Development of a PHP, MVC based, Component Orientated Website Content Management System

By Martin Cassidy

Abstract

A website content management system allows a user, with no technical skills or experience, to produce a website using a web browser based portal. Content is only one aspect of a website however, the second aspect being layout and design. Presently, content management systems offer little or no support for the creation and manipulation of a website layout and design for users without technical skills. This project aims to rectify that, and sees the development of a new content management system which follows an *anything anywhere* approach; where users are able to produce a layout by positioning prefabricated user interface components into a composition – forming a webpage. A new framework was developed specifically to offer the component ideology, adapting the model view controller architecture to enable component orientation.

Research into both existing framework and existing content management systems was carried out to identify their strengths and weaknesses. An investigation into component orientation and the model view controller was conducted to facilitate the formulation of the new frameworks architecture. The research and investigation results were used as the starting point for requirements elicitation.

Both the framework and the content management system were then planned, designed and implemented using the Agile development methodology. Neither products were finished, however the development has produced a proof of concept for both, demonstrating that a content management system can offer layout and design creation functionality; and that a model view controller based framework can incorporate a component orientated ideology.

Whilst still in their infancy, I believe both the framework and the content management system have the potential to one day measure up to, and perhaps even rival existing frameworks and content management systems.

Accompanying Resources

The source code, generated documentation and working applications created during this project are also available online.

http://mcassidy.org.uk/university/final-year-project/

Contents

Table of Figures

Table of Tables

1. Introduction

Content management systems take advantage of recent advances in web technologies by allowing any user to create their own website, often with very complex and dynamic functionality, without needing any programming skills or prior experience. However, in my experience users want more than simply managing their website content, they want to be able to manage the design and layout of their website as well, a severely neglected feature of existing systems. This project will address this issue and see the development of a new content management system.

1.1 What is a CMS?

A website content management system (CMS) allows an authorised user to create, modify and publish different types of content on their website, for example blogs, videos, images or articles. This functionality is usually offered in the form of a web based portal. There are numerous CMSs around, however they all suffer from the same drawback: they live up to their name too literally by being very good at managing content but very bad at managing the design and layout of a website.

Every CMS I have encountered has had a layout and design based on a template approach – where a webpage layout is generated from HTML and CSS template files. This approach will provide three methods of customising the layout of a website: the user must choose from a finite catalogue of templates, pay for a designer to create a template for them or create one themselves.

I believe that even a novice user, creating a website for the first time, will have at least a basic idea of what they want their website to look like, even if they lack the skills to implement it. Choosing from a template catalogue will most likely leave their ideas unfulfilled. As many CMSs are free, and are chosen by a user for that very reason, paying for a designer is an unsuitable option. Creating a template would require a user to know HTML and CSS, and understand how the CMS processes

templates, adds dynamic content and renders a page. A novice user would face a very steep learning curve and would mostly likely be put off by this option.

In this project, I will be developing a new CMS which, as well as letting users create and manage content and media, will offer the functionality for them to create, entirely from scratch, the layout and design for their website using nothing more than a web browser based interface, without the need for any prior knowledge or experience. It will be developed in PHP, with which I am very familiar. It is also a ubiquitous web language and would allow the CMS to be deployable on most web servers.

1.2 Component Orientation and Frameworks

For a system like this to be successful, it will need to offer complete and granular control over every visual aspect of every part of a website. It must be possible for a user to place anything anywhere. This would require the CMS to be developed with an underlying component orientated architecture. Additionally, as is a feature on many CMS's, it will need to be open for developers to write new plug-ins for bespoke projects. Such functionality would call for a framework to be used. There are however no PHP frameworks that I would want to use for this as there are none that offer a sufficient level of component orientation. Whilst on my work placement I worked with two Java frameworks: Apache Wicket and Spring Framework – the best I've ever come across. The CMS I will create will be written using a new PHP framework which I will write, incorporating concepts from Wicket and Spring. As well as being entirely component orientated, the new framework will also address a number of issues which I have noted in other PHP frameworks – issues that have contributed to my dislike of them. Firstly, the use of scriptlets, secondly lack of flexibility when working with mark-up in an object orientated way and thirdly support for Ajax.

1.3 Project Overview

This project consists of two parts; the first is the research and development of a new component orientated framework. I will analyse and evaluate existing PHP frameworks; I will also look at the Wicket and Spring frameworks to better understand the concepts on which they operate so they may be applied to the new framework.

The second part is the research and development of a new content management system. I will look at existing content management systems and evaluate the ways in which they offer layout customisation.

For both parts, I will develop through research a clear set of requirements. Determination of which will be from both negative aspects of existing systems, which I will aim to rectify in the new system, and positive aspects of existing systems, which I will include in the new system.

Development will follow the agile methodology as it is highly flexible and will allow development to start before the final design is completed. Although the software development lifecycle will be completed in iterations, it will be presented here for reasons of context and continuity, as though it were not. Other methodologies for both analysis and development are detailed in appendix one.

Although an integral part of the CMS, the framework will be developed independently, so that it has the flexibility to be utilised by many applications.

2. Software Frameworks

A software framework is a design and implementation of a specific domain, providing abstracted and reusable functionality to client classes. A client class will make use of functionality offered by a framework or extend a class within the framework to specialise its implementation. Frameworks offer developers the means to rapidly develop an application using abstract logic common to all applications in the framework domain. Utilising a framework reduces development time by encouraging code reuse and decreasing software bugs, as the code within the framework is expected to be robust and well tested. (DocForge, 2011)(Riehle, 2000)

> *"Object-oriented frameworks promise higher productivity and shorter time-to-market of application development through design and code reuse (than possible with non-framework based approaches)" (Riehle, 2000)*

This chapter examines several of the most widely used PHP frameworks, providing insight into how they have been designed and implemented, which will aid the design and development of the new framework. Additionally I will expose any drawbacks the frameworks have, refining the reasoning behind my decision not to use an existing framework.

To produce a fair analysis and evaluation of each framework, as well as using accompanying literature, I will conduct some firsthand research by creating the same, simple application using each framework.

In this chapter, the developer is defined as a software developer using a framework to create an application.

2.1 Zend Framework

The Zend Framework is a library of modules written in PHP for developing web applications. (Allen & Lo, 2007) Its modularity breaks up the framework into many components: an MVC system, a portable database component, filters for data validation and sanitation – to name but a few. All of the modules are loosely coupled enabling each one to be individually implemented. A developer may utilise only the components they need for their project – either by integrating a subset of modules into an existing project without disruption, or by using Zend entirely but not overloading the system with unnecessary and unused modules.

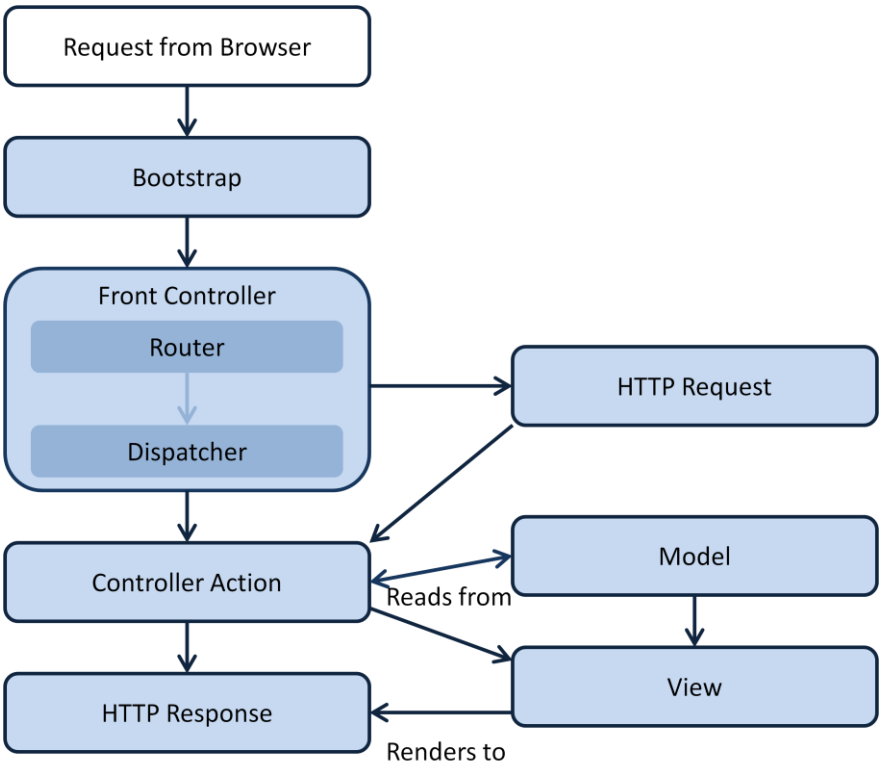

Figure 1 The Zend Framework Lifecycle **(Allen & Lo, 2007)**

The framework is highly flexible and can be adapted to suit the needs of any application. The default settings and behaviour have been designed with the most common scenarios in mind; however the application contains many "*flex points*" which allow it to be customized at any phase of its lifecycle.(Allen & Lo, 2007) This is more of a hook approach than design by primitives, the former being far simpler to implement but lacking the flexibility of the latter.

To use the MVC system a developer will write action classes (controllers) containing multiple methods, each forwarding to a view. Upon receipt of an HTTP request, a bootstrap process to initialise the application will run, followed by the front controller which analyses the request URI, identifying and instantiating an action class, then invoking the required method (routing and dispatching, see Figure 1). The action method may call upon a model object to access or modify information, from a database or other information source. Instance data will be set within the action object and be used by snippets of PHP in the view HTML to render them as dynamic values.

2.2 Cake PHP

Cake PHP is a framework for rapid application development. Rather than providing a set of modules, which are frequently cumbersome to customize (Golding, 2008), cake provides a set of classes for the user to extend, facilitating customization through polymorphism. These classes are wrapped inside a cake application which follows the model view controller architecture.

Cake provides not just code abstraction, but also offers pre-built skeleton components which can be implemented very quickly to create an application. These include "*scaffolding*" for automated view generation, the "*bake script*" for model generation and a set of helpers for common client and server side tasks, such as JavaScript, pagination and caching.

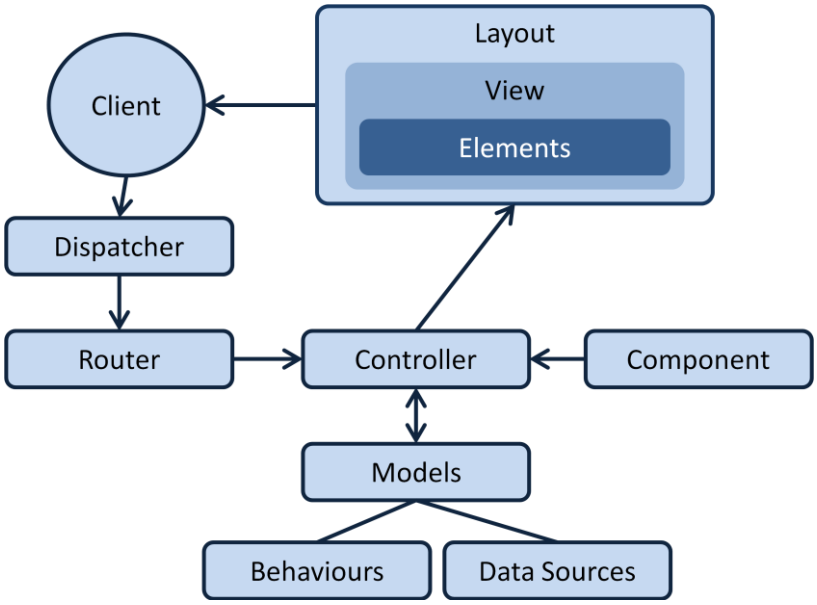

Figure 2 The Cake PHP Application Lifecycle **(Golding, 2008)**

A developer will write controller classes each containing action methods. An HTTP request is analysed by the router to extract, from its parameters, the controller and action it corresponds to, Figure 2. The controller is instantiated, and the action method invoked with arguments derived from the URI parameters. The action method will call upon one or more models, to retrieve any data required for the request; it may also call upon components which run business logic to refine data from the model object or request arguments. Components also help with tasks such as sending emails or authentication. Any dynamic value needed by the view is set in a data object. Finally, a controller sets the target view. A view may be wrapped in a layout which provides a global HTML template. Views may be composed of elements which are fragments of reusable HTML. The HTML of a view will contain scriptlets to output values from the data object. (Cake Software Foundation, Inc., 2011)(Golding, 2008)

2.3 Symfony Framework

Symfony is a PHP framework which optimizes the development of web applications (Zaninotto & Potencie, 2010). Symfony follows the model view controller architecture, provides automation for common tasks and offers many prebuilt classes and tools to speed up development.

Symfony makes use of *YAML* for configuration, *Doctrine* for object relational mapping, and *Twig* for dynamic HTML templating. They are not forced upon developers however, and each may be replaced with a functionally similar system or with plain HTML or XML. Symfony supports this replacement across the entire application. Every module, including the core, is packaged into a "*bundle*". Bundles allow the application to be extended in any area through the creation of custom bundles or usage of bundles written by a third party. (Sensio Labs, 2011)

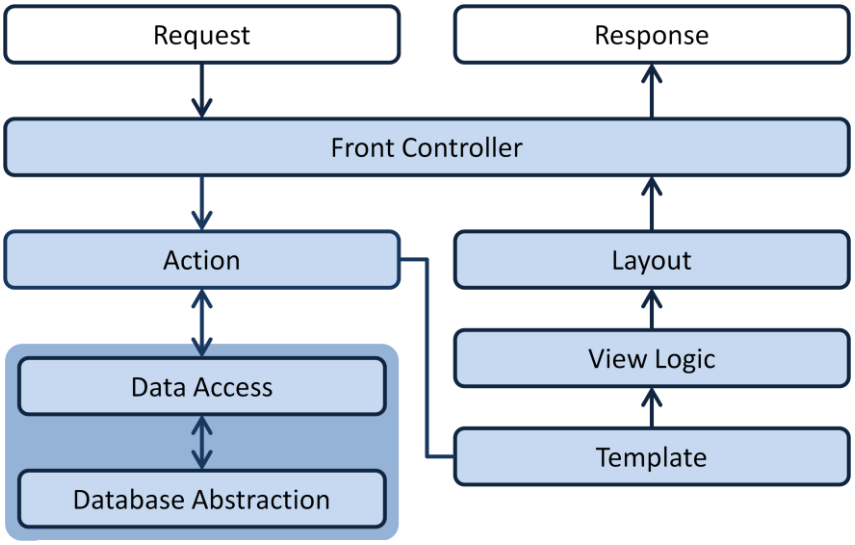

Figure 3 The MVC system of the Symfony Framework **(Zaninotto & Potencie,**

When an HTTP request is received, the front controller will invoke a router which matches the URL against patterns defined in a configuration file. A developer will write controller classes containing multiple methods, each of which produces a response for a particular action. An action is associated with a URL pattern. When a pattern that matches the URL is found, the action method will be invoked. The method may forward the request to a view or another action. Forwarding works in a similar way to invocation and includes arguments which may have been generated by the action

method, retrieved from a database through a data access object, or extracted from the request. A data access object passes database independent queries to an abstraction layer which performs the actual database operation.

The view consists of a template, which is the mark-up for an individual page, the view logic, in which arguments passed by the action are mapped onto the template, and the layout which is a global template for the entire application or a subset of pages. The full process is shown in Figure 3.

2.4 Implementation Analysis

Using each of the frameworks, I have produced a database driven application which allows creation of a music catalogue, using Ajax wherever possible.

2.4.1 MVC System

In years gone by, PHP applications would have no code separation of any kind. HTML and PHP would reside in the same file with no separation of concerns, making applications difficult to debug and maintain. All three frameworks sensibly extract PHP from HTML and, by following the MVC architecture, and taking advantage of recent advancements in PHP object orientation, provide abstracted and reusable code. One of the goals for Zend at its inception was to facilitate code separation in this way. (Allen & Lo, 2007)

A view in both Zend and Cake comprises HTML and PHP in the same file. PHP scriptlets are embedded into static HTML to output dynamic content or perform selection and iteration on the mark-up. Inadequate separation of concerns is not an issue as the scriptlets are only functionally responsible for presentation logic; however a complex layout requiring a significant amount of scriptlets will produce code that is cluttered, unreadable and consequently less maintainable and less reusable. Furthermore, by outputting properties set in the controller, a view is entirely

dependent on those properties having been correctly set. Scriptlets in Cake often contain method invocations to construct and output pre-defined mark-up, increasing complexity further.

A scriptlet will frequently make use of the *this keyword* as though it were within the scope of a class. I consider this out of context usage to be incredibly bad practice as it blindly assumes the value of *this* will be as expected. The value of *this* is usually the instance of the controller invoking the view, tightly coupling the view to the controller, making it impossible to reuse or requiring that the value of *this* is not always the same class instance, which is potentially problematic as methods and properties utilised by the scriptlet would need to be identical across controllers.

Symfony solves the scriptlet problem by instead outputting dynamic content with the aid of the *Twig* templating system, in which placeholders for values to be rendered are placed within HTML. A controller passes arguments to a view to be rendered against their corresponding placeholders. Zend can offer similar functionality by providing an interface for using the *Smarty* templating system. Any templating system however suffers from the problem of mark-up validity. A developer would be required to insert into their mark-up special tags or placeholders that are not valid XHTML. Consequently, these invalid tags would show up as syntax errors if using any form of automated mark-up validation – making actual syntax errors, such as mismatched close tags, harder to distinguish. Furthermore, a WYSIWYG HTML editor would provide little support for templating placeholders and most likely be rendered unusable.

A complex domain will need a complex object to represent it, with accessors and mutators flexible enough to allow bespoke logic or argument validation. With the exception of those in Symfony, a model maps database result sets onto dynamic domain objects which offer no encapsulation and cannot contain any user defined methods as they don't represent classes that literally exist.

Despite the separation of concerns offered by the MVC, there is little formalization for the location in which business or orchestration logic should be placed. As controllers in all three frameworks may

contain multiple actions for handling multiple types of request, they often also contain logic so that it may be shared and not duplicated. This however decreases controller cohesiveness and prevents the encapsulated logic from being reused by other controllers. Although current trends are moving away from "*fat controllers*" (Cooper, 2008), and are encouraging developers to place such logic within a model, none of the frameworks present any real standard for ensuring logic is encapsulated, cohesive and reusable.

2.4.2 Form Processing

Zend provides a set of classes for creating forms, each of which equates to an HTML form element. The controller instantiates the required classes, configures them by setting validation rules and renders to HTML for output in the view. The Cake approach is similar but instead generates form element HTML through method invocations within scriptlets, consequently increasing complexity and lacking the finesse of object orientation. Both approaches enable more complex programmatic manipulation than can be achieved purely with HTML. Symfony differs entirely by generating forms automatically based on domain objects and YAML configuration - an even less flexible approach. All three form creation approaches completely sacrifice mark-up flexibility as the rendered HTML is generated automatically. Although CSS can be applied to alter the appearance of a form, a complex design may need an explicit mark-up structure which would not be implementable when generating forms programmatically.

2.4.3 Ajax

Zend provides no support for Ajax. Any Ajax functionality will require a developer to implement any and all client side JavaScript themselves and create or alter any controllers to return the required response (a fragment of HTML, rather than a whole page) in the appropriate format (XML or JSON). Symfony and Cake offer a series of helpers for generating client side JavaScript within scriptlets, increasing their complexity further. On the server side, controllers must also be modified to return the correct fragment response. Ultimately this approach is very flexible although it has a very

manual setup, requires the view and controller to be written in a specific way, and also requires many values to be stringly typed which leaves code difficult to refactor.

2.4.4 URL Formatting and Generation

Zend and Cake both perform routing automatically, expecting a URL to conform to the name of the action method and the controller class in which it is defined. Any customized routing strategies must be declared programmatically at runtime. URLs for links or form submissions must be manually written in the view. Consequently any subsequent change to a controllers name or methods requires affected URL's to be located and updated individually. Symfony differs by requiring a simple name to be associated with the URL pattern and action pair. This name is specified in the view and a URL is generated automatically, allowing URL patterns and controller structures to be altered without affecting a view. Despite the advantages of the Symfony approach however, all three frameworks rely heavily on hard coded strings for all URLs, ultimately making code refactoring problematic.

2.4.5 Component Orientation

All three frameworks enable the rendering of composite web pages by providing, to a certain extent, a level of component orientation. Mark-up snippets can be extracted from views and placed into external view fragments. These fragments can be rendered by any view whenever it is needed. A problem arises however when the extracted fragment contains one or more scriptlets. To successfully output any dynamic content, the controller of any view implementing the fragment will need to set the properties the scriptlet within the fragment requires, resulting in duplicate code across controllers. Even if the duplicate controller code were extracted into a helper class to remove redundancy, a controller would still be required to implement it for the extracted fragment to work. Ultimately an extracted view fragment is still dependant on the controller. When a fragment is implemented there is no enforcement to ensure its dependency is met.

2.5 Apache Wicket

Many of the drawbacks I have exposed with existing PHP frameworks are not present in Wicket. Although Wicket is based on the MVC architecture, there are some fundamental differences.

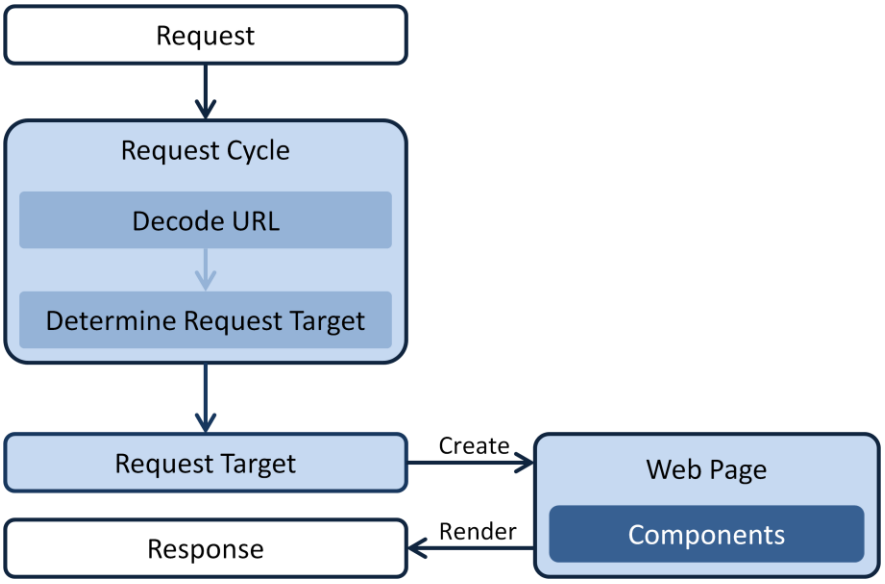

Figure 4 The Wicket request cycle **(Dashorst & Hillenius, 2009)**

Wicket is a Java framework for component orientated web development. The request cycle is a front controller; it establishes which request target a request should be mapped to. A request target is capable of dealing with and producing a response for a particular request. A simple request for a webpage will call the request target to locate a webpage class. A webpage is a composition of numerous components, each representing an HTML element, organised into a composition hierarchy, with the page as the ultimate parent. Once instantiated, a webpage is rendered, recursively rendering all child components, generating a response. Figure 4 illustrates the process.

Each component is made up of a view, containing purely HTML (Dashorst & Hillenius, 2009), and a controller, which programmatically manipulates the mark-up in the view to add dynamic content. A component may make use of a model, bridging it with a domain object. (Dashorst & Hillenius, 2009) Mark-up from the view is parsed into objects at runtime for programmatic manipulation by the controller; making scripts obsolete. As each component contains both a view and controller, which cannot be implemented separately, it is completely self contained and therefore fully reusable.

Components can be rendered as part of a complete page or as individuals, which simplifies server side generation of Ajax responses. The framework is stateful; consequently a webpage instance is stored for use by subsequent requests. An Ajax request for example would update a page instance and re-render the components that were altered. The framework is event driven; links, buttons and forms make use of callback methods to carry out an action in response to a click or form submit. Consequently all URL's are generated automatically to point a request to a listener capable of invoking the callback. Wicket is purely a framework for web components and does not include any tools for working with databases and web services, but does support full integration with Spring.

2.6 Spring Framework

Spring is a Java framework which aims to simplify EJB development. EJB's require interface implantations which can often be overly complex and ultimately unnecessary. (Walls, 2011) Spring returns EJBs back to their POJO roots and empowers them with other features such as dependency injection, enabling loosely coupled objects to collaborate. Additionally, aspect orientated concepts are applied to allow common functionality to be extracted and wrapped around the object that need it, facilitating full separation of concerns. (Walls, 2011) Utilisation of these concepts during development not only simplifies code and makes it more reusable but also reduces plumbing code, increases reusability and provides a simple method for implementing service orientated architecture. Plumbing code is reduced through the API which offers abstract functionality for database interaction, transaction control, web service requests and ORM. When a Spring application starts up, each POJO, for example a DAO, is instantiated automatically and stored in a container – the application context – which allows them to be retrieved by the classes that require them in a unified manner.

2.7 Conclusion

Each of the PHP frameworks I've looked at followed the MVC architecture, enabling separation of concerns, and the framework API simplifies common tasks. Whilst this may encourage code reuse and decrease bugs, thus reducing development time, this is achieved at the expense of flexibility. Although each framework offers various ways in which to customize and override the default functionality to suit the needs of a project, the more code that is written to customize the behaviour of a framework, the less beneficial the usage of a framework becomes. Additionally, if an application uses a framework for a purpose for which it was not originally intended, the amount of bugs, and therefore development and testing time, will increase. (Riehle, 2000)

The anything anywhere approach for the CMS requires a component orientated framework. Despite functionality in all three PHP frameworks to create composite views from mark-up fragments, as both views and fragments are dependent on the controller they are not independent and therefore not reusable components. It is for this reason that I have chosen not to use any existing PHP framework. Although it would be possible to implement the CMS in any of the PHP frameworks I've looked at, reuse of view fragments would be problematic and layout customization would be restricted due to hard coded HTML. Wicket offers a more suitable framework concept, by enabling direct programmatic manipulation of associated mark-up, and solving the view dependency on the controller by combining the two into a single unit.

The functionality offered by the Spring API is comparable to the functionality offered by the three PHP frameworks. Furthermore, although not provided out of the box, Symfony and Zend also offer dependency injection. These elements combined reduce plumbing code, and enable business and orchestration logic to be extracted from controllers whilst maintaining dependencies to any objects they require.

3. Content Management Systems

Within the context of a website, a content management system is a platform which offers authorised users access to create, manipulate and publish different types of content on their website. This is usually presented in the form of a web based portal. (Johnston, 2010) Content can consist of anything from text or images to videos or podcasts.

Typical functionality permits users to create and edit web pages, inserting both static content like text and images, but also dynamic content generated from other sources internal or external to the website, such as news feeds or calendars. Many offer functionality to permit any user to create content, allowing the website to become community driven through forums, wikis or comments sections.

A CMS will also offer design and layout customisation. Typically this is offered by providing built-in and downloadable themes and templates which can be applied to the website to alter its visual appearance. Templates consist of files containing HTML, CSS and a CMS specific template language.

This chapter will look at several popular CMSs, both free and paid for, summarising their overall functionality and unique features, and evaluating the way in which they offer layout and design customization.

In this chapter, user refers to an authorised user making use of a CMS to create a website, end user refers to a user visiting the website created with a CMS, and system refers to the CMS.

3.1 Interspire Website Publisher

Website Publisher is a CMS designed to make it easy for non-technical users to create fully functional websites. (BigCommerce Pty, 2012) It costs about £250 which includes a high level of service and support alongside the software. Website Publisher is written in PHP and has minimal

system requirements, making it usable on most systems. The CMS is entirely browser based and has a very simple interface, all elements of which have an associated tooltip, making it straightforward to learn without the need for specialist training or prior experience. Web Publisher is aimed mainly at businesses and although the target market for the new CMS is novice individual users, Web Publisher is the best CMS I am aware of and is included in this analysis for that reason.

3.1.1 Creating Content and Pages

Web Publisher enables the creation of content in many forms; it is in fact the responsibility of the user to choose the content type they wish to create, such as news reports, articles or blog posts. A content type consists of attributes such as title, body text and author. These attributes are used to generate a form for inserting new content of that type; culminating in the creation of separate web pages for each content entry for each type. Each content type attribute equates to a pre-determined form element for insertion, such as a text field, radio selection or WYSIWYG editor, and has built-in formatting used when displaying the content on a page. A user can define their own content types or alter those that are built-in, resulting in a very flexible and intuitive system, as it works on the basis that not every page on a website will be the same and will therefore need differing attributes – which is of course completely correct.

