Nested Web Application Components Framework
A Comparison to Competing Software Component Models

Svebor Prstačić and Mario Žagar
Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, Zagreb, Croatia

Keywords: Componentization, Framework, Component Nesting, Web Content Management System, Web Application, Comparison, Component based Software Engineering.

Abstract: There are many approaches and component models for Web application component development, of varying complexity, for different platforms using different technologies. All of which have a common problem – constrained component reusability. In this article, we summarize common software component reusability shortcomings of the most popular frameworks and component models they provide, and compare against the solution our own approach “framework as a component” provides to improve reusability.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

In software engineering, there is no shortage of component models. Almost every development framework, complex application or system defines its own component model and provides ways to implement components. Unfortunately, this makes component implementation simple, and component portability and reuse difficult. Components, once implemented, exist only inside a single framework that defines them (Wallace, 2010). As a consequence, similar or identical functionality is implemented over and over again using many different frameworks because none of the existing frameworks focus on or try to provide inter-framework interoperability.

Reusable components are inherently abstract, and harder to engineer, so making complex components reusable is very hard. As a result, most components forgo reusability for ease of maintenance and implementation (Schmidt, 1999). This makes reuse of components, even across applications that are created using the same framework complex or even impossible.

Component based software engineering defines two component model types: models that define components as objects in object-oriented programming, and the more complex components that constitute of multiple classes or other code constructs (Lau and Wang, 2007; Broy et al., 1998). These, more complex components, can be defined as architectural units – that provide a specific functionality to users, ie. a reusable forum application, built from many classes, class libraries, user interface templates, functions etc. Each such component can be thought of as a Web application in itself, and our, previously introduced, framework (Prstačić et al., 2011) focuses on improving such components' reuse.

We defined a component model such that components can nest, even recursively, and be more easily reused even within other applications built using different frameworks. Our testbed implementation is done in PHP, but the model can be applied to other technologies as well.

The framework in question is designed to be simple to reuse and integrate into an arbitrary Web application, as a component. Such an application we call the host application, ie. a blog application could be easily extended to provide commenting functionality for blog posts, and to enable attaching photo galleries to blog posts. The same same commenting component could be simply reused to provide commenting functionality for the photo gallery component.

When such integration of the framework and the host application is achieved, all the components implemented for our framework become reusable in the host application, which considerably improves those components' reusability. Because of this property, we call our framework *the extensions framework* and components built for it *extensions.*

In this paper we identify accepted component properties defined in the literature, that contribute to

Prstačić S. and Žagar M..
Nested Web Application Components Framework - A Comparison to Competing Software Component Models.
DOI: 10.5220/0004421401490156
In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2013), pages 149-156
ISBN: 978-989-8565-62-4
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

components' reusability, and compare our own results in this area, to some of the most popular component models and approaches to component reuse that they provide.

2 SOFTWARE COMPONENT MODELS

One of the pervasive needs of software development is increase of speed and reliability of both software development process, and the software created. Benefits that component-based software development (CBD) promises through reuse of well tested components, with well documented and defined functionality, it should be possible to create new software, combining and connecting existing software components instead of implementing the same functionality over and over again. Unfortunately, current component models fail to deliver on promises of CBD (Lau and Wang, 2007; Bose, 2011). This is evident in many very similar components that have been implemented over and over again for most popular frameworks in existence – a simple Google query for a word “forum” in addition to the framework name of choice will always yield results.

This is because software components are always built for a specific framework or application that uses a component model, which defines what a software component is, how it can be constructed, composed or assembled and finally deployed (Wallace, 2010; Lau and Wang, 2007; Crnković et al., 2011). Depending on the framework, components can vary in possible complexity and functionality.

If a component is built for a specific application, it can be thought of as an architectural unit, and such a component can span through many layers of the application, consisting of many classes and other code or binary constructs. Components built for a specific application are easy to reuse, but only inside other instances of the same application.

Similarly, components built for a specific framework are also architectural units, or even applications themselves. When reuse of such components is intended inside another application, that uses the same framework, required programming effort can be significant. The more complex the component, the harder reuse is. Although reuse can be complex, it is significantly less complex than reusing architectural unit type components built for a specific application, inside another application.

