Ontology-based Modularization of User Interfaces

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

ABSTRACT

Modularization is almost the only feasible way of implementing large-scale applications. For user interfaces, interactions involving more than one module generate dependencies between modules. In this paper, we present a framework that uses ontologies for building UIs from independent, loosely coupled modules. In an example scenario, we show how that framework is used to build an application for emergency management.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User Interfaces*

General Terms

Design, Algorithms

Keywords

Ontologies, User Interfaces, Modularity

1. INTRODUCTION

Whenever building larger software applications, it is necessary to reduce their complexity so that the implementation remains a task which can be handled. The idea of modularizing software is almost as old as software development itself and goes back to the late sixties [20].

The user interface of a large, complex software application is most often large and complex itself, and developing a system's UI takes up to 70% of the overall development's effort [24]. It is therefore a desirable approach to divide the user interface into independent modules. Especially when working in a large, distributed team, such modularity is essential [22].

In this paper, we present an approach of modularization using coarser-grained, loosely coupled components, which we call *plugins*, to build an application's UI, where the plugins are independent of each other. We use a strict definition of *independent*, which states that no plugin's definition must contain a reference to another plugin. Such independence is achieved by using ontologies for formally describing plugins and their interactions.

The rest of this paper is structured as follows. After a brief survey on the relevant background on modularization, UI integration, and ontologies, we introduce a Java-based framework for defining and running user interface plugins defined by ontologies. In an example scenario, we explain how interactions of independent plugins are implemented. The paper continues with a review of related work and closes with a conclusion and an outlook on future developments.

EICS'09, July 15–17, 2009, Pittsburgh, Pennsylvania, USA.

2. DESCRIBING USER INTERFACE PLUGINS WITH ONTOLOGIES

In the past, several technologies for creating modular user interfaces have been proposed. Following [7], those can be roughly divided in *web-based* and *non web-based* techniques.

Modular web-based applications are most often implemented either as *mashups* or as *portals*. In both cases, web-based applications are composed of independent units, called *mashlets* [1] or *portlets* [27], respectively. Each of those units provides a certain view on some data, like a map, an address list, and so on.

A common way of modularizing non web-based or desktop applications is the use of *plugins*. Two approaches, *traditional* and *pure* plugin-based systems, can be distinguished [5]. In a traditional plugin-based system, a host application providing the main functionality can be extended by plugins, while in a pure plugin-based system, the core functionality is reduced to an engine for loading and executing plugins. Unlike in traditional plugin-based systems, the system's main functionalities are also implemented as plugins, not in a fixed host application [7]. One well-known for a pure plugin-based system is the software development environment *Eclipse*.

Whenever interactions between plugins – e.g. highlighting objects in one plugin which have been selected in another one – are to be defined, dependencies between plugins occur. Thus, many of the existing approaches either fail to provide appropriate means for interaction between plugins or give up the concept of loose coupling of plugins. The same holds for portlets and mashlets [7].

Integrating modular user interfaces requires that the plugins (or components, modules, portlets, whatever one likes to call the basic building blocks) and their interfaces are properly described. In [28], several requirements for such

plugin descriptions are listed. These descriptions have to be *formal*, *simple*, *human readable*, and *modular*.

Each of those requirements is met by *ontologies*. An ontology is often defined as "an explicit specification of a conceptualization" [14]. More precise definitions describe an ontology as "an explicit, partial account of a conceptualization" [15] or as "a set of logical axioms designed to account for the intended meaning of a vocabulary" [14] of a domain. By referring to logical axioms, the last definition already points out that an ontology is always a *formal* description.

More concretely, an ontology is a collection of classes and relations between those classes. By those classes and relations, the vocabulary of a domain, e.g. the domain of trucks, is defined. Examples for such relations are subsumption (e.g. "each truck *is a* vehicle"), equivalence (e.g. "truck *is the same as* lorry"), and domain-specific relations (e.g. "trucks *drive on* streets"). Everyone talking about trucks can refer to the identifiers from that ontology and thus, a common understanding can be achieved.