Web Publisher also includes several built-in pages: homepage, error page, search page and several others. None of these pages have a content type but provide either default content which cannot be edited, such as an error message, or no content at all. Each one of these pages may be customized to contain content from other pages, links to other content or many other options, achieved by inserting widgets with the layout editor.

As well as classification by content type and also by category, content may also be set as a child to another piece of content enabling three ways in which a structure for the website can be constructed. Structured content is crucial when designing a navigation system or syndicating content subsets.

3.1.2 Media Content

One form of permitted media within Web Publisher is images. These are uploaded using a Flash based interface which facilitates asynchronous uploads of multiple files with an upload progress tracker; the latter is not available with PHP alone. Other media such as video or audio is uploaded when required by a content type, for example the podcast content type attribute.

3.1.3 Design and Layout

The design of a page is generated from HTML and CSS template files. The layout however is highly customizable. Each section of the template is presented to the user as a panel into which widgets can be placed (Figure 5). The panels consist of the secondary header, footer and a central body panel which can be split into one, two or three columns. Whilst this may seem flexible, each panel is fixed and may not be moved or resized. The only way to achieve such customization would be to alter the HTML and CSS.

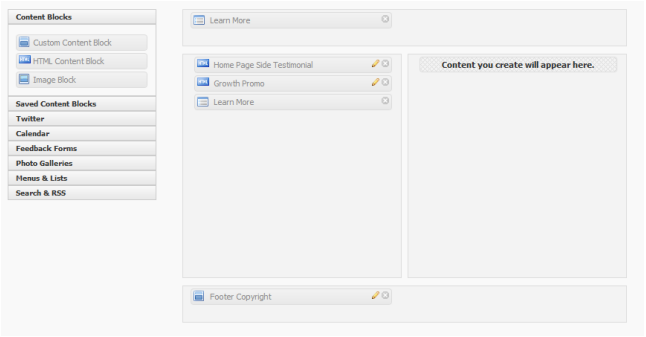

Figure 5 The Interspire Web Publisher layout editor interface

A default mandatory widget is automatically inserted into one of the columns; it acts as a placeholder for the content of a page. Widgets may be inserted above or below it or in neighbouring panels. There are numerous widgets available, ranging from simple text or graphics to dynamic menus, calendars and image slide shows; many of which have individual configuration options.

Layout customization is finely grained; there is a master layout which can be inherited or overridden on a panel by panel basis for each built-in page and content type. Additionally individual pages of each content type may also inherit or override from the content type layout.

3.2.4 Forms

Web Publisher includes a simple and intuitive drag and drop interface enabling basic form components to be placed, ordered and configured. A form submit action can be specified to determine where to store the submitted data and what to show the end user as confirmation of submission. The form may be used as a widget and placed into a layout or included in the body text via WYSIWYG editor insertion. The functionality to apply validation to a form field is not present. Whilst it is possible to mark a field as mandatory, no option can be chosen to indicate what type of data is acceptable, for example email address, phone number, etc.

3.2 Joomla!

Joomla! is a free, open source community driven CMS, built on its own framework of the same name, which grew out of the Mambo project in 2005. (Marriott & Waring, 2011) It offers a web based portal permitting the collaborative creation and sharing of content and is one of the most popular CMSs available. (Open Source Matters Inc, 2012) Joomla! is written in PHP and the name is a derivation of an African word meaning “all together”. (Marriott & Waring, 2011)

3.2.1 Creating Content and Pages

All Joomla! content takes the form of an article, consisting primarily of a title, category and article text. The article text is created with a WYSIWYG editor into which links to other articles or media may be added. There are many other additional attributes for visibility of page elements, authorisation settings and publishing options, permitting great flexibility but resulting in an overly complex and overwhelming interface which would deter novice users.

All organisation in Joomla! is done by assigning an article to a category. The category system is hierarchical so a structure for the website may be created relatively easily.

3.2.2 Media Content

Joomla! permits only one media file to be uploaded at a time and offers no indication of status during the upload. Once complete, uploaded images may be categorised and organised into folders which represent directly the underlying file system, although simplistically exposed through a browser based file manager interface.

3.2.3 Design and Layout

Joomla! layouts are based on HTML and CSS template files. A template contains one or more styles which can be applied to individual pages. This is very granular, but for sites with many pages, ultimately time consuming to implement. The design and layout may be manipulated through the web browser but only by directly modifying the HTML or CSS using browser based text editor (Figure 6). Essentially any non-technically minded user with no knowledge of HTML, CSS or cross browser quirks would be entirely unable to use anything other than a pre-defined template.

Figure 6 The Joomla browser based text editor

3.3 WordPress

WordPress is an open source, communality driven project, initially just a system for publishing a blog; it has since evolved into a full CMS. (Stern et al., 2010) WordPress appeared first in 2003, growing into a well-used platform that prides itself on its simplicity. (Automattic, 2012) WordPress is written in PHP.

3.3.1 Creating Content and Pages

Content in WordPress takes the form of either a page or a post. Each contains mostly the same attributes such as title and body text, but they are published in different ways. A page is static and must be linked to in order to be accessible to an end user. Posts are blog entries and are added to a

chosen blog and displayed automatically in an appropriate area of the website; by default the home page which shows the most recent posts.

3.3.2 Media Content

WordPress takes advantage of new features in HTML5, allowing media files such as images and video to be dragged and dropped straight into the browser. Flash, Silverlight and single upload fields provide suitable backwards computability. With the exception of the single upload field, all methods provide asynchronous uploads of multiple files with a progress bar. Once uploaded, images may be categorised, captioned and edited within the browser.

3.3.3 Design and Layout

WordPress utilizes HTML and CSS template files to generate a layout; however, some browser based customisation is supported. This is achieved by visually breaking the layout into several panels into which widgets can be placed (Figure 7). This is limited however as not every part of the layout is available as a panel. Only the side and footer may be customized in this way. There is little functionality to create a different layout for each page; only two variations may be created for application to either a page or a post.

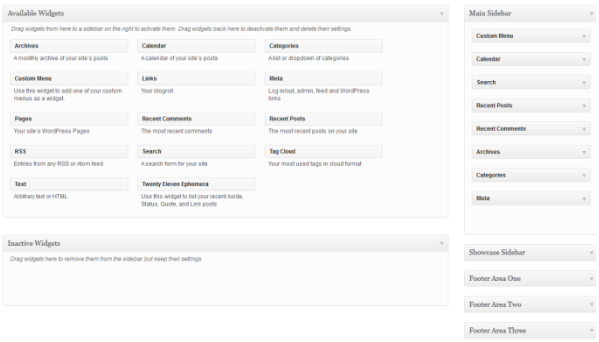

Figure 7 The WordPress layout editor interface

3.4 Analysis

I have used each of the three CMSs to produce the same website – a website for a fictitious design company. I have created four different pages, including a contact us page with a form and a portfolio page with an image slideshow. I have also attempted to customize the layout as much as possible.

Web Publisher is by far the most flexible for content creation through the use of user defined content types. Whilst Joomla! provides virtually the same attributes for every article as is available when defining a content type in Web Publisher, each must be individually disabled. Content types are initially confusing and increase initial setup time, however, they are ultimately more flexible when creating content for a website.

Despite the flexibility of content types, it is not possible to add the attributes required to model any domain. An attribute may be added to a content type only once. Using the example from chapter two, an album content type could not be created with separate attributes for artist, album title and release year. The artist and year would have to be inserted into a body text attribute leaving the album title as the title attribute. It is unreasonable to assume that a novice user would have any understanding of information architecture, but a simple domain such as an album has clearly identifiable attributes which even a novice user could determine. Without the functionality to insert custom attributes into a content type, a user is forced to compromise by placing multiple domain attributes into a single content type attribute. The resulting content will be less semantic and most likely cause issues in formatting consistency.

Each system provided hierarchical category classification for content; WordPress and Web Publisher additionally allowed content to be added as a child to other content. Whilst these aid in creating an organisational structure, all three systems lacked any useful visual representation of the hierarchy. Choosing a parent for an item of content from a drop down field on a form does little to give the user a mental image of the structure they are creating. With more content additions and an ever growing hierarchy, organisation would quickly become overwhelmingly complex and impossible to mentally visualise without any graphical aid.

Uploading large images may take some time; during the waiting period a user needs to be kept apprised of what is currently happening with some sort of status indication such as a progress bar. Without this, it is not possible to determine whether the system has crashed or is simply busy.

Joomla! provides no such functionality and impatient novice users would most likely assume the worst during long waiting periods and terminate the upload to try again.

By providing no browser based layout editors other than a text editor for the raw HTML and CSS, Joomla! requires that any user wishing to customize the visual appearance of their website has a working knowledge of HTML and CSS. Joomla! is a prime example of a CMS lacking layout manipulation functionality, prohibiting non-technical users from altering their sites design. WordPress and Web Publisher do provide a widget based layout editor which is better than nothing, but each fails for a number of reasons. Firstly, the panels into which widgets can be placed are fixed, they cannot be moved, resized or removed nor can new panels be added. Despite Web Publisher allowing a selection of one, two or three column panels the options are still finite and therefore restrictive. Secondly, not every part of the page is represented as a panel. WordPress only presents a side column and footer, Web Publisher is slightly better presenting all but the page header. This means that some parts of the layout may not be customized without altering the underlying template files. Thirdly, the interface used to perform the layout manipulation does not reflect the actual layout being manipulated. Panels are not to scale and, in the case of WordPress, are not even presented in the order they will appear on the resulting page. Additionally, each widget consists of only its title giving no indication as to what the contents of the widget will look like, or how it will fit together with other widgets. It is very difficult to produce a design when that which is being designed cannot be seen as it will actually appear.

Layout is only part of design; text and background colours, spacing, fonts and text decoration, to name but a few, also contribute to the overall appearance of a website. None of the systems I looked at provided the means to alter any such visual attributes beyond directly altering the CSS. Furthermore, even if a user were able to modify the CSS to alter the colour scheme or fonts, the changes would be applied to the entire site in all systems except Joomla! which is the only one that supports style assignment to an individual page. Whilst this may encourage consistency it does so at

the expense of flexibility. As all appearance related attributes are contained within a template, to alter only one of them for only one page would require the template to be duplicated and altered only slightly to provide the customization, leading to redundant and repeated code.

The sample website I attempted to create could not be finished in Joomla! or WordPress owing to a lack of functionality for forms and image galleries. By being open source however, they each have literally thousands of third party plug-ins available to extend the offered functionality. The form and image gallery features I sought after may not have been available out of the box, but will certainly be available to download and install onto the CMS. Despite this however, I believe that such functionality should be built-in as it is hardly bespoke or infrequently needed.

3.5 Conclusion

From the point of view of a non-technically minded user, each CMS is lacking functionality to allow visual aspects to be edited. Despite the widget based layout editors, there is only so much that can be customized without knowledge of HTML and CSS. Additionally, a user will want to see what they are designing – it seems almost ludicrous that neither of the layout editors I looked at provided such a basic requirement by being only text based.

The semantic web, otherwise known as Web 3.0, is an emerging web standard and none of the CMSs I looked at offered any such functionality. Web Publisher was the closest by allowing users to create their own content types rather than being fixed to a single block of text, but without custom attributes the content types cannot be used to represent any content.

4. Software Architecture

Component orientation and the model view controller were mentioned in chapter two. This chapter explores them in more depth and also looks at service orientated architecture (SOA) and aspect orientated programming (AOP) as I feel that these are worth incorporating into this project.

4.1 Model View Controller

First conceived in 1979 and first implemented in the Smalltalk language, the MVC is an architectural pattern designed to *"bridge the gap between the human user's mental model and the digital model that exists in the computer"*. (Reenskaug, 2011) A model represents knowledge, and can be comprised of one or more objects. A view is the presentation logic for a particular model. A view will query a model for data and display information, or send new information to update the model state. A controller deals with the interaction between the user and the computer; it processes input from a user by returning an appropriate view based on that input. Further output to the user is provided by attaching handlers to user actions such as key presses or mouse clicks. (Reenskaug, 1979) A tool comprises both a controller and a view (Figure 8). The controller within a tool will accept input and forward it onto the view. (Reenskaug, 2011)

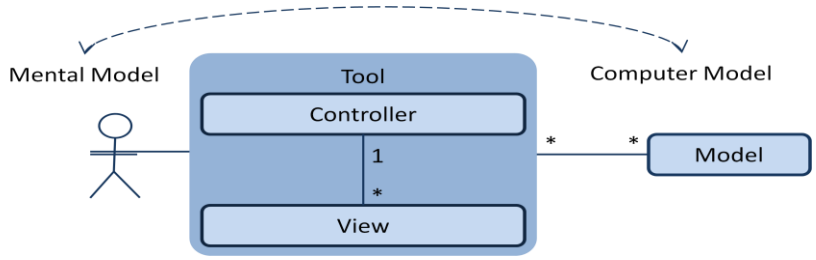

Figure 8 The Model View Controller

The MVC architecture facilitates separation of concerns within an application. By keeping business logic, input processing and presentation logic separate, the maintainability and extendibility is improved and complexity is reduced through decoupling. (Basham et al., 2008) For example, in a web application, the view would be an HTML page, the controller would handle HTTP requests and return an appropriate view, and the model would interact with a database and would contain

business logic. The HTML could be completely redesigned without affecting the controller or the model as they are separate from one another. The models could even be used by a desktop or mobile application with new controllers and views, without needing to be altered.

Each of the PHP frameworks I looked at implemented this pattern in essentially the same way. The implementation also incorporated a front controller. A front controller is a single intercept point for all HTTP requests. (Basham et al., 2008) The front controller analyses a request and forwards it onto an action controller, which is designed only to process that particular request. This frees individual controllers from the responsibility of request processing.

4.2 Service Oriented Architecture

Service orientated architecture is a paradigm which promotes the separation of concerns by specifying that the functionality of a system is decomposed into smaller stateless pieces, each encapsulating a particular concern or function. Functionality is encapsulated into a component, and may be utilised by other components via an interface which exposes the services the component provides. Services are designed so that the encapsulated logic is highly cohesive, promoting reuse and minimizing coupling. Services are interoperable and often collaborate with one another to perform a single task. Despite interoperability, services are loosely coupled, which frees them from constrictive dependencies and promotes independent growth and development.

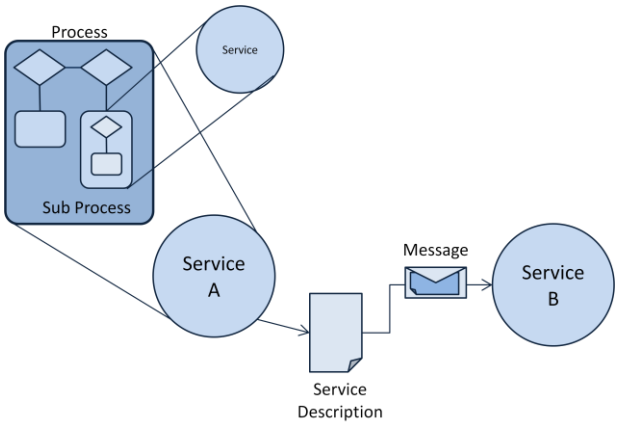

Figure 9 Service Orientated Architecture

Web services was the first technology that allowed SOA to be fully realized. (Erl, 2005) Web services offer a standardised communication of parameterised data between web connected systems, bridging the disparity between and within organisations.

(Erl, 2005) Services expose descriptions of themselves (service name, expected request and response data) so that other services are able to communicate with them; services communicate by sending messages which must be autonomous as once sent, a service loses all control over a message. The full process is shown in Figure 9.

Services are designed so that the encapsulated logic is highly cohesive, promoting reuse and minimizing coupling. The descriptor that a service exposes defines a function and communication contract which the service must adhere to, whilst also enabling service abstraction; whereby only that which is needed for invocation is exposed and all logic complexities are hidden. Services are autonomous and are entirely responsible for the logic they encapsulate. Services facilitate composition so that a collection of services may be used together to achieve a more complex task. Services should be discoverable by being outwardly descriptive enabling discovery, assessment and utilization.

The business processes, requirements, constraints and dependencies within an enterprise are ever changing (Erl, 2005); encapsulation into services improves logic maintainability. The internal logic of a service may be altered to suit new requirements without affecting any of the components using a service, as the interface remains unchanged. Furthermore, by encapsulating enterprise logic into a service, it may be shared across multiple applications, without concern for the platform they are deployed upon.

Services can be layered which enables separation and also exposes the dependencies between them. Typically there are three layers: the application service layer, which contains technology specific logic and usually consists of utility services offering functionality for user interaction and presentation adaptation; the business service layer, in which business logic and constraints are implemented in an application independent manner; and the orchestration service layer, in which the execution sequence is determined. (Erl, 2005)

4.3 Component Orientated Programming

Since the early days of programming, developers have been reusing software (Sametinger, 1997) which is beneficial for quality, cost and productivity. Stability, reliability and performance increase as error fixes and optimisation enhancements accumulate from reuse to reuse.

> *"Software reuse is the process of creating software systems from existing software rather building them from scratch" (Sametinger, 1997)*

Reuse is most commonly attained through function libraries which provide standardized libraries for common tasks such as file handling. Class libraries are the object orientated equivalent and present a greater level of abstraction with the additional flexibility of inheritance and polymorphism, however individual class reuse can be too finely grained for a large system. Frameworks offer reuse, not by providing individual classes, but through a composite set of classes which collaborate with one another.

These and several other forms of reuse, are however entirely informal. Component orientated programming formalises reuse, and presents clear criteria and methodologies for creating and using reusable software components.

> *"Reusable software components are self-contained, clearly identifiable artefacts that describe and /or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status."(Sametinger, 1997)*

Components must be independent and not rely on the implementation of other components for their usage. Components must be clearly separate artefacts and not spread across multiple locations which would make implementation difficult. A component should perform a clearly defined function and that function should be properly documented. The functionality on offer should be properly encapsulated but be sufficiently accessible via an interface. A component should declare its status, in terms of who created and maintains it and who should be contacted in the event of a problem.

Components are units of deployment and cannot be deployed individually but must be part of a larger composition (Szyperski, 1998), such as a framework. Wallace argues that because of this, there is no such thing as a software component. (Wallace, 2010) To a certain extent he is right; a component is a part of a piece of software, and cannot be used without composition with other components and a larger system to provide some means of running the composed application. This does not mean however than component orientation as a design philosophy is any less valid or useful.

4.4 Aspect Orientated Programming

Aspect orientated programming (AOP) aims to increase modularity and decrease code tangling by making functionality available across objects. (Gradecki & Lesiecki, 2003) Tangled code or code scattering is the name given when code to fulfil one system requirement is implemented across classes that are needed to implement another. For example, a product class within an e-commerce application is responsible for representing a particular product, one property would be price. A system requirement could call for any changes in price to be written to a log. Consequently the set price method must invoke the logger. This addition of extraneous functionality is called crosscutting. Authorisation, timing controls and data persistence functionally could also be added to product class to fulfil system requirements but reducing class cohesion. This may be an extreme example, but in object orientated programming it is sometimes necessary for an object to contain multiple concerns.

Tangled code makes it difficult for classes to be changed or reused, and also makes code difficult to trace. Refactoring is one possible solution to the problem but for a complex system, time and money constraints make this impractical. AOP offers an alternative, by separating out the aspects of a class that has been crosscut. For the example above, the logging aspect would be separated into another class; the required crosscutting would also be defined so that it may be invoked when needed, but without any need for the product class to make the invocation. Using the crosscutting definition, the two classes are merged by a process called weaving. This can take place at compile or runtime.

```
class Product
{
  private int price;

  public void setPrice(int price)
  {
    this.price = price;
  }
}
```

```
class Logger
{
  public static void log(String message)
  {
    loggerFile.write(message);
  }
}
```

```
Aspect ProductLogger
{
  pointcut logPrice(String price) :
  call(public void log(String)) &&
  args(price);

  before(String price) logPrice(price)
  {
     Logger.log("old price=" +
     joinPoint.getPrice());
  }
}
```

Figure 10 An AOP example demonstrating two separate classes and the aspect that defines their weaving

To permit the weaving, four elements must be defined. Firstly the *join point*, which defines the execution point within the code of an application where another concern is required. Secondly the *point cut*, which defines in what circumstance the *join point* is applicable, for example when a specific method is invoked. Thirdly the *advice*, which contains the additional behaviour to execute and determines the time when it is executed; for example before or after the *join point*. Finally the *aspect*, which is written like a class and contains the *point cut*, defined like a property, and the *advice* which is defined like a method.

5. Analysis

This chapter defines the requirements the new framework and CMS must fulfil, extrapolated from the analysis and research in chapters two, three and four. The requirements defined in this chapter have been used to form a test specification which is detailed in appendix four.

5.1 Language Choice

I have chosen PHP for this project. Although there are numerous web languages available, for me the choice came down to PHP or Java as these are the two in which I have the most confidence and experience. I chose PHP because of its ubiquitousness; to make the CMS available for the largest possible market I need something which can run on as many systems as possible. PHP is offered as standard by most web service providers (Golding, 2008) and is the most popular language on the web. (Chan et al., 2009)

5.2 The Framework

The framework to be developed will be conceptually based on the Wicket and Spring frameworks. The requirements stem from the need for a system in which the visual elements of a webpage are broken up into reusable software components; each providing suitable server side functionality to manipulate the component for presentation on the client side, and functionality to deal with events and data passed back by the client. These requirements have been extrapolated from the analysis in chapter two, architecture research in chapter four and my own knowledge of the Wicket and Spring APIs.

There are currently no PHP frameworks that come close to Wicket (Wel et al., 2011); there are however numerous posts on various community websites from developers searching for one. (Wel

et al., 2011) (Yinka et al., 2009) The new framework will be created for developers seeking to build an application with the Wicket ideology but in a PHP environment.

The framework will need to be written in PHP. The current version is 5.3.10; (The PHP Group, 2012) however versions 5.3 and onwards are still supported, thus the framework should be backwards compatible to version 5.3. Furthermore, although many modules are available for PHP, it cannot be assumed that a potential developer will have full access to the environment in which an application created with the framework is to be deployed; consequently the framework must not use any PHP module that is not enabled by default to maximise portability.

5.2.1 Functional Requirements

The framework will be a component orientated MVC

There will not be any distinct controllers or views, instead only a component comprised of a class and some associated mark-up. The class will provide the interface to programmatically manipulate the associated mark-up and will contain component specific functionality; a text field for example will require functionality to process and validate post data. Components must exist for each HTML tag that requires such specialist functionality; a generic component will exist for each of the others. The framework will enable the construction of a compositional hierarchy of components at runtime, reflecting directly the underlying HTML structure. The root node on the hierarchy will be a webpage component. Once a webpage component has been constructed, the framework will need to render it and its child nodes using the manipulated associated mark-up. The render must produce plain HTML which can then be written to the response. For components which rely on a source of information, for example a dynamic label or pre-populated form, a model object will be specified. The framework will offer a set of wrapping model classes capable of containing any domain object that may be utilised as the information source.

The framework will enforce mark-up validity and enable scriptlet-free manipulation by flagging mark-up elements

The associated mark-up of a component will be purely valid XHTML; the framework must validate mark-up and make it available at runtime as an object enabling its manipulation. The framework must provide a mechanism for mark-up tags to have a unique identifier attached to them, enabling them to be flagged for programmatic manipulation. The use of an identifier will simplify the process of locating the required tag within the mark-up object without the need for complex traversal. Any flagged mark-up element must have a corresponding object in the component hierarchy; this object will also have an identifier which will match that of identifier in the mark-up tag it corresponds to. The framework will need to analyse the mark-up object and the component hierarchy, and associate matching identifiers allowing a component to manipulate the mark-up tag it is associated with.

The framework will be event driven

The process of handling links, buttons and similar components which require some form of server side callback will be simplified by requiring that such components be associated with a callback function. The framework will generate a URL automatically and place it in an appropriate location within the associated mark-up of the component. The generated URL will contain the information necessary for the framework to retrieve the callback function in a subsequent HTTP request so that it may be executed.

The framework will be stateful

To fulfil the event driven requirement, the framework will need to be stateful in order to persist an associated callback function from one request to another. A callback function may direct the client to another page; however, it may simply alter the current page in some way, for example updating a page containing a form to show an error message. Conceptually it is not a new page that is required, simply a modified version of the previous page. The framework will need to persist the webpage

component and its hierarchy from one request to another so that it may be altered by the callback and re-rendered in its new state.

The framework will provide Ajax support

The new framework, through its use of callback functions, statefulness and component orientation will enable Ajax requests to be handled without any specialised implementation. The component for an Ajax link or button should, like a normal link or button, have an associated callback function. The callback will, as well as updating required components to the new state, specify which components are to be re-rendered on the client side. The framework will be responsible for ensuring that client side JavaScript resources are included on a page on which an Ajax component is present, that the rendered mark-up contains the required JavaScript function invocations on suitable DOM events and that the framework resolves the Ajax request to the correct callback function, re-renders the specified components and produces a response that can be processed and acted upon by the client side JavaScript.

The framework will offer a mechanism to restrict access to pages from non-authorised users

A web application, including the CMS, will require a user to authenticate themselves before accessing restricted pages. The framework will enable a developer to specify a set of webpage classes that require authorisation, and specify a callback function that is to be used to determine whether the current session is authorised. The framework will allow a developer to specify a further callback function that will be invoked by the framework if the session is not authorised. These callbacks will permit the developer to specify which pages require authorisation, what the authorisation criteria is and what action is to be taken in the event of unauthorised access, for example: redirecting to a login screen for authentication or displaying an error message if the session is authenticated but not authorised.

The framework will offer a single consistent method for database operations

In the same way as all of the other frameworks I looked at, the new framework will offer an API for generic database operations, query preparation and result setting mapping. This will be presented in a database system independent manner so that the framework and any application created with it are not bound to any particular DBMS. For the scope of this project, the framework shall support only MySQL.

The framework will enable service orientation

Using the ideas behind Spring, the framework will provide a means with which user classes can be marked for automatic instantiation. These instances, or resources, will need to be stored in a container which offers an interface to retrieve them. The framework will also provide a means for resources to declare a dependency to other resources and for that dependency to be satisfied through injection. Furthermore, any object should be able to declare dependencies and invoke the same injection process. These resources will enable developers to encapsulate logic without losing dependencies, whilst keeping its interface available to the rest of the application, and build a layered architecture as described in chapter four.

The framework will offer aspect orientation

In addition to the separation of concerns facilitated by the MVC architecture and the service orientation, the framework will provide runtime aspect weaving. The framework will offer a resource which will enable a developer to write an aspect, declaring point cuts and advice. The framework must use the defined aspect and invoke the advice at the instance defined by the point cut. Aspect orientation will prevent tangled code in user and extension class that wish to implement functionality such as logging and database transaction control.

5.2.2 Non Functional Requirements

The framework should be able to process a request and return a response within one second; any longer and a user will be left waiting for an unacceptable amount of time. The framework should be

able to support up to 200 simultaneous users with no significant drop in performance. It must be possible to create up to 1000 web pages using the framework, and the component composition hierarchy should be able to reach 100 levels deep without a performance impact. The total number of levels shall not be restricted by the framework, but instead conform only to restrictions determined by the parameters of the PHP runtime configuration.

The framework requirements are explicitly defined in more detail with priorities and dependencies in appendix two.

5.3 The Content Management System

The CMS to be developed will consist of an administrative web portal allowing a user to manage the content, layout and design of their website. Web pages viewable by an end user will be produced by the CMS from user generated content and layouts.

The CMS has two potential users: firstly, the administrator of the website (hereafter referred to as the user) who will use the portal to manage the website, secondly, the end user who visits the website the CMS generates, ideally unaware that a CMS has been used.

For several years, I have worked as a freelance website designer and developer, producing a variety of websites of varying sizes and subjects. For each site, I also supplied a CMS which allowed clients to update their sites with new content. During the development process and sometimes after completion, clients would make requests for elements of the design to be altered. They seemingly believe that moving a page element is as simple as picking it up and moving it – like a widget. The widget ideology is the way in which a non-technical user perceives a website and will form the basis of the new system.

The CMS must work within the same operational constraints as the framework. It will run in PHP 5.3 and above, and will not require any non default PHP modules. As the CMS will need to work within a web browser it must be cross-browser compatible. The CMS will work in the current version of all major browsers: Internet Explorer, Firefox, Safari, Chrome and Opera. The CMS must also be backwards compatible to Internet Explorer version 7, as unlike the other browsers, users are not forced to update. Using the database abstraction offered by the framework, the CMS will not require a specific DBMS; however, for the scope of this project, MySQL will be the only one supported.