Thirdly, in object oriented programming, the most simple components are classes, or libraries of classes. Reuse of such components is easiest to achieve, but functionality that such components provide is minimal compared to an architectural unit type of components.

To enable component reusability in other applications or frameworks, software adapters (Bishop, 2007) can be employed. Software adapters translate required interfaces of one component to provided interfaces of another component. This is obviously inefficient and potentially very complex and has led to implementation of very simmilar components for various frameworks.

2.1 Nested Framework Component Model

Our framework makes runtime and design-time assembly of components possible, which is discussed further in section 4. It makes assembly of nested components possible, ie. a forum component that extends the functionality of that same forum component. This makes possible the creation of a forum to debate a single forum post or topic, which is dubiously useful in itself, but in some cases could be, ie. using a commenting application nesting to enable comment replies, while the component itself provides only the simple, basic functionality.

To achieve this, it provides both abstract classes and objects as base building blocks for component development. Use of these building blocks, for example, provides simple access to host application's execution context (Prstačić et al., 2011) with relevant data commonly found in Web applications (Prstačić et al., 2012) – user session, current user data, current user permissions etc., are all translated by the framework and provided to extensions in a form that they can use. Integration of the framework into the host application involves implementation of abstract classes that are essential to providing this translated execution context.

Once implemented, components can be used to extend existing applications, that were built using a different framework, with minimal effort. This is possible because our framework is designed and built as a component that is ready to be integrated into arbitrary Web applications that we call host applications. When this integration is achieved, reusing any component of the framework is possible using a single line of code, as discussed further in section 4.2.

3 POPULAR COMPONENT MODELS PROVIDED BY THIRD PARTY FRAMEWORKS

There are many component models in existence, and frameworks built to provide component based Web application development. Each defines what a component is, and even a superficial analysis shows that many of the frameworks define components in a similar way and use the same design patterns. For example, separation of user interface definition and design from user input handling and application logic functionality using the model-view-controller (MVC) pattern (MSDN, 2003), localization support, database access – all are common and mandatory in a Web application framework (Walker, 2012; Grails, 2012; Django, 2012; Symfony, 2012; Joomla, 2012). Still, what defines a component in one framework doesn't in another.

In the following sections we introduce some of the most popular Web application frameworks' component models and compare them at the higher level.

3.1 ASP.NET

ASP.NET provides a few different component models to build Web application components, one of them is Web forms and ASP pages, that are a paradigm to separate application presentation from application logic similarly to MVC (MSDN, 2011), (Alpaev, 2011), and make Web application development similar to the event based desktop Windows programming.

ASP.NET and Web forms provide a fast and simple way to reuse components of varying complexity: data access, user interface components, common application logic and combinations of those. However, the is no way to easily integrate architectural unit type of components. For example, a more complex application, such as .NET Nuke CMS, provides its own array of interfaces to create architectural unit type of application components (Walker, 2012). Components for such applications are not easily reusable inside other applications, regardless being built with the same underlying framework. Additionally, more problems can arise if a component is built using another component model than the application uses, for example ASP.NET MVC (explained in the following section). Integrating such a component would require additional programming work to both the application in which we want to reuse it, and the component.

3.1.1 ASP.NET MVC 3

ASP.NET MVC 3 (Prstačić et al., 2011; MSDN, 2003) is the newest component model available in ASP.NET. It provides an alternative way to define Web application components: a “Razor” template engine, which strongly resembles Smarty (Prstačić et al., 2012; Smarty, 2012) for PHP, both in syntax and the way it is used. Razor also provides “HTML helpers” which are a way to provide functionality that Smarty plugins (Prstačić et al., 2012) provide. Consequently, various components can be nested inside each other so that they combine various views of various models. For each reuse case however, one has to know what parameters a component expects, or in other words, a component can define an arbitrary required interface. This provides flexibility when developing components, but also has adverse side effects to component reuse.. For example, reusing components might have to do the work of preparing data that a reused component expects, or a reused component could be implemented in a way to integrate with a specific reusing component, making the reused component hardly reusable for other arbitrary components.

