Integrating Component-Based Web Engineering into Content Management Systems

Stefania Leone[1,⋆], Alexandre de Spindler[2], Moira C. Norrie[2], and Dennis McLeod[1]

[1] Semantic Information Research Laboratory, Computer Science Department, USC
Los Angeles, CA, 90089-0781, USA
{stefania.leone,mcleod}@usc.edu
[2] Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
{despindler,norrie}@inf.ethz.ch

Abstract. Popular content management systems such as WordPress and Drupal offer a plug-in mechanism that allows users to extend the platform with additional functionality. However, plug-ins are typically isolated extensions defining their own data structures, application logic and user interface, and are difficult to combine. We address the fact that users may want to configure their applications more freely through the composition of such extensions. We present an approach and model for component-based web engineering based on the concept of components and connectors between them, supporting composition at the level of the schema and data, the application logic and the user interface. We show how our approach can be used to integrate component-based web engineering into platforms such as WordPress. We demonstrate the benefits of the approach by presenting a composition plug-in that showcases component composition through configurable connectors based on an eCommerce application scenario.

Keywords: Component-based Web Engineering, Content Management System, WordPress.

1 Introduction

Popular content management systems (CMS) such as WordPress[1] and Drupal[2] greatly facilitate the task of designing and developing web applications for small companies and individuals. These systems offer a graphical administrator interface, where users can author content, upload media, customise the layout and integrate a wide variety of plug-ins to extend the platform core with additional functionality. The WordPress Plug-in Directory[3] hosts thousands of plug-ins

⋆ Stefania Leone's work is supported by the Swiss National Science Foundation (SNF) grant PBEZP2_140049. The research has also been funded in part by the Integrated Media Systems Center (IMSC) of the University of Southern California (USC).

[1] http://wordpress.org

[2] http://drupal.org/

[3] http://wordpress.org/extend/plugins/

F. Daniel, P. Dolog, and Q. Li (Eds.): ICWE 2013, LNCS 7977, pp. 37–51, 2013.
© Springer-Verlag Berlin Heidelberg 2013

developed by the community, providing functionality ranging from site access statistic, over sophisticated photo galleries to eCommerce solutions. Plug-ins may define their own data structure, application logic and user interface. Although extremely powerful, plug-ins are simply extensions of the platform core. They are typically isolated and it is difficult to compose them with other plug-ins. For example, a company that runs their online shop based on a WordPress eCommerce plug-in, such as WooCommerce[4], might want to perform a customer satisfaction survey by using a survey plug-in, e.g. WordPress Simple Survey[5]. Ideally, for the participant profile data, the survey plug-in could directly make use of the customer data managed as part of the eCommerce plug-in. However, the current WordPress application model would require the user to familiarise themself with the code of the eCommerce plug-in and to programatically extract and map the customer data from the eCommerce plug-in to the participant format defined by the survey plug-in. This is a task that generally goes beyond the skills of a typical, non-technical WordPress user.

In this paper, we present an approach and a well-defined component model that supports end-users, both non-technical as well as more advanced ones, in performing such composition scenarios. The presented work is in line with recent research on end-user development, where they not only consider how to make web information systems easy to use, but also easy to develop [1]. Our approach enhances and generalises the application model of CMS such as WordPress to support component-based web engineering. Our model is based on the concept of components and explicit connectors between them. A component adheres to a well-defined component structure exposing interfaces for component composition at various levels. To build an application, components are composed through configurable connectors between them. We introduce different connector types to support composition at various levels, i.e. composition at the schema and data level, at the level of the application logic, and at the level of the user interface. We have realised our approach based on WordPress and present a composition plug-in that supports component composition based on configurable connector types. Finally, as proof of concept, we show how an eCommerce solution could be extended and combined with other plug-ins using our composition plug-in.

This paper is structured as follows. We give an overview of the background in Sect. 2, followed by our approach in Sect. 3. We introduce the component and composition model in Sect. 4. Section 5 presents the application of our approach using WordPress, followed by the presentation of the composition plug-in in Sect. 6 and the validation of our approach in Section 7. Concluding remarks are given in Sect. 8.

2 Background

Over the years, numerous frameworks and approaches for designing and developing web information systems have been introduced. Model-driven web

[4] http://wordpress.org/extend/plugins/woocommerce/

[5] http://wordpress.org/extend/plugins/wordpress-simple-survey/

engineering approaches, e.g. [2, 3] offer systematic methodologies based on models describing the structural, navigational and presentation aspects of a Web information system. Models are typically defined graphically and most methodologies offer a platform for application generation and deployment according to the defined models. These solutions, however, were targeted at collaborating groups of database architects, web developers and graphic designers, and explicitly supported the separation of concerns in terms of their roles by providing separate models for the different levels of a web information system.