5.3.1 Functional Requirements

The CMS will allow the creation of a layout structure

Using nothing more than their web browser, the CMS will allow a user to create a layout structure. This structure will comprise various panel types: rows which stretch to the full width of the page and are vertically resizable, columns which may be stacked up side-by-side and are vertically and horizontally resizable, and floating blocks which are also vertically and horizontally resizable but may also be positioned anywhere over other panels. Rows and columns will automatically align with one another and may be positioned only through reordering. Both the layout as a whole and each panel will also need to have an attribute editor in which its background, border and margin can be defined.

The CMS will allow an arrangement to be created for a layout structure

Once a layout structure has been defined, the CMS will offer an interface with which to position widgets within each of layout panels – the arrangement. Each layout, of which there can be many, may have multiple arrangements created for it. The layout structure, arrangement interface and widgets will be displayed exactly as they will appear to an end user, giving the user an exact picture of the design they are creating. Each arrangement must mandatorily have one content widget present; this is a placeholder and will be used to display page specific content.

The CMS will offer a catalogue of categorised, configurable widgets

Widgets that can be placed onto a layout will be organised into categories. When placed into a layout panel, a widget can be configured with widget specific options. Within the scope of this project widgets will be available for: rich text, images, headings, forms and navigation menus.

The CMS will allow a visual theme to be created

A theme will consist of styling and formatting options: font size, weight and colour, margin, padding, and borders for individual elements such as headings, form fields and links. A theme should directly relate to HTML tags and their applicable CSS attributes, although such complexity should be hidden from the user by the interface through use of suitable form elements.

The CMS will enable creation of semantic content through user defined content types

To enable the creation of semantic content, a user should be able to define a content type. A content type will consist of one or more attributes. An attribute will be comparable to a database data type, and each will require a corresponding widget capable of displaying the attribute contents. Each attribute will be named by the user. When creating content, the user must select a content type; the CMS will then display a suitable form with suitable validation to allow the insertion of data. Each content type must have an arrangement and a theme associated with it; both of which will used in the generation of web pages that display the content. Within the scope of this project, content attributes will consist of: text, headings, author, date and image.

The CMS will break content up into pages and posts

Content should be created as either a page or a post. Content types will need to be defined for use by one or the other. When creating a page or a post, a user will be required to select the content type they wish to use. A page is perceived to be a standalone webpage and can be organised into a hierarchy with other pages to create an organisational structure for the website. The page hierarchy must be presented graphically, giving the user a clear representation of the structure they are creating. A page may optionally have an arrangement or theme applied to it, overriding that of its

content type. Although permitting arrangement or theme selection with this granularity may produce an inconsistent layout from one page to the next, I believe such a design decision should be that of the user and not the system. A post is perceived to be a single entry of a particular type, for example a blog post. Posts may only inherit the arrangement and theme of the content type, keeping each entry visually consistent.

The CMS will provide a facility to build and configure forms

A user should be able to create a form for an end user to submit. The user must be able to add any type of form field and, where necessary, define applicable validation options or define a list of choices for the end user to select from. The submit action that is to be performed must also be definable, either displaying a message or directing to another page. A form widget will be added to an arrangement in the location the form is to be shown. When used by an end user, the form submit action will be carried out and the submitted data stored so that a user may view the submission within the portal.

The CMS will require a user to authenticate themselves before performing administration

A user must login before accessing the administration portal. This should be achieved through a simple username and password authentication. The CMS will provide the facility for a logged-in user to create and modify other users and also modify their own password.

The CMS will generate web pages using the defined layout, arrangement, theme and content

When an end user visits the website produced with the CMS, the system shall render the requested webpage using the layout structure, arrangement, widget configuration and theme. The data for each content type attribute will be rendered using the attributes associated widget within the content widget of the arrangement. The CMS will require that the user defines a page as the homepage, which will be the default page shown to an end user unless a specific page is requested.

5.3.2 Non Functional Requirements

The functionality of the CMS portal will be organised into a hierarchical navigation structure that will not exceed four levels deep; any more would cause the user interface to become overly complex and require an excessive numbers of clicks to navigate to a particular page.

As with the framework, it is expected the CMS will take no longer than one second to produce a response, and be capable of handling up to 200 simultaneous requests without a drop in performance.

Use cases for the CMS are detailed in appendix three.

5.4 Evaluation

I chose to model the new framework on Wicket because it enables both mark-up and the code that drives it to be encapsulated into a reusable component, which is perfectly suited for the widget ideology behind the CMS. My existing familiarity with the API, as well as some further research into its documentation, enabled me to develop an understanding of how the framework operated beyond that of a developer merely using its functionality, consequently enabling me to essentially reverse engineer requirements.

The study into existing systems and their documentation produces an insight into their inadequacies, and is one of the most valued sources of requirements. (Maciaszek, 2001) To a certain extent, my past experience as a designer and developer has been an informal form of observation and interview. Although whilst working with clients I wasn't formulating requirements, the experience produced an insight into what users desire from a CMS and the way they expect it work. Questionnaires could have been used for requirement elicitation; however, without the ability to target specific users, the only means of distributing a questionnaire would be through online

communities this random, non-targeted approach would likely produce inconsistent or biased results.

There are several more contemporary techniques for requirements capture such as prototyping, joint application development and rapid application development. These processes are however only useful for projects with stakeholders, customers and development teams, none of which are applicable to this project.

6. Design

The design of the CMS and framework will be completed separately. Although the former relies on the latter, the framework should be usable in more than the just the CMS application and should be an independent entity. The framework will be called Picon (PHP Intelligent Component Orientation) and the CMS will be called Podium.

6.1 The Component and Service Orientated MVC

The MVC architecture requires alterations to achieve component orientation; the view and controller are to be combined. A view is no longer a whole webpage, but merely a fragment of one; a controller is significantly smaller, containing only the logic specifically required for its associated view. This also means that controllers are no longer responsible for multiple views, as shown in Figure 11. The controller and view are now to be considered fragments of their original selves, they are strongly coupled, although no more so than their original relationship, but this is no longer of concern, as they are a single unit – the component. To perform the same function as the original MVC, multiple components will come together into a composition. This plug-and-play approach enables a far greater reusability of views and controllers without any concern for meeting their now non-existent dependencies.

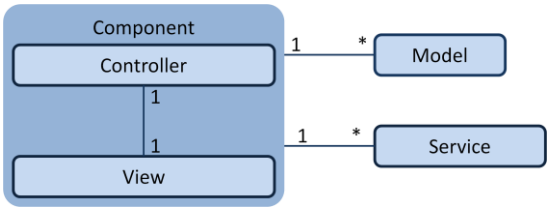

Figure 11 The component and service orientated model view controller

As it has now been removed from the controller, the interaction between the user and system, combined with the new requirement to produce a component composition, should be fulfilled by a front controller which acts as the interface to the underlying components.

Service orientation formalises the encapsulation of abstract application logic, business logic and orchestration logic, into separate service components which may be shared across controller/view components. This prohibits "*fat controllers*" and minimises logic duplication and coupling. Furthermore, logic for retrieval, persistence and interaction with other systems may be removed from the model and encapsulated into service components in the same way. A model may now represent only a simple autonomous domain object, whose usage is required by the view/controller component and whose persistence and business logic is governed by a service component called upon by the component controller. Although described in the context of a web application, this altered architecture could be applied to any application.

6.2 Picon Framework

Although based on Wicket, the new framework cannot be a direct clone as this would most likely fail. Firstly, a clone would be almost cheating and not make for an interesting project. Secondly, Java web applications differ greatly from PHP ones. Java applications start up only once and run continuously on the web server waiting for requests to process. In contrast, a new instance of a PHP application is created for every request. Another Java framework, Struts, has been cloned in PHP but has failed due to the length of time it takes to start up, producing slow response times. (Allen & Lo, 2007) Thirdly, the object orientated functionality is less advanced in PHP than in Java – to replicate a Java framework directly would be virtually impossible. I want the Picon framework to be written using the concepts of Wicket but with the strengths of PHP, and not attempt to emulate what can be done in Java.

In this section, user refers to a developer using the framework and end user refers to a user using an application created with the framework.

6.2.1 Overview

Picon will consist of a single application class which acts as a developer's entry point to the functionality of the framework. The developer will instantiate the class which will run the application process.

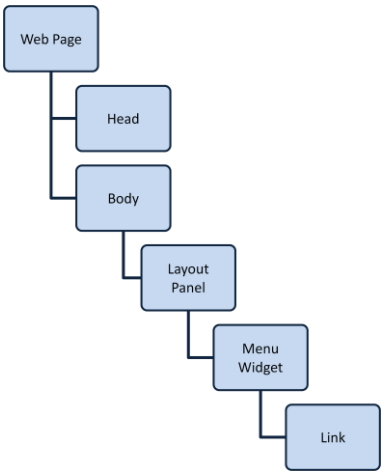

Figure 12 The component composition hierarchy

The application will use a configuration file. The best format is XML, as it can be automatically parsed by PHP into an object at runtime and enables a greater structural complexity than with name value pairs.

Inspired by a similar concept in Java web applications and Spring, the resources described in chapter five will be stored in a container to be known as the context. The configuration object and the context will be universally accessible through the application, allowing framework, user and extension classes to make use of them.

The composition of visual (view/controller) components will be hierarchical (Figure 12). The developer will be responsible for producing this hierarchy, by writing extension classes whose implementation includes the instantiation of components and insertion into the hierarchy.

Framework classes will be separated from user and extension classes by placing each into separate web protected directories. Protection will be achieved with *htaccess*. Another more ideal solution would be to enforce the placement of classes into directories not accessible by the web server. This may not be implementable however, as a user may lack the required level of access to the web server. The framework shall discover the class directories through the value of a constant, so if the user does have the ability to implement the second solution, the updated directory paths need only be altered in one place.

Full class diagrams and framework processes are detailed in appendix five. The design patterns referenced in this chapter are detailed in appendix eight.

6.2.2 Application Lifecycle

The application process will always begin with the receipt of an HTTP request (Figure 13).

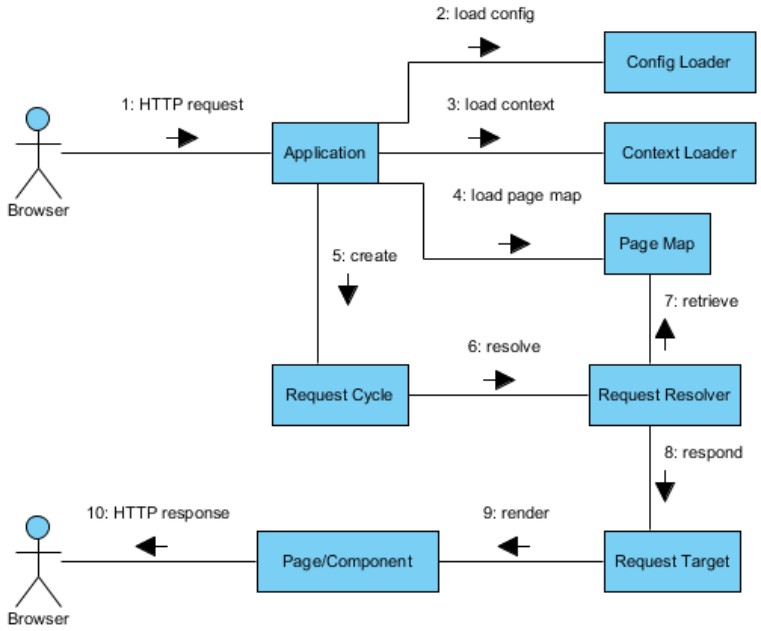

Figure 13 Communication diagram showing the Picon application lifecycle

1. The application is started by the web server to handle an HTTP request
2. The configuration file is loaded, validated and parsed into an object
3. The context loader is invoked to locate, instantiate and inject resources. Injected resources are to be placed into the context container
4. The page map locates and stores all webpage extension classes
5. The request cycle is created to process the request
6. The request cycle resolves the request to a target by analysing the URI
7. The resolver will locate a webpage class in the page map to that corresponds to the request
8. The request target is used to actually respond to the request. It will have been provided with the information it needs to do so by the resolver
9. The webpage object will be rendered producing HTML
10. The HTML is written to the HTTP response

6.2 3 Request Processing

The request cycle is the front controller and processes HTTP requests, producing a response. The request cycle constructs request and response objects. Although PHP provides super globals for accessing request properties and functions for working with the response, encapsulating them elicits a single point of access, permits request attribute immutability and also enables more complex methods to be defined for request and response handling.

The request cycle begins by analysing the request with a series of request resolvers. Using the chain of responsibility pattern, the request is passed from one resolver to the next until a resolver capable of handling the request is found. The found resolve will create a request target which is returned to the request cycle and placed into an array. If a single resolver class were used over a chain of multiple resolver classes, it would contain large amounts of complex, non-cohesive logic which would be difficult to trace and maintain. The chain also enables new resolvers to be added without disrupting the others.

The request cycle iterates through the request target array (the stack). Although initially containing only one request target, during the request target processing, one or more new targets may be added to the stack. This means that a request target need not actually produce a response, but may complete some processing and add to the stack a new request target that will. This approach allows for an internal redirection and facilities better request target cohesion and reuse. After every request target in the stack has been processed, the response object is outputted and the application terminates.

6.2.4 MVC System

The controller within a component consists of a PHP class, the view consists of HTML. The root component will always be a webpage component; it will be abstract and must be extended by the user to create a particular page. This webpage extension will construct the component hierarchy to match its associated mark-up using other components within the framework, or for bespoke

requirements, others defined by the user. An associated HTML file is to be placed in the same directory as the file in which the component class it is associated with is located. Both files must have the same name, although they will have different extensions. The class is expected to have the same name also, following a typical PHP convention. Unlike many others, this naming and file location convention is not in any way restrictive, and is a far simpler implementation than requiring users to specify HTML file locations with configuration or abstract method implementation. It also makes sense, as they are considered the same entity, to keep the class and its HTML together.

Components, and thus their associated mark-up, instance data, and any attached callback functions require persistence from one request to another to enable statefulness and callback execution. These objects are all generated to handle a specific request from a specific end user, and are only applicable to that end user in both context and security. PHP contains built-in serialisation functionality that enables objects to be converted to strings for persistence however, there are numerous places where those strings could be placed. The session provides a suitable storage mechanism; it is capable of storing a reasonably large amount of data and is specific to an individual end user. The session also has the lowest setup requirements; file based storage would require the user to setup specific directory permissions; database storage would force the user to provide the framework with a database connection and setup a specific table structure.

The altered MVC architecture stipulates a model is only to be a domain object, and that all former model functionality for data persistence and retrieval is to be relegated to a service component. With this in mind, it is reasonable to assume that this model object has the potential to be freely passed and stored by multiple components or any other object, which is likely to cause issues when serialised. Storing an object in multiple places would, before serialisation, consist of multiple references to the same object, but after serialisation, potentially consist of multiple references to multiple objects resulting in duplication and synchronicity problems. This issue depends entirely on the implementation of the serialisation process but the potential remains. Consequently, a model

within the framework will not be a domain object, but will act as a proxy to one. This proxy eliminates the need to actually store the model object in multiple places but instead provides an interface to access a single, centrally stored domain object.

Associated Mark-up

Any associated HTML file requires some special mark-up tags to enable the insertion of a tag identifier, and also isolate the mark-up that is to be used for a component from the mandatory tags required to produce a valid HTML document, thereby preventing duplicate tags on the output of component hierarchies made up of multiple HTML files. These additional tags will require the specification of a customised DTD which will define them. To separate them from other HTML tags a namespace prefix will be used. This is the only method that will allow new, non HTML tags and attributes to be specified, whilst keeping the HTML document valid.

Mark-up Parsing

The associated mark-up file for a component class will be parsed into an object using an XML parser built into PHP, of which there are several. *SimpleXML* and *DOMDocument* are unsuitable as they only process into dynamic objects. The framework should be able to specify its own domain objects to represent the mark-up enabling for method definitions. *XMLParser* enables the attachment of callbacks to the parsing process; consequently the resultant object is created by the callbacks and allows for a custom domain object to be used. Furthermore, although *DOMDocument* is more suited to HTML parsing, it is not up-to-date with XHTML and therefore does not handle XML namespaces.

Component Behaviours

The functionality of a component can be specialised for bespoke requirements through sub classing. This is however not necessarily an ideal solution, as bespoke functionality may need to be added and removed at runtime. Furthermore, the class hierarchy may not permit the desired polymorphism. The decorator pattern presents the solution and, although not an original requirement, will form the basis of a component behaviour – encapsulated functionality that may be applied and removed from

a component. Unlike a normal decorator however, it will not wrap the component but work like an observer. The component will notify its behaviours at various stages of its rendering process, so that the behaviour may render mark-up of its own, thus it is the outputted mark-up that is decorated. By not following the wrapping approach, behaviours may be applied and removed from a component in any order, without the need to alter the component hierarchy.

6.2.5 Ajax

Client side JavaScript to send Ajax requests and process the response will be created using *jQuery* which will handle most of the complexity itself, reducing the amount of code that needs to be written or tested. Although libraries other than *jQuery* are available, *YUI* for example, *jQuery* is, in my opinion, the simplest to use and is also one I have worked with before.

The Ajax response generated by the framework will be *JSON*. Although Ajax responses are traditionally *XML*, *JSON* is a JavaScript native format and its usage will reduce client side response processing, thus improving efficiency. *JSON* is also a more lightweight format and *JSON* response sizes will be smaller than the equivalent in XML.

6.2.6 Annotations

Resource declaration and dependency will be done through the use of class metadata, which is best implemented with annotations. Declarations could be alternately achieved with the configuration file, make use of tagging interfaces or naming conventions. Annotations are however a far cleaner and simpler implantation. (Davis, 2012) Annotations are applicable at class, method and property level, and therefore enable a class to declare itself as a resource, and a class property to declare itself as requiring a dependency to be injected into itself. Furthermore, annotations permit the usage of arguments, enabling customisation and declaration to be achieved with a single process.

6.2.7 Database API

To facilitate a database system independent API, the database interface offered to a user will be a facade for several layers of abstraction. Using the data access pattern as a basis, and working with the pretence that a data access object (DAO) will be a resource, the framework will offer a DAO support class, extension classes of which will form DAOs. The support will provide access to a template class, which offers an interface for preparing and running SQL statements. The template will internally use a database driver to actually run the query and process results. The driver is conceptually based on *JDBC*. These abstraction layers are similar to *Spring* and *Symfony* and enable the process of query preparation and result set mapping to be separated from the physical database operations. Each layer will be aware only of the interface offered by the other; this design by contract approach is the only solution that enables a database driver to be swapped with another, without affecting the rest of the system.

Mapping of result sets onto domain objects will be orchestrated by the template, but be specified by the user. Whilst this is perhaps less convenient than the ORM available in the PHP framework I've looked at, the approach facilitates full user control and customisation over mapping and domain object instantiation and building, and also removes the need to create any configuration or follow stringent conventions.

6.2.8 Aspect Orientation

There are numerous implementations of AOP for PHP already, each of which requires a pre-processor. This adds an additional stage to development, and is a form of compilation which deprecates the benefit of PHP being an interpreted language. (Jonnay, 2006) The pre-processing alters a developer's code, adding in the advice invocations in all of the circumstances they are needed, meaning that the developed code is different to the deployed code – and the deployed code has been tangled. Although there are alternatives such as integrating the pre-processor into the IDE or web server, the result is the same.

PHP is an interpreted language so weaving should only take place at runtime. This is achievable through the use of the factory and proxy patterns: a factory to instantiate an object, calling any constructor related advice, and a proxy to wrap around that object, intercepting method calls and invoking advice as necessary. The only disadvantage to this solution is the replacement of the new keyword with an invocation of the factory although, compared to a pre-processor, this method is far simpler, more suited to PHP, and does not attempt to recreate the compile time weaving found in Java.

6.3 Podium CMS

In this section, the user is defined as an administrator using the portal of the CMS, and the end user is a non-administrator who visits the website created with the CMS.

6.3.1 Overview

The CMS will be developed using the Picon framework. Much of its complexity takes places on the client side, particular the layout and arrangement editors. The server side functionality is relatively simple in comparison and primarily deals only with data persistence. Business types, component architectures, and full database and interface designs are presented in appendix six.

6.3.2 System Architecture

The server side functionality of the system will be broken up into layers, following the three tier architecture. The presentation layer consists of all of the webpage extension classes, behaviours, and any other specialised component extension classes such as panels. The service layer consists of all of the encapsulated

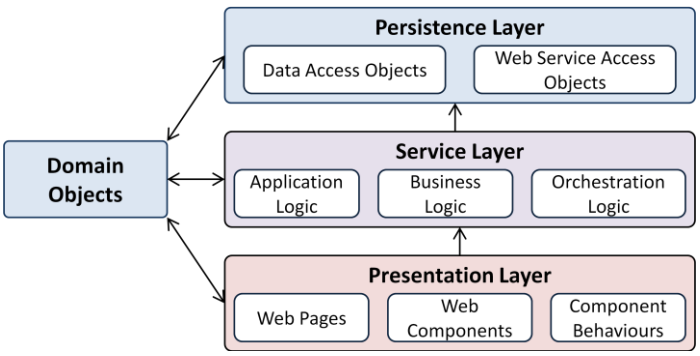

Figure 14 The architecture of the CMS

logic for business constraints and orchestration. Each service class will be declared as a resource for automatic instantiation. The persistence layer consists of data access objects which will be extension classes of *DAOSupport* and will also be declared as resources. Each will run queries and define row mappers. Although out of the scope of this project, web service access objects would exist within the persistence layer, and permit the CMS to be interoperable with external systems and services. Domain objects will be used as the format for messages sent between layers, and also form the model objects for visual components in the presentation layer.

The CMS deals primarily with CRUD operations; despite a large number of processes defined by use cases, ultimately most of them are suitable for implementation as business services.

Extracted from the business type model (Figure 57, appending six), the business services shown in Table 1 enable CRUD operations. Many of the business types have been combined to produce coarsely grained interfaces. These combinations have been completed with cohesion in mind; for example, the layout type is composed of one or more layout blocks. They are never used separately and so they may be combined. Business services are identifiable by the *Mgt* suffix.

Use cases have been used to determine the processes that the system must perform; these processes form the system services and are also shown in Table 1. Figure 58 illustrates these services in more detail, including anticipated methods.

Table 1 Table showing the business and system services for the CMS

System Services	UserLogin, GeneratePage, GenerateArrangement, GenerateWidget, GenerateForm,
Business Services	ContentTypeMgt, UserMgt, ThemeMgt, PageMgt, PostMgt, LayoutMgt, FormMgt, WidgetMgt, SubmissionMgt, ArrangementMgt

The system services depend upon one or more of the business services to perform their processes. The process of producing elements such as the page or the arrangement is relatively complex, as it involves the aggregation of a vast amount of data; for example, a page requires an arrangement, theme and a content type. The four system services that deal with generation have been defined

because I consider these processes, defined by the use cases, to be more than just simple retrieval operations. They use business services to aggregate the data that is needed and then prepare it for presentation.

The user login system service, whilst not a defined use case, is a pre-condition for many of the use cases. The architecture for all of the service components, illustrating dependencies, is shown is Figure 15.

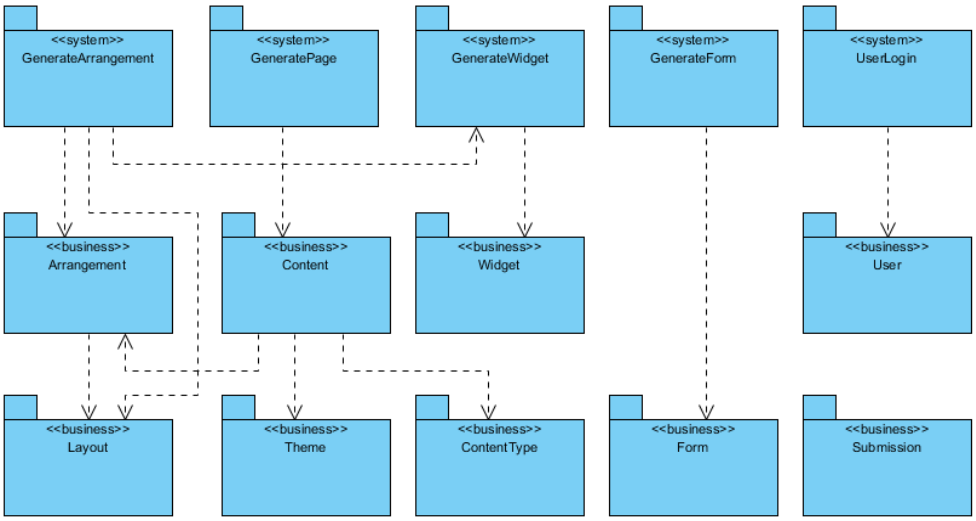

Figure 15 Package diagram showing the CMS service component architecture

Submissions have been purposely left independent of the form. Although the submission represents the submission data that was generated by a form, a form is dynamic and can be changed after a submission was made. Consequently, the submission should consist only of name value pairs corresponding to the form field name and the value the end user entered when submitting. This way, there is no synchronicity issue when altering a form.

Pages and posts have been combined into a single content package, to do otherwise would lead to duplication. The content package is the single dependency of the generate page package, which is the system service responsible for the generation of the physical web pages that an end user will see.

The business and system services form the service layer. Each of the business services will make use of its own DAO, with the exception of pages and posts. These will also use an additional content DAO, which abstracts out the database interaction on the content entity they both share.

6.3.3 Web pages and Components

Webpage extension classes will be created for each of the pages the CMS requires. A panel extension class will be created for each widget. The panel component was created with widgets expressly in mind. Each widget will also require a separate panel containing a form for all of the configuration options. The database will store the names of the panel extension classes, which are to be included in the widget domain object. A factory method is the best pattern for instantiating these panels, and will work through the generate widget service. It will require a widget domain object as an argument, and will extract and instantiate the appropriate panel class.

A layout block will also require a panel extension class. The panel will contain a repeater which will iterate over all of the widgets within a layout block. These layout block panels will themselves be used as elements within a repeater, enabling the construction of a completed arrangement.

6.3.4 User Interface

The portal user interface is fluid, enabling it to use as much of the available space as possible. Page layouts will be broken up into columns, which are detailed in Figure 16.

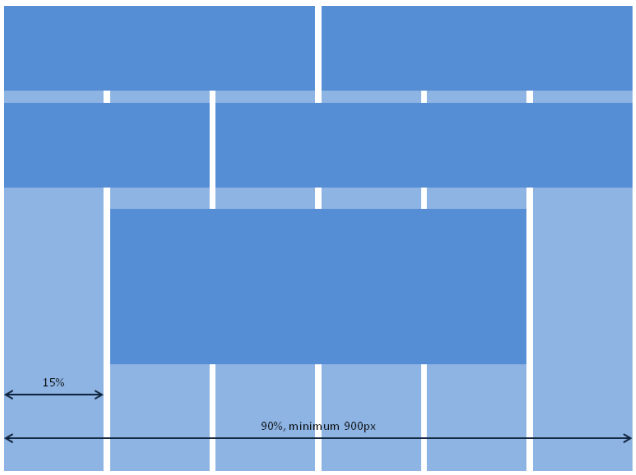

Figure 16 Grid showing the columns of the design

Navigation and Headings

Once logged in to the administrative portal, the main navigation menu along with the current page heading will be present at the top of the page, and on every page of the portal, Figure 17. The upper bar never changes and lists links for all of the main portal sections: content, layout, forms and users. The current section is indicated by a green background which joins up with the section specific sub menu beneath it. The identicalness of the colours clearly indicates the join between the sub menu and the main menu item it corresponds to. For further visibility, the title of the current page is displayed immediately below the navigation.

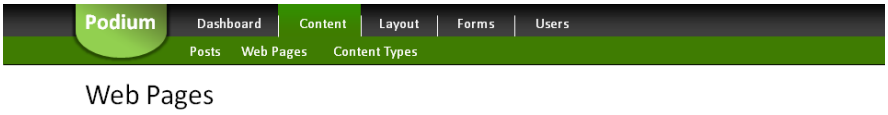

Figure 17 The main navigation menu and heading

In addition to the main and secondary navigation, it is also necessary to further break up pages offering more complex functionality, whilst not separating it onto different pages. A tabbed interface facilities this, Figure 18. Without changing to a new page, a user may see only the functionality applicable to the currently selected tab. The current tab heading is highlighted in a separate, lighter colour and does not include a border, visually linking it with the content it equates to.