3.2 Groovy on Grails

Groovy on grails is a modern and versatile Web application framework that provides Web development using a dynamic language (Groovy) and runs on a Java virtual machine. This combination provides easy use of all the power of Java and Java components, and flexibility of Groovy. Groovy provides a MVC pattern for development and a strong Object-relational mapping (ORM) database access layer. Classes that define data entities are called domain classes, and controllers are called domain controllers. Groovy on grails defines its own template language to define views.

Domain classes, controllers and views can be packed as plug-ins, which makes them easily portable to other Groovy on grails applications or projects. Integration of plug-ins consists of configuration editing – what controller should be invoked for which URL, but interactions between components have to be implemented individually. So, again, reusing an architectural unit component requires programming, and more than just a few lines of code, especially if different components' data has to reference each other.

3.3 Django Framework

Django is a Python Web development framework

that aims to provide great component reuse capabilities (Django, 2012). In theory, each component developed using Django is an application that can be easily reused alongside other applications – thus forming a greater whole application. Django provides a very flexible and powerful ORM. When a developer creates a model – Python class, the framework is capable of creating the database schema, and even provide the administrator's user interface to manipulate data that the model represents.

But reusability of all the Django applications strongly depends on their implementation. The developer has to balance between creating applications that are big and monolithic or too small to provide a unique functionality. Only those well balanced can be easily reused.

For example, a reusable application should provide signals for other applications' models and injection points in its views. This introduces complexity because there exists no convention in which the reusable applications communicate inside the framework, and this communication depends on implementation of each of the components and relies almost completely on the experience and effort of the developer. Even then, models often have to be extended with properties that include references to models of the reusable applications (Altman, 2011; Django, 2012), which decreases reuse inefficiency.

3.4 PHP Symfony Framework

In Symfony (Symfony, 2012) components that provide some reusable application functionality are called bundles. Each bundle is actually a separate application that can be executed by the Symfony framework. Symfony also uses a MVC pattern as its architectural pattern for bundles, so each bundle consists of models which are called entities, controllers and static files.

Other artifacts that can be packaged into a bundle are static files like view templates, configuration files, CSS and JavaScript files. So, reuse of a bundle consists of installing its files, and writing a few lines of configuration that tell the framework to use it. Interaction between various bundles or their MVC components can be achieved only if the components themselves provide communication interfaces, In many cases, the entities (models) also have to be modified to reference each other across bundles, which again introduces a level of inefficiency in component (bundle) reuse.

3.5 Common Component Reuse Failings

Consider a web application that provides news article publishing through one of its components. Such a component would define a database model to store articles, a few classes to handle the data, user interfaces and a controller to handle user actions. If we were to add commenting functionalities, we have a few options, and this applies to all the component models of the previously mentioned frameworks.

The first option is to create a commenting component that integrates tightly with the news component – at the database level, comments reference a specific news article, and at the presentation level – the news component simply invokes rendering of the comments component for each article. This solution is simple, but greatly hinders the commenting component since it is practically implemented as a part of the news component.

The second option is to create a generic comments component that has a mechanism to provide comments for more than just the news component. The commenting component can thus provide a set of required interfaces and handle the data abstraction itself. This is inefficient because it involves implementing functionalities as a part of the component that should be a part of the framework. Additionally, reusability is further hindered by the fact that the component will only function inside a specific framework or application that it was built for. Our framework offers a solution to these problems, specified in the following section.

4 THE EXTENSIONS FRAMEWORK, COMPONENT MODEL

To compare how our framework and component model improve component reusability to competing approaches introduced in the previous chapter, we found the following criteria to evaluate component reusability (Broy et al., 1998; Jeffrey, 1994) to be most applicable:

- existence of visual tools for component design-time assembly,
- existence of run-time visual component composition tools,
- components are portable between frameworks using the same programming language,

- components are cross-platform or cross programming language,
- functional completeness of a component,
- cohesion and low coupling,

In the following sections we explain the technical intricacies of our approach and compare it according to applicable criteria.

