Seamless Integration of Heterogeneous UI Components

Heiko Paulheim
SAP Research CEC Darmstadt
Bleichstrasse 8
64283 Darmstadt, Germany
heiko.paulheim@sap.com

Atila Erdogan
SAP Research CEC Darmstadt
Bleichstrasse 8
64283 Darmstadt, Germany
a.erdogan@sap.com

ABSTRACT
Component-based software engineering is a paradigm aiming at better ways to reuse existing code and to distribute work across teams. Integrating UI components developed with different technologies can be a difficult task which can quickly can lead to code-tangling and loss of modularity. In this demo, we present a prototype framework for integrating heterogeneous UI components, using RDF and formal ontologies for unambiguous event and data exchange and minimizing dependencies between integrated components. We will show an example from the emergency management domain using components written in Java and Flex and demonstrate tight, seamless integration, including dragging and dropping objects from Java to Flex and vice versa.

Author Keywords
User Interfaces, Integration, Component-based Software, Ontologies

ACM Classification Keywords
D.2.2 Software Engineering: Design Tools and Techniques—*User Interfaces*; D.2.13 Software Engineering: Reusable Software—*Reuse Models*

General Terms
Design, Algorithms

INTRODUCTION
The development of a system's user interface produces significant workload; the time devoted to UI development sums up to 50% of a system's total development time [14]. Thus, it is desirable to reuse existing user interface components in order to reduce development time. On the other hand, the reused components are supposed to seamlessly integrate with each other in the composed system.

We use the term *seamless integration* to indicate that UI components are not just rendered next to each other, but that the user can interact with them as if they were one coherent piece of software, as proposed by [2] and [26]. Seamless integration includes cooperative information visualization, dragging and dropping of information items between components, and so on.

UI components can be developed with numerous technologies. Therefore, it is likely that reusing existing UI components also involves the task of integrating *heterogeneous* components. Existing frameworks for UI integration are most often very limited in facilitating cross-component interaction [4], and the use of heterogeneous technologies further complicates integration.

In this demo, we introduce a framework capable of seamless integration of heterogeneous UI components while maintaining modularity. Based on a scenario from the domain of emergency management, we will show an application composed of both Java and Flex based components, and discuss how they can be integrated using our framework. We will show how events can be unambiguously exchanged between both kinds of components to produce a uniform behavior, and we will also demonstrate drag and drop from Java to Flex components and vice versa.

DEMO SCENARIO
In this demo, we show a scenario from the area of emergency management. The overall application consists of several components, each providing different relevant functionality, such as planning tasks, managing resources, visualizing maps, processing messages, simulating the expansion of floodings or fires, and so on. In this demo, we focus on two complex UI components (see Fig. 1): a *resource management component*, implemented in Flex, is used for browsing, viewing, and searching resources, such as fire brigade and ambulance cars, and a *mission account component*, implemented in Java, which is used for managing current and predicted problems, as well as for planning tasks to address those problems.

In this demo, we will show two central interactions with those components that require interoperability between those components. First, related information should be highlighted when selecting an object in one application, using the brushing-and-linking paradigm [7]. In our particular case, the resources allocated to a task should be highlighted when selecting a resource, and vice versa. Second, resources should be allocated to tasks by drag and drop from the resource management component to a task in the mission account component.

EICS'10, June 19–23, 2010, Berlin, Germany.

