



**TECHNISCHE  
UNIVERSITÄT  
DRESDEN**

---

**Faculty of Computer Science** Institute of System Architecture, Chair of Computer Networks

---

Minor Thesis

# **REAL-TIME COLLABORATION SUPPORT FOR JAVASCRIPT FRAMEWORKS**

Franz Josef Grüneberger  
Matrikel-Nr.: 3385643

Supervised by:

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

and:

Dipl.-Medieninf. Matthias Heinrich, Dr. Thomas Springer

Submitted on July 14, 2012





## AUFGABENSTELLUNG FÜR DIE BELEGARBEIT

Name, Vorname: Grüneberger, Franz Josef  
Studiengang: Informatik  
Matrikelnummer: 3385643  
Forschungsgebiet: Service and Cloud Computing  
Forschungsprojekt: -  
Thema: **Real-time Collaboration Support for JavaScript Frameworks**

### ZIELSTELLUNG

Nowadays globalization leads to widely dispersed talents working together towards common visions. Working together in teams, both cross-department and cross-company, is about creating, sharing, discussing and managing information. Furthermore today's competitive global economy forces for example shorter innovation cycles and a reduced time to market. In essence, efficiency is required in the whole value-added chain. Therefore efficient collaboration is indispensable.

While collaboration in the past was restricted to some standard services like email, filesharing or content management, in times of the financial downturn, where companies are faced with financial belt-tightening and financial restrictions on travel expenses become even more important, new technologies for global collaboration are important. There is a need for robust complementary browser-based tools supporting a new generation of internet-centric employees. An IDC report stated that collaborative applications were in great demand already in 2007 creating \$6.3 billion worldwide revenue. Since 1999, when the term "Web 2.0" was coined, which is associated with a richer web facilitating social media, user-generated content as well as interaction and collaboration, business applications have become increasingly web-oriented. While the web in its roots was limited to passive viewing of content created by others, it has moved towards an application platform for desktop-like applications within the last decade. The implementation of such web-based applications is usually realized by plug-in technologies like Adobe Flash or by means of pure HTML, JavaScript and AJAX. During the last years plugin-free implementations have become more popular, driven by the HTML5 movement, faster JavaScript engines, new browser features like key-value-stores and 3D-graphics, etc.

The extensive usage of JavaScript in today's web applications induces the need for frameworks supporting faster development, better reusability and maintainability. As Model-View-Controller (MVC) is a well-known design pattern for server-side application development, it becomes even more important on client-side leading to several prevalent JavaScript MVC frameworks. Therefore client-side JavaScript application frameworks will be of great importance for the development of future web-based business applications including collaborative ones.

The goal of the thesis is to analyze existing JavaScript frameworks with respect to their collaboration support. From the set of reviewed frameworks, one should be selected to design

and implement collaboration enhancements. The developed approach should be evaluated focusing on ease of development of collaborative applications.

## SCHWERPUNKTE

- Existing JavaScript frameworks shall be evaluated regarding their capabilities in terms of real-time collaboration support.
- At least one framework should be enhanced to ease the implementation of collaborative applications.
- Gains with respect to the development effort using the collaboration-enabled framework shall be emphasized.

*Betreuer:* Dipl. Medieninf. Matthias Heinrich (SAP Research)  
Dr. Thomas Springer

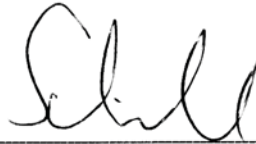
*Verantwortlicher Hochschullehrer:* Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

*Institut:* Institut für Systemarchitektur

*Lehrstuhl:* Rechnernetze

*Beginn am:* 15. Januar 2012

*Einzureichen am:* 14. Juli 2012



*Unterschrift des verantwortlichen Hochschullehrers*

Verteiler: 1 x HSL, 1 x Betreuer, 1 x Student

## **CONFIRMATION**

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, July 14, 2012



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation and Objectives . . . . .	2
1.2	Introductory Application Example . . . . .	3
1.3	Structure of the Thesis . . . . .	3
<b>2</b>	<b>The Web as Application Platform</b>	<b>5</b>
2.1	Evolution of the Web . . . . .	5
2.1.1	The Web as Document Environment . . . . .	6
2.1.2	The Web as Application Environment . . . . .	6
2.1.3	The Web as Desktop-Like Application Environment . . . . .	7
2.2	Classification of Web Applications . . . . .	8
2.2.1	Classification Based on Application Domain . . . . .	8
2.2.2	Classification Based on Implementation Technology . . . . .	9
2.2.3	Classification of Current Web Applications . . . . .	10
2.3	Development Approaches for Collaborative Web Applications . . . . .	11
2.3.1	Operational Transformation Libraries . . . . .	11
2.3.2	Collaborative Widget Libraries . . . . .	12
2.3.3	Transparent Adaptation . . . . .	13
2.3.4	Code Injection . . . . .	14
2.4	Conclusion . . . . .	14

<b>3</b>	<b>Conception of a Collaboration Extension for JavaScript Frameworks</b>	<b>17</b>
3.1	Requirements for a Collaboration Extension . . . . .	18
3.2	Architectural Patterns of JavaScript Frameworks . . . . .	19
3.2.1	Model-View-Controller (MVC) . . . . .	20
3.2.2	Presentation Model (PM) . . . . .	20
3.3	Classification of JavaScript Frameworks . . . . .	21
3.3.1	Architectural Pattern . . . . .	22
3.3.2	Programming Language . . . . .	22
3.3.3	Framework Integration . . . . .	23
3.3.4	User Interface Specification . . . . .	23
3.3.5	User Interface Update . . . . .	24
3.3.6	Data Model Structure . . . . .	26
3.3.7	Listener Registration . . . . .	27
3.4	Conceptual Architecture of a Collaboration Extension . . . . .	29
3.4.1	Architecture Overview . . . . .	29
3.4.2	Synchronization Service . . . . .	30
3.4.3	Collaboration Adapter . . . . .	31
3.4.4	Discussion of Requirements . . . . .	33
3.5	Conclusion . . . . .	34
<b>4</b>	<b>Prototypical Implementation of Collaboration Extensions</b>	<b>35</b>
4.1	Framework Selection . . . . .	35
4.1.1	Framework Selection Characteristics . . . . .	35
4.1.2	Survey of Existing Frameworks . . . . .	36
4.1.3	Selected Frameworks . . . . .	38
4.2	Implementation Structure . . . . .	39
4.3	SAP Gravity . . . . .	39
4.3.1	Architecture . . . . .	40
4.3.2	Data Model . . . . .	40



4.3.3	Primitive and Complex Operations . . . . .	41
4.3.4	Client API . . . . .	42
4.3.5	Synchronization Workflow . . . . .	44
4.4	Shared JSON . . . . .	46
4.4.1	JavaScript Object Notation . . . . .	46
4.4.2	JSONPath . . . . .	46
4.4.3	JSON to Gravity Mapping . . . . .	47
4.4.4	Client API . . . . .	50
4.4.5	Shared JSON Events . . . . .	53
4.4.6	Synchronization Workflow . . . . .	54
4.5	SAPUI5 Collaboration Adapter . . . . .	55
4.5.1	Implementation Structure . . . . .	55
4.5.2	Application Integration . . . . .	57
4.5.3	Synchronization Workflow . . . . .	58
4.6	Knockout.js Collaboration Adapter . . . . .	59
4.6.1	Knockout.js View Model Structure . . . . .	59
4.6.2	Knockout.js Synchronization Extension . . . . .	61
4.6.3	Annotations . . . . .	62
4.6.4	Property Exclusion Mechanism . . . . .	65
4.6.5	Application Integration . . . . .	66
4.6.6	Synchronization Workflow . . . . .	67
4.7	Conclusion . . . . .	68
<b>5</b>	<b>Evaluation</b>	<b>69</b>
5.1	Software Quality Models . . . . .	69
5.1.1	Product Quality Model . . . . .	70
5.1.2	Quality in Use Model . . . . .	72
5.2	Evaluation Process . . . . .	73
5.3	Minimal Application . . . . .	74

5.3.1	Functional Requirements . . . . .	74
5.3.2	Data Collection Techniques . . . . .	75
5.3.3	Results . . . . .	75
5.4	Developer Study . . . . .	76
5.4.1	Participants . . . . .	76
5.4.2	Task Description . . . . .	76
5.4.3	Schedule . . . . .	77
5.4.4	Data Collection Techniques . . . . .	78
5.4.5	Results . . . . .	79
5.5	Limitations . . . . .	83
5.5.1	Observable Properties . . . . .	83
5.5.2	Dynamic Property Additions . . . . .	83
5.5.3	Class Constructor Parameters . . . . .	84
5.5.4	Manual Subscriptions . . . . .	84
5.5.5	Tree Structure . . . . .	84
5.6	Conclusion . . . . .	86
<b>6</b>	<b>Conclusion</b>	<b>87</b>
<b>A</b>	<b>JavaScript Object Notation (JSON)</b>	<b>101</b>
<b>B</b>	<b>Developer Evaluation: Questionnaire</b>	<b>103</b>
<b>C</b>	<b>Developer Evaluation: Results</b>	<b>107</b>

# 1 INTRODUCTION

Gutwin et al. stated in 2011 [GLG11] that the "standard web browser is increasingly becoming a platform for delivering rich interactive applications." This development is driven among other things by the instant application consumption and device-agnostic provisioning appreciated by application users as well as the inherent scalability of the web. Due to the increased use of the web browser as runtime environment, the application stack has moved from the server to the client during the last decade. Potentially encompassing the user interface description, the business logic and an in-memory data representation, it is challenging for developers to handle the application stack in the browser. Low-level JavaScript libraries like jQuery<sup>1</sup>, Prototype<sup>2</sup>, Underscore.js<sup>3</sup>, etc., provide a convenient API for manipulating the Document Object Model (DOM) in a uniform way across different browser implementations. However no means to structure the application code are provided, usually resulting in lots of DOM element selector and callback code. This "spaghetti code of the 21st century" [MT08] is reminiscent of software development in the 1970s and leads to poor maintainability of applications. That's where web application frameworks come into play. Besides disburden developers from writing boilerplate code, frameworks can structure web applications enforcing the separation of different application parts. According to the Model-View-Controller design pattern [Ree79b, Ree79a] frameworks usually separate user interface definition, application data, and business logic. Due to this modularization, frameworks support the fast development of applications and promote reuse as well as maintainability. As of today, several frameworks are available.

Moreover, in today's competitive environment geographically dispersed teams have to work together in joint projects in an effective and efficient manner. Creating, sharing, discussing and managing information together is called collaboration and if supported by computer software referred to as Computer Supported Collaborative Work (CSCW). CSCW systems can support different kinds of collaboration that are summarized in the time-space matrix introduced by Robert Johansen in 1988 [Joh88]. The matrix, shown in Figure 1.1, has two dimensions with two different values. First, the geographical distribution of the collaborating persons which can be located at the same or at different sites. Second, the timing between the interactions, which is called synchronous if there are short timeframes between the interactions of users and asynchronous if there are long periods in between.

The thesis will concentrate on the synchronous interaction of distributed users, referred to as (real-time) collaboration in the following. One crucial characteristic of applications supporting this collaboration type is the synchronization of different document copies among several participants

---

<sup>1</sup><http://jquery.com/>

<sup>2</sup><http://www.prototypejs.org/>

<sup>3</sup><http://documentcloud.github.com/underscore/>

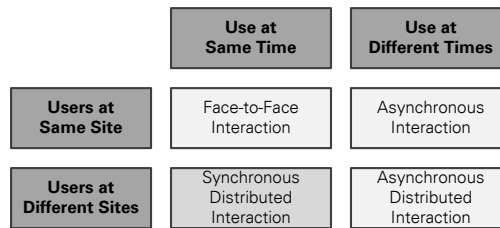


Figure 1.1: Time-Space Matrix of Collaboration (cf. [Joh88])

in a timely manner. An example of a web application supporting this interaction paradigm is Google Docs<sup>4</sup>, which supports the shared editing of rich text documents.

While Section 1.1 will motivate and explain the main contributions of the thesis, Section 1.2 introduces a simple use case of a collaborative application serving for demonstration purposes during the whole thesis. Finally, Section 1.3 outlines the organization of the thesis.

## 1.1 MOTIVATION AND OBJECTIVES

Although real-time collaboration is useful in several areas, e.g. in the collaborative authoring of documents or in distance learning, only few web applications are developed with integrated collaboration features in mind. As of today, a great deal of web applications lack collaboration support. This is caused by the synchronization and conflict resolution service required by all collaborative applications. Synchronization services are in charge of reconciling different document copies, while conflict resolution services handle conflicts raised by simultaneous changes of multiple users at the very same document.

Existing web application frameworks streamline the development of single-user web applications, but do not strive to ease the development of collaborative applications. Consequently application developers have to become familiar with additional techniques for the development of real-time applications. Synchronization and conflict resolution libraries like for example OpenCoWeb<sup>5</sup>, ShareJS<sup>6</sup>, etc., are one possible technique. Since developers have to become familiar with the additional techniques, and in most cases major source code changes are required, the development of real-time applications is a complex, cumbersome and error-prone endeavor.

To disburden the programmer from getting familiar with additional techniques, this thesis proposes the incorporation of collaboration support into existing web application frameworks. Collaboration-enabled frameworks shall promote the efficient development of collaborative web applications. The main contributions of the thesis are:

- Existing development approaches for the incorporation of real-time features into applications are assessed regarding challenges and drawbacks.
- Web application frameworks are analyzed with respect to their structure and concepts important for the integration of a collaboration extension. Examples comprise the enforced encapsulation and structure of application data.
- Based on the framework analysis, a collaboration extension is devised on a conceptual level enabling the integration of collaboration capabilities into framework-based applications in a lightweight fashion.
- To validate the concept, prototypes are implemented for two specific frameworks.
- The implemented prototypes are evaluated in an application developer study.

<sup>4</sup><http://docs.google.com/>

<sup>5</sup><http://opencoweb.org/>

<sup>6</sup><http://sharejs.org/>

## 1.2 INTRODUCTORY APPLICATION EXAMPLE

A small collaborative web application might enable the concurrent editing of a task list by multiple users. One user might add tasks to the task list, whereas another user assigns priorities to the tasks. A third one starts to fulfill the tasks and marks them as done. Moreover, an arbitrary number of users might watch the progress and make changes or corrections and every user is aware of others' changes in real-time. Figure 1.2 shows a mockup of such an application.

Name	Priority	Done
Recruit Participants	100	<input type="checkbox"/>
Develop Questionnaire	70	<input type="checkbox"/>

Figure 1.2: Simple Task List Application

From a more technical point of view, the task list is a one-dimensional array containing task items with a name, priority and done flag. The task items are in the same order for all participants. The task items' names are simple editable texts, the priorities are integer values between 0 and 100 and the done flag is Boolean. Since the task list application supports real-time collaboration, multiple users can modify list items at the very same time.

Typically user edits cause conflicts due to concurrent modifications. Thus, consistency has to be ensured eventually. Assume two users Alice and Bob that want to make a plan for conducting a user study. So far, they agreed on the fact that it is required to recruit participants and to develop a questionnaire. Alice now suggests to evaluate the questionnaire and therefore she adds a new task to the list. In the meantime Bob concludes that a source code analysis would be fine. He adds his suggestion to the task list too. The underlying synchronization mechanism has to reconcile the document copies, i. e. the conflict that arose from the concurrent inserts at the end of the task list has to be resolved. For example, since Alice is before Bob in the lexical order, Alice's task might appear before Bob's at all sites. Note that if there would be no conflict resolution, this conflict would lead to different orders for the task items at different sites.

## 1.3 STRUCTURE OF THE THESIS

The rest of the thesis is organized as follows:

- Chapter 2 outlines the evolution of the web from a simple document sharing environment to an almost full-fledged runtime environment for applications. Furthermore currently existing web applications belonging to several domains are classified and their capabilities in terms of real-time collaboration are assessed. Finally, existing development approaches for collaborative web applications are analyzed with respect to their challenges and drawbacks.
- Chapter 3 describes the design of a collaboration extension for web application frameworks based on JavaScript from a conceptual point of view.
- The prototypical implementation of collaboration extensions for two selected frameworks is delineated in Chapter 4. In addition to the selection process of frameworks serving as foundation for the prototypes, the different implementation layers of the collaboration extensions are described.

- In Chapter 5 the results of a developer study assessing several software quality characteristics for one of the implemented collaboration extensions are reported. Furthermore limitations of the current implementation are carved out.
- Chapter 6 concludes the work and emphasizes major contributions. Moreover an outlook to future research is exposed.

## 2 THE WEB AS APPLICATION PLATFORM

As of today, the web is well on the way to become the predominant application runtime environment. To underpin the importance, Section 2.1 sketches the evolution of the web during the last decades. Because the thesis targets web-based applications enabling synchronous collaboration, selected applications are investigated in Section 2.2 with respect to their real-time collaboration capabilities. To ensure that applications with different scopes are analyzed classifications of web applications regarding their primary application domain as well as their implementation technique are devised before. Due to the fact that only one third of the explored applications provides built-in collaboration capabilities, existing development approaches for collaborative applications are analyzed in Section 2.3 to carve out the major obstacles for application developers. Eventually Section 2.4 sums up the major findings of this chapter.

### 2.1 EVOLUTION OF THE WEB

Since 1990, when the first web browser prototype called "WorldWideWeb" [BL12] was released by Tim Berners-Lee, the web has evolved from a simple document sharing system to a multimedia content distribution and application runtime environment. The evolution, which is depicted in Figure 2.1, took place in three major steps [TM11].

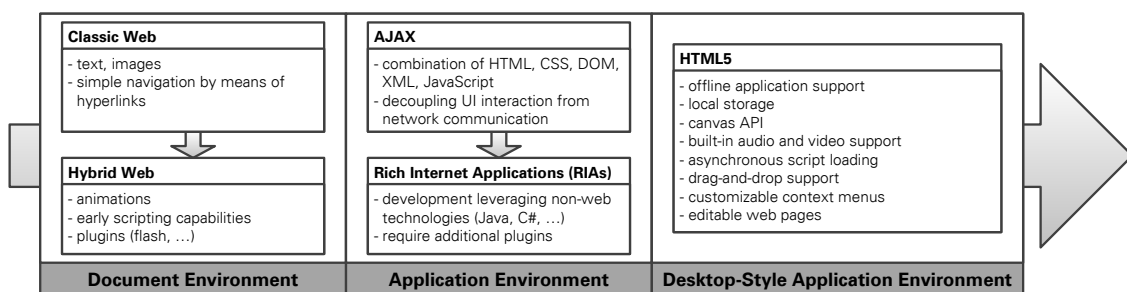


Figure 2.1: Major Evolution Steps of the Web (cf. [TM11])

In the first period (see Section 2.1.1) the web served only as document environment facilitating the distribution of documents among people. During that time the programming capabilities of

the web were very limited. Due to emerging software development capabilities, the web was used increasingly as application platform in the second period (see Section 2.1.2). To enable rich and interactive web applications, plug-in technologies were used, which demand explicit installation. Today the web is used, besides document sharing, for different purposes like online banking, stock trading, e-commerce, etc. Because of its instant availability, i. e. no explicit installation is required, and on-demand usage of applications, the web will become the leading application runtime environment during the third period [TM11] (see Section 2.1.3). Conventional binary programs will be confined to system software, whereas end-user software will shift from binary programs to web-based programs [TM11].

### 2.1.1 The Web as Document Environment

Starting from the early 1990s, the web was used primarily as a simple document viewing environment.

In the era of the **Classic Web** (early 1990s) web pages were simple page-structured documents containing nothing but text with images. Users could navigate between different pages leveraging hyperlinks. Every time the user clicked on a link, a new page was obtained synchronously from the web server. Therefore every page navigation on the user interface caused an HTTP request from the client to the server, some server-side processing and finally the return of a new HTML page to the client. Because the interaction with the user interface was blocked until the new page arrived at the browser, this interaction model between client and server is referred to as synchronous interaction scheme [Gar05].

Due to the introduction of Dynamic HTML (DHTML) [Goo07] in the late 1990s, the web started to move in directions that were not foreseen by the original designers, referred to as **Hybrid Web**. DHTML is a combination of HTML with Cascading Style Sheets (CSS), the Document Object Model (DOM) and the JavaScript programming language. Leveraging these additional technologies, web pages became increasingly interactive, containing richer content like animated graphics. Audio and video contents could be embedded using plug-in technologies like Adobe Flash<sup>1</sup>, Apple QuickTime<sup>2</sup> and Macromedia Shockwave<sup>3</sup>. Furthermore business logic started to move from the server to the client. However, the interaction scheme between client and server was still synchronous.

### 2.1.2 The Web as Application Environment

**Asynchronous JavaScript and XML (AJAX)**, coined by Jesse James Garret in 2005 [Gar05], changed the primary interaction model of the web and therefore increased its use as an application environment. AJAX is rather a combination of existing technologies used together in a particular way than a dedicated technology. HTML and CSS for standards-based presentation, the DOM to enable dynamic user interface manipulations, data interchange and manipulation by means of XML and XSLT, XMLHttpRequests for asynchronous data retrieval and JavaScript [Fla11] linking everything, are the major building parts of an AJAX-based application. Leveraging these technologies decouples user interface interaction from client-server communication and eliminates the click-wait-response interaction scheme, which was inherent to the web before. AJAX enables asynchronous communication with the web server, i. e. every user interaction corresponds only to a JavaScript function call, which is handled locally by the JavaScript engine of the browser. Therefore simple data validation can be carried out on the client without any server communication. More complex modifications use XMLHttpRequests to retrieve data from the server.

---

<sup>1</sup><http://www.adobe.com/de/products/flash.html>

<sup>2</sup><http://www.apple.com/de/quicktime/>

<sup>3</sup><http://get.adobe.com/de/shockwave/>



However, the browser could not be treated as full-fledged application runtime environment during this time, since it lacked a comprehensive set of APIs and sophisticated graphics capabilities.

To provide a rich set of APIs, similar to those for desktop application development, the web moved towards **Rich Internet Applications (RIAs)**. RIAs are built upon a development platform providing specific development tools and a dedicated runtime environment in form of a browser plug-in. Thereby the programming capabilities of the web (HTML, CSS, JavaScript) are hidden from the application developer, e. g. by programming languages like Java. Examples for RIA platforms are Adobe Flash<sup>4</sup>, Java FX<sup>5</sup> and Microsoft Silverlight<sup>6</sup>. The major advantage of these platforms are the powerful APIs available to the application developer, whereas the drawback is the necessity to install and update a dedicated browser plug-in to run the applications.

### 2.1.3 The Web as Desktop-Like Application Environment

The increasing utilization of the web browser as runtime environment for desktop-style applications is driven by several emerging World Wide Web Consortium (W3C) standards since 2010. Earlier standardization efforts can be tracked back to the Web Hypertext Application Technology Working Group (WHATWG)<sup>7</sup>, which has published first drafts already in 2008.

The **HTML5** standards landscape [Hic12a, Hic12c, Hic12b, Mar11] complements existing HTML standards with numerous new features. The following list contains some examples:

- *Asynchronous Script Loading* allows to load scripts asynchronously using the `async` attribute of the `<script>` tag.
- *Drag-and-Drop Support* provides an event-based drag-and-drop mechanism.
- *Customizable Context Menus* enable the adaptation of the web browser's context menus according to the application.
- *Canvas API* describes the `<canvas>` element providing a 2D drawing canvas, which can be used to create graphical shapes procedurally.
- *Built-in Audio and Video Support* describes `<audio>` and `<video>` elements facilitating the playback of audio and video files in the browser without requiring additional plugins.
- *Editable Web Pages* make client-side, rich text page edits possible leveraging the attribute `contentEditable`.
- *Web Storage* provides an API to a key-value store in the browser persisting application data as string values.
- *Web Messaging* allows the communication in HTML documents between different browsing contexts (e. g. iframes).
- *WebGL* enables the native rendering of 3D graphics in the web browser without requiring any plug-in components.

---

<sup>4</sup><http://www.adobe.com/products/flash.html>

<sup>5</sup><http://www.java.com/>

<sup>6</sup><http://www.microsoft.com/silverlight/>

<sup>7</sup><http://www.whatwg.org/>

## 2.2 CLASSIFICATION OF WEB APPLICATIONS

As outlined in the previous section, these days the web is becoming the predominant runtime environment for end-user applications. A myriad of web applications serving different purposes and adopting different implementation technologies is available. To structure the landscape, we devised possible classifications of web applications based on both application domain (see Section 2.2.1) and implementation technology (see Section 2.2.2).

### 2.2.1 Classification Based on Application Domain

The classification based on application domain categorizes web applications with respect to their primary use case. Figure 2.2 depicts six main usage scenarios each with further subordinated use cases. Be aware that the classification shall provide an overview about possible use cases without intending to be exhaustive. Furthermore some of the categories might be rearranged.

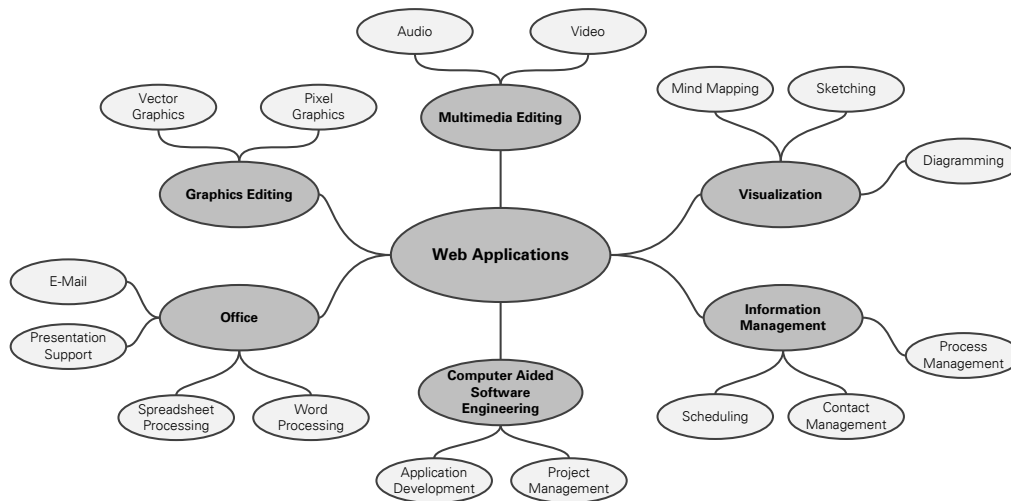


Figure 2.2: Web Application Classification Based on Application Domain

Web applications belonging to the **Graphics Editing** domain provide tools to edit vector or pixel graphics. Accordingly, typical drawing operations like the creation of lines, ellipses, rectangles, text, etc., are supported. Furthermore the filling of shapes or stroke styles might be modified. Graphics editing applications can be divided further with respect to their used data format. *Vector Graphics* tools use geometrical primitives such as points, lines, polygons, etc., based on mathematical expressions to represent the image. *Pixel Graphics* tools use a dot matrix data structure representing a grid of pixels, i. e. points with a specific color. Besides the typical drawing operations pixel graphics tools support the correction of under- and overexposure, blurring as well as noise reduction of images.

**Multimedia Editing** applications support the modification of audio and video data. *Audio Editing* applications enable the modification of audio clips by means of trimming, looping as well as adding effects like reverb, delay, etc. The seamless integration of video, audio, images and text, as well as the insertion of transitions and titles between different video clips is fostered by applications belonging to the *Video Editing* domain.

Frequent tasks at work might be supported by **Office** applications. Communication can be simplified using *E-Mail* applications to create and send (HTML) e-mails, as well as to browse through previous conversations. *Presentation Support* tools enable the creation of slide sets comprising texts, graphics and diagrams joined by transitions. Creating spreadsheets as well as conducting

calculations or data analysis is supported by *Spreadsheet Processing* applications. *Word Processing* applications aim at the creation of rich text documents. Therefore they allow modifying the style of text, the font size as well as the creation of tables and the insertion of images.

The category **Computer Aided Software Engineering** encompasses applications supporting the whole development process of software. Web-based integrated development environments, classified as *Application Development* tools, offer syntax highlighting and syntax checking for several programming languages. Features like automatic code completion as well as test and debug support are provided too. *Project Management* applications help organizing projects throughout their whole lifecycle supporting for example Gantt diagrams.

**Information Management** applications organize business-related and personal information. Allowing for the management and prioritization of tasks, the adding of due dates and notes, *Scheduling* tools help employees to keep track about their next action items. The organization of large contact databases can be achieved leveraging *Contact Management* applications, offering for example categorization of contacts and sophisticated searching capabilities. *Process Management* applications aim at the design, documentation and monitoring of business processes.

**Visualization** tools support the visual representation of varying objects. *Mind Mapping* applications enable the creation of tree-based visualizations, referred to as mind maps, to structure ideas carved out for example during a brainstorming session. Whiteboards are often used to visualize and communicate thoughts. Therefore *Sketching* applications usually provide a collaboration-enabled real-time whiteboard in the browser. Multi-purpose *Diagramming* tools support the creation of several kinds of diagrams like UML diagrams, organization charts, flow charts, networking diagrams, sitemaps, Venn diagrams, etc.

## 2.2.2 Classification Based on Implementation Technology

Because web applications can be implemented using different technologies, another possible classification is based on the used technologies. Figure 2.3 illustrates the different categories of such a technology-based classification out of the developers' perspective.

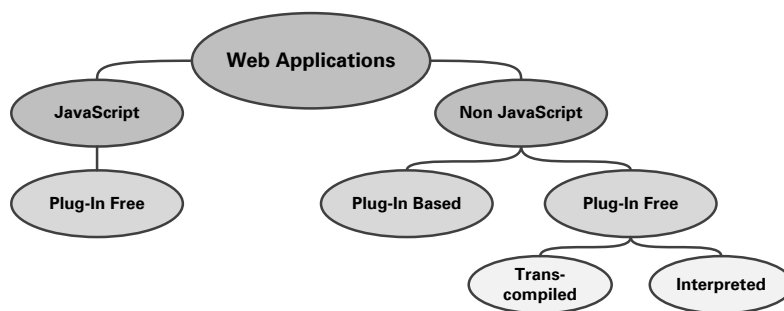


Figure 2.3: Web Application Classification Based on Implementation Technology

Currently **JavaScript** is the predominant implementation technique for *plug-in free* applications in the web. JavaScript applications can be built by means of pure JavaScript, leveraging low-level libraries like jQuery<sup>8</sup> or high-level frameworks like Knockout.js [San12]. Since applications built by means of JavaScript use standardized web technologies, they do not require a dedicated plug-in as runtime environment.

**Non JavaScript**, but *plug-in based* applications require the installation of additional browser plug-ins serving as runtime environment. Example technologies belonging to this category are

<sup>8</sup><http://jquery.com/>

Adobe Flash<sup>9</sup>, Java FX<sup>10</sup> and Microsoft Silverlight<sup>11</sup>. Because of the slipping importance of browser plug-ins, these technologies and the resulting applications are neglected in the following. Some non JavaScript, but *plug-in free* implementation techniques are available too. Some of them enable the developer to write the application's code in a language abstracting from concrete web technologies like HTML, CSS, etc. Therefore a transcompilation step is required to transform the code into pure JavaScript after development. Two example technologies are Google Web Toolkit<sup>12</sup> enabling web application development in Java, and CoffeeScript<sup>13</sup>. CoffeeScript improves the readability of JavaScript and enriches the language for example with list comprehension and pattern matching. In contrast to transcompiled languages, interpreted ones are evaluated at runtime leveraging an interpreter. Because the interpreter itself can be written in JavaScript, no plug-in is necessary. An example for this approach is Objective-J<sup>14</sup>.

### 2.2.3 Classification of Current Web Applications

To corroborate the classifications, Table 2.1 provides application examples for each of the classification categories. Because some applications comprise features relevant for different application domains, the assignment might be ambiguous. Moreover, the table contains information about the applications' real-time collaboration support and provides URLs pointing to the applications' web pages. The support was judged adopting two application instances and provides a first impression of the application share supporting real-time collaboration. Be aware that the provided list is not necessarily exhaustive.