Two of the most popular languages for describing ontologies are OWL [19] and F Logic [2]. Both use concepts such as classes and subclasses, attributes, and relations as their basic building blocks and can therefore be regarded as being simple for software developers, who use those concepts in their everyday work. *Human readers* familiar with the underlying formalisms (set theory in the case of OWL, first order logic in the case of F-Logic) can read ontologies written in those languages.

Modularity has been one design principle for ontologies from the beginning [14], in the sense that new ontologies can be built by extending existing ones. With the framework described in this paper, we will also demonstrate an approach based on modular ontologies.

When integrating modular user interfaces (as well as when integrating data or applications), one problem to tackle is the assessment of syntactic and semantic heterogeneity [7]. The problem of semantic heterogeneity - the use of an ambiguous vocabulary, i.e. different terms for the same thing, and terms with more than one meaning - may be overcome by using ontologies [26]: as stated above, ontologies serve to define and disambiguate the meaning of a vocabulary. Therefore, using an ontology helps avoiding semantic heterogeneity.

These considerations have led to the development of a framework for defining and executing UI plugins described by ontologies.

3. IMPLEMENTING MODULAR USER INTERFACES WITH ONTOLOGIES

The framework discussed in this paper uses three kinds of ontologies: an *application ontology* defining the basic classes for describing UI plugins, a *domain ontology* defining the terms of the application's domain (such as banking, travel, etc.), and one *plugin ontology* per plugin using those two ontologies to define the actual plugin (see Fig. 1).

3.1 Application Ontology

The application ontology defines base classes such as `InteractiveComponent` and `Interaction`. An `InteractiveComponent` is a component which has some interactive behaviour, e.g. a list in which objects can be selected, a table in which objects may be dragged and dropped, and so on.

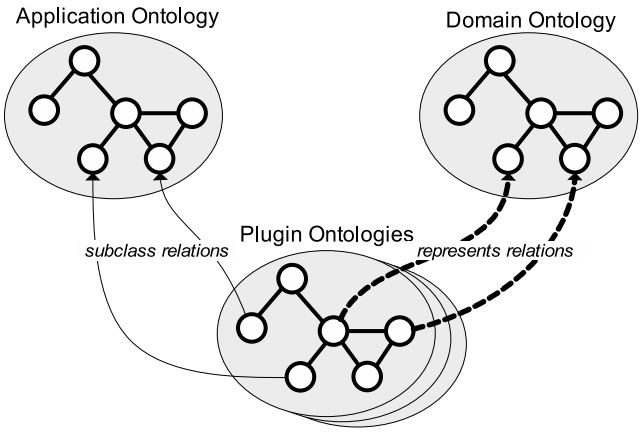

Figure 1: Ontologies for describing plugin-based user interfaces

The application ontology incidentally does not specify the types and appearance of components, such as buttons, tables, trees, and so on, in any more detail – for interactions between independent plugins, that kind of information is not relevant. More strongly worded: making such information about a plugin available to other plugins would contradict to the information hiding principle stated by Parnas [22] and thus to the idea of independent plugins.

An `Interaction` describes how an interactive component reacts to a user's actions. An interaction has two parts: a user action, like dragging, dropping, or selecting an object, and a system action which is triggered by that user action, such as displaying or modifying an object. As an example, an interaction definition for a map component displaying rescue units (such as fire brigade cars) may be: "If the user performs a `SelectAction` with a `Rescue Unit`, the `map component` will perform a `DisplayAction` with that `Rescue Unit`".

There are two aspects to point out in that definition. First, it defines several subclasses of the classes defined in the application ontology, i.e. (the `map component`, which is a subclass of the `InteractiveComponent` class in the application ontology), a new user action (a `SelectAction` with a `Rescue Unit`) and so on. In addition, classes from the domain ontology are referred to in order to define those new classes (here: `Rescue Unit`).

Second, the definition contains no reference to the plugin where the `Rescue Unit` is actually selected. Therefore, a `Rescue Unit` will always be focussed when selected in *any* plugin. Hence, the plugin containing the map component will automatically cooperate with all plugins where such objects can be selected.