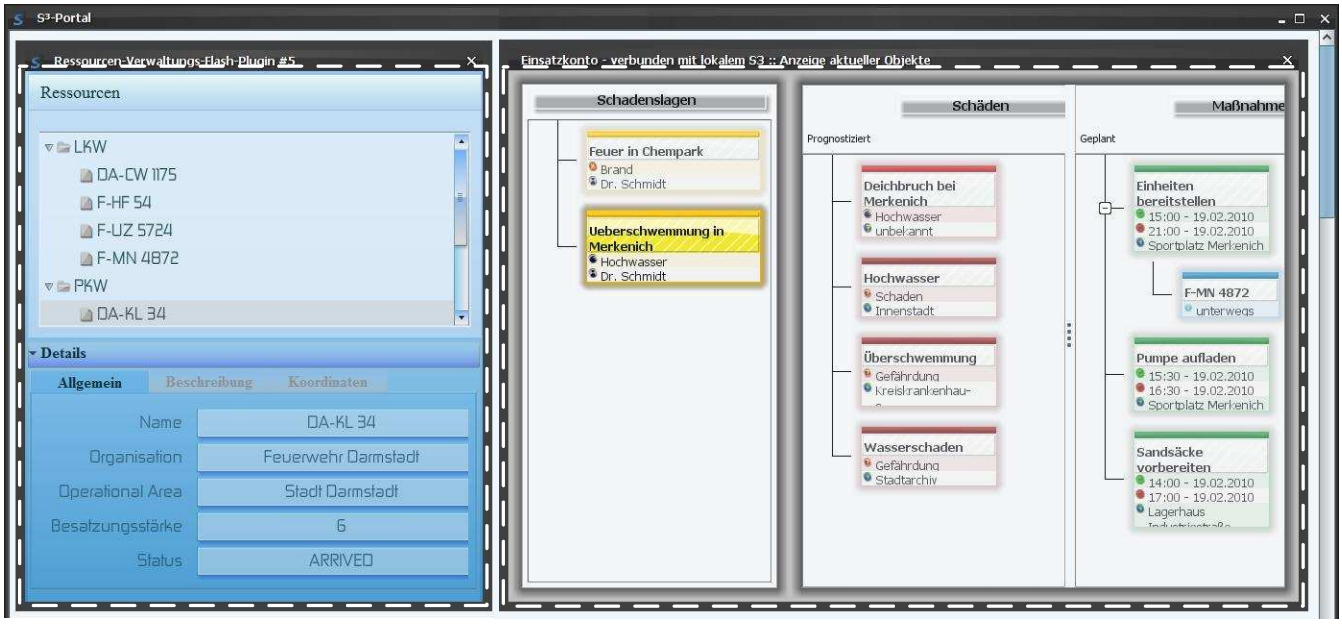

Figure 1. A screenshot with two UI components: a *resource management component* implemented in Flex (left) and a *mission account component* implemented in Java (right). The white dashed lines indicate the borders of the components, the dark frames are the containers displaying the components in a common frame.

LIBRARIES FOR LINKING FLEX AND JAVA

For implementing the integration framework, we have taken five libraries into consideration: *JFlash* [10], *JFlashPlayer* [19], *ComfyJ* [20], *DJ Project Native Swing* [5], and *JDIC* [11]. While JFlash is a pure Java implementation, JFlashPlayer and ComfyJ use an ActiveX bridge to Flex (and are therefore restricted to Windows), and DJ Project and JDIC embed Flex applications via a web browser (and are therefore platform independent). We have analyzed those libraries with regards to stability, flexibility in message exchange between Java and Flex, and API simplicity as well as availability of documentation. Table 1 sums up the results of this analysis.

Library	License	Stability	Flexibility in Message Exchange	API Simplicity	Documentation
JFlashPlayer	Commercial	+	+	+	+
ComfyJ	Commercial	+	+	-	+
DJ Project	Open Source	+	-	+	+
JDIC	Open Source	+	-	-	+

Table 1. A comparison of four APIs for linking Flex and Java.

JFlash is currently an Alpha version and probably discontinued (the last update dates back to 2006), therefore, we performed no further analysis with it. ComfyJ bears a massive coding overhead, since the ActiveX calls are not encapsuled, and with JDIC, the developer has to write additional JavaScript code to enable message exchange between Java and Flex applications.

Therefore, we have shortlisted only JFlashPlayer and DJ Project Native Swing for the implementation of our prototype. In more detailed tests, it turned out that drop events from Java to Flex cannot be captured precisely with DJ Project Native Swing (it can be detected that a drop event has occured *somewhere* within the Flex component, but not *where* exactly), thus narrowing the possibilities using this way of integration. Our prototype is therefore implemented with JFlashPlayer.

FRAMEWORK ARCHITECTURE

We have developed a prototype integration framework which allows the composition and seamless integration of UI components developed with different technologies. The aim was to achieve these aims while preserving *modularity* and *maintainability* of the integrated system, therefore, the dependencies between integrated components should be limited as much as possible.

Each of the integrated components runs in a container, which displays the component within a common frame (see Fig. 2). The containers are connected via an event exchange bus, providing services for interoperability, such as sending and receiving events, marshalling and unmarshalling objects in a common representation, and handling drag and drop between components.

Events can be exchanged both as directed component-to-component communication as well as by broadcasting. However, we encourage the use of broadcasting as this reduces the number of direct dependencies between components and helps preserving modularity of the overall integrated system.

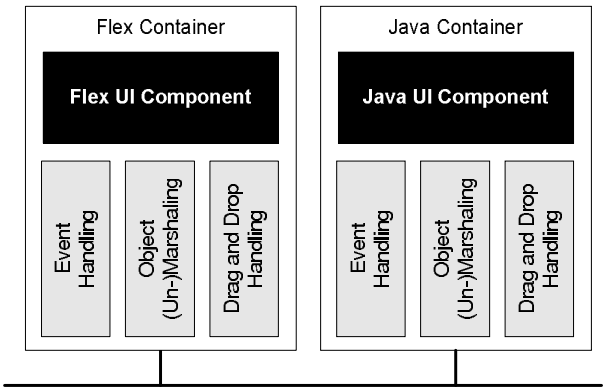

Figure 2. Different components run in containers. Containers are responsible for displaying components in a common frame, as well as providing interoperability services via an event bus.

Further, we allow both decentralized and centralized event processing [4], the latter based on a reasoning component [17]. In this paper, we will concentrate on the decentralized case.

There are specialized container implementations for each technology. Since the integration framework itself is implemented in Java, the container for Java components can be implemented quite straightforwardly. The container for Flex uses the JFlashPlayer library, as discussed above.

UNAMBIGUOUS EVENT AND DATA EXCHANGE

The integration of UI components requires a common formal model [4], and so does data exchange between those components. Since ontologies are a well-proven formalism for creating such models in integration tasks, we use ontologies modeling both the system and its UI components as well as the information objects they process.

Based on these ontologies, the instances of components, events, and data objects can then be represented in RDF [22]. Such a universal representation has been proposed as an *interlingua* for developing distributed systems, reducing the complexity of integrating N components from N^2 to N [21]. RDF can be serialized in different syntactic forms, including XML. Fig. 3 shows the serialization of an event and the data object contained therein in RDF-XML.

The use of RDF as an exchange format has significant advantages, compared to other approaches, such as using plain XML or JSON: an ontology is used to define each of the events and data objects exchanged and thus provides the exact semantics of each event exchanged between two components. Thus, there is a standardized knowledge base grounded on formal logics, which provides a mutual understanding of the events and avoids misinterpretations, e.g. by making hidden assumptions about the contents of a certain XML tag or attribute [23]. This is particularly necessary when using components created by different developers.

Two ontologies are used for defining the space of possible events exchanged between components [16]: An ontology of UIs and interactions defines the possible user actions (selecting, dragging, dropping objects etc.) as well as system actions (modifying, displaying, highlighting objects etc.) which can cause an event. An ontology of the system's real world domain (emergency management in the case of this demo) defines the information objects that may be involved in events.

With the help of these ontologies, the possible events are clearly defined and semantically unambiguous. The ontologies can be used as a part of the system specification, as they provide the agreed vocabulary and knowledge shared by all developers [6].

For exchanging events, each container's object marshaling and unmarshaling service can translate the events to RDF and back to its internal representation – Java objects in the case of Java components, Flex objects for Flex components. The first interaction case – linking selected objects – is implemented as follows:

When the user selects a task object in the mission account component, an event is created, providing the following statement encoded in RDF: "the user has selected an object which represents a task". The detailed data of the Java object itself (consisting of the task's ID, name, place, time, etc.) are also provided in the RDF encoding. To this end, the developer has to invoke a method on the container, providing the URI describing the event in the ontology. The further processing is performed by the framework. The event is then broadcast to and received by all other containers, using their event exchange services.

The resources component reads the event and decodes that a task object has been selected. The object contained in the event is converted from RDF to a Flex object. This object is then handed to the Flex application, which can find the related resource objects and highlight those. The component developer has to hook her code in here and implement a suitable reaction.

DRAG AND DROP WITH USER ASSISTANCE

Implementing drag and drop means an even tighter integration. When a drop action is sensed on some component, the dropped object's origin is most often a different component. In our scenario, this includes the cases that a Java object is dropped on a Flex component, and vice versa. We show a uniform solution for both directions in this demo.

In our framework, drag and drop is implemented with Java Swing, which allows transferring an object. The object transferred contains the RDF representation of the dragged object. Again, the drag and drop implementation for the Java container is quite straightforward.

In the Flex container, an invisible Java proxy is automatically created for each drag source and each drop target. The drag and drop service keeps track of the drag source and drop target components in Flex and their mapping to the proxies. When a drag action is sensed in the Flex component, the

Figure 3. Events are exchanged in a common RDF-XML representation. Developers may use the event inspector to analyze the event exchange in detail.

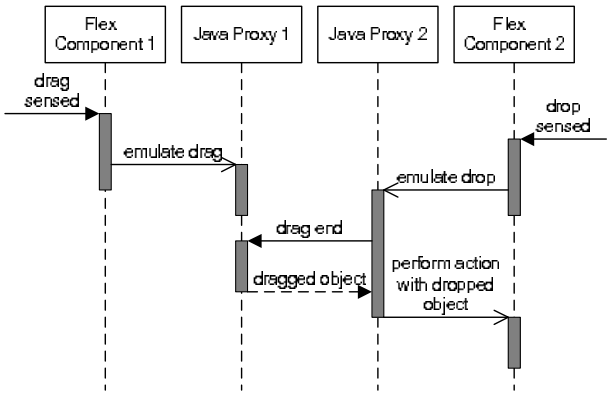

Figure 4. Drag and drop with non-Java components is implemented by creating Java proxies which handle drag and drop on the system level. With this mechanism, drag and drop from/to Java to/from non-Java components can be performed as well as between two non-Java components.

dragged Flex object is gathered, marshaled in RDF, and a drag action is emulated using the corresponding proxy component. When the object is dropped to a Java component, the effect is thus the same as if the drag would have started at a real Java component.

Likewise, when a drop action is sensed on a Flex component, a drop action is emulated on the corresponding Java proxy. With the help of this drop action, a drag end action is sent to the originating component, and the originally dragged object can be obtained, converted to a Flex object, and processed further. The mechanism also works if proxies are involved on both the drag and the drop side (i.e. for dragging and dropping from Flex to Flex or to another non-Java component), thus, the framework has one mechanism supporting every kind of drag and drop interaction (see Fig. 4).

In applications with many components, the user may not always know where a dragged object might be dropped, and with which effect. Similarly to the linking interaction case described above, components may broadcast events that an object has been dragged. Each component receiving this event may then analyze the event and the object contained therein. Based on this information, the component may then highlight the drop-sensitive area and provide a tooltip describing the type of action that a drop action would cause on that area, as shown in Fig. 5. Broadcasting another event when the drag action ends is necessary for putting the highlighted components back into a normal state.

RELATED WORK

There are different approaches for integrating UI components, including component and plugin based systems, web portals and mash-ups. Although they are valid approaches for simply embedding components in a common frame, the means for seamless integration, as described in this paper, are still rather limited [4]. Furthermore, despite the large variety of integration approaches, only a few explicitly address problems of cross-technology integration and often focus only on information visualization [15]. General approaches allowing for tight integration of heterogeneous components are only rarely seen.

Component and plugin based approaches most often expect the plugins to follow a certain architectural style and programming language. There are, however, a few examples explicitly directed at heterogeneous components, such as the OpenInterface workbench [13]. This platform, whose focus is on multi-modal interactions with different hardware de-

Figure 5. Drag and drop is made possible across components in different technologies. Additional user assistance is provided by highlighting possible drop targets upon a drag action and augmenting them with tooltips.

vices, allows components implemented with different technologies to communicate via exposed interfaces and provides tool support for creating the glue layer for making the components cooperate. The approach described in [12] focuses on integrating widgets from heterogeneous widgets platforms by using abstract models of those widgets and combines those models with MDA methods.

The CRUISe integration framework [18] uses the notion of *user interface services* for integrating different user interfaces, including Flex and HTML/JavaScript based components. Using a user interface definition language, they also focus on dynamic retrieval and binding of components, which is out of scope of our approach. The approach shown in [26], which is probably the closest to the one shown in this demo, uses wrappers for components implemented with different technologies, which are responsible for data conversion and communication. XML is used for communication between components using an event bus.

Portals are a technology for bringing together contents from different applications, encapsulated in so-called *portlets*. Portlets following standards such as the JSR 286 standard [9] can be integrated in different portal platforms and may contain components developed with different technologies. An overview of such platforms is given in [1]. Those platforms offer a large variety of functionality, such as Single Sign On and portlet layouting, and the 2.0 version of the JSR portlet standard also offers a basic event processing mechanisms between portlets, including user-defined events. Mash-ups differ from portals as they use less standardized, light-weight frameworks and rather address end users and semi-professional programmers than professional software developers. In [25], a survey of different mash-up platforms is given.

Unlike the work presented in this paper, most of those approaches use the exposed APIs of the integrated components, and the developer has to write code invoking one component from another one (either manually or supported by tools). This leads to code-tangling and an integrated system which not modular and therefore hard to maintain.

CONCLUSION AND OUTLOOK

The approach shown in this demo provides a uniform approach for integrating heterogeneous UI components using formal ontologies for event and data exchange. Modularity is maintained by avoiding direct calls between components. As discussed in [24], such an approach leads to more flexible and modular integrated systems. Furthermore, we claim that meaningful and misinterpretation-free integration can only be achieved with using formal semantic models of the exchanged data, instead of naming conventions and data schemas. Thus, we use RDF, grounded in ontologies, for event and data exchange. By using this inter-lingua, the complexity of integrating components implemented with N different technologies is reduced from N^2 to N. The developer can make use a framework with containers for different technologies which handle the coordination between heterogeneous components.

We have presented a framework for seamless integration of UI components developed using different technologies. In the demo, we have shown a real-world application from the domain of emergency management, where UI components written in Flex and Java have been integrated to form a consistent application. With our framework, seamless integration such as dragging and dropping objects from Flex to Java components and vice versa is possible, and presented a drag and drop mechanism with additional user assistance.

For simple interaction patterns, such as “When selecting a resource, highlight all the tasks that are assigned to this resource”, the components may easily determine the possible drop-sensitive areas and tooltips by themselves. If the interaction patterns become more complex, such as “When selecting a task, highlight all the free resources that are adequate for this task”, a vast amount of domain knowledge may be necessary for evaluating the additional conditions (in this example: finding adequate resources). As sketched above, our framework foresees centrally mediated event exchange using a reasoner for accounting for those cases. In [17], we have shown that even in such complex cases, the framework’s performance is still acceptable.

So far, we have only concentrated on integrated UIs that run on a single, standard desktop PC. Multi-modal UIs, e.g. combining hand-held devices with large display walls, are not yet considered in our approach. However, by adding formal ontologies of hardware devices, such as the FIPA device ontology [8], our approach could be extended to not only integrate UI components on different *software*, but also on different *hardware* platforms.

ACKNOWLEDGMENTS

The work presented in this paper has been partly funded by the German Federal Ministry of Education and Research under grant no. 01ISO7009.

REFERENCES

1. A. Akram, D. Chohan, X. D. Wang, X. Yang, and R. Allan. A Service Oriented Architecture for Portals Using Portlets. In *UK e-Science All Hands Conference*, 2005.

2. J. Amsden. Levels Of Integration - Five ways you can integrate with the Eclipse Platform. Technical report, Eclipse Corner, 2001. `http://www.eclipse.org/articles/Article-Levels-Of-Integration/levels-of-integration.html`, accessed 03/17/2010.

3. G. Calvary, T. C. N. Graham, and P. Gray, editors. *Proceedings of The 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'2009)*. ACM, 2009.

4. F. Daniel, J. Yu, B. Benatallah, F. Casati, M. Matera, and R. Saint-Paul. Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities. *IEEE Internet Computing*, 11(3):59–66, 2007.

5. C. Deckers. The DJ Project Native Swing Website. `http://djproject.sourceforge.net/ns/index.html`, 2009.

6. G. Dobson and P. Sawyer. Revisiting Ontology-Based Requirements Engineering in the age of the Semantic Web. In *International Seminar on Dependable Requirements Engineering of Computerised Systems at NPPs, Institute for Energy Technology (IFE), Halden*, 2006.

7. S. G. Eick and G. J. Wills. High Interaction Graphics. *European Journal of Operational Research*, 84:445–459, 1995.

8. F. for Intelligent Phyiscal Agents. FIPA Device Ontology Specification, 2002. `http://www.fipa.org/specs/fipa00091/index.html`.

9. S. Heppner. JSR 286: Portlet Specification 2.0. `http://www.jcp.org/en/jsr/detail?id=286`, 2008.

10. java.net. JFlash Website. `https://jflash.dev.java.net/`, 2006.

11. java.net. JDIC - JDesktop Integration Components Website. `https://jdic.dev.java.net/`, 2008.

12. D. Kotsalis. Managing Non-Native Widgets in Model-Based UI Engineering. In Calvary et al. [3], pages 313–316.

13. J.-Y. L. Lawson, A.-A. Al-Akkad, J. Vanderdonckt, and B. Macq. An Open Source Workbench for Prototyping Multimodal Interactions Based on Off-The-Shelf Heterogeneous Components. In Calvary et al. [3], pages 254–254.

14. B. A. Myers and M. B. Rosson. Survey on user interface programming. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202, New York, NY, USA, 1992. ACM.

15. C. North and B. Shneiderman. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, pages 128–135, New York, NY, USA, 2000. ACM.

16. H. Paulheim. Ontologies for User Interface Integration. In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 973–981. Springer, 2009.

17. H. Paulheim. Efficient Semantic Event Processing: Lessons Learned in User Interface Integration. In *Proceedings of the 7th Extended Semantic Web Conference*, 2010. To appear.

18. S. Pietschmann, M. Voigt, A. Rümpel, and K. Meißner. CRUISe: Composition of Rich User Interface Services. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Proceedings of the 9th International Conference on Web Engineering (ICWE 2009)*, Edition 5648, pages 473–476, San Sebastian, Spain, June 2009. Springer.

19. V. Software. JFlashPlayer Web Page. `http://www.jpackages.com/jflashplayer`, 2009.

20. TeamDev. ComfyJ Website. `http://www.teamdev.com/comfyj/`, 2010.

21. M. Uschold and M. Gruninger. Ontologies: Principles, methods and applications. *Knowledge Engineering Review*, 11:93–136, 1996.

22. W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-concepts/`, 2004.

23. X. Wang, R. Gorlitsky, and J. S. Almeida. From XML to RDF: How Semantic Web Technologies Will Change the Design of 'Omic' Standards. *Nature Biotechnology*, 23(9):1099–1103, 2005.

24. U. Westermann and R. Jain. Toward a Common Event Model for Multimedia Applications. *IEEE MultiMedia*, 14(1):19–29, 2007.

25. J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12(5):44–52, Sept.-Oct. 2008.

26. J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A framework for rapid integration of presentation components. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *WWW*, pages 923–932. ACM, 2007.