Reusable Awareness Widgets for Collaborative Web Applications – A Non-invasive Approach

Matthias Heinrich[1], Franz Josef Grüneberger[1],
Thomas Springer[2], and Martin Gaedke[3]

[1] SAP Research, Germany
{matthias.heinrich,franz.josef.grueneberger}@sap.com
[2] Department of Computer Science,
Dresden University of Technology, Germany
thomas.springer@tu-dresden.de
[3] Department of Computer Science,
Chemnitz University of Technology, Germany
martin.gaedke@cs.tu-chemnitz.de

Abstract. Creating awareness about other users' activities in a shared workspace is crucial to support efficient collaborative work. Even though the development of awareness widgets such as participant lists, telepointers or radar views is a costly and complex endeavor, awareness widget reuse is largely neglected. Collaborative applications either integrate specific awareness widgets or leverage existing awareness toolkits which require major source code adaptations and thus, are not suited to rapidly enrich existing web applications.

Therefore, we propose a generic awareness infrastructure promoting an accelerated, cost-efficient development of awareness widgets as well as a non-invasive integration of awareness support into existing web applications. To validate our approach, we demonstrate the integration of three developed awareness widgets in four collaborative web editors. Furthermore, we expose insights about the development of reusable awareness widgets and discuss the limitations of the devised awareness infrastructure.

1 Introduction

Collaborative web applications such as Google Docs have become pervasive in our daily lives since they expose a rich feature set, provide broad device support and offer instant accessibility without inducing time-consuming installation procedures. Commonly, those collaborative real-time applications allow multiple users to edit the same document concurrently which requires workspace awareness support. Workspace awareness is defined as the "up-to-the-moment understanding of another person's interaction with the shared space" [1] and in essence, it enables effective collaborative work [2] by answering the "who, what, and where" questions (e.g. who is in the workspace, what are the other participants doing, where are they working). Examples of widely adopted awareness

M. Brambilla, T. Tokuda, and R. Tolksdorf (Eds.): ICWE 2012, LNCS 7387, pp. 1–15, 2012.

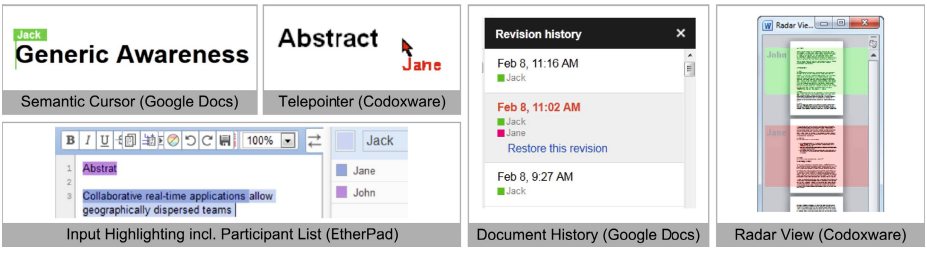

Fig. 1. Screenshots of application-specific awareness widgets extracted from Google Docs, Codoxware and EtherPad

widgets supporting collaborative work are participant lists, telepointers, radar views, etc. (cf. Figure 1).

Even though multi-user applications largely offer the same set of awareness widgets (e.g. most collaborative applications provide some kind of participant list) and software reuse has been advocated for decades [3], there are no web frameworks accommodating out-of-the-box awareness support and promoting a non-invasive integration approach. Thus, developers re-implement awareness widgets for each collaborative web application or face massive source code changes adopting awareness toolkits, in particular, if existing applications are enriched. Both approaches entail major development efforts and costs.

Therefore, we devised a generic awareness infrastructure (GAI) that on the one hand side simplifies the development of awareness widgets by providing basic awareness services. On the other hand side, the GAI promotes the reuse of awareness widgets facilitating a non-invasive integration approach. The widget reuse is achieved by anchoring the GAI in various W3C specifications (e.g. CSS Object Model [4], DOM Core [5] or DOM Events specification [6]). Consequently, standards-based web applications are able to leverage the GAI including the library of reusable awareness widgets.

The main contributions of this paper are three-fold:

- We propose a generic awareness infrastructure facilitating non-invasive awareness support for standards-based web applications.
- We expose a development blueprint supporting developers to devise novel reusable awareness widgets for web applications.
- We evaluate the generic awareness infrastructure by incorporating three implemented awareness widgets into four collaborative web applications and discuss the limitations of the proposed approach.

The rest of this paper is organized as follows: Section 2 elaborates on the challenges devising reusable awareness widgets. Section 3 illustrates the GAI architecture and introduces the development blueprint for novel reusable awareness widgets. While Section 4 presents the validation of the GAI approach, Section 5 carves out strengths and limitations. Section 6 compares our work to the state-of-the-art and Section 7 exhibits conclusions.