In parallel, CMS became a popular way for non-technical users, including individuals and small companies, to create websites and publish their content. Platforms such as WordPress and Drupal provide graphical administrator interfaces, which support the website design of content and structure in terms of general publishing units and presentation styles. The extensibility mechanism inherent to these platforms allows for the integration of arbitrary data and services to support the creation of complex web information systems. The configuration and use of plug-ins is typically also performed through the administrator interface. However, as already stated, these extensions, while extremely powerful, cannot easily be combined or mashed-up. Plug-ins are typically isolated units developed by community members, and there is little control or conventions with respect to the plug-in internals. In the case of WordPress, developing and composing plug-ins requires knowledge of PHP as well as a detailed understanding of the WordPress platform and its inner workings.

A number of approaches support web application development from reusable components. With WebComposition [4], web applications are built through hierarchical compositions of reusable application components. Similarly, web mashups are composed through the orchestration of reusable, self-contained services, which interact at the message level and may span multiple applications and organisations. Various mashup editors offer graphical tools as an alternative to programmatic interfaces to do the composition process, both for general, e.g. [5–7] and domain-specific mashup creation, e.g. [8, 9]. While some mashup editors help users to integrate information from distributed sources, others provide infrastructure for building new applications from reusable components. For example MashArt [7] enables advanced users to create their own applications through the composition of user interface, application and data components. The focus is on supporting the integration of existing components based on event-based composition, where components can react to events of other components.

We build on and extend these ideas for the CMS domain targeting non-technical users. In contrast to previous work, our approach offers component-based web engineering based on the definition and configuration of explicit connectors that encapsulate the collaboration logic between components. As stated in [10], one of the main challenges of modular system development lies in the fact that modular units may not be compatible for composition. As a consequence, our component model is inspired by the Architecture Description Language (ADL) [11, 12], an approach to component-based software engineering, where the component model consists of components and explicit connectors

between them. Through the definition of explicit connectors between components, we circumvent the problem of component incompatibility. Connectors encapsulate the composition logic, exhibiting functionality ranging from simple message passing, to complex collaboration logic, such as data transformation operations, and, consequently, would allow for the composition of arbitrary components. We introduce different types of connectors, which can be configured to define the composition for a particular composition scenario. For example, a schema connector type could be configured to support the structural composition of the eCommerce and survey plug-ins.

Our approach and model is not dissimilar to the application model introduced by the Google Android platform[6] for developing and running mobile applications. Their application model propagates the reuse of different types of application components across applications, where applications are configured through so-called intents that define the glue code between the various components. While intents allow base values to be passed in the form of key-value pairs between components, our connectors generalise this approach and may define arbitrary complex collaboration logic between components.

3 Approach

We introduce a component-based approach to web engineering based on ideas of ADL where applications are modelled based on reusable components and explicit connectors between them. Components may provide arbitrary functionality and define their own data structure, application logic and user interface.

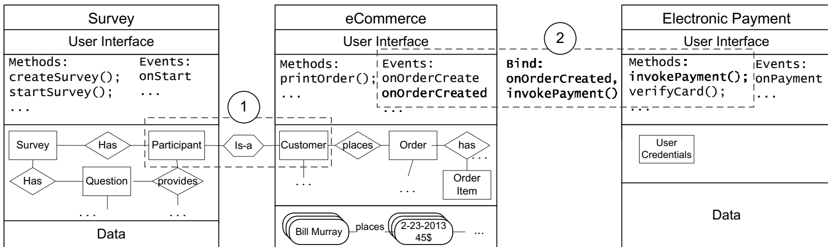

Fig. 1. Composition Scenario

We will introduce our approach based on the example of a company that makes use of a CMS extended with an eCommerce component for their online business. Figure 1 gives an overview of the scenario. The eCommerce component, in the centre, allows users to create and manage an online store, including product, customer and order management. The component defines a schema that represents the eCommerce application domain by means of entity types and relationships,

[6] `http://developer.android.com/guide/`

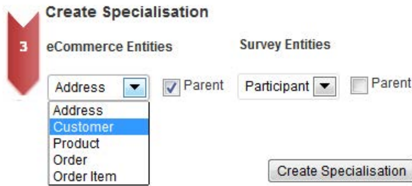

Fig. 2. Specialisation Screenshot

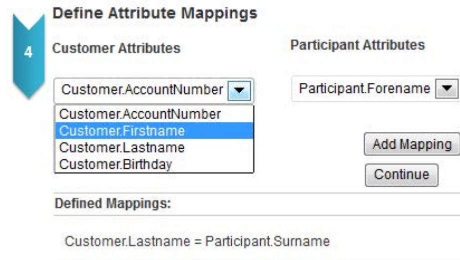

Fig. 3. Attribute Mapping Screenshot

and manages data structured accordingly. Furthermore, the component defines application logic by means of methods and events, which implement the online store functionality, and this functionality is made available to the user through a graphical user interface.

To evaluate customer satisfaction, the company decides to perform a customer satisfaction survey and they would like to make use of their customer data when performing the survey. For this purpose, they have selected a survey component, shown on the right, that offers the required functionality to define and run surveys. The survey component, in turn, consists of a user interface, application logic and a schema, and the component may manage data structured accordingly. However, they want to avoid having two separate user entities and therefore want to create a connection between the eCommerce customer and the survey participants.

Connector (1), on the left in Fig. 1, defines the composition between the two components. It is a specialisation connector that defines an `is-a` relationship between the `Customer` entity of the eCommerce component and the `Participant` entity of the survey component. Through this specialisation connector, the customer data can automatically be used as participant data for the survey.

Figures 2 and 3 illustrate, based on screenshots, how a user configures a specialisation connector through a graphical composition wizard. We assume that the user has already selected the components to be composed as well as the connector type. Figure 2 shows how the user creates the actual `is-a` relationship by selecting the `customer` entity of the eCommerce component and the `participant` entity of the survey component. The user also defines that the `customer` entity should become the parent entity by checking the *parent* checkbox. Next, the user has the possibility to define attribute mappings between the matching/overlapping attributes of the two entity types. In our example, both entities participant and customer define `name` attributes. Figure 3 shows how such mappings are created. Here, the user is about to create a mapping between the `User.firstname` and the `Participant.forename` attributes. At the bottom of the figure, the list of defined mappings is shown, where the attribute `User.lastname` was mapped to `Participant.surname`. With these mappings, the specialisation connector ensures that each time the name of a survey

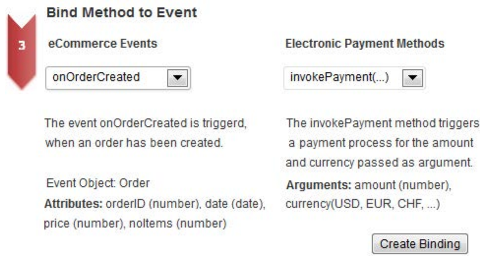

Fig. 4. Binding Creation Screenshot

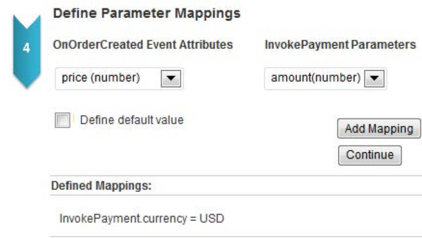

Fig. 5. Parameter Mapping Screenshot

participant is accessed, the corresponding customer name from the eCommerce component is retrieved and displayed. Note that, in this example, the specialisation does not require any data mappings, since the survey component does not yet manage data. However, when composing two components with data, the specialisation definition also requires the definition of a data mapping and a conflict resolution strategy, also supported through our composition wizard.

While this is the basic functionality provided by the specialisation connector, advanced users are free to extend the configured connector programatically. For example, the connector could be extended to perform data mining by defining queries that combine survey data with customer data to answer questions such as "Do customers who selected answer (a) in question 4 buy similar products?".

In a second step, the company decides to offer support for electronic payment, a functionality that is not provided by the eCommerce component. For this composition, the eCommerce component is composed with an electronic payment component, shown on the right of Fig. 1. The event handler registration connector (2) operates at the application logic level, based on events and callback methods. Figure 4 and 5 show the steps involved in configuring this connector. Again, we assume that the user has already selected the components to be composed and the connector type. Furthermore, the user has decided that the electronic payment component should be invoked as a result of an event that occurs in the eCommerce component. The screenshot in Fig. 4 shows how a user defines that binding by selecting events and methods. In the current example, the user has selected the `onOrderCreated` event from the survey component. According to the description shown below the drop-down menu, the event gives access to the created order and its attributes. On the left, the user has selected the `invokePayment` method of the electronic payment component and the description of the method and its parameters is displayed, saying that the method takes two parameters `amount` and `currency`.

After creating the basic binding, the user may define mappings between the event object attributes and the method parameters, as shown in Fig. 5. In the current example, the user intends to map the `price` attribute of the order to the `amount` parameter of the `invokePayment` method. Also, at the bottom, a list of created mappings is illustrated. The user has already created a static mapping for the currency parameter by assigning it the default value "USD". Note that

users are free to define such default values for parameters in cases where the attributes and parameters do not match or are incompatible and we support basic type transformation.

As these two composition examples illustrate, connectors provide the glue between components and are configured by the user to adhere to a particular composition scenario. We offer different types of connectors that support composition at various levels of a component. Figure 6 gives an overview of the composition levels and shows, from left to right, that connectors may be used at the data level, the schema level, the application logic level and the user interface level. We provide connector types for all these levels and will present our component model including the various connector types in the next section.

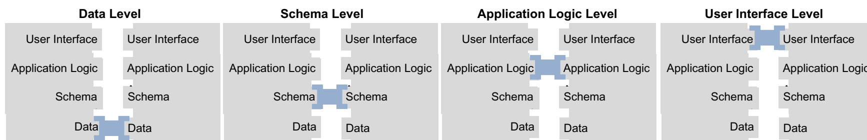

Fig. 6. Composition Levels

4 Component Model

A component is an application providing arbitrary functionality to its users. Components may be composed with other components using explicit connectors between them to form more complex applications. The general component model along with the composition interface is shown on the left in Fig. 7, while the eCommerce component introduced in Sect. 3 is shown on the right as an example following this model.

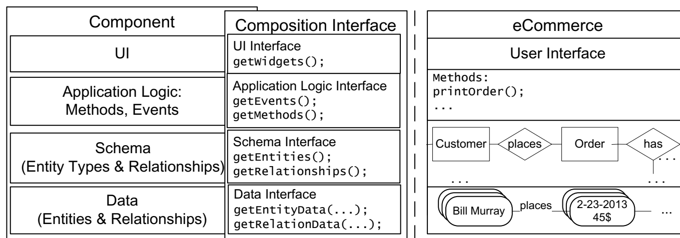

Fig. 7. Component Example

Formally, a component is defined as a tuple of the four elements

$$Component = \langle Schema, Data, Application\ Logic, User\ Interface \rangle$$

The *Schema* is a data model instance describing the component *Data* in terms of a set of entity types $\{E_1, \ldots, E_M\}$ and relationships $\{R_1, \ldots, R_N\}$. The *Application Logic* includes a set of methods $\{m_1(), \ldots, m_U()\}$ implementing the application logic and events $\{e_1, \ldots, e_V\}$ related to these methods. Components typically contain basic CRUD methods supporting the management of their entity types and relationships, as well as higher-level methods providing domain-specific functionality. Component developers are free to define an arbitrary number of events triggered by the execution of such methods. For example, a component may define events marking the start and end of CRUD method executions.

Finally, the *User Interface* defines the graphical user interface. In CMS, the user interface is typically specified by layout themes defining the general presentation of the provided publishing units for the complete web site. As part of the user interface, components may define a set of widgets $\{W_1, \ldots, W_N\}$ displaying specific component data or providing component services to the users. Widgets represent complete user interfaces including user interface controls, layout and style templates. Note that components do not necessarily specify multiple or all of these four elements. For example, while the eCommerce component specifies *Schema*, *Application Logic* and *Widgets* elements, other components may for example only specify *Application Logic* and *Schema* elements.

Components expose a composition interface which defines in which way they may be composed with other components. In order to implement such an interface, component developers need to specify which of the component elements they wish to make available for composition. Component interfaces do not need to expose component elements at all levels. At the schema-level, the interface specifies the subset $\{E_i, \ldots\} \subseteq \{E_1, \ldots, E_M\}$ of composable entity types and the subset $\{R_j, \ldots\} \subseteq \{R_1, \ldots, R_N\}$ of composable relationships. The specification of the data available for composition consists of a query Q over the composable schema elements. Similar to the schema interface definition, application logic made available for composition is defined in terms of subsets $\{m_k(), \ldots\} \subseteq \{m_1(), \ldots, m_U()\}$ and $\{e_l(), \ldots\} \subseteq \{e_1, \ldots, e_V\}$. Finally, user interface widgets are exposed in terms of the subset $\{W_i, \ldots\} \subseteq \{W_1, \ldots, W_N\}$.

In Figure 7, a programmatic representation of the composition interface is shown, with getter methods to access the defined subsets of composable widgets, methods, events, schema elements and data.

Connectors specify how components are connected and at which level. For example, the specialisation connector presented in Sect. 3 defines an `is-a` relationship at the schema level, and the event handler registration component binds a callback function to an event at the application logic level. Figure 8 shows the basic types of connectors—categorised based on the composition level. The widget connector, shown at the top, supports composition at the UI level through the integration of widgets between components. The connector forms the union of widgets defined as *User Interface* $:= \{W_1, \ldots\} \subseteq UserInterface_A \cup UserInterface_B$. In the example in Fig. 8, the connector integrates a widget of component A into the user interface of component B.

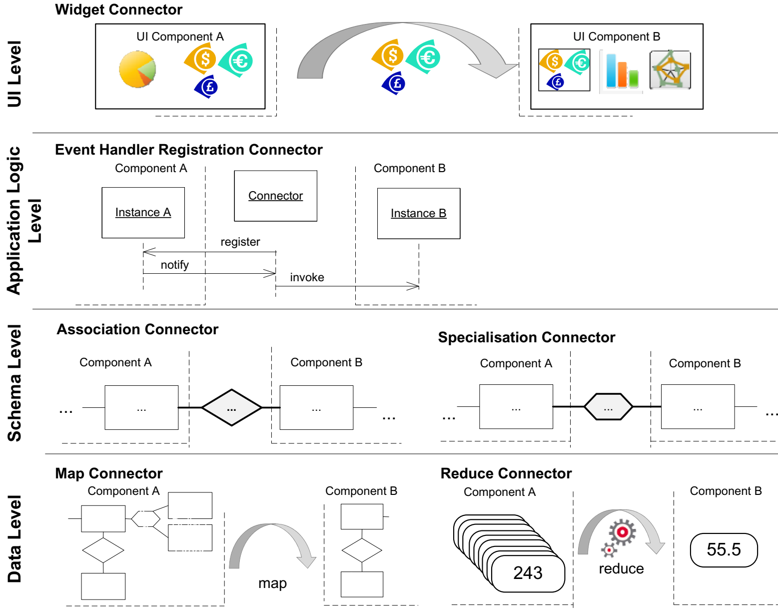

Fig. 8. Basic Connector Types and Composition Scenarios

At the application logic level, Fig. 8 illustrates the Event Registration Connector based on a UML sequence diagram that reflects the collaboration between components and connectors in an event-based setting. The connector is specified as $Application\ Logic := \{m_1()\{e_i \rightarrow m_j()\}, \ldots\}$ defining functions binding events from one component to methods of another component.

Schema-level connectors compose components based on schema elements, such as specialisation and associations [13]. As shown in Fig. 8, a specialisation connector defines an `is-a` relationship establishing a specialisation relationship among entity types from different component schemata and an association connector defines a relationship between two entity types from different components. More generally, a schema connector may define arbitrary schema elements among component entities $Schema := \{\{E_1, \ldots\}, \{R_1, \ldots\}\}$.

Finally, data connectors allow data from one component to be reused by another component. As shown in Fig. 8, data reuse may be defined by a mapping connector that maps the schema of one component to the schema of another component, or by a reduce connector that transforms data from one component to a format specified by another component. Generally, data connectors may be defined as combinations of map and reduce functions of the form $Application\ Logic := \{map()\{E_i.a_j \leftarrow reduce(E_k.a_n, E_l.a_m)\}, \ldots\}$. Such map and reduce functions may in turn be bound to data mapping connector events to define whether the mappings should occur once, multiple times or periodically.

Note that we have given a minimal specification of the various connector types, but they may define richer functionality. For example, the association connector may also define application logic in the form of CRUD methods to create

associations, as well as a widget that allows new associations to be graphically created and viewed. Similarly, a reduce connector may define a user interface, where the reduce function could be configured.

As seen with these examples, connectors consist of the same building block as general components and, therefore, can be seen as a special type of components, where the functionality is not targeted at the application domain, but rather at the composition of domain-specific application units. Figure 9 shows the metamodel of our component model. A component defines a user interface, application logic, schema and data, and, depending on the implementation technology, these elements may be realised in different ways. A connector is a sub-type of component, and therefore, they can in turn be composed. Connectors are classified according to their supported composition level, which defines the access points of a connector. A concrete connector is an instance of such a connector type and is instantiated with values that are particular to a composition scenario. For example, a specialisation connector will be instantiated with an `is-a` relationship between two entity types.

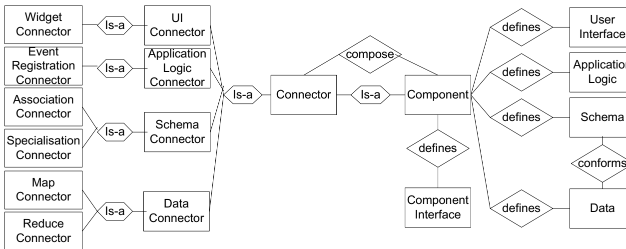

Fig. 9. Component Metamodel

5 WordPress Extension

We have extended WordPress with our approach and component model to support component-based web engineering. We will first give a short introduction to the WordPress plug-in mechanism before presenting our extension.

The WordPress plug-in mechanism allows the original blogging model to be extended in terms of data structure, application logic and user interface widgets by hooking into the WordPress core. A number of such hooks are provided, which allow plug-ins to inject additional functionality, data structures and presentation into the WordPress core execution environment. Hooks may represent plug-in lifecycle events such as their installation or uninstallation, as well as administrative or end-user activities including the creation, manipulation, retrieval, selection, display and deletion of posts, pages or plug-in-specific data. Typically, the plug-in code includes functions for creating and deleting database tables, for inserting and selecting table data and the assignment of these functions to particular hooks. Users are free to define their own hooks, which allows

plug-ins to react to events of other plug-ins. For the user interface, plug-ins rely on the WordPress publishing process and themes that define the structure and layout of the complete web site. A plug-in may, however, define widgets that can be placed in various places of the user interface. To install a plug-in, the files containing the plug-in code, typically one or more PHP files and JavaScript, are uploaded into the target WordPress platform through the WordPress administrator dashboard and can be activated and deactivated. Upon activation, the additional functionality, data structures and presentation facilities become part of WordPress and are available for immediate use.

We have extended the WordPress plug-in model to adhere to our approach. On the level of the application logic and user interface, the WordPress plug-in model matches our component model. At the level of the user interface, plug-ins may define widgets and the WordPress core handles the generation of the user interface from themes including the placement of such widgets. Application logic is represented by PHP functions and events. At the data and schema level, however, WordPress only supports a basic notion of types and data may be stored in any possible way and format. Also, plug-ins do not define a composition API, as defined by our approach. We therefore build on our previous work [14] where we introduced DataPress, a WordPress plug-in which supports the generation of tailored WordPress plug-ins from user-defined ER models. With DataPress, a user graphically defines an application domain by means of ER models through the dashboard and DataPress automatically generates a plug-in that allows data to be managed accordingly. For each defined entity type and relationship, DataPress generates data structures, CRUD methods and user interface components to create and manage the data.

By building on this approach, we not only gain support for ER modelling, but we could also extend the automatic generation of plug-ins to conform to our component model. We additionally generate two hooks for each of the generated CRUD operations—a before and after hook. For example, for the creation of an order entity, the two hooks `onOrderCreate` and `onOrderCreated` are generated. Also, we generate a configuration file that represents the composition API that gives access to the composable plug-in elements. The file lists the names of the entities, relationships, methods, events and widgets defined by a plug-in and the user can simply remove elements that should not be offered for composition. The configuration of a connector for a particular composition scenario is based on these names defined in the respective plug-in configuration files.

6 Composition Plug-in

To support composition by non-technical users, we provide a composition plug-in that supports the composition process graphically, as illustrated in the screenshots in Sect. 3. Figure 10 gives an overview of the composition plug-in architecture. The composition plug-in, shown in the centre, is a regular WordPress plug-in that is integrated into the dashboard. It provides access to locally installed plug-ins, shown on the left, and the connector type plug-ins, shown on the

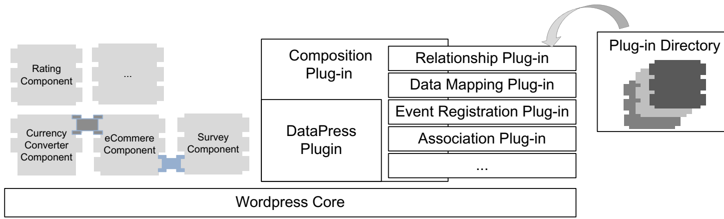

Fig. 10. Composition Plug-in Architecture

right. It builds on an extended version of DataPress and supports the generation of plug-ins from user-defined ER models structured according to our component model. Using the composition plug-in, new plug-ins can be composed with the installed ones by configuring one of the provided connector types. Assuming that all plug-ins would be structured according to our approach, a user could also download, install and compose plug-ins from the Wordpress Plug-in directory, shown on the left.

Each connector type has been realised as parameterised plug-in, which gets "instantiated" upon composition. The composition plug-in automatically generates and installs the configured connector plug-ins. Below, we show a configured version of the event handler registration connector that corresponds to the configuration shown in Figs. 4 and 5. Through the configuration process, the connector has been named *Payment Connector* and the event and method names to be bound, have been injected into the plug-in template. In WordPress, the `add_action` method registers a specific hook with a specific method. The `add_action` method further defines the priority of the method invocation, as well as the numbers of arguments that are passed from the event to the method. While WordPress assumes that the number and types of attributes provided by the hook match the parameters of the callback method, we have generalised this approach by giving the user the possibility to define attribute-parameter mappings, as shown in Fig. 5. In this example, the event `onOrderCreated` defines four attributes while the method `invokePayment` only takes two parameters.

```
/* Plug-in Name: PaymentConnector*/
...
add_action('onOrderCreated', 'invokePaymentTemp', 1, 4);
add_action('onPayedElectronically', 'redirectToShop');

function invokePaymentTemp($orderID, $date, $price, $noItems){
    $currency=''USD'';
    invokePayment($price, $currency);
}
function redirectToShop(){...}
```

The mappings are reflected in the connector code. Upon `onOrderCreated`, a helper method `invokePaymentTemp` method is invoked, accepting the four attributes defined by the `onOrderCreated` event as parameters. The method realises the defined attribute mappings and invokes the actual payment method

using these mappings. Here, the attribute `$price` from `onOrderCreated`, and the attribute `currency` set to the default value “USD” are used as parameters.

Note that more advanced users are free to extend a configured connector plug-in with additional code. In the current example, the user has also defined a second binding, which, upon completion of the electronic payment, invokes the locally defined method `redirectToShop` to automatically redirect the customer back to the eCommerce component.

While the configured event registration connector plug-in only defines application logic between two components, other connectors also define schema, data and widgets. For example, the association connector creates a database table as part of its installation process where associated pairs of entities are stored. Furthermore, it also defines a widget allowing users to graphically create associations and, as part of the connector configuration, the user decides whether the widget may be visible along with one or both composed components as part of a dynamic sidebar injected into the layout theme, or as part of the dashboard. The widget connector is realised in a similar way: A configured widget connector injects a widget from one component into the user interface of another component, based on a user’s configuration, by placing them in a dynamic sidebar.

7 Scenario

We have used the composition plug-in to compose an extended eCommerce application from various components. Figure 11 illustrates the various composition scenarios. As a first step (1), the eCommerce component has been composed with the survey component through a specialisation connector as described in Sect. 3. On the left, the specialisation is defined by means of an `is-a` relationship and the two attribute mappings.

In a second step, the eCommerce component has been composed with a review component to allow products to be reviewed by customers. The composition is based on an association connector, shown in (2). The connector defines the association between the product and the review entity including the cardinality constraints. A product may have *0* or *n* reviews and a review is for exactly one product. The connector also defines a widget, illustrated by the editor icon on the left. The connector is configured in such a way that the widget is displayed alongside the product view, allowing customers to write and view reviews while browsing products. Next, the eCommerce component is extended with electronic payment support, by composing it with an electronic payment component at the level of the application logic, shown in (3). The connector configuration defines the binding of the `onOrderCreate` event with the `invokePayment` method.

As the customer base of the company becomes more international, the company decides to make use of a currency converter component. In (4) the composition of the eCommerce component with a currency converter component is shown. The composition takes place at the user interface level: the connector defines that the currency converter widget is to be displayed along with products and orders, so that customers can make use of it when browsing products or to convert the price of an order.

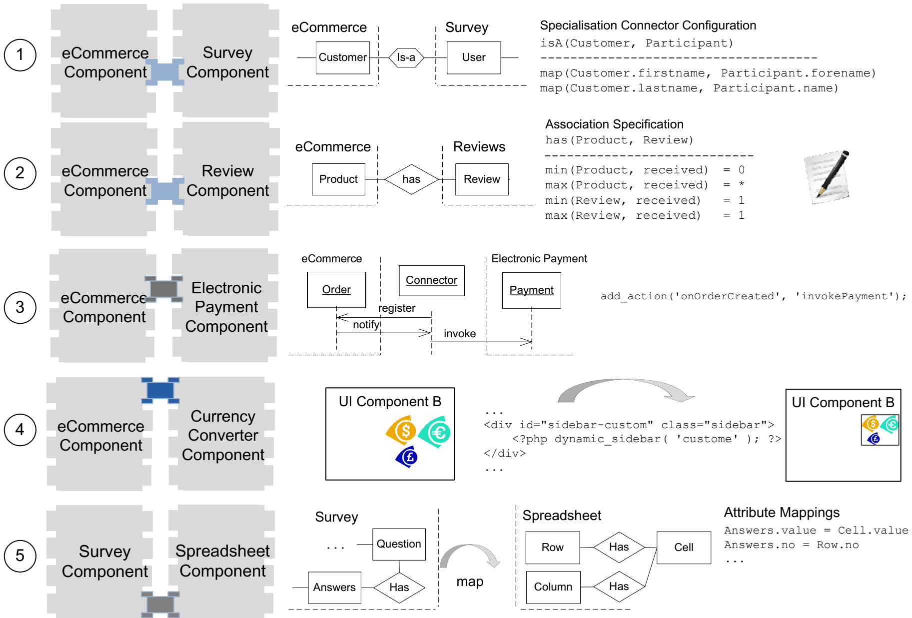

Fig. 11. Scenario Application

Finally, the company would like to evaluate the outcome of the survey using a spreadsheet component. To do so, the data of the survey component is mapped to the data format of the spreadsheet component, as shown in (5). The mapping is specified by a number of attribute mappings between the entities of the survey component and the entities of the spreadsheet component, where the survey questions and answers are mapped to the spreadsheet format.

8 Conclusion

Our approach, model and implementation is a practical solution to enhance today's content-management systems with support for component-based web engineering. By defining components and explicit connectors between them, we not only circumvent possible incompatibilities between components, but we also make sure that composed systems are resilient to plug-in updates, since the composition logic is completely encapsulated within the connector code.

We see our work as a further step towards providing systems that are easy to develop. While we target non-technical end-users and support the composition process through graphical user interfaces, this approach is clearly more limited than the programmatic definition and extensions of connectors, as can be done by more experienced users. However, the presented scenario has shown that a small set of relatively simple connector types covers a wide range of composition scenarios allowing for the design of relatively complex applications through graphical user interfaces. As a next step, we plan to conduct a user study to

further validate and refine our approach. We also note that our approach is extensible, and new connector types could be added at any level.

References

1. Lieberman, H., Paterno, F. (eds.): End User Development. Human-Computer Interaction Series. Springer (2006)
2. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): A Modeling Language For Designing Web Sites. Computer Networks 33(1-6) (2000)
3. Vdovják, R., Frăsincar, F., Houben, G.J., Barna, P.: Engineering Semantic Web Information Systems in Hera. Journal of Web Engineering 1(1-2) (2003)
4. Gellersen, H.W., Wicke, R., Gaedke, M.: Webcomposition: An object-oriented support system for the web engineering lifecycle. Computer Networks 29(8-13), 1429–1437 (1997)
5. Ennals, R., Brewer, E., Garofalakis, M., Shadle, M., Gandhi, P.: Intel Mash Maker: Join the Web. ACM SIGMOD Record 36(4), 27–33 (2007)
6. Murthy, S., Maier, D., Delcambre, L.: Mash-o-Matic. In: Proc. ACM Symposium on Document Engineering (DocEng 2006) (2006)
7. Daniel, F., Casati, F., Benatallah, B., Shan, M.-C.: Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In: Laender, A.H.F., Castano, S., Dayal, U., Casati, F., de Oliveira, J.P.M. (eds.) ER 2009. LNCS, vol. 5829, pp. 428–443. Springer, Heidelberg (2009)
8. Imran, M., Soi, S., Kling, F., Daniel, F., Casati, F., Marchese, M.: On the Systematic Development of Domain-Specific Mashup Tools for End Users. In: Brambilla, M., Tokuda, T., Tolksdorf, R. (eds.) ICWE 2012. LNCS, vol. 7387, pp. 291–298. Springer, Heidelberg (2012)
9. Chudnovskyy, O., Nestler, T., Gaedke, M., Daniel, F., Fernández-Villamor, J.I., Chepegin, V.I., Fornas, J.A., Wilson, S., Kögler, C., Chang, H.: End-user-oriented Telco Mashups: The OMELETTE Approach. In Proc. World Wide Web Conf. (WWW 2012) (Companion Volume) (2012)
10. Shaw, M.: Modularity for the Modern World: Summary of Invited Keynote. In: Proc. Intl. Conf. on Aspect-Oriented Software Development (AOSD 2011) (2011)
11. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Softw. Eng. 26(1), 70–93 (2000)
12. Clements, P.C.: A Survey of Architecture Description Languages. In: Proc. Intl. Workshop on Software Specification and Design (IWSSD 1996) (1996)
13. Leone, S., Norrie, M.C.: Building eCommerce Systems from Shared Micro-Schemas. In: Meersman, R., Dillon, T., Herrero, P., Kumar, A., Reichert, M., Qing, L., Ooi, B.-C., Damiani, E., Schmidt, D.C., White, J., Hauswirth, M., Hitzler, P., Mohania, M. (eds.) OTM 2011, Part I. LNCS, vol. 7044, pp. 284–301. Springer, Heidelberg (2011)
14. Leone, S., de Spindler, A., Norrie, M.C.: A Meta-Plugin for Bespoke Data Management in WordPress. In: Wang, X.S., Cruz, I., Delis, A., Huang, G. (eds.) WISE 2012. LNCS, vol. 7651, pp. 580–593. Springer, Heidelberg (2012)