4.1 Component Nesting and Communication

Using all the mentioned Web application frameworks, nesting of components is possible, but the frameworks don't support it directly, it has to be specifically implemented for each component pair to make it possible. If a component wants to use another, it has to manage each used component explicitly through a custom set of interfaces that the child exposes. Additionally, integration points can be implemented on different layers, and all these choices are a subject to developer's whim. This introduces complexity in the way components interact, which has to be handled separately for each pair of interacting components and forces a component developer to implement component interoperability rather than focus on developing a single component.

This also causes components' cohesion, a level in which components work together to offer a composite functionality to be lower, and coupling, meaning interdependence between components higher, which also makes probability of failure of the system higher if just one of the components fails (Eder, 1994).

Components, in general, are built to interface with the framework using predefined required interface(s) (Figure 1). These interface are defined by the framework and provide data necessary for component execution, but also provide a way for a component to return execution results on framework request. Also, from Figure 1 we can see that a component can define a set of provided interfaces. This is fine, but those interfaces are not defined by any of the frameworks, so components that connect through them are implemented in a specific way. Obviously, this is a problem and this approach introduces complexity in both integration and future component maintenance.

To solve this, our framework acts as a mediator between components. Every component that wants to interface with another can achieve this through framework's IHookData interface (Figure 2), that is translated into an IHook interface.

In this way, the framework as a component effectively delegates interfaces between the two components. As a consequence, component reuse and

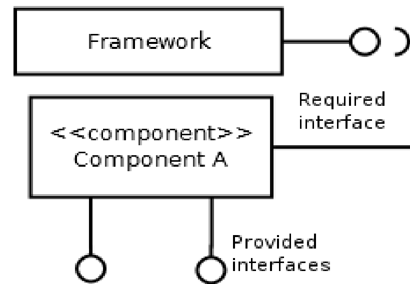

Figure 1: Usual way frameworks interface components.

implementation in our framework has to be done according to only two simple rules:

- components (extensions) must be used through the *IHookData* interface of the framework.
- extensions will always be provided a IHhook, which is extensions' required interface (Prstačić et al., 2011)

Pondering the basis of component integration, we can conclude that these rules make it simpler and predictable. Components have to exchange data in order to provide a unified user experience or provide a unified functionality. For all the previously mentioned frameworks, prerequisites are that a component communicates with the framework, and always, communication between components is left to be defined and implemented by the component's developer.

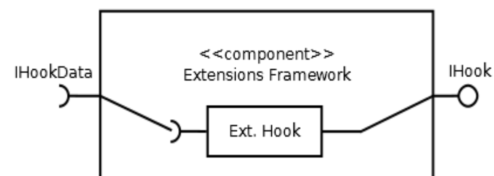

Figure 2: Extensions framework component interface.

Our framework, through enforcement of the two mentioned rules, provides component reuse and nesting. This both makes functional completeness easier to provide and also lowers coupling because components only communicate with the framework. Implicitly, it increases cohesion – if a component communicates and works efficiently with one other component, it will also work and communicate with another that communicates using the same interfaces.

4.2 Technical Properties

There are four main properties that the component model defined for our framework provides:

- nested and composite components can be defined by the user on run time, as well as developers on design time
- components don't have to reference other components' data directly, and they don't have to handle the abstraction themselves
- the framework provides a way for components to notify child components of parents' data changes
- user interfaces assembly, such as form nesting, is handled by the framework
- component reuse and integration can be trivially implemented in the higher layers of component architecture, such as the presentation layer

In the following sections we explain these differences in more detail. We will also compare the benefits of the component model used in our framework, as opposed to competing models.

Let's once again consider the example introduced in section 3.5. To use the commenting functionality for the news component, we should have to write a trivial one line of code. Ideally, this one line should make reuse of additional components possible. Assuming the integration of the framework as a component into the host application is achieved as further explained in (Prstačić et al., 2012), the host application provides the execution context to the extensions framework through the integration interfaces (II), while the component of the host application requests execution of a specific extensions framework component through component interface (CI), called the *IHookData*. Data provided by the host application component through CI and the host application execution context through II is combined by the framework to provide the computed execution context for the Extension (Figure 3).