However, the definition still needs a bit of refinement. The user working with the system does not deal with the `Rescue Units` themselves (as there are no fire brigade cars in the computer), but only with *information objects* which *represent* those units. This motivates the introduction of the `InformationObject` class and the `represents` relation. The above definition may then be correctly rephrased as follows: "If the user performs a `SelectAction` with an `InformationObject` representing a `Rescue Unit`, the `map component` will perform a `DisplayAction` with that `InformationObject`".

This distinction may seem a little nit-picky at first glance. However, there is some rationale behind it: As shown in Fig. 1, the only type of relation between the information system's description and the domain objects is the `represents` relation. This is facilitated by introducing the `InformationObject` class; the approach allows clean modularization and avoids mixing up the information system's model with the domain model. As the application ontology itself is domain-independent, it may be used for defining applications for arbitrary domains, each using a different ontology for describing the system's domain.

3.2 Architecture

While the application ontology is a fixed part of the framework, the domain ontology is dynamically linked to the application built with the framework. It is also possible to use more than one domain ontology when developing an application touching concerns from more than one domain. Each plugin is described in its own plugin ontology, which extends the application ontology (by defining interactions, user actions etc.) and which refers to the domain ontology. Therefore, the information describing the plugins is distributed across multiple ontologies and hence follows the idea of a modular description, as depicted above.

Figure 2 shows an overview of the framework's architecture. A central part is an event exchange mechanism. In order to make the UI plugins work together, it is necessary that they exchange data and messages. As has been argued in [28], the most promising way of implementing such an exchange is using events and a bus-based architecture. In our framework, the event exchange mechanism plays that role.

Each plugin is connected to that event bus, which means that it can send and receive both directed as well as broadcast events. As an extension of the Model-View-Controller (MVC) pattern [17], each plugin consists of (at least) four parts:

1. a user interface (or view in MVC), responsible for rendering the plugin and receiving the user's actions and input,
2. a class model (or model in MVC), which contains Java implementations of the domain classes processed by that plugin,
3. a business logic (or controller in MVC, which can range from very simple forms or data display to arbitrarily complex data processing, including the use of external components or services), and
4. a plugin ontology which defines the components and interactions of the plugin, as described above.

One important part of the framework is the reasoning component, which is responsible for processing the different ontologies and triggering the appropriate events. Since ontologies are based on well understood and examined formalisms, a variety of such reasoning components exist. We use the OntoBroker infrastructure [21] as an off-the-shelf component for processing and querying the ontologies. Both the ontologies and the queries are expressed in F-Logic.

User action events generated by a plugin's UI are read by the reasoner component. The reasoner queries the ontologies for components defining interactions triggered by this type of event, and notifies the respective plugins. In the terminology of [7], this is a centrally-mediated communication. That way of communication was chosen to leverage the degree of independence between plugins - no plugin sends a message to another plugin directly.

In our approach, we assume that each plugin may have their own class model for the objects processed, and thus, the same domain object may be modelled differently by different plugins. This design decision allows a greater possibility to reuse existing implementations, and, again, lowers the degree of dependencies between plugins. For being able to mediate between different class models, we introduce an object-to-ontology mapping registry (see Fig. 2). Each plugin can register mappings from its class model to the domain ontology, i.e., each class and each class' attribute (represented by a respective `getter` and `setter` method) can be mapped to a class or an attribute of the domain ontology. The registry can be used for looking up objects that are sent with events.

In the following section, we will show how the different parts - ontologies, reasoning component and the mapping registry - play together in creating an interactive system.

4. EXAMPLE SCENARIO

4.1 Setting

In this scenario, we look at the SoKNOS application for emergency management [10]. Two plugins are involved in the example. The first shows an inventory of available units from different organizations. The second is a geographic information plugin, which mainly consists of a map component. On that map, locations of problems and rescue units are displayed. The two plugins are shown in Fig. 3.

