Iterative Model-Driven Development of Software Extensions for Web Content Management Systems

Dennis Priefer[1,2], Peter Kneisel[2], and Daniel Strüber[3]

[1] Philipps-Universität Marburg, Germany
[2] Institute for Information Science, Technische Hochschule Mittelhessen, Germany
`{dennis.priefer,peter.kneisel}@mni.thm.de`
[3] Institute for Computer Science, University of Koblenz and Landau, Germany
`strueber@uni-koblenz.de`

Abstract. Dynamic web applications powered by Web Content Management Systems (WCMSs) such as Joomla, WordPress, or Drupal dominate today's web. A main advantage of WCMSs is their functional extensibility by standardized WCMS extensions. However, the development and evolution of these extensions are challenging tasks. Due to dependencies to the core platform and other WCMS extensions, the code structure of an extension includes a large defect potential. Mistakes usually lead to website crashes and are hard to find, especially for inexperienced developers.

In this work, we define a model-driven development (MDD) process and apply it during the development of software extensions for the WCMS Joomla. To address two separate scenarios, involving the development of independent and dependent WCMS extensions, we use an MDD infrastructure, comprising a domain-specific language, a code editor, and reverse engineering facilities. In addition, we provide evidence indicating that our model-driven approach is useful to generate extensions with consistent interdependencies, demonstrating that the main issues of extension development in the WCMS domain can be addressed using a model-driven approach. By applying the MDD infrastructure on actual projects, we additionally present the lessons learned.

Keywords: Model-Driven Development; Web Content Management Systems; Joomla

1 Introduction

In today's web engineering practice, the creation of functionally rich web applications from scratch is an outdated process. Instead, web developers use a variety of *Web Content Management Systems* (WCMSs) [16] providing the main functionality of typical web applications, such as management of users, content, menus, media and templates, as well as multi-language support.

If the functional needs of WCMS administrators exceed the core functionality of a WCMS, it needs to be functionally augmented. Examples for additional functionality include web shops, file repositories, image galleries, or the management of domain-specific data, such as conference information. When using an open source WCMS, developers can change the code basis in order to add additional features to the WCMS. A less intrusive mechanism is based on *software extensions* that can be deployed to a running WCMS instance by an administrator without changing the platform (see Fig. 1). This approach can ensure a consistent system even if the WCMS platform undergoes a version update.

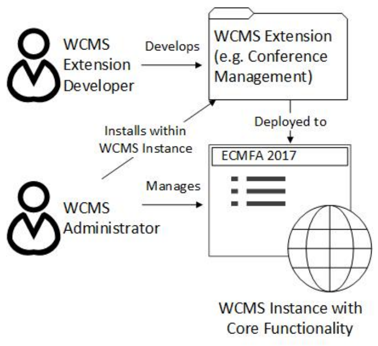

WCMS Extension Developer
Develops
WCMS Extension (e.g. Conference Management)
Installs within WCMS Instance
Deployed to
ECMFA 2017
Manages
WCMS Administrator
WCMS Instance with Core Functionality

Fig. 1: Functional Extension of a WCMS Instance

The most popular WCMSs are *WordPress* [5], *Joomla* [2], and *Drupal* [1]. WordPress holds the largest market share by far (58,8% of all CMSs and 27,6% of all web pages [4]) followed by Joomla (7,1% of all CMSs and 3,3% of all web pages), and Drupal (4,7% of all CMSs and 2,2% of all web pages). For other well-known CMSs like *Magento* [3] and *Typo3* [6], the market share is significantly lower compared to these top three WCMSs. All of these systems provide extensibility in the form of installable software extensions. Norrie et al. [17] explain the success of WordPress as a result of its user-friendliness and extensibility. In particular, end-users without advanced technical skills benefit from its capability to create and publish a web site in a few minutes. Extensibility in the form of a plug-in mechanism empowers users with a technical background to customize a WordPress instance according to their needs. Yet this plug-in mechanism is relatively simple, based on the exposure of an interface by the platform core used to augment the core with additional code. Support for more complex extensions, such as domain-specific data management and presentation or event-triggered extensions, is lacking. If such extensions are developed using hand-written code, significant challenges to maintainability arise.

A more sophisticated extension mechanism is offered by Joomla. In contrast to most other WCMSs such as WordPress or Drupal, Joomla supports a variety of extension types to facilitate the development of feature-rich extensions. For instance, **components** are an extension type that provides full data management capabilities, whereas **modules** offer presentation utilities for the data managed by some component. This allows the development of new extensions using data of existing ones, e.g. a module presenting data of a 3rd party component. The extension mechanism provided by Joomla is based on an API as well as naming conventions: For a consistent deployment to the core platform, an extension must conform to an elaborate standard file and code structure.