2 Challenges

Devising a generic solution instead of an application-specific one imposes additional challenges since universal solutions have to abstract from certain specifics to aspects that hold for an entire class of applications. Characteristics that are especially challenging while developing generic awareness support for web-applications are

1. the diversity of collaborative web applications,
2. the multitude of available browsers, and
3. the proliferation of web-enabled devices.

The diversity of collaborative web applications embraces aspects like the targeted runtime engine (e.g. standards-based browser runtime or plug-in technology based one) and the addressed application domain (e.g. text editing, graphics editing, etc.). Considering the variety of plug-in technologies (e.g. Adobe Flash, Java Applets or Microsoft Silverlight) and their slipping importance with respect to web engineering, we will primarily try to tackle this challenge for W3C standards-based applications.

Another aspect that a generic awareness solution should take into account are the various browser implementations. Since the set of available browsers in its entirety also encompasses peculiar implementations such as text browsers (e.g. Lynx) we will focus on modern browsers (e.g. Apple Safari 5, Google Chrome 16, Mozilla Firefox 10) that cover a wide range of novel HTML5 features.

In the age of tablets and smartphones, the third challenge – supporting a myriad of web-enabled devices – becomes even more important since an ever increasing share of web traffic is generated by tablets and mobile devices. Because awareness widgets are part of the application's user interface, device aspects such as screen size and interaction mode (e.g. touch, mouse or keyboard interaction) are the crucial ones which have to be considered by a generic awareness solution.

3 Generic Awareness Infrastructure

In this section, we introduce an approach enabling application-agnostic awareness support which is materialized by the generic awareness infrastructure. Furthermore, we expose a specific development blueprint for developing reusable awareness widgets.

To devise an approach for generic awareness support, we set out to identify an abstract editor architecture and carved out the editor components depicted in Figure 2(a). All web-based editors, except plug-in based solutions, adhere to this architecture that divides the application stack in application-specific and application-agnostic artifacts. While the editor components (e.g. the user interface) and the associated editor APIs are specific to each editor, the standardized W3C APIs (e.g. DOM API [5]) and the underlying document object model (DOM) are application-agnostic. Awareness widgets using editor APIs directly (e.g. to keep track of document changes) turn out to be application-specific. In

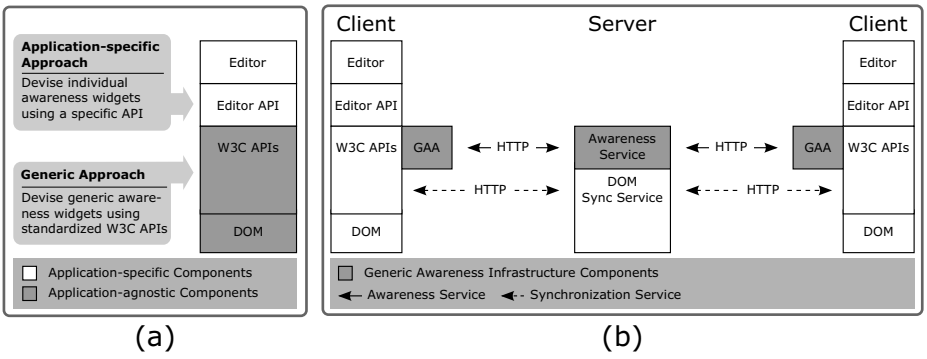

Fig. 2. a) Abstract architecture of web-based editors as well as approaches to anchor awareness support and b) Overview of the generic awareness infrastructure for web applications

contrast, awareness widgets leveraging the standardized W3C API layer (cf. Figure 2(a)) are application-agnostic and consequently capable of serving multiple web editors.

Therefore, we claim that anchoring awareness support at a standardized layer is the key to create an application-agnostic solution. Moreover, taking into account that W3C APIs are implemented in most modern browsers for PCs, tablets or even smartphones and that recently developed collaborative web-applications are predominantly standards-based, we conclude that the aforementioned challenges can be overcome.

The proposed approach to link awareness support to well-established W3C APIs is embodied in the generic awareness infrastructure illustrated in Figure 2(b). The distributed collaboration system consists of a server and an arbitrary number of clients. Clients comprise the very same application stack shown in Figure 2(a) associated with one additional component denoted as the generic awareness adapter (GAA). The GAA comprises registered awareness widgets and is devoted to execute three essential tasks:

1. Initializing registered awareness widgets.
2. Pushing collected awareness information from registered awareness widgets to the server.
3. Receiving, interpreting and eventually visualizing awareness information from other clients by means of registered awareness widgets.

To distribute awareness information among all clients, the central server propagates the respective data sets. Additionally, the server provides concurrency control services encompassing the synchronization of various DOM instances as well as a conflict resolution mechanism which is able to resolve potential conflicts arising if numerous participants change the same document artifacts (e.g. a graphic or a phrase) simultaneously. Even though the generic DOM synchronization service is a crucial part of the collaboration system, details which were specified in [7] are beyond the scope of the paper.