4.2 Domain Ontology

For developing an application for emergency management, an emergency management ontology is needed; in the SoKNOS project, the ontology described in [4] is used. This ontology defines classes and relations required for emergency management, such as incidents, damages, measures, and units.

4.3 Defining Interactions

We now assume that the two plugins, as mentioned above, are loaded and that the geographic information plugin shows the resources contained in the inventory plugin on a map. The first desired interaction is that resources selected in the inventory are automatically focussed on the map, as depicted in Figure 3, an interaction technique known as *linking* [11]. Therefore, like in the examples above, the geographic information plugin's ontology would contain a definition such as "If the user performs a `SelectAction` with an `InformationObject` representing something which has the `Quality` '`Position`', the map component will perform a `FocusAction` with that `InformationObject`".

Note that this definition does not use the class `Rescue Unit` explicitly. Instead, it makes use of the domain ontology, which defines that each `Rescue Unit` has a position. It will therefore also work for other things having a position, such as `Building` or `City`. Furthermore, if the domain ontology is extended while the system is developed further, and new plugins are implemented using new information objects which represent things having a position, the interaction definition for the map component does not have to be changed.

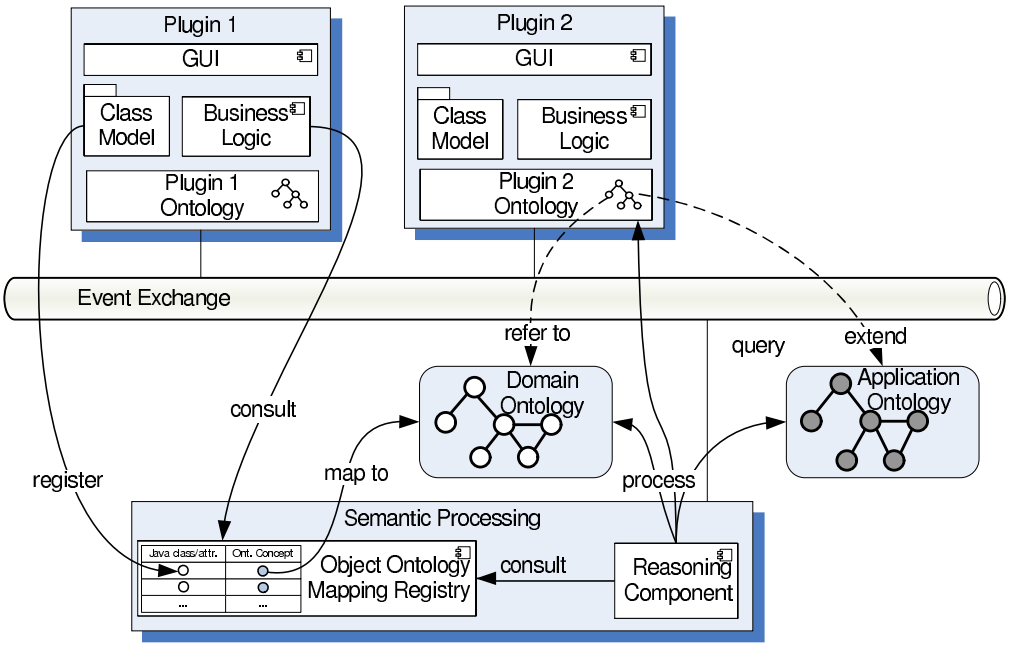

Figure 2: Framework architecture

It automatically supports interaction with every new and updated plugin, as long as that plugin commits to the same application and domain ontology.

4.4 Plugin Interoperation

When the user selects an entry in the inventory plugin, that plugin broadcasts an event stating that a `SelectAction` has been performed with that device entry. That event contains the plugin's original `DeviceEntry` object.

The event is received by the semantic processing component, which first has to decode the event. To that end, it consults the mapping registry and looks up the corresponding ontology class for the contained object, getting the answer that it is `Device`. In the next step, it formulates a query for the reasoning component, querying for components defining interactions which are triggered by a `SelectAction` with an `InformationObject` representing a `Device`. The query result contains the map component in the geographic information plugin, as well as the description of the respective system action.

The semantic processing component then sends an event to the geographic information plugin, indicating that the map component is supposed to perform a `FocusAction` with the `Device`. That event still contains the original object from the inventory plugin.

The geographic information plugin receives the event, consults the mapping registry to find out which attribute represents the contained position, invokes the corresponding getter method via Java's reflection mechanism, and focuses the map on the respective position.

This example shows that an interaction between plugins can be achieved based on ontologies without the need for direct references between those plugins.

5. RELATED WORK

In this paper, we have proposed the approach of using ontologies for describing UI components. Such abstract descriptions of UI components, also known as *user interface description languages*, have been well studied; a comprehensive overview of such languages is given in [25]. From the variety of such languages, there are two that come close to the ontologies-based approach presented in this paper: the eXtensible Interaction Scenario Language (XISL) [16] and the eXtensible Interface Markup Language XIML [23].

There are a few approaches using ontologies for the description of UI components. The work described in [6] and [12] propose ontologies for describing different types of user interfaces on a rather general level, such as characterizing different input and output devices. While we concentrate on the use of ontologies for supporting UIs, other approaches, like the ones described in [18] and [24], also use them for generating UIs, thus providing an MDA approach. The latter work proposes the use of modular ontologies, comparable to the ones described in this paper.

A few projects exist in which ontologies are used in building web portals and mashups. The approach described in [8] uses semantic web services, i.e., web services described by means of ontologies for communication between a portlet and its backend system. The work described in [9] uses ontologies to annotate the contents delivered by portlets. The authors of [3] propose an approach of using ontologies for building mashup applications to integrate contents from diverse annotated data sources. Interaction between portlets and mashlets is not covered by those approaches.

6. CONCLUSION AND OUTLOOK

In this paper, we have presented an approach for modularizing user interfaces, which are built from independent plugins. That approach aims at reducing dependencies between plugins. It was implemented in a domain-independent, Java-