Even though the extension mechanism of Joomla is powerful, extension developers face several issues during development and evolution. Developing a new extension is a challenging task even for experienced developers. A typical procedure is to create a clone of an existing extension complying with the standard structure and to modify the clone to satisfy the new requirements. However, this procedure shows a high susceptibility to errors. For instance, mismatches between class identifiers and file names might go unnoticed. Another problem occurs when the underlying platform evolves and existing extensions have to be updated to adapt to the platform changes. If the amount of extensions to migrate grows, the required effort for updating the extensions can increase tremendously.

In this work, we propose to apply a model-driven approach to the development of WCMS extensions. Our approach is based on the observation that a good amount of the file and code structure of popular WCMSs is made up of generic and schematically recurring fragments. The use of a domain-specific language and code generator is a promising means to reduce the development effort for WCMS extensions. In particular, we consider the two typical scenarios during WCMS extension development: The development of completely independent extensions as well as of extensions depending on existing extensions. We show how a model-driven approach is suitable to support developers during these tasks.

We exemplify our approach by illustrating its application to Joomla, a mainstream WCMS with a particularly sophisticated extension mechanism. In particular, we propose **JooMDD**, an infrastructure for the model-driven development of Joomla extensions. JooMDD comprises a DSL and model editor, a code generator, and a model extraction tool.

This work is the first to address the distinct challenge of developing interdependent WCMS extensions, an issue that does not occur in the simpler case of regular web application development. However, the model-driven development of WCMSs has been addressed in earlier works. In particular, the approaches presented in [21], [24], and [19] address the model-driven development of concrete WCMS instances, but do not take their extensibility into account.

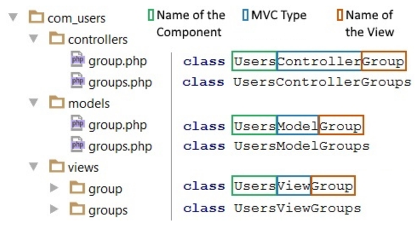
We introduce the technical background and common use-cases of developing Joomla extensions in Sect. 2 and present our MDD tools, including support for reverse engineering, in Sect. 3. Sect. 4 describes our process to address the typical development scenarios faced by Joomla extensions developers. We give example applications of our approach in Sect. 5 and share the lessons learned in Sect. 6.

2 Extension of Joomla Instances

This section describes the technical background of Joomla extensions and introduces typical use-cases for their development. The use-cases concern the differences in the extensions interdependencies between each other. So, we can clarify the different variants during our process definition in Sect. 4.

The Joomla platform provides a custom API for the functional expansion of its core functionality through installable extensions. Extensions come in different types of varying complexity, spanning the full range from complex extension types with their own dedicated data management to simple function libraries.

The most complex extension type is called **component**. Components usually have their own data management. To this end, the Joomla core database is extended with additional tables. To work together with the Joomla core, components must exhibit a file and code structure that follows a specific scheme. Fig. 2 shows an example of this scheme.



```
class UsersControllerGroup
class UsersControllerGroups

class UsersModelGroup
class UsersModelGroups

class UsersViewGroup
class UsersViewGroups
```

Fig. 2: File and Code Structure of a Joomla Component (Extract)

On file and code level, the scheme implements the Model-View-Controller (MVC) pattern [13]: views use models for data access; controllers can perform data updates using the models and can also process view requests. While components typically use their own custom data, a common practice is to use data of other components within the view. The MVC classes must comply with the illustrated scheme. Otherwise, the Joomla instance containing the component may produce errors.

Another wide-spread extension type are **modules**, which can be used to place any content within pre-defined module positions on a page of a Joomla instance. Common module types are menus, search fields, breadcrumbs, or login sections. Modules often use the data of an available component. Typically, modules display a data entry from an underlying database of a component.

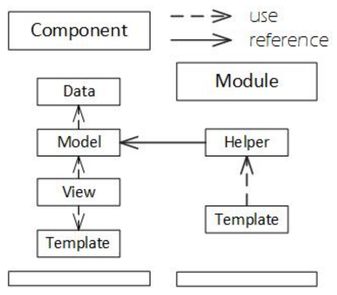
Developing Joomla extensions features two use-cases: First, developing independent extensions such as components which have their own data management and, second, dependent extensions, which use artefacts of existing extensions such as modules, that in turn use the database of a component.

2.1 Use Case 1: Creating Independent Extensions

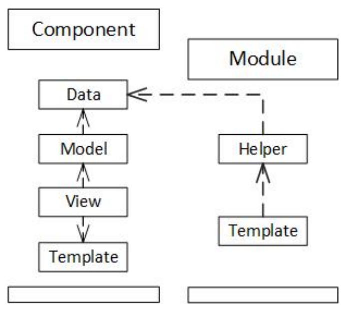
The first use-case is the development of independent extensions which can be used within a running Joomla instance. The advantage of independent extensions occurs during their evolution. If a developer changes the extension, no side-effects due to dependencies occur. However, it is important to comply with the development guidelines to ensure a correct interplay between the extension and the running core system where it is installed. Even subtle errors can lead to unexpected crashes that are not discovered until runtime.