In our case, extending the news article with commenting capabilities can be achieved with one simple line of code. The following is an example usage of our framework's Smarty plug-in *v2ext* that

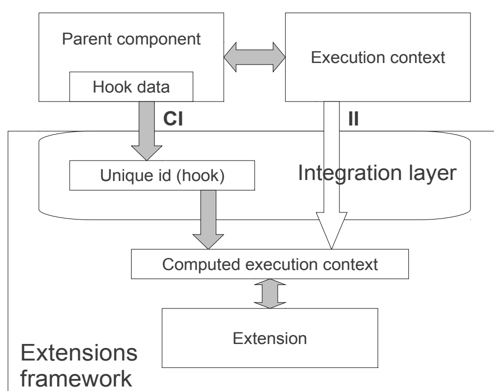

Figure 3: Component and framework interfaces.

achieves extension reuse:

```
{v2ext _name="Comments"
_id="comments_`$mod_prefix`"
_content_name="news_article"
_content_id=$news.news_id
_handler="portlet_news"}
```

The v2ext plug-in is an encapsulation of the *IHookData* interface. The extension must contextualize all the data it saves so that it can provide meaningful functionality and present expected data throughout different instances. This contextualization is implicitly done by the framework, which provides a hook – the only data an extension ever references. All the required data to create a hook are provided by the component that uses the v2ext plug-in:

- _name – a name of the extension main controller class;
- _id – a unique identifier of the extension for the parent component
- _content_name – name of the content type that the parent component wants the extension to hook onto
- _content_id – the id representing a single data entity, ie. a news article
- _handler – the handler class for the content type defined by the _content_name argument. This is optional and ie. used when the framework wants to access a URL of a specific article.

Since there are no relational data connections between components, the framework provides an event driven mechanism to handle data changes. The host application, or an extension can raise a data change event, giving it the same context it would when creating a hook. Consequently, if such a hook exists, the framework will notify all the hooked components that are expected to perform required actions with their own data.

4.3 User Defined Component Structure

The usual way Web application components can be reused and made to interact, inside their native frameworks, is by writing a few lines of configuration and program code. This is true for all the competing mentioned frameworks. More flexibility is provided by content management systems (CMS) that often enable users to arrange components, content and their locations in the application on runtime. But if a user wants to extend functionality of a com-

ponent with an arbitrary component on runtime, even a CMS will fall short.

Our framework solves this by providing empty hooks. Extension developers can define these hooks in the desired places of an extension's view, and the user can choose which extension should run at these places.

A hook can be defined as easily as we initiated execution of a specific extension in the example from the previous section, using the *v2ext* plug-in. The difference is, we can omit the *_name* and *_id* parameters:

```
{v2ext
_content_name="news_article"
_content_id=$news.news_id
_handler="portlet_news"}
```

4.4 Presentation Layer Integration and Component Reuse

What defines Web applications from functionalities' point of view, are the graphical user interfaces. So, we wanted to make combining and constructing functionality of multiple components at the presentation layer possible. Extending applications in this way makes a lot of sense, but also, creates some issues that have to be solved. In such composition, data that one component is displaying should be used as a reference for some functionality or data provided by another component.

Our framework solves this by simplifying component and data interaction by forcing each extension to define two main user interface states of a component. The default state and a state in which an extension is displaying a user interface to edit data. We found this simplification good enough for most test cases. For example, if we' re editing an article in a content management application, and that article has a photo gallery extension hooked onto it, we can request the gallery extension to also enter edit mode and provide controls to manipulate the images and the gallery, which cascades to gallery's child extensions. This doesn't imply extensions cannot provide multitude of states that can be invoked by the parent component, but for now, only the default and edit states are handled by the framework.

5 CONCLUSIONS

To be able to provide functional completeness of components, we had to forgo some performance and data consistency that comes inherently with more tightly coupled components, for example, acting on data changes takes longer when dispatching events, than letting the database act on relational rules and data triggers, but the value provided is greater.

Our framework doesn't provide any kind of component definition language which would enable programming language translation of components, so in the current state, it is only usable for PHP host applications. Although PHP works on all major operating systems and Web servers which makes it cross-platform, this is a hindrance.