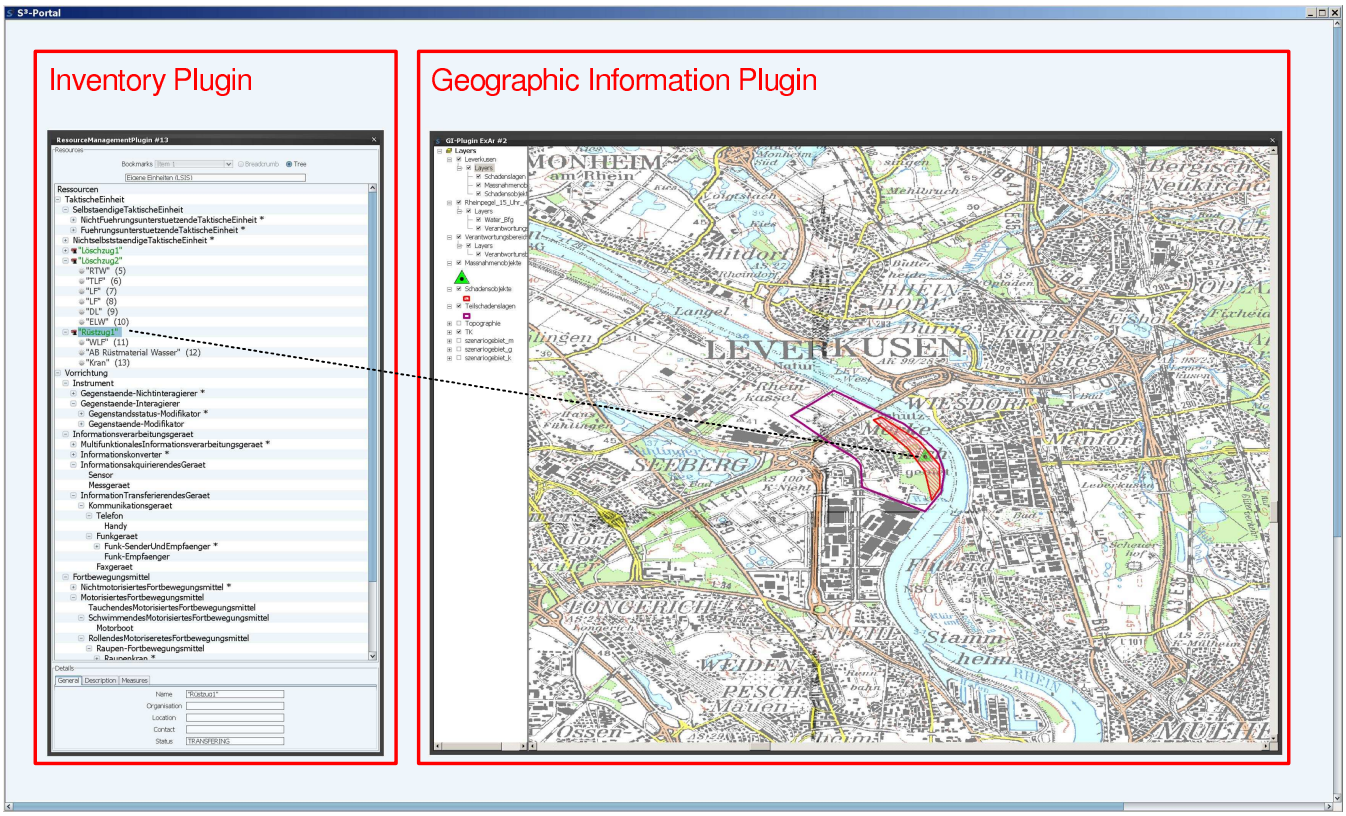

Figure 3: Screenshot of two plugins. Selecting resources in the inventory plugin causes the geographic information plugin to focus on those items' positions, indicated by the dotted line

based framework, which is currently used for an emergency management system. That system, consisting of twelve plugins, is developed by different teams distributed across eight locations, thus requiring a fairly large amount of modularity.

We use an application ontology to define plugins. That application ontology fits the requirements of a description of user interface components, as it is formal, simple, human-readable, and modular. It is complemented by a domain ontology describing the terms of the domain that an application is built for. Mappings between the plugins' class models and the domain ontology help facilitating data exchange without having to agree on a shared class model. As long as every module commits to the same application and domain ontology and contains a respective plugin ontology, interoperability between those plugins can be achieved by run-time ontology reasoning, even when they are updated or recombined.

Since the individual plugins are only connected by the semantic processing component based on ontologies, no information about the user interface plugins' implementation is needed in any other plugin. Therefore, the approach could potentially be extended to work with user interface modules written in different programming languages. Extending the framework for integrating plugins written in different programming languages will be one major area of future research.

We are well aware that both the application ontology and framework discussed in this paper have some potential for improvement. First of all, we have only considered pairwise interactions between two plugins so far. In the future, we strive at analysing more complex interactions as well. Restricting interactions according to users' rights is also a possible extension.

So far, our approach works for class models that have heterogeneous names for different concepts. Heterogeneities on the model level are not yet fully addressed. Formal frameworks as the approach described in [13] could extend our approach to be able to cope with those issues, too.

The use of ontologies for describing plugins provides the potential for user assistance. We have already implemented some first approaches in our framework, such as the augmentation of drop targets with tooltips generated from the ontology.

As user interfaces have to be reactive, great care has to be taken regarding the performance, especially of the semantic processing components. Therefore, we have to take into account highly performant reasoning components, and provide some additional measures, e.g. caches.

In summary, we have shown how ontologies can be used in the development of user interfaces. Our approach can help in modularizing user interfaces and in developing UIs from independent, loosely coupled plugins.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] S. Abiteboul, O. Greenshpan, and T. Milo. Modeling the Mashup Space. In *WIDM '08: Proceeding of the 10th ACM workshop on Web information and data management*, pages 87–94, New York, NY, USA, 2008. ACM.

[2] J. Angele and G. Lausen. *Ontologies in F-Logic*, chapter 2, pages 29–50. International Handbooks on Information Systems. Springer, 2004.

[3] A. Ankolekar, M. Krötzsch, T. Tran, and D. Vrandecic. The Two Cultures: Mashing Up Web 2.0 and the Semantic Web. In *WWW '07: Proceedings of the 16th International Conference on World Wide Web*, pages 825–834, New York, NY, USA, 2007. ACM.

[4] G. Babitski, F. Probst, J. Hoffmann, and D. Oberle. Ontology Design for Information Integration in Catastrophy Management, 2009. currently under review.

[5] D. Birsan. On plug-ins and extensible architectures. *ACM Queue*, 3(2):40–46, 2005.

[6] J. Coutaz, C. Lachenal, and S. Dupuy-Chessa. Ontology for Multi-surface Interaction. In *Proceedings of IFIP INTERACT03: Human-Computer Interaction*, pages 447–454. IFIP Technical Committee No 13 on Human-Computer Interaction, 2003.

[7] 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.

[8] T. Dettborn, B. König-Ries, and M. Welsch. Using Semantics in Portal Development. In *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering*, 2008.

[9] O. Díaz, J. Iturrioz, and A. Irastorza. Improving portlet interoperability through deep annotation. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 372–381, New York, NY, USA, 2005. ACM.

[10] S. Doeweling, F. Probst, T. Ziegert, and K. Manske. SoKNOS - An Interactive Visual Emergency Management Framework. In *Proceedings of the Workshop on Geographical Information Processing and Visual Analytics for Environmental Security*. Springer, 2009. to appear.

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

[12] Foundation for Intelligent Phyiscal Agents. FIPA Device Ontology Specification, 12 2002. `http://www.fipa.org/specs/fipa00091/index.html`, accessed 2008-01-22.

[13] C. Ghidini, L. Serafini, and S. Tessaris. On relating heterogeneous elements from different ontologies. In B. N. Kokinov, D. C. Richardson, T. Roth-Berghofer, and L. Vieu, editors, *CONTEXT*, volume 4635 of *Lecture Notes in Computer Science*, pages 234–247. Springer, 2007.

[14] T. R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. volume 43, pages 907–928, Duluth, MN, USA, 1995. Academic Press, Inc.

[15] N. Guarino, editor. *Formal Ontology and Information Systems*. IOS Press, 1998.

[16] K. Katsurada, Y. Nakamura, H. Yamada, and T. Nitta. XISL: a language for describing multimodal interaction scenarios. In *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*, pages 281–284, New York, NY, USA, 2003.

[17] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[18] B. Liu, H. Chen, and W. He. Deriving User Interface from Ontologies: A Model-Based Approach. In *ICTAI '05: Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, pages 254–259, Washington, DC, USA, 2005. IEEE Computer Society.

[19] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/, 2004.

[20] P. Naur and B. Randell. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO, 1968.

[21] ontoPrise. OntoBroker Website. `http://www.ontoprise.de/de/en/home/products/ontobroker.html`, 2009.

[22] D. L. Parnas. Information Distribution Aspects of Design Methodology. In *IFIP Congress (1)*, pages 339–344, 1971.

[23] A. Puerta and J. Eisenstein, 2001. `.http://www.ximl.org/documents/XimlWhitePaper.pdf`, accessed 2009-01-15.

[24] K. A. Sergevich and G. V. Viktorovna. From an Ontology-Oriented Approach Conception to User Interface Development. *International Journal "Information Theories and Applications"*, 10(1):89–98, 2003.

[25] N. Souchon and J. Vanderdonckt. A Review of XML-compliant User Interface Description Languages. volume 2844, pages 377–391. Springer, 2003.

[26] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based integration of information - a survey of existing approaches. In H. Stuckenschmidt, editor, *IJCAI–01 Workshop: Ontologies and Information Sharing*, pages 108–117, 2001.

[27] C. Wege. Portal Server Technology. *IEEE Internet Computing*, 6(3):73–77, 2002.

[28] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A framework for rapid integration of presentation components. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 923–932, New York, NY, USA, 2007. ACM.