Domain	Subdomain	Application	Real-Time Collaboration		Implementation Technology		URL
			Collaboration	Plug-In	Plug-In	Free	
Graphics Editing	Vector Graphics	SVG-edit	no		JavaScript		<a href="http://code.google.com/p/svg-edit/">http://code.google.com/p/svg-edit/</a>
		Aviary Raven	no	Flash		<a href="http://advanced.aviary.com/tools/vector-editor">http://advanced.aviary.com/tools/vector-editor</a>	
	Pixel Graphics	Splashup	no	Flash		<a href="http://www.splashup.com/">http://www.splashup.com/</a>	
		Pixenate	no		JavaScript		<a href="http://pixenate.com/">http://pixenate.com/</a>
Multimedia Editing	Audio	Aviary Myna	no	Flash			<a href="http://advanced.aviary.com/tools/audio-editor">http://advanced.aviary.com/tools/audio-editor</a>
	Video	Movie Masher	no	Flash			<a href="http://www.moviemasher.com/">http://www.moviemasher.com/</a>
Office	E-Mail	Roundcube	no		JavaScript		<a href="http://www.roundcube.net/">http://www.roundcube.net/</a>
		Presentation Support	Prezi	yes	Flash		<a href="http://prezi.com/">http://prezi.com/</a>
		SlideRocket	yes	Flash			<a href="http://www.sliderocket.com/product/">http://www.sliderocket.com/product/</a>
	Spreadsheet Processing	EditGrit	yes		JavaScript		<a href="http://www.editgrid.com/">http://www.editgrid.com/</a>
		GelSheet	no		JavaScript		<a href="http://sourceforge.net/projects/gelsheet/">http://sourceforge.net/projects/gelsheet/</a>
	Word Processing	CKEditor	no		JavaScript		<a href="http://ckeditor.com/">http://ckeditor.com/</a>
TinyMCE		no		JavaScript		<a href="http://www.tinymce.com/">http://www.tinymce.com/</a>	
Computer Aided Software Engineering	Application Development	Cloud9 IDE	no		JavaScript		<a href="http://c9.io/">http://c9.io/</a>
		CodeRun Studio	no		JavaScript		<a href="http://www.coderun.com/studio/">http://www.coderun.com/studio/</a>
Engineering	Project Management	TeamGantt	no		JavaScript		<a href="https://teamgantt.com/">https://teamgantt.com/</a>
		Redmine	no		JavaScript		<a href="http://www.redmine.org/">http://www.redmine.org/</a>
Information Management	Scheduling	Sandglaz	yes		JavaScript		<a href="http://www.sandglaz.com/">http://www.sandglaz.com/</a>
	Contact Management	BigContacts	no		JavaScript		<a href="http://www.bigcontacts.com/">http://www.bigcontacts.com/</a>
		Relenta	no		JavaScript		<a href="http://www.relenta.com/">http://www.relenta.com/</a>
	Process Management	Process Maker	no		JavaScript		<a href="http://www.processmaker.com/">http://www.processmaker.com/</a>
		IYOPRO	no	Silverlight			<a href="http://www.iyopro.com/">http://www.iyopro.com/</a>
Visualization	Mind Mapping	MindMeister	yes	Flash			<a href="http://www.mindmeister.com/">http://www.mindmeister.com/</a>
		MindDomo	yes	Flash			<a href="http://www.mindomo.com/">http://www.mindomo.com/</a>
	Sketching	Twiddla	yes		JavaScript		<a href="http://www.twiddla.com/">http://www.twiddla.com/</a>
		Dabbleboard	yes	Flash			<a href="http://www.dabbleboard.com/">http://www.dabbleboard.com/</a>
	Diagramming	LucidChart	yes		JavaScript		<a href="http://www.lucidchart.com/">http://www.lucidchart.com/</a>
		Creately	yes	Flash			<a href="http://creately.com/">http://creately.com/</a>
		GENI Family Tree	no	Flash			<a href="http://www.geni.com/">http://www.geni.com/</a>

Table 2.1: Classification of Available Web Applications

Table 2.1 shows 29 applications belonging to different application domains. The utilized implementation technique is usually JavaScript or Flash. Other technologies like Silverlight play only a secondary role. Due to this finding and the HTML5 standardization movements (see Section 2.1.3), a trend towards JavaScript applications might be assumed. Moreover only one-third of the applications provide real-time collaboration functionality. While almost all visualization tools provide collaboration functionalities, graphics editing, multimedia editing, office, as well as computer aided software engineering and information management tools usually lack collaboration support.

<sup>9</sup><http://www.adobe.com/products/flash.html>

<sup>10</sup><http://www.javafx.com/>

<sup>11</sup><http://www.microsoft.com/silverlight/>

<sup>12</sup><https://developers.google.com/web-toolkit/>

<sup>13</sup><http://coffeescript.org/>

<sup>14</sup><http://cappuccino.org/>

## 2.3 DEVELOPMENT APPROACHES FOR COLLABORATIVE WEB APPLICATIONS

Although several approaches for the incorporation of real-time collaboration features into applications are available, about two-thirds of the inspected web applications provide no possibilities to work together in real-time (cf. Table 2.1). To bring the reasons to light, different development approaches are analyzed regarding their impact on the source code of existing applications as well as imposed challenges on developers.

### 2.3.1 Operational Transformation Libraries

Operational transformation libraries implement a synchronization algorithm for documents subject to concurrent changes by different users named operational transformation (OT). The underlying idea of the algorithm is the representation of every document modification as operation. Operations are generated locally by a user and broadcasted to all other users afterwards. Incoming operations of remote users are checked locally, whether they are referring to an application state different from the current one. In the case of discrepancies, the OT algorithm transforms the incoming operations to be responsive to the differences. Afterwards incoming operations are treated as they were created locally based on the current application state. Details about the OT algorithm are exposed in Section 3.4.2. In the following some examples of OT implementations are introduced and assessed.

The **Open Cooperative Web Framework (OpenCoWeb)**<sup>15</sup> provides an OT implementation for the synchronization of shared strings and arrays. OpenCoWeb conforms to a client-server architecture. The client is implemented in JavaScript, whereas server implementations in Python based on the Tornado web server<sup>16</sup> as well as in Java using the CometD library<sup>17</sup> are available. Operations are called CoWeb events and can be created and received via a JavaScript API. OpenCoWeb is managed as open source project under the BSD license on GitHub.

**ShareJS**<sup>18</sup> provides collaborative editing capabilities for plain text and JSON data [Int11]. ShareJS follows also a client-server architecture. Both client and server are implemented in CoffeeScript, which can be transcompiled to JavaScript. The server runs on Node.js<sup>19</sup>, whereas the client can run either on Node.js or the web browser. Node.js is built on Chrome's V8 JavaScript engine, providing an event-driven non-blocking I/O model for the development of applications written in JavaScript. The API visible to the developer exposes methods to manipulate shared data and to react on changes to the data due to incoming operations.

**SAP Gravity** implements a context-based operational transformation algorithm similar to the algorithm introduced by David and Chengzheng Sun in [SS06]. During the last decade OT has evolved continuously and was used for several data structures including tree structures [DSL02]. SAP Gravity takes a step forward and brings OT to directed, labeled and attributed graph structures. Because SAP Gravity will be used as foundation for the prototypes in this thesis, further details are explained in Section 4.3.

Formerly known as Google Wave, **Apache Wave**<sup>20</sup> is a full-fledged real-time collaboration platform. Synchronization and conflict resolution is done by an operational transformation implementation, which supports the collaborative editing of XML documents. A special feature of

---

<sup>15</sup><http://opencoweb.org/>

<sup>16</sup><http://www.tornadoweb.org/>

<sup>17</sup><http://cometd.org/>

<sup>18</sup><http://sharejs.org/>

<sup>19</sup><http://nodejs.org/>

<sup>20</sup><http://incubator.apache.org/wave/>

Apache Wave is an operation composer reducing the amount of operations transmitted to the server by means of subsumption of operations. Implemented in a client-server fashion, Apache Wave consists of a stand-alone server which is based on Java, and skeletons for Google Web Toolkit (GWT)<sup>21</sup> client applications. GWT applications are implemented in Java and transcompiled to JavaScript afterwards.

Although OT libraries are used by almost all collaborative applications today, they impose some challenges to the developer. First, the application developer has to become familiar with the API provided by the OT library. Second, the OT library has to be connected with the application under development or with an already existing single-user application. This requires cumbersome and scattered source code changes. Moreover, most of the libraries restrict the choice of the development language. In case of Apache Wave all programming tasks have to be accomplished in Java, i. e. the utilization of JavaScript frameworks or libraries is not possible.

### 2.3.2 Collaborative Widget Libraries

Widget libraries, also known as widget toolkits or GUI toolkits, consist of several widgets used to create graphical user interfaces. Examples for such widgets are buttons, menus, scrollbars, editable text areas, etc. The widgets provided by a toolkit usually have the same look-and-feel, which is either hard-coded or exchangeable. While all toolkits procure an API for developers, some of them provide visual tools for constructing application UIs. Collaborative widget libraries accommodate widgets enriched with real-time synchronization and conflict resolution capabilities useful for the construction of collaborative applications. In the following, libraries targeting the desktop as well as the mobile application domain are introduced and assessed.

The **Multi-User Awareness UI Toolkit (MAUI Toolkit)** [HG04] developed by Hill and Gutwin is a collaborative UI widget library. MAUI is based on the Java runtime environment and supports the development of desktop applications primarily. Standard Swing widgets like buttons, menus, text boxes, etc., are enriched with functionality for synchronization and conflict resolution. Multi-user versions of the UI widgets provide the same publically accessible methods as the corresponding single-user widgets. Therefore single-user widgets can be replaced in a manner transparent to application and developer. Groupware specific widgets like a participant list are supplied too. Moreover all widgets are packaged as JavaBeans, which enables their reuse and ensures a good integration with existing integrated development environments (IDEs). One of the major drawbacks of the MAUI toolkit is, that in case of the conversion of an existing single-user application UI widgets have to be replaced with their collaborative counterparts. This leads to cumbersome source code changes, but provides the ability to mix single-user with multi-user UI elements to synchronize only parts of the application. The development of new applications is restricted to the use of Java. Furthermore MAUI is only applicable for the desktop landscape, i. e. standards-based web applications are neglected.

**Flexible Java Applets Made Multiuser (Flexible JAMM)** [BRS99, BSS01] is a widget library dedicated to Swing-based Java applications running in a desktop runtime environment or as applet in the web browser. To achieve synchronization, Flexible JAMM uses an OT algorithm. Advantageous compared to the MAUI toolkit is the dynamic replacement of single-user UI components with their multi-user counterparts. The major disadvantage of the Flexible JAMM idea is its limitation to platforms providing runtime replacement mechanisms for classes. The used platform has to provide dynamic binding, i. e. the resolution of a function invocation or data property access at runtime. Furthermore, due to the replacement of widget classes at runtime with collaborative counterparts, no selective replacement as with the MAUI toolkit is possible. For example all instances of an editable text box are replaced. The current Flexible JAMM implementation is also restricted to Swing-based applications and therefore neglects the domain of standards-based web applications.

---

<sup>21</sup><https://developers.google.com/web-toolkit/>

**WatchMyPhone**<sup>22</sup> provides a reusable set of collaboration-enabled UI components for the Android mobile platform<sup>23</sup>. The library of widgets encompasses common ones like editable text boxes, buttons, etc., as well as several groupware-specific widgets. Groupware widgets are for example a buddy list showing all the local user's friends, or a chat enabling communication via text messages in a collaboration session. Real-time synchronization is achieved using the CEFX operational transformation library<sup>24</sup>. Because mobile devices often have to deal with phases of disconnection, a message queue is available ensuring reliable data transfer. Because WatchMyPhone as well as the MAUI toolkit rely on a tight coupling between UI controls and synchronization features, both approaches have the same disadvantages regarding their usability for developers. Furthermore WatchMyPhone can only be used for applications built on the Android platform, which narrows the potential use cases substantially.

### 2.3.3 Transparent Adaptation

Transparent Adaptation (TA) enables the conversion of single-user applications into collaborative ones. It is neither a low-level solution like the OT libraries (see Section 2.3.1), nor targeted to applications built upon a special UI widget library (see Section 2.3.2). The term transparent adaptation was coined, since the transformation of a single-user application into a collaborative multi-user application does not require any source code changes in the original application. Instead an already existing API provided by the application is used. Therefore the conversion of off-the-shelf applications developed with no collaboration support in mind is fostered.

The **Transparent Adaptation** approach was pioneered by Sun et al. [XSS+04, SXS+06]. Initially two commercial single-user applications (Microsoft Word and Microsoft PowerPoint) were transformed into collaborative applications, named CoWord and CoPowerPoint. Word was chosen for transformation due its comprehensive data types and operations, its sophisticated API as well as its prevalent adoption. PowerPoint has been subject to investigation, because slide authoring tools make use of different data types and operations compared to word processors. In the meantime several other applications were transformed. Examples comprise Microsoft Visio [LCSD07], OpenOffice Writer [SSZ07], Autodesk Maya [ALX+08], AutoCAD [ZSS09] and Eclipse [FS12].

Document synchronization and conflict resolution is achieved by an operational transformation algorithm. To bridge the gap between application data model and collaborative data model of the OT implementation a collaboration adapter is used. This adapter utilizes the API of the single-user application as well as the OT API. Because of the transformation via the application API, functionalities as well as look-and-feel of the single-user applications are retained. Therefore no additional trainings concerning the single-user features for end-users working with transformed applications are necessary. However the TA approach is only suitable, if an application API is already available and this API provides comprehensive access to the application's data and operations. Moreover the devised collaboration adapter is only suitable for a specific application or a set of applications providing the very same API.

Developed by Heinrich et al., the **Generic Transformation Approach** [HLSG12] enables the transformation of existing single-user web applications into collaborative applications. The conversion is achieved by means of a JavaScript file import and a simple configuration. Standardized World Wide Web Consortium (W3C) APIs, available in almost all browser implementations, are used to synchronize different application instances using a collaboration adapter based on SAP Gravity (see Section 4.3). Since W3C APIs are application-agnostic, the adapter can be reused for almost all applications built upon web standards. This reuse supports the cost-efficient transformation of existing and the development of new collaborative web applications. However, only

---

<sup>22</sup><https://github.com/ubuntudroid/WatchMyPhone>

<sup>23</sup><http://www.android.com/>

<sup>24</sup><http://sourceforge.net/projects/cefx/>

application state changes reflected in the DOM are synchronized. Therefore the generic transformation approach is only suitable for applications storing their whole application state in the DOM. If an application uses for example a JavaScript array containing DOM elements to maintain the application state, a transformation is not possible. Moreover the generic transformation approach suffers from cross-browser issues, because DOM representations of a specific application state might be different among several browsers. For instance, adding and afterwards removing line breaks in text nodes is handled differently in browsers. While in Chrome and Safari a text node is still split into two parts after a line break is removed, in Firefox the two parts are merged again.

### 2.3.4 Code Injection

Code injection approaches leverage miscellaneous mechanisms to inject additional source code into existing application code. Aspect-oriented programming [KLM<sup>+</sup>97] enables the addition of source code at special points in the application code referred to as join points. Therefore application concerns that were intermingled with and scattered over the whole source code before, can now be specified separately. Thus, collaboration functionality can be defined as aspect and injected into existing application code. Annotation-based code injection provides the possibility to augment original source code with annotations. These annotations can also be used to inject collaboration functionality into existing single-user applications.

The **Zipper** system [RP05] leverages aspect-oriented programming to inject collaboration functionality into applications. Zipper is delivered as Eclipse feature accommodating several Eclipse plugins. Relying on the Eclipse platform is advantageous because the platform abstracts operating system dependencies, provides graphics and widgets libraries and is open source. Because of the open, human-readable source code, the identification of join points to inject collaboration functionality via aspect oriented programming techniques is possible. Collaborative applications using Zipper comply with a synchronized state architecture [Pat95] where multiple copies of the entire application are distributed within the network. The mandatory state synchronization and conflict resolution is done by means of an OT algorithm. Even if this approach enables the injection of collaboration code into existing application without touching the original source code, it is quite hard to use. Because join points have to be identified by the developer, familiarity with the application's source code in detail is necessary. In case of large application this is a big challenge.

To the best of our knowledge no approach using annotation-based code injection to incorporate real-time collaboration functionality into applications is available currently.

## 2.4 CONCLUSION

Today the web is a real application runtime environment with desktop-style user interaction capabilities, persistence mechanisms, etc. Due to the HTML5 movement, the web is well on the way to become the predominant runtime environment for web applications.

To provide a structured overview about available web applications, two possible classifications were introduced. The classification based on application domain as well as on the used implementation technology. Typical application domains comprise graphics editing, multimedia editing, office, computer aided software engineering, information management as well as visualization. Application implementations use JavaScript or non JavaScript technologies that either require browser plug-ins or not. Because plug-ins are not available on each and every browser, plug-in based applications are of slipping importance. Therefore we assume that JavaScript will be the predominant client-side implementation technique for future web applications.



Although several development approaches for collaborative web applications are available, about two-third of the analyzed web applications lack real-time collaboration features. Even though real-time collaboration might not be of crucial importance in some use cases (for example contact management), other application domains like office applications could profit substantially from the introduction of collaboration support. The small amount of collaboration-enabled applications may be due to current implementation techniques for real-time support requiring too much effort. Using OT libraries requires the familiarity with the OT API as well as major source code changes in the whole application. Widget libraries abstract the low-level OT API but still require scattered source code changes in the application. Furthermore they force developers to use a special UI toolkit. Transparent Adaptation techniques were designed to enable the transformation of existing single-user applications without source code changes. Because TA approaches communicate with the application only via its API, the API has to provide comprehensive access to the application. Aspect-oriented programming techniques require the analysis of the application's source code to specify join points. This can be a challenge in large applications. Furthermore the reuse of predefined aspects is limited due to their dependency on concrete identifiers in the source code. Due to the necessity for major source code changes or the availability of comprehensive APIs, it can be concluded that current development approaches are hard to handle for developers.

Because of the findings in this chapter, the following ones will concentrate on JavaScript-based web applications. To ease the development of collaborative web applications a novel development approach for collaborative real-time applications will be introduced. This approach tries to eliminate some of the shortcomings of existing development approaches.



### 3 CONCEPTION OF A COLLABORATION EXTENSION FOR JAVASCRIPT FRAMEWORKS

Changing the source code of an application at various points to integrate collaboration features is an error-prone and cumbersome task. Therefore, we aim to enhance existing web application frameworks with collaboration extensions. These extensions shall enable a lightweight and rapid integration of collaboration features into framework-based applications.

The basic structure of a framework-based application running two instances enriched with a collaboration extension is depicted in Figure 3.1. Each instance is a separate copy of the whole application and comprises different application parts according to the Model-View-Controller design pattern (see Section 3.2.1). Thus, multiple copies of the application data exist, which might become inconsistent between the instances. Therefore consistency management for the shared application, i.e. synchronization and conflict resolution, has to be ensured via the integrated collaboration extension.

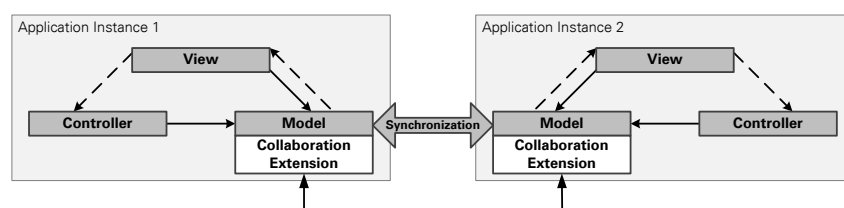


Figure 3.1: Application Running on Multiple Sites with a Collaboration Extension

Section 3.1 introduces important requirements for collaboration extensions to circumvent the issues of current development approaches for collaborative applications (see Section 2.3).

Since a collaboration extension ideally shall be applicable to several framework-based applications, frameworks have to be evaluated to find common characteristics that enable the specification of a reusable collaboration extension. Frameworks usually share structural blueprints provided by design patterns. Those structural constraints might be suitable for the integration of a reusable collaboration extension. Thus, the most important structural patterns are described in Section 3.2.

Even though design patterns fix the general structure of frameworks and thus framework-based applications, the actual implementation can differ in several ways. As a result, no collaboration

extension suitable for all framework-based applications adhering to a specific design pattern can be devised. Fortunately, groups of frameworks provide similar implementations of the patterns. These similarities enable the conception of collaboration extensions applicable to a group of frameworks having comparable properties.

To distinguish frameworks, facets like the framework architecture, programming language, framework integration, UI specification, UI update as well as data model structure and listener registration are important. While some of the facets crucially influence the design of collaboration extensions, others are only important in terms of the overall development efficiency. Section 3.3 gives a structured overview of the different facets. Moreover, conceptual architectures of collaboration extensions compliant with frameworks adhering to a specific set of characteristics are introduced in Section 3.4. Section 3.5 summarizes the findings of this chapter.

### 3.1 REQUIREMENTS FOR A COLLABORATION EXTENSION

The existing development approaches for collaborative real-time applications introduced in Section 2.3 suffer from several issues leading to an error-prone and dull development process and/or a steep learning curve for the application developer. Therefore a novel framework extension is devised that eliminates most of the existing shortcomings. In the following the major requirements to tackle this challenge successfully are explained.

Investigations on the history of the web [TM11] showed that the web browser will become the predominant runtime environment for applications in the future. In addition the survey of existing web applications (see Section 2.2) revealed JavaScript as the predominant programming language for web applications. Therefore the framework extension shall enrich a framework that supports JavaScript and uses the web browser as target runtime environment.

Since real-time collaboration helps geographically dispersed talents to work together efficiently, the framework extension has to support concurrent work in a shared workspace by multiple participants. Because it is convenient for end-users to use the very same application in both single-user as well as collaborative scenarios, the user interface and functionality shall remain unchanged in the collaborative case. This requirement is referred to as application compatibility [SXS<sup>+</sup>06]. Furthermore unconstrained collaboration shall be supported. Especially relaxed What You See Is What I See (WYSIWIS) [SBF<sup>+</sup>87] shall be supported, enabling the participants to have completely different views on the shared data.

Out of the developers' perspective the framework extension shall have a flat learning curve. Ideally development skills already acquired during the development of single-user applications shall be applicable in the collaborative case. Moreover, the integration of collaboration functionality shall happen without major source code changes. Additionally, the application shall not be forced to expose a dedicated API.

Because the web browser runtime environment differs slightly among different implementations, i. e. various browsers, cross-browser scenarios shall be supported.

The following list summarizes the challenges and requirements explained in detail before:

- **R01:** The target runtime platform shall be the web browser.
- **R02:** The framework extension shall be based on JavaScript.
- **R03:** Concurrent work in a shared workspace shall be enabled by the extension.
- **R04:** Application compatibility shall to be ensured.

- **R05:** Users shall be able to perform arbitrary operations on any data object at any time.
- **R06:** The framework extension shall provide a simple API.
- **R07:** The framework extension shall have a flat learning curve so that developers can get started almost immediately.
- **R08:** The integration of collaboration functionality into an application should not require major source code changes.
- **R09:** The selective sharing of application parts shall be possible.
- **R10:** No specific application API providing comprehensive access to the application shall be required to use the framework extension.
- **R11:** The framework extension shall be reusable for several framework-based applications.
- **R12:** An application state reflecting not the whole JavaScript application state in the DOM, i. e. a scattered application state, shall be supported.
- **R13:** Cross-browser scenarios shall be supported.

## 3.2 ARCHITECTURAL PATTERNS OF JAVASCRIPT FRAMEWORKS

To disburden programmers from writing boilerplate code, well-known software engineering principles are supported by today's web application frameworks. **Separation of Concerns** [Dij82] is one representative. Introduced by Edsger W. Dijkstra in 1974, it requires different aspects of an application to be kept separate, i. e. the whole system shall be decomposed into subsystems with almost distinct functionality.

Another popular metaphor to structure applications is **Tools and Materials** established by Dirk Riehle and Heinz Züllighoven in 1995 [RZ95]. *Materials* are things the user can work on using different tools. *Tools* are necessary to modify the materials during work, because users are not allowed to access the materials directly. Providing efficient means of work, tools enable users to work with the material under a certain perspective. However, materials can be processed adopting different tools, all of them having different perspectives on the material.

Structuring applications according to the mentioned design principles is assisted by frameworks implementing design patterns that aim at a clean separation of different application facets [MT08]. The following application facets are kept separate in general:

- The user interface enabling the user to communicate with the application,
- application data representing the application's state, and
- the interaction of the user with the application triggering particular actions.

In the following sections two concrete examples for design patterns will be presented. The Model-View-Controller (see Section 3.2.1) and the Presentation Model pattern (see Section 3.2.2). Those patterns represent the basis for the derivation of a conceptual architecture for web application framework collaboration extensions.

### 3.2.1 Model-View-Controller (MVC)

The Model-View-Controller (MVC) pattern was developed by Trygve Reenskaug in 1979 [Ree79a, Ree79b], when he was visiting scientist at Xerox Palo Alto Research Laboratory. After Reenskaug left Xerox, Jim Althoff and others did the first implementation for Smalltalk 80, which is described in [KP88]. Later on, the pattern has been adopted to varying degrees in GUI libraries and application frameworks.

Focusing on the inspection and manipulation of data displayed via multiple views, the pattern describes the encapsulation of data with related methods in a data model isolated from its presentation and manipulation by the user. A schematic overview of the pattern structure is depicted in Figure 3.2.

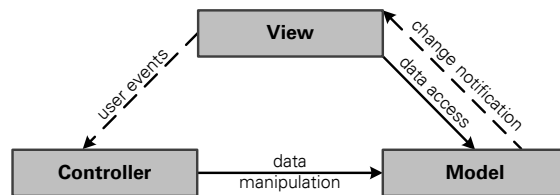


Figure 3.2: Model-View-Controller (MVC) Pattern

The MVC pattern comprises three major building parts:

The **Model** represents a collection of data with associated manipulation methods. Since the model contains no details about data representation on the screen, the views are immediately notified on any changes. Because views access the model only via a specific interface, the model can hide how the data is actually stored, i. e. whether it is using an in-memory data structure, a local file or a relational database.

A **View** serves as visual representation of one model, i. e. it determines how the model is displayed on the screen. Due to the fact that one model might have several views focusing on different aspects, a view can be envisioned as filter extracting certain aspects of the underlying model and neglecting others. To stay in sync with their underlying model, views subscribe to model changes and request updates whenever changes have occurred. Moreover views are in charge of delegating user events to the controller.

Formerly the **Controller** was treated as container for views, determining where they are placed on the user interface. During the last decades the role of the controller has changed to a container encapsulating the application's business logic. Therefore the controller has to update the model on behalf of user interactions with the view, i. e. the controller maps user actions to model updates. Because the MVC pattern envisages no direct reference between view and controller, both of them can be used interchangeably. A view might exchange its associated controller to behave differently, or a controller might be reused among multiple views.

### 3.2.2 Presentation Model (PM)

Nowadays user interfaces are often created by UI designers. However the state of the view, the user of an application is interacting with, is often represented only implicitly by the UI controls. This intermixture of UI representation with application state impedes the maintenance of both UI and business logic. Therefore the Presentation Model (PM) pattern, published by Martin Fowler in 2004 [Fow04], pulls state and behavior out of the controls. Providing an explicit representation of the UI state and behavior in an additional layer independent of concrete controls in the user

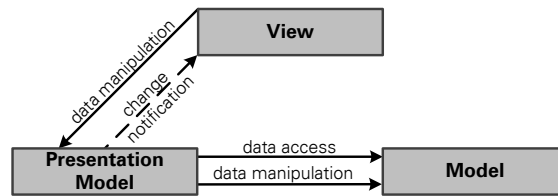


Figure 3.3: Presentation Model (PM) Pattern

interface, the presentation model has to synchronize its state with the UI continuously. Figure 3.3 illustrates the general structure of the pattern.

The **View** comprises different visual elements like graphics, buttons, input fields, etc., visualizing the presentation model on the screen. Almost always defined declaratively, a view does not store any state. Instead every view has assigned exactly one presentation model serving as data store for the view's state. Therefore the view and its assigned presentation model have to be synchronized anytime. Every time the presentation model changes, the view is updated accordingly and vice versa.

Abstracting a view, the **Presentation Model** is a self-contained pure code representation of the view's data and behavior. Because the presentation model does not have any information about the controls that render its data on the UI, it is not dependent on any specific UI framework. Presentation models contain data fields for all dynamic information of the view, i. e. not only for the content of the view, but also for all properties indicating for example whether some control is currently enabled or not. In case of HTML-based views any text node, style attribute, etc., can be mapped to a corresponding data structure. Due to the fact that views often require the data to be displayed in a special format, the presentation model contains value converters for their properties. Besides simple transformation functions, the presentation model contains complex operations to manipulate the application's data on behalf of the user.

The **Model** persists application data for example leveraging a XML data structure or a relational database. Being completely UI independent, the model communicates only with different presentation models that can be treated as a specialization of the model focusing on different aspects.

John Gossman picked up the Presentation Model pattern in conjunction with the Windows Presentation Foundation<sup>1</sup> in one of his blog posts in 2005 [Gos05]. Because presentation models serve as models of views he renamed the pattern to *Model-View-ViewModel (MVVM)* being the more popular term today. In the MVVM pattern the view model corresponds to the presentation model in the PM pattern. Gossmann focuses on the connection between the view and its corresponding view model. Synchronization between those two layers can be achieved either procedurally registering manually written callback functions or declaratively leveraging data bindings handled by a framework. To disburden developers from writing synchronization code, the use of a two-way data binding mechanism is recommended. Usually handled by a framework, this mechanism ensures that view and view model are synchronized on any changes, i. e. changes to the view model are reflected in the view immediately and vice versa. If referring to the PM pattern in the following, we will use the term MVVM.

### 3.3 CLASSIFICATION OF JAVASCRIPT FRAMEWORKS

The design patterns introduced in Section 3.2 specify only abstract archetypes for application structures enforced by frameworks. Therefore frameworks can implement the patterns differently resulting in different application structures. Because there is a plentitude of different implementations available in the web, in the following different facets that might be used to classify

<sup>1</sup><http://windowsclient.net/wpf/>

frameworks are introduced. Based on this classification different collaboration extensions suitable for a group of frameworks with specific properties are derived in Section 3.4. In the following, frameworks implementing the MVC (see Section 3.2.1) or MVVM (see Section 3.2.2) pattern are referred to as MV\* frameworks.

The classification facets for MV\* frameworks can be grouped into three major classification dimensions as depicted in Figure 3.4.

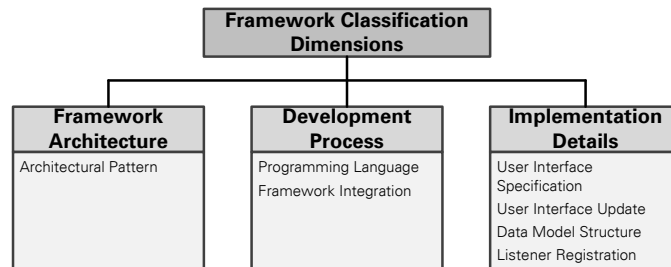


Figure 3.4: MV\* Framework Classification Dimensions

The **Framework Architecture** dimension encompasses the architectural pattern the framework implements. This facet is of crucial importance for the conception of collaboration extensions, since its values determine the application part which is target for synchronization. Differences in the programming language and the steps to incorporate a framework successfully into an application are the subject matter of the **Development Process** dimension. This classification dimension is of importance when selecting frameworks for the actual implementation of a collaboration extension, because the accommodated facets influence the overall development efficiency. Several technical aspects distinguishing MV\* frameworks are subsumed by the **Implementation Details** dimension. Especially the facet dealing with the data model structure impairs the design of collaboration extensions.

### 3.3.1 Architectural Pattern

MV\* frameworks might be classified with respect to the architectural pattern they adhere to. Currently the most popular patterns are **MVC** (see Section 3.2.1) and **MVVM** (see Section 3.2.2).

### 3.3.2 Programming Language

Developers using a framework are obliged to write their programs in a particular programming language. Which language is fostered by a framework is of crucial importance, since getting familiar with a new language is challenging and time consuming. Moreover languages differ in the supported programming paradigms and expressiveness.

Nowadays **JavaScript** is the most prevalent language for developing the client-side of web applications. Being developed since 1995, it was standardized for the first time in June 1997 by ECMA International in the "Standard ECMA-262" [Int97]. In June 2011 version 5.1 of the standard was released [Int11]. JavaScript is a dynamically typed, class-free and prototypal language [Cro08] able to mimic the object oriented, procedural as well as functional programming style. The popularity of JavaScript is nurtured by the runtime environment included in every up to date web browser.

**CoffeeScript**<sup>2</sup> is developed since 2009 by Jeremy Ashkenas and currently released in version 1.3.1 under the MIT license. Based on JavaScript it adds some syntactic sugar inspired

<sup>2</sup><http://coffeescript.org/>



by Ruby, Python and Haskell. Compared to JavaScript it uses for example line breaks to terminate expressions and indentation instead of curly braces. Moreover pattern matching as well as list comprehension is supported. Since CoffeeScript is transcompiled to JavaScript, existing JavaScript libraries can be used seamlessly.

Appeared for the first time in 2008, **Objective-J**<sup>3</sup> is today an inherent part of the Cappuccino framework. Having a syntax comparable to Objective-C, Objective-J adds for example class-based programming capabilities to JavaScript. Programs written in Objective-J can be transcompiled to JavaScript or interpreted at runtime leveraging a dedicated interpreter.

### 3.3.3 Framework Integration

Classification in terms of the framework integration targets the question, what steps have to be accomplished to make use of a framework.

Using a **single JavaScript include** is the simplest way to incorporate a framework into an application. In this case only a single JavaScript file has to be included in the application. Listing 3.1 depicts a simple application leveraging the Knockout.js framework [San12]. Notice the `<script>` tag including the JavaScript file named `knockout-2.0.0.js` that contains the framework code.

```
<html>
  <head>
    <script type="text/javascript" src="knockout-2.0.0.js"></script>
  </head>
  <body>
    <h1 data-bind="text: heading"/>
    <script type="text/javascript">
      viewModel = {
        heading: "My Task Application"
      };
      ko.applyBindings(viewModel);
    </script>
  </body>
</html>
```

Listing 3.1: Single JavaScript Include of Knockout.js framework

Frameworks leveraging a programming language aside from JavaScript often require a **preprocessor** to transcompile the application code into pure JavaScript. If the language cannot be transcompiled, often the integration and configuration of a dedicated runtime interpreter written in JavaScript is necessary.

### 3.3.4 User Interface Specification

Another classification dimension for MV\* frameworks is the way the developer can create the application's user interface.

Frameworks using **templates** to create the user interface usually allow the developer to specify HTML templates, i. e. simple HTML pages with placeholders for dynamic content filled at runtime. Revisiting the introductory example (see Section 1.2) Listing 3.2 shows the template definition for a single task item with name and priority leveraging Handlebars<sup>4</sup> semantic templates. Note the placeholders embraced by double curly brackets (`{{ . . . }}`).

<sup>3</sup><http://cappuccino.org/>

<sup>4</sup><http://handlebarsjs.com/>

```

<div class="taskItem">
  <span class="title">{{title}}</span>
  <span> - </span>
  <span class="priority">{{priority}}</span>
</div>

```

Listing 3.2: Template-Based UI Specification by means of Handlebars

Frameworks can also provide predefined **UI widgets** that might be used by the application developer to construct the application's user interface. Listing 3.3 shows once again the specification of a single task item having a name and priority, but this time by means of SproutCore<sup>5</sup> leveraging predefined UI widgets. Notice the `valueBinding` properties specifying placeholders with links to their corresponding dynamic content.

```

SC.View.design({
  childViews: ["labelView0", "labelView1", "labelView2"],
  labelView0: SC.LabelView.design({
    tagName: "span",
    valueBinding: ".title"
  }),
  labelView1: SC.LabelView.design({
    tagName: "span",
    value: " - "
  }),
  labelView2: SC.LabelView.design({
    tagName: "span",
    valueBinding: ".priority"
  })
});

```

Listing 3.3: Widget-Based UI Specification by means of SproutCore

### 3.3.5 User Interface Update

Application user interfaces have to be updated to represent the current state of the application. Because the way frameworks support the UI update differs, MV\* frameworks might be classified according to their UI update technique.

Usually used by MVVM frameworks (see Section 3.2.2), **data binding** enables the declarative specification of links between placeholders in the UI and dynamic content in the view model. On the basis of these links, synchronization code is automatically inserted in the application ensuring the automatic update of the view whenever the underlying view model changes, as well as the update of the underlying view model whenever the UI changes. Leveraging the Knockout.js framework [San12], Listing 3.4 shows the UI template and its corresponding data binding to the view model for a simple list of task items having only a name.

The `viewModel` variable encapsulates the view model of the application. In case of the example it only comprises an array containing tasks. This array is created as `observableArray` to enable the use of data bindings to the UI. The UI specification is template-based and binds an `<ul>` element to the `tasks` array. Therefore the `<li>` element contained in the `<ul>` element is rendered for each of the `tasks` array items. The `text` binding in the `<li>` element specifies that the element shall be filled with the string of the corresponding array item. Since the used `data-bind` attributes are not native to HTML, the browser is not able to interpret the attributes. Therefore the invocation of `ko.applyBindings(viewModel)` triggers the evaluation explicitly.

Another possibility to update the UI of an application is the **procedural view update**. In this case, code manipulating DOM elements as a consequence of model changes and vice versa, has

<sup>5</sup><http://sproutcore.com/>

```

<html>
  <head>
    <script type="text/javascript" src="knockout-2.0.0.js"></script>
  </head>
  <body>
    <ul data-bind="foreach: tasks">
      <li data-bind="text: $data"/>
    </ul>

    <script type="text/javascript">
      viewModel = {
        tasks: ko.observableArray([
          "Task One",
          "Task Two"
        ])
      };
      ko.applyBindings(viewModel);
    </script>
  </body>
</html>

```

Listing 3.4: Data Binding by means of Knockout.js

to be written by the developer. Listing 3.5 shows the model definition and a UI update function triggered due to model changes on the basis of Backbone.js<sup>6</sup>.

```

<html>
  <head>
    <script type="text/javascript" src="backbone.js"></script>
  </head>
  <body>
    <ul id="taskList"/>

    <script type="text/javascript">
      Task = Backbone.Model.extend({});
      TaskList = Backbone.Collection.extend({
        model: Task
      });
      tasks = new TaskList;

      tasks.on("add", function(element) {
        var ul = document.getElementById("taskList");
        var li = document.createElement("li");
        var text = document.createTextNode(element.get("name"));
        li.appendChild(text);
        ul.appendChild(li);
      });

      tasks.create({name: "Task One"});
      tasks.create({name: "Task Two"});
    </script>
  </body>
</html>

```

Listing 3.5: Procedural View Update by means of Backbone.js

The variable `tasks` references an instance of an observable collection, i.e. a collection that allows listeners to be notified upon the addition or removal of elements. In the example the collection contains simple task items having only a name. Every time a task is added to this collection the registered callback function is called. This function first obtains the root element of the task list from the DOM by means of its unique identifier. Afterwards it creates DOM elements representing the newly added task item. Finally the new elements are added to the list of task items in the DOM. Note that this function only captures the addition of new task items. To support the removal of task items another callback function has to be written.

<sup>6</sup><http://documentcloud.github.com/backbone/>

### 3.3.6 Data Model Structure

Besides eliminating boilerplate code, MV\* frameworks help to structure the application. Because data models are structured differently among frameworks, classification with respect to the model structure is possible.

**Access object-based** frameworks use the concepts of the data access object pattern [Mic02] to encapsulate their data. A typical resulting application structure is depicted in Figure 3.5.

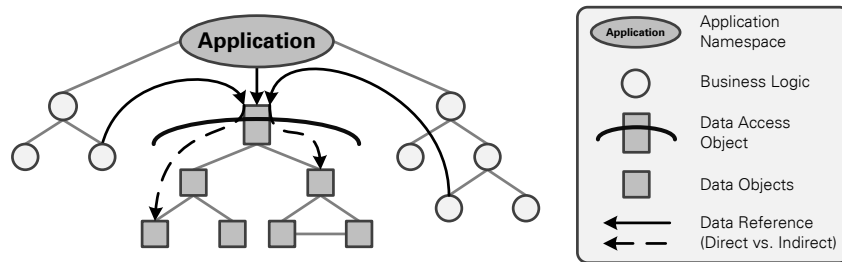


Figure 3.5: Application Structure of an Access Object-based Data Model

The application references the root object of the data model serving as data access object. Acting as intermediary between the business logic and the data, the root object provides methods to retrieve and store data. Because no direct access to the data objects in the data model is possible, the root object encapsulates the whole data model. To specify data objects that should be retrieved, a query language like XPath [CD99] might be used. The major advantage of this approach is the clean separation between application data and business logic. The interface provided by the data access object serves as single interaction point between business logic and application data. Therefore the abstraction from different storage mechanisms, for example file-based storage, a relational database system, an object-oriented database, etc., is possible.

**Remote proxy-based** frameworks can be treated as a specialization of access object based frameworks. A common application structure using a remote proxy data model is illustrated in Figure 3.6.

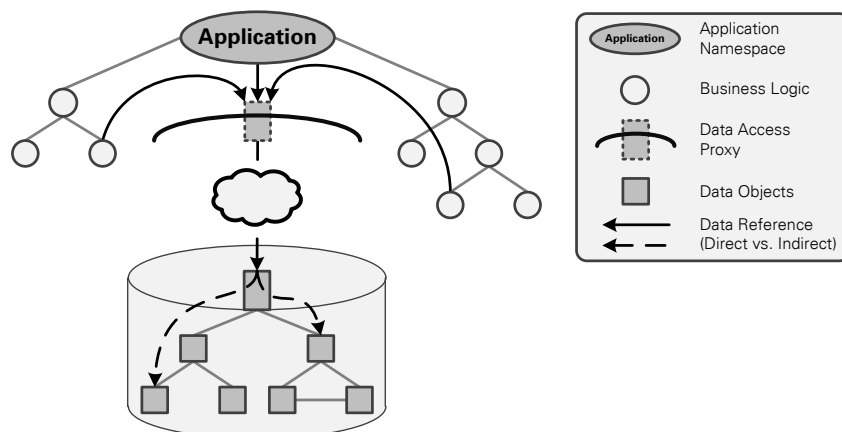


Figure 3.6: Application Structure of a Remote Proxy-based Data Model

Encapsulating the application data, the remote proxy can be seen as reification of the proxy design pattern [GHJV95]. The application has only a reference to the proxy object, acting as data access object that hides a remotely stored data structure. Data objects that shall be retrieved are specified by means of a query language like XPath [CD99] too. Besides abstracting from the data storage mechanism used, the proxy hides also the location where the data is stored.

Frameworks complying with a **subgraph-based** data model structure provide an encapsulation of the data model from the application's business logic not as strict as access object-based or remote proxy-based data models. Figure 3.7 depicts a common application structure possessing a subgraph-based data model structure.

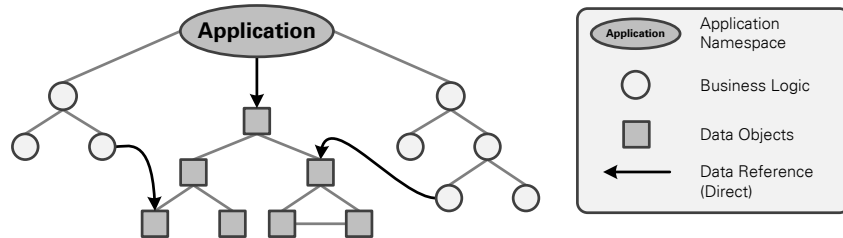


Figure 3.7: Application Structure of a Subgraph-based Data Model

The application has a reference to a root object of all data objects. Compared to access object-based and remote proxy-based frameworks, the root object serves not as interface for data access, i. e. even though being an entry point to the application data it is not a separating interface between the data and the application logic. Therefore the business logic has to traverse the data model itself, resulting in explicit references to the data objects within the data model subgraph. Storing data is not only done via the root object, but instead by means of functions provided by the data objects in the data model itself.

The weakest data model encapsulation concept frameworks can obey is referred to as **scattered data model**. Figure 3.8 outlines a possible application structure.

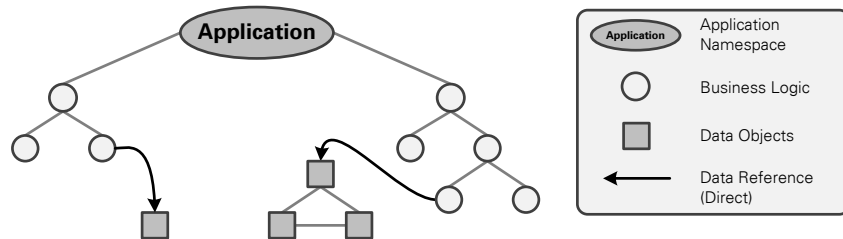


Figure 3.8: Application Structure of a Scattered Data Model

Compared to the three already discussed data model encapsulation possibilities, no central access point to the application data is available. Data is scattered over the whole application, i. e. it is interwoven with business logic that contains direct references to the data. The major disadvantage of this data model type is the intermixture of business logic with application data and the resulting break with the separation of concerns [Dij82] design principle.

### 3.3.7 Listener Registration

MV\* frameworks provide abilities to observe changes of the application state. To accomplish the notification every time the application state has changed, the registration of listener functions is possible. Handling the registration of listener functions differs in the place where the listener functions are registered.

In frameworks compliant with **model-based** listener registration all listeners are registered at the root object of the data model. This approach is not possible for subtree-based or scattered data models, because no access object encapsulating the whole data model is available. Revisiting the introductory example, Listing 3.6 shows a model based listener registration by means of SAPUI5 assuming that a task's name property shall be observed.

```

var model = new sap.ui.model.json.JSONModel();
model.setData({tasks: [{name: "Task One"}, {name: "Task Two"}]});

var binding = model.bindProperty("/tasks/0/name");
binding.attachChange(function(name) {
    window.console.log("The new task name: " + name);
});

```

Listing 3.6: Model-based Listener Registration using SAPUI5

First a new data model containing JSON data is created. Afterwards the model is filled with an object having a property `tasks` that references an array. This array contains several objects representing task items, each of them having a `name` property. To register the listener, a binding object is created accommodating all listeners to the property addressed by the path `/tasks/0/name`. The syntax of the addressing language is reminiscent of the XPath language [CD99]. Finally the concrete listener function is registered.

In frameworks using a **class instance-based** listener registration, listeners for single properties are registered at the surrounding class instance of the property. For data access object-based or remote proxy-based data models this registration pattern is not suitable, because no direct access to the class instances is possible. Listing 3.7 depicts the class instance-based listener registration for observing the name property of a simple task item using Backbone.js<sup>7</sup>.

```

Task = Backbone.Model.extend({});
var task = new Task({name: "Task One"});

task.on("change:name", function(task, name) {
    window.console.log("The new task name: " + name);
});

```

Listing 3.7: Class Instance-based Listener Registration using Backbone.js

First a class `Task` representing task items is created. Afterwards the `Task` class is instantiated for a task named `Task One`. Finally a listener is registered by means of the function with the name `on(event, callback, [context])`. The event parameter specifies the event to which the callback function provided in `callback` shall be registered. In our case `change:name` indicates, that all changes to the name property shall be tracked. The optional `context` parameter determines the value of the special JavaScript variable `this` during the callback execution. Eventually the registered listener function prints out the new value of the task's name property whenever it was changed.

Frameworks adhering to a **property-based** listener registration allow for the direct registration of listeners on single JavaScript object properties. Listing 3.8 shows the registration of a listener to a task item's name property using the API provided by the Knockout.js framework [San12].

```

var Task = function(data) {
    this.name = ko.observable(data.name || "default name");
}
var task = new Task({name: "Task One"});

task.name.subscribe(function(newValue) {
    window.console.log("The new task name: " + newValue);
});

```

Listing 3.8: Property-based Listener Registration using Knockout.js

First a `Task` class is created using a JavaScript object constructor function. Every task has an instance property `name` that is observable, i. e. listeners can be notified upon changes. The listener registration is accomplished by means of `subscribe(callback)`, where the `callback` parameter specifies the callback function executed on any changes to the property. The simple listener registered in the example always prints out the new value of the property on the browser console.

<sup>7</sup><http://documentcloud.github.com/backbone/>

## 3.4 CONCEPTUAL ARCHITECTURE OF A COLLABORATION EXTENSION

An important milestone in the evolution of software engineering was the NATO Software Engineering Conference in 1968. At this conference McIlroy introduced the idea of mass-produced and reusable software components [McI68]. Because today reuse belongs to the established software engineering principles, existing MV\* frameworks shall be reused and enriched with real-time collaboration capabilities via a collaboration extension. The advantages of this approach are three-fold:

- First, because the mature and stable code base of a framework is reused, the software quality of applications is improved.
- Second, the utilization of existing frameworks leads to a flat learning curve for developers, because experiences with the development of single-user applications can be reused in the collaborative case. This leads to improved efficiency when developing multi-user applications.
- Third, improved software quality and higher development efficiency lead to cost-savings.

In the following, the devised collaboration extension for MV\* frameworks is presented on a conceptual level. Section 3.4.1 provides an architectural overview of the collaboration extension. While the underlying synchronization service is explained in detail in Section 3.4.2, the integration of the collaboration extension into a framework-based application is discussed in Section 3.4.3. Finally in Section 3.4.4 the devised architecture is discussed with respect to the established requirements from Section 3.1.

### 3.4.1 Architecture Overview

Applications built with a collaboration-enabled MV\* framework adhere to a synchronized state architecture [Pat95]. Every application instance holds a complete copy of the application stack. To ensure consistency of the application data, the synchronization mechanism has to synchronize the parts of the application maintaining the application data. Figure 3.9(a) depicts two instances of an application complying with a MVC architecture. Since we assume all application data encapsulated and maintained in the model, the model is target for synchronization. Note that this assumption is not valid for all available MVC frameworks and/or depends on the specific use case of the framework. But what about updating the application's UI according to model changes? According to the MVC pattern (see Section 3.2.1) the model is in charge of notifying the view about changes. Hence, the view is updated on any model changes properly even if only the model is synchronized.

Figure 3.9(b) shows two MVVM applications. Even though the model part of the application encapsulates data, it is not the target for synchronization. Model data is only updated at special points in time, especially when application data shall be saved persistently. Therefore changes to the application data could be tracked on the model only insufficiently. In contrast the view model comprises the transient data the application is currently working with. Since all changes to the application data are reflected immediately in the view model, it is the target for synchronization in MVVM frameworks.

Model as well as view model are reconciled via a synchronization service. The service provides a dedicated sync service API (see Figure 3.9) enabling concurrent modifications on a shared data model. Therefore the data model of the application, i. e. the model or view model, has to be mapped to the collaborative data model of the sync service. All modifications on the application models have to be propagated to the collaborative data model and vice versa. Mapping as well as change propagation is ensured by a collaboration adapter bridging the gap between application data model and collaborative data model.

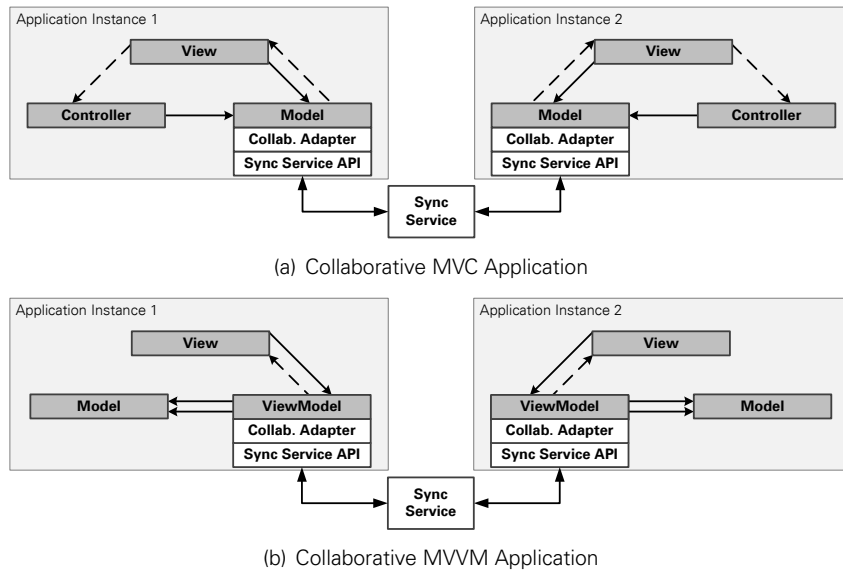


Figure 3.9: Conceptual Architecture of Collaborative MV\* Applications

### 3.4.2 Synchronization Service

The synchronization service depicted in Figure 3.9 is in charge of synchronizing different document copies and resolving conflicts in case of concurrent changes. Synchronization and conflict resolution can be achieved via different algorithms. Two prominent examples are the differential sync algorithm [Fra09] and operational transformation (OT) [EG89]. Because OT was enhanced during the last decades and is prevalent today, some insights are provided.

#### Operational Transformation

Operational transformation (OT) is a technique to maintain consistency of distributed documents that are subject to concurrent changes. Developed by Ellis and Gibbs in 1989 [EG89] for simple text documents, during the last decades several extensions of the original algorithm were developed to support for instance XML documents [DSL02]. The basic idea of OT is the representation of every document modification as operation. After being created, operations are applied to the document. If parallel modifications take place, concurrent operations are created that have to be transformed against each other before their application on the document to preserve the users' intention. Figure 3.10 emphasizes the necessity of transformations using a simple example.

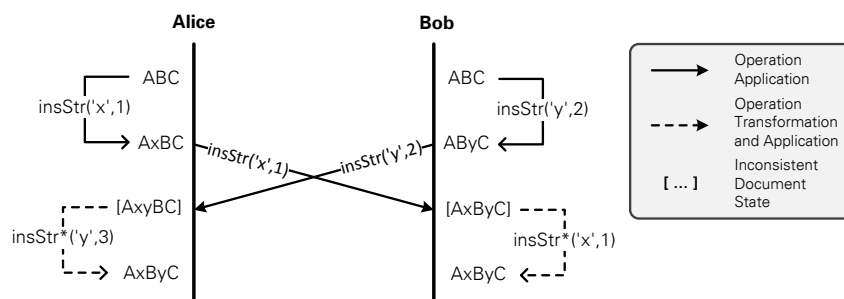


Figure 3.10: Synchronization Example Based on Operational Transformation for Two Participants Editing a Simple Text Document



The Figure shows two participants: Alice and Bob. Both of them have an own copy of the shared document "ABC" and manipulate it at the same time. Thus, simultaneous modifications on the shared document are possible. Alice inserts the character "x" at position 1. The generated insert operation  $insStr("x", 1)$  changes Alice's document to "AxBc". In the meantime Bob inserts the character "y" at position 2 in his document. Therefore his document is updated by an insert operation  $insStr("y", 2)$  and contains now the string "AByC". To notify the other participant about the changes, the participants exchange their operations. Based on this information the other participant can update its local document copy. If the incoming operations would be applied without any transformation, Alice's document would change to "AxyBC", whereas Bob's document copy would be "AxByC". Thus, the document copies would be inconsistent. Therefore the OT implementation transforms incoming remote operations against the local operations so that the different document copies become consistent and the users' intentions are preserved. How the operations are transformed against each other is defined by a transformation function  $T : Op \times Op \rightarrow Op$  where  $Op$  is the set of supported operations by the OT implementation. The following equation shows the partial transformation function definition for the transformation of two insert string ( $insStr$ ) operations.  $C(insStr(...))$  indicates the creation time of the operation and  $l(y)$  the length of the string  $y$ .

$$T(insStr(x, i), insStr(y, j)) = \begin{cases} insStr(x, i + length(y)) & \text{iff } i > j \\ insStr(x, i) & \text{iff } i = j \ \&\& \\ & C(insStr(x, i)) \leq C(insStr(y, i)) \\ insStr(x, i) & \text{iff } i < j \end{cases}$$

According to this definition Alice has to transform Bob's operation  $insStr("y", 2)$  against her local operation  $insStr("x", 1)$  leading to the operation  $insStr*("y", 3)$  resulting in the document state "AxByC". Bob transforms Alice's operation  $insStr("x", 1)$  against his own operation  $insStr("y", 2)$  leading to  $insStr*("x", 1)$ . If this operation is applied to Bob's document it changes also to "AxByC". Eventually Alice as well as Bob share the same consistent document state once again.

### Handling of Late Joiners

Late joiners are participants that want to join an ongoing collaboration session. Because the collaboration extension complies with a synchronized state architecture, late joiners have to receive a copy of the application in its current state. In the case of web applications, a copy of the application can be retrieved by downloading the necessary files from the web server and executing them locally. However, this simple approach would lead to an application in the wrong state. Therefore the application state has to be synchronized additionally. Assuming that the whole application state is reflected in the model or view model of an application, a current copy of the collaborative data model is obtained via the sync server API and pushed into the application data model via the collaboration adapter. Thus, no complex mechanisms to copy a process image of the running application from one address space into another is necessary.

### 3.4.3 Collaboration Adapter

The collaboration extension is integrated into a framework-based application by connecting the collaboration adapter with the application data model. Depending on the framework data model structure (see Section 3.3.6) two different ways to couple the collaboration extension with the application data model are available.

## Collaboration Proxy

The collaboration proxy can be used for access object- or remote proxy-based frameworks (see Section 3.3.6). The collaboration proxy mimics the interface of the access object or remote proxy and adds additional functionality for synchronization. In this case the collaboration adapter complies with the proxy design pattern [GHJV95].

Figure 3.11 shows the structure of a collaboration proxy. The `IDataModel` interface belongs to the single-user application and specifies the interface for the data access object or remote proxy. This interface is stable in all applications using a specific framework and defines methods for accessing the data model. For example a `getProperty(path)` method might be defined to obtain the value of a property at a specific location in the data model specified by the parameter `path`. `DataModelImpl` is the implementation class for the data access object or remote proxy. The `CollaborationAdapter` class is the new collaboration proxy providing a multi-user data model to the application. The adapter class implements the same interface as `DataModelImpl` and therefore can replace the original model implementation in a manner transparent to the application and its developers. The collaboration proxy holds a reference to an instance of the original data model class. Thereby the method implementations can add some additional code to handle the synchronization, but delegate the main work to the original method implementation.

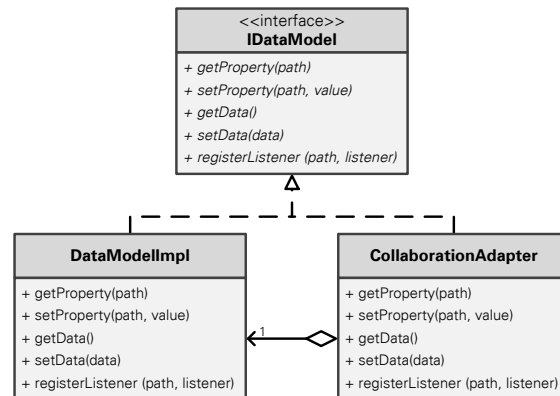


Figure 3.11: Class Structure of a Collaboration Proxy Implementation

## Annotation-based Code Injection

Even though subgraph-based data models have a root object, they lack a central access point to the data model. Hence, no central proxy object can be installed. Nevertheless the model can be synchronized by injecting synchronization code directly into the data model, i. e. into the objects in the data model subgraph. Because the root object of the data model can be accessible either via the global JavaScript namespace, inside a JavaScript function closure, or only inside a function, annotations might be used to mark the variable containing a reference to the root object of the data model. This annotation can be evaluated at runtime and triggers code traversing the data model. The traversal algorithm injects additional code to synchronize the data models of different application instances. To ensure that the whole data model is synchronized, the traversing algorithm determines the transitive closure of the data model, i. e. the set of all objects and properties reachable from the root object. A comparable approach was used by Hosking and Chen in [HC99] to implement persistence for object structures.

Figure 3.12 shows the runtime structure of an application adhering to a subgraph-based data model with injected synchronization code. The root object of the data model was marked with an annotation indicating that the application data model shall be synchronized. This annotation was

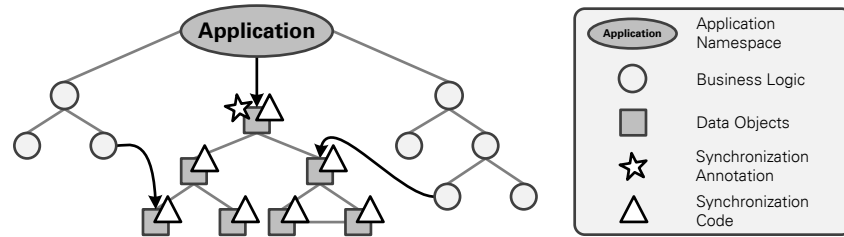


Figure 3.12: Subgraph-based Data Model with Injected Synchronization Code

evaluated at runtime and synchronization code was injected in the whole data model subgraph indicated by the white triangles.

Annotation-based code injection can also be applied in case of scattered data models. Rather than one synchronization annotation marking the root node of the data model subgraph, several synchronization annotations have to be spread throughout the source code. These annotations have to mark all scattered data elements that shall be synchronized.

### 3.4.4 Discussion of Requirements

In the following the requirements introduced in Section 3.1 shall be discussed with respect to the proposed framework extension.

Since JavaScript is the predominant programming language for web applications today, only JavaScript frameworks are going to be enriched with a collaboration extension. Therefore the programming language to be used by developers will also be JavaScript and the web browser will serve as runtime environment. Thus, the requirements R01 and R02 are fulfilled. Because existing frameworks are enriched, requirement R07 asking for a flat learning curve is achieved too. Already acquired development skills for single-user applications might be reused in the collaborative case.

The requirements R03 and R05 aiming at unconstrained collaborative work are fulfilled, since the collaboration extension adds synchronization as well as conflict resolution to the model or view model of a framework-based application. Because the view remains untouched in both cases, application compatibility (R04) is ensured. Furthermore the synchronization at model level solves cross-browser issues (R13). Even if the visual representation and therefore the DOM differ among several browsers, the underlying JavaScript data model is the same and can be synchronized successfully. Moreover only a partial representation of the application state in the DOM does not hinder the synchronization since modifications to the application state are handled directly at JavaScript level (R12).

The collaboration extension can be integrated into an application in a lightweight and convenient fashion annotation-based or via a collaboration proxy. Developers have only to be familiar with the instantiation of the collaboration proxy class or with a small set of annotations. This complies with the requirements R06 and R08.

Since collaboration extensions only assume a framework-specific data model structure, they can be reused among several applications relying on the same framework (R11). Furthermore no application-specific API has to be available providing comprehensive access to the application state (R10).

To ensure the requirement R09, the collaboration extension has to be enriched with a mechanism able to exclude parts of the application data model from synchronization. To specify parts

of the application data that shall not be synchronized, a configuration file and a path language reminiscent of XPath [CD99] might be used. In case of the collaboration proxy extension, on any modification the configuration might be used to check whether any synchronization is necessary or not. Annotation-based approaches might check the exclusion list while traversing the data model and inject collaboration functionality only partially.

Eventually, it can be stated that all the specified requirements for a beneficial and usable collaboration extension can be fulfilled by the introduced architecture.

## 3.5 CONCLUSION

Based on the issues of existing development approaches (see Section 2.3), the major challenges and requirements for convenient, lean, and reusable collaboration extensions for web application frameworks enabling unconstrained collaborative work were inferred in Section 3.1. The extension shall be based on JavaScript and run in any web browser without additional plug-ins. The integration with applications shall happen in a transparent manner, whereas the application's look-and-feel is retained. Besides ensuring a flat learning curve for developers, reuse shall be fostered.

Since a collaboration extension is in charge of reconciling different instances of a framework-based application it needs access to the application data. To enable reuse among several applications, commonalities in the data model structure are necessary. Design patterns describing the separation of an application's user interface from its business logic and application data, permit first structural claims. Widespread patterns are the Model-View-Controller (MVC) as well as the Model-View-ViewModel (MVVM).

Since patterns only describe an abstract application structure, the actual application structure might differ among several framework implementations. Thus, frameworks implementing the MVC or MVVM pattern were analyzed with respect to several characteristics that influence the design of collaboration extensions as well as the overall development efficiency of the final collaboration-enabled frameworks. It turned out that the architecture of the framework as well as the data model structure are of crucial importance for the design of a collaboration extension, whereas other facets like UI definition and UI update have not to be considered to ensure effectiveness of the extension.

Finally conceptual architectures of framework extensions suitable for both MVC and MVVM frameworks were described. Operational transformation was explained in more detail, since it is the predominant synchronization algorithm today. Furthermore special attention was paid to the integration of the extensions into framework-based applications depending on the enforced data model structures by the frameworks. The discussion of the proposed collaboration extension architectures with respect to the requirements revealed compliance with all requirements.

To prove the concept, Chapter 4 presents prototypical implementations of two collaboration extensions, which are evaluated in Chapter 5.

# 4 PROTOTYPICAL IMPLEMENTATION OF COLLABORATION EXTENSIONS

To prove that the suggested collaboration extensions described in Section 3.4 are able to extend current frameworks with real-time collaboration capabilities, two prototypes based on current frameworks were implemented. The process to select SAPUI5 and Knockout.js as basis for prototypical implementations is explained in Section 4.1. Section 4.2 describes the implementation architecture of the collaboration extensions for both frameworks. Both extensions were implemented based on the OT engine SAP Gravity and an extension enabling the concurrent editing of JSON documents referred to as Shared JSON. Hence, Section 4.3 introduces SAP Gravity, whereas Section 4.4 describes the Shared JSON extension. Framework-specific collaboration adapters that are part of the collaboration extensions bridge the gap between the framework data models and the Shared JSON extension. The implementations of the adapters are explained in detail in the Sections 4.5 and 4.6. Finally, Section 4.7 summarizes the gained experiences.

## 4.1 FRAMEWORK SELECTION

The selection of two MV\* frameworks to prove the feasibility of the devised framework collaboration extensions took place in three steps. First, the framework classification dimensions described in Section 3.3 are used in Section 4.1.1 to define preferred characteristics for every classification facet. Afterwards in Section 4.1.2 available frameworks are classified with respect to the introduced classification facets. This overview is used to select two frameworks serving as basis for the prototypical implementation of collaboration extensions. Finally, based on the described characteristics and the assessed frameworks two frameworks are chosen in Section 4.1.3.

### 4.1.1 Framework Selection Characteristics

The conceptual architecture covers collaboration extensions for both MVC and MVVM frameworks. Therefore one MVC as well as one MVVM framework should be selected. Since JavaScript is the most widespread programming language for web applications and pre-processed languages require an additional development step, the selected frameworks should be

based on JavaScript. Moreover, the integration of a framework into an application should be as easy as possible for the developer. Thus, only frameworks includable via a single JavaScript file should be considered.

With respect to the UI specification one template and one widget-based framework should be selected since both approaches exhibit advantages as well as disadvantages. Templates offer more freedom for customization but developers have to write more source code. In contrast widget-based frameworks force the developer to stick with predefined widgets. However the developer has to write less source code to built up the user interface. To enable UI updates according to application state changes in a convenient way, data binding should be supported. This enables developers to focus on the implementation of business logic without having to deal with boilerplate code.

Even if scattered data models might be supported via annotation-based code injection, the distribution of the data model throughout the whole application would require adding annotations at several places in the source code. Because scattered source code changes are not convenient for developers, only frameworks with a data model adhering to a subgraph, access object or remote proxy should be considered.

#### 4.1.2 Survey of Existing Frameworks

Table 4.1 provides an overview of available MV\* frameworks. Rows in the table show the different frameworks, while the columns classify them with respect to the classification dimensions introduced in Section 3.3. The character "x" equals to "yes" and "-" equals to "no". Furthermore additional information for every framework is provided:

- The license under which the framework is released,
- the size of the framework,
- watchers and forks on GitHub<sup>1</sup> (May 8, 2012) indicating the community support, as well as
- the website of the framework usually providing tutorials and a more or less comprehensive documentation.

While watchers on GitHub indicate the number of people watching the activities on the repository, forks show how often the project was forked to be developed further in an additional branch.

The table reveals the uniform distribution of MVC and MVVM frameworks as well as JavaScript being the predominant programming language. Furthermore most of the frameworks require no special integration process to be used within an application. Usually it is enough to include a single JavaScript file into the application's source code. Only frameworks using transcompiled programming languages like CoffeeScript, or frameworks with a full-fledged build process like SproutCore need a preprocessing step. All MVVM frameworks offer template-based UI specification whereas MVC frameworks usually provide predefined widgets that can be used for constructing an application user interface. Using data binding for updating the user interface is possible in almost all MVVM frameworks. Because data binding requires some subscription mechanism to notify the UI about changes, updating the UI in a procedural way is also possible with these frameworks. Because the latter is not the intended way, this possibility is indicated in the table by means of "(x)" in the corresponding column. MVC frameworks instead provide either data binding or a procedural mechanism to update the user interface. Regarding the data model MVVM frameworks usually adhere to a subgraph-based or scattered data model. The "(x)" in the remote proxy column indicates that even if the data model is scattered, the data objects

---

<sup>1</sup><https://github.com/>

Name	Architectural Pattern	Programming Language	Integr.		UI Spec.		UI Update		Data Model			Listener			License	Size	Watchers	Forks	Website
			Preprocessor	JS Include	Templates	Widgets	Data Binding	Procedural	Access Object	Remote Proxy	Subgraph	Scattered	Model	Class Instance					
AngularJS	MVVM	JavaScript	-	x	x	-	x	(x)	-	x	x	-	-	MIT	64 kB min. (Version 0.9.19)	919	126	<a href="http://angularjs.org/">http://angularjs.org/</a>	
Backbone.js	MVVM	JavaScript	-	x	-	-	x	(x)	x	-	x	x	-	MIT	16.1 kB min. (Version 0.9.2)	7998	1070	<a href="http://documentcloud.github.com/backbone/">http://documentcloud.github.com/backbone/</a>	
Batman.js	MVVM	CoffeeScript	x	-	x	-	-	(x)	x	-	-	-	-	MIT	15 kB min. (Version 0.9.0)	868	80	<a href="http://batmanjs.org/">http://batmanjs.org/</a>	
Cappuccino	MVC	Objective-J	-	x	-	x	-	x	x	x	-	-	-	LGPL	306 kB (Version 0.9.5)	1655	239	<a href="http://cappuccino.org/">http://cappuccino.org/</a>	
Ember.js	MVVM	JavaScript	-	x	-	-	x	(x)	-	x	-	-	-	MIT	144 kB min. (Version 0.9.7.1)	2807	304	<a href="http://emberjs.com/">http://emberjs.com/</a>	
JavaScript MVC	MVC	JavaScript	-	x	x	-	-	x	-	x	-	-	-	MIT	85.8 kB min. (Version 3.2.0)	405	72	<a href="http://javascriptmvc.com/">http://javascriptmvc.com/</a>	
Knockout.js	MVVM	JavaScript	-	x	x	-	-	(x)	-	-	x	-	-	MIT	38 kB min. (Version 2.0.0)	1873	221	<a href="http://knockoutjs.com/">http://knockoutjs.com/</a>	
SAPUI5	MVC	JavaScript	-	x	-	x	-	(x)	x	-	-	x	-	prop.	240 kB (Version 1.3.3)	-	-	<a href="http://scn.sap.com/community/developer-center/front-end">http://scn.sap.com/community/developer-center/front-end</a>	
SproutCore	MVC	JavaScript	x	-	x	-	x	(x)	x	-	-	-	-	MIT	23.5 MB (Version 1.8)	1739	222	<a href="http://sproutcore.com/">http://sproutcore.com/</a>	

Table 4.1: Survey of Existing Web MV\* Frameworks

can be remote proxies providing access to remotely stored data. MVC frameworks usually comply with an access object- or remote proxy-based data model. Listeners announcing changes in the data model are registered usually at class instance level. Model-based as well as property-based listener registration is used rarely. The MIT license is the most common license for all reviewed frameworks. Only two frameworks are published under other licenses such as LGPL or proprietary ones. Only frameworks like SproutCore having a complex build process reach some megabytes in size, whereas the typical size is some kilobytes. Backbone.js as well as Ember.js have a very active community indicated by many watchers and forks on GitHub. Since it is a relatively new framework, Knockout.js has not as many watchers and forks, but is nevertheless of big importance. In contrast, frameworks like JavaScript MVC are of slipping importance due to their limited community support.

### 4.1.3 Selected Frameworks

Analyzing the MV\* frameworks listed in Table 4.1 with respect to the characteristics mentioned in Section 4.1.1, three frameworks were identified as possible candidates for the extension with collaboration features: Angular.js<sup>2</sup>, Knockout.js [San12] and SAPUI5 [SAP12]. Because SAPUI5 is the only MVC framework meeting the requirements, it is the first elected framework. Both Angular.js and Knockout.js adhere to the MVVM architecture. Due to its bigger community support on GitHub and its massive developer adoption during the last three month resulting in 110 000 downloads, the Knockout.js framework was selected as second framework.

#### SAPUI5

SAPUI5 [SAP12] is SAP's next generation platform for rich user interfaces of web applications. The framework provides a large set of predefined UI controls that are customizable. Since the framework is compliant with OpenAJAX<sup>3</sup>, it can be used together with almost any JavaScript libraries provided by third party vendors. Examples include jQuery<sup>4</sup>, jQuery UI<sup>5</sup>, etc. Since SAPUI5 provides access object as well as remote proxy-based data models (see Section 3.3.6) it can be used to prove that the integration of collaboration functionality via a collaboration proxy (see Section 3.4.3) is viable.

#### Knockout.js

Knockout.js [San12] is a JavaScript framework released under the open source MIT license. The implementation is based on JavaScript and has a very small footprint (39 kB minified). Since Knockout.js has no dependencies it can be used in conjunction with any other JavaScript library or framework. For developers a very comprehensive set of documentation is available comprising an API specification, interactive tutorials as well as running examples. Because applications built with the Knockout.js framework comply with a subgraph-based data model (see Section 3.3.6), the framework can be used to demonstrate the feasibility of annotation-based code injection (see Section 3.4.3) to integrate collaboration functionality into an application.

---

<sup>2</sup><http://angularjs.org/>

<sup>3</sup><http://www.openajax.org>

<sup>4</sup><http://jquery.com/>

<sup>5</sup><http://jqueryui.com/>



## 4.2 IMPLEMENTATION STRUCTURE

The implementation architectures of the collaboration extensions for SAPUI5 and Knockout.js are depicted in Figure 4.1. To drive reuse among the extension implementations, the collaboration proxy of SAPUI5 as well as the Knockout.js adapter is not directly based on SAP Gravity. Instead a reusable implementation enabling the concurrent change of JSON data referred to as Shared JSON is used. Therefore the actual implementation stacks of the collaboration extensions comprise three layers:

1. A framework-agnostic synchronization layer comprising the SAP Gravity server as well as client implementations offering the SAP Gravity API. Details on Gravity are exposed in Section 4.3.
2. A framework-agnostic extension of Gravity enabling the concurrent editing of JSON documents. This extension, referred to as Shared JSON, is discussed in depth in Section 4.4.
3. Finally a framework-specific collaboration adapter implementation. This adapter might be either a collaboration proxy, or an annotation-based collaboration adapter injecting synchronization code based on annotations. The SAPUI5 collaboration proxy will be explained in detail in Section 4.5 whereas the Knockout.js collaboration adapter will be exposed in Section 4.6.

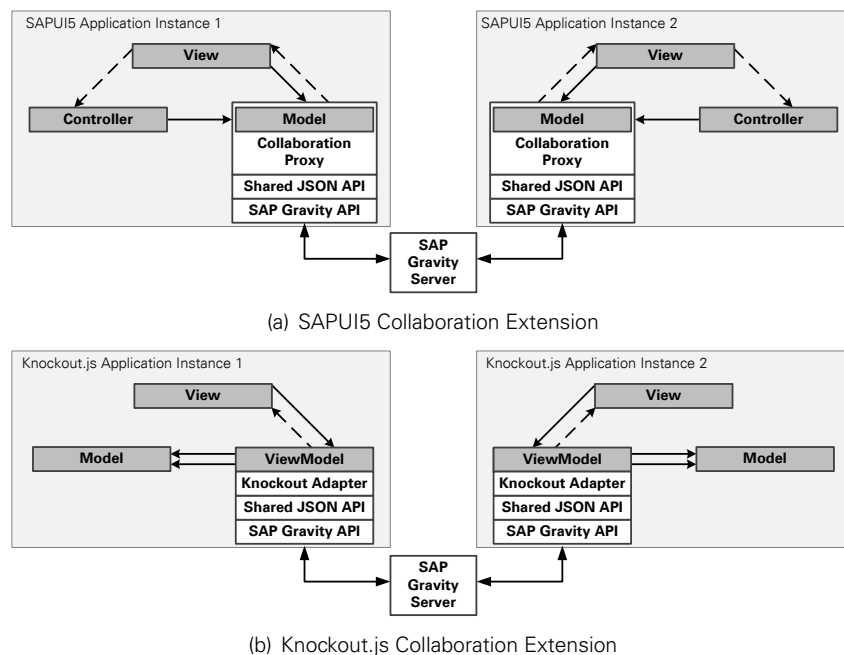


Figure 4.1: Implementation Architecture of SAPUI5 and Knockout.js Collaboration Extension

## 4.3 SAP GRAVITY

SAP Gravity is an OT engine (see Section 2.3.1) implementing an OT algorithm for directed, labeled and attributed graph structures. An OT implementation usually comprises three parts: The data model describing the available structures to store shared data, the control algorithm deciding which operations are subject to transformation, and the protocol coordinating various clients. Since the control algorithm and protocol are complex topics in the research domain of OT implementations, a detailed explanation is neglected in this thesis. Rather, emphasis is placed on the data model and the API available for developers to manipulate shared data.

### 4.3.1 Architecture

Figure 4.2 shows the architecture of SAP Gravity. Because fully distributed peer-to-peer protocols are more complex than centralized protocols, Gravity adheres to a client-server architecture.

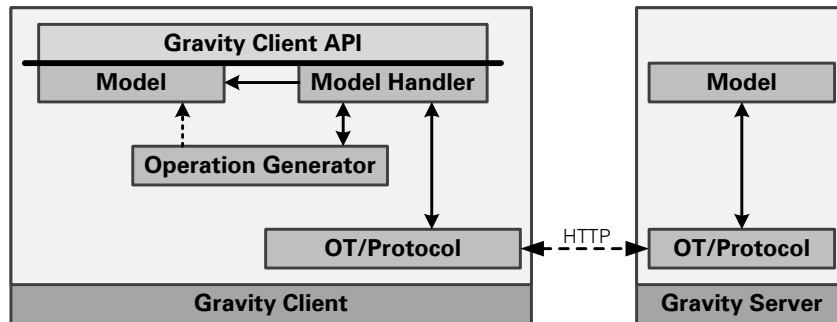


Figure 4.2: SAP Gravity Architecture

The **Gravity Server** implemented in Java acts as coordinator handling all communication between distributed clients working in a shared workspace. Besides storing a persisted version of the shared data, the server ensures that all clients eventually receive a consistent copy of the data. Persistence is handled by the *Model* whereas communication, synchronization and conflict resolution are managed by the *OT/protocol* component.

The **Gravity Client**, written in JavaScript, is replicated at all participants working together in a shared workspace. A JavaScript API (*Gravity client API*) providing access to the shared data and abstracting from the communication with other clients and the server is offered. The client API is explained in detail in Section 4.3.4. Each client accommodates a transient copy of the shared data (*model*). While reading access to the model is possible directly via the client API, leveraging the *model handler* is necessary to modify the shared data. The model handler ensures atomicity of modifications, i. e. all modifications submitted in a batch are applied entirely or not at all. To achieve atomicity the model handler implements a rollback mechanism that undoes already applied modifications of a batch if one of the modifications inside the batch could not be executed successfully. During the application of modifications on the shared data the *operation generator* keeps track of all changes and generates corresponding operations. This complies with the basic idea of OT to represent every modification on a shared data structure as operation. The operations generated for each single data modification are referred to as *primitive operations*. The set of primitive operations generated on behalf of a operation batch executed by the model handler, is termed *complex operation*. The *OT/protocol* component of each client implements the synchronization as well as conflict resolution mechanisms according to the OT approach. Furthermore communication with the server is handled.

### 4.3.2 Data Model

Gravity's data model is a directed, labeled and attributed graph with ordered references. Nodes in the graph are identified via a unique string. Besides the identifier every node can have several attributes of primitive types like string, number, boolean, etc. To connect nodes, Gravity supports two different types of references. Atomic references representing a 1:1 relationship as well as ordered references which can be treated as lists of referenced nodes (1:N relationship). For all nodes referenced by an ordered reference a relative, but not an absolute order is maintained. Relative ordering means that even if the indices of elements might change over time, the relative positions (before, after) towards other elements remain unchanged.

Figure 4.3 depicts an instance of a data model. The model comprises three nodes denoted by `node1`, `node2`, `node3`. The first node has an attribute named `attr1` with the boolean value `true` assigned. Moreover this node maintains an ordered reference `oref1` which points to `node2` at index 0 and to `node3` via index 1. An atomic reference `aref1` connects `node2` with `node3`. Furthermore two attributes (`attr2` and `attr3`) are assigned to `node2`. The attribute named `attr2` contains the string value `"string"` whereas the attribute `attr3` holds the number `123`.

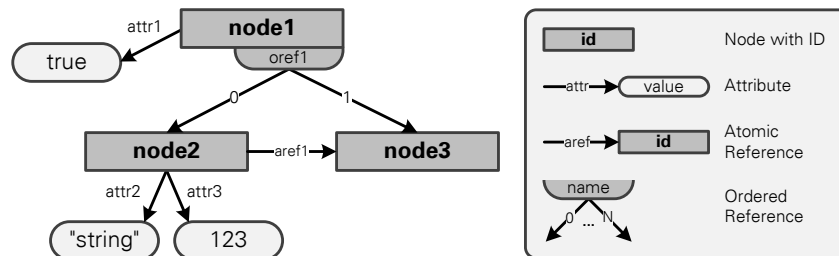


Figure 4.3: Example of a Data Model Instance in SAP Gravity

### 4.3.3 Primitive and Complex Operations

Modifications on a shared data model (see Section 4.3.2) are represented via six primitive operations, which are generated by the operation generator (see Figure 4.2). Table 4.2 lists the operations by their operation signature accompanied with a short description.

Operation Signature	Description
<code>addObj(id)</code>	Adds a new node with the identifier <code>id</code> .
<code>delObj(id)</code>	Removes the object with the identifier <code>id</code> .
<code>upd(o, a, v, w)</code>	Sets the attribute value for attribute <code>a</code> of object <code>o</code> to the value <code>w</code> . The old value <code>v</code> is replaced.
<code>setRef(o, r, p, q)</code>	Sets an atomic reference <code>r</code> from <code>o</code> to <code>q</code> . The old target <code>p</code> is replaced.
<code>addRef(o, r, p, i)</code>	Adds a link from node <code>o</code> to <code>p</code> at position <code>i</code> in the ordered reference <code>r</code> .
<code>delRef(o, r, p, i)</code>	Removes the link from node <code>o</code> to <code>p</code> at position <code>i</code> from the ordered reference <code>r</code> .

Table 4.2: Primitive Operations in SAP Gravity

Applications usually provide application-specific operations to change the application state. The task list application introduced in Section 1.2 would provide operations to create new task items, to rename existing task items, to change the priority of tasks and to mark task items as completed. Since the Gravity OT implementation only offers a graph-based data model (see Section 4.3.2) and the operations introduced before, the application-specific operations have to be translated when a Gravity data model is used to represent the application state. All application-specific operations would be mapped to a set of primitive operations accommodated by an application-agnostic complex operation as depicted in Figure 4.4. Complex operations act on behalf of application-specific operations and are target to transformation by the OT implementation.

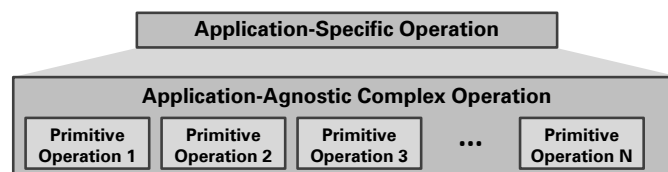


Figure 4.4: SAP Gravity Mapping from Application-Specific to Application-Agnostic Operations

The creation of a new task item might be mapped to the complex operation depicted in Equation 4.1. In the equation  $Op1; Op2$  indicates that operation  $Op1$  is executed before operation  $Op2$ .

$$\begin{aligned}
 \text{complexOp}(\text{"addTask"}) &= \text{addObj}(\text{"t3"}); \\
 &\quad \text{addRef}(\text{"root"}, \text{"tasks"}, \text{"t3"}, 2); \\
 &\quad \text{upd}(\text{"t3"}, \text{"name"}, \text{undefined}, \text{"TaskThree"}) \\
 &\quad \quad \text{upd}(\text{"t3"}, \text{"priority"}, \text{undefined}, 25); \\
 &\quad \quad \text{upd}(\text{"t3"}, \text{"done"}, \text{undefined}, \text{false})
 \end{aligned}
 \tag{4.1}$$

Because all seven primitive operations are issued on behalf of the application-specific operation  $\text{addTask}(\text{"TaskThree"})$ , atomicity for the set of primitive operations depicted in Equation 4.1 is ensured by the encapsulation in a complex operation.

### 4.3.4 Client API

The Gravity client API provides methods to handle the lifecycle of a collaboration session, to obtain information about participants in a collaboration session as well as methods to manipulate shared data.

#### Methods to Control the Lifecycle of a Collaboration Session

Methods to control the lifecycle of a collaboration session include for example the function `createCollaboration()` to create a new collaboration session object. This object acts as starting point for all interactions with the API. Joining a created session requires the invocation of the method `session.join(modelId, userDetails)`. The parameter `modelId` is a unique identifier of the model that shall be manipulated in the session. Details about the joining user are encapsulated by the `userDetails` parameter and might comprise a display name, an e-mail address as well as a thumbnail URL. To finally leave a collaboration session the method `session.leave()` can be used.

#### Methods to Handle Participants

Besides the lifecycle methods, the client API provides methods to obtain information about the participants involved in a collaboration session. The local participant of a collaboration session is returned via `session.getLocalParticipant()`, whereas `getAllParticipants()` returns an array containing all participants of the session. Every participant provides methods like `participant.getDisplayName()` to obtain the externally visible name of the participant, etc. Since every participant is associated with a unique color code once a collaboration session is joined for the first time, `participant.getColor()` can be used to retrieve the color code of the participant.

Gravity Client API Method	Primitive Operation	Model Change Event
<code>model.addNode(id)</code>	<code>addObj(id)</code>	<code>nodeAdded(node)</code>
<code>model.deleteNode(id)</code>	<code>delObj(id)</code>	<code>nodeRemoved(node)</code>
<code>node.setAttribute(name, value)</code>	<code>upd(o, a, v, w)</code>	<code>attributeSet(node, name, newValue, oldValue)</code>
<code>node.setAtomicReference(name, targetNode)</code>	<code>setRef(o, r, p, q)</code>	<code>atomicReferenceSet(node, name, newTarget, oldTarget)</code>
<code>node.addOrderedReference(name, index, targetNode)</code>	<code>addRef(o, r, p, i)</code>	<code>orderedReferenceAdded(node, name, index, target)</code>
<code>node.removeOrderedReference(name, index, targetNode)</code>	<code>delRef(o, r, p, i)</code>	<code>orderedReferenceRemoved(node, name, index, target)</code>

Table 4.3: Mapping of API Method Calls to Primitive Operations and Model Change Events in SAP Gravity

### Methods to Manipulate Data and Fire Model Change Events

At the core of the Gravity client API are methods to manipulate the data model introduced in Section 4.3.2 and events to notify subscribers about changes in the shared data. Whenever a data modification is triggered via the client API, an operation and a model change event are generated. Table 4.3 shows the primitive operations explained in Section 4.3.3 together with the corresponding API calls and generated model change events.

To underpin the correspondence between API methods, primitive operations and model change events, Listing 4.1 displays the API calls leading to the primitive operations depicted in Equation 4.1.

```
taskNode = model.addNode();
rootNode = model.getNode("root");
rootNode.addOrderedReference("tasks", 2, taskNode);
taskNode.setAttribute("name", "Task Three");
taskNode.setAttribute("priority", 25);
taskNode.setAttribute("done", false);
```

Listing 4.1: Gravity Client API Calls to Create a Task List Item

To ensure atomicity (see Section 4.3.3), methods manipulating the shared data cannot be called directly on the model. Instead the model handler has to be leveraged. The model handler provides the method `changeModel(function(model) {...}, [description])` that enables the execution of a number of methods on the model instance of the ongoing collaboration session. The first parameter is a function encapsulating a set of methods changing the model. All methods within such a function are executed entirely or, in case of a conflict with another operation, not at all. The second parameter of the `changeModel` method is an optional short description of the complex operation. This kind of execution of methods is reminiscent of the command design pattern [GHJV95]. A command is reified as object and passed to a handler that executes the command. In case of Gravity the function serving as first parameter is the encapsulating object and the model handler is the executive instance.

During the execution of the primitive operations created on behalf of the API calls from Listing 4.1 the following model change events are created: `nodeAdded`; `orderedReferenceAdded`; `attributeSet`; `attributeSet`; `attributeSet`.

Besides the methods to manipulate the data model, the client API provides also operations to retrieve the current model state. An array containing the identifiers for all nodes in a data model can be retrieved by means of `listNodes()`. The method `getNode(id)` enables developers to retrieve a node specified by its identifier `id` from the model. Furthermore methods to get attributes or references are available (e.g. `getAttribute(name)`, `getAtomicReference(name)`, `getOrderedReference(name)`).

## Methods to Subscribe for Model Change Events

Local as well as remote changes to shared data are advertised via model change events (see Table 4.3) and callback functions can be registered to keep track of the emitted events. Callbacks that shall be registered are grouped into so called listener objects. A listener object can be registered via the model or model handler interface: `model.addModelListener(modelListener)` and `modelHandler.addModelListener(modelListener)`. The place of registration determines when the contained callback functions are invoked due to raised events.

If the listener object was registered at the model, the callbacks are called directly after any modification on the model. The disadvantage of the direct notification is visible in case of the API calls to add a new task item (see Listing 4.1). The callback `nodeAdded` would be executed before the attributes of the new node are set. If the listener object is registered at the model handler, listeners are notified whenever a complex operation was finished successfully. Therefore the model handler buffers all emitted events during the execution of a complex operation and after finishing all cached events are advertised in one batch. The generation order of the events is maintained. The advantage of this approach is that for example the addition of a task item would be announced only when the task item is configured entirely.

Figure 4.5 summarizes the behavior of both possibilities for listener object registration.

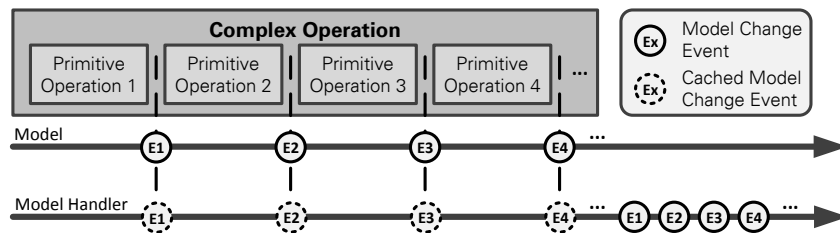


Figure 4.5: Model Change Event Handling in SAP Gravity

### 4.3.5 Synchronization Workflow

Applications built directly on top of Gravity adhere to the MVC architecture (see Section 3.2.1). The Gravity data model is used as data model for the application. Controller and View are implemented application-dependent. In the following the general synchronization and conflict resolution workflow of a Gravity-based task list application shall be exposed. Figure 4.6 depicts the workflow for local as well as remote modifications to the application state represented by a shared data model.

#### Processing of Local Modifications

On every modification of the application state triggered by the local user the following workflow is executed:

1. To add a new task item to the task list, the user interacts with the view of the task list application. On behalf of the user the view triggers some application-specific operation like `addTask("Task Three", 25, false)` on the controller (T1).

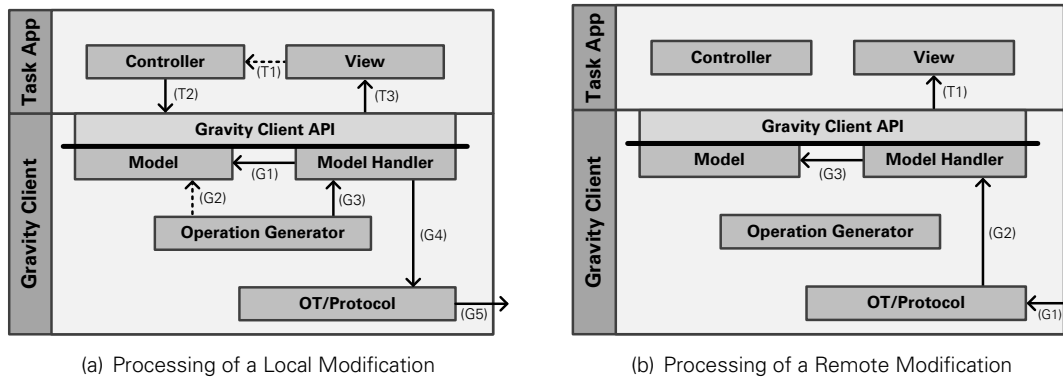


Figure 4.6: SAP Gravity Synchronization and Conflict Resolution Workflow

2. The controller maps the application-specific operation to an application-agnostic complex operation accommodating several primitive operations. To submit the complex operation the controller calls the corresponding methods (see Listing 4.1) via the model handler interface offered by the Gravity client API (T2).
3. Since atomicity has to be ensured for every complex operation, the model handler executes all model manipulations entirely or not at all (G1).
4. During the model manipulation the operation generator observes changes on the model (G2) and generates the corresponding operations (see Table 4.3). The generated complex operation for the addition of a third task item to the list can be seen in Equation 4.1.
5. If the complex operation could be executed successfully, the operation generator hands over the generated complex operation to the model handler (G3).
6. Afterwards the model handler passes the complex operation to the OT/protocol component (G4).
7. The synchronization protocol transfers the new complex operation to a Gravity server (G5).
8. Finally, the model handler notifies the application's view about the changes to the model via the Gravity client API (T3) by emitting model change events in correspondence to the model changes (see Table 4.3). More precisely the following sequence of model change events is created: `nodeAdded`; `orderedReferenceAdded`; `attributeSet`; `attributeSet`; `attributeSet`.

### Processing of Remote Modifications

Every remote modification of the application state leads to the following workflow:

1. Whenever the Gravity client receives a complex operation from the server (G1) it is handled first by the OT/protocol component. Because the remote operation might be in conflict with concurrent local operations, the remote operation is transformed against all concurrent local operations. This transformation might cause the rollback of conflicting local operations.
2. After transformation the transformed remote operation is handed over to the model handler (G2).
3. The model handler executes the remote operation on the model (G3) and afterwards notifies the application's view about the changes via model change events (T1). Thus, the application can react to the model changes properly.

## 4.4 SHARED JSON

Shared JSON is an extension of SAP Gravity that was introduced in Section 4.3 and enables the shared editing of JSON documents. Besides some foundations like the JavaScript Object Notation, the actual implementation will be explained in the following.

### 4.4.1 JavaScript Object Notation

The JavaScript Object Notation (JSON) is a text-based, language-independent data interchange format for structured data. The format was described initially in 2006 by Douglas Crockford in the Request for Comments 4627 [Cro06].

The format specifies four primitive data types: strings, numbers, Boolean and null. Besides the primitive types JSON comprises two structured data types: object and array. An object is an unordered collection of zero or more key-value pairs. Keys have to be strings, whereas values are of arbitrary data type. An array is just an ordered sequence of zero or more values. Values of any data type are referred to as JSON values in the following. A comprehensive description of the JSON data types and syntax by means of syntax diagrams can be found in Appendix A.

Revisiting the introductory example (see Section 1.2), Listing 4.2 shows a possible JSON document for the data of a task list containing two task items.

```
{
  "tasks": [
    {
      "name": "Task One",
      "priority": 80,
      "done": true
    },
    {
      "name": "Task Two",
      "priority": 50,
      "done": false
    }
  ]
}
```

Listing 4.2: JSON Representation of a Task List

The root of the JSON document is an object containing exactly one key value pair. The key is "tasks" and the value is an array containing the description of two task items. Every task item description is an object having three key-value pairs:

- The key "name" with a value of type string to represent the task's name,
- "priority" with a number value containing the task's priority, and
- "done" with a Boolean value to indicate whether the task is already finished or not.

### 4.4.2 JSONPath

JSONPath is a path-based addressing language reminiscent of XPath [CD99] that was devised for querying JSON values in JSON documents. The syntax of JSONPath is described in Listing 4.3 using the Extended Backus-Naur Form (EBNF) [EBN96]. Boldface strings correspond to non-terminal symbols whereas normal type indicates terminal symbols.



```

JSONPath = "/" | {pathPart}-;
pathPart = "/", (index | string);
index = 0 | 1 | 2 | 3 | ...
string = any unicode string without "/" or control characters

```

Listing 4.3: JSONPath Syntax Definition Using the EBNF [EBN96]

JSONPaths can be absolute or relative. All absolute JSONPath expressions start with the character "/" and are evaluated starting at the root element of every JSON document. Relative JSONPath addresses have no slash at the beginning and are evaluated from the current context. For example the relative path "0/name" in the context "/tasks" is equal to the absolute path "/tasks/0/name".

Sticking with "/" refers to the root of a JSON document. The root can be a structured JSON value like an object or array, or a primitive value like string, number, Boolean or null. To address a value inside a structured JSON value, the set of steps on the way to the value has to be specified separated by "/". For values in JSON objects a step is identified by the key of the desired key-value pair. In arrays steps are built using the indices of the favored element in the array.

Table 4.4 shows some JSONPath expressions accompanied by a description of the addressed document part based on the example JSON document from Listing 4.2.

JSONPath Expression	Addressed Document Part
"/"	The whole JSON document.
"/tasks"	The array containing all task items of the task list.
"/tasks/1"	The task item with index 1 in the array tasks.
"/tasks/1/name"	The value of the name property of the task item with index 1 in the array tasks.

Table 4.4: JSONPath Expressions and Corresponding Addressed Document Parts

### 4.4.3 JSON to Gravity Mapping

The Shared JSON implementation shall enable shared editing of JSON data. Rather than implementing a special-purpose OT algorithm for JSON data, Gravity shall be reused. Thus, JSON documents have to be mapped to the Gravity data model. In the following the mapping of all possible JSON data types is described.

#### Mapping of Primitive Data Types

The mapping of primitive JSON data types is depicted in Figure 4.7. Every JSON value adhering to a primitive data type is mapped to a node with two attributes. The attribute `type` is a string-based identifier determining the data type of the mapped value. The `value` attribute contains the actual JSON value.

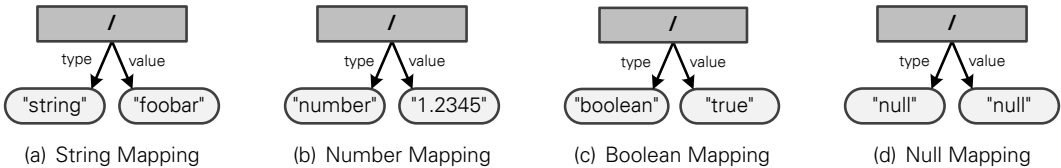


Figure 4.7: Gravity Mapping of Primitive JSON Data Types

## Mapping of JSON Objects

JSON objects (see Figure A.2) are mapped as depicted in Figure 4.8. Every object is mapped to a node having exactly one attribute `type` with the value `"object"` to indicate that an object was mapped. For every key-value pair in the object one atomic reference is created. The name of the reference corresponds to the key. Since keys inside an object have to be unique it is ensured that the names of the atomic references are distinct too. The root node of the mapped value serves as target of the atomic reference. The identifiers of the nodes are equal to the JSONPath (see Section 4.4.2) that might be used to address the JSON value represented by the subtree under the particular node.

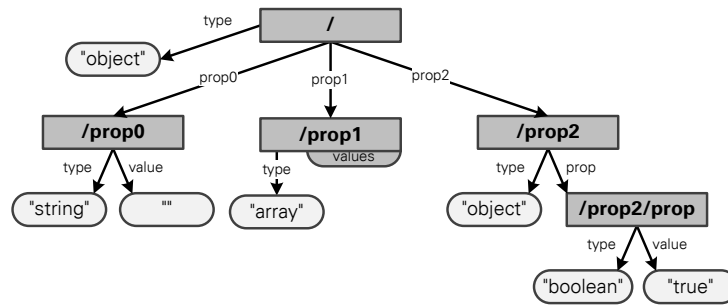


Figure 4.8: Gravity Mapping of JSON Objects

## Mapping of JSON Arrays

Figure 4.9 depicts the mapping of JSON arrays (see Figure A.3). Arrays are mapped to a node having an attribute `type` with the value `"array"` to indicate that this node is the root node of a mapped array. Moreover the node has an ordered reference named `values` accommodating references to the root nodes of the mapped array elements. For every value in the array, a reference is added to the ordered reference at the position equal to the array element's index.

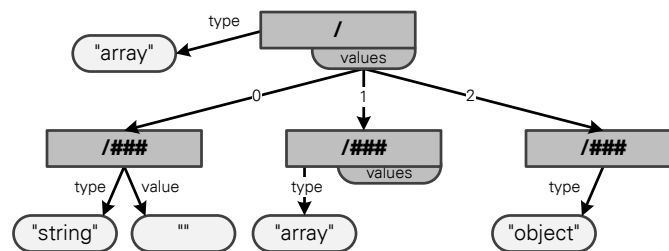


Figure 4.9: Gravity Mapping of JSON Arrays

The identifiers of the nodes usually correspond to the JSONPath (see Section 4.4.2) that might be used to address the JSON value represented by the subtree under the node. The only exception are nodes representing array elements. The JSONPath path part addressing the first item of an array would usually be `"/0"`, but instead a randomly hashed identifier indicated in the figure by `"/###"` is used. This is necessary because the concurrent editing of JSON documents shall be supported. If pure JSONPath expressions would be used as identifiers and two participants would insert different JSON values at the same position of the very same array at the same time a conflict would be recognized by Gravity and one of the users actions would be rolled back. Consider the following example:

- Alice as well as Bob would like to insert a new element at position 0 to an empty array. Alice wants to insert the string `"foo"` whereas Bob wants to insert the number `123`.

- Due to the mapping of JSON to Gravity, this would lead to the following calls on the Gravity client API (see Section 4.3.4) if pure JSONPath expressions are used for the node identifiers:
  - Alice:
 

```
node = model.addNode("/0"); node.setAttribute("type", "string"); ...
```
  - Bob:
 

```
node = model.addNode("/0"); node.setAttribute("type", "number"); ...
```
- Since different values are set to the attribute `type` of the very same node, the complex operations of Alice and Bob would be in conflict. Therefore one of the operations would be rolled back by the Gravity OT implementation. Hence, the intention of Alice or Bob would not be preserved.
- If hashed identifiers for the indices in the array are used the two nodes representing the array elements would get different identifiers and therefore no conflict would occur since the complex operations would be transformed successfully.

Therefore every node belonging to the representation of an array element has a hashed part in the identifier. Therefore every JSON value inside an array might be addressed by two different path expressions: The JSONPath expression (e.g. `"/tasks/0"`) referred to as *external path* in the following, as well as by the identifier of the node that is the root node of the element's Gravity representation (e.g. `"/tasks/###"`) where `###` stands for the hashed identifier. The latter path expression is referred to as *internal path* in the following. While both of the identifiers might be used to address an array element, the characteristics are different. Internal paths are stable during the lifetime of an array element. Due to concurrent insertions into and deletions from an array the actual position of an array element might change and therefore its external path might also change over time.

## Data Mapping Example

The full mapping of the JSON document from Listing 4.2 is presented in Figure 4.10. The root node is of type `"object"` and holds one atomic reference pointing to the node representing the array containing all task list items. The node `"/tasks"` has an ordered reference named `"values"` with two referenced nodes representing the two task items. Every task item node has the type `"object"` and three atomic references representing the three object properties `name`, `priority` and `done`. To emphasize the difference between internal and external paths consider the paths of the first array item. The internal path is `"/tasks/#0"` whereas the external path in the current document state is `"/tasks/0"`.

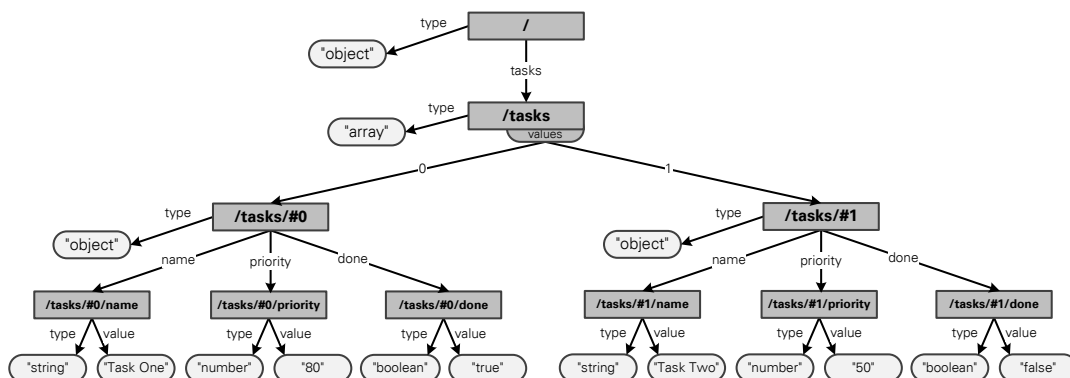


Figure 4.10: JSON to Gravity Mapping Example

## Operation Mapping Example

Due to the mapping of JSON documents to the Gravity data model, every modification to a shared JSON document has to be reflected in the Gravity data model. If for example a new task item (see Listing 4.4) shall be added to the JSON document from Listing 4.2 the Gravity client API calls depicted in Listing 4.5 have to be executed to update the mapped data in the Gravity data model.

```
{
  "name": "Task Three",
  "priority": 25,
  "done": false
}
```

Listing 4.4: JSON representation of a New Task Item

```
modelHandler.changeModel(function(model) {
  var tasks = model.getNode("/tasks");
  var taskNode = model.addNode("/tasks/#2");
  tasks.addOrderedReference("values", 2, taskNode);

  var nameNode = model.addNode("/tasks/#2/name");
  nameNode.setAttribute("type", "string");
  nameNode.setAttribute("value", "Task Three");
  taskNode.setAtomicReference("name", nameNode);

  var priorityNode = model.addNode("/tasks/#2/priority");
  priorityNode.setAttribute("type", "number");
  priorityNode.setAttribute("value", 25);
  taskNode.setAtomicReference("priority", priorityNode);

  var doneNode = model.addNode("/tasks/#2/done");
  doneNode.setAttribute("type", "boolean");
  doneNode.setAttribute("value", false);
  taskNode.setAtomicReference("done", doneNode);
});
```

Listing 4.5: Gravity Client API Calls to Add a New Task Item

### 4.4.4 Client API

The Shared JSON client API enables the concurrent editing of shared JSON documents. Therefore the API provides methods to initialize a collaboration session, to modify the document as well as to keep track of changes indicated by advertised events. The implementation of all API methods relies on the Gravity client API (see Section 4.3.4).

#### Methods to Handle Collaboration Lifecycle and Participants

The Shared JSON implementation procures a facade object acting as single interaction point for developers. The method `initJSONAdapter(initParams)` enables the developer to create an instance of this root object. Encapsulated in a JavaScript object the parameter `initParams` subsumes several configuration parameters explained in the following list:

- `initParams.gravityHost` contains a URL indicating the host name or IP address, and port of the Gravity server that shall be used. This parameter is mandatory.

- `initParams.modelId` is a required string identifier for the Gravity data model that shall be used. This parameter can be used to separate different collaboration sessions by specifying different model identifiers.
- `initParams.userId` is the unique identifier of the participant that is used to decide whether the participant is allowed to modify the shared data model. This parameter is required.
- `initParams.initData` is an optional JSON value that would be set as shared JSON document when the collaboration session is initially created.
- `initParams.onJoinedCallbacks` is an array of callback functions that are executed as soon as the collaboration session specified by `initParams.modelId` is joined successfully. The specification of callbacks is optional.
- If and only if the parameter `initParams.blockUserInteraction` is set to `true`, a modal window blocking all user interactions with the application is shown after the application is loaded. The window disappears once the collaboration session is initialized successfully.

During the creation of the root object the collaboration session is not joined automatically. Therefore all API calls modifying the shared JSON document are cached until the session is actually joined. Even if no connection to the server is established at this point in time, callback functions for Shared JSON events (see Section 4.4.5) can be registered.

To initialize the collaboration session the method `initializeCollaboration()` has to be called. A successfully initialized session triggers the following steps.

- If and only if `initParams.blockUserInterface` was set to `true`, the modal window is removed.
- All callback functions specified in `initParams.onJoinedCallbacks` are executed.
- The cached JSON data modification API calls are executed.

## Methods to Access and Modify Shared JSON Documents

Besides the lifecycle methods, the Shared JSON client API provides methods to query and modify a shared JSON document.

The method `getData()` returns the current state of the JSON document as JavaScript value. If the document is not initialized an exception of type `QueryError` is thrown. If the current JSON document shall be obtained as string value `getJSON()` can be used instead. This method throws also a `QueryError` if the model is not initialized. To retrieve single JSON values of a JSON document the method `getProperty(sPath, sContext)` is provided. This method returns the JSON value at the location addressed by the JSONPath specified via `sPath` and `sContext`. The parameter `sContext` is only necessary if `sPath` is a relative JSONPath (see Section 4.4.2). Assume that the JSON document in Listing 4.2 is the currently shared JSON document. Calling `getProperty` with the parameters `sPath="name"` and `sContext="/tasks/0"` would then return `"Task One"`.

The shared JSON document can be replaced with another document represented as JavaScript value leveraging the `setData(oData)` method. The `oData` parameter references a JavaScript value representing the JSON document to be stored. If the parameter contains no valid JSON document the method throws a `TypeError` exception. String-based JSON document can be stored via `setJSON(sJSONText)`. All modifications to a shared JSON document are achieved via the

method `setProperty(sPath, value, sContext, bSkipCallbacks, bOverwrite)`. This method can be used to modify primitive JSON values as well as objects and arrays. Key-value pairs inside an object, referred to as properties, or array elements might be inserted, deleted and replaced. If old values are replaced or deleted the method returns the old value. The following list explains the parameters of the method.

- The parameters `sPath` and `sContext` address the JSON value that shall be modified via JSONPath. If no context is provided via `sContext` the root context `("/")` is assumed. If the path that is passed is invalid a `QueryError` exception is thrown.
- The JSON value that shall be stored at the specified path is passed as JavaScript object in the `value` parameter. If a property shall be deleted `undefined` has to be passed as value.
- If and only if the optional parameter `bSkipCallbacks` is set to `true` all registered callback functions are not executed due to the changes of the shared JSON document by the current modification.
- If and only if `bOverwrite` equals `false` already set properties are not overwritten. Hence, array elements are shifted instead of being replaced. If an object property shall be replaced and `bOverwrite` equals `false` a `ParamError` exception is thrown.

## Methods to Subscribe for Shared JSON Events

During the modification of a shared JSON document several Shared JSON events (see Section 4.4.5) are advertised. To be notified about emitted events the Shared JSON implementation provides the possibility to register event listeners.

Listeners are registered with the method `registerListener(sPath, fCallbackFunction, bRecursive, sContext, bAllowPrereg)`. The following list explains the different parameters.

- Since Shared JSON event listeners are registered for a specific part of a JSON document, the parameters `sPath` and `sContext` contain a JSONPath expression addressing the actual part of the document. The path to be specified is an external path (see Section 4.4.3). To ensure that the listener is notified upon changes of the very same element over its whole lifetime, the unstable external path is immediately converted into a stable internal path. The internal path is used afterwards to determine whether the registered listener function has to be executed or not in case of changes to the shared JSON document.
- The callback function `fCallbackFunction` is called on every Shared JSON event that matches the specified path. Once executed, the listener functions receive an event object (see Section 4.4.5) as parameter that describes the JSON event that was emitted.
- The optional parameter `bRecursive` can be used to specify whether the registered listener should only be notified due to JSON events that are matching the specified path exactly (`false`) or also on JSON events with a more specific path (`true`). Revisiting the JSON document from Listing 4.2 the behavior of the parameter can be described as follows. A listener function registered at the path `"/tasks/0"` with `bRecursive=true` would be notified upon the addition or removal of key-value pairs to the first task item as well as the modification of the existing task properties. In contrast listeners registered with `bRecursive=false` at the very same path would only be notified on the addition or removal of key-value pairs to the task item.
- Basically the registration of listeners is only possible for valid JSONPath's, i. e. for paths that can be resolved in the current shared JSON document. To circumvent this restriction the optional parameter `bAllowPrereg` can be set to `true`.

Whenever a callback function was registered successfully an object is returned that provides the method `unregister()` to unregister the listener. Otherwise an exception is thrown.

#### 4.4.5 Shared JSON Events

Shared JSON events are emitted on all changes to a shared JSON document. Registered listeners are notified by calling the registered listener functions. Every listener receives an event object as parameter that describes the actual changes to the document. Listing 4.6 represents an example of an event object that might be passed to a listener registered for the JSONPath `"/tasks/"` to indicate the addition of a new task item to an array named `tasks`.

```

{
  type: "ADD",
  internalPath: "/tasks",
  externalPath: "/tasks",
  position: 2,
  newValue: {
    name: "Task Three",
    priority: 25,
    done: false
  }
}

```

Listing 4.6: Example of a Shared JSON Event Object

Every event object has a property `type` indicating the type of the change. In the example the type is `"SET"` standing for the addition of a new key-value pair to a JSON object or of an array element to a JSON array. Moreover, every event object accommodates the properties `internalPath` as well as `externalPath` indicating where the change in the JSON document occurred. In the example a new task item was added at index 2 to an array named `tasks`. The property `newValue` contains the JSON representation of the newly added element.

Due to the representation of shared JSON documents using Gravity data models, any modification on a shared JSON document can be tracked by means of Gravity model change events (see Table 4.3). Shared JSON events are then derived from the Gravity events. Since different modifications on shared JSON documents are possible different changes to the underlying Gravity data model have to be observed. Table 4.5 provides an overview about the possible Shared JSON event types, their properties as well as the underlying Gravity event that triggers the creation of the particular Shared JSON event. The registration point defines whether the corresponding Gravity model change callbacks are registered at the model handler or the model (see Section 4.3.4).

Event	Description	Properties	Gravity Event	Registration
SET	Indicates that a property was added to an object or replaced with a new value. In case of replacements before the SET event an event of type DEL is issued.	internalPath, externalPath, newValue	atomic Reference Set	model handler
DEL	Indicates that a property was deleted from an object.	internalPath, externalPath, oldValue	atomic Reference Set	model
ADD	Indicates that a new element was inserted into a JSON array.	internalPath, externalPath, position, newValue	ordered Reference Added	model handler
REM	Indicates that an element was removed from a JSON array.	internalPath, externalPath, position, oldValue	ordered Reference Removed	model

Table 4.5: Shared JSON Event Types with Corresponding Properties

In the following the entries in the table will be explained in detail. The Shared JSON event properties `externalPath` and `internalPath` provide the JSONPath as well as the identifier

of the node representing the property or array of the shared JSON document in the Gravity data model. In case of array modification events (ADD and REM) the parameter `position` indicates the index of the element inside the array. The properties `oldValue` and `newValue` contain the old or new value of the property, or the added or removed array item in case of array modifications. The shared JSON events SET and ADD are triggered based on Gravity events emitted by the model handler. This ensures that the Gravity model has reached a consistent state, i.e. the currently running complex operation on behalf of a Shared JSON operation has been finished. Thus, it is ensured that new values are fully configured before the registered Gravity callbacks are executed. In case of DEL or REM events the Shared JSON events are triggered directly when the corresponding Gravity model change event is fired. This ensures that the data to be removed from a shared JSON document can be read a last time before it is finally deleted. Therefore the Shared JSON implementation has to ensure, that first the subtree in the Gravity model representing the value to be deleted is isolated from the rest of the JSON document representation.

#### 4.4.6 Synchronization Workflow

Instead of using the Gravity client API directly to implement a collaborative application, the Shared JSON client API might be used. Since the application data is represented in a valid JSON document, data interchange with other applications is fostered. In the following the synchronization workflow of a collaborative task list application (see Section 1.2) realized by means of the Shared JSON client API shall be explained. Figure 4.11 depicts the workflow for local as well as remote modifications to the application state represented by a shared JSON document.

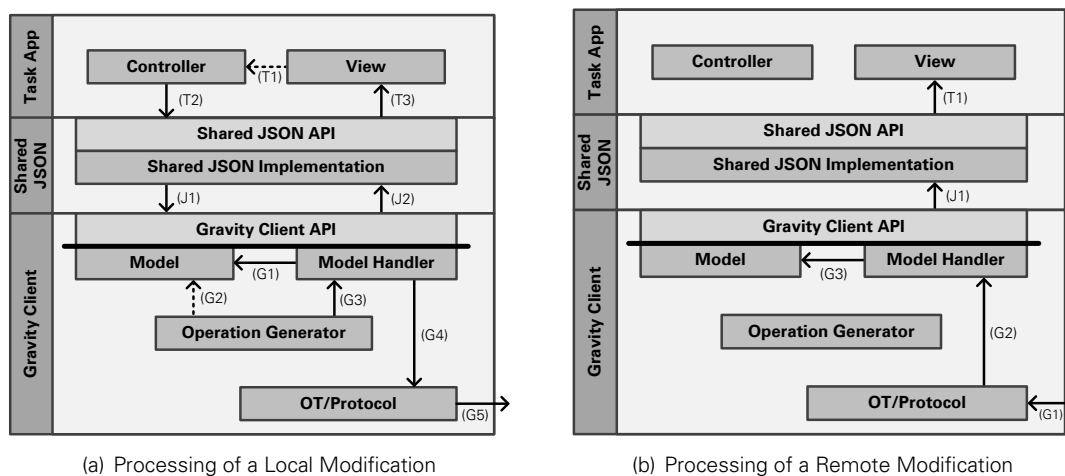


Figure 4.11: Shared JSON Synchronization and Conflict Resolution Workflow

#### Processing of Local Modifications

The synchronization workflow in case of a local application state manipulation is depicted in Figure 4.11(a). It is assumed that the current application state is represented by the JSON document in Listing 4.2.

1. If the user specifies a new task item via the UI of the application and presses the add button the application controller would be notified (T1).



2. The controller creates a JSON representation of the new task item (see Listing 4.4). This JSON representation is stored to the shared JSON document via `setProperty("/tasks/2", {"name":"Task Three", "priority":25, "done":false})` called on the Shared JSON API (T2).
3. The Shared JSON implementation is in charge of mapping modifications submitted via the Shared JSON API to Gravity client API calls that modify the Gravity data model. In case of the addition of an extra task item the Shared JSON implementation maps the operation to the set of Gravity client API calls presented in Listing 4.5 and dispatches a new complex operation via the Gravity Client API (J1).
4. The workflow inside the Gravity Client (G1, G2, G3, G4, G5) is similar to the pure Gravity case described in Section 4.3.5.
5. After Gravity successfully handled the complex operation it will fire several model change events. Since Gravity model change events trigger the creation of high-level Shared JSON events (J2), the Shared JSON event depicted in Listing 4.6 will be emitted.
6. The emitted JSON event can be used to update the view of the application properly (T3).

### Processing of Remote Modifications

The synchronization workflow in case of a remote application state manipulation is depicted in Figure 4.11(b).

1. Whenever the Gravity Client receives a complex operation it is processed. The processing steps (G1, G2, G3) are similar to the ones described in Section 4.3.5.
2. Gravity model change events (J1) are processed by the Shared JSON implementation resulting in Shared JSON events describing the changes to the shared JSON document in detail.
3. Finally the Shared JSON implementation notifies registered listeners about the new Shared JSON events. These events can be used to update the application's view according to the changes (T1).

## 4.5 SAPUI5 COLLABORATION ADAPTER

The SAPUI5 collaboration extension was outlined in Section 4.2. On top of the Shared JSON implementation a framework-specific collaboration adapter bridges the gap between the framework-based application data model and a shared JSON document. The implementation of this SAPUI5-specific collaboration adapter as well as its utilization shall be described in the following.

### 4.5.1 Implementation Structure

Because SAPUI5 provides access object as well as remote proxy-based data models, the collaboration adapter is implemented as collaboration proxy wrapping existing data models of the framework. Different model implementations are available out of the box: a JSON model that stores JSON data (see Appendix A), a XML model storing XML data [BPSM<sup>+</sup>06] and the OData model that provides transparent access to data offered via the Open Data Protocol [Cor11].

Figure 4.12 depicts the class structure of the SAPUI5 JSON model with associated classes to realize the data binding mechanism and the classes for the collaboration proxy. All classes necessary for the collaboration proxy are aligned at the bottom line of the figure.

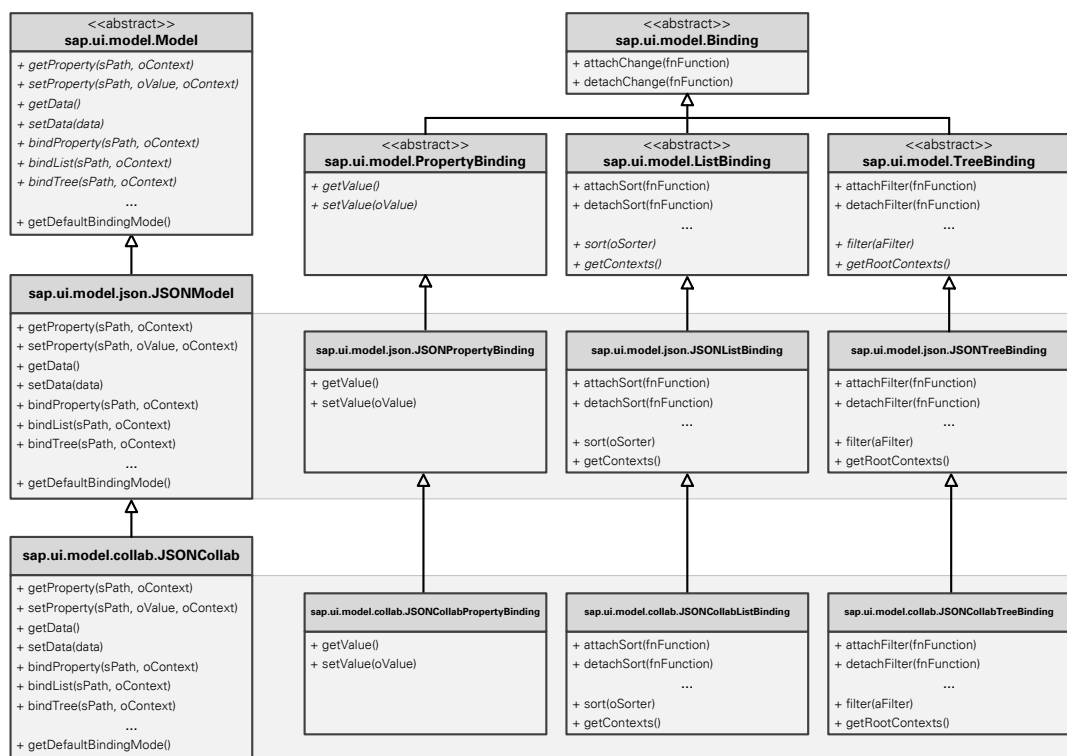


Figure 4.12: Class Structure of Collaborative SAPUI5 JSONModel with Associated Data Binding Classes

## Collaborative Data Model

Model implementations in SAPUI5 inherit from the abstract superclass `sap.ui.model.Model`. This class defines the interface implemented by actual model implementations and provides some default implementations. The set of methods comprises getter and setter for data like `getProperty(sPath, oContext)` and `setProperty(sPath, oValue, oContext)` where the parameter `sPath` specifies the path to the read or written property and `oContext` the context in which the path shall be resolved. The model implementation for JSON data is encapsulated by the class `sap.ui.model.json.JSONModel`.

As described in the conceptual architecture of a collaboration proxy (see Section 3.4.3), a proxy implementation wraps the actual implementation of a model and provides the same interface. Since different models (JSONModel, XMLModel, ODataModel) use different addressing expressions in the parameters `sPath` and `oContext` a collaboration proxy suitable for all model implementations would have to be able to handle all of these different addressing languages. Thus, no general purpose collaboration proxy is possible. Therefore a collaboration proxy has to be implemented for each of the models. Due to resource constraints only one exemplary implementation was provided for the JSONModel. JSON was selected because it is more lightweight compared to XML and the OData model is currently read-only.

The collaboration proxy for the JSONModel is represented by the class `sap.ui.model.collab.JSONCollab` that inherits from the original model implementation class `sap.ui.model.json.JSONModel`. The proxy implements exactly the same interface as the original model implementation and can therefore be used to replace an instance of the JSONModel in a transparent manner. Every method call on the proxy instance is either forwarded directly to the super class (for example in the case of `getDefaultBindingMode()`) or it is implemented within the proxy. The implementation of the proxy relies on the Shared JSON implementation (see Section 4.4). Any method call to get or set data is translated into a corresponding call on the Shared JSON client API manipulating a shared JSON document.

## Collaborative Bindings

SAPUI5 supports declarative data binding with the UI. Therefore every data model has to enable the creation of bindings that are an explicit representation of the connection between UI elements and model data. After any changes in their associated data model bindings are notified. Afterwards they check whether the change in the data model influenced the state of the binding. Whenever the binding state has changed all listeners of the binding are notified.

Bindings are created via `bindProperty()`, `bindList()` or `bindTree()`. These bindings are represented by instances of model-specific subclasses of the abstract classes `sap.ui.model.PropertyBinding`, `sap.ui.model.ListBinding` or `sap.ui.model.TreeBinding`. An instance of `JSONModel` for example would create instances of `JSONPropertyBinding`, `JSONListBinding` as well as `JSONTreeBinding`.

Since the original `JSONModel` returns a reference to the stored JSON data on any `getData()` or `getProperty(...)` method calls, but the `JSONCollab` implementation returns only a copy of the current data stored in the shared JSON document, the binding classes associated with the `JSONModel` have to be reimplemented for the `JSONCollab` proxy. On initialization bindings usually obtain a reference to their data of interest. Afterwards this reference is used during the whole lifetime of the binding. Since the data is only referenced, any changes to the data model are visible to the bindings automatically. Because the `JSONCollab` proxy deals with copies of the shared data, all bindings have to be reimplemented so that they retrieve a new copy of the data whenever they are notified about changes. Otherwise the binding would not be able to notice changes in the data. Therefore subclasses of the bindings for the `JSONModel` are implemented (`JSONCollabPropertyBinding`, `JSONCollabListBinding`, `JSONCollabTreeBinding`).

### 4.5.2 Application Integration

Because the collaboration proxy provides the very same API as the original model implementation, the integration of the collaborative data model into a SAPUI5 application is very easy. The class `JSONCollab` can be used instead of the class `JSONModel` without further efforts. Listing 4.7 shows the single-user JSON model as well as the collaborative JSON model configured to be used with a SAPUI5 application.

```
//////////////////////////////// single-user //////////////////////////////////
//create single-user JSON model instance
var model = new sap.ui.model.json.JSONModel();
//////////////////////////////// multi-user //////////////////////////////////
//create multi-user JSON model instance
var model = new sap.ui.model.json.JSONCollab("mySession");

//////////////////////////////// single-user / multi-user //////////////////////////////////
//set the data for the model
model.setData(data);
//set the model as core model
sap.ui.getCore().setModel(model);
```

Listing 4.7: Utilization of Single-User as well as Multi-User JSON Data Model in SAPUI5

In the single-user case first an instance of `JSONModel` is created. Afterwards the initial data values are set and the model is set as global data model for the application. This ensures that all bindings are resolved towards this data model. When the application shall be used collaboratively, the instance of `JSONModel` has just to be replaced with an instance of `JSONCollab`. The parameter of the constructor function is the identifier of the collaboration session that shall be joined. Afterwards the initial data can be set in the same way as in the single-user case and the collaborative JSON model can be used as global data model of an application.

### 4.5.3 Synchronization Workflow

Revisiting the introductory task list example (see Section 1.2), Figure 4.13 depicts the synchronization as well as conflict resolution workflow for a SAPUI5-based task list application.

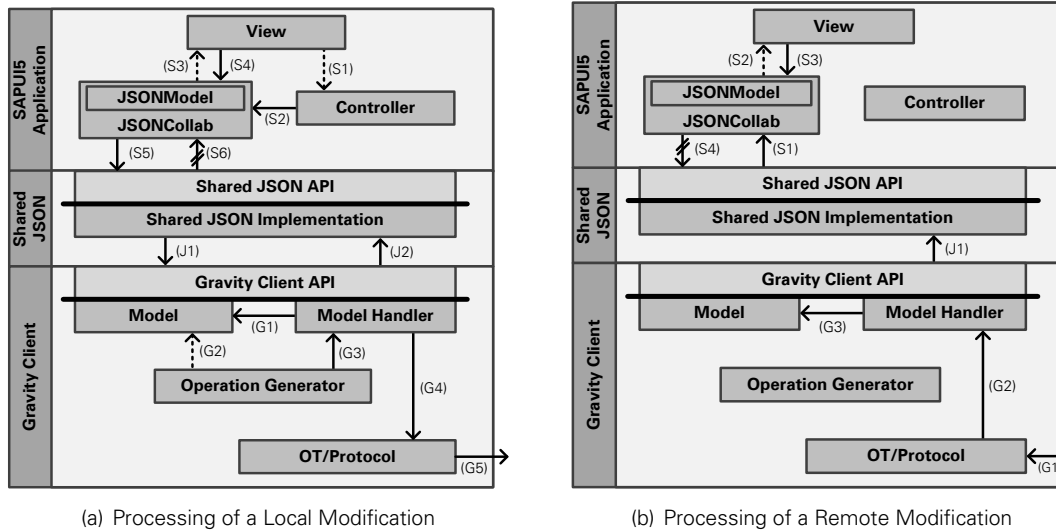


Figure 4.13: SAPUI5 Synchronization and Conflict Resolution Workflow

#### Processing of Local Modifications

Whenever a user of the SAPUI5-based task list application wants to add a new task item to the list of tasks, the name and priority of the new task item are provided via the user interface. Afterwards the following workflow is executed:

- Once the add button is pressed, the controller of the application is triggered (S1).
- The controller implementation obtains the values of the task properties (name, priority, etc.) from the corresponding input fields of the UI and creates a new task item representation in JSON. A possible JSON representation of a task item is depicted in Listing 4.4. This task item representation is stored in the data model by calling the method `setProperty` on the JSONCollab collaboration proxy (S2).
- The call is processed by the proxy implementation. First, the data is stored in the original JSON model. This model notifies the view accordingly (S3) and the view updates its state (S4). Afterwards the new data is stored to a shared JSON document via the Shared JSON client API (S5).
- Now the Shared JSON as well as the Gravity implementation processes the request (J1, G1, G2, G3, G4, G5, J2). The explanation of these steps can be found in Section 4.4.6.
- After the new data was stored successfully by the Shared JSON implementation, a Shared JSON event is emitted that notifies the JSONCollab proxy about the changes. If the event would be processed by the JSONCollab model implementation the announced modification in the data would be propagated again via the Shared JSON API resulting in an endless event loop. To avoid this cycle, the processing of the Shared JSON event (S6) has to be suppressed.

## Processing of Remote Modifications

Whenever a remote user modifies the shared data model, the following workflow is carried out:

1. Remote modifications are first processed by the Gravity as well as the Shared JSON implementation (G1, G2, G3, J1). Details of these processing steps can be found in Section 4.4.6.
2. After the modification was successfully incorporated into the shared JSON document, a Shared JSON event is generated and handled by the JSONCollab collaboration proxy implementation (S1).
3. On behalf of the event the original JSONModel is updated, leading to the update of the UI (S2, S3).
4. Finally, the changes to the JSONModel would be propagated to the shared JSON document via the Shared JSON client API resulting in an event loop. Therefore the propagation has to be inhibited (S4).

## 4.6 KNOCKOUT.JS COLLABORATION ADAPTER

The Knockout.js collaboration extension outlined in Section 4.2 comprises three parts. While the Gravity layer (see Section 4.3) as well as the Shared JSON layer (see Section 4.4) were already described, this Section shall describe the framework-specific collaboration adapter on top of the Shared JSON layer. Since Knockout.js applications comply with a subgraph-based data model structure (see Section 3.3.6), the collaboration adapter is connected with the data model using annotation-based code injection (see Section 3.4.3). Thus, synchronization code that keeps track of local changes and replays remote changes has to be injected into the view model.

### 4.6.1 Knockout.js View Model Structure

To be able to develop an algorithm injecting code in a view model that captures and replays changes, the view model structure is analyzed. View models in Knockout.js are simple JavaScript objects that might be created using the object literal `{ . . . }`. Listing 4.8 shows the view model definition for a simple task list application in the style of the introductory example (see Section 1.2). To simplify the example, task list items only have a name property whereas the priority and done flag are neglected.

```
var Task = function (data) {
    this.name = ko.observable(data.name || "default name");
}
Task.prototype.delete = function() {
    viewModel.tasks.remove(this);
}

var viewModel = {
    input: ko.observable(),
    tasks: ko.observableArray()
}

viewModel.addTask = function() {
    viewModel.tasks.push(new Task({name: viewModel.input()}));
    viewModel.input("");
}
```

Listing 4.8: Knockout.js View Model for a Task List Application

The `viewModel` variable references a simple JavaScript object serving as root object of the view model. This object holds a property `input` containing the input of the name input field in the view of the task list application. The property `tasks` is an array holding a list of task items. Moreover the view model accommodates a function `addTask` to add a new task item to the list of tasks. This function first reads the current value of the input field via `viewModel.input()`. Afterwards a new task item is created using the object constructor function `Task`. Object constructor functions are comparable to constructor functions in Java. Whenever the constructor is called a new instance of the class represented by the function is created. Properties and functions added to the prototype property of a constructor function like `Task.prototype.delete`, are added automatically to all instances of the particular class.

To notify the view about changes in the view model, all model elements have to be declared as observables. Observables are simple JavaScript properties enriched with a notification mechanism. The notification mechanism is used to inform subscribers about changes. In general, a Knockout.js view model can comprise three different types of observable properties: observables, observable arrays and computed observables.

## Observables

Primitive observables are the simplest form of observables representing a single value that can be observed. The value can be primitive (string, number, etc.) or structured (object, array). The current value of a primitive observable can be read by invoking the observable itself. The current value of the view model's input property can be read for example via `viewModel.name()`. A new value is written by calling the observable itself, but with the new value as parameter. Calling `viewModel.input("Task Three")` would set the input property of the view model from Listing 4.8 to "Task Three".

## Observable Arrays

Observable arrays can be understood as collection of things enabling the detection of changes. Observable arrays notify observers about changes to the array like the addition or removal of items. However, changes to the array elements itself have to be observed separately. From a more technical point of view observable arrays are simple observables with an array as value. Furthermore array specific operations to modify the elements of the underlying array are provided. Table 4.6 shows some examples for array specific functions.

Method	Description
<code>push(element)</code>	Adds an element to the end of the array.
<code>pop()</code>	Removes the last element from the array and returns it.
<code>unshift(element)</code>	Adds a new element as the first element to the array.
<code>indexOf(element)</code>	Returns the index of the first array element matching the passed element.
<code>remove(element)</code>	Removes all values that equal the passed element parameter from the array.

Table 4.6: Examples for Methods of Observable Arrays

## Computed Observables

Computed observables enable the computation of aggregated values based on other observables. The simplest example might be the full name of a person calculated using the first name as well as the last name. Computed observables are usually read-only and their value is updated whenever the underlying observables change. Knockout.js offers the possibility to register a write-callback function that is invoked whenever the computed observable is updated with a new value. This callback decomposes the joined value into its parts and saves them in the particular variables.

## 4.6.2 Knockout.js Synchronization Extension

Since Knockout.js observables provide features to notify observers about changes this subscription mechanism is used to keep track of changes inside the view model and synchronize them among different instances. Thus, listeners that store changes to a shared JSON document via the Shared JSON client API have to be registered with any observable.

Knockout.js provides the concept of *Extenders* to enrich observables with additional functionality. Extenders are created by adding an extender function to the global object `ko.extenders`. This function takes the observable as first argument and any additional information (string, number, object, etc.) as the second parameter. The return value can be the enriched observable itself or anything else that shall be used instead of the enriched observable. Listing 4.9 shows the source code of a synchronization extension for a simple observable.

```
ko.extenders.syncChange = function(target, path) {
  //observe value changes of the observable
  target.subscribe(function(newValue) {
    //store new value of observable via Shared JSON client API
    sharedJSON.setProperty(path, newValue);
  });

  //register change listener to the corresponding shared JSON event
  sharedJSON.registerListener(path, function(event) {
    if (event.type === "SET") {
      target.call(null, event.newValue);
    }
  })

  //store the initial value
  sharedJSON.setProperty(path, target());

  return target;
};
```

Listing 4.9: Knockout.js Synchronization Extension for Observables

If the extension is added to an observable, first a listener is subscribed. This listener keeps track of changes and whenever the observable changed, the new value is stored in the shared JSON document via `sharedJSON.setProperty(path, newValue)`. The parameter `path` specifies the JSONPath to the observable inside the Knockout.js view model. Since the view model might be a graph structure, an observable might be reachable via different paths from the root object of the view model leading to different JSONPath expressions. To ensure that every observable is reachable via exactly one JSONPath in the view model, the view model structure is restricted to tree-based view models. This limitation of the current approach is discussed in detail in Section 5.5.5.

Since remote changes of the observable have to be observed and integrated in the view model, a listener for Shared JSON events targeting the path of the observable is registered. Anytime an event of type "SET" (see Section 4.4.5) occurs, the observable is updated with the current value via `target.call(null, event.newValue)`. The invocation of the call function on the observable invokes the observable itself with `event.newValue` as parameter whereas the value of JavaScript's `this` variable is bound to `null`.

For observable arrays an extension in the same way was created. Because the callback function registered at the observable only gets the new array content as parameter, an algorithm has to check whether new elements were added or old ones were deleted from the array. The logic compares the current shared JSON document state with the state of the observable array. Two difference sets are obtained:  $A = \text{Array} \setminus \text{JSONArray}$  and  $B = \text{JSONArray} \setminus \text{Array}$  where  $X \setminus Y$  indicates the array difference between  $X$  and  $Y$ , i. e. all elements that are in  $X$  but not in  $Y$ . All elements from  $A$  have to be stored to the JSON document whereas the elements in  $B$

have to be deleted from the JSON document. The elements that were neither added nor deleted ( $Array \cap JSONArray$ ) are compared with the JSON document state with respect to their order. Changes in the order of elements have to be propagated too.

For computed observables no extension is necessary, because they depend on other observables. If the dependencies of a computed observable are tracked transitively, a set of observables and observable arrays would result as seed. Thus, it is enough to synchronize observables and observable arrays. Computed observables will be automatically recomputed on any changes of the underlying observables.

The observable and observable array Knockout.js extensions have to be added to all observables inside the view model that shall be synchronized. Therefore a special traversal function was devised that traverses a specified view model and adds the extenders. For example the algorithm would add an extender to the property name of the first task item in the view model from Listing 4.8 by calling `viewModel.tasks()[0].name.extend({syncExtension: "/viewModel/tasks/0/name"})`. Note the additional part `"/viewModel"` in the path passed as parameter to the extension. Since one application might comprise several view models responsible for different views, multiple view models have to be synchronized. To ensure that no interference is possible because for example two view models have the same property with different semantics, namespaces were introduced in the paths, so that different view models are separated by different prefixes.

### 4.6.3 Annotations

Since the integration of function calls to extend the view model with synchronization code requires scattered source code changes in the view model, a declarative approach was devised based on annotations that are used to inject the necessary code snippets automatically. The set of annotations encompasses: `@Session`, `@Class` and `@Sync`. To prevent JavaScript errors the annotations are always accommodated by comments. Thus, if the annotations are not evaluated the application still runs as single-user application. The view model from Listing 4.8 enriched with the annotations is depicted in Listing 4.10. Changes to the original view model are emphasized in bold face.

```
// @Session("mySession")

var Task = function (data) {
  this.name = ko.observable(data.name || "default name")
}
// @Class("Task");

Task.prototype.delete = function() {
  viewModel.tasks.remove(this)
}

var viewModel = {
  input: ko.observable(),
  tasks: ko.observableArray()
}
// @Sync("viewModel")

viewModel.addTask = function() {
  viewModel.tasks.push(new Task({"name": viewModel.input()}));
  viewModel.input("");
}
```

Listing 4.10: Knockout.js View Model for a Task List Application with Annotations



## @Session

The annotation `@Session(<sessionId>)` is used to establish a collaboration session. Different collaboration sessions are distinguished using unique session identifiers. This is necessary since different user groups might want to use the same application. Based on the identifier a corresponding Shared JSON as well as Gravity session is set up on behalf of the Knockout.js application. The `@Session` annotation will be replaced at runtime with the following function call: `com.sap.gcm.knockout.initSession(modelId: "<sessionId>");`.

## @Sync

The annotation `@Sync(<viewModel>)` marks the view model that shall be synchronized among all application instances. The parameter `<viewModel>` specifies the name of the variable that contains the view model. In case of the example the variable `viewModel` references the view model and the property `input` as well as the array `tasks` are synchronized. To ensure synchronization the evaluation of the annotation triggers the traversal of the view model. Hence, all observables are enriched with the synchronization extensions explained in Section 4.6.2. Every `@Sync` annotation is replaced at runtime with the following procedural JavaScript source code: `syncViewModel(<viewModel>, "<viewModel>");`

## @Class

The `@Class(<className>)` annotation marks object constructors. This is necessary to enable the recreation of objects remotely using the appropriate constructor function.

Objects in the view model might not be created just by means of object literals, but rather by a JavaScript constructor function. Objects constructed with the latter might have an associated JavaScript prototype object that contains properties and functions shared among class instances. If the synchronization extension (see Section 4.6.2) captures for example the addition of a task item to the array of tasks, the object has to be added to all task arrays at remote sites. This requires that the object is transferred to all other sites.

One possible solution would be the serialization and deserialization of the object including all properties and functions. Since the view models are synchronized by means of the Shared JSON implementation the stringification of functions would be required. This stringification using JavaScript's `toSource()` function is not feasible due to JavaScript's function scope. Consider the example in Listing 4.11.

```
Task = function(id) {
  this.getId = function() {
    return id;
  }
};

task = new Task("123");
task.getId(); //returns "123"

//stringify the object to reconcile the object
source = task.toSource();

//recreate the object
newTask = eval(source);
newTask.getId(); //ReferenceError: id is not defined
```

Listing 4.11: JavaScript Object Recreation Using Serialization

The listing shows the constructor function definition of a task item that has only an identifier with a corresponding getter function. Afterwards a new task item is created with the identifier "123" and stored in the variable `task`. Calling `getId()` on the task item would return the identifier "123". If we would use serialization to reconcile the objects the object would have to be transformed into a string representation that is stored in the variable `source` in the example. This string-based representation would be transferred to remote clients. Afterwards the incoming object would be reconstructed at the remote site using JavaScript's `eval` function. Calling `newTask.getId()` would result in a JavaScript exception `ReferenceError: id is not defined`. The reason for this behavior is that the variable is resolved in different scopes. During the initial creation of the task item, the `id` was resolved within the scope of the constructor function. When the function is recreated remotely `id` was resolved against the global JavaScript namespace.

To circumvent this problem, the original constructor functions are used to reconstruct objects remotely. Therefore a global mapping between unique keys and constructor functions is established. The collaboration adapter implementation maintains a global map that maps every class constructor function to a unique class name. This map is used to obtain a string-based identifier for the class of every object. Furthermore a reverse map is used to enable the easy look up of the constructor function for a given class identifier. Whenever an `@Class` annotation is used it is replaced with the following source code `registerClass(<className>, "<className>")` where the first parameter is the variable that contains the constructor function and the second parameter is the unique identifier for the class.

To reconstruct a class instance remotely the following workflow is carried out:

1. The synchronization extension of an observable array is notified of a new element added to the array.
2. The class constructor map is used to determine the corresponding unique key for the class instance.
3. Along with the JSON representation of the object, the class identifier is stored in the shared JSON document using a special property `GCMClass`.
4. Once the change is handled by remote clients the `GCMClass` property is evaluated. The corresponding constructor function is determined using the reverse mapping.
5. The constructor function is invoked with the data obtained from the JSON representation of the object as parameter. Therefore the constructor functions have to comply with a special parameter format that is discussed in Section 5.5.3.

### **Annotation-based Code Injection Mechanism**

To replace the annotations with actual source code all files containing annotations have to be processed. Therefore the file locations are specified in a configuration file and afterwards processed by a code injection mechanism according to the following steps:

1. Every file is loaded asynchronously using an `XMLHttpRequest`.
2. Once a file was loaded successfully, it is analyzed with respect to contained annotations. JavaScript's regular expressions are used to find and replace the annotations inside the source code. Table 4.7 provides an overview of the regular expressions that are used to find and replace the annotations.

Annotation	Regular Expression	Source Code Replacement
@Session	/\\\/@Session\ (" (\w+) "\)/	initSession(modelId: "<sessionId>");
@Sync	/\\\/@Sync\ (" (\w+) "\)/g	syncViewModel(<viewModel>, "<viewModel>");
@Class	/\\\/@Class\ (" (\w+[\.\w+]*) "\)/g	registerClass("<className>")

Table 4.7: Knockout.js Annotations with Regular Expressions and Source Code Replacements

#### 4.6.4 Property Exclusion Mechanism

Once a view model variable is marked with a @Sync annotation for synchronization all observables are extended with the synchronization extension. Since some of the observables control only the visibility of for example additional input fields or div containers these observables have to be excluded from synchronization. Otherwise changing the visibility of an input field in one application instance would cause this change in all other instances too.

To provide a convenient way for developers to exclude properties, a path-based property exclusion mechanism was devised. Within a configuration file paths with a language reminiscent of JSONPath (see Section 4.4.2) might be specified to exclude several view model properties from synchronization. Listing 4.12 shows the language definition for the property exclusion language using the EBNF notation [EBN96]. Boldface strings correspond to non-terminal symbols whereas normal type indicates terminal symbols.

```

exclusionPath = modelPart, pathPart ;
modelPart = "/", viewModelName;
pathPart = "/", ( propertyName | "*" | "/" );
propertyName = any unicode string without "/" or control characters
viewModelName = any unicode string without "/" or control characters

```

Listing 4.12: Property Exclusion Language Definition

Every exclusion path comprises a model part specifying the view model for which some properties might be excluded as well as at least one path part. Path parts might be the keys of some JavaScript object properties, array indices or one of the wildcards \* or /. The wildcard symbol \* indicates that any property name will match the path at the position of the wildcard. The wildcard symbol / indicates that an arbitrary number of path parts can lie between the wildcard and the appendix of the exclusion path.

Table 4.8 shows some examples of property exclusion expressions accompanied by a short description.

Exclusion Expression	Description
/viewModel/input	Excludes the observable input of the view model viewModel.
/viewModel/tasks/*/editing	Excludes the property editing of every task in the array tasks of the view model viewModel.
/viewModel//editing	Excludes all observables with the name editing in the whole view model viewModel.

Table 4.8: Examples for Property Exclusion Expressions

## 4.6.5 Application Integration

To describe the workflow of coupling an application with the Knockout.js collaboration extension, a simplified introductory application example where tasks only have a name is used. To enable real-time collaboration in an application, the following steps have to be followed:

1. The JavaScript file of the application containing the view model definition has to be enriched with annotations.
2. A configuration file has to be provided specifying the files that shall be analyzed regarding annotations, property exclusion expressions, etc.
3. The view definition of the application, which is usually a simple HTML file, has to import a JavaScript file accommodating the implementation of the Knockout.js collaboration extension. To actually incorporate the synchronization code into the view model, the collaboration extension uses the code injection mechanism explained in Section 4.6.3.

The incorporation of annotations inside applications' view model definitions was already described in Section 4.6.3 and the configuration file is just a simple text-based file. Listing 4.13 depicts an excerpt of an HTML page representing the UI definition for the task list application based on Knockout.js.

```
...
<!-- <script type="text/javascript" src="viewmodel.js"/> -->
<script type="text/javascript" src="knockoutadapter.js"/>
...
<input data-bind="value: name"/>
<button data-bind="click: addTask">Add Task</button>

<ul data-bind="foreach: tasks">
  <li>
    <span data-bind="text: name"></span>
    <a href="#" data-bind="click: delete">Delete Task</a>
  </li>
</ul>
...
```

Listing 4.13: Knockout.js View Definition

The main HTML elements are an input element to enter the name of a new task item, a button to add the new task as well as a list showing all tasks accompanied by a delete link. The data-bind attributes establish data bindings (see Section 3.3.5) to the view model (see Listing 4.10). Changes to link the collaboration extension with the Knockout.js application are limited to exchange the import of the file `viewmodel.js` (now encapsulated in the `<!-- / -->` tags) with the file `knockoutadapter.js`. This script imports the SAP Gravity as well as the Shared JSON code. Furthermore it contains the actual Knockout.js adapter implementation that connects the view model with a shared JSON document.

## 4.6.6 Synchronization Workflow

Figure 4.14 depicts the synchronization workflow for local as well as remote modifications to the application state of a task list application based on Knockout.js.

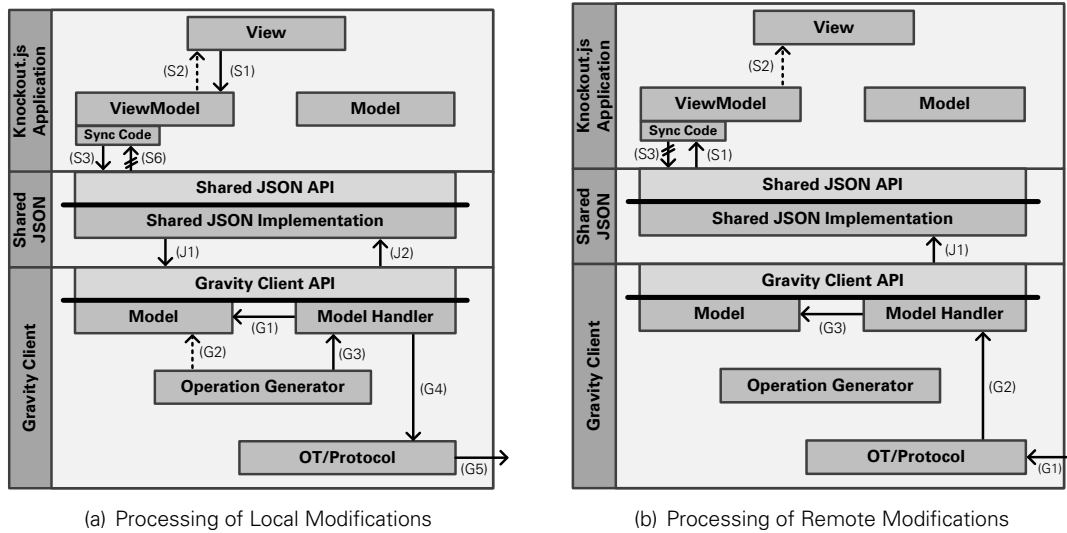


Figure 4.14: Knockout.js Synchronization and Conflict Resolution Workflow

### Processing of Local Modifications

The synchronization workflow in case of a local application state manipulation is depicted in Figure 4.14(a). Assuming that the task list application based on Knockout.js currently maintains two task items, the addition of a third task item would lead to the following workflow:

1. The user adds a task item to the tasks array by specifying the properties of the task and pressing the add button. Therefore the method `addTask` is called on the view model (S1).
2. The invocation of `addTask` creates a new instance of the class `Task` and adds it to the array `tasks`. Due to the two-way data binding of the view model with the view, the view is updated to show the new task item in the list (S2).
3. Since extenders were added to all observables, the change is captured and stored to the shared JSON document representing the current view model state via the Shared JSON client API (S3).
4. In the following the modification of the shared JSON document is handled as described in Section 4.4.6 by the Shared JSON as well as the Gravity implementation (J1, G1, G2, G3, G4, G5, J2).
5. After the request has been processed by Gravity as well as the Shared JSON implementation a Shared JSON event is fired. This event would be processed by the synchronization extension of the changed observables leading to an incorporation of the changes into the observables. This would trigger the change tracking again resulting in an endless event loop. Therefore the handling of Shared JSON events has to be inhibited in case of local manipulations (S6).

## Processing of Remote Modifications

The synchronization workflow in case of a remote application state manipulation is depicted in Figure 4.14(b). Any remote modification triggers the execution of the following steps:

1. Every incoming modification is processed first via the Gravity as well as the Shared JSON implementation (G1, G2, G3, J1). Details of these processing steps can be found in Section 4.4.6.
2. Once the processing finished, the Shared JSON implementation emits Shared JSON events that represent the changes in the shared JSON document (S1).
3. The Knockout.js synchronization extension incorporates the changes into the view model. Due to the declarative data binding the view is updated immediately (S2).
4. Since the incorporation of changes in the view model would trigger the listeners of the synchronization extension, the replay of remote modifications would trigger the propagation of the changes once again leading to an endless event loop. Therefore the capturing and propagation of changes in the view model has to be suppressed in the case of incoming remote modifications (S3).

## 4.7 CONCLUSION

Two frameworks should be selected to proof that the conceptual architecture of collaboration extensions for MV\* frameworks is feasible to enrich frameworks with support for the development of collaborative real-time applications. To select the frameworks, characteristics were inferred using the framework classification dimensions introduced in Section 3.3. Based on the derived characteristics SAPUI5 and Knockout.js were selected and served as basis for the prototypical implementation of collaboration extensions.

Based on the conceptual architecture, implementation architectures for both selected frameworks were described. The collaboration extensions were not implemented from scratch, especially no OT algorithm was implemented. Rather, SAP Gravity providing an OT implementation for directed, labeled and attributed graph structures was reused.

Since the Gravity client API offers only simple operations to modify a graph-based data model, a Gravity extension was devised. This extension, referred to as Shared JSON, provides a high-level API for the modification of a shared JSON document by multiple participants.

Based on the Shared JSON implementation a collaboration adapter bridging the gap between the data model of a SAPUI5 application and a shared JSON document was designed and implemented. Since SAPUI5 adheres to an access object or remote proxy-based data model structure, the collaboration adapter was conceived as collaboration proxy. Since the proxy offers the very same API as the original data model, it can replace a model instance in a single-user application transparently to enable real-time collaboration.

The Knockout.js collaboration adapter was also implemented based on the Shared JSON implementation. Since Knockout.js adheres to a subgraph-based data model, an annotation-based code injection approach was used to couple the data model of an application with a shared JSON document. The set of annotations comprises `@Session`, `@Class` and `@Sync`.

Even though only SAPUI5 and Knockout.js were enriched with collaboration support, these two prototypes proved that the implementation of collaboration extensions is feasible. Thus, the approach might be transferred to other JavaScript MV\* frameworks in the future.

# 5 EVALUATION

The previous chapter described the implementation of collaboration extensions for SAPUI5 as well as Knockout.js. But how do these solutions perform when used by developers creating collaborative web applications?

To assess the worth and merits of the developed collaboration extensions an evaluation was conducted determining the software quality compared to SAP Gravity. A common definition of software quality is given by the ISO/IEC 250xx series of standards. However, software quality models and associated characteristics are usually tied to end-user applications that have to be evaluated. Therefore two common quality models are adapted to the use case of framework evaluation in Section 5.1.

The evaluation of a framework or framework extension is a time-consuming task. Developers have to become familiar with the framework itself and different development tools. Finally, a full-fledged application has to be implemented. Since test persons could only be recruited at the SAP laboratory or at university, and the time those persons could spend for the study was limited, only the Knockout.js collaboration extension was evaluated and compared to SAP Gravity. The overall evaluation process is outlined in Section 5.2.

The evaluation process comprised two major steps: A preliminary user study with an experienced web developer illuminated in Section 5.3, and a comprehensive developer study in form of a practical course at the Dresden University of Technology. The latter is the main subject of Section 5.4.

Gaining experience with the development of applications using the prototypical implemented Knockout.js collaboration extension, several limitations were carved out that are explained in Section 5.5.

Finally, Section 5.6 summarizes the experiences of the evaluation.

## 5.1 SOFTWARE QUALITY MODELS

The ISO/IEC 250xx series of standards, referred to as "Software Product Quality Requirements and Evaluation (SQuaRE)", fosters the specification and evaluation of software quality requirements. Serving as umbrella document for a series of standards as well as guideline for software evaluation, the ISO/IEC 25000 standard [ISO05] defines software quality as the "capabil-

ity of software products to satisfy stated and implied needs when used under specified conditions." Since this definition is quite abstract, detailed quality models are defined in the standard ISO/IEC 25010 [ISO10]. These models specify concrete characteristics for the evaluation of software products that will be used to assess the Knockout.js collaboration extension (Knockout.js CE). Characteristics can be subdivided into subcharacteristics that might be separated further to measurable quality-related properties associated with quality measures. Furthermore guidelines for using the characteristics are offered.

The standard ISO/IEC 25010 [ISO10] defines two quality models: A product quality model concerning static properties of software as well as a quality in use model, which is related to the usage of a system in a particular context. Since all quality characteristics defined by the models are abstract, adaptation with respect to the Knockout.js CE was necessary. Furthermore quality characteristics of interest were selected, since a software product can be evaluated from different points of view. In case of the Knockout.js CE three different perspectives are possible:

1. The *framework developer* perspective, which emphasizes characteristics related to framework construction and implementation.
2. The *application developer* perspective, where framework characteristics that enable the effective and efficient development of framework-based applications are emphasized.
3. The *application user* perspective that assesses a framework-based application in use.

Because the main goal of the evaluation was to find out, whether the implemented Knockout.js CE fosters the effective and efficient development of web-based collaborative applications in a manner satisfying framework users, the application developer perspective was chosen. The next sections will explain the selection of evaluation characteristics in detail.

### 5.1.1 Product Quality Model

The product quality model defined in ISO/IEC 25010 [ISO10] revolves around the static quality of a software product. Figure 5.1 depicts the eight quality characteristics accommodated by the model. Since the Knockout.js CE was judged out of the application developer perspective, not all characteristics were assessed. Therefore neglected characteristics are indicated by strikethrough text.

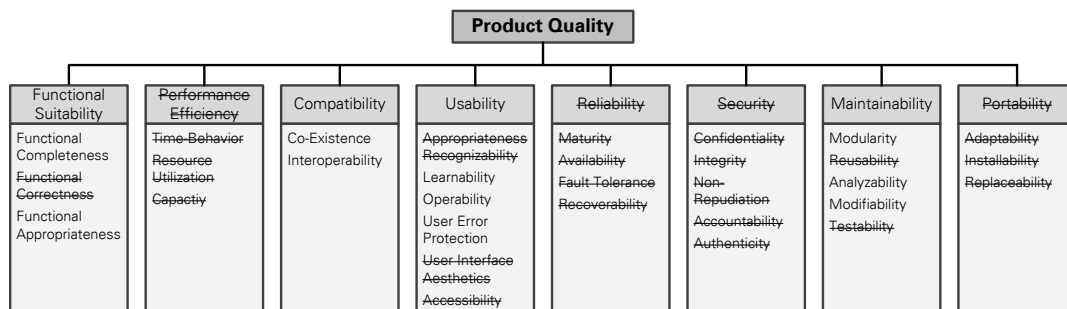


Figure 5.1: Product Quality Model According to ISO/IEC 25010 [ISO10]

**Functional Suitability** is the degree to which SAP Gravity/the Knockout.js CE provides the necessary functionality to develop a collaborative web application. Out of the three subcharacteristics of functional suitability, *functional correctness* was excluded, since an application developer is not responsible for the correct functioning of a framework. The adapted definitions of *functional completeness* and *functional appropriateness* are listed in Table 5.1.



Characteristic	Description
Functional Completeness	Degree to which SAP Gravity/the Knockout.js CE enables the development of arbitrary web-based collaborative applications.
Functional Appropriateness	Degree to which SAP Gravity/the Knockout.js CE facilitates the development of collaborative web applications.

Table 5.1: Subcharacteristics of Functional Suitability According to ISO/IEC 25010 [ISO10]

**Performance Efficiency** aims at assessing the performance of SAP Gravity/the Knockout.js CE relative to the used resources. Since the performance of a framework has to be enforced by framework developers rather than application developers, this characteristic was excluded from evaluation.

**Compatibility** is defined as the degree to which SAP Gravity/the Knockout.js CE can be used together with other technologies within the very same collaborative application. The subcharacteristics of compatibility accompanied by a short description are listed in Table 5.2.

Characteristic	Description
Co-Existence	Degree to which SAP Gravity/the Knockout.js CE restricts the choice of other technologies.
Interoperability	Degree to which SAP Gravity/the Knockout.js CE works well with other technologies.

Table 5.2: Subcharacteristics of Compatibility According to ISO/IEC 25010 [ISO10]

**Usability** is the degree to which SAP Gravity/the Knockout.js CE can be used to develop a collaborative application in an effective, efficient and satisfying manner. Out of the six subcharacteristics only the three listed in Table 5.3 were considered to be relevant. Usually evaluating if developers can decide whether a framework out of a plentitude of available frameworks is appropriate for their needs, the criterion of *appropriateness recognizability* is neglected since only the Knockout.js CE was evaluated and no comparable alternatives were available. The degree to which a user interface is recognized as pleasing and satisfying by users, *user interface aesthetics* were ignored since the Knockout.js CE has only a prototypical programming interface rather than an extensively designed user interface. *Accessibility* checks whether people with disabilities that might be caused by age as well as people with different backgrounds are able to use the Knockout.js CE. Since only students were available as human resources for the evaluation, accessibility was skipped.

Characteristic	Description
Learnability	Degree to which the use of SAP Gravity/the Knockout.js CE for the development of collaborative web applications can be learned in an effective and efficient manner.
Operability	Degree to which SAP Gravity/the Knockout.js CE is easy to use after the initial learning phase.
User Error Protection	Degree to which SAP Gravity/the Knockout.js CE prevents users from making errors.

Table 5.3: Subcharacteristics of Usability According to ISO/IEC 25010 [ISO10]

**Reliability** evaluates whether SAP Gravity/the Knockout.js CE can fulfill a specific level of service for a period of time. Since reliability has either to be evaluated by the framework developer for any use cases or with respect to a dedicated application, this characteristic was neglected in the evaluation from the application developer's view. The same holds for **Security**.

**Maintainability** is the degree to which SAP Gravity/the Knockout.js CE enables the effective and efficient maintenance of implemented applications. Out of the five subcharacteristics three were considered to be relevant and are listed in Table 5.4. *Reusability* targets the reuse of assets in other systems. Since SAP Gravity as well as Knockout.js are frameworks, reuse is inherent.

Thus, reusability was not evaluated explicitly. How test criteria for SAP Gravity/the Knockout.js CE can be established and how easy they can be ensured is covered by *testability*. Since testing a framework regarding functional correctness, security, etc., is in the responsibility of framework developers the criteria was also neglected.

Characteristic	Description
Modularity	Degree to which SAP Gravity/the Knockout.js CE helps to modularize applications in a way that changes to one component have minimal impact to other components.
Analyzability	Degree to which SAP Gravity/the Knockout.js CE enables developers to foresee the impact of intended changes in a collaborative application.
Modifiability	Degree to which an application based on SAP Gravity/the Knockout.js CE can be modified without introducing defects.

Table 5.4: Subcharacteristics of Maintainability According to ISO/IEC 25010 [ISO10]

**Portability** assesses whether SAP Gravity/the Knockout.js CE can be transferred to another operating system, platform, etc. Since framework developers are in charge of porting frameworks, this characteristic was ignored.

### 5.1.2 Quality in Use Model

The quality in use model focuses on the quality of a software product in concrete usage scenarios. The five characteristics of the model are depicted in Figure 5.2. Since the overall use case in the conducted evaluation was the development of a collaborative web application based on both SAP Gravity and the Knockout.js CE, not all characteristics were considered to be relevant. Neglected (sub-)characteristics are indicated by strikethrough text.

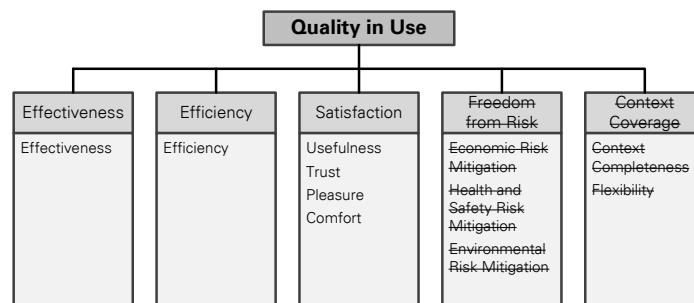


Figure 5.2: Quality in Use Model According to ISO/IEC 25010 [ISO10]

**Effectiveness** is the degree to which users can fulfill the functional requirements specified for a collaborative web application based on SAP Gravity/the Knockout.js CE.

**Efficiency** is characterized by the development time as well as the lines of code necessary to fulfill a specified set of functional requirements when developing a collaborative application based on SAP Gravity/the Knockout.js CE.

**Satisfaction** is the degree to which SAP Gravity/the Knockout.js CE satisfies users when leveraged for the development of a collaborative web application. The subcharacteristics of satisfaction are listed in Table 5.5.

**Freedom from Risk** usually contemplates potential risks to economic status, human life, health or the environment. Since the Knockout.js CE is only a prototype, which is not used in production, this characteristic was skipped.

Characteristic	Description
Usefulness	Degree to which users are satisfied with their perception of achieved goals when using SAP Gravity/the Knockout.js CE for the development of a collaborative application.
Trust	Degree to which users trust SAP Gravity/the Knockout.js CE to lead to the intended behavior.
Pleasure	Degree to which users are pleased when using SAP Gravity/the Knockout.js CE.
Comfort	Degree to which users feel comfortable when using SAP Gravity/the Knockout.js CE.

Table 5.5: Subcharacteristics of Satisfaction According to ISO/IEC 25010 [ISO10]

Because checking SAP Gravity as well as the Knockout.js CE for different use cases would require a comprehensive study with several different tasks residing in different domains the characteristic of **Context Coverage** was ignored too due to human resource constraints.

## 5.2 EVALUATION PROCESS

The main goal of the evaluation was to confirm assumed improvements of the Knockout.js CE compared to existing development approaches for collaborative applications. Because nowadays OT libraries are used to develop collaborative applications, the Knockout.js CE was compared to SAP Gravity with respect to the selected software quality characteristics introduced in Section 5.1. Hence, each characteristic was considered for both SAP Gravity and the Knockout.js CE.

As depicted in Figure 5.3 the evaluation was accomplished in two steps:

1. A minimal collaboration application was built. This application revealed a first impression of the effectiveness as well as efficiency of the Knockout.js CE compared to SAP Gravity. This preliminary study is described in Section 5.3.
2. To underpin the results of the preliminary study and to obtain further results in terms of the selected quality characteristics, a developer study was conducted. This study is subject to description in Section 5.4.

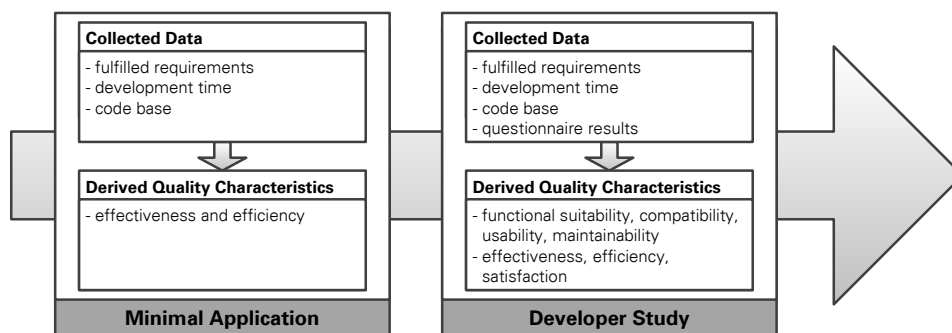


Figure 5.3: Overall Evaluation Process

## 5.3 MINIMAL APPLICATION

The development of a collaborative task list application similar to the introductory example (see Section 1.2) was carried out in three steps:

- First, a single-user version of the application was implemented based on Knockout.js [San12].
- Second, the single-user application was enriched with collaboration functionality using the Knockout.js CE (see Section 4.6).
- Finally, SAP Gravity (see Section 4.3) was used to enrich the single-user application with collaboration functionality once again.

All implementation tasks were fulfilled by an experienced developer familiar with Knockout.js, the Knockout.js CE and SAP Gravity.

### 5.3.1 Functional Requirements

Figure 5.4 depicts a screenshot of the task list application's user interface.

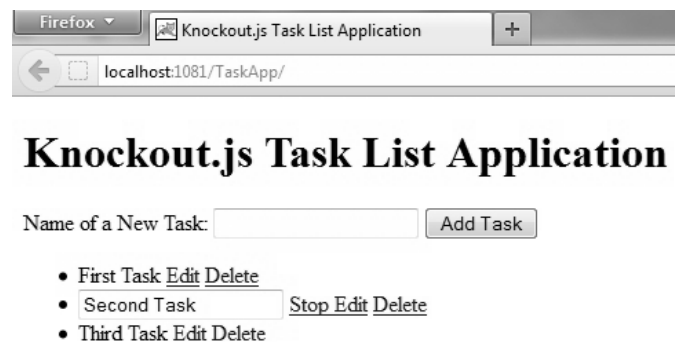


Figure 5.4: Screenshot of Simple Task List Application

The application met the following set of functional requirements:

- **R01:** The application shall display a list of tasks.
- **R02:** Every task shall have a name.
- **R03:** The name of each task shall be editable.
- **R04:** The dynamic addition of tasks to the list shall be possible.
- **R05:** Tasks shall be deletable.
- **R06:** All changes apart from inputs of a new task's name shall be synchronized in real-time.

### 5.3.2 Data Collection Techniques

The data collected during the development of the task list application comprises the time spent for the development as well as the application's lines of code in JavaScript and HTML. Furthermore the used annotations as well as lines for configuration were captured. Above all, the fulfilled functional requirements were determined.

Since counting lines of code is a cumbersome and error-prone task, the command line tool CLOC (Count Lines of Code)<sup>1</sup> was used. CLOC counts empty lines, comments and actual source code lines in over hundred different programming languages. Besides line counting, CLOC can be used to compute differences between two versions of a code base. The results are available in various formats encompassing plain text, XML, CSV, etc.

### 5.3.3 Results

Regarding the effectiveness of the Knockout.js CE compared to SAP Gravity, the developed task list applications revealed, that both approaches are suitable to meet the specified requirements described in Section 5.3.1.

The results valuable for assessing the efficiency are summarized in Figure 5.5. For both single-user and multi-user applications the total development times are shown. Moreover the figure presents the lines of code in HTML and JavaScript, the amount of annotations, and the lines spent for configuration of the Knockout.js CE.

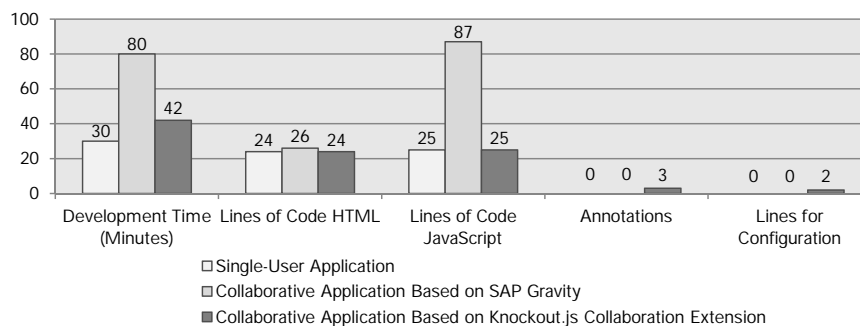


Figure 5.5: Evaluation Results of Minimal Task List Application

The total development time shortened from 80 minutes when using SAP Gravity to only 42 minutes in case of the Knockout.js CE. This is a reduction of 47.5 percent. It is also noteworthy that the development time for adding collaboration capabilities to the single-user application, calculated by total development time minus development time for the single-user application, was decreased by 76 percent (50 minutes vs. 12 minutes). Both approaches lead to an almost equal amount of lines of code in HTML, because only the amount of JavaScript file imports differs. The lines of code in JavaScript could be reduced from 87 lines necessary for Gravity to 25 lines in case of the Knockout.js CE. That is, the JavaScript code base was reduced by 71.3 percent. However, the Knockout.js CE added three additional annotations and two lines of configuration to the code base whereas SAP Gravity did not. We can thus confirm that the overall code base encompassing HTML, JavaScript, annotations and lines for configuration was scaled down from 113 lines using Gravity to only 54 lines for the Knockout.js CE. This equals an overall reduction of 52 percent.

<sup>1</sup><http://cloc.sourceforge.net/>

## 5.4 DEVELOPER STUDY

The developer study to elaborate the results from the preliminary study described in Section 5.3 was conducted in form of a practical course at the Dresden University of Technology. Former experiences of the participants with respect to the development of collaborative web applications are presented in Section 5.4.1. The actual task is described in Section 5.4.2 whereas the schedule of the internship is presented in Section 5.4.3. Subject of Section 5.4.4 are the used data collection techniques. Eventually the results of the developer study are described in Section 5.4.5.

### 5.4.1 Participants

The participants registered themselves for the course that was announced via the webpages of the chair of computer networks at the Technical University Dresden. Eight students studying computer science or computational engineering took part. To get an impression of the participants' experiences regarding the development of collaborative web applications, the questionnaire depicted in Figure B.1 was devised. The results are summarized in Table C.1. All of the participants had at least some first programming experiences. The programming languages used before the internship are for example Java, C, C++, C#, VisualBasic, PHP, etc. However, only three participants were experienced with the JavaScript programming language. Familiarity with HTML and CSS before the internship was claimed by four participants. None of the participants had already developed a collaborative web application before the internship.

### 5.4.2 Task Description

The collaborative application developed during the course represents a business analytic tool. Such an application, which is typically adopted by economists having to be aware of the current market situation, the strengths and weaknesses of their company, etc., is depicted in Figure 5.6.

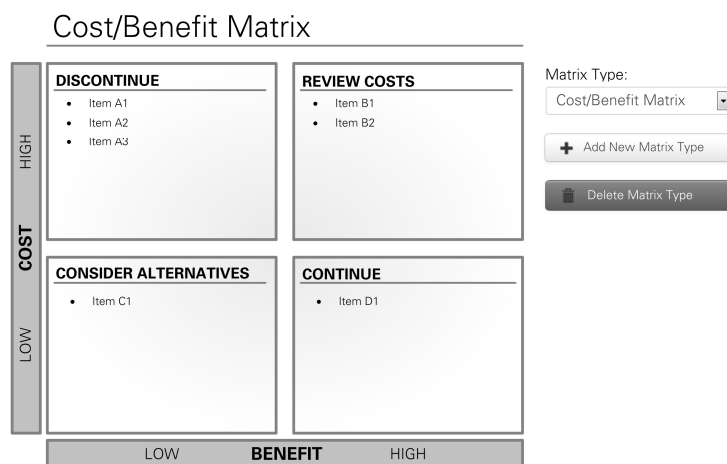


Figure 5.6: Mockup of Collaborative Business Analytic Tool

The tool can be visualized by means of a two by two matrix. This matrix supports the classification of items (e.g. products, stakeholder, etc.) along two dimensions (e.g. costs and benefits). Since every matrix cell is associated with a recommendation that describes how to handle the contained items, it is a valuable means for decision making in companies. Let's consider the concrete matrix example depicted in Figure 5.6, which can be used to conduct a cost-benefit analysis. The matrix dimensions are costs and benefits. Once all items are spread across the

matrix, the cells can be considered in detail. Some ideas will be too costly with little benefit and therefore should be abandoned. Other ideas should be reviewed whether the costs can be reduced. For others alternatives should be searched, whereas items with high benefit and reasonable costs should be considered further. In essence, the matrix can help to evaluate products in a company portfolio by weighing costs against benefits.

The collaborative version of this application shall fulfill the following functional requirements:

1. **R01:** A two by two matrix shall be visualized.
2. **R02:** The matrix shall have a heading.
3. **R03:** The two matrix dimensions shall have legends.
4. **R04:** Every matrix cell shall have a heading.
5. **R05:** The matrix and cells headings as well as the legends shall be editable.
6. **R06:** Every matrix cell shall be able to accommodate items.
7. **R07:** Users shall be able to add and delete items in the matrix cells.
8. **R08:** Every item shall have at least a name property associated.
9. **R09:** All properties of the items (e. g. name) shall be editable.
10. **R10:** Moving and reordering items by means of drag-and-drop shall be ensured.
11. **R11:** The addition and deletion of different matrix type presets shall be possible.
12. **R12:** All changes to the application state shall be synchronized in real-time and conflicts shall be resolved automatically.

### 5.4.3 Schedule

During the practical course the collaborative web application described in Section 5.4.2 was developed based on Knockout.js [San12] and one of the following development approaches for collaborative applications: the Knockout.js collaboration extension (see Section 4.6) and SAP Gravity (see Section 4.3). The schedule of the course is exhibited in Table 5.6.

Week	Description
1	Kickoff meeting
2	Introduction to task and schedule; Tutorial about Knockout.js
3	Development of the single-user application
4	Development of the single-user application
5	Submission of the single-user application; Tutorial about the Knockout.js collaboration extension
6	Development of the collaborative application using the Knockout.js collaboration extension
7	Submission of the first collaborative application; Tutorial about SAP Gravity
8	Development of the collaborative application by means of SAP Gravity
9	Development of the collaborative application by means of SAP Gravity
10	Final submission; Questionnaire

Table 5.6: Schedule of Practical Course for Developer Study

The practical course lasted ten weeks and followed a schedule based on the Scrum software development method [Sch95]. Before the course all participants were briefed to work alone, although they were allowed to discuss problems in the group or ask the supervisor for help. The whole course was divided into three major sprints: Within the first sprint the participants were asked to develop the web-based application described in Section 5.4.2 as single-user application.

In the second sprint the single-user application was enriched with collaboration capabilities by means of the Knockout.js CE. Finally, in the third sprint a collaborative application was developed again, but this time using SAP Gravity.

In order to establish a common understanding regarding the used technologies, all students received an introductory training at the beginning of every sprint. Lasting one and a half hours, the trainings introduced the technologies required to fulfill the task of the particular sprint.

After every sprint the participants had to submit their solutions including source code and a short document (see Section 5.4.4) listing the efforts and problems during the sprint.

### 5.4.4 Data Collection Techniques

During the course data was collected using several qualitative as well as quantitative sources. Documentation was solicited by the participants after every sprint, the source code of the developed applications was analyzed and finally a questionnaire was answered by every participant.

#### Sprint Documentation

To acquire measures for deriving the efficiency of the approaches as well as to expose limitations, the participants of the course were asked to document their efforts and problems. Figure 5.7 shows a template for such a document.

## Internship Documentation

---

Name:

#### Time Recording

In this section you should record your efforts that were necessary to fulfill the task. Please do the time recording as exact as possible and provide a short description of what you have done.

Time (in minutes)	Task Description

#### Problems

In this section you should record the problems you encountered during your development activities. Please describe the problem, when it occurred and how you solved it.

- 
- 
- 

Figure 5.7: Template for Sprint Documentation

The document contains two parts: In the first part the efforts for fulfilling the particular sprint are listed. Participants were requested to record the time as exact as possible and provide a short description what was achieved during the different periods. In the second part of the documentation, the participants should briefly describe encountered problems and corresponding solutions.



## Source Code Analysis

The submitted applications of the three sprints were analyzed regarding their fulfillment of the functional requirements (see Section 5.4.2) as well as the lines of code in JavaScript and HTML. Whether the applications fulfill the functional requirements was checked by running and testing the applications in use. Furthermore in case of the collaborative application based on the Knockout.js CE the number of annotations and lines used to configure the extension was evaluated. To analyze the source code the tool CLOC (see Section 5.3.2) was used.

## Questionnaire

To inquire the attitude of the participants in terms of the two used development approaches, a questionnaire was devised. The questionnaire accommodated an overall set of 34 statements, where 17 statements were targeted to each of the approaches. To ensure that the answers of the participants are comparable with respect to the approaches, the statements were designed to be equivalent.

Figure 5.8 shows the questions that were devised grouped in terms of the covered software quality characteristic (see Section 5.1).

Product Quality	Functional Suitability	(Q01) Gravity/the Knockout.js extension provides the necessary functionality to develop collaborative web applications.
		(Q02) The functionality of Gravity/the Knockout.js extension eases the development of collaborative applications.
	Compatibility	(Q03) Gravity/the Knockout.js extension restricted the choice of other technologies.
		(Q04) Other selected technologies worked well with Gravity/the Knockout.js extension.
	Usability	(Q05) I could easily learn how to use Gravity/the Knockout.js extension.
		(Q06) After the initial learning phase, Gravity/the Knockout.js extension was easy to use.
		(Q07) Gravity/the Knockout.js extension prevented me from making mistakes during the development of the collaborative web application.
Maintainability	(Q08) Gravity/the Knockout.js extension helped me to separate synchronization code from the rest of the application.	
	(Q09) The impact of changes made to the application's source code could be foreseen easily.	
	(Q10) The cause of failures could be reconstructed with modest effort.	
	(Q11) Parts of a developed application that have to be modified could be easily determined.	
	(Q12) The developed application could be easily enriched with additional application features without introducing defects.	
Quality in Use	Satisfaction	(Q13) Gravity/the Knockout.js extension is useful to develop collaborative applications.
		(Q14) The Gravity API calls and callbacks/the annotations lead to the expected behaviour.
		(Q15) The callback mechanism/the property exclusion mechanism is convenient.
		(Q16) It is very easy to use Gravity/the Knockout.js extension to add collaboration capabilities to an application.
		(Q17) Gravity/the Knockout.js extension is a comfortable means for developing real-time applications.

Figure 5.8: Questions to Participants of Developer Study

The full questionnaire is attached in Appendix B. Comprising several balanced five-point Likert items, the questionnaire represents a Likert scale [Lik32]. The items measure the degree of disagreement or agreement. Since five-point items with the meanings strongly disagree, disagree, neither agree nor disagree, agree, and strongly agree are used, the neutral position is an easy choice for participants that are unsure how to respond.

Gutwin and Greenberg discovered in one of their studies [GG98] that responses under between-subject analysis are lacking of significant distinction. This finding is probably sourced by the lack of a base for judgments. Therefore the questionnaire and its evaluation follow a within-subject design that allows participants to make comparisons when rating the approaches. Therefore every participant had to pass all three sprints of the course.

## 5.4.5 Results

In the following, the results obtained from the collected data (see Section 5.4.4) are described. First, effectiveness and efficiency of the Knockout.js CE compared to SAP Gravity are discussed. Afterwards, details on further software quality characteristics are exposed.

## Effectiveness and Efficiency

To reveal improvements regarding the effectiveness as well as efficiency when using the Knockout.js CE for the development of collaborative web applications instead of SAP Gravity, the implemented applications were investigated at runtime and the source code was analyzed. To interpret the data, it was aggregated first by calculating the means for the different measures like development time, lines of code in JavaScript, etc. Figure 5.9 summarizes the calculated mean values. Note that the mean values were calculated only from the results of seven participants, because one participant could not fulfill all functional requirements successfully and was therefore excluded from the source code analysis to ensure comparable results.

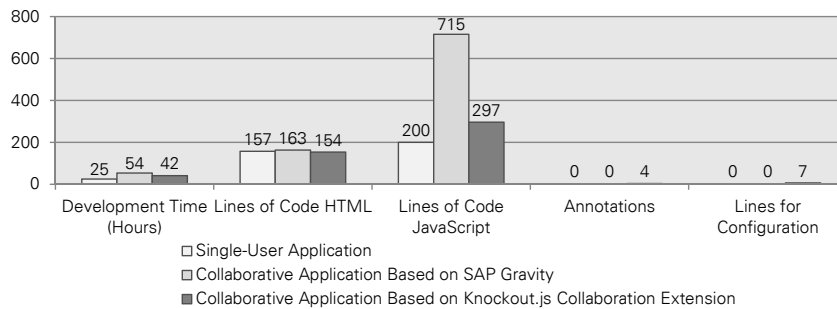


Figure 5.9: Source Code Results of the Developer Study

Regarding the effectiveness it can be stated, that all developed applications fulfilled the full set of functional requirements listed in Section 5.4.2. Thus, effectiveness is ensured by both development approaches.

The total development time for the collaborative application amounted to 54 hours when using SAP Gravity and 42 hours in case of the Knockout.js CE in average. That is the Knockout.js CE diminished the time by 22 percent compared to SAP Gravity. Note that the addition of real-time capabilities to the single-user application actually required 41 percent less time when using the Knockout.js CE (17 hours) instead of SAP Gravity (29 hours). Small differences in the lines of code in HTML were caused by different amounts of JavaScript file imports required for the approaches as well as by additional technologies used by the participants.

Let's consider the lines of code written in JavaScript. While SAP Gravity caused 715 lines to be written, the Knockout.js CE required 297 lines in average. Thus, the Knockout.js CE reduced the lines of code in JavaScript by 59 percent compared to SAP Gravity. However, the Knockout.js CE added in average four additional annotations and seven lines of configuration to the code base. Differences in the lines of code in JavaScript between the single-user application and the collaborative applications are source from modifications to meet the limitations explained in Section 5.5 in case of the collaboration extension and by application-specific adapter code to glue the application's view model to the Gravity client API. In summary the overall code base including HTML, JavaScript, annotations, and lines for configuration could be reduced from 878 lines to 462 lines by using the Knockout.js CE instead of SAP Gravity. This equates to an overall reduction of 47 percent.

## Further Software Quality Characteristics

Figure 5.10 shows the questionnaire results usable for evaluating the Knockout.js CE regarding further software quality characteristics besides effectiveness and efficiency. The left hand side shows the questions that were answered by the respondents grouped by the software quality characteristics described in Section 5.1. On the right side bar charts display the mean values

of the responses. The error bars in the charts do represent confidence intervals [SRL03] for a confidence level of 90 percent. Thus, the significance level is  $\alpha = 0.1$ . A confidence interval is calculated from the observations and determines the range of values which probably contains the mean value if more course groups were sampled and the mean would be calculated. These confidence intervals might be used to judge whether differences in the mean between SAP Gravity and the Knockout.js CE are significant or not [SRL03]. Clearly non overlapping confidence intervals indicate that the difference of the mean values is significant. If the confidence intervals overlap no conclusion is possible.

If only the mean values of SAP Gravity and the Knockout.js CE are compared it is obvious that the collaboration extension performs better than Gravity.

With respect to the **Functional Suitability** of both approaches the following conclusions can be drawn. SAP Gravity as well as the Knockout.js CE provides the necessary functionality to develop collaborative applications. Since the provided functionality of the Knockout.js CE eases the development of collaborative applications significantly compared to SAP Gravity, the functionality of the collaboration extension is judged to be more appropriate for the development of collaborative applications.

**Compatibility** was rated as follows. The respondents disagreed that SAP Gravity or the Knockout.js CE restricts the choice of other technologies. However, there is a trend that the Knockout.js CE worked better with other technologies than Gravity. Thus, it can be concluded that developers are free in their choice of other technologies when using Gravity or the Knockout.js CE.

Furthermore the Knockout.js CE showed some improvements regarding the **Usability**. The mean values regarding the learnability of the different approaches indicate that there is a flatter learning curve for the Knockout.js CE compared to SAP Gravity. However, after passing the initial learning phase both approaches were rated to be easy to use. In protecting users from errors the Knockout.js CE performed slightly better compared to SAP Gravity, but both approaches might be improved.

The **Maintainability** of applications built using the Knockout.js CE is also improved, since applications are modularized significantly better compared to SAP Gravity. This is sourced by the declarative nature of the annotation-based code injection used in the collaboration extension. How source code changes impact the application can be foreseen when using SAP Gravity almost as easy as with Knockout.js CE. Moreover the Knockout.js CE tends to ease the reconstruction of failure causes with modest effort. This might be due to the declarative nature of the Knockout.js CE too, since only the application's view model and business logic code has to be checked for errors. However, tracking failures is still a complex and cumbersome endeavor. Because the business logic is not interwoven with synchronization code in case of the Knockout.js CE, the look up of application parts that have to be modified in case of changes is easier. When application parts were determined successfully, both approaches enable the addition of new application features with modest effort.

In terms of **Satisfaction** the following conclusions can be drawn. While both approaches are judged to be useful for the development of collaborative applications, the Knockout.js CE tends to be more useful. The main development concepts of both approaches - the callbacks in SAP Gravity as well as the annotations of the Knockout.js CE - lead to the expected behavior. The same holds for the convenience of the callbacks of SAP Gravity and the property exclusion mechanism of Knockout.js CE. A significant difference between the approaches could be found regarding the ease of use. Participants rated that the Knockout.js CE is significantly better to use in contrast to SAP Gravity. The reason might be the use of annotations instead of procedural code, since only some minor source code changes to address the limitations of the Knockout.js CE (see Section 4.6) are necessary. Furthermore the Knockout.js CE was perceived as a significantly more comfortable means for the development of collaborative web applications.

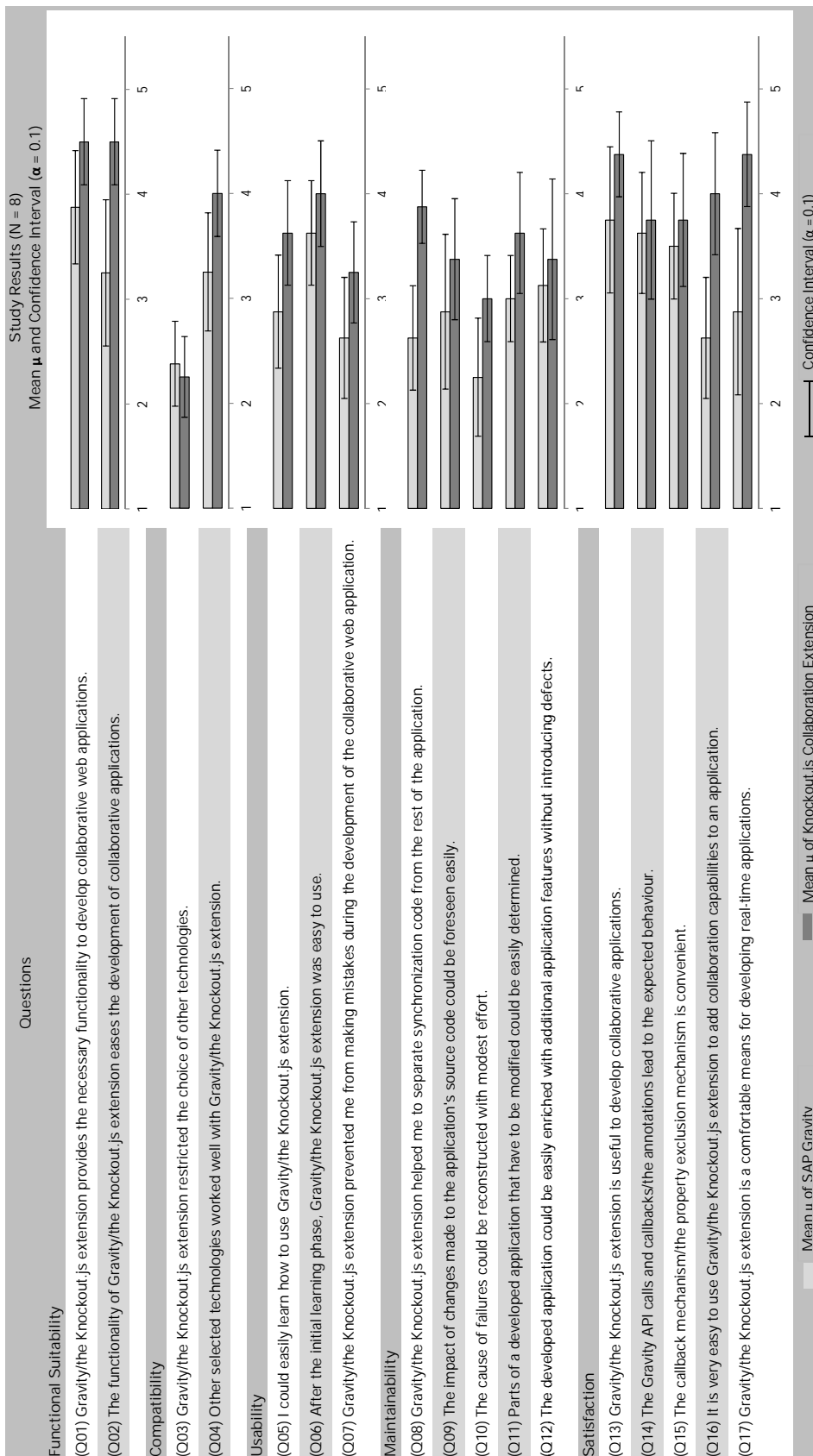


Figure 5.10: Results of the Developer Study Questionnaire

## 5.5 LIMITATIONS

During the developer study several limitations of the Knockout.js CE were carved out by the participants. In the following, the major challenges are explained in detail.

### 5.5.1 Observable Properties

If a view model is marked for synchronization via the `@Sync` annotation (see Section 4.6.3), a traversal algorithm adds the Knockout.js synchronization extension described in Section 4.6.2 to all primitive observables and observable arrays. Therefore it is necessary that all properties of a view model to be reconciled among different view model instances are observables, observable arrays or computed observables. Otherwise the synchronization extension is not added successfully to all parts of the view model resulting in a partial synchronization. Figure 5.11 depicts an invalid view model structure on the left hand side, as well as a revised one compliant with the Knockout.js CE on the right hand side. Note that all properties on the right hand side are observables or observable arrays.

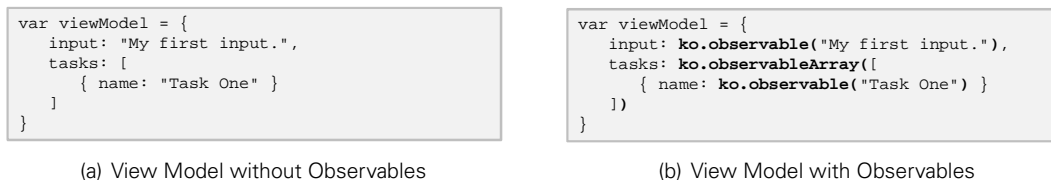


Figure 5.11: Knockout.js View Model without and with Observables

### 5.5.2 Dynamic Property Additions

Once a collaborative Knockout.js application is loaded, the annotations in the view model (see Section 4.6.3) are evaluated. If a view model is marked with the `@Sync` annotation, it is traversed and the Knockout.js synchronization extension is added to all observables. Whenever observables are added to the view model after the traversal, they are not automatically enriched with the synchronization extension and therefore not synchronized. Extending at runtime added observables automatically would require the tracking of additions to the view model at any point in time. Since JavaScript provides no mechanism to observe property additions to objects, a continual task would have to scan the view model for newly added properties. Hence, observing changes comes with a performance penalty and was not implemented. Therefore all properties have to be part of the view model already at design time. Figure 5.12 depicts source code examples for a dynamic property addition at runtime on the left hand side and a fixed version of this source code compliant with the Knockout.js CE at the right hand side.

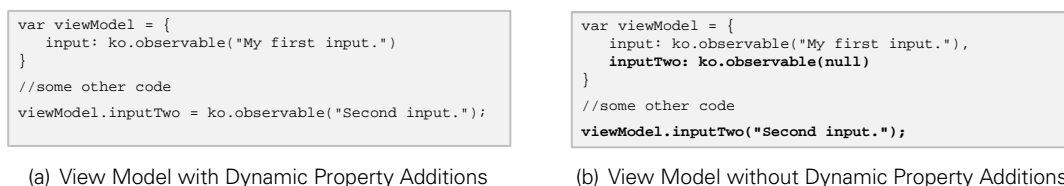


Figure 5.12: Knockout.js View Model with and without Dynamic Property Additions

### 5.5.3 Class Constructor Parameters

Class constructor functions used to create class instances must have only one parameter if they are marked with the `@Class` annotation to enable remote replay. The parameter can be a JavaScript value of simple type (string, number, etc.) or a parameter object encapsulating all parameters. This requirement is raised by the JavaScript object structure and multi-parameter constructor functions. Constructors with multiple parameters have to be called with the parameters in a specific order. Since objects in JavaScript are simple sets of key-value pairs, the order how the parameters were passed to the constructor function formerly cannot be derived from object instances. Therefore parameter objects have to be used that support an arbitrary order of parameters. Figure 5.13 depicts the two possible parameter styles for constructor functions in JavaScript. Figure 5.13(a) shows a constructor for a task item with multiple parameters, whereas Figure 5.13(b) shows the very same constructor function using a parameter object compliant with the Knockout.js collaboration extension.

<pre>var Task = function(name, priority, done) {   this.name = name    "default name";   this.priority = priority    50;   this.done = done    false; } var task = new Task("My Task", 25, false);</pre>	<pre>var Task = function(spec) {   this.name = spec.name    "default name";   this.priority = spec.priority    50;   this.done = spec.done    false; } var task = new Task({   name: "My Task",   priority: 25,   done: false });</pre>
(a) Constructor Function with Separate Parameters	(b) Constructor Function with Parameter Object

Figure 5.13: JavaScript Constructor Function Styles

### 5.5.4 Manual Subscriptions

Advanced users of Knockout.js might subscribe listeners for every observable manually. In case of synchronized class instances, registered listener functions have to be reregistered remotely. If the listeners were added to a class instance after the constructor function was called, those listeners will not be replayed remotely. Therefore it is necessary to subscribe all individual listener functions directly in the class constructor function. Serializing and remotely restoring the listener functions is not an option due to JavaScript's function scope (see Section 4.6.3). Figure 5.14(a) depicts the non-valid subscription of a listener that will not be replayed remotely. The registration within a constructor function, compliant with the Knockout.js collaboration extension, is depicted in Figure 5.14(b).

<pre>var Task = function(name) {   this.name = ko.observable(name    "Default Name"); } var task = new Task("My Task Item"); task.name.subscribe(function(newValue) {   console.log("The new name is: " + newValue); });</pre>	<pre>var Task = function(name) {   this.name = ko.observable(name    "default name");   this.name.subscribe(function(newValue) {     console.log("The new name is: " + newValue);   }); } var task = new Task("My Task Item");</pre>
(a) Listener Registration after Class Instance Creation	(b) Listener Registration during Class Instance Creation

Figure 5.14: Manual Listener Registration in Knockout.js

### 5.5.5 Tree Structure

As already mentioned in Section 4.6.2, the view model of a Knockout.js-based application has to adhere to a tree structure to be compliant with the Knockout.js CE. View models that are graph structures might not be synchronized properly.

Let's consider the definition of a view model structure representing a graph. Figure 5.15(a) shows the source code whereas in Figure 5.15(b) the runtime structure is visualized.



Figure 5.15: View Model with Graph Structure

In the source code snippet first a class constructor function for tasks is defined. Afterwards two task instances `task1` and `task2` are created. The variable `viewModel` contains the actual view model, which is represented by a simple JavaScript object with two properties: an observable array accommodating the tasks and a primitive observable holding a reference to the currently selected task item. If this view model is marked for synchronization using the `@Sync` annotation, the traversal algorithm adds the synchronization extension (see Section 4.6.2) to all observables in the view model. During the traversal the name of `task2` is reached twice at different JSONPath addresses: first at `/viewModel/tasks/1/name` and second at `/viewModel/selectedTask/name`. Therefore the name property is stored twice in the underlying JSON document provided by the Shared JSON implementation (see Section 4.4). At remote sites this would lead to two different copies of the very same object and eventually to inconsistent view models. To avoid this, the view model has to be a tree, so that every observable is addressable by exactly one JSONPath.

This limitation exists only due to the current implementation of the collaboration extension. There are two possible ways to improve the Knockout.js CE:

1. First, the traversal algorithm might be enriched to deal with primary and foreign keys known from relational databases. Once the observable property `name` is reached via `/viewModel/tasks/1/name` the corresponding observable would get an ID that is stored with the observable's value in the shared JSON document. When the traversal algorithm recognizes the name observable again via `/viewModel/selectedTask/name` the ID would be read. The ID indicates that this element was already reached and stored in the shared JSON document. Therefore the ID of the already stored observable would be stored in the JSON document for the current observable rather than the actual value. At remote sites this foreign key might be used to reconstruct the actual value.
2. Second, neglecting reuse the Knockout.js CE might be implemented directly on top of SAP Gravity. If one of the observables would be addressable via different paths in the view model, these paths would be directly mapped to a data model based on SAP Gravity, since it provides a directed, labeled and attributed graph structure.

Rather than re-implementing the Knockout.js CE or to introduce primary and foreign keys in the collaboration extension, keys can be used in the view model by the application developer to ensure that the view model is tree-structured. This workaround is depicted in Figure 5.16 for the view model introduced in Figure 5.15. The left hand side shows the source code of the fixed view model whereas on the right hand side the runtime structure is visualized. Dashed lines indicate indirect references by a foreign key.

The direct reference to `task2` by `selectedTask` is replaced by a foreign key reference. The `selectedId` is an observable that contains just the primary key (ID) of the currently selected task item. Via the computed observable `selectedTask` that looks up the corresponding task according to the ID provided by `selectedId`, the actual task item is returned. Since computed observables have not to be extended with the change tracking extension, this new version of the view model is a tree.

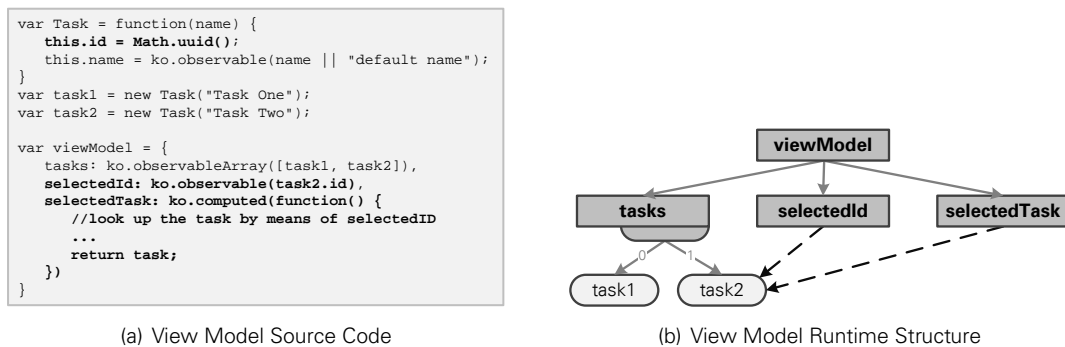


Figure 5.16: View Model with Tree Structure

## 5.6 CONCLUSION

Due to resource constraints only the Knockout.js collaboration extension was evaluated in detail. The evaluation revolved around the ISO/IEC 250xx series of standards, especially the quality models defined in ISO/IEC 25010 [ISO10]. Since the assessment took place from an application developer's point of view, some software quality characteristics defined in the models were neglected. Examples comprise performance efficiency, reliability, etc., and have to be ensured already by the framework developer.

The evaluation was accomplished in two steps. First, a minimal application built by an experienced web developer was evaluated with respect to effectiveness and efficiency. Afterwards, the preliminary results were reconsidered in a developer study.

The minimal task list application that was developed successfully showed a reduction of development time due to the Knockout.js CE of 48 percent compared to SAP Gravity. The development time for code dealing with synchronization and conflict resolution was diminished by 76 percent. Moreover the total code base could be reduced by 52 percent.

During the developer study a business analytic tool supporting different matrix-based analysis was developed within ten weeks. Each of the eight participants had to develop the application alone. The results showed a reduction of the total development time by 22 percent in average when using the Knockout.js CE instead of SAP Gravity. Adding collaboration capabilities by means of the collaboration extension could be achieved in 41 percent less time in average compared to Gravity. Eventually the code base was reduced by 47 percent in average. Besides efficiency further software quality characteristics were assessed via a questionnaire. The results revealed that the collaboration extension significantly eases the development of collaborative applications. Moreover the extension has a flatter learning curve compared to SAP Gravity, and the separation of synchronization and conflict resolution code from the business logic is significantly improved. Since the reconstruction of failures is still a complex endeavor, in further research the debugging facilities have to be improved. Furthermore the Knockout.js CE was treated to be significantly more comfortable for the development of collaborative applications.

Moreover, the developer study exposed some limitations of the Knockout.js collaboration extension. While some of these are inherent to the conceptual architecture of the collaboration extension (observable properties, class constructor parameters, manual subscriptions), others are specific for the current implementation (dynamic property additions, tree structure). The latter can be resolved by a revised implementation.

In essence, it can be concluded that the collaboration extension improves the efficiency when developing collaborative web applications while effectiveness is still ensured. Furthermore developers can easily become familiar with the approach.



## 6 CONCLUSION

The HTML5 movement paved the way for rich interactive web applications. Collaborative applications like Google Docs are nowadays well established and adopted by millions of users. However, a classification of current web applications and assessment in terms of real-time collaboration capabilities revealed that about two thirds of today's web applications do not provide any support. This is the case, because collaborative applications are hard to implement, test and maintain. Even though web application frameworks provide structure to applications by enforcing a structural design pattern like MVC or MVVM, concurrency control and conflict resolution has to be implemented manually using complementary techniques.

To ease the development, the augmentation of existing MV\* frameworks with real-time collaboration capabilities was proposed and a conceptual architecture of collaboration extensions for MV\* frameworks was introduced. A collaboration extension comprises two main parts: A synchronization service (e. g. an operational transformation engine) that is in charge of reconciling different application states, and a collaboration adapter coupling the synchronization service with a framework-based application. Owing to the fact that the collaboration extension is only dependent on framework- instead of application-specific concepts, reuse among several applications built on top of the very same framework is possible. Since the proposed collaboration extensions complement frameworks in an orthogonal way rather than interfering with existing framework features, application developers can profit from already gained experiences with the development of framework-based applications. Moreover, collaboration features can be integrated into an application in a lightweight fashion either by a collaboration proxy or annotation-based code injection.

To prove the feasibility of the proposed collaboration extensions, two frameworks (SAPUI5 and Knockout.js) were selected. For both SAPUI5 and Knockout.js the conceptual architecture for collaboration extensions was refined towards specific implementation architectures. The operational transformation engine SAP Gravity was used as synchronization service. The collaboration extension implementation comprised two layers on top of SAP Gravity: An abstraction layer enabling the concurrent modification of shared JSON documents, and the collaboration adapter bridging the gap between a shared JSON document and the framework's data encapsulation concepts. Although only SAPUI5 and Knockout.js were augmented, the conceptual architecture might be implemented also for other frameworks.

Due to resource constraints only the Knockout.js collaboration extension was evaluated in detail. A developer study was conducted to compare the collaboration-enabled Knockout.js framework with Knockout.js in conjunction with SAP Gravity when developing a collaborative web application. Several software quality characteristics were captured and compared. Examples include effectiveness, efficiency, functional suitability, usability, etc. Almost all evaluated characteristics

were judged positively by the study's participants. Room for improvements could be seen only regarding error protection and error tracking. Furthermore several limitations of the Knockout.js collaboration extensions were carved out during the study. Even though the approach currently suffers from several limitations, developers are empowered to efficiently program new collaborative applications and to migrate existing single-user applications.

Despite the gained results, there are several open problems that might be subject to research in the future:

- A developer study might be conducted for the SAPUI5 collaboration extension that was implemented. This study could expose insights to the gains with respect to several software quality criteria compared to the development of collaborative applications using SAPUI5 in conjunction with SAP Gravity.
- Because the developer study carried out for the Knockout.js collaboration extension revealed several limitations, future versions of the extension might address these limitations. Especially the limitations requiring a tree-structured data model and prohibiting dynamic property additions should be considered.
- Besides the developer study, a user study could be conducted for a real-size application. Therefore the very same application would have to be implemented by means of a traditional development approach (e. g. an operational transformation library) and a collaboration-enabled web application framework. Assuming that application-specific implementations provide the best possible collaboration experience, this evaluation would show to which extent the collaboration extension reaches the perfect result in terms of usability of the final application for end-users.
- Aside from concurrent work, group awareness is another important principle for collaborative applications. Awareness is defined as the "up-to-the-moment understanding of another person's interaction with the shared space." [GG02] In essence, it enables effective collaborative work by answering the "who, what, and where" questions [GSG95] (e. g. who is in the workspace, what are the other participants doing, where are they working). The integration of awareness services in an application currently requires techniques apart from the used web application framework. How awareness can be incorporated as integral part of web application frameworks that enable the development of collaborative applications is an open research question.
- Even though no performance measurements are available yet, the collaboration extensions might be improved by using a synchronization service that provides smaller latencies compared to SAP Gravity. Another option would be the use of new communication technologies (e. g. WebSockets) instead of the currently used HTTP long polling in SAP Gravity.

# BIBLIOGRAPHY

- [ALX<sup>+</sup>08] Agustina, Fei Liu, Steven Xia, Haifeng Shen, and Chengzheng Sun. CoMaya: Incorporating Advanced Collaboration Capabilities into 3D Digital Media Design Tools. In *CSCW*, pages 5–8, 2008.
- [BL12] Tim Berners-Lee. WorldWideWeb, the first Web Client. <http://www.w3.org/People/Berners-Lee/WorldWideWeb.html>, 2012.
- [BPSM<sup>+</sup>06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). <http://www.w3.org/TR/2006/REC-xml11-20060816/>, 2006.
- [BRS99] James Begole, Mary Beth Rosson, and Clifford A. Shaffer. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems. *ACM Trans. Comput.-Hum. Interact.*, 6(2):95–132, 1999.
- [BSS01] James Begole, Randall B. Smith, Craig A. Struble, and Clifford A. Shaffer. Resource Sharing for Replicated Synchronous Groupware. *IEEE/ACM Trans. Netw.*, 9(6):833–843, 2001.
- [CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath/>, 1999.
- [Cor11] Microsoft Corporation. Open Data Protocol (OData) Specification. [http://www.odata.org/media/16352/\[ms-odata\].pdf](http://www.odata.org/media/16352/[ms-odata].pdf), 2011.
- [Cro06] Douglas Crockford. RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON). <http://www.ietf.org/rfc/rfc4627.txt>, 2006.
- [Cro08] Douglas Crockford. *JavaScript: The Good Parts*. O’Reilly Media, Inc., 2008.
- [Dij82] Edsger W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.
- [DSL02] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. Generalizing Operational Transformation to the Standard General Markup Language. In *CSCW*, pages 58–67, 2002.
- [EBN96] ISO/IEC 14977 : 1996 (E) - Information technology – Syntactic Metalanguage – Extended BNF. <http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>, 1996.
- [EG89] Clarence A. Ellis and Simon J. Gibbs. Concurrency Control in Groupware Systems. In *SIGMOD Conference*, pages 399–407, 1989.

- [Fla11] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., 2011.
- [Fow04] Martin Fowler. Presentation Model. <http://www.martinfowler.com/eaDev/PresentationModel.html>, 2004.
- [Fra09] Neil Fraser. Differential Synchronization. In *ACM Symposium on Document Engineering*, pages 13–20, 2009.
- [FS12] Hongfei Fan and Chengzheng Sun. Achieving Integrated Consistency Maintenance and Awareness in Real-Time Collaborative Programming Environments: The Co-Eclipse Approach. In *CSCWD*, pages 94–101, 2012.
- [Gar05] Jesse James Garrett. AJAX: A New Approach to Web Applications. <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications>, 2005.
- [GG98] Carl Gutwin and Saul Greenberg. Effects of Awareness Support on Groupware Usability. In *CHI*, pages 511–518, 1998.
- [GG02] Carl Gutwin and Saul Greenberg. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *Computer Supported Cooperative Work*, 11(3-4):411–446, 2002.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
- [GLG11] Carl Gutwin, Michael Lippold, and T. C. Nicholas Graham. Real-time Groupware in the Browser: Testing the Performance of Web-based Networking. In *CSCW*, pages 167–176, 2011.
- [Goo07] Danny Goodman. *Dynamic HTML: The Definitive Reference*. O'Reilly Media, Inc., 2007.
- [Gos05] John Gossman. Introduction to Model/View/ViewModel Pattern for Building WPF Apps. <http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>, 2005.
- [GSG95] Carl Gutwin, Gwen Stark, and Saul Greenberg. Support for Workspace Awareness in Educational Groupware. In *CSCW*, pages 147–156, 1995.
- [HC99] Antony L. Hosking and Jiawan Chen. Mostly-Copying Reachability-based Orthogonal Persistence. In *OOPSLA*, pages 382–398, 1999.
- [HG04] Jason Hill and Carl Gutwin. The MAUI Toolkit: Groupware Widgets for Group Awareness. *Computer Supported Cooperative Work*, 13(5-6):539–571, 2004.
- [Hic12a] Ian Hickson. HTML5 - A Vocabulary and Associated APIs for HTML and XHTML. <http://www.w3.org/TR/html5/>, 2012.
- [Hic12b] Ian Hickson. HTML5 Web Messaging. <http://dev.w3.org/html5/postmsg/>, 2012.
- [Hic12c] Ian Hickson. Web Storage. <http://dev.w3.org/html5/webstorage/>, 2012.
- [HLSG12] Matthias Heinrich, Franz Lehmann, Thomas Springer, and Martin Gaedke. Exploiting Single-User Web Applications for Shared Editing: A Generic Transformation Approach. In *WWW*, pages 1057–1066, 2012.
- [Int97] ECMA International. ECMAScript: A General Purpose, Cross-Platform Programming Language; ECMA-262; First Edition. <http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf>, 1997.

- [Int11] ECMA International. ECMAScript: A General Purpose, Cross-Platform Programming Language; ECMA-262; 5.1 Edition. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>, 2011.
- [ISO05] ISO/IEC FDIS 25000 : 2005 (E) - Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE, 2005.
- [ISO10] ISO/IEC FDIS 25010 : 2010 (E) - Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models, 2010.
- [Joh88] Robert Johansen. *GroupWare: Computer Support for Business Teams*. The Free Press, New York, NY, USA, 1988.
- [KLM<sup>+</sup>97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.
- [KP88] Glenn E. Krasner and Stephen T. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. [http://www.itu.dk/courses/VOP/E2005/VOP2005E/8\\_mvc\\_krasner\\_and\\_pope.pdf](http://www.itu.dk/courses/VOP/E2005/VOP2005E/8_mvc_krasner_and_pope.pdf), 1988.
- [LCSD07] Kai Lin, David Chen, Chengzheng Sun, and R. Geoff Dromey. Leveraging Single-User Microsoft Visio for Multi-user Real-Time Collaboration. In *CDVE*, pages 353–360, 2007.
- [Lik32] Rensis Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 22(140):1–55, 1932.
- [Mar11] Chris Marrin. WebGL Specification, Version 1.0. <https://www.khronos.org/registry/webgl/specs/1.0/>, 2011.
- [McI68] Doug McIlroy. Mass-Produced Software Components. In P. Naur and B. Randell, editors, *Proceedings of NATO Software Engineering Conference*, pages 138–155, 1968.
- [Mic02] Sun Microsystems. Core J2EE Patterns – Data Access Object. <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>, 2002.
- [MT08] Tommi Mikkonen and Antero Taivalsaari. Web Applications Spaghetti Code for the 21st Century. In *SERA*, pages 319–328, 2008.
- [Pat95] John F. Patterson. A Taxonomy of Architectures for Synchronous Groupware Applications. *SIGOIS Bull.*, 15(3):27–29, 1995.
- [Ree79a] Trygve M. H. Reenskaug. Models - Views - Controllers. <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>, December 1979.
- [Ree79b] Trygve M. H. Reenskaug. Thing-Model-View-Editor - an Example from a Planning-system. <http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf>, 1979.
- [RP05] Steven L. Rohall and John F. Patterson. The Zipper System for Flexible, Replicated Application Sharing. [http://domino.watson.ibm.com/cambridge/research.nsf/0/8df3bbbd96965284852570a500603bab/\\$FILE/TR\\_2005-08.pdf](http://domino.watson.ibm.com/cambridge/research.nsf/0/8df3bbbd96965284852570a500603bab/$FILE/TR_2005-08.pdf), 2005.
- [RZ95] Dirk Riehle and Heinz Züllighoven. A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. In *Pattern Languages of Program Design*, pages 9–42. Addison-Wesley, 1995.
- [San12] Steven Sanderson. Knockout : Home. <http://knockoutjs.com/>, 2012.

- [SAP12] UI Development Toolkit for HTML5 Developer Center. <http://scn.sap.com/community/developer-center/front-end>, 2012.
- [SBF+87] Mark Stefik, Daniel G. Bobrow, Gregg Foster, Stan Lanning, and Deborah G. Tatar. WYSIWIS Revised: Early Experiences with Multiuser Interfaces. *ACM Trans. Inf. Syst.*, 5(2):147–167, 1987.
- [Sch95] Ken Schwaber. SCRUM Development Process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 117–134, 1995.
- [SRL03] G.G. Simpson, A. Roe, and R.C. Lewontin. *Quantitative Zoology: Revised Edition*. Dover Books on Biology, Psychology and Medicine. Dover Publications, 2003.
- [SS06] David Sun and Chengzheng Sun. Operation Context and Context-based Operational Transformation. In *CSCW*, pages 279–288, 2006.
- [SSZ07] Haifeng Shen, Chengzheng Sun, and Suiping Zhou. Leveraging Single-User OpenOffice Writer for Collaboration by Transparent Adaptation. In *SNPD (1)*, pages 15–20, 2007.
- [SXS+06] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, 2006.
- [TM11] Antero Taivalsaari and Tommi Mikkonen. The Web as an Application Platform: The Saga Continues. In *EUROMICRO-SEAA*, pages 170–174, 2011.
- [XSS+04] Steven Xia, David Sun, Chengzheng Sun, David Chen, and Haifeng Shen. Leveraging Single-User Applications for Multi-User Collaboration: The CoWord Approach. In *CSCW*, pages 162–171, 2004.
- [ZSS09] Yang Zheng, Haifeng Shen, and Chengzheng Sun. Leveraging Single-User AutoCAD for Collaboration by Transparent Adaptation. In *CSCWD*, pages 78–83, 2009.

# LIST OF FIGURES

1.1	Time-Space Matrix of Collaboration (cf. [Joh88]) . . . . .	2
1.2	Simple Task List Application . . . . .	3
2.1	Major Evolution Steps of the Web (cf. [TM11]) . . . . .	5
2.2	Web Application Classification Based on Application Domain . . . . .	8
2.3	Web Application Classification Based on Implementation Technology . . . . .	9
3.1	Application Running on Multiple Sites with a Collaboration Extension . . . . .	17
3.2	Model-View-Controller (MVC) Pattern . . . . .	20
3.3	Presentation Model (PM) Pattern . . . . .	21
3.4	MV* Framework Classification Dimensions . . . . .	22
3.5	Application Structure of an Access Object-based Data Model . . . . .	26
3.6	Application Structure of a Remote Proxy-based Data Model . . . . .	26
3.7	Application Structure of a Subgraph-based Data Model . . . . .	27
3.8	Application Structure of a Scattered Data Model . . . . .	27
3.9	Conceptual Architecture of Collaborative MV* Applications . . . . .	30
3.10	Synchronization Example Based on Operational Transformation for Two Participants Editing a Simple Text Document . . . . .	30
3.11	Class Structure of a Collaboration Proxy Implementation . . . . .	32
3.12	Subgraph-based Data Model with Injected Synchronization Code . . . . .	33

4.1	Implementation Architecture of SAPUI5 and Knockout.js Collaboration Extension . . . . .	39
4.2	SAP Gravity Architecture . . . . .	40
4.3	Example of a Data Model Instance in SAP Gravity . . . . .	41
4.4	SAP Gravity Mapping from Application-Specific to Application-Agnostic Operations . . . . .	41
4.5	Model Change Event Handling in SAP Gravity . . . . .	44
4.6	SAP Gravity Synchronization and Conflict Resolution Workflow . . . . .	45
4.7	Gravity Mapping of Primitive JSON Data Types . . . . .	47
4.8	Gravity Mapping of JSON Objects . . . . .	48
4.9	Gravity Mapping of JSON Arrays . . . . .	48
4.10	JSON to Gravity Mapping Example . . . . .	49
4.11	Shared JSON Synchronization and Conflict Resolution Workflow . . . . .	54
4.12	Class Structure of Collaborative SAPUI5 JSONModel with Associated Data Binding Classes . . . . .	56
4.13	SAPUI5 Synchronization and Conflict Resolution Workflow . . . . .	58
4.14	Knockout.js Synchronization and Conflict Resolution Workflow . . . . .	67
5.1	Product Quality Model According to ISO/IEC 25010 [ISO10] . . . . .	70
5.2	Quality in Use Model According to ISO/IEC 25010 [ISO10] . . . . .	72
5.3	Overall Evaluation Process . . . . .	73
5.4	Screenshot of Simple Task List Application . . . . .	74
5.5	Evaluation Results of Minimal Task List Application . . . . .	75
5.6	Mockup of Collaborative Business Analytic Tool . . . . .	76
5.7	Template for Sprint Documentation . . . . .	78
5.8	Questions to Participants of Developer Study . . . . .	79
5.9	Source Code Results of the Developer Study . . . . .	80
5.10	Results of the Developer Study Questionnaire . . . . .	82
5.11	Knockout.js View Model without and with Observables . . . . .	83
5.12	Knockout.js View Model with and without Dynamic Property Additions . . . . .	83
5.13	JavaScript Constructor Function Styles . . . . .	84
5.14	Manual Listener Registration in Knockout.js . . . . .	84



5.15 View Model with Graph Structure . . . . .	85
5.16 View Model with Tree Structure . . . . .	86
A.1 JSON Value Syntax Definition . . . . .	101
A.2 JSON Object Syntax Definition . . . . .	101
A.3 JSON Array Syntax Definition . . . . .	102
A.4 JSON String Syntax Definition . . . . .	102
A.5 JSON Number Syntax Definition . . . . .	102



# LIST OF TABLES

2.1	Classification of Available Web Applications . . . . .	10
4.1	Survey of Existing Web MV* Frameworks . . . . .	37
4.2	Primitive Operations in SAP Gravity . . . . .	41
4.3	Mapping of API Method Calls to Primitive Operations and Model Change Events in SAP Gravity . . . . .	43
4.4	JSONPath Expressions and Corresponding Addressed Document Parts . . . . .	47
4.5	Shared JSON Event Types with Corresponding Properties . . . . .	53
4.6	Examples for Methods of Observable Arrays . . . . .	60
4.7	Knockout.js Annotations with Regular Expressions and Source Code Replacements	65
4.8	Examples for Property Exclusion Expressions . . . . .	65
5.1	Subcharacteristics of Functional Suitability According to ISO/IEC 25010 [ISO10] . . . . .	71
5.2	Subcharacteristics of Compatibility According to ISO/IEC 25010 [ISO10] . . . . .	71
5.3	Subcharacteristics of Usability According to ISO/IEC 25010 [ISO10] . . . . .	71
5.4	Subcharacteristics of Maintainability According to ISO/IEC 25010 [ISO10] . . . . .	72
5.5	Subcharacteristics of Satisfaction According to ISO/IEC 25010 [ISO10] . . . . .	73
5.6	Schedule of Practical Course for Developer Study . . . . .	77
B.1	Questionnaire Experiences of Participants . . . . .	103
B.2	Questionnaire Knockout.js Collaboration Extension . . . . .	104

B.3	Questionnaire SAP Gravity . . . . .	105
C.1	Questionnaire Result Distribution Experiences of Participants . . . . .	107
C.2	Questionnaire Result Distribution Functional Suitability . . . . .	107
C.3	Questionnaire Result Distribution Compatibility . . . . .	108
C.4	Questionnaire Result Distribution Usability . . . . .	108
C.5	Questionnaire Result Distribution Maintainability . . . . .	108
C.6	Questionnaire Result Distribution Satisfaction . . . . .	109

# LISTINGS

3.1	Single JavaScript Include of Knockout.js framework . . . . .	23
3.2	Template-Based UI Specification by means of Handlebars . . . . .	24
3.3	Widget-Based UI Specification by means of SproutCore . . . . .	24
3.4	Data Binding by means of Knockout.js . . . . .	25
3.5	Procedural View Update by means of Backbone.js . . . . .	25
3.6	Model-based Listener Registration using SAPUI5 . . . . .	28
3.7	Class Instance-based Listener Registration using Backbone.js . . . . .	28
3.8	Property-based Listener Registration using Knockout.js . . . . .	28
4.1	Gravity Client API Calls to Create a Task List Item . . . . .	43
4.2	JSON Representation of a Task List . . . . .	46
4.3	JSONPath Syntax Definition Using the EBNF [EBN96] . . . . .	47
4.4	JSON representation of a New Task Item . . . . .	50
4.5	Gravity Client API Calls to Add a New Task Item . . . . .	50
4.6	Example of a Shared JSON Event Object . . . . .	53
4.7	Utilization of Single-User as well as Multi-User JSON Data Model in SAPUI5 . . . . .	57
4.8	Knockout.js View Model for a Task List Application . . . . .	59
4.9	Knockout.js Synchronization Extension for Observables . . . . .	61
4.10	Knockout.js View Model for a Task List Application with Annotations . . . . .	62
4.11	JavaScript Object Recreation Using Serialization . . . . .	63

4.12 Property Exclusion Language Definition . . . . .	65
4.13 Knockout.js View Definition . . . . .	66

# A JAVASCRIPT OBJECT NOTATION (JSON)

The JavaScript Object Notation (JSON) is a lightweight and language-independent data interchange format. Since it is a text format, it is human- as well as machine-readable. In the following, the JSON syntax is described using syntax diagrams according to [Cro08]. Dark gray rectangles represent non-terminal symbols, whereas rounded rectangles as well as circles stand for terminal symbols.

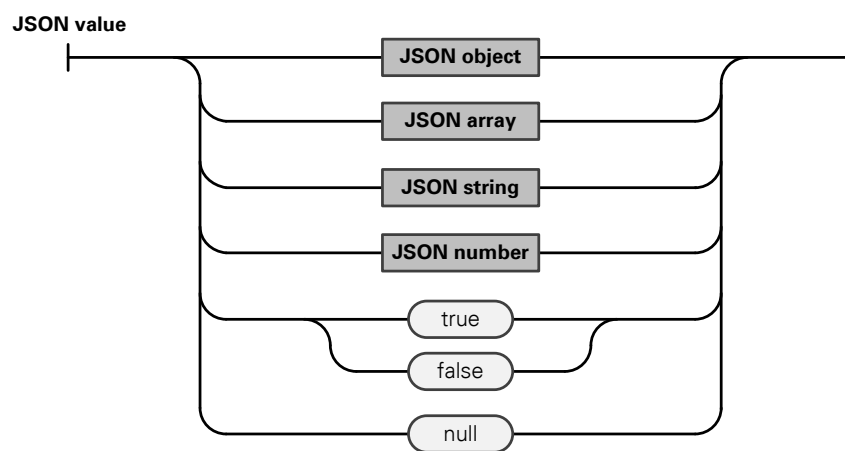


Figure A.1: JSON Value Syntax Definition

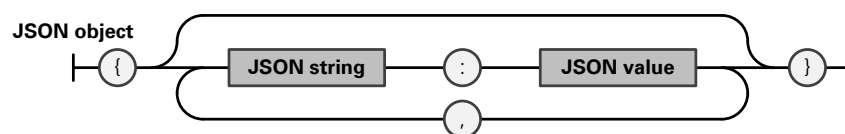


Figure A.2: JSON Object Syntax Definition

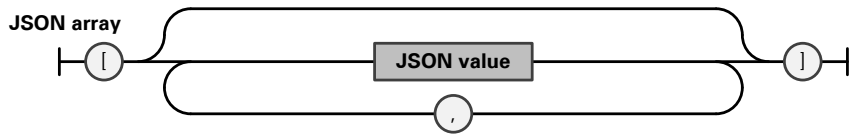


Figure A.3: JSON Array Syntax Definition

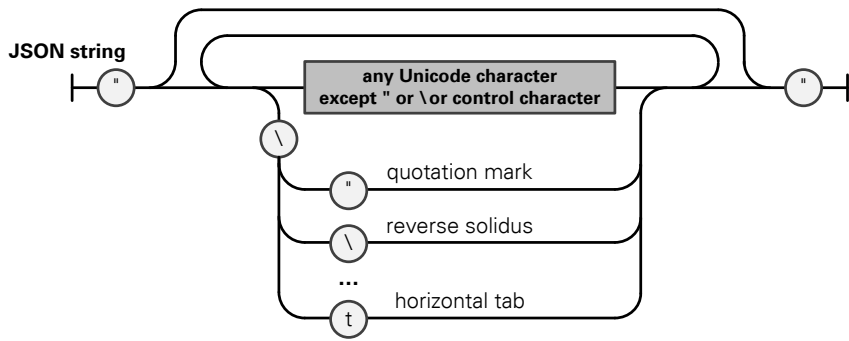


Figure A.4: JSON String Syntax Definition

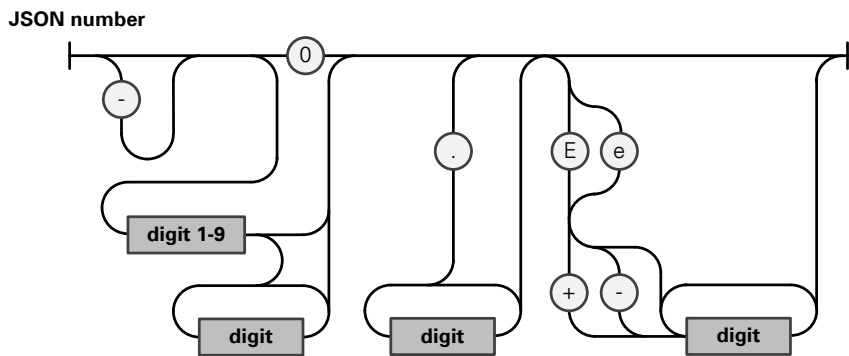


Figure A.5: JSON Number Syntax Definition



## B DEVELOPER EVALUATION: QUESTIONNAIRE

Question	strongly disagree			strongly agree	
	1	2	3	4	5
I had strong programming skills before the internship.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Which programming languages did you use before the internship? _____					
I was familiar with HTML and CSS before the internship.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was familiar with JavaScript before the internship.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I was familiar with the development of collaborative web applications before the internship.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Table B.1: Questionnaire Experiences of Participants

Question	strongly disagree			strongly agree	
	1	2	3	4	5
<b>Functional Suitability</b>					
The Knockout.js extension provides the necessary functionality to develop collaborative web applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The functionality of the Knockout.js extension eases the development of collaborative applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
What functionality did you miss? What hindered your development? _____					
<b>Compatibility</b>					
The Knockout.js extension restricted the choice of other technologies.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Other selected technologies worked well with the Knockout.js extension.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
What problems did you encounter using other technologies? _____					
<b>Usability</b>					
I could easily learn how to use the Knockout.js extension.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
After the initial learning phase, the Knockout.js extension was easy to use.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
What was hard for you to learn? _____					
The Knockout.js extension prevented me from making mistakes during the development of the collaborative web application.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
What kind of errors did you encounter? _____					
<b>Maintainability</b>					
The Knockout.js extension helped me to separate synchronization code from the rest of the application.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The impact of changes made to the application's source code could be foreseen easily.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The cause of failures could be reconstructed with modest effort.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Parts of a developed application that have to be modified could be easily determined.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The developed application could be easily enriched with additional application features without introducing defects.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<b>Satisfaction</b>					
The Knockout.js extension is useful to develop collaborative applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The annotations lead to the expected behaviour.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The property exclusion mechanism is convenient.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
It is very easy to use the Knockout.js extension to add collaboration capabilities to an application.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The Knockout.js extension is a comfortable means for developing real-time applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Table B.2: Questionnaire Knockout.js Collaboration Extension

Question	strongly disagree			strongly agree	
	1	2	3	4	5
<b>Functional Suitability</b>					
Gravity provides the necessary functionality to develop collaborative web applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The functionality of Gravity eases the development of collaborative applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
What functionality did you miss? What hindered your development? _____					
<b>Compatibility</b>					
Gravity restricted the choice of other technologies.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Other selected technologies worked well with Gravity.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
What problems did you encounter using other technologies? _____					
<b>Usability</b>					
I could easily learn how to use Gravity.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
After the initial learning phase, Gravity was easy to use.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
What was hard for you to learn? _____					
Gravity prevented me from making mistakes during the development of the collaborative web application.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
What kind of errors did you encounter? _____					
<b>Maintainability</b>					
Gravity helped me to separate synchronization code from the rest of the application.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The impact of changes made to the application's source code could be foreseen easily.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The cause of failures could be reconstructed with modest effort.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Parts of a developed application that have to be modified could be easily determined.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The developed application could be easily enriched with additional application features without introducing defects.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<b>Satisfaction</b>					
Gravity is useful to develop collaborative applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The Gravity API calls and callbacks lead to the expected behaviour.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The callback mechanism is convenient.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
It is very easy to use Gravity to add collaboration capabilities to an application.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Gravity is a comfortable means for developing real-time applications.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Table B.3: Questionnaire SAP Gravity



# C DEVELOPER EVALUATION: RESULTS

Question	Answer												
I had strong programming skills before the internship.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>0</td><td>3</td><td>2</td><td>1</td><td>2</td></tr> </table>	Rating	1	2	3	4	5	Count	0	3	2	1	2
Rating	1	2	3	4	5								
Count	0	3	2	1	2								
I was familiar with HTML and CSS before the internship.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>0</td><td>3</td><td>1</td><td>3</td></tr> </table>	Rating	1	2	3	4	5	Count	1	0	3	1	3
Rating	1	2	3	4	5								
Count	1	0	3	1	3								
I was familiar with JavaScript before the internship.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>4</td><td>1</td><td>0</td><td>1</td><td>2</td></tr> </table>	Rating	1	2	3	4	5	Count	4	1	0	1	2
Rating	1	2	3	4	5								
Count	4	1	0	1	2								
I was familiar with the development of collaborative web applications before the internship.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>6</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	Rating	1	2	3	4	5	Count	6	1	0	0	0
Rating	1	2	3	4	5								
Count	6	1	0	0	0								

Table C.1: Questionnaire Result Distribution Experiences of Participants

Question	SAP Gravity	Knockout.js																								
(Q01) Gravity/the Knockout.js extension provides the necessary functionality to develop collaborative web applications.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>0</td><td>1</td><td>1</td><td>3</td><td>2</td></tr> </table>	Rating	1	2	3	4	5	Count	0	1	1	3	2	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>0</td><td>0</td><td>1</td><td>2</td><td>4</td></tr> </table>	Rating	1	2	3	4	5	Count	0	0	1	2	4
Rating	1	2	3	4	5																					
Count	0	1	1	3	2																					
Rating	1	2	3	4	5																					
Count	0	0	1	2	4																					
(Q02) The functionality of Gravity/the Knockout.js extension eases the development of collaborative applications.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>0</td><td>3</td><td>2</td><td>1</td><td>2</td></tr> </table>	Rating	1	2	3	4	5	Count	0	3	2	1	2	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>0</td><td>1</td><td>1</td><td>2</td><td>4</td></tr> </table>	Rating	1	2	3	4	5	Count	0	1	1	2	4
Rating	1	2	3	4	5																					
Count	0	3	2	1	2																					
Rating	1	2	3	4	5																					
Count	0	1	1	2	4																					

Table C.2: Questionnaire Result Distribution Functional Suitability

Question	SAP Gravity	Knockout.js
(Q03) Gravity/the Knockout.js extension restricted the choice of other technologies.		
(Q04) Other selected technologies worked well with Gravity/the Knockout.js extension.		

Table C.3: Questionnaire Result Distribution Compatibility

Question	SAP Gravity	Knockout.js
(Q05) I could easily learn how to use Gravity/the Knockout.js extension.		
(Q06) After the initial learning phase, Gravity/the Knockout.js extension was easy to use.		

Table C.4: Questionnaire Result Distribution Usability

Question	SAP Gravity	Knockout.js
(Q07) Gravity/the Knockout.js extension prevented me from making mistakes during the development of the collaborative web application.		
(Q08) Gravity/the Knockout.js extension helped me to separate synchronization code from the rest of the application.		
(Q09) The impact of changes made to the application's source code could be foreseen easily.		
(Q10) The cause of failures could be reconstructed with modest effort.		
(Q11) Parts of a developed application that have to be modified could be easily determined.		
(Q12) The developed application could be easily enriched with additional application features without introducing defects.		

Table C.5: Questionnaire Result Distribution Maintainability

Question	SAP Gravity	Knockout.js																								
(Q13) Gravity/the Knockout.js extension is useful to develop collaborative applications.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>2</td><td>1</td><td>1</td><td>2</td><td>3</td></tr> </table>	Rating	1	2	3	4	5	Count	2	1	1	2	3	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	Rating	1	2	3	4	5	Count	1	1	2	3	4
Rating	1	2	3	4	5																					
Count	2	1	1	2	3																					
Rating	1	2	3	4	5																					
Count	1	1	2	3	4																					
(Q14) The Gravity API calls and callbacks/the annotations lead to the expected behaviour.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>2</td><td>1</td><td>5</td><td>1</td><td>0</td></tr> </table>	Rating	1	2	3	4	5	Count	2	1	5	1	0	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>2</td><td>2</td><td>2</td><td>3</td></tr> </table>	Rating	1	2	3	4	5	Count	1	2	2	2	3
Rating	1	2	3	4	5																					
Count	2	1	5	1	0																					
Rating	1	2	3	4	5																					
Count	1	2	2	2	3																					
(Q15) The callback mechanism/the property exclusion mechanism is convenient.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>2</td><td>0</td><td>6</td><td>0</td></tr> </table>	Rating	1	2	3	4	5	Count	1	2	0	6	0	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>1</td><td>3</td><td>1</td><td>3</td></tr> </table>	Rating	1	2	3	4	5	Count	1	1	3	1	3
Rating	1	2	3	4	5																					
Count	1	2	0	6	0																					
Rating	1	2	3	4	5																					
Count	1	1	3	1	3																					
(Q16) It is very easy to use Gravity/the Knockout.js extension to add collaboration capabilities to an application.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>3</td><td>2</td><td>2</td><td>0</td></tr> </table>	Rating	1	2	3	4	5	Count	1	3	2	2	0	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>1</td><td>1</td><td>3</td><td>3</td></tr> </table>	Rating	1	2	3	4	5	Count	1	1	1	3	3
Rating	1	2	3	4	5																					
Count	1	3	2	2	0																					
Rating	1	2	3	4	5																					
Count	1	1	1	3	3																					
(Q17) Gravity/the Knockout.js extension is a comfortable means for developing real-time applications.	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>3</td><td>2</td><td>0</td><td>2</td></tr> </table>	Rating	1	2	3	4	5	Count	1	3	2	0	2	<table border="1"> <tr><th>Rating</th><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><th>Count</th><td>1</td><td>2</td><td>1</td><td>0</td><td>5</td></tr> </table>	Rating	1	2	3	4	5	Count	1	2	1	0	5
Rating	1	2	3	4	5																					
Count	1	3	2	0	2																					
Rating	1	2	3	4	5																					
Count	1	2	1	0	5																					

Table C.6: Questionnaire Result Distribution Satisfaction