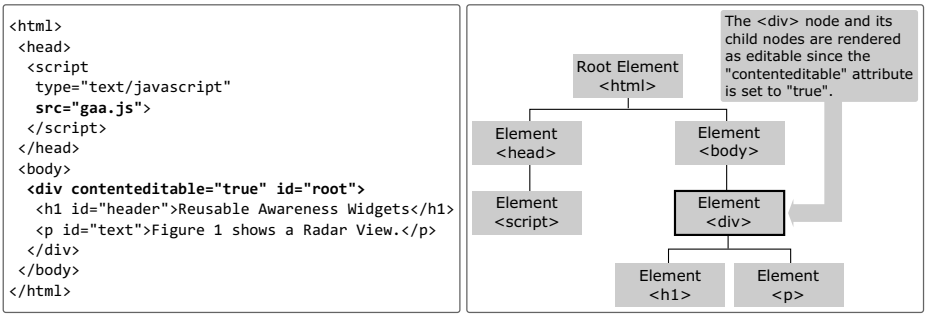

Fig. 3. HTML page of a minimal web editor and the corresponding DOM tree

3.1 Generic Awareness Adapter

The generic awareness adapter depicted in Figure 2(b) is the key component of the GAI and primarily in charge of accommodating awareness widgets as well as providing essential services to those widgets. Arbitrary widgets (e.g. telepointers, radar views) that adhere to a specific development blueprint (cf. Section 3.2) can be registered at the GAA. Once the registration is successful, awareness widgets have to be initialized before they actually capture and visualize awareness information.

In the following paragraphs, we illustrate the integration of the GAA into existing web applications and describe the setup as well as the operation's mode of the GAA.

Integration: The GAA pursues a lightweight integration approach to ease the process of converting existing editors to awareness-enabled editors. To accomplish the integration, a specific JavaScript file named "gaa.js" has to be embedded in the HTML code of the original web application (cf. Figure 3). The awareness integration process operates non-invasively meaning that the original JavaScript source code is not subject to changes. That implies that developers don't have to become familiar with the internal source code of the original application and can thus accomplish the integration of awareness support in a lightweight and rapid fashion.

Setup: Once the "gaa.js" is successfully embedded in the HTML application and the browser loaded the modified collaborative application, the GAA will be initialized. The initialization phase comprises the following tasks:

- Setup an HTTP communication channel connecting client and server to allow pushing and receiving awareness information.
- Establish session and user management (e.g. assign a color to each participant to create a color coding adoptable by telepointers and other widgets).

- Generate global identifiers for all relevant DOM nodes to have a uniform referencing mechanism (e.g. a newly inserted text node can therefore be addressed using the very same identifier at all sites).
- Initialize registered awareness widgets.

Operation: After the initialization, the GAA switches into its operation's mode where awareness information captured by different widgets is serialized and propagated to the server. Besides sending awareness messages, the reception of awareness-related data is also accomplished by the GAA. Received data is deserialized and forwarded to registered awareness widgets. All message exchanges are based on the JavaScript Object Notation (JSON) which is beneficial since it is a standardized format [8] with standardized methods for serialization (`JSON.stringify()`) and deserialization (`JSON.parse()`). Note that messages sent from the GAA are propagated by the server (cf. Figure 2(b)) to all clients except the sender client.

3.2 Awareness Widget Blueprint

The awareness widget blueprint serves as a guideline for awareness widget developers. We have adopted this blueprint for our widget development but it may also serve other web developers. The blueprint divides each widget in three components (initialization, capturing and visualization component) and reassures that implemented widgets are reusable. The widget implementation is illustrated providing a concrete example revisiting the minimal text editor shown in Figure 3. The text editor allows to modify the text of the `<h1>` and `<p>` elements because the `contenteditable` attribute of the embracing `<div>` element is set to `true`.

Initialization Component: As stated before, the blueprint divides each awareness widget into three building blocks whereas the initialization component is in charge of the following tasks:

- Visualization Setup: Create a visualization context for the awareness widget and render an initial visualization.[1]
- Event Listener Registration: Add event listeners to the awareness root (identified by the GAA) to record modifications in the shared workspace.

Setting up the initial visualization for awareness widgets demands the creation and positioning of an additional `<div>` container that acts as an encapsulated awareness model which is not interwoven with the actual document model. All

[1] The Separation of Concerns (SoC) principle [9] has to be enforced to prevent synchronization issues. The document model accommodating the content artifacts (e.g. text or graphical shapes) is subject to continuous synchronizations to assure all participants are working on a consistent document. In contrast, the visualization of awareness widgets is client-specific (e.g. the radar view depends on the local scrolling position) and therefore should be encapsulated and excluded from all sync processes.

awareness-related visualizations (e.g. the telepointer cursor, the input highlighting, etc.) are drawn onto this special overlay. Besides the creation of the visualization layer, additional tasks required by specific awareness widgets are executed upon request. For instance, a radar view widget might copy a DOM subtree into its visualization layer (i.e. the extra `<div>` element) to build a miniature view or a highlighting widget for SVG [10] tools might clone the SVG root element to build up a special SVG tree for shapes highlighting remotely created shapes.