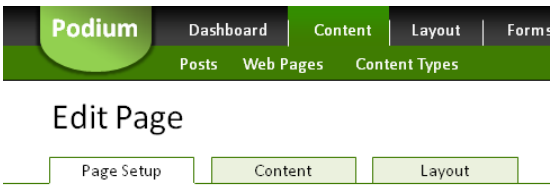

Figure 18 The tab interface

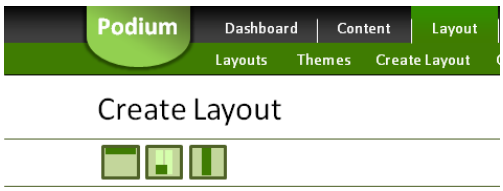

Figure 19 The toolbar interface

Many of the more complex pages are not suitable for tabbed layouts; whilst they may include a significant amount of functionality it cannot be broken up. The layout and arrangement editors are good examples of this. In these cases, options may be broken up into a toolbar, Figure 19. Each element is a button performing a different function. Although the icon portrays, as much as possible, what that function is, a tool tip is essential here to fully convey it.

Tables and Lists

The CMS will present any tabular data using the data table component, Figure 20. Rows and headings are visually separated through the use of colour. This is however only applicable for data that can be represented in this way; a hierarchical set of tabular data will use the list view component and distinguish hierarchal levels with indentation, Figure 21. This will follow the same colour separation rules to keep both elements constant.

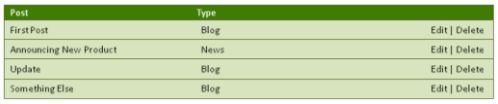

Figure 20 An example data table

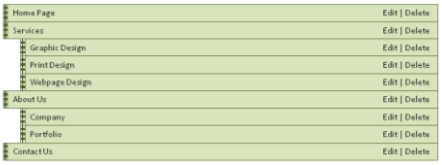

Figure 21 An example hierarchical list view of tabular data

Links and Buttons

A more clear indication, than available by default, of the role of a button on a form will be conveyed through colour, Figure 22. When more than one button is placed side by side, often text alone is not significant enough to distinguish function. Green will be used to indicate positive reinforcement, for example: save, continue, create, etc. Grey will be used for negative reinforcement, for example: delete, cancel, etc.

Figure 22 Form buttons

Links will be presented in bold to distinguish them from standard text; they will not have any text decoration (Edit and Delete links are shown in Figure 20 and Figure 21). State visibility will be provided through the addition of an underline on mouse over. Furthermore, buttons will lighten slightly on mouse over, as will the background of main menu items.

Forms

Form elements will maintain consistency with very little custom styling applied, so as to remain as close as possible with form elements a user will have seen on other applications – web or otherwise, Figure 23. Every form element must be clearly labelled to indicate its function, and aligned on the grid so that a clear association between the label and its field can be made.

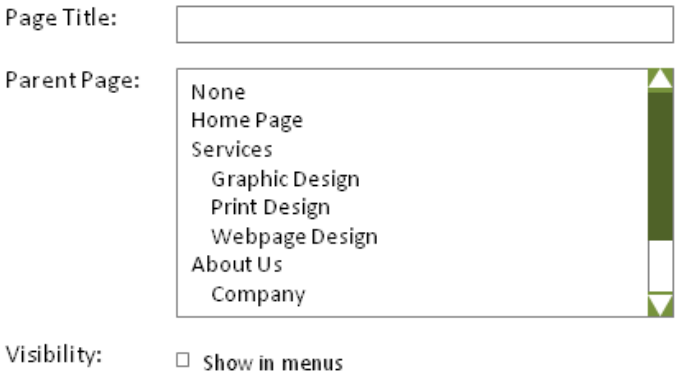

Figure 23 Example form components

Modal Windows

The layout and arrangement editors both require configuration panels; for example, the layout block attribute editor and the widget configuration editor. Navigating to another page is somewhat unnecessary and inconvenient as it would contextually break the user's workflow. A modal window offers a non-disruptive solution to presenting configuration options over the top of the existing page, Figure 24. Furthermore, on the form editor, in which configuration is mandatory, the modal window may appear upon inserting a new field, preventing the insertion of others until the first has been configured.

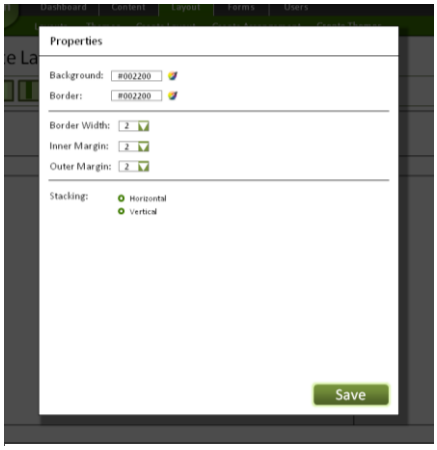

Figure 24 A sample modal window

Feedback Messages

Feedback given by a component in response to a change in its state is often not enough to adequately inform a user that their changes have had any effect, particularly a persisted effect. Using the feedback panel component, messages will be displayed to a user in response to any form submissions or a link click that results in a permanent change, for example deletion. These messages need to be fairly prominent, thus they are highlighted in a stronger colour and are also accompanied by a clear icon indicating the type of message: a positive message such as confirmation of success or a negative message enforcing constraints, Figure 25.

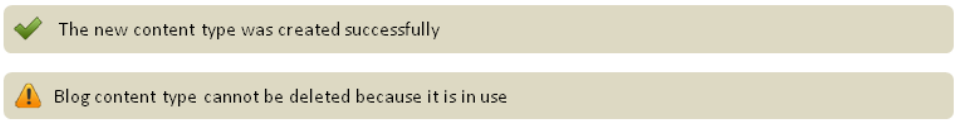

Figure 25 Example feedback messages

Browser Based Interactivity

One of the widgets enables the insertion of rich text. The ideology behind the CMS does not require a user to know HTML, thus HTML or any other special mark-up (such as BB code or wiki mark-up) is not an acceptable input. *TinyMCE* is a browser based, JavaScript powered *WYSIWYG* HTML editor. It offers an interface comparable with Microsoft Word and enables text to be formatted visually, Figure 26.

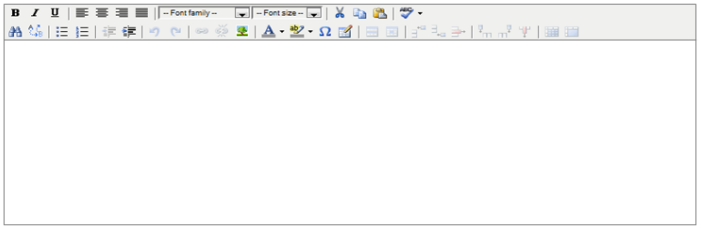

Figure 26 The tinyMCE WYSIWYG HTML editor

The positioning of layout blocks on the layout editor and the arranging of widgets on the arrangement editor requires the use of drag and drop. Layout blocks must also be resizable. *JQuery* and *jQuery UI* offer both drag and drop, sorting, and resizing functionality with minimal implementation requirements, and an API with numerous customisation options. Whilst other similar libraries exist, such as *YUI*, I'm choosing *jQuery* as I have more experience using and customising it.

As much as possible, interface components that are common to all systems are being utilised. A user's familiarity with these components will promote affordance, despite some minor visual alterations, such as colour. Clear labelling, sensible icons and tooltips, present a high level of visibility for both the function and state of a component. Mouse over actions and feedback messages provide a high level of feedback to all actions. On the whole, the user interface is extremely consistent, achieved through the reuse of components, colour palettes and iconography.

6.4 Schedule

The implementation will be grouped into *sprints*; each has its own goal and has been defined so that dependant tasks are completed before their dependencies.

Picon Framework

All work on the Picon framework will be completed first and is shown in Table 2.

Table 2 Table showing the tasks and goals for the Picon framework implementation sprints

Sprint	Tasks	Goal
Sprint 1	Create the main application class Create the application initialiser Create the request cycle Create the component render process Create labels, repeaters, lists and links Create basic model	By the end of this sprint, the context container, page map, configuration loader and request cycle should be completed. It should be possible to create a webpage extension class and associated HTML, and view it through a web browser. The stateless callback request should be implemented allowing the use of the link component.
Sprint 2	Create components for forms and form elements Create the database API Create the data table component and data provider Create property and compound property model	By the end of this sprint it should be possible to create and submit a form. The form should update the models of its form element components. It will be possible to create a DAO, capable of persisting submitted form data and also retrieving the data for display in a data table.
Sprint 3	Implement Ajax functionality Create resource requests Implement resource rendering	By the end of this sprint it should be possible to create an Ajax link and update a webpage through Ajax. It should also be possible to submit a form with Ajax. JavaScript resources will need to be included for this to work; the HTML head render process should be completed.
Sprint 4	Bug fixes Setup aspect orientation Create sample application	Any bugs encountered will be looked at in this sprint. The sample application will be created so the full test plan may be completed. AOP functionality will be implemented to improve efficiency.

Podium CMS

Work on the CMS will begin after Picon framework sprint three. Any new tasks for the framework which arise out of CMS requirements or bug discoveries will be completed in Picon framework sprint four. The schedule of work for the CMS is shown in Table 3.

Table 3 Table showing the tasks and goals for the implementation sprints of the content management system

Sprint	Tasks	Goal
Sprint 1	Create the layout editor Create the arrangement editor	By the end of this sprint it should be possible to create a simple layout of rows, columns and floating blocks and then populate that layout with “dummy” widgets
Sprint 2	Create widgets Create configuration panels for widgets Create the form manager Implement a user login and user management system	By the end of this sprint, the dummy widgets will have been replaced with real widgets which are fully configurable. It should be possible to create a form, define fields, options and validation and, for the form widget, specify a target form. The system should also require authentication with a user login, and it should be possible to create and edit users.
Sprint 3	Implement themes Implement content types Implement pages Implement posts Create the front end	By the end of this sprint it should be possible to define a content type and use it to create either a page or post. It should also be possible to create a theme. The front end pages should bring together all elements and construct a viewable, useable page for an end user.

7. Implementation and Testing

The process of implementation and testing is too extensive to document fully, however a few of the key issues and challenges I faced during the process are explored in this chapter.

7.1 The Framework

Before I begun development on the framework I conducted a few experiments to determine the best way of implementing callbacks. PHP enables the creation of anonymous functions called closures, which can be stored within a variable like a normal object; the closure class implements the invoke magic method, making it callable. Persistence of closures however proved problematic as a closure is inherently unserialisable. Some quick research online led me towards the *reflection* and *SPL* classes both of which, when used together, are able to locate the file in which the closure is defined and extract a string representation of its code. This string, alongside any arguments bound to the closure, which are themselves serialisable, can be serialised. To deserialise, the *eval* function can be used to recreate the closure from its original code. Essentially this solution does not enable the serialisation of closures, but provides a method to deconstruct and reconstruct them.

During the implementation of the URL structures described in appendix five, a problem arose with how those URLs could be reused. It is possible to bookmark URLs to revisit at a later date, or press the browser refresh button causing the request to be sent again. This caused a problem as the URL for either of the callback requests invokes an associated callback. It is entirely possible that this will cause a problem with the state of the page or that the callbacks associated component will no longer exist in the hierarchy. Consequently, such URLs should not be bookmarkable or refreshable. I was able to get around this using redirection; when either of the callback requests is received, after callback invocation, the framework will send a redirect header in the HTTP response, consisting of either a standard page request (for stateless pages) or a new non-callback stateful request (for

stateful pages), consisting of only the page ID. The redirection header uses the 301 HTTP status code, which informs search engine spiders to ignore the original request, thus not hindering search engine optimisation.

The usage of the PHPs *XMLParser* for HTML caused a serious issue with HTML entities. XML uses an entirely different entity set to HTML, and consequently any HTML entity not present in the XML entity set would cause a parsing error. It became necessary to implement a string replacement of entities within the HTML before parsing, followed by string replacement back again after parsing. Although a find and replace is hardly an elegant solution, and is likely to fall over on invalid syntax, as the syntax is validated, I do not consider this too much of a problem.

The framework ignores all HTML head tags other than that in the root component associated mark-up; this is achieved through the special tags defined in appendix five. Any CSS or JavaScript resources present in these ignored head tags are therefore not part of the rendered page. Consequently, it would be necessary to add any required tags programmatically during the render head process. To improve this, I have implemented a new special tag *picon:head*. When any tag is placed within the *picon:head* tag, it will be picked up by the framework and rendered during the render head stage.

Many of the requirements depend upon traversal of the component hierarchy, and yet no real definition of how this was to be achieved had been specified. The most suitable method for implementing this feature is the visitor pattern, which is designed for traversing a composition. I therefore added a new *visitChildren* method to the component class, which accepts a callback function as an argument. The visit method implementation recursively iterates through the entire component hierarchy, invoking the callback function for each component.

During development, a number of phases during both the application initialisation process and the component lifecycle emerged as needing additional actions applied in some circumstances. For example, to enable any component to use a resource from the context container, it became a

requirement for a component to call the injector on itself. The observer pattern, implemented already by component behaviours, was an excellent solution to this problem. I created an observer notification for component instantiation to invoke the injector. Furthermore, I created observer notification events for component initialisation, before and after rendering as well as application configuration loading, context container loading and page map initialisation.

Referencing a webpage class was somewhat problematic as PHP has no class identifier implementation. Using a string to refer to a page was one solution, but this reduced the maintainability of the framework by making it difficult to refactor – an issue I criticised in other frameworks. Consequently I created an identifier of my own: an object which is created dynamically at runtime and stores data about the class it identifies. Any class may now be referred using the identifier rather than a string.

During the development of the framework, I conducted testing of each feature as I completed it. As a result, there were very few issues that needed attention during the execution of the test plan. The test plan itself is however inadequate for comprehensively testing a framework. Although it was grey box and tested the interfaces offered by the framework, this offered little insight into how well the logic of framework was functioning. Thus, in addition to the test plan, I also implemented a series of white box tests using *phpUnit*. These ensured the interior operations such as serialisation, injection, and request processing were being handled correctly. The white box testing implementation also acted as a sanity check on the logic of the framework.

One of the issues that arose during the ongoing testing of the framework was efficiency. The amount of time taken to produce a response increased as more classes were added, and as more stateful page objects were added to the page map. After some investigation I discovered the causes. Firstly, all of the stateful pages were being deserialised at the beginning of a request and serialised at the end of a request, regardless of whether they were needed. Only one stateful page is ever needed for a request, and the deserialisation and reserialisation of every page caused a significant overhead.

Secondly, the class auto loader used recursive directory scanning to locate class files; this would iterate for each file in each directory every time a class was needed, regardless of previous iterations. Despite having initially rejected it during design, I turned to a file based approach for perstance as it would allow page objects to be selectively loaded and deserialised. Files would also allow for the creation of a cache so that the outcome of tasks, such as the class auto loader, could be stored so the tasks need not be performed again. Upon implementing the file based caching, the response reduced from an average of five seconds to an average of one, conforming to the non-functional response time requirement.

7.2 The Content Management System

The domain objects that evolved out of the business type model (Figure 57) are fairly complex and contain a large amount of data. Owing to this complexity it was necessary to run multiple queries on multiple tables on the database, to acquire all of the data required to construct them. In many cases the full object is not actually required. For example, the layout list requires only basic information about a layout; the full domain object which also includes all of the layout blocks within a layout are simply not needed. To improve the efficiency of the system, by reducing the unneeded database queries, I created two versions of many of the domain objects: a basic “list” version which is to be used to represent the business type on a list page, and a full version which extends the basic version and includes all of the properties the business type possesses. This full version is only constructed when it is needed, for example when clicking edit on a list page. The builder pattern was of great use here as it allowed for the domain objects to be constructed in steps.

The development of the CMS provided an informal method of testing on the framework. Although by the time CMS development started the framework was predominately completed and the test plan executed, a number of new issues that had not previously been considered arose. The usage of nested repeaters for example required a callback function to be declared inside of another. This had

not been considered when the serialiser of the framework had been written, and the process could not handle these nested closures and needed alteration to do so.

New requirements for the framework also emerged as a result of CMS development. I decided to merge the layout and arrangement dropdowns on the CMS interface using option groups to separate them, a feature that the dropdown component was not designed to handle, thus further development was required to enable the new feature.

The design of the user interface, whilst good on paper, proved to be not quite so desirable when implemented in terms of optimisation, particularly the navigation. Although the tabs and toolbars were successful, the primary navigation at the top of the page was frustrating to use, as the secondary navigation menu could not be accessed without first clicking on the corresponding primary navigation link and waiting for a new page to load. I altered the design slightly, turning the secondary navigation into a dropdown menu that appears on mouse over of the corresponding primary navigation item. This makes it significantly faster for a user to navigate to the page they desire and also reduces the number of clicks they have to make.

In the original design, no page had been considered as a “homepage” for the administration portal. Each of the use cases can be extracted to a page, but none was suitable to direct the user to immediately after logging in. I amended the user interface design to include a homepage – the dashboard, which offers information about the system and provides quick links for common tasks and also an audit trail of recent system activity.

The component architecture specified a number of system services for performing the complex task of data aggregation and presentation preparation. Whilst these system services had been correctly derived from processes, when it actually came to implementing them, I began to consider alternatives. With business services now performing the aggregation using the builder pattern, each business service populated the domain object entirely. Consequently, many of the system services

were now merely factories. This was not a problem, apart from the fact that they did not need to be services any more, nor declared as resources; thus I altered them to conform to the factory method pattern and offer a set of static methods for the various types of object instantiation the system required.

One of the non-functional requirements stipulates the cross-browser compatibility of the CMS. All of the JavaScript uses the *jQuery* library which is already tested across all major browsers, as is *tinyMCE*. However, the *jQuery* customisations needed to be tested as did the CSS I have created, to ensure that everything looks and works in the same, correct way in all the specified browsers. This is a very manual process, as no automated test can be written to check whether positioning, margins, etc look the same in one browser than in another. I conducted the tests quite simply be executing part of the test plan in each of the required web browsers.

8. Conclusions and Evaluation

This chapter looks back at the entire project, its processes, and the new software that has been developed.

8.1 Project Success

To suggest that I had bitten off more than I could chew would be an understatement. There was a tremendous amount of work to be completed by just one person in a very short period of time. Despite this however, I am pleased with the level that I have achieved. Whilst neither the framework nor the CMS is in a state where it could actually be used in a production environment, they are at a suitable stage of development to act as a proof of concept. I never expected to finish in entirety, but hoped that I would be able to demonstrate my solution, and I have achieved this.

The overall goal for the framework was to implement the concepts of Wicket and Spring in PHP, enabling the level of component orientation required for the CMS. I believe this has been successful; as a developer who has worked extensively with Wicket, I can use the similarities in the API and the same process of putting together a web application to easily migrate to Picon. Future development aside, the framework meets most of its requirements. Aspect orientation was of a low priority and, whilst useful for efficiency, it was not essential for the CMS and was never implemented. Furthermore, a number of the non-functional performance related requirements have not been fulfilled as successfully as I would have liked. The framework does not produce a response within one second in all circumstances; also, as the composition hierarchy gets larger, the time taken to render a page increases significantly, sometimes to a point where a timeout limit is reached. My main focus during implementation was to get the core functionality working, so implementation of the CMS could begin; unfortunately this meant that performance issues were somewhat neglected.

The initial concept of this project was to produce a CMS that enabled a user to create, entirely from scratch, the layout and design for their website using nothing more than their web browser and with no need for any technical knowledge. Whilst the CMS is unfinished, it does demonstrate that this concept is possible. The widget approach, combined with the abstraction of layouts, themes and arrangements, makes the process of producing a design simple. I believe that a non-technical user would be able to comprehend it relatively quickly, as I produced the user interface in a manner that works in a way that my experience has led me believe that a non-technical user perceives a website design. I am a great believer in the concept of good design being a design that works in a way a user expects, and does not force the user to change the way they work to accommodate it. I have epitomised this with the CMS user interface as much as possible, by hiding complexities and presenting dragable widgets that can be placed anywhere.

Picon is a very different framework when compared to the other PHP frameworks that already exist. When working with Wicket, the ideas I had about developing web applications, and the role of a framework completely changed – for the better. It was these new ideas that drove me to create a PHP implementation of the concepts. I think that other developers, whether they have worked with Wicket or not, will want to explore these ideas and therefore use the new framework.

Initially, my goal for the CMS was simply about managing design as well as content, but my research turned up another flaw in existing systems, specifically the lack of semantic control over content. The semantic web is evolving today and thus CMSs need to keep up with the changes. By expanding an idea from one of the systems I looked at to be more semantic, Podium is now one of the few CMSs which facilitates the creation of semantic content. Creation of content in this way, coupled with the component orientation of the Picon framework, presents a very powerful mechanism for building semantic websites; a Podium content type with the Picon panel component for example, is the perfect device for the implementation of Microformats.

8.2 Further Work

There are many new features that need to be added to the framework if it is to functionally stand up against the other PHP frameworks. Firstly, there is no support for internationalisation, thus making it difficult to use the framework to develop a website that is to support different locales. The white box testing coverage of the framework is also relatively low. Whilst the grey box tests which satisfy the functional requirements have been mostly fulfilled, a good set of white box tests should execute every line of code in order to ensure that it performs as expected. This is not the case however, as my tests have a code coverage of only about 38% and are focused mainly on the initialisation and serialisation processes; this should be expanded to encompass the rendering process and the component classes; as an untested framework with questionable robustness is essentially unusable.

The performance and efficiency of the framework must also be looked at; as mentioned previously, not all of the non-functional requirements have been fulfilled. Whilst some performance issues were addressed during development through the introduction of the cache, more work is needed to keep the response time to a consistently short period. Perhaps one second is a little ambitious for all scenarios; some further research and user acceptance testing would be beneficial to determine an ideal response time.

Many of the features of the CMS are still in their infancy; some were never implemented at all. For example, content types should enable the usage of a greater variety of attributes, the widget catalogue should be expanded to include a much greater selection of widgets for all requirements, and authorisation granularity should be expanded to more than just a single global setting. Arrangements and themes can be specified at content type and page level; however, they cannot be customised at these levels as I had originally designed. Furthermore, the layout block attribute editor was never implemented and the theme editor is very rudimentary. These features, and several others, would need a lot of further work to bring the CMS to a production ready stage.

Additionally, to become an industry standard product, future development should see the creation of an API to enable third party development.

The documentation of both the framework and the CMS should be improved significantly. Although all of the framework classes have doc blocks which explain their purpose, many of the methods and properties do not. Currently the only form of documentation available for the framework has been generated from the doc blocks. For the framework to be usable by anyone, doc blocks should be specified on every method and property, and also a definitive user guide should be created. Only when this is completed, will the framework fully fulfil the documentation requirement of component orientated programming.

Ultimately, the defining reason for not achieving full completion of the implementation was time and ambition. I think I was over ambitious when planning the project and gave myself far too much work to do. One of the most important lessons that I have learned from the development of these two products is time management and planning. Although I did plan my work in advance, and used *JIRA* to keep track of my progress, many tasks took significantly longer than I anticipated. Also, my focus was split across two products, instead of just one. I pressured myself into finishing the framework as early as possible, so that development on the CMS could begin. Consequently, a number of requirements, particularly performance requirements, were neglected. In future I think it would be far more sensible to work solely on one product at a time, and not move on until the first is satisfactorily completed.

I have also learned the importance of automated testing. Although time restricted the amount I was able to actually design, having completed manual tests during development, I did not expect many bugs upon development completion. However, despite keeping components loosely coupled, sometimes a significant change in one will disrupt another, thereby invaliding previous tests and requiring testing to be completed again. In future, I will ensure that I allow time to write unit and

integration tests using tools like *phpUnit* and *selenium*, automating the testing process and saving time in the long run.

8.3 Research

Locating reliable sources of information online can be difficult but, I found the method that yielded the best results was to use tools like *Google Scholar*, which enable searching for articles or online journals that had been cited by other people. Wikipedia as a resource can be very untrustworthy, however, as Wikipedia articles are required to cite their sources, I was able to locate the original works and use them instead.

The best source of information for the frameworks and CMSs I looked at was the official documentation, most of which was presented as either a wiki or eBook. Generated API documentation also proved to be an invaluable resource, particularly for requirements elicitation. Furthermore, as the documentation was official, and frequently written by the company or individual responsible for creating the framework or CMS, it was a very trustworthy source.

I did not use quite so many journals as perhaps I could have. Elements of this project, such as service orientated architecture, component orientated programming and aspect orientated programming are still relatively new and emerging areas, and are the subject of many journals. However, whilst such journals would have been suitable and interesting background reading, there was not enough time or relevance to the project to explore them in as much detail as I would have liked.

8.4 Requirements Elicitation and Analysis

As an individual, system planning methodologies were somewhat irrelevant for me, even more so as I already knew precisely the system I wanted to create. This does not mean that I did not use some

of the techniques on offer. I found *SWOT* the most useful; I analysed the strengths and weaknesses of systems created by other organisations, both software frameworks and CMSs. I did this partially to expose the inadequacies that I was focusing on in this project (component orientation in PHP frameworks and layout creation in CMSs); it was also a good opportunity to explore how they actually worked in as much detail as possible and discover the positive aspects they included. Ultimately, the exposing of weaknesses enabled me to more clearly understand what my problem domain was, and the discovery of strengths enabled me to build upon them and incorporate them into my design. Most importantly however, the research into the inner workings of the system gave me an incredible amount of background knowledge that was vital, not just in determining requirements, but also for design and implementation, as I had learned by observation how other more experienced developers were putting systems together.

Most of my requirements elicitation was achieved through analysis of documentation and to some degree observation, as explained in chapter five. Interviews and questionnaires are very useful techniques, but with no definitive users to target, it would have been very difficult to have sent questionnaires or conducted interviews. I think it would have been beneficial however, to have employed these techniques if I had had more time. I would have liked to locate companies and individuals who use either a framework or a CMS, to gain insight from them through interviews or questionnaires. Whilst I was able to define requirements through other techniques, and hardly needed to employ another method to acquire more, insight from real users would have at least acted as a form of validation for my existing requirements.

8.5 Design Processes

The design of the interfaces for the service components of the CMS was achieved using the methodology defined by Cheesman and Daniels, and involved the extrapolation of business and system services from business types and use cases respectively. Ultimately though, many of the

system services were removed during implementation, as the functionality they were to contain was more suited for implementation in one or more of the business services. The processes that had been identified for these removed system services to fulfil were ultimately just very complex business type operations. My decision to categorize these processes as system rather than business services was most likely caused by my unfamiliarity with the service identification process; the concepts of which are relatively new to me and I have not previously used them independently. I have taken from this project, a more clear understanding of the differences between business and system services, and I will endeavour to explore the process of design in this manner in more depth in the future.

During my research into frameworks, I was able to locate a doctoral thesis on framework design using a role modelling approach. It was a useful resource, particularly for terminology, but also presented a methodology for the design of a framework. Whilst its content was fascinating, it was extremely difficult to follow; it was unlike any design methodology I had come across. I had hoped upon discovery of the thesis that I would be able to use it to enrich my framework design; however I did not have the time or a suitable level of background knowledge to assimilate the concepts and processes the thesis described. Essentially, I did not use a real methodology to formulate the framework design; everything evolved out of experimentation with other frameworks and research into their APIs. I do not consider this a bad thing as I believe I have a sensible framework design; however in the future, it would be prudent to seek out specialist processes and practices for framework design, at least as a method to validate my own design.

In the area of user interfaces, I followed a process of wireframe prototyping. Although I previously discredited prototyping in this project as there are no users from which to obtain feedback, visual design is very different from requirements discovery. Frequently my opinion of my designs changed over time; I often created something I was happy with, but a few days later I was not. Prototyping allowed me to continuously evolve my designs, critiquing and tweaking them until I was completely

satisfied. This is a process that I feel is highly effective; I have used it the past and will continue to do so.

8.6 Development Methodologies

I chose Agile as a development methodology for a number of reasons. Firstly, Agile is iterative and each iteration involved design, implementation and testing. This process allowed me to start working sooner, before a complete design had been produced. Secondly, it allowed me to alter requirements as an ongoing process; for example, the introduction of file based caching was not an original requirement but grew out of the need to improve framework performance. The issue was identified at a fairly early stage and the additional work was easily incorporated into sprint 3. Agile allows for this sort of workflow, by encouraging adaptability over adhering to a plan. Thirdly, documentation in Agile is a secondary concern to working software; whilst this does not excuse a lack of documentation it did allow me to prioritise my work more effectively.