2.2 Use Case 2: Creating Dependent Extensions

Components and modules stand out for their interplay with other extensions. It is common practice to use artefacts of existing extensions within a component or module to increase the functionality of a Joomla system without developing software fragments anew. Components may reuse models or view templates by other components, while modules use the database of existing components, since they usually provide no own data management. This allows developers to augment existing extensions (e.g. 3rd-party extensions) without changing their code base.



(a) Module referencing to an existing component model for data access



(b) Module uses the database of a component directly

```
class ConferenceHelper
{
    public static function &getList($params = null)
    {
        JModelLegacy::addIncludePath(
            JPATH_ROOT .
            '/components/com_conference/models',
            "Conference"
        );
        $model = JModelLegacy::getInstance(
            'Talks',
            "ConferenceModel",
            array('ignore_request' => true)
        );
        $items = $model->getItems();
        return $items;
    }
}
```

(c) Module referencing to an existing component model for data access (code)

```
class ConferenceHelper
{
    public static function &getList($params = null)
    {
        $db     = JFactory::getDbo();
        $query  = $db->getQuery(true);
        $query->select($db->quoteName(
            array('id', 'title', 'speaker')));
        $query->from($db->quoteName(
            '#__conference_talks'));
        $db->setQuery($query);
        $items = $db->loadResults();
        return $items;
    }
}
```

(d) Module uses the database of a component directly (code)

Fig. 3: Usual dependencies between modules and components

Figure 3 illustrates common dependencies between modules and components on the example of an existing conference component, which is augmented by a dependant module for the representation of conference talks. Figs. 3a and 3b illustrate the dependency variants in an abstract manner, whereas Figs. 3c and 3d show the minimal amount of required code to establish these dependencies. The code shows the corresponding variants of both dependencies as part of a helper file within a module. This file represents the model of a module. The first variant of dependency features the use of a component model within a module via **reference**, using inclusion methods of Joomla's singleton implementation (`JModelLegacy`), whereas the second variant illustrates the **direct use** of a component's database by using SQL statements.

In this work we focus on the model-driven development of components and models, with support for both use-cases. While we addressed the less complicated use-case 1 in our earlier work [18], our extended infrastructure and process address both use-cases.

3 JooMDD - MDD Infrastructure for Joomla Extensions Including Reverse Engineering Support

This section presents JooMDD, our infrastructure for the model-driven development of Joomla extensions[4].

JooMDD supports Joomla extension developers with a set of MDD tools: A DSL and editor for the creation of extension models, a code generator for Joomla extensions, and a tool to extract extension models from legacy extension code.

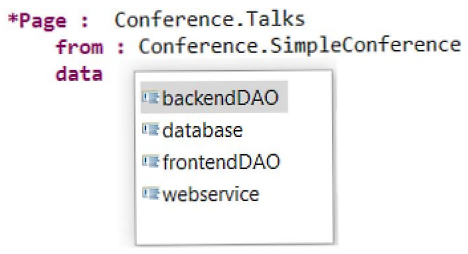
We used Xtext [12] and Xtend [11] to develop the infrastructure. Xtext allows the definition of a DSL in the form of annotated EBNF grammar. Based on this definition, it supports the generation of infrastructure components, such as a text-based instance editor, an EMF domain model, and an API for the DSL which can be used independently within a Java-based application. Xtend is a Java-based programming language with dedicated support for the definition of code generator templates. By using these tools, the rapid development and implementation of a high quality MDD infrastructure is enabled.

3.1 Domain-Specific Language for Joomla Extensions

We created **eJSL**, a DSL for the description of Joomla-based software extensions. The language consists of three parts: a part to model the data management of Joomla extensions (**entities**), a part for the definition of a page flow of extension views (**pages**), and a part for the description of an extension structure (**extensions**).

The *entities* and *pages* parts are platform-independent, that is, not bound to either Joomla or to WCMSs in general. The design of these parts has been influenced by the Simple Web Application Language (SWAL) presented in [9], which describes the data and the page flow of a web application.

The purpose of the *extension* part is the specification of particular Joomla extensions, rendering it the platform-specific part of eJSL. Extensions can be mapped to existing pages and entities. In particular, components and modules are extension types which mainly consist of views to illustrate any kind of data. Therefore the language allows optional references between these extension types and pages. By adding additional information to such a page reference, it is possible to describe dependencies between extensions in an abstract way. Figure 4 illustrates the definition of a page reference within an extension definition in an eJSL model. It is possible to define a use reference to the model of a component (frontend or backend), the database



```
*Page :  Conference.Talks
   from : Conference.SimpleConference
   data
```

Fig. 4: Page Reference within an Extension (eJSL Model)

[4] JooMDD can be downloaded from GitHub `https://github.com/icampus/JooMDD`.

of a conference or an existing webservice, which can also be part of a component. Hereby it does not matter, if the page reference is made within a component, or a module description. If no such reference is specified by the user, this knowledge must be provided in the generator instead.

To use the DSL we provide plugins for the most commonly used development environments in the WCMS domain which are *IntelliJ IDEA*, *PhpStorm*, and *Eclipse*. The editor plugin of JooMDD is customized for integration with each of these environments. The plugin provides a textual editor with syntax highlighting, error messages, dependency checks, and auto completion support for keywords and references between model elements.

3.2 Generator for Joomla Extensions

We have implemented our generator using Xtend templates. The main decomposition of the generated code follows the division of the DSL into entities, extensions, and pages, supporting traceability between models and the generated code. The generator supports the two use-cases we described earlier in Sect. 2.1 and 2.2. If a new and independent extension is to be developed, the generator creates the full extension code. In the case of using a an existing extension as reference within another extension, it is possible to describe the augmentation within the model (e.g. as part of a page reference as described in Sect. 3.1). So, the generator is able to only generate the depending extension, but not the existing extension anew. Both cases will be examined further within Sect. 4 and Sect. 5.

3.3 Tool Support for the Reverse Engineering of Existing Joomla Extensions

JooMDD supports developers during the creation or *forward engineering* of a new Joomla extension and the reengineering or migration of a legacy extension.

We developed the prototype **jext2eJSL** to support the reverse engineering of Joomla extensions by a model extraction from the code of existing Joomla 3.x extensions (PHP, HTML, JavaScript, and SQL files) as input. The tool creates an extension model based on the eJSL language with the main model elements as entities, pages and extensions. In particular, it supports the common Joomla extension types. Our specialized use-case for the tool is described in Section 2.2: The creation of a new extension with dependencies to an existing one. Usually the existing extension must be modelled as well to allow references on the model level. This step can completely be dropped by using the model extraction tool. The extracted model contains all information needed to model (and generate) new extensions based on the existing one. jext2eJSL matches the Joomla standard file and code schemes. Therefore, the input extensions must follow these schemes and implement the required patterns such as MVC for components to ensure that the extracted models are as complete as possible.

4 Iterative Process for Extension Development

In this section, we describe an iterative process to reduce development effort and error-proneness during the development of independent Joomla extensions. In addition we take the interdependencies between extensions into account to allow the development of new extensions that use artefacts of existing extensions as well. Each of these use-cases can be addressed during one iteration of our process. In fact, the second use-case requires an iteration of the process every time the existing extension evolves, to avoid side effects due to inconsistencies.

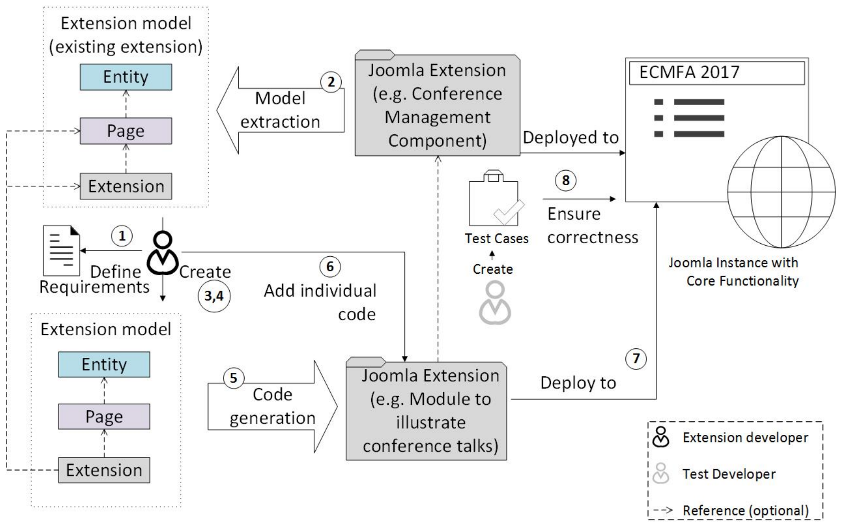


Fig. 5: One Iteration during the Iterative Development of Joomla Extensions

The process, outlined in Fig. 5, is made up of eight steps. In this illustration we focus on the development of components and modules, whereas our infrastructure provides support to adapt the process to all available extension types. One iteration consists of the following steps:

(1) Collect Requirements: Requirements for the extensions are collected in a suitable form, such as an analysis model in the form of a class diagram. In particular, these requirements may comprise managed data, views, relationships between views and data, and the extension structure. In this phase the decision to use artefacts of an existing extension within a new one can be made. This decision effects the subsequent steps of our process.

(2) Model Extraction (optional): In the case that a model of existing extensions, usually a component, is required, the *jext2eJSL* tool is used to extract such models automatically. Though, every evolution on existing extensions requires a new run of the process with this step as a main requirement to ensure the correct interplay between existing and dependant extensions. If an independent extension is to be developed, this step is skipped.

(3) Model Engineering: The identified requirements are used to create or update an extension model. In the initial development case, the modeller creates a new model with entities, pages, and the desired extension structure with regard to the requirements. If the extension should use artefacts of an existing extension, such as a module using the model of an existing component, a corresponding model of the existing extension must exist (created in step 2). References between extensions will be addressed during the subsequent code generation. To distinguish existing from new extensions, existing model artefacts can be denoted with a `@preserve` annotation. If the existing extension evolved since the last iteration and the corresponding model has changed in step 2, all affected dependencies must be re-engineered in the extension model. Otherwise it is not guaranteed that the extensions work correctly at runtime. In the case that a referenced part is completely removed within the existing extension, the dependency must be removed as well. Alternatively, the removed parts could be part of a new extension which can be used by the dependant one.

(4) Model Validation: To ensure that the code generator produces a valid result, the consistency of the input model needs to be validated upfront. In particular, the model of the existing extension which is used by a new extension must correspond to the existing extension's code. Code changes could lead to side effects in the Joomla page, which uses both extensions. Therefore, step 2 of the process must be performed in every iteration to ensure consistency between the model and code of the existing extension. If the modeller uses our text editor, the check happens automatically during editing.

(5) Code Generation: The component or module code is generated from the extension model. In each case the generator creates the full code for an installable extension. Thereby code is generated for all model elements that do not carry an `@preserved` annotation. In the case of creating an dependent extension, the specified references are incorporated within the generated code.

(6) Add Individual Code (optional): To support extensions with an elaborate application logic, the user may add individual code fragments to the generated code. A dedicated mechanism is required to guard such individual fragments for later runs of the code generator.

(7) Deployment to Joomla instance (optional): The generated extension can be installed within a running Joomla instance. In the case of a depending extension it must be provided, that all required extensions are installed as well. In addition they must be consistent to their corresponding models to ensure a flawless interaction between the new and the existing extensions.

(8) Test Creation: The correctness of the generated extension is ensured by tests. By performing integration tests, the correct interplay between the new and already installed extensions. Currently, these tests are required to be written by a human developer. Since the extensions under test are schematically redundant, the test cases usually present a large extent of schematic duplication as well, offering an opportunity for further automation. However, the automation of this step is left to future work.

The process is supported by our MDD infrastructure as follows: We provide jext2eJSL as model extractor for step 2, a DSL and corresponding editors for step 3 and 4, and a code generator for step 5. We do not provide dedicated support of the handling of individual fragments in the generated code, as required for the optional step 6, but an off-the-shelf solution can be used for this purpose.

5 Application of the Approach

In this section, we describe our experiences of applying the previously described process for both use-cases, creating a component and expanding an existing one by a depending module.

5.1 Creating a New Component

We devised a simple conference management component as an extension to the Joomla core. During the requirements step, we identified the analysis model shown in Fig. 6 to support the management of a conference with its participants, talks, agenda, and rooms. In our case it was sufficient to display these data in the standard Joomla CRUD views for the management of component-related data.

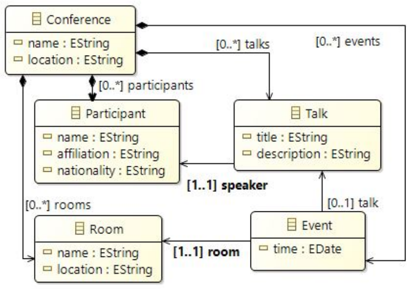
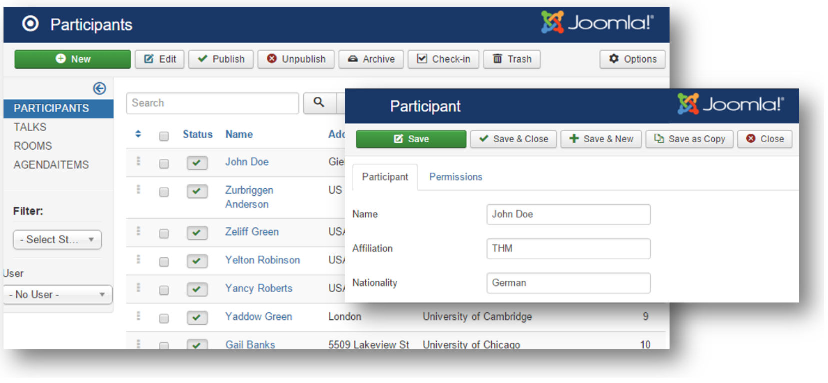


Fig. 6: Analysis Model for Conference Management

Specifically, each entity should be displayable in a custom list and details view, such as those shown in Fig. 7.



		Status	Name	Adc		
		✓	John Doe	Gie		
		✓	Zurbriggen Anderson	US		
		✓	Zeliff Green	US/		
		✓	Yelton Robinson	US/		
		✓	Yancy Roberts	US/		
		✓	Yaddow Green	London	University of Cambridge	9
		✓	Gail Banks	5509 Lakeview St	University of Chicago	10

Name	John Doe
Affiliation	THM
Nationality	German

Fig. 7: List and Details View within a Joomla Instance (Backend)

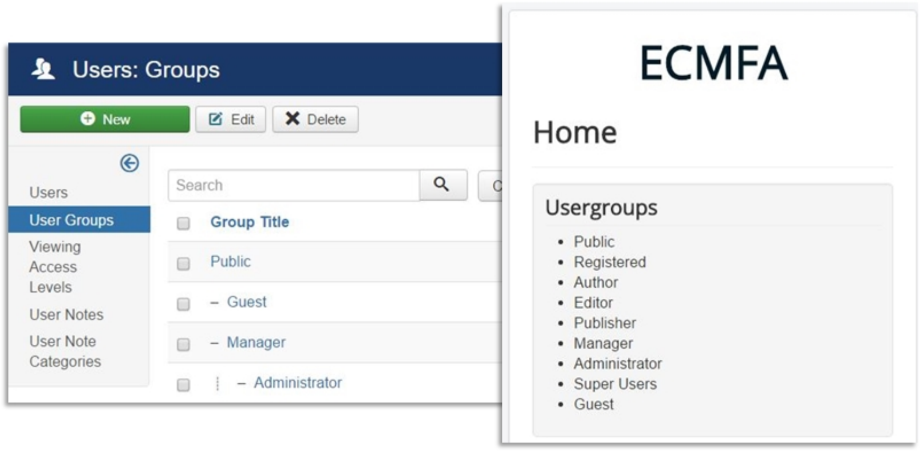
The image shows these views from the perspective of a Joomla administrator who can make the same views visible to site visitors using a menu entry. Based on these requirements, we designed an extension model[5] which can be used as input

[5] An excerpt of the extension model can be found in [18].

for our code generator. The generator then creates a full installable conference component that can be used to manage the required entities; no manual addition of individual source code is required.

5.2 Creating a Module using an Existing Component

We applied our approach to the *users* component, a core component which is pre-installed on each Joomla instance. The component manages the users and user groups of a Joomla instance as Fig. 8a illustrates. However, there is no way of illustrating the existing user groups within the frontend of a Joomla site. Therefore, we explore the case of adding a new module to the existing component using its model as DAO, to provide a new representation of user groups.



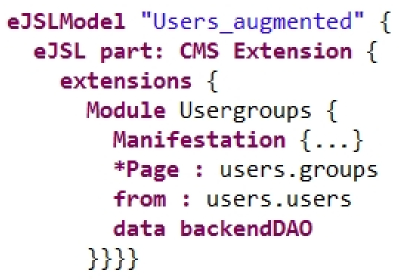
(a) Users Component (Management of User Groups in the Backend)

(b) New Usergroups Module using Data of the Users Component (Frontend)

Fig. 8: User Groups Management within Existing Component and New Module

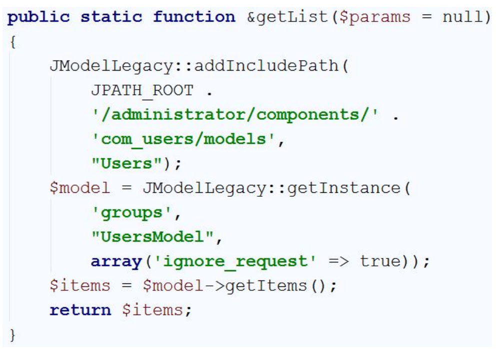
The users component is developed by core developers of the Joomla community. It was suitable for the exploration of our approach due to its high level of compliance with the Joomla standard, a general requirement for step 2 above and of our reverse engineering tool jext2eJSL. To this end, we first use jext2eJSL to extract an extension model from the given component. The resulting model provides entities, pages, and the extension specification which can be referred by new extensions. To avoid the generation of code for these existing elements, they are annotated with a `@preserve` tag.

To implement the new module, we create a new eJSL model and add a module specification as Fig 9a illustrates. Using the model as input, the code generator creates the module with the dependency to the existing component as shown in Fig. 9b. Since the generated file and code structure is fully compliant with the Joomla standard, the module can be deployed directly to an existing Joomla



```
eJSLModel "Users_augmented" {
  eJSL part: CMS Extension {
    extensions {
      Module Usergroups {
        Manifestation {...}
        *Page : users.groups
        from : users.users
        data backendDAO
      }}}}
```

(a) New Usergroups Module (eJSL Model)



```
public static function &getList($params = null)
{
    JModelLegacy::addIncludePath(
        JPATH_ROOT .
        '/administrator/components/' .
        'com_users/models',
        "Users");
    $model = JModelLegacy::getInstance(
        'groups',
        "UsersModel",
        array('ignore_request' => true));
    $items = $model->getItems();
    return $items;
}
```

(b) References within the DAO of the new Module

Fig. 9: New Usergroups Module

website. Once installed, it works together with the already installed organizer component by using its model as DAO for the data which has to be shown - in our case the user groups which are managed by the component (see Fig. 8b).

To explore the usefulness of our process for other components, we successfully applied it to the conference component of the first case and a component for resource management [7] we developed by hand over the course of six years.

6 Lessons Learned

In this section we address the lessons learned of our process based on its application. We discuss the strengths and weaknesses of the approach and point out the limitations of this work.

We investigated the usefulness of our approach by applying it in the domain of the Joomla WCMS, a particularly critical example domain due to its sophisticated extension mechanism that leads to many code and structure duplications. Due to these duplications the biggest strength of our approach reveals. During both application scenarios, development speed increased since most of the code was generated. In addition, the defect potential of the new extensions was tremendously decreased, because all generated fragments adhere to the given coding guidelines of Joomla. In both cases the extensions were installable and applicable without adding a line of code by hand.

Within the application of the second use case a requirement for creating a dependency from a new extension to an existing one was the creation of a model using our model extraction tool. After the extraction, the existing extension is depicted in an abstract manner. So, it could be used for being referenced by new extensions. A nice effect is the capability of using the model of the existing extension as a means of documentation, or a first version of the same extension, which may be developed in a model-driven manner. If an existing extension evolves, the model must be extracted anew. This could lead to inconsistencies between models of the existing and new extension. However a re-engineering on the model level allows a more rapid adjustment in contrast to a manual change of the dependencies in the extensions' code - especially if the dependencies concern different code fragments but are specified in same part of the model.

Beside the described strengths of the approach we discovered some weaknesses during the application. The main weakness is the management of individual code. If an extension which is developed using our approach evolves, individual fragments are not considered within the extension model. This could lead to problems at runtime, since the individual parts could depend on generic fragments, which have been changed or removed. To detect and fix error-prone fragments, an adequate test suite is required. Otherwise they wont be detected.

During the case of developing a new dependant extension, the problem of individual code occurs in an earlier stage. During the model extraction of an existing extension, only the parts which adhere strictly to the Joomla standard can be found and abstracted. Individual parts remain unnoticed and can only be reused by a new extension if the dependencies are added to the generated code by hand. However, this procedure impairs the benefit of our approach.

Even though our approach can be successfully applied, our work includes some threats to validity with regard to the applied development scenarios. In our application, we create a new independent component and a module, which uses artefacts of an existing component. These use-cases are common in the domain, but not the only ones existing. This is a threat to the conclusion validity, since our process is intended to develop extensions in general independent of their type. Especially, the second case should be further examined in future work. This includes the collection of possible dependencies between different extension types and their incorporation into our process. The main threat to external validity is that we only instantiated out approach for the Joomla WCMS. It yet has to be studied if it is also suitable for other WCMSs, since the infrastructure parts must be rewritten to the specific needs of the given WCMS. More extensive studies of the generalizability of our results are left to future work. However, the successful approach which is illustrated in this paper, allows an optimistic expectation for other WCMSs.

7 Related work

Several related works deal with applying model-driven engineering to application development in the WCMS domain. Most of these works propose platform-independent meta-models for the development of specific WCMS instances [15, 21, 24]. The approach by Saraiva et al. is the first to also investigate code generation for concrete WCMS instances [19]. However, none of these works addresses the extensibility of WCMSs through standardized extension types taking their interdependencies into account. As we have argued in this work, the creation of such WCMS extensions is a tedious and error-prone process of significant practical relevance. Dependencies between newly developed and existing extensions are not provided in any of these works. Our work is the first to tackle this challenge by providing suitable abstractions and automation facilities.

Model-driven principles have been applied to address augmentation issues in the WCMS domain. Trias et al. [22] introduce a reengineering method and a reverse engineering tool for the migration of complete WCMSs, for instance, from a web page to WordPress. Even though this approach can potentially improve the model extraction step in our process, it is currently tailored to WordPress, a WCMS with limited extensibility features. The usefulness for other WCMSs has yet to be investigated. Vermolen et al. [23] present an approach for the evolution of data models. As this approach provides a well-defined strategy to deal with changes to existing data entities, incorporating it into our work will help us to improve the flexibility during the augmentation of existing extensions.

Apart from these works, there is little recent research on the development practices for WCMSs, an observation that is confirmed by Norrie et al. [17].

General MDD approaches for the web domain such as the ones in [8, 14, 10, 20] can be used to create complete websites in a model-driven manner, but are not suitable for our considered problem since they do not address WCMSs and the model-driven development of their extensions.

8 Conclusion

Instances of Web Content Management Systems are commonly used as dynamic web applications in today's web. Using an open source WCMS, developers can add additional features by the use of software extensions, which can be installed into a running WCMS instance. However developing these extensions can be a time-consuming and complex task, even for experienced extension developers. Especially, the interdependencies between different extensions can lead to unwanted errors if they are not sufficiently considered during development. In this work, we introduce an iterative process using a set of tools to develop Joomla extensions in a model-driven way. In addition, we introduce a domain-specific language for the creation of abstract extension models and a code generator which derives a platform-specific implementation for the Joomla platform. This allows the rapid development of Joomla 3.x extensions adhering to both the platform-specific development guidelines and interdependencies between different extensions. To ensure the usefulness of our approach, we applied it to two development scenarios - the development of a new and independent conference component and of a new user groups module, which illustrates the data of an existing Joomla core component.

Our future plans span over two research directions. First, we plan to improve the existing DSL and tools, in particular to provide support for other WCMSs, such as WordPress and Drupal. Second, based on anecdotal evidence from our communication with Joomla representatives, there is interest in using JooMDD for the development of extensions within the Joomla community [18]. This situation allows us to provide our infrastructure directly to a large group of developers for a field study in vivo. Using this exposure opportunity, we intend to infer the usefulness of our approach empirically.

References

1. Drupal.org [online]: https://www.drupal.org.
2. Joomla!.org [online]: https://www.joomla.org.
3. Magento - eCommerce Software & eCommerce Platform Solutions[online]: https://magento.com/.
4. Usage Statistics and Market Share of Content Management Systems for Websites [online]: http://w3techs.com/technologies/overview/ content_management/all.
5. WordPress.org [online]: https://wordpress.org.
6. TYPO3 - The Enterprise Open Source CMS [online]: https://typo3.org/
7. J. Antrim. Technische Hochschule Mittelhessen - THM Organizer [online]: https://www.thm.de/organizer/.
8. M. Brambilla. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML.* Morgan Kaufmann, Waltham, MA, 2015.
9. M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice.* Morgan & Claypool, San Rafael, Calif., 2012.
10. S. Ceri, P. Fraternali, and A. Bongio. *Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks*, 33(1-6):137–157, 2000.
11. S. Efftinge and M. Spoenemann. Xtend - Modernized Java [online]: http://www.eclipse.org/xtend/, 02.12.2015.
12. S. Efftinge and M. Spoenemann. Xtext - Language Engineering Made Easy! [online]: https://eclipse.org/Xtext/, 11.02.2016.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1995.
14. A. Kraus, A. Knapp, and N. Koch. *Model-Driven Generation of Web Applications in UWE.* Ludwig-Maximilians-Universität München, München, 2008.
15. S. Martínez, J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, and J. Cabot. *Towards an Access-Control Metamodel for Web Content Management Systems.* In *Current Trends in Web Engineering*, volume 8295 of *Lecture Notes in Computer Science*, pages 148–155. Springer International Publishing, Cham, 2013.
16. S. McKeever. *Understanding Web content management systems: Evolution, lifecycle and market.* In *Industrial Management & Data Systems*, 103(9):686–692, 2003.
17. M. C. Norrie, L. Di Geronimo, A. Murolo, and M. Nebeling. *The Forgotten Many? A Survey of Modern Web Development Practices.* In *Web Engineering*, volume 8541 of *Lecture Notes in Computer Science*, pages 290–307. Springer International Publishing, Cham, 2014.
18. D. Priefer, P. Kneisel, and G. Taentzer. *JooMDD: A Model-Driven Development Environment for Web Content Management System Extensions* - Demonstration Paper. In *ICSE Companion '16: Comp. Proc. of the Int. Conf. on Software Engineering*, in press, New York, NY, USA, 2016. ACM.
19. J. d. S. Saraiva. *Development of CMS-based Web Applications with a Multi-Language Model-Driven Approach.* Dissertation, Universidade Técinica de Lisboa, Lisbon, Portugal, 2012.
20. V. Svansson and R. E. Lopez-Herrejon. *A Web Specific Language for Content Management Systems.* In *Proc. of the OOPSLA Workshop on Domain-Specific Modeling*, Montréal, Canada, 2007.
21. F. Trias. *Building CMS-based Web applications using a model-driven approach.* In *2012 Sixth Int. Conf. on Research Challenges in Information Science (RCIS)*, pages 1–6.

22. F. Trias, V. de Castro, M. López-Sanz, and E. Marcos. *RE-CMS: a reverse engineering toolkit for the migration to CMS-based web applications.* In *SAC '15: Proc. of the Annual ACM Symp. on Applied Computing*, pages 810–812, New York, NY, USA, 2015. ACM.
23. S. D. Vermolen, G. Wachsmuth, and E. Visser. *Generating Database Migrations for Evolving Web Applications.* In *GPCE '11: Proc. of the ACM Int. Conf. on Generative programming and component engineering*, pages 83–92.
24. K. Vlaanderen, F. Valverde, and O. Pastor. *Model-Driven Web Engineering in the CMS Domain: A Preliminary Research Applying SME.* In *Enterprise Information Systems*, volume 19 of *Lecture Notes in Business Information Processing*, pages 226–237. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.