To keep track of modifications in the shared workspace, awareness widgets have to register event listeners. As mentioned before, compatibility with the majority of web applications has to be ensured in order to devise a reusable solution. Therefore, awareness widgets directly leverage standardized DOM Events [6]. We identified three groups of DOM events (1) mouse events (e.g. `click`, `mouseover`), (2) keyboard events (e.g. `keydown`, `keyup`) and (3) mutation events (e.g. `DOMNodeInserted`, `DOMAttrModified`) that are relevant since they trigger important awareness-related application updates. For example, semantic cursors have to be adapted upon `DOMCharacterDataModified` events, telepointer positions have to be updated on `mousemove` events and document history widgets have to be refreshed if `DOMNodeInserted` events are fired. The setup of event handlers can be accomplished by means of the `element.addEventListener(...)` method.

Suppose we want to enrich the text editor depicted in Figure 3 with a primitive awareness widget that highlights newly entered characters at all remote clients. This entails the following implementation tasks. First, an additional `<div>` container encapsulating the visualization artifacts has to be created which is straightforward (`document.createElement("div")`). Second, the insertion of characters has to be monitored and therefore event handlers have to be attached to the `<div>` node which is illustrated in Figure 4(a). Note that the listener registration does not require to add listeners to each individual DOM node since installed event listeners also listen to changes of the respective child nodes. In our example, the attached event handler would also listen to modifications of the `<h1>` or `<p>` node.

Capturing Component: After the initialization, local changes are recorded by a capture component that gathers changes for dedicated awareness widgets. The main objectives of the capture component are:

- Awareness Information Filtering: Retrieve essential awareness information from the vast set of data provided by registered event handlers
- Data Preparation: Prepare relevant awareness information for the message transfer.

Gathering and filtering awareness-related information is accomplished by event handlers registered during the initialization. As soon as event handlers are called, awareness information is prepared according to the requirements of awareness widgets. In some cases, the information capturing is trivial since `Event` objects [6] directly expose the required properties. For example, a telepointer could capture

```
captureChanges = function() {
 range = document.getSelection().getRangeAt(0);
 //serialize range information into JSON
};

document.getElementById("root").addEventListener(
 "DOMCharacterDataModified", captureChanges, true);
```

(a) Initialize Implementation

```
{
 start : {
  parentNode : range.startContainer.parentNode.id,
  relPos : getRelativePos(parentNode, startContainer),
  offset : range.startOffset
 },
end : {
  parentNode : range.endContainer.parentNode.id,
  relPos : getRelativePos(parentNode, endContainer),
  offset : range.endOffset
 }
}
```

(b) Capture Implementation

```
highlightChanges = function (json) {
 startContainer = resolveNode(json.start.parentNode, json.start.relPos);
 endContainer = resolveNode(json.end.parentNode, json.end.relPos);

 range = document.createRange();
 range.setStart(startContainer, json.start.offset);
 range.setEnd(endContainer, json.end.offset + 1);

 rect = range.getBoundingClientRect();

 div = document.createElement("div");
 style = {
  "top" : rect.top,
  "left" : rect.left,
  "height" : rect.height,
  "width" : rect.width,
  "background" : red,
  "pointer-events" : none
 }
 div.css(style);
 document.body.appendChild(div);
};
```

(c) Visualize Implementation

Fig. 4. Minimal awareness implementation capable of highlighting local text changes at all remote clients

the X- and Y-coordinates by retrieving the `screenX` and `screenY` attributes from the `MouseEvent` object. However, this is only appropriate for strict what you see is what I see (WYSIWIS) tools [11] where all clients share the same window size, viewport, etc. In relaxed WYSIWIS environments [11] where participants might have different viewports, zoom levels, etc., the information capturing is much more complex and cannot leverage window coordinates (e.g. highlighting a word at zoom level 100 covers a different screen area than highlighting the same word at a zoom level of 200 percent). Therefore, advanced mechanisms are required to calculate positions. A robust way to capture fine grained positioning values is offered by the HTML Editing API [12]. It defines selections in HTML documents that can be obtained using the `window.getSelection()` method. The call returns a `Selection` object that exposes either the caret position or the text selection potentially spanning across multiple elements. It can comprise one or more `Range` objects [13] (indicated by the `rangeCount` property). Each `Range` object represents a contiguous part of the selection. In order to reconstruct selections or caret positions, the start and end of every `Range` object have to be transmitted to other clients.

The preparation of update messages is the second important capture task. Awareness information has to be serialized before the data transmission can take place. Therefore, JSON-compliant objects are employed as data containers combining all relevant awareness information. These JSON objects are then passed to the GAA which eventually serializes these objects and sends out awareness update messages.

Regarding the example of the minimal text editor, the capture mechanism would be triggered upon text modifications affecting the `<h1>` or `<p>` node. This capture mechanism is defined in the `captureChanges` function depicted in Figure 4(a). First, the `getSelection` method retrieves a list of `Range` objects representing currently selected DOM elements. If the caret resides in the `<h1>` or `<p>` node, there is only one `Range` instance that is retrieved through

`getRangeAt(0)`. This `Range` instance is exploited to create a JSON object as illustrated in Figure 4(b). The JSON message contains information about the affected node (`id`), the caret position (`offset`), etc. After the message construction, the JSON string is transferred to the server.

Visualization Component: The third building block of the proposed blueprint is the visualization component which processes and eventually renders incoming awareness information. In detail, this component carries out the following tasks:

- Awareness Information Processing: Distribute, interpret and render received awareness information.
- Awareness Information Re-Rendering: Refresh UIs of awareness widgets upon local change events.

Awareness widgets receive its data via JSON-compliant data exchange objects that were created by the GAA during the deserialization of awareness messages. Data exchange objects contain awareness information collected by the capturing component. For example, a data object dedicated for an input highlighting widget might carry information about the captured range of newly inserted characters. If this information has been passed to the specific widget, the visualization process can start. First, a new `Range` object has to be created (`document.createRange()`). Second, the start and end of the range have to be set invoking `range.setStart(startNode, startOffset)` and `range.setEnd ( endNode, endOffset)` respectively. The start and end nodes can be obtained via `document.getElementById(...)` using the identifiers stored in the data exchange object. After these two initial steps, the visualization engine can profit from the rich APIs specified in the CSS Object Model Standard [4]. It enriches existing DOM interfaces like `Document`, `Window` or `Element` with sophisticated functions like `caretPositionFromPoint()`, `getClientRects()`, etc. The `range.getClientRects()` method, for instance, returns a collection of `ClientRect` objects that serve as a representation for the range's occupied screen area. Each rectangle exposes read-only `left`, `top`, `right` and `bottom` properties. These properties and its assigned values are used as CSS properties for the established `<div>` overlay element. Note, that this `<div>` element can be styled according to your application's look and feel since solely CSS properties have to be changed. The defined procedure ensures the correct handling of relaxed WYSIWIS situations, because abstract awareness information is interpreted locally and therefore adapted to the local environment (e.g. zoom level, viewport, etc.). For graphics tools a rectangular highlighting of the modified DOM element might not be appropriate. In particular, inline SVG graphics embedded in the DOM tree require advanced highlighting mechanisms. A compelling way to highlight SVG elements is to first clone the affected SVG element to the `<div>` overlay container of the corresponding awareness widget. Afterwards the cloned SVG element can be styled, i.e. its properties (e.g. fill or stroke color) are cleared and then a new stroke is created. Setting the `stroke-width` and `stroke-color` properties completes the sophisticated SVG highlighting.

If local changes occur (e.g. window size modifications, scrolling, etc.) the awareness visualization has to be re-rendered to adapt to the updated environment. To keep track of those local changes, additional event listeners have to be registered while initializing the awareness widget.

In the introduced example, the simple awareness widget has to highlight the characters recently entered by the remote user. Figure 4(c) illustrates the required steps. In summary, a new rectangular `<div>` element is constructed that has the same dimensions and coordinates as the newly created characters. Dimensions and coordinates fetched from the deserialized `Range` object are applied to the created `<div>` by assigning a CSS style.

4 Validation

To assess the reusability of awareness widgets which were devised leveraging the GAI approach, we opted for a two-step validation. First, we implemented three example widgets according to the presented architecture blueprint (cf. Section 3). Second, we incorporated these awareness widgets bundled with the configured GAA into four collaborative web editors.

In the first step aiming to produce exemplary awareness widgets – due to resource restrictions – we had to choose three awareness widgets from the multitude of common widgets. Therefore, we based our selection on a classification dividing widgets in *extrinsic* and *intrinsic* ones. *Extrinsic awareness widgets* are encapsulated in a single UI container and do not intermingle with the UI representing the document content (e.g. participant list, radar view or document history). *Intrinsic awareness widgets* are intermingled with the UI representing the document content (e.g. input highlighting, semantic cursor or telepointer). Besides taking into account the classification, we also wanted to cover the prevalent application domains (i.e. shared editing and shared drawing). Therefore, we decided to build widgets exposing the following characteristics: (1) intrinsic for shared editing as well as (2) intrinsic for shared drawing and (3) extrinsic for arbitrary collaborative applications. Correspondingly, we developed (1) an input highlighting widget for text editors as well as (2) an input highlighting widget for graphics editors and (3) a generic participant list.

To test the three developed awareness widgets, we set out to find collaborative editors lacking awareness support. Existing multi-user editors were not appropriate since they already offer awareness features to some extent. Hence, we chose to convert available single-user applications into multi-user applications and leveraged the transformation approach described in [7] that produces shared editing tools featuring document synchronization and conflict resolution. The produced collaborative web editors were suitable test applications since they did not provide any awareness support. According to our proposition, we transformed editors from different application domains including two text editors and two graphics editors. In the following paragraph, we briefly introduce the four web applications that were successfully converted to collaborative applications.

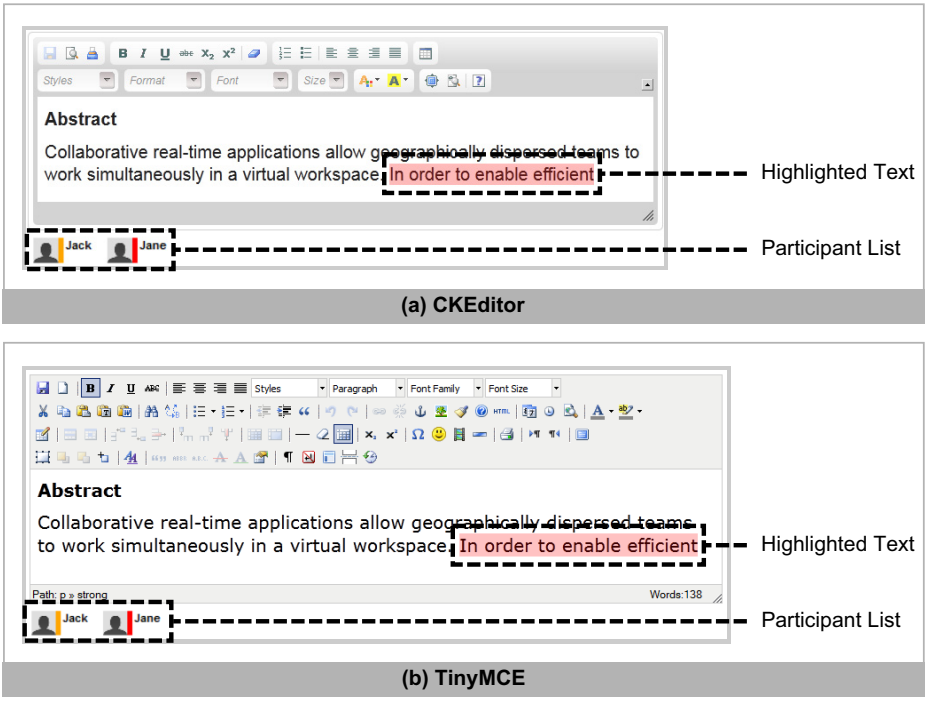

Fig. 5. User interfaces of the web-based text editors CKEditor [14] and TinyMCE [15] enriched with awareness support (changes by the remote user Jane are highlighted red)

CKEditor [14] and *TinyMCE* [15] are two popular web-based text editors offering common features such as text formatting, image insertion or text alignment. Both editors were enhanced with the very same participant list widget as well as an input highlighting widget (cf. Figure 5). Input highlighting is accomplished by adding a colored overlay to newly created characters for a certain period of time. The color overlay corresponds to the color coding established in the participant list. *SVG-edit* [16] and *FNISVGEditor* [17] are editors for scalable vector graphics providing common graphics tools to accomplish reoccurring drawing tasks such as create lines, ellipses or rectangles. Both editors incorporated a participant list widget and an input highlighting widget (cf. Figure 6). Note that the input highlighting widget differs from the text input highlighting. In contrast to emphasizing newly created characters, in this case, recently created shapes (e.g. circles, rectangles) are subject to highlighting.

Eventually, we could show that it is feasible to reuse awareness widgets by incorporating them non-invasively in four distinct collaborative applications. The editor screenshots depicted in Figure 5 and Figure 6 demo the achieved awareness support. Furthermore, the resulting collaborative editors are demonstrated on our GAI demo page `http://vsr.informatik.tu-chemnitz.de/demo/GAI/`. Note that during the widget integration some issues were encountered. One class

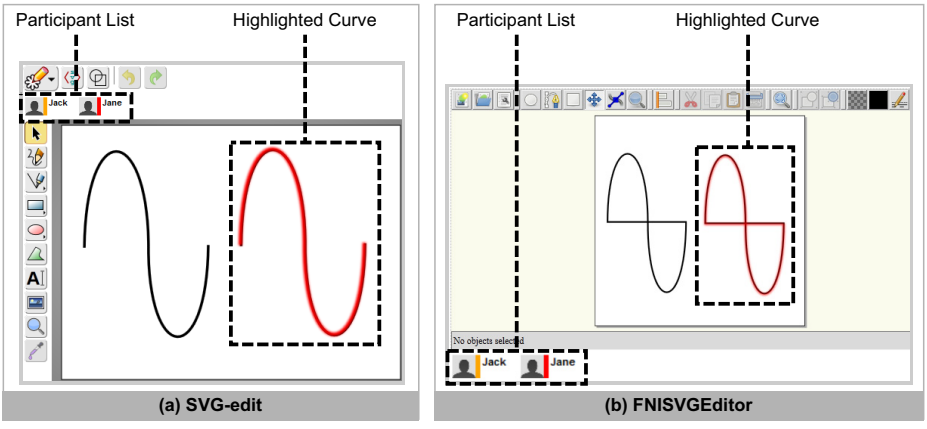

Fig. 6. User interfaces of the graphics editors SVG-edit [16] and FNISVGEditor [17] enriched with awareness support (changes by the remote user Jane are highlighted red)

of issues was related to the positioning of highlighted ranges. Since the document viewport establishes an extra coordinate system that is embedded in the browser window coordinate system, coordinate calculations have to take offsets into account. Another identified issue was discovered incorporating the participant list widget. The proposed approach to draw the participant list on an extra overlay layer requires an empty window portion which is not always the case. Therefore, the participant list has to be embedded directly into the application layer. The outlined issues could all be solved configuring the GAI accordingly.

5 Discussion

In our validation, we illustrated the strength of the GAI approach. A major advantage is the provisioning of awareness support at infrastructure level. Once implemented, the awareness features supported by our GAI are reusable by numerous applications from different domains. Moreover, the creation of awareness widgets is also simplified. Widgets can exploit awareness information supplied by the GAA and mediated by the GAI. Therefore, development effort and time for awareness widget implementations are significantly reduced in contrast to conventional approaches.

We also showed in our validation the non-invasive integration of awareness widgets into existing collaborative applications. Extrinsic as well as intrinsic widgets were successfully integrated without having to change the source code of the original application. Moreover, the devised awareness widgets were adopted in different applications of the same domain and even across domains which demonstrates the high reusability of GAA widgets.

However, the gained experience developing numerous GAA-compliant awareness widgets revealed two critical limitations that are immanent to the proposed GAI approach:

Application Model not Represented in the DOM: The defined GAI implementation and in particular the information capturing components rely on standardized DOM events that are fired if the DOM is manipulated. If this notification mechanism is somehow bypassed and not actively producing events anymore, the GAI information tracking cannot operate properly and eventually, registered awareness widgets are affected since required awareness information is not supplied. For example, if one part of the application is implemented using a plug-in technology (e.g. Adobe Flash or Microsoft Silverlight), changes affecting a plug-in internal data structure do not emanate DOM events and thus awareness information cannot be retrieved. Therefore, it is mandatory that the editor document and its content (e.g. the text of a text editor or the shapes of a graphics editor) are represented as a part of the DOM.

Cross-Browser Inconsistencies: Differing browser engines (e.g. Apple Safari, Google Chrome, etc.) are not fully consistent with respect to their model representation (i.e. the DOM) even though they request and render the very same serialized HTML file (e.g. an element is represented as one single node in one browser engine and as multiple nodes in another browser engine). This can break the global identification scheme and impair the awareness information association. For instance, adding a line break to a text embedded in an HTML `textarea` results in a `Text` node split. Removing this line break once again is handled differently by different browser engines. While some engines merge the text nodes, other browser engines keep two separate text nodes.

Even though the GAI approach entails some limitations, we argue that a standards-based solution such as the GAI can efficiently tackle the aforementioned challenges (cf. Section 2). In particular in the light of the HTML5 movement where standards are aggressively pushed and rapidly adopted.

6 Related Work

Our GAI approach is related to work in two major research domains, namely full-fledged collaboration frameworks and user interface (UI) toolkits accommodating also awareness support.

UI toolkits like the Multi-User Awareness UI Toolkit (MAUI Toolkit) [18], WatchMyPhone [19] or GroupKit [20] provide sets of awareness-ready UI components and also facilitate document synchronization. Most of the approaches are tailored to a particular runtime environment. While the MAUI Toolkit targets the Java runtime, WatchMyPhone is a solution dedicated for the Android platform. Even though some toolkits encapsulate functionality like the distribution of awareness information into generic components, there is a tight coupling between UI controls and awareness support. Thus, reusability of awareness widgets is achieved at the design phase rather than at the runtime phase. Developers have to become familiar with the applications' source code and eventually are asked to replace standard UI controls with their collaborative counterparts. In contrast, our non-invasive GAI approach allows to incorporate awareness features in a rapid and cost-efficient manner since it only involves the integration of an extra JavaScript file without requiring source code changes.

Advanced frameworks for the development of collaborative applications like Apache Wave [21], BeWeeVee [22] or CEFX [23] focus on the provisioning of concurrency control mechanisms but neglect the aspect of awareness support. In our approach the generic awareness support is embedded into the GAI which decouples the UI layer from the awareness support. This increases reusability of awareness support and results in reduced effort for developing awareness widgets on top of the GAI.

7 Conclusion

Workspace awareness is a key feature for collaborative real-time applications enabling effective collaborative work. At the present time, well-established and pervasively available collaborative web applications like Google Docs implement awareness features in an application-specific manner, even if the same set of awareness widgets could be shared among various applications. As a result, the time and resource consuming task of implementing and testing awareness widgets is repeated again and again.

In this paper, we presented an application-agnostic approach for the creation of out-of-the-box awareness widgets which are reusable in collaborative web applications. Our solution is based on the idea to anchor basic awareness support at the application-independent level of standardized W3C APIs. The proposed generic awareness infrastructure captures information about user interactions at this generic level and mediates it to all participating users via a server hosting the awareness service as well as the concurrency control service. As part of the generic awareness infrastructure, generic awareness adapters are able to incorporate arbitrary awareness widgets which have to be developed following a predefined development blueprint.

To validate our approach, we implemented a set of awareness widgets which have been integrated into four collaborative web editors for text and graphics. As demonstrated in our validation, created awareness widgets cannot only be used for the development of new collaborative web-applications; in particular, they are tailored for the incorporation into existing ones. Since awareness widgets can be adopted within several applications of the same domain (e.g. input highlighting widget) or even across application domains (e.g. generic participant list) our approach ensures reusability of awareness features to a large extent.

In future work, we will extend the set of available awareness widgets to create a base for performance and user studies. Especially, the quality/impact of provided awareness features will be explored in detail.

References

1. Gutwin, C., Greenberg, S.: A Descriptive Framework of Workspace Awareness for Real-Time Groupware. Computer Supported Cooperative Work 11(3-4), 411–446 (2002)

2. Gutwin, C., Stark, G., Greenberg, S.: Support for Workspace Awareness in Educational Groupware. In: CSCL, pp. 147–156 (1995)
3. Sommerville, I.: Software Engineering, 9th edn. Addison Wesley (2010)
4. van Kesteren, A.: CSSOM View Module, http://www.w3.org/TR/2011/WD-cssom-view-20110804/ (working draft August 4, 2011)
5. Hors, A.L., Hégaret, P.L.: Document Object Model (DOM) Level 3 Core Specification (2004), http://www.w3.org/TR/DOM-Level-3-Core/
6. Schepers, D., Rossi, J.: Document Object Model (DOM) Level 3 Events Specification (2011), http://www.w3.org/TR/DOM-Level-3-Events/
7. Heinrich, M., Lehmann, F., Springer, T., Gaedke, M.: Exploiting single-user web applications for shared editing: a generic transformation approach. In: WWW, pp. 1057–1066 (2012)
8. Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational) (July 2006)
9. Hürsch, W.L., Lopes, C.V.: Separation of Concerns. Technical report (1995)
10. Ferraiolo, J.: Scalable Vector Graphics (SVG) 1.0 Specification (2001), http://www.w3.org/TR/SVG10/
11. Stefik, M., Bobrow, D.G., Foster, G., Lanning, S., Tatar, D.: WYSIWIS Revised: Early Experiences with Multiuser Interfaces. ACM Trans. Inf. Syst. 5, 147–167 (1987)
12. Gregor, A.: HTML Editing APIs, Work in Progress. http://dvcs.w3.org/hg/editing/raw-file/tip/editing.html (last update January 19, 2012)
13. Kesselman, J., Robie, J., Champion, M., Sharpe, P., Apparao, V., Wood, L.: Document Object Model (DOM) Level 2 Traversal and Range Specification (2000), http://www.w3.org/TR/DOM-Level-2-Traversal-Range/
14. CKSource: CKEditor - WYSIWYG Text and HTML Editor for the Web (2011), http://ckeditor.com/
15. Moxiecode Systems: TinyMCE - JavaScript WYSIWYG Editor (2011), http://www.tinymce.com/
16. Schiller, J., Rusnak, P.: SVG-edit - A Complete Vector Graphics Editor in the Browser (2011), http://code.google.com/p/svg-edit/
17. Leppa, A.: FNISVGEditor - JavaScript-based Online Editor for SVG Graphics (2010), http://code.google.com/p/fnisvgeditor/
18. Hill, J., Gutwin, C.: The MAUI Toolkit: Groupware Widgets for Group Awareness. In: Computer-Supported Cooperative Work, pp. 5–6 (2004)
19. Bendel, S., Schuster, D.: Providing Developer Support for Implementing Collaborative Mobile Applications. In: Third International Workshop on Pervasive Collaboration and Social Networking, PerCol 2012 (2012)
20. Roseman, M., Greenberg, S.: Building Real-Time Groupware with GroupKit, a Groupware Toolkit. ACM Trans. Comput.-Hum. Interact. 3, 66–106 (1996)
21. Apache Software Foundation: Apache Wave (2011), http://incubator.apache.org/wave/
22. BeWeeVee: BeWeeVee - Life Collaboration Framework (2011), http://www.beweevee.com
23. Gerlicher, A.: Collaborative Editing Framework for XML (2009), http://sourceforge.net/projects/cefx/