Despite following some of the Agile manifesto, I have not been truly agile. The main reason for this was the individual nature of the project. Agile is team working methodology and promotes good team communication and collaboration through a number of processes, such as the daily stand-ups and end of sprint scrums, neither of which are applicable to me. Agile also defines the process of continuations integration, in which the application being developed is tested automatically every ten minutes. Although I have developed some automatic tests, their code coverage is not enough, nor are they run as frequently as they could be. Primarily, I think this was caused by time constraints – I was more focused on getting the applications finished. Although I performed manual testing, as previously mentioned, creation of tests should be given a higher priority in future. Although I personally find them time consuming to write, once written they can be run continuously with little effort – ultimately saving time.

A scrum would be a pointless exercise with only one person; this does not mean however that I did not continually evaluate and review the work that I was doing. I tried to review my own code and modify it accordingly as much as possible a few days after I had written it; waiting enabled me to be less subjective.

Consistent code styles, formatting and best practices are also promoted in Agile as they enable greater collaboration. Whilst I have followed the formatting style I prefer and have used best practice as much as I can, without the aid of any software to check standards as part of a continuous integration process, it is unlikely standards are maintained throughout. Tools such as *PMD* and *CheckStyle* would have been beneficial to use, as they would highlight any style errors, code smells, over complicated expressions and many other standards or formatting related issues. This would not only have helped identify potential bugs early on, but also keep the code base more consistent. In future, I will try to incorporate the usage of these tools into my code review workflow.

8.7 Final Thoughts

In hindsight, I still stand by my decision not to use any existing PHP framework. Whilst not doing so would have allowed me to refocus my efforts on implementing the CMS, and would most likely have resulted in a more complete system, I do not think that I would be nearly as happy with the PHP code running it. I would have been able to produce the CMS in an existing framework, but not in the same component orientated way that I envisaged.

I have learned a tremendous amount during this project. Through both reading and practical experimentation and implementation, I have deepened my understanding of PHP, particularly in the area of serialisation. I have also expanded my knowledge with regard to components, services, aspects and the MVC. Although not fully realised and implemented in my solution, the concepts I explored in chapter four, particularly those concerning separation of concerns and code tangling,

contributed to my design and implementation of a framework that I believe is highly cohesive, loosely coupled, and most importantly, extremely reusable.

With regard to completing an independent project, I've learned that I need to be iterative and adaptive. An iterative process applied not just my design and implementation, but also my research and analysis. As I continued to learn, through both my own personal work and university tuition, I found it necessary to go back and research some areas in more, and to rework aspects of this report to improve what I had already done. I thought that I had a very clear idea of precisely what needed to be done in terms of implementation before I even began to fully define the problem, as I have been thinking about developing a CMS for several years. I had to be adaptive though, to evolve my original thoughts as research elicited new ideas that were worth including, for example semantic content. I also had to make compromises as the project progressed, and it become evident I would not have time to complete everything I had planned, for example aspect orientation.

I have also learned that there was a lot more layout and design functionality available in existing CMSs that I anticipated, and many of them actually implemented it reasonably well. As explored in chapter three though, there are many shortcomings to the existing approaches. I think the fact that layout and design functionality is available in existing systems in some form, illustrates that there is a need for it, and that CMS users are looking to work with more than just content. I believe this justifies my project.

Overall, I think this project has been a success. There is still a lot of work to be done to bring the two products I have developed to a production standard, but I am able to demonstrate them as realisations of my ideas. Personally, I have learned an incredible amount about new technologies and concepts, and how to apply some of them; and also about how focused and motivated I am when working independently on a project of such a daunting size.

A1. Methodologies

This appendix explores the methodologies used throughout this project, comparing and evaluating them against their alternatives.

A1.1 Software Development

Requirements analysis, design, implementation, testing, deployment and maintenance are the processes of producing a piece of software common to most software development methodologies. This section looks primarily at methodologies for implementation and testing.

A1.1.1 Agile

Agile is a form of rapid application development, and is a method of project management designed specifically for software development.

Agile is highly adaptive, increases productivity and improves organization, technical and personal success. (Shore & Warden, 2007)

The agile manifesto lays out four values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan (Beck et al., 2001)

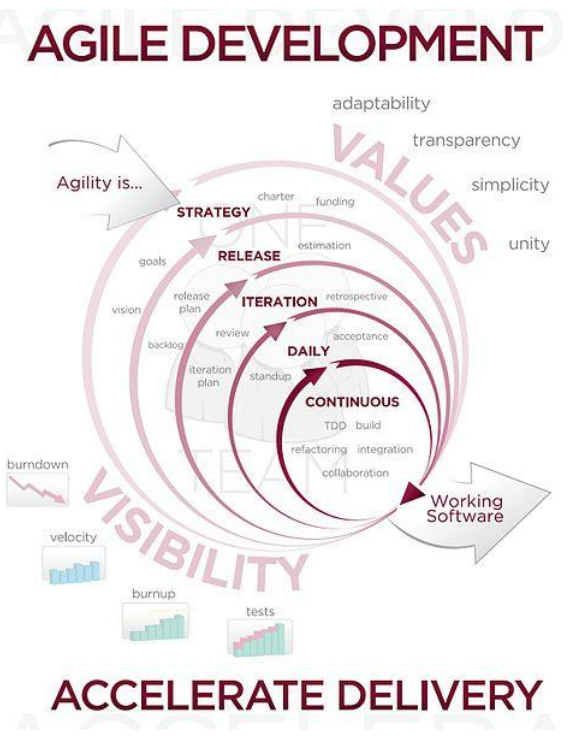

Figure 27 The agile development process **(Wikipedia, 2012)**

In Agile, teams are self organizing, can respond quickly to changing requirements, produce working software in short iterations (weeks) instead of releases (months), keep everyone in the know on a daily basis, integrate components continuously to save time later and have a consistent pace. (Beck et al., 2001) The software development life cycle (plan, analysis, design, code, test, deploy) is reduced and compressed. There is less work on design, testing and documentation and the focus is placed on multiple iterations of coding on which developers work every day. This is more productive and software can be produced extremely quickly. This is known as “Extreme Programming”. (Shore & Warden, 2007) The full process is shown in Figure 27.

Continuous

There is often a delay between development being done and the time the application is actually ready to use. Continuous integration is an automated build of the application that will ensure it is always in a state that can be deployed. It also ensures that developers keep their code up-to-date with modifications made by others, updating several times per day. This means that merging code is a lot faster than it would be if done only every few days or even weeks, as code can change rapidly.

Due to speed at which development progresses, implementations can cause code to become disorganized, redundant or inefficient. Code refactoring is the process of changing the structure of the code, making it cleaner and more efficient, without changing what it does. This process is performed constantly to keep code organized. (Shore & Warden, 2007)

Coding standards are important in Agile as they keep source code consistent. This consistency is vital when working in a team, as it means developers will be able to work more easily on each other’s code. This means that a developer working on code written by another can start working on a task straight away, and does not need to spend time understanding and interpreting the code. (Shore & Warden, 2007) Standards apply, not just to formatting, but also the way code is structured, for example keeping methods short or not passing null references.

Daily

Every day, the development team gathers for a meeting known as a stand-up. (Shore & Warden, 2007) In this meeting, each team member summarizes what they did the previous day, what they plan to do today and any problems or "*blockers*" they have. This is a great opportunity for the project manager to find out what the team is doing and make sure work is progressing at a good pace. It also allows any problems to be addressed and tasks to be re-assigned if needed, based on team members' other commitments or suitability of skill set. Primarily though, the stand-up is a fast way of keeping all members of the team informed as to what every other member is working on. Meeting and communicating in this way is emphasized by Agile, which encourages verbal face-to-face communication over written documentation.

Iteration

Each iteration of development is known as a *sprint*, typically lasting for a couple of weeks. Each *sprint* will have a number of goals, which are to be fulfilled by completing a number of tasks shared out amongst developers. At the end of sprint, the team meets to reflect on their progress, and plan the next sprint in a meeting known as a *scrum*. The retrospective activities are not used to assign blame to any failures, but to allow the team to learn and improve, to be honest about success and failures, so that future work can be improved; thus the team will become more cohesive. (Shore & Warden, 2007) As well as a retrospective on the last *sprint*, teams will plan tasks for the next one, based on new, unfinished or outstanding requirements. As project requirements or situations can change rapidly, processes for the next *sprint* will be modified as required. (Shore & Warden, 2007)

Release

After multiple iterations the software goes through a testing and documentation phase. *"Post Hoc"* documentation decreases the cost of writing it and makes it more accurate. (Shore & Warden, 2007) Software is tested internally, and then goes into customer trials and finally general release. *"Done Done"* is the final step when an application is production ready. It will have been refactored and

integrated to the developers' satisfaction; all end-to-end tests will have been completed successfully and will have been reviewed and accepted by customers. (Shore & Warden, 2007)

A1.1.2 Rapid Application Development

Rapid application development prioritizes speed of delivery over technical excellence and planning. (Maciaszek, 2001) In RAD, software evolves out of prototypes. During the requirements elicitation phase, quick mock-ups of the GUI are developed with hard coded values and no real backend implementation. This process allows potential users to immediately see and critique the envisaged system. As the system is merely a prototype, alterations to it through multiple iterations are fast and cost effective. Prototypes are produced continuously until the users and stakeholders of the new system are happy to proceed with full implementation. In RAD, the final prototype is improved by implementing backend coding and replacing hard coded values with dynamic ones – producing the finished software.

In RAD, time constraints are extremely strict. *Time boxing* is a method which imposes a stringent deadline for a project to be completed. If any new requirements emerge during implementation they are considered "*scope creep*" and are consequently ignored. If a deadline is approaching and the project seems unlikely to be completed on time, the scope is reduced so that the deadline can be met.

In RAD, a project will involve the best programmers and designers that are available, known also as *Specialists with Advanced Tools*. These specialists work closely with the users and stakeholders of the system to be developed in *Interactive Joint Application Development* (Interactive JAD). Interactive JAD is a process which gathers developers, stakeholders and users to discuss the new system and develop a set of requirements. JAD capitalizes on the group dynamic, facilitating faster learning, increased productivity, production of better solutions, and eliminates more errors. (Maciaszek, 2001)

A1.1.3 Waterfall

The waterfall methodology divides software development into distinct phases: feasibility study, requirements analysis, design, implementation, testing and maintenance. The name refers to process in which one phase flows into the next. (Mall, 2004)

The feasibility study determines whether it financially and technically possible to develop the proposed system. During the study, an abstraction of the problem definition is produced along with potential solutions.

Requirements analysis includes the process of capturing, analyzing and documenting the requirements that the system must fulfil. Requirements are elicited from the systems potential users through a number of techniques explored in A1.2.

In the design phase, the requirements are developed into software architecture. In object orientated design, this process consists of the formulation of the classes the system must have to fulfil its requirements, the attributes and operations those classes have, and the relationships those classes have with each other.

The implementation phase begins with the translation of the design into source code, producing the modules of the system which, in object orientated programming, would be the classes. During this phase, unit testing is conducted on each of the modules at a white box level to test fulfilment of functional requirements.

In the testing phase, integration and system testing is performed. The modules created during implementation are brought together one at a time to test their interoperability. System testing is an overall black box testing process completed on the system as a whole.

The final phase is maintenance and is performed after the software has been deployed. It revolves around the correction of any errors or bugs in the system, improvement of its implementation and alterations of the system to permit it to work in new environments.

A1.1.4 Conclusions

RAD may be fast, but it comes at a price; documentation is deficient, code is of poor quality and thus difficult to maintain in both the tracing and fixing of bugs and implementation of new features, and user interfaces are often inconsistent. Furthermore, reuse of developed software components is problematic as the code is specialized and bespoke. Although a form of RAD, Agile incorporates a number of processes which solve many of these issues. Firstly, code quality is encouraged through the adherence to coding standards, the formalization of refactoring and the process of code review. Code reviews happen on a continuous basis, culminating in the formation of tasks at the end of a *sprint* to make code improvements, based on critiques from other team members. The process of refactoring is designed to generalize code, which is far better in the long run, as components become increasingly reusable.

Agile does not emphasize documentation during the implementation process, it is expected that time will be taken at the end of implantation to produce it. By being adaptive, software can change dramatically between the start and end of its implementation; consequently it is far more sensible to complete documentation tasks at the end when changes are no longer happening, and thus documents will not be invalidated or become outdated.

Agile is not perfect however; it is designed for teams and offers many processes to improve team communication, such as the daily stand-up meeting. These processes however ultimately make it difficult to scale for much larger teams spanning multiple locations.

Comparatively, the waterfall model is inferior to the iterative processes of RAD, as it assumes that each phase of its lifecycle can be completed successfully and flawlessly before moving on, which is very unlikely to occur when put into practice. Waterfall does not take changing requirements into account. It is very common for users and stakeholders to be unaware of what they actually want out of a system, and if required functionality or interface designs change after their respective phases are completed, much of the work completed in subsequent phases is invalidated. Furthermore, it is

frequently difficult to know until implementation whether a design can actually be fully realized in the anticipated manner. Consequently, it is often preferable to alter the design of a piece of functionality, rather than persist in implementing what has turned out to be a faulty design based on bad predications or assumptions.

Ultimately, both RAD and waterfall have their disadvantages. For this reason, many organizations opt for a hybrid approach between the two. (Maciaszek, 2001)

A1.2 Analysis

This section focuses on the processes that motivate the need for new software systems to be developed and some of the techniques used for requirements elicitation and definition.

A1.2.1 System Planning

The process of system planning comprises the identification, classification and prioritization of potential software systems that an organization will need, in order to fulfil their goals or improve an existing system or service. (Maciaszek, 2001)

Strengths, Weaknesses, Opportunities, Threats (SWOT)

Areas within an organization such as management, production and development, can be analyzed to define their strengths and weaknesses. Additionally, factors external to an organization such as new technologies, economic or political developments, or rival organizations can be used to elicit potential opportunities and threats. When these four aspects are combined, they can be used to define objectives for an organization, which can ultimately evolve into a motivation for a new piece of software to be developed to accomplish those objectives.

Value Chain Model (VCM)

The value chain model views all the processes within an organization as a sequential chain of activities spanning all areas such as shipping, customers, and production. VCM works on the idea that any weak link within the chain will affect the entire organization. Thus the model aims to identify which chain configuration will yield the best competitive advantage. The development of new software systems can then be targeted to the processes and operations that will realize those advantages.

A1.2.2 Requirements Elicitation

Requirements elicitation is the process of consultation and discovery of the requirements that a new software system must fulfil. These techniques have been mentioned briefly in chapter five, but are explored in more depth here.

Any system will have targeted users or customers; two common techniques for discovering requirements are interviewing these users or supplying them with questionnaires. Both techniques suffer from the disadvantage that users infrequently know what they actually want out of a new system, and if they do, they often find it difficult to express. (Maciaszek, 2001) Furthermore, interviewees may be reluctant to answer questions about failures or inadequacies in current systems, an issue not present in questionnaires as they can be anonymous. Questionnaires are however less productive, as clarification of answers cannot be immediately sought. Interviews are not restricted to users or customers; experts on the problem domain can also be interviewed so that an analyst can gain a better understanding of the domain of the envisaged software.

A more effective technique is observation which consists of the analyst observing the customers or users working with the current system that is in place. This is more beneficial when conducted passively, so that the users or customers are not aware of the observation and will thus work and behave as they normally would. An alternative to this is active observation, whereby an analyst actually participates in the usage of the current system alongside the potential users or customers.

The study of documentation for existing systems and processes is an invaluable technique for the discovery of requirements and also for the comprehension of the problem domain. In addition to documentation, reports of problems and requests for new features and improvements are equally just as important to review.

There are also a number of more contemporary techniques for requirements elicitation: prototyping, JAD and RAD, described in the previous section. Prototyping and JAD are purely for requirements discovery and continuous discussion and review, as requirements evolve. RAD, whilst also being a full development methodology, does include specification for requirements discovery and analysis.

A2. Framework Requirements

This appendix is a continuation of the framework requirements analysis in chapter five. A complete set of functional requirements are explicitly listed, classified, prioritised, and requirement dependencies are stated in this appendix. As in chapter five, the user is assumed to be a developer using the framework to build an application.

A2.1 MVC System Requirements

Identity	Dependency	Priority	Description
1	-	High	The framework will offer the functionality to build a compositional hierarchy of components at runtime. Each component will consist of a class and some associated mark-up. The root node of the hierarchy will be a webpage component. A component class should exist for each type of HTML tag that requires specialist server side functionality and a single generic component for those that do not. (See A2.2)
2	1, 3-5	High	Associated mark-up for a component will be derived from an HTML file or from HTML inherited from the parent component. It is required that the root component has an associated HTML file as there is no parent to inherit from. If a component has an associated HTML file, all of the mark-up within it will form the associated mark-up, unless specified otherwise. Components which inherit mark-up from a parent component will do so through the use of an identifier. The component class will be given a unique identifier, which will correspond to an identifier on the HTML tag it is to be associated with in the parent component mark-up. A component class will locate its mark-up by searching for a tag with a matching identifier in the parent mark-up.
3	2, 4	High	The associated mark-up will need to be made available to the component class at runtime so that it may be manipulated; associated HTML files will need to be parsed into an object representation to facilitate this. The object must enable the tag name, attributes, child tags and text nodes to be altered, and must also facilitate recursive searching so that a tag with an identifier can be located within the object.
4	-	High	As part of the mark-up parsing process, the mark-up must be validated. Invalid mark-up will result in an invalid object representation which may produce unpredictable results when the object is used.
5	-	High	The framework shall offer the means for an HTML tag and an instance of a component class to be given an identifier. The identifier in the HTML must not invalidate the mark-up in any way. After a tag has been parsed, the identifier for that tag must be present in the object that represents it.
6	4, 2	High	Components with an associated HTML file will not necessarily be root components in the hierarchy. An associated HTML file is expected to be valid; however, to fulfil that requirement it will need html and body tags. Consequently, when associated mark-up for a non root component is utilised there will be a duplication of html and body tags, as these tags will already be present in the webpage associated mark-up. To prevent this from happening, the framework must present a method to isolate

			the mark-up required by a component in its associated HTML file, separating it from the mandatory tags required to keep the HTML valid.
7	1-5, 8	High	A component must provide a mechanism to be rendered. Once the component class has been created and added to the hierarchy it will need to load its associated mark-up object; it will then need to alter the mark-up object as required. The rendering process will need to parse the altered mark-up object, producing plain HTML. The component must recursively invoke the render processes on its child nodes. The render process may not start on the root node; as a component may need to be rendered individually.
8	2	High	The framework will enable the definition of a global layout through mark-up inheritance. The HTML file associated with a component class must enable inheritance in the same way as the class does. If a class with an associated HTML file has one or more parent classes which also have an associated HTML file; that mark-up may be inherited or overridden. If a class does not have an associated HTML file but a parent class does, the parent HTML file will form the associated mark-up for the component. If both the class and its parent class have mark-up, the framework must provide the user with means to specify if the parent mark-up is being overridden, or if the parent mark-up is to be extended by merging it into the mark-up of the child class.
9	2	Medium	The framework should provide the means for a component class to validate the tag type of the mark-up it is associated with. For example, a form component should only be associated with an HTML form tag.
10	7	High	The rendering process must validate after completion ensuring that all components have been rendered.
11	1	High	The framework must allow a component to be associated with a model. A model will be a wrapper for any domain object. This internal object may be specifically placed inside the model; alternatively the model may be given the information it needs to retrieve an internal model from another location. (See A2.3) The model object in some circumstances must conform to a specification required by the component it is associated with. A model is optional for most components; however, those that require an information source to complete their function must mandatorily have a model specified.
12	1, 7	High	The HTML head tag must be automatically associated with a header component class. When the header component is rendered, it will inform every other component in the hierarchy so they may be permitted to supply a header contribution – a CSS or JavaScript resource for example.
13	15	High	The framework must provide a routing and dispatching process which analyses the URI of an HTTP request and determines the action to take. (See A2.4)
14	15	High	The framework must be capable of generating URLs automatically, these URLs will conform to the request scenario requirements in A2.4
15	-	High	Each component must declare whether it is stateful or not. The framework must recursively analyse a webpage and all of its child components for this stateful declaration. If one or more components are stateful, the framework will serialise the component hierarchy and all of its dependencies, for use in subsequent requests against an automatically generated page ID.
16	1	High	The framework must provide the means to allow one or more callback methods to be associated with a component.
17	1, 14	High	The framework must provide the means to generate a component path for any component in the hierarchy.
18	1, 13	High	The framework must provide the means to locate a component within the hierarchy using a component path.

19	-	High	The directory in which framework classes, assets and other resources are stored must be web protected.
20	12, 13	High	The framework must act as a proxy so that resources, such as JavaScript and CSS, in web protected directories can be referenced in the HTML head.
21	13, 15	High	The framework must contain components which support Ajax behaviour. They will need to have callback functions associated with them for invocation during an Ajax request.
22	21	High	The framework must be able to detect whether a webpage contains an Ajax component, and ensure that the required client side JavaScript resources are added to the HTML head.
23	1	High	The framework shall offer the facility for a component to report a feedback message with the following severity statuses: fatal, error, warning, information and success. For reasons of context, messages should not persist from one request to another, regardless of whether they are actually output in any form.
24	-	High	The framework shall provide a data provider which, although not referred to as a model, is functionally a model. The data provider will be used as an information source for a data table component (see A2.2). The data provider shall provide an interface which is capable of returning a data set. The interface must enable the total numbers of records in a data set to be returned and a sub set of the records to be retrieved, enabling pagination. The returned subset of records should consist of a list of domain objects of the same type, each representing one record.
25	1, 26	High	The framework will allow a user to specify one or more component classes which require authorisation. Each time one of those classes is instantiated, a user defined callback function will be invoked by the framework. It is expected that the callback function will determine whether the authorisation criteria is met and return a boolean. If the returned value is false (not authorised), a second user defined callback function will be invoked, in which it is expected the callback function will direct the end user (the user accessing an application created with the framework) to a login page, insufficient permission page or similar.
26	13	High	The framework shall provide the facility to internally redirect to another page. This should be achieved by restarting the framework processing at the routing and dispatching stage but specifying a target webpage, overriding the default router and dispatcher which determines the page based on the HTTP request.

A2.2 Components

This section lists and describes all of the visual components that the framework must contain, whether they use mark-up inherited from a parent component or sourced from an associated HTML file, and whether the component must be the last child node in the composition hierarchy or supports the addition of other child components. All components are high priority and, whilst the framework may contain other components for other more complex and bespoke functions, these have been identified as required for the scope of this project. Unless specified otherwise, a

component has an optional model with no model object specificity and may correspond to any HTML tag.

Name	Associated	Inherited	Child Nodes	Description
Webpage	●		●	This component will always be the root within the composition hierarchy. The user will define web pages by creating their own extension class; it will be their responsibility to write the class and create an HTML file for it to use as its associated mark-up
Panel	●		●	A panel will enable a block of components to be reused across multiple pages. This is comparable to an extracted fragment of mark-up. The panel component facilitates this reuse by having associated mark-up of its own. A panel component must still have an identifier, as this will specify the tag in the parent component mark-up where the panel is to be rendered. However, the panel replaces the inherited mark-up entirely with its own. A user must create a panel by writing their own extension class and associated HTML file.
Border	●	●	●	The border component works in the same way as a panel although it does not replace the existing mark-up of the tag in the parent component mark-up that its identifier corresponds to, but instead offers a method for wrapping the border associated mark-up around it.
Form		●	●	A form component must correspond to an HTML form tag. The form component will provide functionality to process post data. A form component may have two callback functions associated with it; both are to be invoked on submit but one to be used if the form is valid, one if it is not. The component will automatically add a URL to the action attribute of the form tag, which will resolve to the form processing process. The processing process must locate the form field components and pass each one its posted value for sanitation and validation. If the entire form validates successfully, the model object of each form field will be updated to reflect its new value. Once completed, it will invoke the appropriate callback. It is expected the user defined callback for a valid submit will implement any necessary persistence for the newly updated model objects.
Text Field		●	●	A text field component may correspond only to an input tag with a type attribute that has the value of text. The component must enable one or more validation rules to be specified and also require a data type to be set. A data type may be string, integer, floating point number or boolean. The type of the model object must match the specified data type of the field.
Text Area		●	●	A text area component may correspond only to a text area tag. The component must present a setting to make it mandatory. The model object must be a string.
Drop Down		●	●	A drop down component may correspond only to a select tag. A user must mandatorily specify a list of choices that are available; the drop down must process these choices and render suitable option tags. The choices need not be any particular type, however if an object is specified, the user must provide a callback function which is to be used to render the string to use as the option text. The callback will be invoked for each object in the list. The model object of the component must match the type of the objects in the choice list. The choice list objects must all be the same type.
Check Box		●	●	A check box component may correspond only to an input tag with a type attribute that has the value of checkbox. A check box stipulates that its model object must be a boolean, enabling it to detect whether or not it is selected.
Check Box Group		●	●	A check box group is a component which may be added as a parent component to one or more check boxes. When utilised, the retrieval of the model object is deferred to the check box group from the check boxes. The check box group must have an array model. This component enables a group of check boxes to work together. Each check box must have a non-inherited model specified; this model becomes the value for the check box. When the check box is checked, the value is placed into the array model; when it is not checked the value is removed from the array model. The initial state of the array model, combined with the values for each check box, can be used to determine an initial checked or not checked state of each checkbox.
Radio		●	●	A radio may only be added to an input tag with a type attribute that has the value of radio. As a radio is useless by itself it can be used only in conjunction with a radio group.

Radio Group		●	●	The radio group works in exactly the same way as a check group, but takes as its model object only a single object and not an array. This single object will be altered, based on the value of the currently selected radio. The value of the radio is specified in the same way as the value of a checkbox in a checkbox group – through a non-inherited model.
Radio Choice		●	●	A radio choice is a wrapping component for a radio group. The radio choice takes in a list of choices in the same way as a drop down, and will dynamically create a radio for each item in the list, setting the model as required. The radios will be child nodes of a radio group, which will also be created by the radio choice component. The model object for the radio choice must conform to the same specifications as the radio group. The radio choice will internally pass its model to the radio group it creates.
Check Choice		●	●	The check choice component works in the same way as the radio choice but instead creates check boxes and a check box group from the list of choices. The model is internally passed to the check box group in the same way as in radio choice; the model must however be an array model.
Button		●	●	A form button component may only correspond to an input tag with a type of submit or a button tag. The component will have two associated callbacks, one to invoke if the form is validated successfully, and one if there are errors. When the button is clicked, the form will be submitted using the URL in the form action attribute, however, the framework will ensure that the name attribute of the button in the rendered mark-up is set to the component path of the button. When the form is processing, the form component will internally locate any buttons that are child nodes of itself; if the form finds a button which matches the component path in the post data, after executing its own callbacks it will execute those of the button, enabling form and button specific callbacks to be defined.
Link		●	●	The link component may correspond only to an anchor tag. The component will have an associated callback to execute when the link is clicked. The component will generate URL for itself and place it in the href attribute.
Label		●		A label is a simple component capable of displaying the contents of a model as the text within any tag (although primarily intended for use on span or label tags, this is not imposed). As the entire contents of the tag will show the value of the model, this component may not have any child nodes.
Ajax Link		●	●	This component works in precisely the same way as the link component, however, the generated URL is instead wrapped inside a JavaScript function invocation (where the URL forms an argument) and placed in the onClick attribute. The JavaScript function will be located in a JavaScript file, which included automatically in the HTML head by the framework. The function will initiate a GET request using the provided URL, and update the onscreen HTML based on the received response.
Ajax Button		●	●	This component works in precisely the same way as the button component, however, the type attribute will be set to button (it is anticipated the type value will be initially submit which would cause the button to submit the form in the normal way if not altered). The generated URL will be placed in the onClick attribute wrapped inside the JavaScript function invocation, in the same way as the Ajax Link component. The JavaScript function will however be different, and will locate the form the button belongs to, extract the form field values and use them to construct a POST request using the provided URL. The response will be interpreted in the same way as with the Ajax Link.
List		●	●	This component must mandatorily have an array model specified. The mark-up of the list component will be rendered multiple times, once for each item in the array within the array model. For each iteration, a new list item component will be added as a child to the list component. The list item component will directly use the mark-up associated with the list component, which will not itself render any HTML output as it defers this to the list item component. For each list item component, a user defined callback will be invoked so that any child component that is required in each list item may be added to the hierarchy.
Repeater		●	●	A repeater component works using the same principle as the list component in terms of mark-up handling and deferred rendering, however, a repeater does not need a model. Each component added as a child to the repeater component, regardless of what type of component it is, will form the equivalent of the list item and directly use the associated mark-up of the repeater.
Data Table		●	●	A data table will be a repeater of repeaters, allowing for both rows and columns to be created. A data table will run from a data provider object which the user must specify. The data table will automatically add the required child components to the repeater of repeaters, enabling a data set to be shown. The data table will automatically add in a row for column headings and provide pagination controls based on a user defined number of rows per page. The user must also define the names of the columns and the property of

				the row object to use as the value for each column.
Feedback Panel	●		●	A panel component which contains a list, whose model is derived from the list of feedback messages reported by components.
Tab Panel	●		●	A tab panel component will consist of a list and a panel. Each item in the list will contain a link that when clicked changes the content of the panel. The user must define a list for the tab panel to use. Each item in the list must contain the name of the tab and a panel extension class that corresponds to it.
Mark-up Container		●	●	A generic component which can correspond to any tag.

A2.3 Models

The table below describes the types of model that the framework will provide. All models are high priority.

Name	Description
Basic Model	A model capable of containing any object. The object must be specified when the model is created.
Property Model	A model which obtains its object dynamically from instance data. The target object and the property to obtain the model object must be specified.
Compound Property Model	A model which can be inherited down the composition hierarchy. The model object is located in the same way as the property model. When a component attempts to resolve its model, if none is specified but a compound property model in a parent component is available, the model will be inherited. The model object will be located firstly, by locating the object from the instance data of the target object. A property, whose name is the same as the identifier of the component inheriting the model, will then be located within the object located in the first step. The value of this property will be the model object for the component.
Array Model	Works like a basic model but stipulates that the provided object must be an array.

A2.4 Request Routing and Dispatching Scenarios

The framework routing and dispatching process must be capable of handling the request scenarios described below.

Name	Description	URI Requirements	Resolves to
Standard page request	An HTTP request for a webpage. This is a browser bookmarkable URL.	The URI must contain the name of the webpage	A webpage component. When receiving this request, the framework must extract the name of webpage component the request is expecting, instantiate and render it.
Stateless callback request	An HTTP request for a callback to be invoked. This type of request can only be initiated by clicking on a link, button or similar, on a page	The URI must contain the name of the webpage and the component path	As with the page request, the target webpage component will be extracted and instantiated. The URI must contain the component path to the component with an associated callback. This callback will be invoked. The webpage will then be

	generated by the framework. The page did not contain any stateful components.		rendered.
Stateful callback request	The same as a stateless callback request, except the page contained one or more stateful components. The webpage component, its hierarchy and dependencies will have been serialised.	The URI must contain the page ID and the component path	The URI will contain the page ID. The framework must locate and deserialised the webpage component that corresponds to the ID. As with the stateless callback request, the URI will also contain the component path to the component with the associated callback. The framework will invoke the callback and render the page.
Ajax request	An HTTP request initiated by client side JavaScript in response to a DOM event. The page on which the DOM event was initiated, will have been generated by the framework and will be marked as stateful and therefore will have been serialised.	The URI must contain the page ID and the component path	As with the stateful callback request the page will be located and deserialised using the page ID from the URI. The callback will be located and invoked using the component path. The callback will however specify one or more components in the hierarchy that are to be re-rendered. The framework must individually render each of the specified components, and place the resulting HTML into a response in a format that can be interpreted by the client side JavaScript, enabling the existing client side HTML to be updated.
Resource request	A request for a resource such as JavaScript or CSS. Although such resources are usually specified with a direct path, a resource automatically added to a page by the framework (for example the JavaScript file containing the Ajax functions) will be stored in a web protected directory and must be resolved with this type of request.	The URIL must contain the component class name and resource name	Using the same method as an associated HTML file, a component class may also have one or more resources associated with it. The URI for a resource request is expected to contain the resource name and the component class it is associated with. The framework must use this information to locate the resource file and write its content to the response, ensuring that the content type is set accordingly, based on the type of resource.

A2.5 Resource Container Requirements

Identity	Dependency	Priority	Description
27	-	High	The framework shall offer a means for a class to mark itself for automatic instantiation
28	28	High	The framework will provide a bootstrap process to locate and instantiate marked classes (instances of which are to be known as resources); a name must be created for each resource
29	27, 28	High	The framework must place the resources into a container which provides an interface to retrieve a resource by name
30	-	High	The framework must provide the means for a resource to declare which other resources it depends upon.

31	29, 30	High	The framework shall provide a dependency injector which will, for each resource, retrieve and insert the required resources based on the dependency declaration
32	29, 30	High	The dependency declaration and resource dependency injector should be expandable for use in any other framework, user or extension class.

A2.6 Database Requirements

Identity	Dependency	Priority	Description
33	-	High	The framework will provide a database driver which connects to a database, performs the physical database operations, and disconnects when the connection is finished with
34	-	High	The framework will provide a utility to prepare SQL for execution
35	-	High	The framework will provide a utility for result set mapping onto user defined domain objects
36	-	High	The framework will include a means of specifying database connections that need to be established

A2.7 Aspect Requirements

Identity	Dependency	Priority	Description
37	-	Low	The framework must provide a means for a user to declare an aspect containing point cuts and advice.
38	37	Low	The framework must offer a weaving function which locates aspects, analysing the defined point cuts and ensures that the correct advice is executed at the join point when the point cut circumstance is fulfilled.

A3. CMS Use Cases

This appendix is a continuation of the requirements analysis from chapter five and defines the use cases for the CMS.

A3.1 Actors

The CMS has only two actors, the first is the website owner. The website owner is a user who is using the CMS to create their website. They have access to the administrative portal and are able to manage the website layout and content. This actor will hereafter be referred to as the user.

The second actor is a website visitor who is a normal user of the Internet visiting the website created with the CMS. They do not have access to the administrative portal. This actor will hereafter be referred to as the end user.

A3.2 Pages and Posts

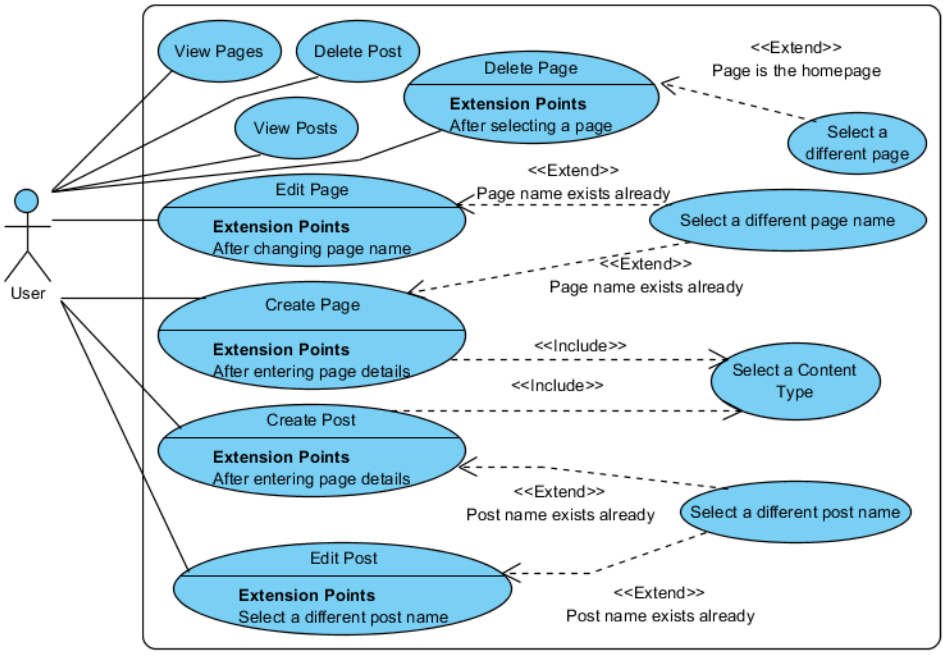

Figure 28 Use case diagram for pages and posts

Create a page

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects create page
2. User enters page details
3. User selects a content type
4. User selects an arrangement
5. System presents a form with fields extracted from the chosen content type
6. User fills out form
7. System creates the page

Extensions

2a. Page name exists already

    1. User must enter another page name

Create a Post

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects create post
2. User enters post details
3. User selects a content type
4. System presents a form with fields extracted from the chosen content type

5. User fills out form
6. System creates post

Extensions

2a. Post name exists already

1. User must enter another post name

Edit a Page

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System displays list of pages
2. User selects the page to edit
3. System retrieves page details
4. Users edits page details
5. User edits values for content type attributes
6. System updates page

Extensions

4a. The user changes the page name to one that exists already

1. User must enter a different page name

Edit a Post

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System displays list of posts
2. User selects the post to edit
3. System retrieves post details
4. Users edits post details
5. User edits values for content type attributes
6. System updates post

Extensions

4a. The user changes the post name to one that exists already

1. User must select a different post name

Delete a Page

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System displays a list of all pages
2. User selects the page to delete
3. System deletes the page

Extensions

2a The selected page is the home page

1. The user must select a different page to delete

Delete a Post

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System displays a list of all posts
2. User selects the post to delete
3. System deletes the post

View Pages

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view pages
2. System presents a list of all pages

View Posts

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view posts
2. System presents a list of all posts

A3.3 Content Types

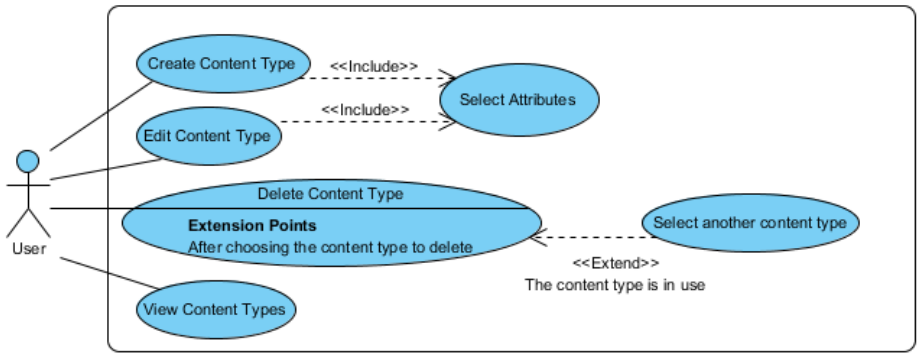

Figure 29 Use case diagram for content types

Create a Content Type

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects create content type
2. User enters content type name
3. User selects attributes for the content type

4. System saves content type

Extensions

2a The name is already in use

1. The user enters a different name

Edit Content Type

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. The system presents a list of content types
2. The user chooses the content type to edit
3. The system retrieves the content type to be edited
4. The user edits the name
5. The user edits the attributes
6. The system saves the content type

Extensions

4a The edited name is already in use

1. The user must enter a different name

Delete Content Type

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. The system presents a list of all content types
2. The user selects the content type to delete
3. The system deletes the content type

Extensions

2a The content type is in use

    1. The user must select another content type to delete

View Content Types

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. The user selects view content types
2. The system presents a list of all content types

A3.4 Layouts and Arrangements

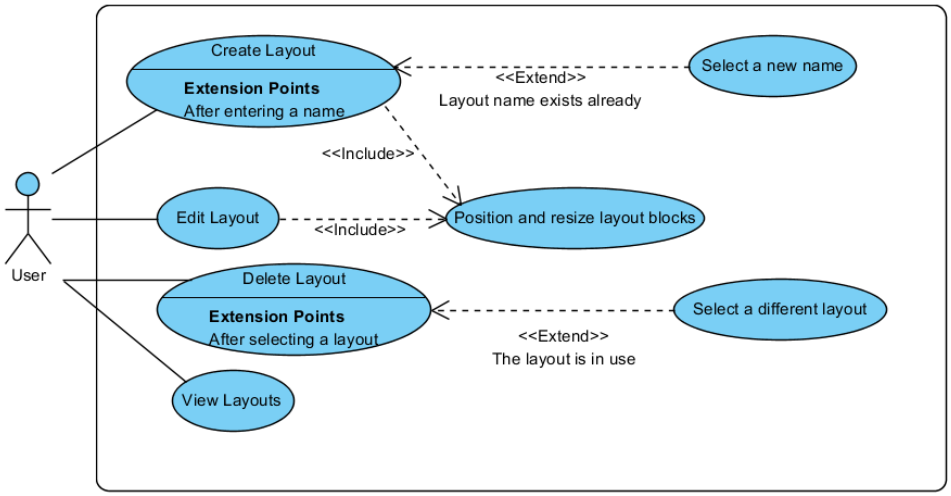

Figure 30 Use case diagram for layouts

Create Layout

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects create layout
2. User enters a name for the layout
3. User positions and resizes layout blocks
4. System saves layout

Extensions

2a The name for the layout already exists

1. User enters a different name

Edit Layout

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents a list of all layouts
2. User selects a layout to edit
3. User positions and resizes layout blocks
4. System saves layout

Delete Layout

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents a list of layouts
2. User selects the layout to delete
3. System deletes layout

Extensions

2a The layout is in use

    1. The user selects another layout to delete

View Layouts

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view layouts
2. System presents a list of all layouts

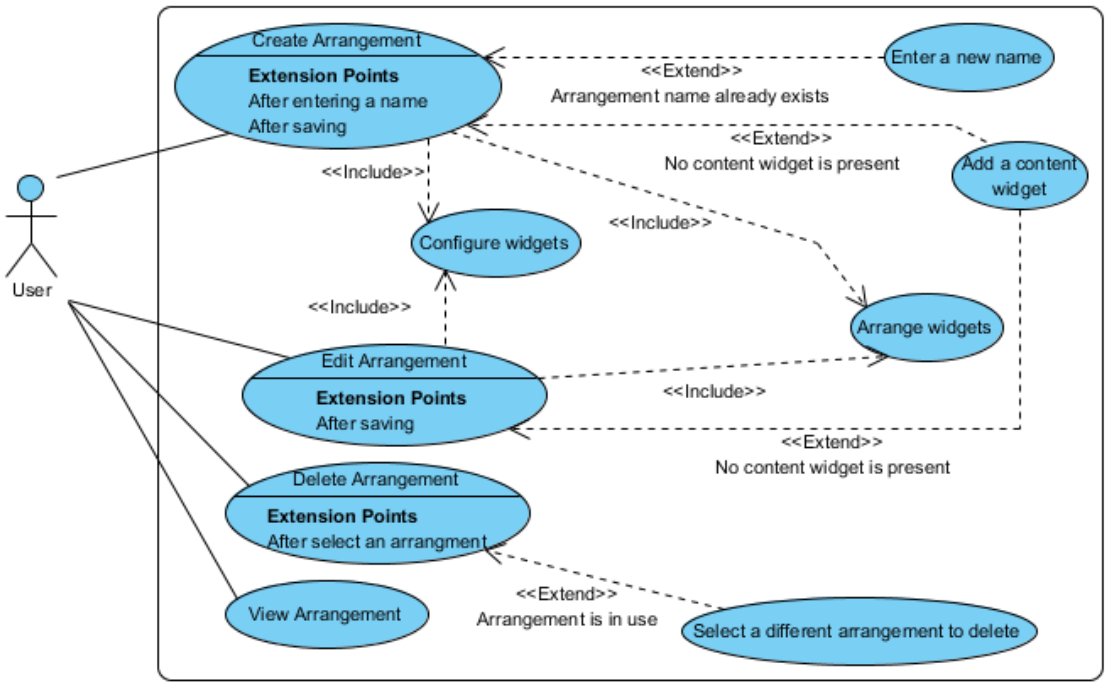

Figure 31 Use case diagram for arrangements

Create Arrangement

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. Users selects create arrangement
2. User enters a name

3. User selects the layout on which to create the arrangement
4. System displays the layout blocks
5. User places and arranges widgets on the layout blocks
6. User configures widgets
7. System saves arrangement

Extensions

2a The arrangement name is in use

1. The user enters another name

7a The user has not added a content widget

1. The user adds a content widget

Edit Arrangement

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents a list of all layouts
2. User selects a layout for which to edit arrangements
3. System presents a list of all arrangements for the chosen layout
4. User selects an arrangement to edit
5. User places and arranges widgets on the layout blocks
6. User configures widgets
7. System saves arrangement

Extensions

7a The user has not added a content widget

1. The user adds a content widget

Delete Arrangement

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents a list of all layouts
2. User selects a layout for which to delete an arrangement
3. System presents a list of all arrangements for the chosen layout
4. User selects an arrangement to delete
5. System deletes arrangement

Extensions

4a The arrangement is in use

1. The user selects a different arrangement

View Arrangements

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view layouts

2. System presents a list of all layouts
3. User selects a layout for which to edit arrangements
4. System presents a list of all arrangements for the chosen layout

A3.5 Themes

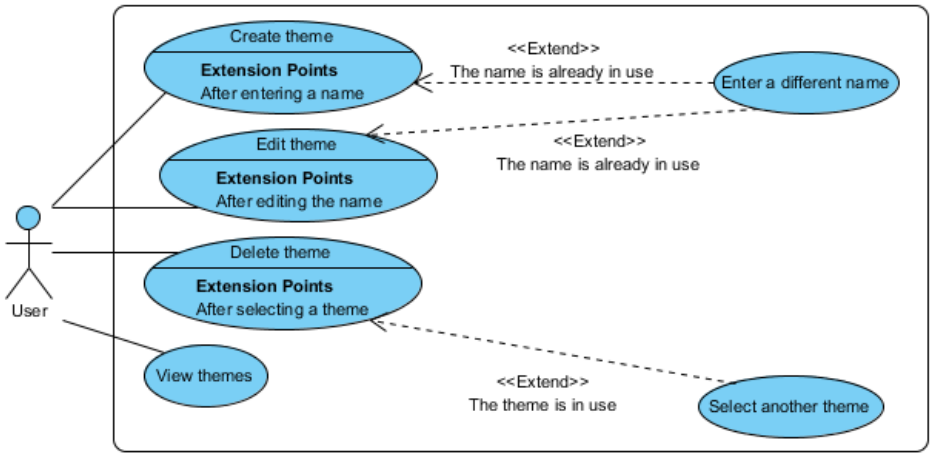

Figure 32 Use case diagram for themes

Create Theme

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects create theme
2. User enters a theme name
3. User configures theme attributes
4. System saves theme

Extensions

2a The theme name is already in use

1. User enters a different name

Edit Theme

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. Systems presents a list of all themes
2. User selects the theme to edit
3. User edits theme name
4. User edits theme attributes
5. System saves theme

Delete Theme

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents a list of all themes
2. User selects theme to delete
3. System deletes theme

Extensions

2a The theme is in use

    1. User selects another theme to delete

View Themes

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view themes
2. System presents a list of all themes

A3.6 Forms and Submissions

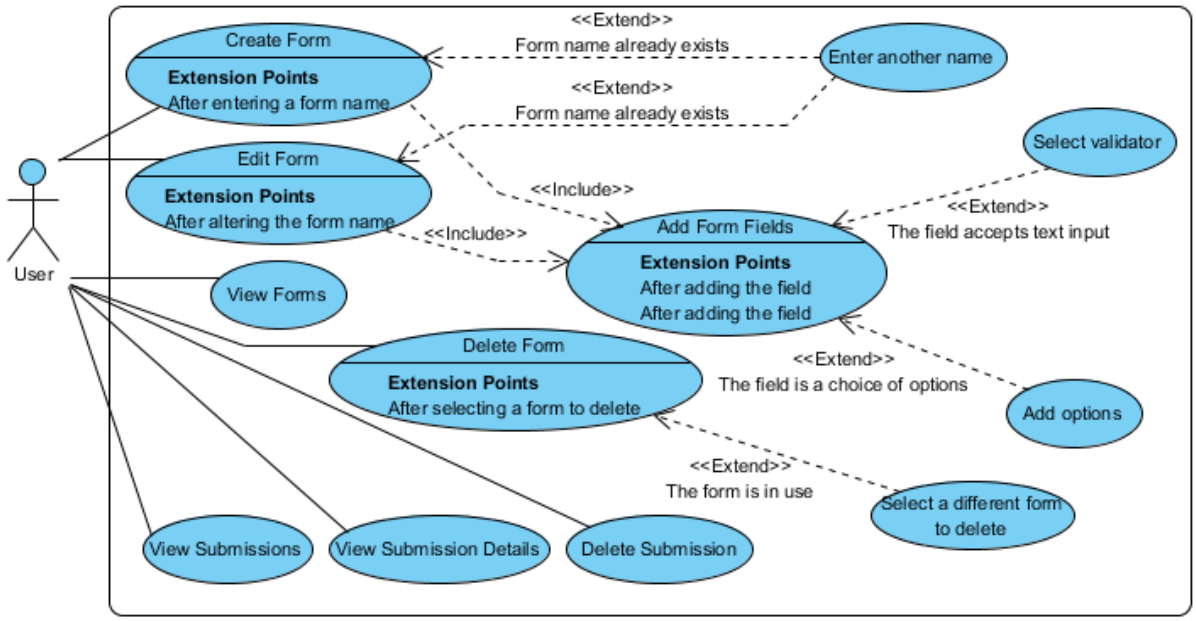

Figure 33 Use case diagram for forms and submissions

Create Form

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects create form
2. User enters form name
3. User selects form submission action
4. User places form fields
5. User configures form fields (user enters a field name and selects mandatory/optional)
6. System saves form

Extensions

2a The form name already exists

1. User must enter a different name

5a The form field is a text input field

1. User selects a text validator
2. User enters validation options

5b The form field is a choice from a list of options

1. User enters options

Edit Form

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents a list of all forms
2. User selects the form to edit

3. User edits form name
4. User edits form submit action
5. User places or removes form fields
6. User configures form field
7. System saves form

Extensions

3a The form name already exists

2. User must enter a different name

6a The form field is a text input field

3. User selects a text validator
4. Users enters validation options

6b The form field is a choice from a list of options

2. User enters options

Delete Form

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. Systems presents a list of all forms
2. User selects the form to delete
3. System deletes form

Extensions

2a The form is in use

1. User selects a different form to delete

View Form

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view forms
2. System present a list of all forms

View Submissions

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view submissions
2. System presents a list of all form submissions

View Submission Details

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents a list of all form submissions

2. User selects a submission to view
3. System displays the details of the selected submission

Delete Submission

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view submissions
2. System presents a list of all submissions
3. User selects submission to delete
4. System deletes submission

A3.7 Users

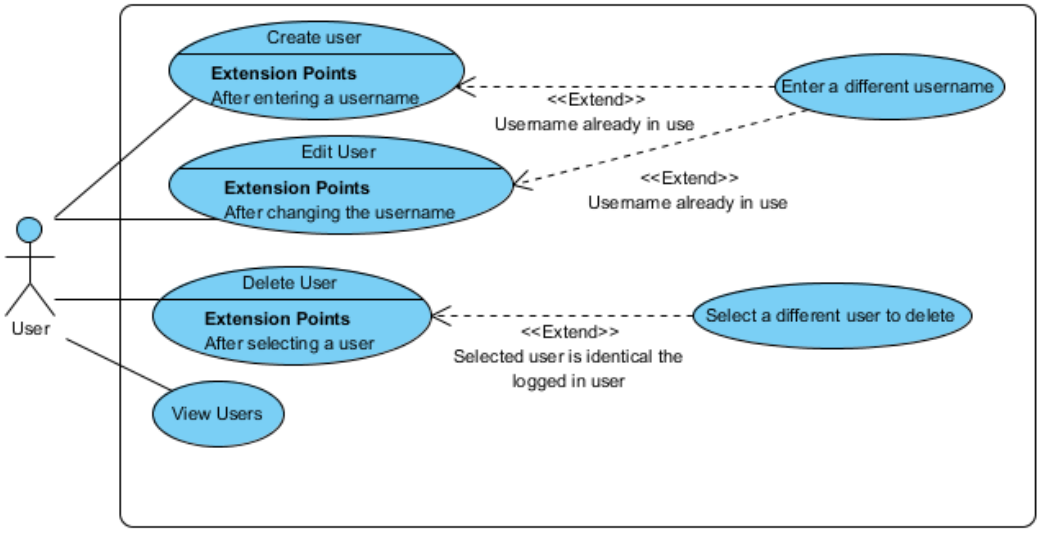

Figure 34 Use case diagram for users

Create User

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects create user
2. User enters username and password
3. System saves user

Extensions

2a Username is already in use by another user

1. User must enter a different username

Edit User

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents list of all users
2. User selects the user to edit
3. User enters a new username and/or new password
4. System saves edited user

Extensions

3a The edited username is in already in use

1. The user must enter a different username

Delete User

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. System presents a list of all users
2. User selects the user to delete
3. System deletes the user

Extensions

2a The selected user is the same as the currently logged in user

1. The user must select another user to delete

View Users

Actors: The user

Preconditions: The user has logged in to the portal

Main Success Scenario

1. User selects view users
2. System presents a list of all users

A3.8 Front End

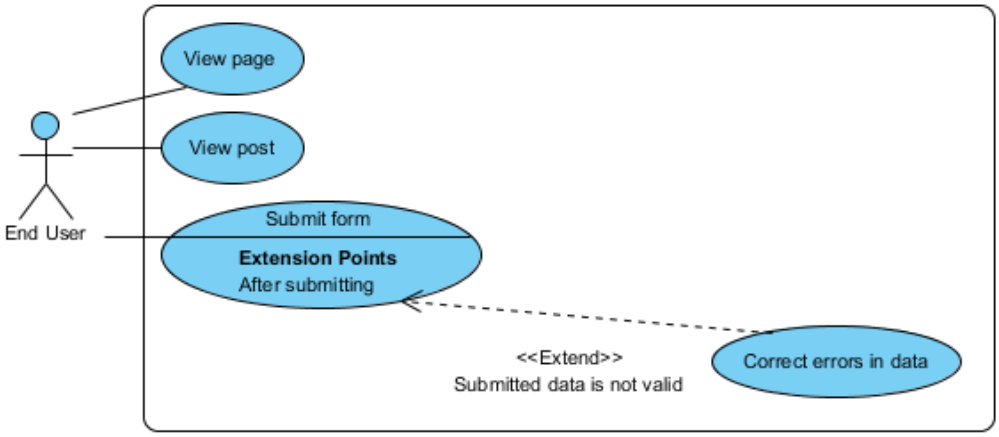

Figure 35 Use case diagram for the front end

View Page

Actors: The end user

Preconditions: None

Main Success Scenario

1. End user navigates to a page
2. System retrieves page content, arrangement and theme, and generates the page

View Post

Actors: The end user

Preconditions: None

Main Success Scenario

1. End user navigates to a page for a post
2. System retrieves post content, arrangement and theme, and generates the page

Submit Form

Actors: The end user

Preconditions: The end user has navigated to a page which contains a form

Main Success Scenario

1. The end user fills out the form and selects submit
2. The system saves the submissions and performs the form submit action

Extensions

1a The submitted form data does not comply with the imposed validation rules

1. The end user must correct the data and resubmit the form

A4. Test Specification

This appendix lists the test plan for both the framework and the CMS. The framework test plan is grey box and the CMS test plan is black box, both have been extracted from the requirements defined in chapter five, appendix two and appendix three.

A4.1 Framework Grey Box Test Plan

The tests for the framework are derived from the functional and non-functional requirements the framework is expected to fulfil. The test plan is to be run on the sample application developed using the framework. This application includes each of the visual components, model types and request scenarios specified in appendix two. To fully test for exceptions and more fundamental functionality, grey box testing is employed, whereby user and extension classes are to be written specifically for tests. As these are grey box tests, some have been written alongside design and implementation – although they refer back to the functionality requirements.

All tests are expected to be atomic. Unless specified otherwise, all code alterations in the described tests are to be undone after completing the test.

Test Block 1: Associated mark-up and component hierarchies

Test	Description	Action	Expected
1a	Webpage component with associated mark-up	Create a webpage extension class and HTML file as in listing 1a. Navigate to the page	HTML identical to the associated HTML is returned
1b	Webpage component without associated mark-up	From test 1a, remove the HTML file. Navigate to the page	Mark-up not found error is produced
1c	Webpage component with invalid associated mark-up	Recreate the HTML file for the webpage class, but remove the closing HTML tag. Navigate to the page	Invalid mark-up error is produced
1d	Child components are provided with their mark-up	Replace the test page class and HTML with listing 1b, creating a DIV tag and MarkupContainer. Navigate to the page	HTML identical to the HTML file is returned. This HTML includes the new DIV tag
1e	Child components do not have mark-up	From 1d, remove the picon:id attribute from the DIV. Navigate to the page	Missing picon:id error is produced
1f	Child components that do not exist	From 1d, remove the addition of the MarkupContainer to the page. Navigate to the page	Missing component for element with picon:id element is produced
1g	Creation of complex hierarchies	Update the test page class and HTML to match listing 1c, creating more DIV tags	The HTML identical to the updated HTML is returned,

		with MarkupContainers. Navigate to the page	including the new DIV tags.
1h	Invalid hierarchies	From 1g, change the MarkupContainer for element3 so it is added to the page and not element2. Navigate to the page	Missing component for element with picon:id element3 is produced
1i	Invalid hierarchies	Repeat 1h, for each element in turn.	Missing component for the element in question error is produced
1j	Mark-up overriding	Restore the test page class and HTML to listing 1a. Add the sub test page class and HTML as in listing 1d	The sub class HTML is returned
1k	Mark-up inheritance	From 1j, delete the sub class HTML. Navigate to the sub class page	The parent class HTML is returned
1l	Mark-up merging	From 1j, Immediately after the `</h1>` in TestPage.html add `<picon:child/>` Wrap the H2 in the sub test page HTML in `<picon:extend>` tags. Navigate to the sub class page	Merged HTML is returned. The contents of picon:extend from the sub class has been inserted in the location of the picon:child tag in the parent class HTML
1m	Panel associated mark-up	Create a panel extension class with an associated HTML file as shown in listing 1e. Restore the test page class and HTML to listing 1b. Change `new MarkupContainer` to `new TestPanel`	The content of the panel is displayed inside the DIV tag
1n	Invalid panel mark-up	Remove the `</html>` from the test panel HTML	Invalid mark-up error is produced
1o	No panel mark-up	Delete the test panel HTML file	Mark-up not found error is produced
1p	No picon:panel tags	Restore the test panel HTML to listing 1e and delete the picon:panel tags	Missing picon:panel tags error is produced
1q	Border associated mark-up	Create the test border as in listing 1f. Restore the test page class and HTML to listing 1b. Change `new MarkupContainer` to `new TestBorder` and add some text into the DIV tag	The HTML from the test page is shown but the text from the DIV tag now appears after “Before” and before “After”
1r	No border mark-up	Delete the test border HTML	Mark-up not found error is produced
1s	Invalid border mark-up	Remove the `</html>` from the border mark-up	Invalid mark-up error is produced
1t	No picon:border tags	Remove the picon:border tags from the test border HTML	Missing picon:border tags error is produced
1u	Missing picon:body tag	Remove the picon:body tag	Missing picon:body tag error is produced
1v	Missing body content	Remove the `$this->add($this->getBorderBody());` from the border class	Missing body component error is produced
1w	Duplicate component	Restore test page to listing 1b, duplicate the `$this->add(new MarkupContainer('element'));`	Component with name element exists already error is produced

Listing	Files	PHP	HTML

1a	TestPage.php
	TestPage.html
		`class TestPage extends WebPage`
		`{`
		`}`
			`<html><head><title>Test page</title></head><body><h1>Hello world!</h1></body></html>`

1b	TestPage.php
	TestPage.html
		`class TestPage extends WebPage`
		`{`
		`  public function __construct()`
		`  {`
		`  parent::__construct();`
		`  $this->add(new MarkupContainer('element'));`
		`  }`
		`}`
			`<html><head><title>Test page</title></head><body><h1>Hello world!</h1><div piocon:id="element"></div></body></html>`

1c	TestPage.php
	TestPage.html
		`class TestPage extends WebPage`
		`{`
		`  public function __construct()`
		`  {`
		`  parent::__construct();`
		`  $element1 = new MarkupContainer('element1');`
		`  $this->add($element1);`
		`  $element2 = new MarkupContainer('element2');`
		`  $element1->add($element2);`
		`  $element3 = new MarkupContainer('element3');`
		`  $element2->add($element3);`
		`  }`
		`}`
			`<html><head><title>Test page</title></head><body><h1>Hello world!</h1><div piocon:id="element1"><div piocon:id="element2"><div piocon:id="element3"></div></div></div></body></html>`

1d	SubTestPage.php
	SubTestPage.html
		`class SubTestPage extends TestPage`
		`{`
		`}`
			`<html><head><title>Test page</title></head><body><h2>sub page</h2></body></html>`

1e	TestPanel.php
	TestPanel.html
		`class TestPanel extends Panel`
		`{`
		`}`
			`<html><head><title>Test panel</title></head><body><picon:panel><h1>test panel</h1></picon:panel></body></html>`

1f	TestBorder.php TestBorder.html	`class TestBorder extends Border` `{` `  public function __construct($id)` `  {` `  parent::__construct($id);` `  $this->add($this->getBorderBody());` `  }` `}`	`<html><head><title>Test page</title></head><body><body><picon:border>Above <picon:body /> Below</picon:border></body></body></html>`

Test Block 2: Models

Test	Description	Action	Expected
2a	Test basic model	Navigate to the label page of the sample application	The label text is shown.
2b	Altering basic model	Open the label webpage extension class and alter the value passed to the basic model to *test label*. Reload the page	The label text changed successfully
2c	Basic model from a variable	Change the input of the basic model to a variable declared with value of *text*. After instantiating the model, alter the variable to a value of *different*. Reload the page	The label text reads *text*
2d	Test property model	Change the model of label to use a property model, create a text property in the label page class, and set its value to *label text*. Set the property to use the	The label text is shown

		new property. Reload the page	
2e	Test value alteration	After instantiating the property model, change the value of the property to *different.* Reload the page	The label text reads *different*
2f	Test array model	Navigate to the list page of the sample application	The list of fruit is displayed
2g	Test altered array model	Add an additional fruit to the array, before the model is instantiated. Reload the page	The new fruit is shown
2h	Test altered array after model creation	After the model is instantiated, add a new fruit to the array. Reload the page	The new fruit is shown
2i	Test compound property model	Navigate to the form fields page	The values of the Example Domain object are used to populate the value of the form field
2j	Test adding additional property	Add an additional property to the Example Domain. Reload the page	The page remains unchanged and the new property is ignored
2k	Test property removal	Remove the textbox property	Property textbox does not exist in example domain error is shown

Test Block 3: Request Scenarios

Test	**Description**	**Action**	**Expected**
3a	Standard page request	Navigate to the sample application, with no additional path in the URI	The default home page is shown.
3b	Standard page request with path	Append the 3a URI with /LabelPage, navigation to the new URI	The label page is shown
3c	Stateless callback request URL generation	Navigate to the link page. Review the value of href on link one	The link page is shown and the href value is: *LinkPage?listener=alterLink:* comprising the page name and link component path
3d	Stateless callback request re-rendering	Click on link one	The same page reloads but with altered text
3e	Stateless callback request internal redirection	Click on link two	The home page is loaded
3f	Stateful callback request URL generation	Navigate to the validation page. Review the action attribute of the form	The form page is shown and the action attribute is *ValidationPage?pageid=page126 &listener=form:* comprising the page name (for reference only), the page id and component path
3g	Stateful callback request re-rendering	Click submit	The form is submitted. The page will update with error messages as the form fields were left blank
3h	Stateful callback request internal redirection	Alter the button callback on the validation page to direct to the homepage in the onError callback. Navigate back to the homepage and then back to the validation page. Click submit	The form is submitted, the homepage is loaded
3i	Ajax callback	Navigate to the Ajax link	The Ajax link page is shown. The href is

	request URL generation	page. Review the href attribute and the URL from the onclick attribute	*javascript:;* the onClick is *AjaxLinkPage?pageid=page129 &listener=alterLink:& behaviour=behaviour_1&ajax=ajax*
3j	Ajax callback request action	Click on the Ajax link. Review the response (not visible on the page, use firebug)	The page is updated but not reloaded in anyway. The response is JSON and contains component id/HTML pairs.
3k	Resource request URL generation	Navigate to the Ajax link page. Review the mark-up	The header should contain a JavaScript reference to *?picon-resource=picon.AbstractJQueryBehaviour:jquery.js*
3l	Resource request	Navigate to the resource URL	The jQuery JavaScript is returned.

Test Block 4: General Components

Test	Description	Action	Expected
4a	List mark-up iterator	Navigate to the list page. Review the HTML	The li tag should be repeated for each list item
4b	List callback invocation	Remove the code within the list callback function in the list page class to confirm the callback is being run	A component not found for name error is produced
4c	Repeater mark-up duplicator	Navigate to the list page. Review the HTML	The DIV tags should be repeated for each child component added to the repeater
4d	Repeater child component id	Alter the list page, set output mark-up to true for each repeater child component. Reload the page and review the HTML	The DIV tags should each have a unique and incrementing id value
4e	Label component text	Navigate to the label page	The text “Label text” is shown
4f	Label replacement text	Alter the label page HTML, change the text within the `<span picon:id=”label”>` to any value. Reload the page	The new value is not shown, but replaced by the label text
4g	Tab Panel list	Navigate to the tabs page	Both of the sample tabs are shown
4h	Adding a new tab	Alter the tab page class to include a new tab. Reload the page	The new tab is shown on the list
4i	Removing all tabs	Alter the tab page class, removing all tab additions. Reload the page	The tab list is empty and nothing is shown
4j	Tab links	Restore the tab page class and reload the page. Click on each of the tabs one after the other.	The panel corresponding to the tab name is shown when the link is clicked.
4k	Link callback execution	Navigate to the link page, click on the first sample link	The same page should reload but with the value of a label altered to read *Update in callback text*
4l	Link callback execution	Navigation to the link page. Click on the second sample link	The home page should be loaded
4m	Feedback panel visibility	Navigate to the validation page. Review the HTML	An empty DIV should be located at the top, confirming the feedback panel is present but does not yet contain any messages
4n	Feedback message	On the validation page, submit the form	Error messages should appear

	rendering		in the feedback panel, for each field
4o	Info messages	In the validate page class, alter the submit button invalid callback to render an info message. Navigate to the validation page and submit the form	The new info message is shown
4p	Success messages	In the validate page class, alter the submit button invalid callback to render a success message. Navigate to the validation page and submit the form	The new success message is shown
4q	Warning messages	In the validate page class, alter the submit button invalid callback to render a warning message. Navigate to the validation page and submit the form	The new warning message is shown

Test Block 5: Security and Authentication

Test	Description	Action	Expected
5a	Picon directory web protection	Navigate to the sample application homepage and add /picon to the URL	Error 403
5b	Assets directory web protection	Navigate to the sample application homepage and add /assets to the URL	Error 403
5c	Component authorisation	Navigate to the authorised page	Redirection to the login page
5d	Satisfy authorisation requirements	Navigate to the login page (attempting to go to the authorised page will achieve this) Click on the login link to authorise the session. Navigate to the authorised page again	The authorised page is shown.
5e	Authorisation persistence	Navigate back to the home page and then back to the authorised page.	The authorised page is shown
5f	Revoke access	Navigate to the authorised page and click revoke. Navigate back to the authorised page	The login page is shown

Test Block 6: Form Components

Test	Description	Action	Expected
6a	Form component association	Navigate to the form field page of the sample application	The form page is shown
6b	Form tag validation	Alter the form page, change the form tag to a SPAN	Form component must be added to a form tag error is produce
6c	Form attribute addition	Review the HTML of the form page	The form has had the method attribute set to POST and a method attribute set to a stateful callback URL
6d	Form field processing	Navigate to the form page; alter each of the form fields. Click submit	The form page should reload and the changes will be reflected in the read-out at the bottom of the page

6e	Valid callback invocation	Alter the form page class, add an *onSubmit* callback that creates a success message and an *onError* that creates an invalid message. Change *TextField* to be mandatory. Navigate to the form page. Fill out the text field and press submit	The success message is shown
6f	Invalid callback invocation	From 6e, navigate to the form page, remove all text from the text field, press submit	The invalid message is shown
6g	Text field tag validation	Alter the form page HTML, change the textbox to a SPAN tag. Navigate to the form page	Text field must be added to an input tag error is produced
6h	Text field attribute validation	Alter the form page HTML, change the attribute type value to radio. Navigate to the form page	Text field must have a type value of text error is produced
6i	Text field model object type validation	On the form page, alter the data type of the text box, set it to boolean. Reload the page	Text field needs boolean model error is produced
6j	Updated model validation	From 6i, update the example domain textbox value to be a boolean true. Reload the page	True is shown in the text field
6k	Text field model object type validation	On the form page, alter the data type of the text box, set it to integer. Reload the page	Text field needs integer model error is produced
6l	Updated model validation	From 6k, update the example domain textbox value 14. Reload the page	14 is shown in the text field
6m	Text field model object type validation	On the form page, alter the data type of the text box, set it to double. Reload the page.	Text field needs double model error is produced.
6n	Updated model validation	From 6m, update the example domain textbox value to be 4.4. Reload the page	4.4 is shown in the text field
6o	Mandatory fields	Navigate to the validation page. Leave all the fields blank, submit the form	A field required error is displayed for each one
6p	E-mail address validation	Enter *invalidEmail* into the email field, submit the form	An invalid email address error is displayed
6q	E-mail address validation	Enter *validemail@example.com* in the email address field, submit the form	No message is displayed for the email field
6r	Numerical validation	Enter *16* in the number less than 14 field, submit the form	Number must be less than 14 error is displayed
6s	Numerical validation	Enter *12* in the number less than 14 field, submit the form	No message is displayed for the email field
6t	Numerical validation	Enter *notanumber* into the number less than 14 field. Submit the form	Not a valid number error is displayed
6u	Numerical validation	Enter *12* in the number greater than 34 field. Submit the form	Number must be greater than 34 error is displayed
6v	Numerical validation	Enter *47* in the number greater than 34 field. Submit the form	No message is displayed for the field
6w	Numerical validation	Enter *notanumber* into the number greater than 34 field. Submit the form	Not a valid number error is displayed
6x	Numerical validation	Enter *3* in the number greater than 4 and less than 45 field. Submit the form	Number must be greater than 4 and less than 45 error is displayed
6y	Numerical validation	Enter *47* in the number greater than 4 and less than 45 field. Submit the form	Number must be greater than 4 and less than 45 error is displayed
6z	Numerical validation	Enter 35 in the number greater than 4	No message is displayed for the

		and less than 45 field. Submit the form	field
6aa	Numerical validation	Enter *notanumber* into the number greater than 4 and less than 45. Submit the form	Not a valid number error is displayed
6ab	String length validation	Enter *morethantentext* into the string less than 10 field	Length must be less than 10 error is displayed
6ac	String length validation	Enter *lessthan* into the string less than 10 field. Submit the form	No message is displayed for the field
6ad	String length validation	Enter *les* into the string more than 4 field. Submit the form	Length must be greater than 4 error is displayed
6ae	String length validation	Enter *morethanfour* into the string more than 4 field. Submit the form	No message is displayed for the field
6af	String length validation	Enter *les* into the string between 4 and 10 field. Submit the form	Length must be between 4 and 10 error is displayed
6ag	String length validation	Enter *toomanycharacterstext* into the string between 4 and 10 field. Submit the form	Length must be between 4 and 10 error is displayed
6ah	String length validation	Enter *enough* into the string between 4 and 10 field. Submit the form	No message is displayed for the field
6ai	Text area tag validation	On the form fields page, alter the HTML of the text area, change it to a SPAN. Navigate to the page	Text area may only be added to a text area tag error is produced
6aj	Text area model validation	Alter the text area property of example domain, change it to an integer. Reload the form page	Text area needs a string model error is produced
6ak	Mandatory text area	Navigate to the validation page, fill in the text area, submit the form	No error message for the text area is produced
6al	Drop down tag validation	Alter the form page HTML, change the drop down tag to a SPAN. Navigate to the page	Drop down may only be added to a select tag error produced
6am	Choice rendering strings	Alter the choices array in the form page class to contain 4 different strings. Reload the page	The drop down reflects the new choice array
6an	Choice rendering integers	Alter the choice array in the form page class to contain 4 different integers. Reload the page	The drop down reflects the new choice array, showing integers
6ao	Choice rendering doubles	Alter the choice array in the form page class to contain 4 different doubles. Reload the page	The drop down reflects the new choice array, showing the doubles
6ap	Choice rendering objects	Alter the choice list to contain any 4 objects. Reload the page	Unable to find converter error is produced
6aq	Choice rendering objects	Alter the drop down, add a renderer callback to generate strings from the objects. Reload the page	The objects in the choice are shown as strings, generated using the callback method
6ar	Mixed choice arrays	Alter the choice array to contain an integer and a string	Choice array does not contain the same value error is produced
6as	Model type validation	Restore the choice array to their original strings. Alter the drop down property of the example domain, set to an integer. Reload the page	Model does not match choice type error is produced
6at	Checkbox tag validation	Alter the form page HTML, change the checkbox to a SPAN tag. Navigate to the form page	Checkbox must be added to an input tag error is produced
6au	Checkbox attribute validation	Alter the form page HTML, change the attribute type value to radio. Navigate	Checkbox must have a type value of checkbox error is

		to the form page.	produced.
6av	Checkbox model validation	Alter the example domain, set the checkbox property to a string. Reload the form page	Checkbox must have a boolean model error is produced
6aw	Checkbox state rendering	Set the checkbox property in the example domain class to true. Reload the form page	Checkbox renders checked
6ax	Checkbox state rendering	Set the checkbox property in the example domain class to false. Reload the form page	Checkbox renders unchecked
6ay	Checkbox group model validation	Alter example domain, set the checkbox group property to a string	A checkbox group must have an array model error is produced
6az	Checkbox group array insertion	Navigate to the form page, check the first two options in the checkbox group. Submit the form	The page reloads, the first two checkboxes are ticked and the model read-out indicates the first two options are in the array
6ba	Checkbox group array insertion	From 6az, reverse the checkbox selection and submit the form	The page reloads, the last two checkboxes are ticked and the model read-out indicates the last two options are in the array
6bb	Radio tag validation	Alter the form page HTML, change the radio to a SPAN tag. Navigate to the form page	Radios must be added to an input tag error is produced
6bc	Radio attribute validation	Alter the form page HTML, change the attribute type value to checkbox. Navigate to the form page	Radios must have a type value of radio error is produced
6bd	Radio group wrapping	Alter the HTML and the class of form page, adding a new radio directly to the form. Reload the page	A radio must only be used inside of a radio group error is produced
6be	Radio group model validation	Alter the example domain, set the radio group property to be an array	Radio group does not support array error is produced
6bf	Radio state rendering	Navigate to the form page, choose the first radio in the radio group, submit the form	The page reloads, the first radio is still selected and the model read-out indicates the value of the first radio is selected
6bg	Radio state rendering	Navigate to the form page, choose the third radio in the radio group, submit the form	The page reloads, the third radio is still selected and the model read-out indicates the value of the third radio is selected
6bh	Check choice model validation	Alter the check choice property in example domain, set it as any string	A check choice requires an array model error is produced
6bi	Check choice generation	Alter the check choice array in the form page class to include some new elements. Reload the page	The list of choices in the check choice reflects the new options in the list
6bj	Check choice state rendering	Alter the example domain, set the check choice property to be an array with a single element: the string of the third choice value (*other option*). Reload the page	Only the third checkbox is checked
6bk	Check choice state rendering	From 6bj, add the value of the first checkbox to the example domain check choice property array. Reload the page	The first and third checkboxes are checked
6bl	Check choice state persistence	Navigate to the form page. Check the second and fourth checkboxes and	The page reloads, the second and fourth checkboxes are still

		submit the form	selected and the model read-out indicates the second and fourth values are in the model array
6bm	Radio choice model validation	Alter the radio choice property in example domain, set it as an array	A radio choice cannot use an array error is produced
6bn	Radio choice generation	Alter the radio choice array in the form page class to include some new elements. Reload the page	The list of choices in the radio choice reflects the new options in the list
6bo	Radio choice state rendering	Alter the example domain, set the radio choice property to be the third option value (*other option*). Reload the page	The third radio in the radio check is selected
6bp	Radio choice state rendering	Alter the example domain, set the radio choice property to be the first option value (*default*). Reload the page	The first radio in the radio check is selected
6bq	Radio choice state persistence.	Navigate to the form page. Check the fourth radio. Submit the form	The page reloads, the fourth radio is still selected and the model read-out indicates the fourth value is the current model object
6br	Button tag name validation	On the form page HTML, alter the submit button tag, change it to button. Navigate to the page	The form page is shown
6bs	Button tag name validation	On the form page HTML, alter the submit button tag, change it to SPAN. Navigate to the page	Button may only be added to an input or button tag error is produced
6bt	Button attribute validation	On the form page HTML, alter the submit button, set the type to *text*. Navigate to the page	Button can only be added to an input type with a type of text error is produced
6bu	Button field name	Review the HTML of the button	The submit button name attribute should be the component path *form:button:*
6bv	Form button delegation	Navigate to the form page, click the first form button	The page reloads and produces the message “button one pressed, valid form”
6bw	Form button delegation	Navigate to the form page, click the second form button	The page reloads and produces the message “button two pressed, valid form”
6bx	Form button delegation	Alter the text field on the form page, make it mandatory. Reload the page, click the first button	The page reloads and produces the message “button one pressed, invalid form”
6by	Form button delegation	Continue from 6bx, click the second button	The page reloads and produces the message “button two pressed, invalid form”

Test Block 7: Ajax Components

Test	**Description**	**Action**	**Expected**
7a	Ajax resource inclusion	Navigate to the Ajax link page. Review the HTML	The jQuery JavaScript and Picon JavaScript has been referenced in the head
7b	Ajax link DOM event	On the Ajax link page, review the HTML of the link	The href should contain *javascript:;* and the onClick should contain some JavaScript

			invoking an Ajax get function
7c	Execution of Ajax link callback	On the Ajax link page, click on the Ajax link	The label is updated but the page does not reload
7d	Ajax button DOM event	Navigate to the Ajax button page	The onClick should contain some JavaScript invoking an Ajax post function
7e	Ajax button submit override	On the Ajax button page, alter the HTML giving the submit button a type attribute of *submit*, reload the page and review the HTML for the submit button	The submit button shown has a type of button, preventing default form submission
7f	Ajax button valid callback	Navigate to the Ajax button page, insert some new text into the form field and press submit	The label is updated to reflect the value of the text field, the page did not reload, confirming the form data was posted via Ajax
7g	Ajax button invalid callback	Navigate to the Ajax button page and remove all text from the text field. Submit the form	An error message appears indicating that the field may not be empty. The page does not reload

Test Block 8: Data Table and Provider

Test	**Description**	**Action**	**Expected**
8a	Data table row and column rendering	Navigate to the data table page. Review the HTML	TD elements are repeated with repeating TR elements for each record
8b	Suitable pagination option generation	Navigate to the data table page	A list of 3 links, 1 for each page, is shown
8c	Suitable pagination option generation	On the data table page, click the 2 link	The second page of results is shown
8d	Suitable pagination option generation	On the data table page, click the 3 link	The last page of results is shown
8e	Suitable pagination option generation	Alter the data table page, set the rows per page to 100. Reload the page	No pagination options should appear, all records appear on the first and only page
8f	Column heading rendering	Navigate to the data table page. Review the table HTML	Column headings for each column are present in TH tags.
8g	Column heading alteration	Alter the data table page; change each of the column headings. Reload the page	The column headings reflect the change
8h	Data provider record count	Alter the example data provider, set the return value of the number of records to 100. Reload the data table page	The number of pages in the page list reflects the increase of records

Test Block 9: Resource Container

Test	Description	Action	Expected
9a	Confirm automatic instantiation and injection	Create the service and repository classes as detailed in listing 9a and 9b. Create a new context load listener in 9c, which recursively prints the context and then dies. Run the application	The context is printed, showing an internal array containing two objects: an instance of resource and an instance of second resource. Each object has a property with the value of the other
9b	Confirm dependency injection into any object	From 9a, alter the context listener to inject a new object, listing 9d and create the class to use as shown in 9e. Run the application	The separate object is printed out and both properties are set to instances of resource and second resource
9c	Invalid resource declaration	Restore the listener to listing 9c. Update service so its property is called *nonExistantResource*. Run the application	A resource not found error is produced
9d	Test direct resource access	Update the content listener to listing 9f. Run the application	Both the resource and second resource objects are printed with properties containing objects of the other
9e	Test invalid direct access	Alter the listener, change the name of the resource string to *nonexistent*. Run the application	A resource not found error is produced

Listing	Files	PHP

9a	Service.php	`/**`
		`*@Service`
		`*/`
		`class Resource`
		`{`
		`  /**`
		`  *@Resource`
		`  */`
		`  private $secondResource;`
		
		`}`

9b	Repository.php	`/**`
		`*@Repository`
		`*/`
		`class SecondResource`
		`{`
		`  /**`
		`  *@Resource`
		`  */`
		`  private $resource;`
		`}`

9c	LoadListener.php	`class ApplicationContextLoadListener implements`
		`ApplicationInitializerContextLoadListener`
		`{`
		`  public function onContextLoaded(ApplicationContext $context)`
		`  {`
		`  print_r($context); die();`
		`  }`
		`}`

9d	LoadListener.php	`class ApplicationContextLoadListener implements`
		`ApplicationInitializerContextLoadListener`
		`{`
		`  public function onContextLoaded(ApplicationContext $context)`
		`  {`
		`  $separate = new Separate ();`
		`  Injector::get()->inject($separate);`

		print_r($separate); die();
		}
		}

9e	Separate.php	class Separate
		{
		/**
		*@Resource
		*/
		private $resource;
		
		/**
		*@Resource
		*/
		private $secondResource;
		}

9f	LoadListener.php	class ApplicationContextLoadListener implements ApplicationInitializerContextLoadListener
		{
		public function onContextLoaded(ApplicationContext $context)
		{
		print_r($context->getResource('resource'));
		print_r($context->getResource('secondResource)); die();
		}
		}

Test Block 10: Configuration

Test	Description	Action	Expected
10a	Valid XML	Navigate the sample application; the sample application XML will already be valid	The application runs as normal
10b	Invalid XML	Invalidate the configuration file by removing any of the close tags. Navigate to the sample application	Configuration error is produced
10c	Missing mandatory attributes	Remove the value for the homepage tag. Navigate to the sample application	Configuration error is produced
10d	Missing mandatory attributes	Remove the start-up type tag value. Navigate to the sample application	Configuration error is produced
10e	Missing mandatory attributes	Remove the mode tag value. Navigate to the sample application	Configuration error is produced
10f	Homepage value	Change the homepage to *ListPage*. Navigate to the sample application	The default page showing is the list page
10g	Homepage value	Change the homepage value to *NonExistantPage.* Navigate to the sample application	A page not found error is produced
10h	Mode value	Change mode to *nonexistent*. Navigate to the sample application	Configuration error is produced
10i	Start up value	Change the start up value to *nonexistent.* Navigate to the sample application	Configuration error is produced

Test Block 11: Database API

Test	Description	Action	Expected
11a	Configuration test	Setup a data source (*testDataSource)* in the configuration for a MySQL database. Navigate to the sample application	The sample application loads
11b	Invalid host test	Alter the data source host to *invalid*. Navigate to the sample application	A failed to connect error is produced
11c	Invalid username test	Alter the data source username to *invalid*. Navigate to the sample application	A failed to connect error is produced
11d	Invalid password test	Alter the data source password to *invalid*. Navigate to the sample application	A failed to connect error is produced
11e	Invalid database test	Alter the data source database to *invalid*. Navigate to the sample application	A failed to connect error is produced
11f	Insertion of data	Create the DAO defined in listing 11a and create the SQL table defined in listing 11b. Create the DAO test page defined in listing 11c and 11d. Navigate to the page	The label value is 1
11g	Insertion of data	From 11f, reload the page	The label value is 2
11h	Insertion of data	Alter the query within the insert method, remove *INSERT*. Reload the page	Error in query message is produced
11i	Insertion of data	Restore the query, alter the second argument, change the *3* to '*three*'	Expecting integer error is produced
11j	Updating data	Restore the files from listing 11a, 11c and 11d. Alter the test DAO page to invoke update instead of insert. Reload the page	The label value is 2
11k	Updating data	From 11j, reload the page	The label value is 0
11l	Updating data	Alter the query within the update method, remove *UPDATE*. Reload the page	Error in query message is produced
11m	Updating data	Restore the query, alter the second argument, change '*update'* to 45	Expecting string error is produced
11n	Querying data	Restore the files from listing 11a, 11c and 11d. Alter the test DAO page to invoke query instead of insert. Insert some random test data into the test table. Reload the page	The label value should show all of the values of the number column, separated by a comma
11o	Querying data	Alter the query within the update method, remove *SELECT*. Reload the page	Error in query message is produced
11p	Querying data	Restore the query, alter the row mapper to not accept the row argument	Callback must accept one argument error is produced
11q	Querying data	Restore the row mapper; alter its implementation to not return the number property of row	Row mapper may not return null error is produced