In comparison to other solutions, it does satisfy most of the criteria:

- it provides run-time visual component composition
- components are portable between frameworks using the same programming language and since PHP is cross-platform, it also inherits cross-platform capabilities
- it does provide ways to create functionally complete and independent a components
- and through these properties it does increase cohesion and makes low coupling of components possible

It does not provide design-time visual composition tools and isn't cross language, although it could be easily implemented to work for other programming languages.

Our framework, in contrast to other mentioned is a software component by itself, an adapter between components built on top of it (extensions), and also between extensions and the host application. It is a component that is easily used and whose rendering output can be easily obtained. This property of our framework is what makes it unique, and what makes reuse of many components possible with the effort of integrating only one.

Our future work will include analysis and application of software complexity metrics to measure component reusability efficiency increase that our extensions framework provides.

ACKNOWLEDGEMENTS

This work is supported in part by the Croatian Ministry of Science, Education and Sports, under the research project “Software Engineering in Ubiquitous Computing”, project number 036-0361959-1965.

REFERENCES

Wallace, B., 2010. There is no such thing as a Component, accessed 19 April 2012, <http://existentialprogramming.blogspot.com/2010/05/hole-for-every-component-and-every.html>.

Schmidt, D. C., 1999. Why Software Reuse has Failed and How to Make It Work for You, *C++ Report magazine*.

Lau K, and Wang, Z., 2007. Software component models, *IEEE Transactions on software engineering, vol. 33*, no. 10.

Broy, M. et al, 1998. What characterizes a (software) component?, Software – Concepts & Tools 19, pp. 49-56, *Springer – Verlag*.

Prstačić, S., Voras, I. and Žagar, M., 2011, Nested componentization for advanced Web platform solutions, *Proc. ITI 2011, 33rd Int. Conf. Information Technology Interfaces*.

Poulin, J. S., 1994. Measuring Software Reusability. *Proceedings of 3rd International Conference on Software Reuse,* Brazil.

Eder J., Kappel G., and Schrefl M., 1994. Coupling and Cohesion in Object-Oriented Systems, Technical Report, Univ. of Klagenfurt.

Bose, D., 2011. Component Based Development – application in software engineering, Indian Statistical Institute.

Crnković, I. et al., 2011. A Classification Framework for Software Component Models, *IEEE Transactions on Software Engineering: Volume 37*, Issue 5.

MSDN, 2009. Layered application guidelines, accessed 20 July 2012, <http://msdn.microsoft.com/en-us/library/ee658109.aspx>.

Prstačić, S., Kroflin K. and Žagar, M., 2012. Interfaces of nested Web application framework as reusable software component, MIPRO 2012: 35th Int. Convention *Proceedings, Croatian Society for Information and Communication Technology, Electronics and Microelectronics*.

Bishop, J., 2007. C# 3.0 Design Patterns, O'Reilly Media

MSDN, 2003. Model – View – Controller, accessed 20 December 2012, <http://msdn.microsoft.com/en-us/library/ff649643.aspx>.

Walker, S., DotNetNuke 4.0 Module Developers Guide, accessed 20 April 2012 <http://www.dotnetnuke.com/Resources/BooksandDocumentation/ProjectandTechnicalDocumentation/tabid/478/Default.aspx>.

Grails, 2012. Groovy on Grails documentation, accessed 5 May 2012., <http://grails.org/doc/latest/>.

Django, Django documentation, accessed 5 May 2012., <http://docs.djangoproject.com/en/1.3/>.

Symfony, 2012. The book, accessed 5 May 2012., <http://symfony.com/doc/current/book/index.htm>.

Joomla, 2012. Developing a MVC component, http://docs.joomla.org/Component [1/10/2011].

MSDN, 2011. ASP.NET 4.0 Web Pages, accessed 20 April 2012, <http://msdn.microsoft.com/en-us/library/fddycb06>.

Alpaev, S., 2005. Applied MVC Patterns, VikingPLoP

Smarty, 2012. Smarty documentation, accessed 20 December 2012, <http://www.smarty.net>.

Altman, P., 2011. How I write Django Reusable Apps, accessed 5 May 2012., <http://paltman.com/2011/12/31/how-i-write-django-reusable-apps>.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS