publish/Subscribe interface

In the preceding sub-section it was defined how to identify data formats in the context of the approach described in this thesis. This sub-section will demonstrate how the basic Publish/Subscribe interface detailed in 4.5.2 can be extended to incorporate information on the data formats used or accepted employing these identifiers.

For sending a message, the publish method of the HubClient is simply extended by one additional string parameter which represents the data format used. For subscriptions, two additional parameters are added to the signature of the subscribe method: the string parameter format stating the accepted format identifier and the integer parameter priority. Figure 5.2 shows the resulting extended HubClient interface.

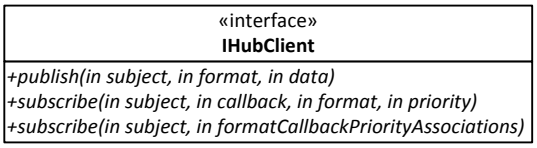

Figure 5.2 Extended HubClient interface

Priorities are introduced in order to grant the Widget Developer influence on the choice of a data format which should be delivered to his widget if there are several accepted formats available. It can be used, for instance, to prefer a data format which is easy to parse and process for the widget over others, which it still is capable of understanding, but which require more extensive parsing or even interaction with the underlying service. The *priority* assigned to a certain format identifier has to be a positive integer greater than zero. Although it is recommended to use this parameter, its usage is optional. If subscribe is called with only three parameters – subject, callback and format – priority will internally default to one. The priority of the all-format subscription using the wildcard identifier "*" will always be zero, regardless of which priority was assigned to it. Even the format parameter can be considered optional, if subscribe is called with subject and callback only an all-format subscription is assumed, i.e. subscribe(subject,callback) equals to subscribe(subject,callback,"*",0).

Apart from defining a linear order on the accepted formats, the priority values bear no further significance. A format with priority 4 is not twice as likely to be received as one with priority 2. Instead, assigned priorities influence the choice of a data format in that the framework will always return a message in the data format with the highest priority of all accepted formats of the recipient which it can produce, i.e. where it can find a way to transform the original message into this format. This behaviour is similar to the "*best effort delivery*" employed in networks [64].

For convenience, an overload of the subscribe function has been added which allows to subscribe on a subject with several formats, callbacks and priorities. The associations between formats, callbacks and priorities are declared as an array object of several anonymous objects with format, callback and priority properties as shown in Listing 5.1. The resulting subscriptions according to the rules defined previously are indicated in Table 5.1.

```
onSubjectXml = (xmlMessage) ->
[…]
onSubjectJSON = (JSONMessage) ->
[…]
onSubject = (message) ->
[…]

subject = "subject"
formatCallbackPriorityAssociations = [
  {"format": "xml/format1", "callback": onSubjectXml,
"priority":5},
  {"format": "json/format2", "callback": onSubjectJSON },
  {"callback": onSubject }
]

IWC.Hub.subscribeMany(subject, formatCallbackPriorityAssociations)
```

Listing 5.1 HubClient usage example: subscription with several formats

format	priority	Callback
xml/format1	5	onSubjectXml
json/format2	1	onSubjectJSON
*	0	onSubject

Table 5.1 Resulting subscriptions

5.5.3 Data transformations

In the previous sub-section, an extension to the Publish/Subscribe interface was presented which Widget Developers can use to declare the data format of the messages they send and to specify the data formats of messages their widgets are able to understand when receiving them, the means of achieving the desired behaviour, however, have been treated as a black box. This sub-section will detail the bottom-most aspect of the black box which is the implementation of the single transformations.

Any transformation can be interpreted as a function t which relates an input x of format F_x to an output y of format F_y. Transformation inputs are elements of the *transformation function*'s domain, outputs are elements of its codomain. This can be seen in Figure 5.3

$$t: F_x \rightarrow F_y$$
$$t(x) = y, x \in F_x \; y \in F_y$$
t – Transformation function
F_x – Domain of t
F_y – Codomain of t

Figure 5.3 Transformation function

Therefore, each transformation is defined by three parts. The abstract class Transformation shown in Figure 5.4 represents those three parts: the property "from", storing a data format identifier, is equivalent to the domain, the property "to" represents the codomain and the property "transformation" defines the transformation function.

In section 5.4 several common possibilities to transform data have been discussed. They can be integrated into the abstract transformation definition by creating concrete, i.e. non-abstract sub-classes which inherit from the abstract Transformation class. For the prototype implementation, two implementations will be demonstrated: algorithmic transformations and XSL transformations. Algorithmic transformations will be implemented because they represent the most powerful and flexible way for developers to define their transformations in JavaScript source code and XSL transformations as a representative of the group of formalized transformations represented by transformation-domain specific languages such as XSLT, JsonT or XSLTJSON. The choice of

XSLT can be reasoned with both the popularity of XML in service APIs as indicated in 5.4 and with its widespread support in major browsers without necessitating the inclusion of additional frameworks [65].

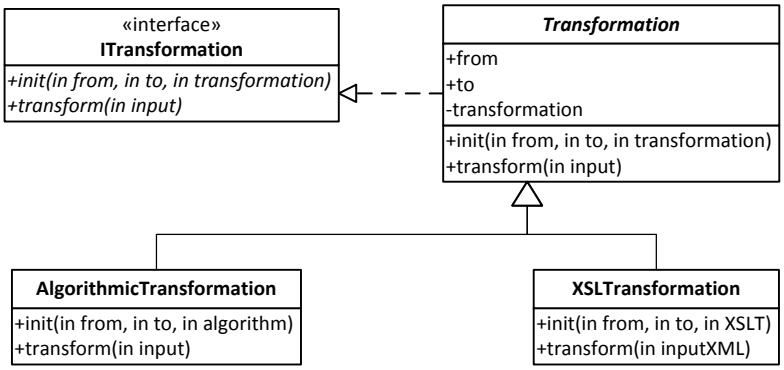

Figure 5.4 Transformation class hierarchy as strategy pattern

In the class diagram in Figure 5.4 algorithmic and XSL transformations are represented by the two classes correspondingly named on the bottom. Together with the ITransformation interface and the abstract Transformation class, they form an instance of the Gang of Four design pattern called *strategy pattern*. "ITransformation" defines a general strategy confining the algorithms "init" and "transform" and is implemented by the two concrete strategies "AlgorithmicTransformation" and "XSLTransformation". A context component – the Mediator which will be detailed in 5.5.4 – can use transformation class instances, abstracted by ITransformation, to perform interchangeable implementations of the concrete transformation. [38]

Stepping away from the strict formalism of a Gang of Four pattern, the strategy pattern can be seen twice in this diagram. In addition to the one explained, each concrete transformation, i.e. AlgorithmicTransformation and XSLTransformation, can be considered as an abstraction of the even more concrete transformation which is created at runtime when the transformation parameter is initialized. Because its content itself defines an algorithm, the class instances can be considered as concrete strategies of the – not formally – abstract classes AlgorithmicTransformation and XSLTransformation.

XSLTransformation implements the "transform" method by having a DOM parser parse both the "inputXml" string and the string property "transformation" containing the XSLT. The XSLT is loaded into an XSLT processor which performs the transformation on the input XML. The output of the transformation is of type "Docu-

mentFragment" and holds a reference to the parent document context accessible via its "ownerDocument" property. Therefore, due to the Same Origin Policy [43], this resulting document fragment cannot be passed directly to the recipient and has to be serialized as string. Unfortunately, except for the additional, non-standard "xml" property in the Internet Explorer implementation, there is no default serialization method available for DocumentFragment. In order to achieve string serialization nonetheless, a new DOM element is created using "document.createElement", the document fragment is appended to it using "appendChild" and the serialized string is retrieved by reading the element's "innerHTML" property. [66]

For algorithmic transformations, the prototype provides the AlgorithmicTransformation class. Its "transform" method defines an empty function called "process" which takes one parameter. Thereafter, the JavaScript "eval" function is called on the transformation string property set in init. The output is retrieved by calling the process function on the transformation input and casting the result to string.

This procedure requires the developer of a transformation to implement it as a function accepting one string parameter which will contain the transformation input. Within this function, further encapsulated functions and properties may be defined to achieve the transformation. The framework requires the entire definition of the custom process function as a string. Calling eval on this definition will override the previously declared empty process function and enables to call it subsequently on the transformation input.

In order to pass the function definition to the framework, developers do not have to write it down statically in code, as this would be cumbersome due to formatting problems and the need to escape special characters. Instead, developers can implement the process function directly in code and pass the Function object to the init method of Transformation. However, developers should be aware that toString() will be called on the Function object serializing it to string and hence any references will be lost and will not be available to the process function at runtime. Moreover, when process is called it runs inside the scope and runtime context of the framework, therefore the use of dynamic keywords like this or document should be avoided; details on the environment of eval code can be found in the ECMAScript specification, sub-section 10.4.2. [55]

5.5.4 Transformation planning and caching

In 5.5.3 the implementation of transformations has been explained. This sub-section examines the possibility to use these transformations in order to transform a published original message which has to be delivered to several subscribers into the set of representations which is needed by these subscribers. To be more specific, a Mediator component is introduced into the framework on its communication layer, which is responsible for selecting the suitable transformations, planning an appropriate order in which these transformations have to be executed and caching preliminary results in order to efficiently fill a storage object which the communication layer implementation can then use to retrieve the required representations to build messages for each subscriber. In Figure 5.5 the class diagram of the Mediator can be seen. When receiving a publication the "mediate" method is called by the communication layer passing the original message, a set of available transformations, a storage object for message representations and the notification queue containing information about the subscribers. Although this prototype focuses on data format transformations, the mediate method could be extended to perform other mediation tasks also.

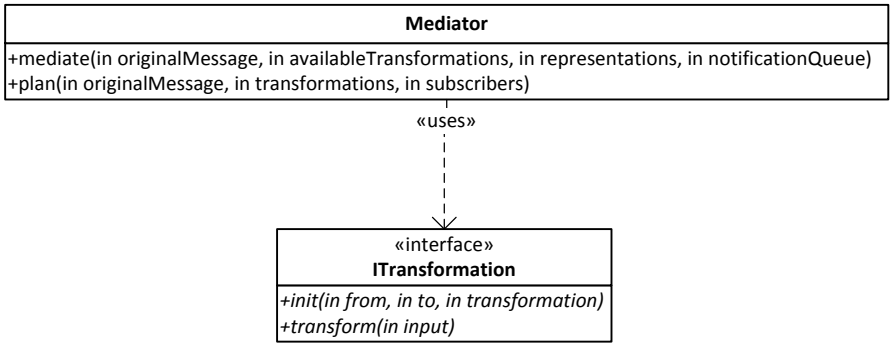

Figure 5.5 Mediator

Previously the concept of transformation functions has been defined. These transformation functions define the set of possible data format conversions. However, given a certain set of transformation functions, also transformations which are not explicitly defined, i.e. transformations without a corresponding transformation function, can be achieved by the mathematical concept of function composition as shown in .Figure 5.6.

This implies that in the framework, transformations can be achieved for which no transformation implementation exists by chaining existing transformations together

forwarding the output of the preceding transformation to the input of the subsequent one.

$$\exists t_1 : F_x \rightarrow F_y \wedge \exists t_2 : F_y \rightarrow F_z$$
$$\Rightarrow \exists\, t_c : F_x \rightarrow F_z$$
$$t_c(x) = (t_2 \circ t_1)(x) = t_2\big(t_1(x)\big)$$

t_1, t_2 – Explicit transformation functions
$\mathrm{t_c}$ – Composite transformation function

Figure 5.6 Composite transformation function

If the framework receives an original message m_0 with format F_0 the Mediator has to plan the necessary transformation sequence. This means it has to find a suitable execution order of transformations which yields messages of accepted formats for the subscribers and tries to minimize the amount of transformations used. The Mediator looks for a plan Π representing an execution order o of transformations P which fulfils the requirements listed in Figure 5.7.

$$\forall\, s_k \in S: \;\; A_k^* \neq \emptyset \Rightarrow \exists t \in P:$$
$$t : F_i \rightarrow F^*, F^* \in A_k^*, \forall F_j \in A_k^* : prio\big(s_k, F_j\big) \leq prio(s_k, F^*)$$

where:

$$\Pi = (P, o)\; P \subset T,\; o : \mathbb{N}_p \rightarrow T$$
$$P = \{o(l) |\; 0 \leq l \leq p\}$$
$$o : \mathbb{N}_p \rightarrow T,\; o(n) = t_{ij} : F_i \rightarrow F_j, n \in \mathbb{N}_p$$
$$prio : S \times F \rightarrow \mathbb{N}$$
$$A_k = \{F_i |\; s_k \in \mathrm{S}:\; s_k \text{ accepts } F_i\}$$
$$A_k^* = \{F_i |\; F_i \in A_k \wedge \exists\, (t_c)_i : F_0 \rightarrow F_i\}$$

S – Subscribers, $prio$ – Priority function
A_k – Accepted formats of subscriber s_k
A_k^* – Available accepted formats of subscriber s_k
$\mathrm{t_c}$ – Composite transformation function

Figure 5.7 Execution order requirements

In the following this problem is being reduced to a graph and a special path-finding algorithm will be investigated. The structure of the graph can be seen in Figure 5.8.

$$G = (V, E)$$

$$E = \{e = (v_i, v_j) | \exists\, t_{ij} : F_i \rightarrow F_j\}$$

$$V = \{v_i | \exists\, t_{ij} : F_i \rightarrow F_j \vee \exists\, t_{ji} : F_j \rightarrow F_i\} \cup A \cup \{v_0\}$$

$$A = \{F_i | \exists\, s_k \in S: F_i \in A_k\}$$

V – Vertices, E – Edges, A – Accepted formats, v_0 – Start vertex representing F_0

Figure 5.8 Transformation graph definition

Its vertices represent data formats, its edges represent transformations. There is a vertex for each source and destination format of each existing transformation function, for each accepted format of each subscriber and a specially marked start vertex representing the format F_0 of the original message m_0.

The above research indicates that finding an optimal Plan, i.e. minimizing p cannot be efficiently computed. One could start with performing a modified breath-first search in G which identifies the shortest paths for each vertex visited and detects unreachable sub-graphs (by not visiting their vertices) which can be removed from the graph. Identifying each A_k^* is then easy because any format in A_k which has not been deleted from G belongs to A_k^*. For each subscriber $s_k \in S$ with $A_k^* \neq \emptyset$ the target format F^* with maximum priority can be determined. Choosing the shortest path to F^* is locally optimal. The problem however, is that due to the costs (the time necessary to compute it) caused by a transformation preliminary results will be cached, i.e. once a transformation has been performed its result, which is a representation of the original message in a different format, will be stored and can be re-used by subsequent transformations. Unfortunately, this aspect significantly complicates the planning phase. In Figure 5.9 one can see an example transformation graph.

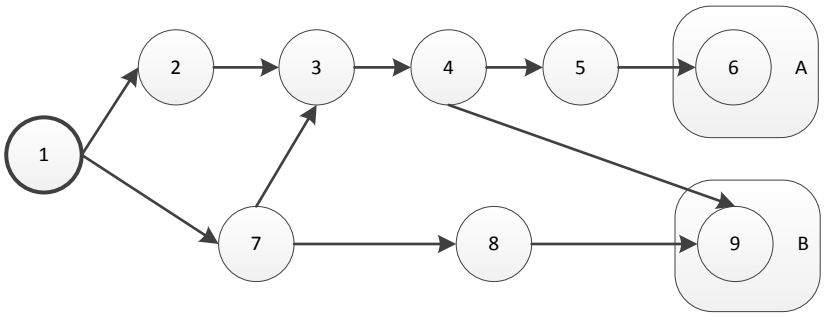

Figure 5.9 Transformation graph example

Vertex 1 represents the format of the original message; the two subscribers A and B each have only one accepted format: A 6 and B 9. Using breadth first graph traversal, one identifies the shortest paths to A(6) as {(1,2,3,4,5,6), (1,7,3,4,5,6)} each of length 5 and to B(9) {(1,7,8,9)} of length 3. However, any path to A contains vertex 4 which implies that if one adds e.g. the sequence (1,2,3,4,5,6) to the beginning of the plan, format 4 will be available in the cache. If the transformations along the graph-wise shortest path to B (1,7,8,9) are performed, the summarized amount of transformations is 5 + 3 = 8. However, once 6 was computed for A, the transformation (4,9) could have been executed reducing the amount of transformations to 5 + 1 = 6.

The example illustrates that due to caching the optimal solution for p is not necessarily the sum of of the optimal sub-solutions for each maximum priority accepted available subscriber format, i.e. it is does not obey to the principle of *Bellman Optimality* [67]. Caused by caching, each local decision for a sub-solution potentially affects all other partial solutions. Imagine the graph in Figure 5.9 has some more subscribers and the planning algorithm tries to identify the globally best path for A. Previously, 2 minimum paths with the same length have been found. Choosing one of them can affect other subscribers because one of the vertices along the chosen path can become the starting point of a shortcut like vertex 4 for the original example. Therefore, identifying the optimal solution would require following both decisions and continuing, again following each decision path which would create a decision tree. The number of variants v in the tree is limited by the maximum amount of decisions d_{max} at one stage and the number of levels l, its upper limit (the number of vertices in a complete decision tree of depth l) is exponential in l: $v \leq d_{max}^{l}$. Moreover, decisions are not only caused by the choice between several minimal paths of the same length, but by any possible path to a specific target vertex due to the non-Bellman nature of the problem. Figure 5.10 illustrates the reason for this: if l and r are shortests paths then $l \leq s + ls$ and

$r \leq s + rs$. Choosing each optimal path yields length $l + r$. However, an optimal solution may exploit a shared path s and the connecting paths rs and ls. Due to caching, the cost for traversing s two times is only s instead of $2s$. Thus for two subscribers, there are two potentially optimal solutions: either $l + r$ or $s + rs + ls$. Due to the optimality of l and r mixed combinations like $r + s + ls$ cannot be better. The number of potentially optimal solutions for three subscribers is 4 and increases exponentially in the number of subscribers $|S|$. Hence, computing an optimal solution by checking all (non-mixed) combinations of paths in G is not feasible.

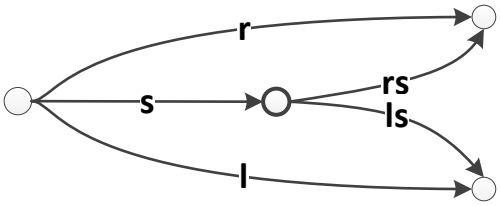

Figure 5.10 Shared path

Although the previous discussion does not formally prove the complexity of the planning problem it does provide strong evidence in favour of it. An idea for a proof is to find a polynomial-time reduction of the *Travelling Salesman Problem* onto this problem, as they are quite similar. Informally speaking, the planning problem presented above is like a Travelling Salesman Problem with the possibility to teleport back to any place already visited. If a polynomial-time reduction can be identified, it is highly likely that there is no efficient solution for the planning problem assumed $P \neq NP$. The concrete proof is considered to be out of the scope of this thesis. [68] As no efficient solution has been found, the framework will use a simpler algorithm trying to approximate a good solution. Its pseudocode is shown in Listing 5.2.

The algorithm adds the initial format F_0 to the set of cached formats C and creates the transformation graph as defined in Figure 5.8. An initial breadth-first traversal starting with the original message format F_0 is employed to remove all isolated sub-graphs, which are unreachable and annotates the shortest path for each vertex.

Algorithm Transformation planning

Input: Original format F_0, subscribers S, available transformations T
Output: Transformation plan $\Pi = (P, o)\ P \subset T,\ o: \mathbb{N}_{|P|} \rightarrow T$

$C \leftarrow \{F_0\},\ D \leftarrow \emptyset,\ P \leftarrow \emptyset, n \leftarrow 0$
Build G as defined in Figure 5.8
Starting with F_0: traverse G breadth-first completely, marking visited vertices and annotating shortest paths
Remove all vertices which are not marked
for all $s_k \in S$ **do**
    if $A_k^* \neq \emptyset$ **then**
        Select $F^* \in A_k^*$ with maximum priority
        $D \leftarrow D \cup \{F^*\}$
        Select shortest path to F^*: $p(F^*) = \{F_0, \dots, F^*\}$
        $W \leftarrow W \cup p(F^*)$
    end if
end for
$W \leftarrow W \cup D$
for all cached formats $F_c \in C$ **do**
    for all destination formats $F_d \in D$ **do**
        if $\exists t \in T$ such that $t: F_c \rightarrow F_d$ **then**
            $P \leftarrow P \cup \{t\}$
            $o(n) \leftarrow t, n \leftarrow n + 1$
            $C \leftarrow C \cup \{F_d\}$
            $D \leftarrow D \setminus \{F_d\}$
            $W \leftarrow W \setminus \{F_d\}$
        end if
    end for
    if $D = \emptyset$ **then return** $\Pi = (P, o)$
    else
        if no destination formats have been created **then**
            Select wanted format $F_w \in W$ and $t \in T$, $t: F_c \rightarrow F_w$
            $P \leftarrow P \cup \{t\}$
            $o(n) \leftarrow t, n \leftarrow n + 1$
            $C \leftarrow C \cup \{F_w\}$
            $W \leftarrow W \setminus \{F_w\}$
        end if
    end if
end for

Listing 5.2 Transformation planning algorithm

In the remaining graph, for each subscriber the available accepted format with maximum priority is added to the set of destination formats D and the vertices along its shortest path to the set of wanted formats W. Additionally, all destination formats are added to the wanted formats. Then, the algorithm iteratively checks the set of available representations against the set of available transformations T until each destination format can be created. It looks for a transformation t yielding a destination or wanted format. When choosing a transformation, it gives priority to transformations yielding destination formats. Otherwise intermediate wanted formats are created. In both cases the transformation will be added to the plan Π and its destination format removed from the respective sets.

Eventually, the algorithm returns the plan set Π. Performing the transformations in P the order defined by o and filling a representations cache just like the cache of the planning algorithm fills this cache with representations until all available subscriber formats, obeying to the custom priorities, are created. The communication layer can then deliver appropriate messages to the subscribers taking the prepared representations from the cache.

Additionally, the results of the planning, i.e. the ordered transformation plan and the association of each subscriber with its best available format, are cached. Upon subsequent invocations of the mediate method, if a cached entry is available matching in sender, subject and format, mediate retrieves the stored transformation plan and best formats available for each subscriber according to the subscription priorities and utilizes this cached information to perform the representations filling. Otherwise, a complete planning is performed. The plan cache computes hashes of (sender, subject, format) tuples and stores all information necessary to re-perform the filling, which grants easy access on to the cached data in a similar mediation situation. Senders are parts of the hash because if a message on the same subject with the same format is published by a different sender, the initial sender may occur as subscriber with additional preferred formats thus altering the initial situation.

In addition to the hash-based cache itself, a list of associations between subjects and hashes is maintained in order to enable access to cache entries based on the subject only. This is required by the handling of cache persistency. If a change in the set of available transformations occurs, the entire cache is emptied because not only does this change enable potentially more efficient transformation plans due to additional edges in the planning graph, but, what is more, it affects the set of available formats for subscribers and hence the “best effort delivery” behaviour defined in 5.5.2 due to poten-

tially adding new vertices. The second possibility for outdating cached planning results are new subscriptions. As these subscriptions merely affect cached entries for the subscription subject, the above explained list of subject hash associations is used to clear all cache entries of the respective subject.

5.6 Summary

In this chapter, problems when combining Publish/Subscribe with Mediation have been discussed. A guideline for data format identifiers has been introduced and the existing Publish/Subscribe interface was extended to incorporate data format information. A general concept and two concrete options to define transformations have been presented and implemented. Finally, the complex issue of transformation planning has been detailed and an approximation algorithm was presented. As a result, this thesis advocates the use of groundings, i.e. main representations, for IWC subjects as this converts transformation graphs into star-shaped graphs thus limiting both planning and transformation complexity. Figure A.2 shows the state of the IWC prototype framework as of this chapter. The UML diagram has to be considered in conjunction with Figure A.1 as only new components and components which have been changed are depicted.

6 Plugins

Already in the early days of mediation there was the idea of the sharing of mediator modules [14]. In the previous chapter a mediation approach was introduced which, inter alia, allows a decoupling between the transformation and the widget logic. This chapter will elaborate on this concept leveraging the separated transformations for the purpose of extensibility and re-use. In order to achieve this, a plugin architecture is incorporated into the existing framework. In the following, problems will be outlined, general extensibility and re-use solutions will be considered and the vital aspects of the implementation of a plugin architecture within the IWC framework will be discussed.

6.1 Situation

The preceding two chapters detailed how to create a client-side Publish/Subscribe IWC framework which enables widgets to receive and send messages independent of syntactical concerns by adding the Mediation aspect. In sub-section 5.5.3 a method of defining data transformations has been introduced. Obviously, it is not feasible to create a framework which implements all transformations possibly required by the widgets using it. Therefore, in 3.2.2 the concept of a "Thin architecture" which provides support for extensibility of its own, limited functionality has been presented and argued in favour of in 3.2.4. The basic Publish/Subscribe functionality is regarded as core functionality whereas the capabilities of the Mediator, i.e. the set of available transformations, should be extensible.

6.2 Problem

In order to achieve this extensibility, it is necessary to determine which changes to the existing framework architecture as of chapters 4 and 5 and which additional components are required. Providing additional transformations to the framework furthermore requires a transformation component model which wraps the transformation logic of sub-section 5.5.3 into re-usable modules and defines lifecycle aspects such as

configuration and deployment. Moreover, security concerns of executing foreign components have to be considered.

6.3 Related Work and Discourse

Considering current IWC solutions like those presented in 2.2, no explicitly extensible solution has been encountered yet. Hence, this section provides a brief overview and discourse on non IWC-specific extensibility approaches.

6.3.1 OSGi

Increasingly gaining popularity over the past few years, the OSGi specifications define a dynamic component system. The specification's core is formed by the OSGi Framework which defines components called *bundles* and the handling of these bundles by layers such as the Services, Life Cycle or Modules layer. Bundles are OSGi components which are created independently by different developers and can be used in an OSGi compliant architecture. The Module layer describes code import aspects of a bundle and the Service layer handles deployment and communication aspects. [69]

As OSGi is focused on Java [69] whereas the proof of concept prototype developed along with this thesis is implemented in CoffeeScript, which compiles to JavaScript, only general concepts can be considered.

The OSGi Module layer details bundle configuration, dependencies and dynamic imports. The extensive configuration options are required as OSGi bundles represent various complex components of different forms and purposes [69]. In contrast to this, the required transformation extensions are uniform and well-defined (cf. 5.5.3) and therefore do not necessitate additional configuration information.

Another important aspect of the Module layer, dependencies and dynamic imports, [69] could be limitedly relevant. Although a transformation is atomic and does not depend on other transformations, the possibility to handle external dependencies on JavaScript libraries such as jQuery[44] would be beneficial. These dependencies could be defined and dynamically loaded as described in OSGi. However, in a client-side JavaS-

[44] http://jquery.com/

cript framework this is not directly possible due to the Same Origin Policy of Web browsers [43]. The framework, which runs in the origin of the mashup application, cannot load resources from arbitrary domains directly, i.e. using a standard XMLHttpRequest.

There are workarounds such as JSON-P [70], a W3C Working Draft called Cross-Origin Resource Sharing (CORS) [71] and the possibility to use the mashup application Web server as proxy server to fetch the requested resources. However, as all these variants require a server component, this would violate the idea of an independently useable client-side framework and will hence not be pursued.

On the Services layer, OSGi defines *service interfaces* which the bundles can use to register their services with the *Framework service registry*. This is achieved by creating a service object which implements the service interface and registers the service. [69]

Again, the specification aims at supporting a wide range of different bundles whereas in the IWC and mediation context only one specific type of functionality is required. The basic idea of abstracting functionality behind a well-defined interface, creating an object implementing this interface and using a registry component to register the service provided will be employed however, simplified in that only one interface which transformation developers can implement will be defined.

The optional OSGi Security layer specifies a "fine-grained controlled environment" for OSGi compliant applications. It consists of two aspects: Code Authentication and Permissions. Code Authentication, i.e. confirming that the code provided by a bundle originates in a specific source, is achieved by digital signing of the jar files containing the bundles and verifying the signing chains. There are two authentication options: location and signer. The second aspect are Permissions which control access, including read, write, delete etc., to specific resources such as files and bundle properties. These permissions are filter-based and follow a whitelisting approach. [69]

All these high level security concerns are reliant on the security capabilities of the underlying Java VM. As JavaScript does not provide comparable security features such as protected or private accessibility, plugin security in the context of a client-side IWC framework has to focus on achieving the best level of security possible within the constraints of the JavaScript environment. Therefore, the ideas of the OSGi Security layer will not be further pursued.

6.3.2 JavaScript framework plugins

In the preceding sub-section it became clear that approaches based on the capabilities of other runtime environments such as the Java VM cannot be easily adopted into the setting of a JavaScript engine running inside a browser. Hence, this sub-section presents a brief summary of how JavaScript frameworks such as jQuery support extensibility.

JQuery is a popular JavaScript library which provides simplified DOM traversal and event handling abstracting differences between several browsers. The core functionality of the framework is provided by loading the jQuery library which defines the global object "window.jQuery" which is usually accessed using the abbreviated form "$". This is similar to the "window.IWC" object presented in 4.5.1. Extending jQuery by adding a plugin is achieved by either defining "jQuery.fn.myPlugin" or calling "jQuery.fn.extend()" for convenience. Taking a closer look on "jQuery.fn" one finds that it is a shortcut for "jQuery.prototype", so plugins are simply added to the prototype of the jQuery object. Thereafter, they can be called directly using the jQuery object or its shortcut "$". [72]

Considering the jQuery plugin approach, it has to be noted that although this approach is particularly tailored for a JavaScript environment, it aims at a different category of framework extensions. The jQuery plugins are extending the public functionality and thus the external API of the framework whereas the transformation extensions required for the IWC prototype extend internal functionality. The same technical method, i.e. extending the prototype of a particular object, e.g. "IWC.Communication.MediationPlugins", could be employed for this purpose nonetheless. Yet its advantage – the availability of the extended functionality on the external framework interface – could not be benefited from as the approach presented in 3 requires a transparent communication without explicitly discovering and calling transformations. Moreover, when identifying the transformations available the framework would have to iterate over the methods defined on the particular plugins object using JavaScript reflection capabilities which can be cumbersome.

6.4 Implementation

The brief analysis of existing extensibility approaches in 6.3 indicated that it is necessary to find a solution which is simpler, less general and more mediation specific than OSGi and which takes into account the limitations of the JavaScript runtime environment. Therefore, in this section the prerequisites, necessary changes to the framework architecture as of chapter 5 and relevant considerations of implementing a plugin architecture for extending the existing transformation capabilities will be presented.

6.4.1 Plugin interface

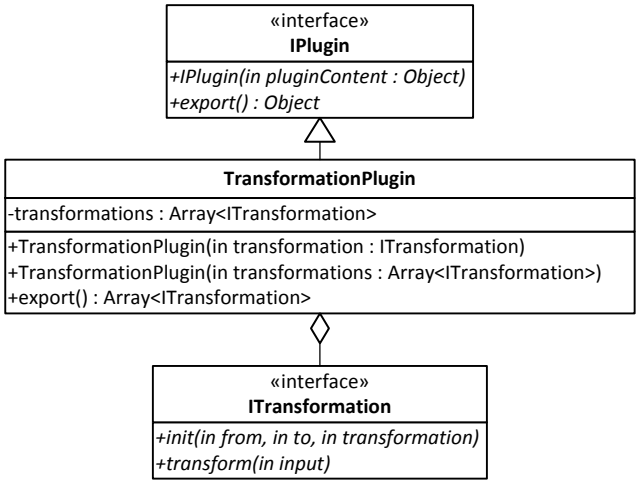

Figure 6.1 Plugin interface

In Figure 6.1 a simple plugin interface, "IPlugin", is proposed. It contains only one "export" method which exposes an object representing the contained functionality to the caller. This general interface allows for future extensibility as for example further mediation functionality could be provided using the same plugin interface. In the context of this thesis, however, there is only one implementation of IPlugin: the "TransformationPlugin" class. Instances of this class are employed to provide sets of transformations to the framework. It therefore aggregates "ITransformation" implementation instances and its export method returns an Array object containing the transformations provided by the plugin. The usage of one plugin for a set of transformations instead of one plugin per transformation allows grouping related transformations such

as transformation and inverse transformation or a set of transformations of the same subject in one plugin.

6.4.2 Extending the communication layer architecture

In the previous sub-section, the plugin interface was defined. This sub-section describes the necessary alterations of the communication layer architecture presented in 5 and depicted in Figure A.2 to accommodate the plugin provided extensibility.

As shown in Figure 6.2, a plugin registry component is added to the communication layer. The plugin registry aggregates instances implementing the IPlugin interface, currently only transformation plugins are implemented. Plugins are registered with the plugin registry by calling its "register" method. In this method, the registry calls the export function of the plugin and inspects the plugin type to decide how to treat the exported object.

Upon registration of a transformation plugin, the export method yields an Array object containing implementations of the ITransformation interface. Hence, the plugin registry will iterate over this Array and register each transformation with the transformation registry. The transformations newly added this way are then available to be used by the Mediator as explained in 5.

The architecture is inspired by the OSGi Service layer presented in 6.3.1. The implementations of the IPlugin interface are similar to the service interfaces and the plugin registry has similar responsibilities like the OSGi framework service registry.

This architecture can be easily extended towards a lazy-loading behaviour of the plugin provided functionality. To achieve this, the proxy pattern [38] can be employed: for example, the transformation plugins have to be changed towards exporting an Array of lightweight transformation proxies which store only basic information like the from and to properties. These proxies can be used in the planning algorithm reducing memory usage. If the transformation is required eventually, the transformation proxy delegates the transform call to the plugin which either provides the proxy with the necessary information to perform the transformation or performs it on behalf of the proxy and returns the results.

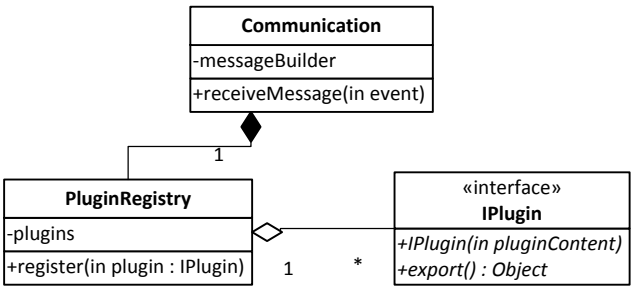

Figure 6.2 Communication layer architecture extended by plugins

6.4.3 Plugin loading and deployment

The preceding two sub-sections detailed the interface which plugins have to implement and the extension of the communication layer architecture required to incorporate the plugin-provided functionality. This sub-section describes, how plugins developed in accordance with these principles can be loaded and deployed into the framework.

There are two basic options to provide plugins to the framework:

- Static JavaScript library
- Dynamic plugin

The first option enables Mashup Platform Providers to use a set of trusted plugins which are well-known to them and which are therefore implicitly made available to all widgets using the IWC framework. To load these plugins, they should be included using the HTML script tag into the mashup application. In order to reduce page loading latency, it is recommendable to use a single JavaScript file containing all desired plugins instead of including one file per plugin.

This method is a static, server-side approach in that the set of plugins has to be known and available at the time of the initial HTTP request to the mashup site. There is a limited degree of dynamics as the Web server can dynamically create the mashup site itself at request time using server side includes (SSI)[45], common gateway interface

[45] http://httpd.apache.org/docs/2.0/howto/ssi.html

(CGI)[46] based frameworks such as C#.NET MVC[47] or dedicated Web servers with support for a certain programming language such as Node.js.

From the client-side framework perspective, however, these plugins are static and have to be loaded upon initialization of the framework. The problem in this case is to avoid a race condition which could lead to the plugins attempting to register with the framework before the framework has loaded completely and thus losing the registration. Thus, an approach for successfully registering the plugins has to be found.

On the IWC server side, i.e. the parts of the mashup which reside in the mashup's domain as opposed to the widgets, which are encapsulated in iframes with different origins thus forming the client side, the IWC framework consists of two parts:

- IWCServer.js and
- IWC.js

The former contains the server specific functionality whereas the latter is a library containing functionality shared between server and client side (widgets include IWC.js and the client-specific IWCClient.js). The IWC server script attaches its initialisation function to the load event of the window in order to set up the IWC stack as described in 4.5.1. Obviously, a script containing the plugins to be loaded into the framework cannot register them by calling a method of an object of the IWC stack as this stack is not yet initialized. Postponing the plugin registration to the onload phase of the window could either, if simply assigning to window.onload, overwrite the initialization of the server itself or otherwise complicate the onload handler.

Therefore, IWC.js provides a registerPlugins function which can be called by plugin scripts passing an array of the plugins to register. These plugins are simply stored in a temporary array which the IWC server processes upon initialization. This approach influences the order of script inclusion in that IWC.js has to be included before – in terms of HTML script tags above – the plugin scripts. However, this order is required in any case as plugins provide additional functionality by using or extending classes defined in the IWC core library. In particular, transformation plugins contain instances of the XSL transformation or algorithmic transformation classes defined in IWC.js if one wants to avoid reduplication of these class definitions in the plugin scripts, which

[46] http://tools.ietf.org/html/rfc3875
[47] http://www.asp.net/mvc

seems highly unrecommendable due to the negative impact on maintainability if new versions of the framework require changes to these classes.

The second possibility to load plugins is dynamically provided plugins. These plugins are either developed or at least included by Widget Developers and registered at runtime, when the widget is added into the mashup. In order to perform the plugin registration, the existing communication infrastructure of the IWC framework is utilized. These plugins do not necessarily require to be available on the mashup Web server at the time of the HTTP request if the widgets of the mashup are retrieved using a separate widget repository. This bears both advantages and disadvantages.

On the one hand, it provides an increased flexibility and less effort for the Mashup Platform Provider. For example, Widget Developers can provide the particular transformations they require along with their widgets. Moreover, as the plugins are initialized on the widget side – and so are the contained transformations – it becomes possible to load resources such as XSLT files from the Web server of the widget, which would not be possible without an extension of the mashup Web server (e.g. JSONP, CORS, proxy etc.) as indicated in 6.3.1.

On the other hand, the Mashup Platform Provider can exercise less control over the mashup. In contrast to the widgets, which run inside iframes with different origins and are hence isolated from the mashup, the dynamic plugins are loaded from the widgets into the framework to be run inside the origin of the mashup which poses a certain security threat as it lacks the isolation aspect of the widgets. A malicious plugin could access or manipulate any JavaScript object in the mashup origin.

Therefore, Mashup Platform Providers deciding to employ this IWC solution have to choose whether to permit or prohibit dynamic plugins. The prototype implements them as an opt-in possibility so as to allow Mashup Platform Providers to enable their usage if they consider the increased degree of flexibility more important than the security concerns or possess means of controlling them, e.g. by explicitly controlling the widgets which are used in their mashups.

6.4.4 Security concerns

As detailed above, dynamic plugins bear a certain security threat to the framework. In particular, contained algorithmic transformations (cf. 5.5.3) can perform malicious

code. The current implementation of algorithmic transformations passes the contained JavaScript source code to the built-in *eval* function, expecting a function "process(input)" to be declared inside and subsequently calling this function. The interpreted code could, for instance, directly access parts of the framework and manipulate the communication of other widgets. In order to avoid this, there are, apart from the obviously not feasible idea of automatic code inspection, two options: sandboxing the plugin code or running the plugin code in the context of the providing widget.

A basic integrated sandboxing possibility is the "strict mode" of ECMAScript. Enabling strict mode within the transform function of algorithmic transformations can restrict access to global variables such as the IWC object. However, strict mode also influences the eval function in that "Strict mode eval code cannot instantiate variables or functions in the variable environment of the caller to eval. " [55] This renders the eval-based definition of a function "process(input)" for algorithmic transformations described in 5.5.3 unfeasible.

Another variant of employing the strict mode to sandbox dynamically provided algorithmic transformation code was investigated: Switching the implementation of the transformation code storage from a string, which requires the eval function to be parsed, towards accommodating a real "Function" object. The Function object could then be invoked directly, without having to use eval and the surrounding context, i.e. the transform function, could enable strict mode to sandbox the foreign code. Unfortunately, this idea cannot be implemented due to another security restriction: In order to have an algorithmic transformation with a Function object available at the framework side, the plugin containing it has to be transported using postMessage. PostMessage creates a "structured clone" of the object passed to it as message [50]. According to subsection 2.8.5 of the HTML5 specification, however, creating a "structured clone […] [i]f input is another native object type (e.g. […] Function)" will "[t]hrow a DataCloneError exception and abort the overall structured clone algorithm" [73].

As it is not possible to transfer the function itself, the string representation variant introduced in 5.5.3 remains the only option, although requiring eval. Therefore, sandboxing with native JavaScript primitives cannot be achieved.

Another idea is sandboxing in JavaScript with additional tools. An interesting project, which has been used in the mashup context for OpenSocial gadget containers such as

in Apache Shinding[48] or in iGoogle[49], is Google Caja[50]. It secures HTML, CSS and JavaScript components by turning it into a "Caja module […] represented as a single JavaScript module function that can be run within a Caja container". This approach however, requires a Caja server to run. [74]

Due to the nature of JavaScript, sandboxing without an additional server which parses the original code and compiles it into a secure proxy code executed on behalf of the original, seems not feasible. As an additional server component would significantly alter the client-side IWC approach presented in this thesis for the reason of security, which is not a focused IWC dimension [22] in this approach, securing plugins by sandboxing will not be pursued.

The second possibility, running the plugin code within the widget context bears the advantage of isolating it from the framework context and thus avoiding disclosing framework parts. However, this would require extensive additional communication between the plugin on the widget side and the framework on the mashup application side when for example mediation requires the provided transformation. In contrast to the communication on top level, i.e. between widgets, this intra system communication would require an RPC-like model with acknowledgements and re-sending mechanisms to ensure the successful execution of the transformation. Moreover, messages of foreign widgets would be forwarded to the widget running the transformation. Consequently, this possibility is not attempted to integrate into the framework.

The above brief analysis revealed that, unless there are changes to JavaScript itself, security for dynamic plugins cannot be achieved without significant modifications to the IWC framework. However, it has to be argued that security is no main concern of this thesis. The prototype implementation is intended as a proof of concept for semantic interoperability in a client-side IWC framework with limited user involvement (cf. to IWC dimensions in [22]) and a lean, easy to use interface for Widget Developers. Other IWC projects are focussed on security of communication, such as Smash [48] and OMOS [49].

Moreover, dynamic plugins are but one of two methods implemented for plugin loading bearing a rather experimental character and hence have to be opted in by the Mashup Platform Provider. The decision in favour of or against them shall be taken

[48] http://shindig.apache.org/
[49] http://www.google.de/ig
[50] http://code.google.com/p/google-caja/

depending on the intended usage, i.e. depending on whether the mashup application only employs trusted widgets or allows for free choice. Static plugins still provide a stable base for extending the framework being subject to the same reviewing possibilities like any component or framework the Mashup Platform Provider decides to employ.

6.5 Summary

This chapter detailed how to modify the existing Publish/Subscribe and mediation architecture of the framework in order to incorporate extensibility. To achieve this, a plugin-based concept has been elaborated, considering related approaches, which defines a plugin interface and a plugin registry. The interface can be implemented and used to provide re-usable extensions to the framework, which has been demonstrated for transformations as declared in 5.5.3. Two methods of plugin loading – static and dynamic – have been presented and advantages and security concerns have been explained. As a result, this thesis advocates the opt-in of dynamic plugins in trusted environments. In Figure A.3 the final state of the IWC prototype framework can be seen. Again, this UML diagram has to be considered in conjunction with Figure A.1 and Figure A.2 as only new components and components which have been changed are depicted.

7 Evaluation

In order to assess the presented IWC approach – a client-side IWC framework supporting semantic interoperability and the guidelines of its intended usage – it has been analysed according to the requirements compiled in 2.1. Table 7.1 presents the results of this evaluation.

Requirement	Evaluation
Communication	+
Rich set of provided interactive components	+ +
Simplicity	+ +
Rich set of compatible components with minimum effort	+
Independence of the development process	+ +
Lean communication interface	+
+ + excellent, + good	

Table 7.1 Evaluation of the approach presented in this thesis

Communication

Regarding Communication, the approach presented is rated "good" because it offers the ability to have the widgets in a mashup communicate, however, like in 2.2.2, the rating contrasts slightly between communication frameworks offering a wider range of messaging patterns and those purpose-built for a certain aspect of communication – semantic interoperability in this particular case – and therefore only featuring one communication model – client-side topic- and type-based Publish/Subscribe [27].

Rich set of provided interactive components

With the IWC approach presented in this thesis being a modified Widget-developer-defined interoperability approach, the "Rich set of provided interactive components" is evaluated with "excellent". Widget Developers know best which information to share and which information to receive in their widgets. Following the presented approach, they are enabled to use IWC and define interoperability capabilities of their widgets which has the potential of creating rich sets of widgets which can communicate and may hence be provided to the End-User as interactive components.

Simplicity

From an End-User's perspective, the proposed IWC approach works "out-of-the-box" as widgets start communicating based on Publish/Subscribe and mediation, immediately when added to the UI mashup without requiring further user interaction. Therefore, the approach has to be considered simple for End-Users and receives the "excellent" rating.

Rich set of compatible components with minimum effort

Considering the two components of this requirement, the first part a "Rich set of compatible components" can be achieved with the presented IWC approach as already reasoned in the corresponding End-User requirement; the partial rating is "excellent". Its second component, the "minimum effort" constraint, depends on how many static plugins (cf. 6.4.3) Mashup Platform Providers are employing, how they are retrieving and reviewing them and which infrastructure, which community builds around plugin development. For instance, popular plugins could be integrated into the framework core and would then be provided with the framework without additional effort. The resulting partial rating for the effort component is "good" to "sufficient" yielding the overall rating of "good" for this requirement. This is a slight deterioration compared to the "excellent" rating of the Widget-developer-defined category.

Independence of the development process

With a full realization of the proposed IWC approach, the high level of independence in the process of widget development of the Widget-developer-defined category can be maintained. Knowledge of the existence or internal structure (e.g. events, methods,

parameters etc.) of foreign widgets is not required. The employment of groundings, as proposed in 5.6, could be considered a light degree of dependency, however on the framework not on foreign widgets, and could help to increase independency from other widgets in earlier phases of the realization of the presented approach when the set of available transformations is still small. However, the evaluation is based on the assumption of an extensive realization of the approach. Consequently, the rating "excellent" is achieved.

Lean communication interface

One of the objectives of this thesis was to elaborate a Widget-developer-defined interoperability IWC approach which, in contrast to existing solutions of this category as analysed in 2.2.2, features a "lean communication interface". This could be achieved up to a "good" degree. The basic operations of the topic-based Publish/Subscribe paradigm have been extended towards a topic- and type-based implementation [27], yet the amount of communication operations remains two. One additional operation, the plugin registration, was introduced which is, however, very similar in its usage to the Publish/Subscribe subscription.

The only additional concept introduced is the concept of transformations (cf. 5.5.3); however, great care was taken in order to keep their implementation as simple as possible. With the framework already providing prepared transformation classes, transformations can be implemented by either programming a single process(input) function or providing an XSLT file, where XSLT transformations is a representative for any transformation specific language, support for which can be easily added to the framework.

Taking into account the additional mediation aspect introduced by transformations, the "lean communication interface" of the approach presented is rated with "good".

Synopsis

The above evaluation reveals that the presented IWC approach could maintain the high ratings of Widget-developer-defined interoperability solutions, with the exception of a slightly increased Mashup Platform Provider effort, while substantially improving results regarding a "Lean communication interface" from "poor" to "good".

8 Conclusion and future research

The research presented in this thesis has produced an approach for improving semantic interoperability in IWC-enabled choreographed UI mashups. The basic idea of this approach is, on the one hand side, to follow a hybrid topic- and type-based Publish/Subscribe paradigm and combine IWC with mediation and, on the other hand side, to support this approach by implementing a client-side framework.

The IWC approach presented is characterized by recommending a limited set of IWC subjects and the corresponding widget developer guideline proposed in 3.4 as well as by the proposed declaration of data formats described in 5.5.1. In antithesis to the existing interoperability approaches in IWC analysed in 2, this thesis particularly focused on ad-hoc built, choreographed UI mashups, and a less complex, easier to use IWC interface in order to encourage utilization of IWC. Moreover, a very low level of user involvement has been achieved, especially taking into account UI mashups as an approach for End-User Development focussing users without technical experience.

Additionally, this thesis revealed that the technical requirements for semantic interoperability in IWC can be achieved without an additional server component by a simple, client-side JavaScript framework. What is more, in contrast to other approaches in this field, this client-side framework forms an independent, modular component which can be easily used in arbitrary UI mashups regardless of the mashup engine, widget repository or widget specification employed.

The framework which was presented in this thesis can be implemented from scratch easily by following the explanations of chapters 4, 5 and 6 step by step. Moreover, it has been designed for extensibility in that it provides a plugin architecture and prepared classes that can be used to extend its mediation capabilities by adding extra transformations to the mediator.

Related future research could confine conducting a survey to identify further default IWC subjects, refining those already presented and integrating support for commonly used representations of these default subjects into the framework core, discovering possibilities to use transformation domain specific languages other than XSLT or even defining an entirely new transformation language and investigating more complex

mediation. For instance, aggregation of several incoming messages into one single outgoing notification or even more complex, rule-based messaging patterns could be integrated into the mediator component (cf. message and process mediation [16]). Another interesting future research aspect is the selection of plugins if several providing the same functionality are available based, for example, on performance measurements and faults. Regarding plugins, another research topic is the lazy loading behaviour described in 6.4.2 and the security of foreign source code in client-side JavaScript contexts without additional server components as indicated in 6.4.4.

Also, in future a certain level of user control over the communication could be introduced into the framework, however concerned with keeping the amount of user interaction for IWC purposes low and maintaining a good usability. For instance, a model similar to the dynamic configuration of modern software firewalls could be employed, asking upon subscriptions whether to permit or prohibit the subscription and additionally offering the possibility to permit or prohibit all subsequent subscriptions / subscriptions of this widget / subscriptions on this subject etc.

Bibliography

[1] S. Wilson, F. Daniel, U. Jugel, and S. Soi, "Orchestrated user interface mashups using w3c widgets," in *Current Trends in Web Engineering*, A. Harth and N. Koch, Eds. Springer, 2011, pp. 49–61.

[2] M. Stecca and M. Maresca, "An execution platform for event driven mashups," *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services - iiWAS '09*, pp. 33–40, 2009.

[3] S. Wilson, "Design challenges for user-interface mashups: user control and usability in inter-widget communications," 2012. [Online]. Available: http://scottbw.wordpress.com/2012/03/07/design-challenges-for-user-interface-mashups-user-control-and-usability-in-inter-widget-communications/. [Accessed: 29-May-2012].

[4] O. Chudnovskyy, S. Müller, and M. Gaedke, "Extending Web Standards-based Widgets towards Inter-Widget Communication," in *Proceedings of the 4th International Workshop on Lightweight Integration on the Web (ComposableWeb 2012)*, 2012.

[5] E. Isaksson and M. Palmer, "Usability and inter-widget communication in PLEs," in *Proceedings of the 3rd Workshop on Mashup Personal Learning Environments*, 2010.

[6] S. Siniša, D. Škvorc, and D. Skrobo, "Widget-Oriented Consumer Programming," *AUTOMATIKA: Journal for Control, Measurement, Electronics, Computing and Communications*, vol. 50, pp. 252–264, 2009.

[7] I. Zuzak, M. Ivankovic, and I. Budiselic, "A Classification Framework for Web Browser Cross-Context Communication," *Arxiv preprint arXiv:1108.4770*, 2011.

[8] OpenAjax Alliance, "OpenAjax Hub 2.0 Specification," 2009. [Online]. Available: http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification. [Accessed: 10-May-2012].

[9] M. Cáceres, "Widget Packaging and XML Configuration," *W3C Recommendation 27 September 2011*, 2011. [Online]. Available: http://www.w3.org/TR/widgets/. [Accessed: 19-Aug-2012].

[10] M. Cáceres, "Widgets 1.0: The Widget Landscape (Q1 2008)," *W3C Working Draft 14 April 2008*, 2008. [Online]. Available: http://www.w3.org/TR/widgets-land/. [Accessed: 03-May-2012].

[11] M. Cáceres, "Widget Interface," *W3C Proposed Recommendation 22 May 2012*, 2012. [Online]. Available: http://www.w3.org/TR/widgets-apis/. [Accessed: 19-Aug-2012].

[12] C. Fischer, O. Chudnovskyy, and M. Gaedke, "Unterstützung der Endbenutzer-Entwicklung mittels Inter- Widget-Kommunikation und User Interface Mashups," in *Chemnitzer Informatik-Berichte*, 2012, pp. 15–26.

[13] F. Daniel, S. Soi, S. Tranquillini, F. Casati, et al., "Distributed Orchestration of User Interfaces," *Information Systems*, vol. 37, no. 6, pp. 539–556, 2011.

[14] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, vol. 25, no. 3, pp. 38–49, Mar. 1992.

[15] G. Wiederhold and M. Genesereth, "The conceptual basis for mediation services," *IEEE Expert*, vol. 12, no. 5, pp. 38–47, Sep. 1997.

[16] C. Wu and E. Chang, "An Analysis of Web Services Mediation Architecture and Pattern in Synapse," in *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, 2007, pp. 1001–1006.

[17] D. Beneventano, S. El Haoum, and D. Montanari, "Mapping of heterogeneous schemata, business structures, and terminologies," in *Proceedings of the 18th International Workshop on Database and Expert Systems Applications (DEXA 2007)*, 2007, pp. 412–418.

[18] D. Beneventano, M. Orsini, L. Po, and S. Sorrentino, "The MOMIS - STASIS approach for Ontology-based Data Integration," in *Proceedings of ISDSI 2009*, 2009.

[19] J. Howe, *Crowdsourcing: Why the Power of the Crowd Is Driving the Future of Business*. Crown Business, 2009.

[20] O. Chudnovskyy, H. Chang, T. Nestler, M. Gaedke, et al., "End-user-oriented telco mashups," in *Proceedings of the 21st international conference companion on World Wide Web - WWW '12 Companion*, 2012, pp. 235–238.

[21] M. M. Burnett and C. Scaffidi, "End-User Development," in *Encyclopedia of Human-Computer Interaction*, M. Soegaard and R. F. Dam, Eds. Aarhus, Denmark: The Interaction-Design.org Foundation, 2011.

[22] I. Zuzak, "Inter-Widget Communication: Lecture series: Mash-up Personal Learning Environments." 2011.

[23] M. Kay, "XSL Transformations (XSLT) Version 2.0," *W3C Recommendation 23 January 2007*, 2007. [Online]. Available: http://www.w3.org/TR/xslt20/. [Accessed: 19-Aug-2012].

[24] C. Pautasso and E. Wilde, "Why is the Web Loosely Coupled ? A Multi-Faceted Metric for Service Design," in *Proceedings of the 18th International World Wide Web Conference (WWW2009)*, 2009.

[25] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Prentice Hall International, 2008.

[26] C. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.

[27] P. T. Eugster, P. a. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[28] I. Zuzak, M. Ivankovic, and I. Budiselic, "Cross-context Web Browser Communication with Unified Communication Models and Context Types," *Arxiv preprint arXiv:1108.4770*, pp. 690–696, 2011.

[29] B. Hoisl and H. Drachsler, "User-tailored Inter-Widget Communication-Extending the Shared Data Interface for the Apache Wookie Engine," in *Proceedings of the ICL2011*, 2010.

[30] Google Inc., "OpenSocial Core Gadget Specification 2.0.1," 2011. [Online]. Available: http://opensocial-resources.googlecode.com/svn/spec/2.0.1/Core-Gadget.xml. [Accessed: 31-May-2012].

[31] S. Pietschmann, C. Radeck, K. Meißner, and K. M. De, "Semantics-Based Discovery, Selection and Mediation for Presentation-Oriented Mashups," in *Mashups '11 Proceedings of the 5th International Workshop on Web APIs and Service Mashups*, 2011.

[32] A. Soylu, F. Wild, F. Mödritscher, P. Desmet, et al., "Mashups and Widget Orchestration," in *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, 2011, pp. 226–234.

[33] D. Lizcano, J. Soriano, M. Reyes, and J. J. Hierro, "EzWeb / FAST : Reporting on a Successful Mashup-based Solution for Developing and Deploying Composite Applications in the Upcoming ' Ubiquitous SOA ',," in *UBICOMM 2008*, 2008, no. section 2.

[34] V. Hoyer, F. Gilles, T. Janner, and K. Stanoevska-Slabeva, "SAP Research Roof-Top Marketplace: Putting a Face on Service-Oriented Architectures," in *Proceedings of the 2009 Congress on Services - I SERVICES '09*, 2009, pp. 107–114.

[35] M. Reyes Ureña, "EzWeb Developing Approach," 2010. [Online]. Available: http://forge.morfeo-project.org/wiki/images/3/32/DeveloperGuide.pdf. [Accessed: 19-Aug-2012].

[36] M. Bates, "Models of natural language understanding Colloquium Paper : Bates SpechnIone," in *Proceedings of the National Academy of Sciences of the United States of America*, 1995, vol. 92, no. October, pp. 9977–9982.

[37] S. Pietschmann, J. Waltsgott, and K. Meißner, "A Thin-Server Runtime Platform for Composite Web Applications," *2010 Fifth International Conference on Internet and Web Applications and Services*, pp. 390–395, 2010.

[38] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, vol. 206, no. 0. Amsterdam: Addison-Wesley, 1995.

[39] R. Fielding, J. Getty, J. Mogul, H. Frystyk, et al., "RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1," *IETF Request for Comments*, 1999. [Online]. Available: http://tools.ietf.org/html/rfc2616.

[40] Web Intents Task Force, "webintents.org," 2012. [Online]. Available: http://webintents.org/. [Accessed: 30-May-2012].

[41] G. Billock, J. Hawkins, and P. Kinlan, "Web Intents," *W3C Working Draft 26 June 2012*, 2012. [Online]. Available: http://www.w3.org/TR/web-intents/.

[42] G. Banavar, T. Chandra, R. Strom, and D. Sturman, "A case for message oriented middle- ware.," in *Proceedings of the 13th International Symposium on Distributed Computing*, 1999, pp. 1–18.

[43] W3C, "Same Origin Policy," *W3C Web Security Wiki*, 2010. [Online]. Available: http://www.w3.org/Security/wiki/Same_Origin_Policy. [Accessed: 31-Jul-2012].

[44] W3Schools, "Browser Statistics," 2012. [Online]. Available: http://www.w3schools.com/browsers/browsers_stats.asp. [Accessed: 31-Jul-2012].

[45] S. Hanna, E. C. R. Shin, D. Akhawe, A. Boehm, et al., "The emperor's new APIs: On the (in) secure usage of new client-side primitives," in *Proceedings of the IEEE Web 2.0 Security and Privacy Workshop (May, 2010)*, 2010.

[46] J. Cook, "Web Workers API Shim," 2009. [Online]. Available: http://html5-shims.googlecode.com/svn/trunk/demo/workers.html. [Accessed: 31-Jul-2012].

[47] OpenAjax Alliance, "OpenAjax Hub 2.0.7 Reference implementation," 2009. [Online]. Available: http://sourceforge.net/projects/openajaxallianc/. [Accessed: 09-Jul-2012].

[48] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, et al., "SMash : Secure Component Model for Cross-Domain Mashups on Unmodified Browsers," in *Proceedings of the 17th International Conference on World Wide Web*, 2008, pp. 544–535.

[49] S. Zarandioon, D. (Daphne) Yao, and V. Ganapathy, "OMOS: A Framework for Secure Communication in Mashup Applications," in *2008 Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 355–364.

[50] I. . Hickson, Ian (Google, "HTML5 Web Messaging," *W3C Candidate Recommendation 01 May 2012*, 2012. [Online]. Available: http://www.w3.org/TR/webmessaging/. [Accessed: 02-May-2012].

[51] Caniuse.com, "Compatibility tables: Cross-document messaging," 2012. [Online]. Available: http://caniuse.com/#feat=x-doc-messaging. [Accessed: 31-Jul-2012].

[52] JSON-RPC Working Group, "JSON-RPC 2.0 Specification," 2011. [Online]. Available: http://www.jsonrpc.org/specification. [Accessed: 19-Aug-2012].

[53] I. Zuzak, "Pmrpc implementation," 2010. [Online]. Available: https://github.com/izuzak/pmrpc. [Accessed: 19-Aug-2012].

[54] I. Zuzak, "Pmrpc discovery and publish-subscribe support + systematization of cross-context browser communication systems," 2010. [Online]. Available: http://ivanzuzak.info/2010/06/15/pmrpc-discovery-and-publish-subscribe-support-systematization-of-cross-context-browser-communication-systems.html. [Accessed: 12-Jul-2012].

[55] Ecma International, "ECMAScript Language Specification," no. June. 2011.

[56] D. Kaye, *Loosely Coupled: The Missing Pieces of Web Services*, 1st ed. RDS Press, 2003.

[57] M. E. Pierce, R. Singh, Z. Guo, S. Marru, et al., "Open community development for science gateways with apache rave," in *Proceedings of the 2011 ACM workshop on Gateway computing environments - GCE '11*, 2011, pp. 29–36.

[58] S. Goessner, "JsonT - Transforming Json," 2006. [Online]. Available: http://goessner.net/articles/jsont/. [Accessed: 19-Jul-2012].

[59] M. Motovilov, "How to use JXL to transform JSON data?," 2011. [Online]. Available: https://github.com/MaxMotovilov/adstream-js-frameworks/wiki/Jxl-usage. [Accessed: 19-Jul-2012].

[60] J. van Horn, "json2json," 2012. [Online]. Available: https://github.com/joelvh/json2json. [Accessed: 19-Jul-2012].

[61] B. Stein, "XSLTJSON: Transforming XML to JSON using XSLT." [Online]. Available: http://www.bramstein.com/projects/xsltjson/. [Accessed: 19-Jul-2012].

[62] N. Freed and N. Borenstein, "RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," *IETF Request for Comments*, 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2046.txt. [Accessed: 19-Aug-2012].

[63] ISO, "ISO 6709:2008 Standard representation of geographic point location by coordinates," vol. 2008. 2008.

[64] T. Sheldon, *Encyclopedia of Networking & Telecommunications*. Osborne McGraw-Hill, 2001.

[65] W3Schools, "XSLT Browsers," 2012. [Online]. Available: http://www.w3schools.com/xsl/xsl_browsers.asp. [Accessed: 21-Jul-2012].

[66] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, et al., "Document Object Model (DOM) Level 2 Core Specification," *W3C Recommendation 13 November, 2000*, 2000. [Online]. Available: http://www.w3.org/TR/DOM-Level-2-Core/. [Accessed: 19-Aug-2012].

[67] R. Bellman, *Dynamic Programming*. Dover Publications, 1972.

[68] D. L. Applegate, R. E. Bixby, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

[69] Osgi Alliance, "The OSGi Alliance OSGi Core Release 5," 2012. [Online]. Available: http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf. [Accessed: 19-Aug-2012].

[70] K. Simpson, "JSON-P." [Online]. Available: http://www.json-p.org/. [Accessed: 27-Jul-2012].

[71] A. van Kesteren, "Cross-Origin Resource Sharing," *W3C Working Draft 3 April 2012*, 2012. [Online]. Available: http://www.w3.org/TR/cors/. [Accessed: 19-Aug-2012].

[72] The jQuery Project, "jQuery Documentation: Plugins/Authoring," 2010. [Online]. Available: http://docs.jquery.com/Plugins/Authoring. [Accessed: 27-Jul-2012].

[73] I. Hickson, "HTML5: A vocabulary and associated APIs for HTML and XHTML," *W3C Working Draft 29 March 2012*, 2012. [Online]. Available: http://www.w3.org/TR/html5/. [Accessed: 02-May-2012].

[74] Google Inc., "Google Caja Wiki," 2010. [Online]. Available: http://code.google.com/p/google-caja/wiki/. [Accessed: 03-Aug-2012].

Appendix A

Framework architecture

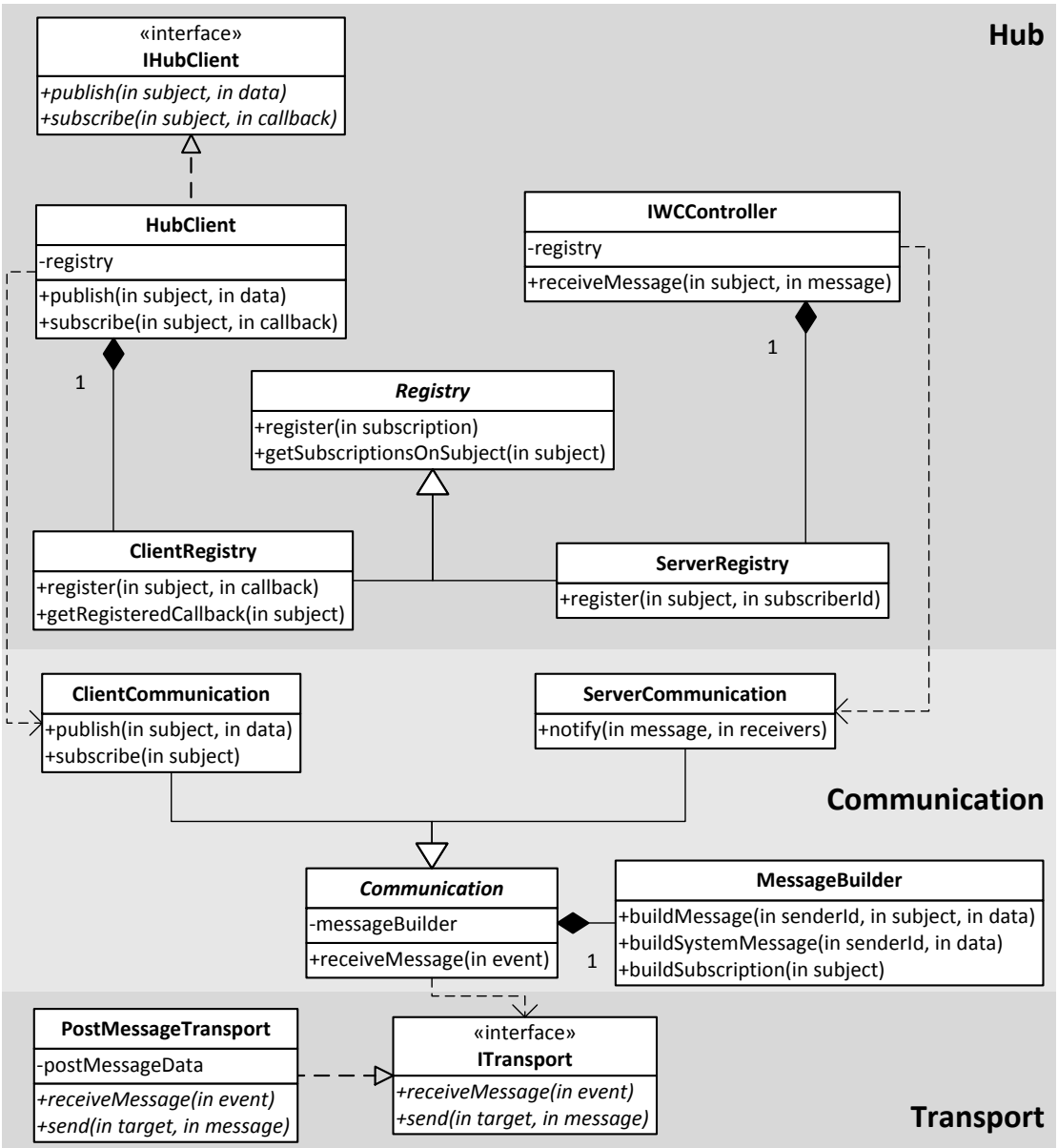

Figure A.1 Publish/Subscribe

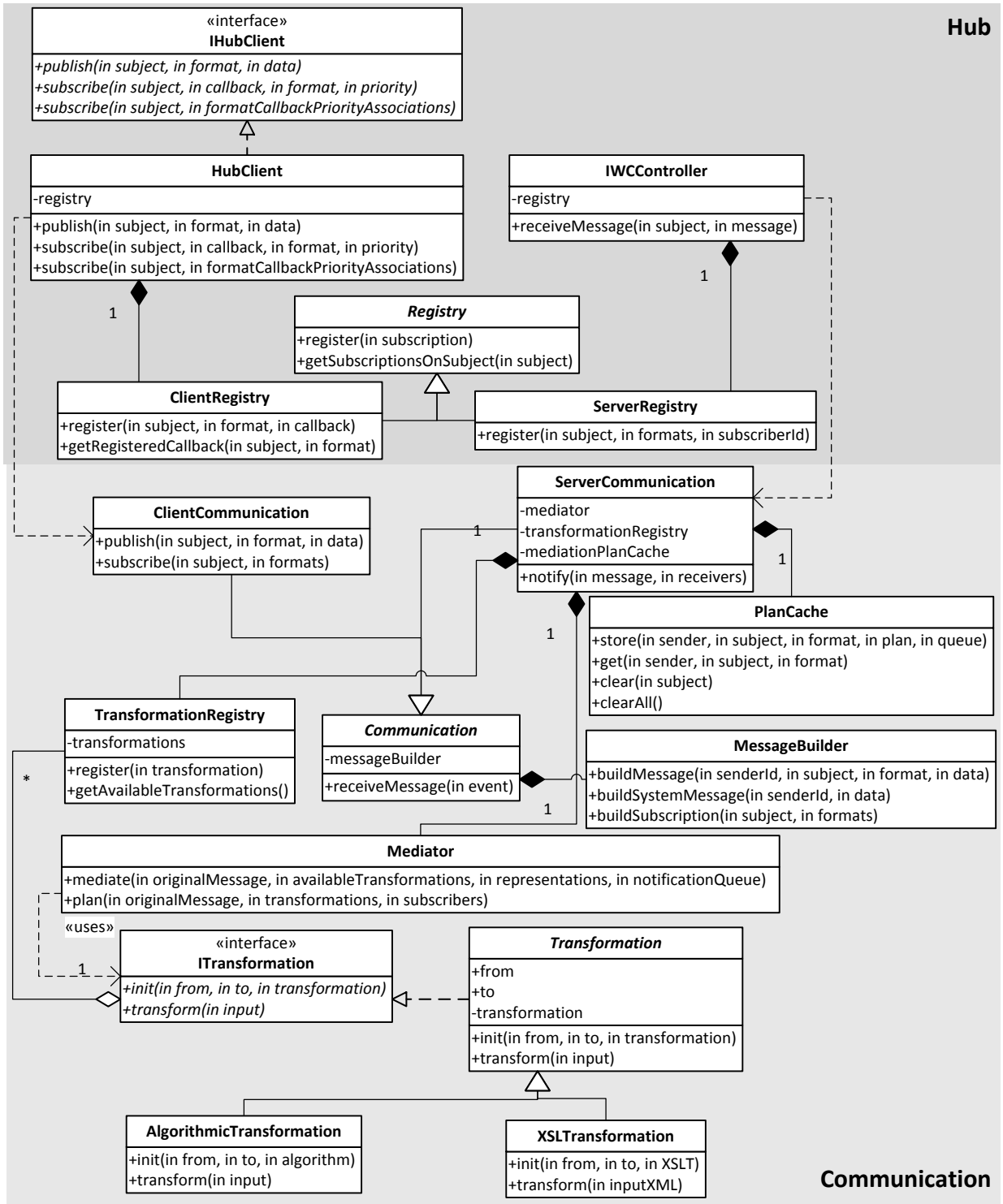

Figure A.2 Mediation

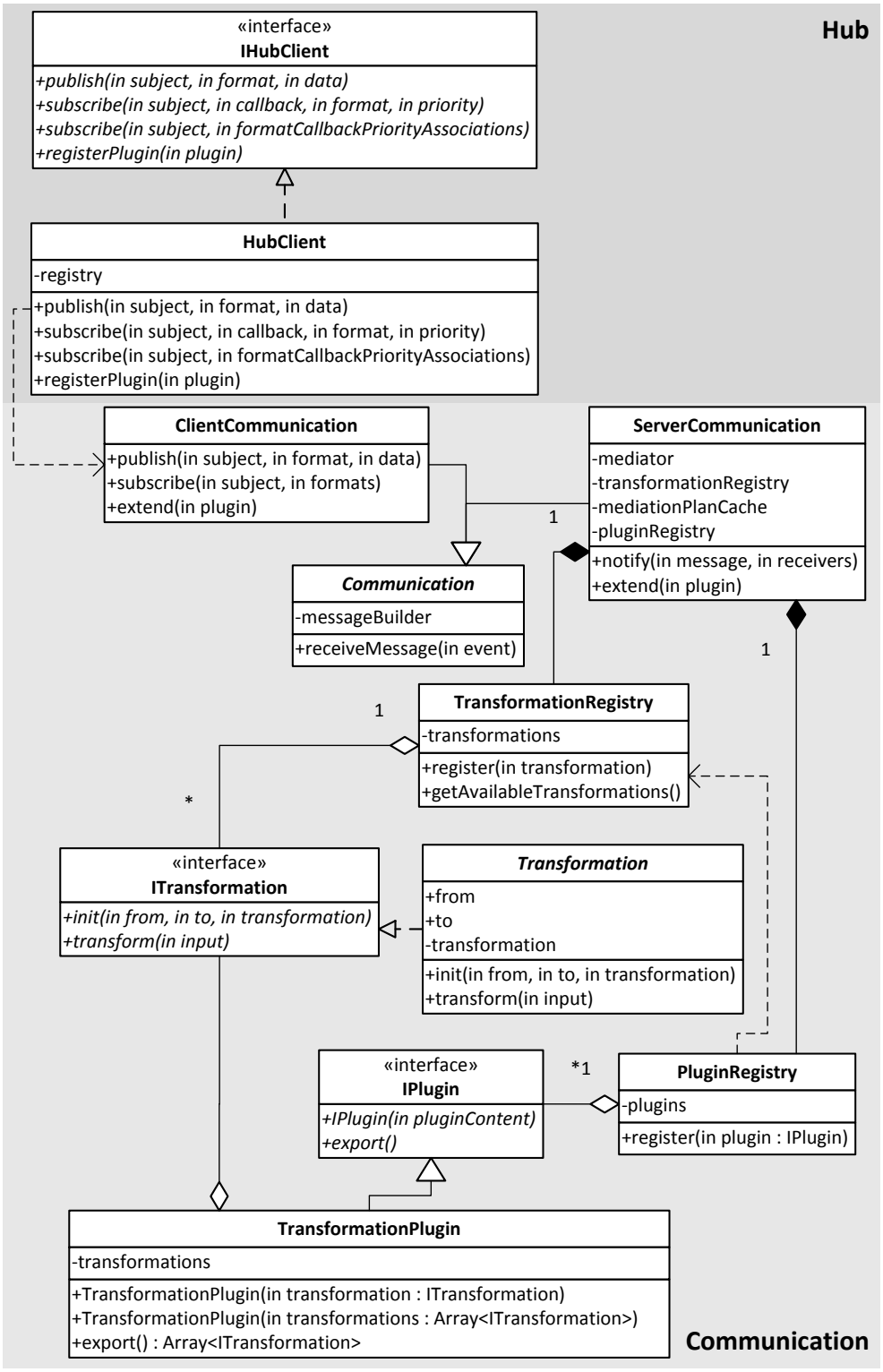

Figure A.3 Plugins

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Aufgabenstellung

zur

Abschlussarbeit
im Studiengang Master Data and Web Engineering

für

Herrn Sebastian Heil
geb. am 17. Mai 1988 in Zwickau

zum Thema

Supporting Semantic Interoperability in Inter-Widget-Communication-enabled User Interface Mashups

Betreuer/ Prüfer:	Prof. Dr. Martin Gaedke
Ausgabedatum:	18.07.2012
Abgabedatum:	27.12.2012
Tag der Abgabe:	

Unterschrift: .....

Prof. Dr. F. Hamker
Vorsitzender des Prüfungsausschusses

Aufgabenstellung

Supporting Semantic Interoperability in Inter-Widget-Communication-enabled User Interface Mashups

User Interface Mashups (UI Mashups) ist ein vielversprechender Ansatz zur Entwicklung von Webanwendungen auf Basis von wiederverwendbaren Komponenten (den sogenannten Widgets), der Endnutzer in den Mittelpunkt der Entwicklung stellt. Dabei spielt die Inter-Widget-Kommunikation eine wichtige Rolle – z.B. zur Definition von Datenflüssen im Mashup oder Widget-Zustandssynchronisation. Für eine effiziente Widget-Aggregation ist ein Ansatz erwünscht, der eine unabhängige und in Bezug auf Inter-Widget-Kommunikation kompatible Widget-Entwicklung ermöglicht. Im Rahmen dieser Arbeit soll ein Inter-Widget-Kommunikationsmechanismus konzipiert werden, welcher auf der einen Seite einfache Konfiguration des Nachrichtenaustauschs in UI Mashups ermöglicht und auf der anderen Seite Hilfsmittel zur Verbesserung der semantischen Interoperabilität zwischen unabhängig voneinander entwickelten Widgets zur Verfügung stellt.

Zuerst wird in der Arbeit eine gründliche Analyse durchgeführt, die bestehende Inter-Widget-Kommunikations-Ansätze in Bezug auf ihre Einbindungsmöglichkeiten und Sicherstellung der Interoperabilität evaluiert. Anschließend wird ein Ansatz zur Inter-Widget-Kommunikation entwickelt, der unter Einbeziehung von Vorgehensweisen und Konzepten aus anderen Domänen wie beispielsweise Web Intents die Kommunikation zwischen semantisch kompatiblen Widgets unterstützt. Darüber hinaus soll der zu entwickelnde Ansatz es ermöglichen, Funktionalität zum Erreichen von syntaktischer Kompatibilität von Inter-Widget-Kommunikationsnachrichten wiederzuverwenden und für zukünftig entwickelte Widgets bereitzustellen. Die entworfenen Konzepte sollen im Rahmen eines Prototyps entwickelt und evaluiert werden.

TECHNISCHE UNIVERSITÄT CHEMNITZ

Zentrales Prüfungsamt
(Anschrift: TU Chemnitz, 09107 Chemnitz)

Selbstständigkeitserklärung*

Name: Heil Vorname: Sebastian geb. am: 17.05.1988 Matr.-Nr.: 180623	**Bitte Ausfüllhinweise beachten:** 1. Nur Block- oder Maschinenschrift verwenden.

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: 27.08.2012 Unterschrift: Heil

* Diese Erklärung ist der eigenständig erstellten Arbeit als Anhang beizufügen.