Listing	Files	Contents
11a	TestDao.php	`/** * @Repository */ class TestDao extends picon\DaoSupport { /** * @Resource(name = 'testDataSource') */ private $dataSource; protected function init() { $this->setDataSource($this->dataSource); } public function testInsert() { return $this->getTemplate()->insert("INSERT INTO `testtable` (`something` ,`number`)VALUES ('%s', %d);", array('test', 3)); } public function testUpdate() { return $this->getTemplate()->update("UPDATE `testtable` SET `something` = '%s' WHERE `number` = 3", array('update')); } public function testQuery() { $results = $this->getTemplate()->update("SELECT * FROM testtable;", new picon\CallbackRowMapper(function($row) { return $row->number; })); return implode(', ', $results); } }`
11b	-	`CREATE TABLE IF NOT EXISTS `testtable` ( `id` int(11) NOT NULL AUTO_INCREMENT, `something` varchar(255) NOT NULL, `number` int(11) NOT NULL, PRIMARY KEY (`id`) )`
11c	TestDaoPage.php	`class TestDaoPage extends picon\WebPage { /** * @Resource */ private $testDao; public function __construct() { parent::__construct(); $return = $this->testDao->insert(); $this->add(new picon\Label('label', new picon\BasicModel($return))); } }`
11d	TestDaoPage.html	`<html><head><title>Test page</title></head><body> <span picon:id=”label”>Label content</span> </body></html>`

Test Block 12: Robustness and Performance

Test	Description	Action	Expected
12a	Page load time	Make an HTTP request to the sample application 200 times	The page should load within 1 second
12b	Stateless callback load time	Extract the URL for the list page from the link on the sample application homepage. Make an HTTP request for that URL 200 times	The loading of the list page should take no longer than 1 second
12c	Stateful callback load time	Navigate to the validation page. Extract the form action URL and make an HTTP request for that URL 200 times	The loading of the page after submitting the form should take no longer than 1 second
12d	Ajax request load time	Navigate to the Ajax link page. Extract the URL from the *onClick* event of the link. Make an HTTP request for that URL 200 times	The Ajax response should take no longer than 1 second
12e	Resource request load time	On the Ajax link page, review the HTML. Extract the jQuery resource URL. Make an HTTP request for that URL 200 times	The resource should be loaded within 1 second
12f	Simultaneous request handling	Repeat tests 12a to 12e, altering the request process so the 200 requests are made simultaneously	The response should be produced within 1 second
12g	Page quantity handling	Create the 3 files given in listing 12a-12c in the assets directory. Run create.php. Navigate to the sample application	There should be no noticeable drop in quality; the page should still load in 1 second
12h	Page component hierarchy limits	Create the 4 files given in listing 12d-12g. Navigate to the test hierarchy page	The page loads within 1 second

Listing	Files	Contents

12a	TestPage.php	`abstract class TestPage extends picon\WebPage`
		`{`
		`  public function __construct()`
		`  {`
		`  parent::__construct();`
		`  $this->add(new picon\Label('test', new picon\BasicModel('page'.$this->getNumber())));`
		`  }`
		
		`  protected abstract function getNumber();`
		`}`

12b	Create.php	`$content = '<?php class TestPage%d extends TestPage { protected function getNumber() {return %d;} }';`
		
		`for($i = 0; $i< 1000;$i++)`
		`{`
		`  file_put_contents("Page$i.php", sprintf($content, $i, $i));`
		`}`

12c	TestPage.html	`<html>`
		`<head>`
		`</head>`
		`<body>`
		`<span picon:id="test"></span>`
		`</body>`
		`</html>`

12d	HierarchyTestPage.php	`class HierarchyTestPage extends picon\WebPage`
		`{`
		`  public function __construct()`
		`  {`
		`  parent::__construct();`
		`  $this->add(new HierarchyTestPanel('test'));`
		`  }`
		`}`

12e	HierarchyTest Page.html	`<html>`
		`<head>`
		
		`</head>`
		`<body>`
		`<span picon:id="test"></span>`
		`</body>`
		`</html>`

12f	HierarchyTest Panel.php	`class HierarchyTestPanel extends picon\Panel`
		`{`
		`  public function __construct($id, $index = 0)`
		`  {`
		`  parent::__construct($id);`
		`  $this->add(new picon\Label('value', new picon\BasicModel($index)));`
		
		`  if($index>100)`
		`  {`
		`  $this->add(new picon\EmptyPanel('inner'));`
		`  }`
		`  else`
		`  {`
		`  $this->add(new HierarchyTestPanel('inner', $index+1));`
		`  }`
		`  }`
		`}`

12g	HierarchyTest Panel.html	`<html>`
		`<head>`
		
		`</head>`
		`<body>`
		`<picon:panel>`
		`<span picon:id="value"></span>`
		`<span picon:id="inner"></span>`
		`</picon:panel>`
		`</body>`
		`</html>`

A4.2 CMS Black Box Test Plan

This test plan has been designed to match the requirements described in chapter five, the user cases defined in appendix three and the interface design in appendix six. A number of features are not fully implemented and tests for those features are not defined.

Test Block 1: Authentication and Authorisation

Test	Description	Action	Expected
1a	Test the login process with valid credentials	Navigate to the admin portal login page, enter a valid username and password	The system logs you in and the dashboard is displayed
1b	Test the logout process	Log in using valid credentials, click logout	The system logs you in and the login page is displayed
1c	Test the login process with invalid credentials	Navigate to the admin portal login page and enter random strings in both the username and password fields, click login	Invalid username or password message is shown
1d	Test the login process with invalid credentials	Navigate to the admin portal login page; enter a valid username but a random, invalid password. Click login	Invalid username or password message is shown
1e	Test the login process	Navigate to the admin portal login page;	Invalid username or password

	with invalid credentials	enter a valid password but a random, invalid username. Click login	message is shown
1f	Test the login process with invalid credentials	Navigate to the admin portal login page; enter a random string in username, leave password blank. Click login	Password is required message is shown
1g	Test the login process with invalid credentials	Navigate to the admin portal login page; enter a random string in password, leave username blank. Click login	Username is required message is shown
1h	Test the login process with invalid credentials	Navigate to the admin portal login page; leave both username and password blank. Click login	Username and password is required message is shown
1i	Test page authorisation	Without logging in, navigate to the layout page.	Test page authorisation
1j	Test page authorisation	Without logging in, navigate to the users page.	The login page is shown
1k	Test page authorisation	Without logging in, navigate to the dashboard page.	The login page is shown

Test Block 2: Layout

Test	**Description**	**Action**	**Expected**
2a	Test navigation to the layout page	From the main menu, click *layout*	The layout list page is shown
2b	Test navigation to the create layout page	From the layout list page, click the create layout link	The create layout page is shown
2c	Test navigation to the create layout page	From layout secondary navigation menu, click create layout	The create layout page is shown
2d	Test layout valid name	On the create layout name page, enter the name *new layout*, click continue	The layout block editor is shown
2e	Test layout invalid name	On the create layout name page, leave the name field black, click continue	A layout name is required message is shown
2f	Test duplicate layout name	Create and save a layout called *duplicate*. On the create layout page, enter the name *duplicate*. Click continue	A layout with that name already exists message is shown
2g	Test layout block addition	On the layout block editor of the create layout page, drag a new row into the work area	The new row appears in the work area and its controls are displayed on hover
2h	Test layout block addition	On the layout block editor of the create layout page, drag a new column into the work area	The new column appears in the work area and its controls are displayed on hover
2i	Test layout block addition	On the layout block editor of the create layout page, drag a new floating block into the work area	The new floating block appears in the work area and its controls are displayed on hover
2j	Test layout block addition	On the layout block editor of the create layout page, drag a new column into the work area. Then, drag a second column and drop it next to the first	The new column appears next to the first, its height adjusts to the first columns height automatically
2k	Test layout block addition	On the layout block editor of the create layout page, drag a new column into the work area. Then, drag a second column and drop it below to the first	The new column appears below the first, in its own column block
2l	Test layout block reordering	Insert two rows, two columns side by side in one column block, and two columns side by side in a second block.	The first row aligns with the other elements in its new position at the bottom, every

		Drag and drop the first row to the bottom of the work area	other element is moved up to fill the space previously occupied by the moved row
2m	Test layout block reordering	From 2l, drag the row from the bottom and return it to its original position	The first row aligns with the other elements in its new position, every other element moves down to accommodate the position of the moved row
2n	Test layout block reordering	From 2l, drag and drop the first column in the first block to the right	The moved column aligns within the column block, both columns snap left to fill the space previously occupied by the moved column
2o	Test layout block reordering	From 2n, drag and drop the column back to its original position	The moved column aligns within the column block, the second column snaps right to make space for the moved column
2p	Test layout block reordering	From 2l, drag and drop the first column in the first block, place it on the right of second column in the second block	The moved column aligns and resizes to fit into the new block. The column in the first block snaps left to fill the space previously occupied by the moved column
2q	Test layout block reordering	From 2p, drag and drop the column back to its original position	The moved column aligns and resizes itself to fit into the first column block. The second column in the first blocks snaps right to accommodate the moved column
2r	Test layout block reordering	Place a floating block into the work area. Drag and drop it anywhere within the work area boundaries. Repeat several times	When the floating block is dropped, it remains in its new position
2s	Test work area boundaries	From 2r, attempt to move the floating block out of the work area	When dropped, the floating block snaps back to its original position
2t	Test work area boundaries	From 2l, attempt to move each of the rows and columns in turn out of the work area	The move row or column snaps back to its original position when dropped.
2u	Test column block reordering	From 2l, drag the first column block to the top, above the first row	The column blocks align with the other elements which all snap down to allow the moved block space to fit into the layout
2v	Test column block reordering	From 2u, move the column block back to its original position	The column block aligns with the other elements in its new position, the elements above the block snap upward to create space to accommodate the new block
2w	Test create new layout saving	Complete tests 2d, 2l and 2r in sequence, click save on the toolbar	The layout list page is shown with the new layout present. A successful creation message is shown
2x	Test create new layout saving	From 2w, click edit	The layout editor is shown with the rows, columns and floating

			blocks in the same positions as before
2y	Test editing a layout	From 2w, on the layout edit page, click edit. Complete test 2g and click save, when the layout list page shows, click edit again	The edit page reflects the modification to the layout structure that was made
2z	Test editing a layout	Repeat test 2y, however substitute test 2g with 2h. Repeat again for test 2i to 2v	The edit page reflects the modification to the layout structure that was made for all repetitions
2aa	Test layout deletion	From test 2w, on the layout list page, click delete on *new layout*	The layout disappears and a successful deletion message is shown
2ab	Test layout deletion constraints	Create an empty layout. Create an arrangement for the layout. Create a content type and apply the created arrangement to it. Navigate to the layout list page and click delete for the created layout	The layout may not be deleted because it is in use message is shown
2ac	Test layout deletion constraints	From 2ab, delete the content type. On the layout list page, click delete for the created layout	The layout disappears and a successful deletion message is shown
2ad	Test deletion of layout blocks	From 2w, on the layout list page, click edit on the created layout. Delete the first row, click save. From the layout list page click edit on the created layout	The layout block that was deleted will not be present
2ae	Test deletion of layout blocks	Repeat test 2ad for each layout block	The layout block that was deleted will not be present in any of the repetitions

Test Block 3: Arrangement

Test	**Description**	**Action**	**Expected**
3a	Test navigating to the arrangement page	Under the layout option on the main navigation menu, select layouts	The layout pages is shown with arrangement listed beneath each layout
3b	Test navigating to the create arrangement page	Under the layout option on the main navigation menu, select create arrangement	The create arrangement page is shown
3c	Test navigating to the create arrangement page	Navigate to the layout page, click create arrangement	The create arrangement page is shown
3d	Test valid arrangement information	Complete test 2w to create a layout. On the create arrangement page, enter *new arrangement*, and select the created layout from the dropdown. Click continue	The arrangement editor is shown
3e	Test invalid arrangement information	Complete test 2w to create a layout. On the create arrangement page, leave both fields blank and click continue	The name and layout fields are mandatory message is displayed
3f	Test invalid arrangement information	Complete test 2w to create a layout. On the create arrangement page, enter *new arrangement* in the name field, leave the layout dropdown blank. Click	The layout field is mandatory message is displayed

		continue	
3g	Test invalid arrangement information	Complete test 2w to create a layout. On the create arrangement page, leave the name field blank and select the created layout from the dropdown. Click continue	The name field is mandatory message is displayed
3h	Test duplicate arrangement information	Complete test 2w twice creating two layouts *layout one* and *layout two*. Create an arrangement for *layout one* called *duplicate*. On the create arrangement page, select *layout one* from the dropdown and enter the name *duplicate*. Click continue	Arrangement name is already in use message is displayed
3i	Test duplicate arrangement information	From 3h, on the create arrangement page, select *layout two* from the dropdown and enter the name *duplicate*. Click continue	The arrangement editor is shown
3j	Test widget insertion	On the arrangement editor, under the general category on the tool bar, drag a new heading widget into the first row block	The heading widget configuration modal window appears
3k	Test widget insertion	On the arrangement editor, under the general category on the tool bar, drag a new form widget into the second row block	The form widget configuration modal window appears
3l	Test widget controls	Insert a widget of any type into the first row, close the modal window that appears. Place the mouse curser over the widget	The drag, edit and delete control icons appear
3m	Test widget reordering	Place two widgets in the first row side by side. Drag and drop the second widget, placing on the opposite side of the first	The first widget snaps to the right, making space for the moved widget
3n	Test widget reordering	From 3l, return the second widget to its original position	The first widget snaps left, to its previous position, allowing the second widget the space it needs for its new position
3o	Test moving widgets	From 3l, move the first widget into the second row	The second widget snaps to the left, filling the space previously occupied by the first widget. The first widget appears within the second row
3p	Test moving widgets	From 3n, return the first widget to its original position	The second widget snaps right to allow the first widget space to return to its position
3q	Test configuration of widgets	From 3l, click the edit configuration icon on the first widget	The configuration modal window is shown
3r	Test deletion of widgets	From 3l, click the delete icon on the first widget	The widget disappears
3s	Test arrangement saving	Run tests 3d and 3m sequentially, insert a content widget into the first column and click save. On the layout/arrangement list page, click edit on the created arrangement	The name on the list page is the name specified; the arrangement appears on the selected layout. The arrangement edit page shows both widgets in the first row
3t	Test editing an arrangement	From 3s, on the arrangement editor, complete test 3j. Click save, on the list	The widgets on the arrangement reflect the change

		page click edit	that was made
3u	Test editing an arrangement	Repeat 3t, substitute test 3j for 3k. Repeat for 3l – 3r	The widgets on the arrangement reflect the change that was made in each repetition
3v	Test deleting an arrangement	From 3s, on the list page, click delete	The arrangement disappears and successful deletion message is shown
3w	Test arrangement deletion constraints	From 3s, create a content type and apply the created arrangement to it. On the list page, click delete on the arrangement	The arrangement may not be deleted because it is in use message is shown
3x	Test arrangement deletion constraints	From 3w, delete the content type. On the list page, click delete on the arrangement	The arrangement disappears and successful deletion message is shown
3y	Test content widget constraint	Repeat test 3s but do not insert a content widget. Click save	A content widget is mandatory message is shown
3z	Test content widget constraint	Repeat test 3s but insert two content widgets. Click save	An arrangement can only have one content widget message is shown

Test Block 4: Widgets

Test	**Description**	**Action**	**Expected**
4a	Test the heading widget	Navigate to the arrangement editor on the create arrangement page, filling in the name and layout fields as needed. Drag a heading widget into any of the layout blocks. On the modal window that appears, select *heading1* from the type dropdown and enter *heading test* in the text field. Click save	The modal window closes and the widget contains a heading 1 with heading test text
4b	Test the heading widget	From 4a, save the arrangement, navigate back to the list page and click edit on the arrangement	The heading widget is still present and has the same heading 1 text
4c	Test the heading widget	From 4a, on the arrangement editor, click on the edit icon of the widget and alter the type to *heading 2*, click save	The modal window closes and the widget now shows the same text but as a heading 2
4d	Test the heading widget	From 4c, save the arrangement. Navigate to the list page and click edit on the arrangement	The heading widget is still present and has the same heading 2 text
4e	Test rich text widget	Navigate to the arrangement editor on the create arrangement page, filling in the name and layout fields as needed. Drag a text widget into any of the layout blocks. On the modal window that appears, enter some text and format it with bold or italics. Click save	The modal window closes and the widget contains the entered, formatted text
4f	Test rich text widget	From 4e, save the arrangement, navigate back to the list page and click edit on the arrangement.	The text widget is still present and has the same formatted text
4g	Test rich text widget	From 4e, on the arrangement editor, click on the edit icon of the widget and alter the formatting to underline, click	The modal window closes and the widget contains the entered, formatted text

		save	
4h	Test rich text widget	From 4g, save the arrangement. Navigate to the list page and click edit on the arrangement	The text widget is still present and has the same formatted text
4i	Test form widget	Create a form with any fields. Navigate to the arrangement editor on the create arrangement page, filling in the name and layout fields as needed. Drag a form widget into any of the layout blocks. On the modal window that appears, select the created form and press save	The modal window closes and the widget contains the fields from the chosen form
4j	Test form widget	From 4i, save the arrangement, navigate back to the list page and click edit on the arrangement	The form widget is still present and has the form fields within it
4k	Test form widget	Create a second form with any fields. From 4i, on the arrangement editor, click on the edit icon of the widget and alter the form to the second form. Click save	The modal window closes and the widget now shows the second form
4l	Test form widget	From 4k, save the arrangement. Navigate to the list page and click edit on the arrangement	The form widget is still present and has the form fields within it
4m	Test image widget	Navigate to the arrangement editor on the create arrangement page, filling in the name and layout fields as needed. Drag an image widget into any of the layout blocks. On the modal window that appears, select an image file to upload. Click save	The modal window closes and the widget contains the chosen image
4n	Test image widget	From 4m, save the arrangement, navigate back to the list page and click edit on the arrangement	The image widget is still present and has the same image within it
4o	Test image widget	From 4m, on the arrangement editor, click on the edit icon of the widget and select a different image to upload, then click save	The modal window closes and the widget now shows the new image
4p	Test image widget	From 4o, save the arrangement. Navigate to the list page and click edit on the arrangement	The image widget is still present and has the same second image within it
4q	Test navigation widget	Create a page and a post with any content type. Navigate to the arrangement editor on the create arrangement page, filling in the name and layout fields as needed. Drag a navigation widget into any of the layout blocks. On the modal window that appears, add a new link and select the page from the drop down. Click save	The modal window closes and the widget contains a single link
4r	Test navigation widget	From 4q, save the arrangement, navigate back to the list page and click edit on the arrangement	The navigation widget is still present and has the same single link within it
4s	Test navigation widget	From 4q, on the arrangement editor, click on the edit icon of the widget and alter the first link from the page to the post, click save	The modal window closes and the widget now shows a single link
4t	Test navigation widget	From 4s, save the arrangement.	The navigation widget is still

		Navigate to the list page and click edit on the arrangement	present and has the same single link within it

Test Block 5: Theme

Test	Description	Action	Expected
5a	Navigate to theme page	Under the layout option on the main navigation menu, select themes	The theme list is shown
5b	Navigate to create theme page	From the theme list page, click create theme	The create theme page is shown
5c	Navigate to create theme page	Under the layout option on the main navigation menu, select create theme	The create theme page is shown
5d	Test creation of a theme	On the create theme page, enter the name *test theme*. Click save	The theme list is shown with the new theme present. A creation success message is shown
5e	Test creation of a theme with invalid data	On the create theme page, leave the name field blank. Click save	A theme name is required message appears
5f	Test duplication of themes	Create a theme called *duplicate*. On the create theme page, enter the name *duplicate* in the name field. Click save	The theme name is in use already message is shown
5g	Test theme attributes	Navigate to the create theme page. Enter the name *attribute test theme* in the name box. Alter the page background colour property. Save the theme. On the theme list page, click against the newly created theme	The page background colour property should show the altered colour and not the default
5h	Test theme attributes	Repeat test 5g for each attribute on each tab in turn	The altered attribute should always reflect the change after saving and returning to a theme
5i	Test editing a theme	From 5g, click edit against the created theme and alter the page background colour property and click save. Navigate to the theme list page and click edit against the edited theme	The altered page background colour should still show the alteration and not the previous colour
5j	Test editing a theme	Repeat test 5i for each property on each tab in turn	The altered attribute should be reflected when returning to the theme
5k	Test delete a theme	Complete test 5g. Navigate to the theme list page and click delete against the created theme	The theme disappears from the list and a deletion successful message is shown
5l	Test theme deletion constraints	Complete test 5g. Create a content type and apply the created theme to it. Navigate to the theme list page and click delete against the created theme	The theme cannot be deleted as it is in use message is shown
5m	Test theme deletion constraints	From 5l, delete the created content type. Navigate to the theme list page and click delete against the created theme	The theme disappears from the list and a deletion successful message is shown

Test Block 6: Form

Test	Description	Action	Expected
6a	Test navigation to the form page	On the main menu, click forms	The form list page is shown
6b	Test navigation to the create form page	On the main navigation menu, under forms, click create form	The create form page is shown
6c	Test navigation to the create form page	On the form list page, click create form	The create form page is shown
6d	Creating a form	On the create form page, enter the name *test form* in the name field. Add a text field and submit button to the form and click save	The form list page loads and the new form is present
6e	Creating a form with invalid data	On the create form page, leave the name field blank. Add a text field and submit button to the form and click save	The name field is required error is shown
6f	Test form name duplication	Create a form with the name *duplicate*. On the create form page, enter the name *duplicate* into the name field. Add a text field and submit button to the form and click save0000	A form with that name already exists error is shown
6g	Test form field constraints	On the create form page, enter a name *test constraints* and click save	A form must have at least one field error message is shown
6h	Test form field constraints	On the create form page, enter a name *test constraints,* add a text field and click save	A form must have at least one submit button is shown
6i	Test form field constraints	On the create form page, enter a name *test constraints,* add a text field and a reset button and click save	A form must have at least one submit button is shown
6j	Test field insertion	Navigate to the create form page, enter a name *test* and insert a submit button. Add a text field; give it a name of *field* and click save. Navigate to the list page and click on edit alongside the created form	The form is created successfully and the field with the correct name is present when the form editor is returned to
6k	Test field insertion	Repeat 6j for each of the field types	The form is created successfully in each case and the field with the correct name is always present when the form editor is returned to
6l	Test mandatory fields	Repeat test 6j and 6k, although each time set the field to mandatory. Upon returning to the form editor, click the edit icon next to the field name	The form is created successfully each time and each case the field with the correct name is still present. Edit opens a modal window revealing the mandatory checkbox is still checked
6m	Test field reordering	Create a new form with one of each field type. Reorder the form fields by dragging and dropping them, so the button is first and the text field is last; then click save. Navigate to the form list page and click edit on the created form	The field order reflects the alteration that was made, the button is first and the text field is last
6n	Test field reordering	Repeat test 6m, substituting text field and button for dropdown and check group	The field order reflects the alteration that was made

6o	Test field reordering	Repeat test 6m, substituting text field and button for text area and checkbox	The field order reflects the alteration that was made
6p	Test field reordering	Repeat test 6m, substituting text field and button for radio buttons and checkbox	The field order reflects the alteration that was made
6q	Test validatable form fields	On the form create page, enter *validatable form* in the name field and add a text field. Select minimum text length from the validation drop down and enter the minimum length as 8. Add a submit button to the form and click save. Navigate to the form list page and click edit on the created form. Click on the edit icon for the text field	The form should be created successfully. The text field should still be present when the form editor is returned to. The validator is selected in dropdown when the edit modal window opens and 8 is present in the minimum length field
6r	Test validatable form fields	Repeat test 6q for each validator type, using 8 as the minimum length (if the validator needs a minimum) and 23 as the maximum length (if the validator needs a maximum length)	The form should be created successfully each time and, when returning to the form editor, the field is still present and its validation option are present and correct in the edit modal window
6s	Test form fields with options	On the form create page, enter *options form* in the name field. Add a drop down field and add the options *option1*, *option2*, and *option3*. Add a submit button and click save. Navigate to the form list page and click edit on the created form. Click on the edit icon for the dropdown field	The form should be created successfully and, when returning to the form editor, the dropdown field is still present. When the edit modal window is shown, all three options are still present
6t	Test form fields with options	Repeat test 6s for radio buttons and check box group	The form is created successfully each time and, when returning to the form editor, the field is present and, in the modal window, the options are still present
6u	Test option deletion	Repeat tests 6s and 6t although after adding all three options, delete the second	The form is created successfully each time and, when returning to the form editor, the field is present and, in the modal window, the first and third options are present and the second is not
6v	Test button types	On the create form page, enter the name *button form* in the name field and insert a button, setting the type to *submit*. Add a text field and click save. Navigate to the form list page and click edit on the created form. Click on the edit icon for the button	The form is created successfully. When returning to the form edit page, the button is present and, in the modal window, the submit option is selected
6w	Test button types	Repeat test 6v, although use *reset* as the button type	The form is created successfully. When returning to the form edit page, the button is present and, in the modal window, the reset option is selected
6x	Test field deletion	On the form create page, insert *delete*	The text field disappears when

		field form into the name field and add one field of each type, including a submit button. Click the delete icon next to the text field. Click save, navigate to the form list page and click edit next to the created form	the delete icon is clicked. The form is created successfully and, when returning to the form editor, the text field is not present
6y	Test field deletion	Repeat test 6x for each type of field	The field disappears each time when the delete icon is clicked. The form is created successfully in each repetition and the deleted field is never present
6z	Test form editing	Create a new form called *edit form*, adding a text field and submit button. Rerun tests 6j-6y, although complete them not by creating a form, but by editing the *edit form*	The tests should run as previously described
6aa	Test form deletion	Create a form with any name, inserting the required field and submit button. Navigate to the form list page and click delete against the created form	The form is deleted and a deletion successful message is shown
6ab	Test form deletion constraints	Create a form with any name, inserting the required field and submit button. Create a layout and an arrangement, inserting into the arrangement a form widget choosing the created form. Navigate to the form list page and click delete against the created form	The form cannot be deleted as it is in use error message is shown
6ac	Test form deletion constraints	From 6ab, delete the arrangement. Navigate to the form list page and click delete against the created form	The form is deleted and a deletion successful message is shown
6ad	Test navigation to form submissions	On the main navigation menu, under forms, click create submissions	The submission list is shown
6ae	Test form submissions	Create a new form, select it within a form widget on an arrangement, navigate to a page with that arrangement, fill out and submit the form. Navigate to the submissions page	The submission list page is shown, the submission should be present
6af	Test form submissions	From 6ae, click on the submission	The full details of the submission are shown
6ag	Test delete form submissions	From 6ae, click delete alongside the submission	The submission disappears and a deletion successful message is shown

Test Block 7: Users

Test	**Description**	**Action**	**Expected**
7a	Test navigate to user list	On the main navigate menu click users	The user list is shown
7b	Test navigate to create new user	On the main navigation menu, under users, click create user	The create user page is shown
7c	Test navigate to create new user	Navigate to the user list page and click create user	The create user page is shown
7d	Test create new user	Navigate to the create new user page. Enter the username *test* and a password	The user list page is shown with the new user present

		of *password*. Click create	
7e	Test login of new user	From 7d, logout and attempt to login using the new user's credentials	The login is successful and the dashboard is displayed
7f	Test create a user with invalid details	On the create user page, leave both the username and password fields blank. Click create	Username and password fields are required error message is shown
7g	Test create a user with invalid details	On the create user page, enter *test* into the username field but leave password blank, click create	Password is required error is shown
7h	Test create a user with invalid details	On the create user page, leave the username field blank and enter *password* in the password field, click create	Username is required error is shown
7i	Test duplicate username constraint	Create a user with a username of *duplicate.* On the create user page, enter *duplicate* into the username field and *password* into the password field. Click create	A user with that username already exists error message is shown
7j	Test editing a user	From 7d, navigate to the user list page and click edit against the created user, alter the users password to *password2* click save. Logout and attempt to login as that user with the new password	The login is successful and the dashboard is displayed
7k	Test deleting a user	From 7d, navigate to the user list page and click delete against the created user	The user is deleted
7l	Test user deletion constraint	From 7d, login as the created user, navigate to the user list page and click delete against the created user	The user is currently logged in and cannot be deleted
7m	Test user deletion constraint	From 7l, logout and log back in using different credentials to the created user. Navigate to the user list page and click delete against the created user	The user is deleted

Test Block 8: Content Types

Test	**Description**	**Action**	**Expected**
8a	Test navigation to content types	On the main navigation menu, under content, click content types	The content type list page is shown
8b	Test navigation to create content type	On the content type list page, click create content type	The create content type page is shown
8c	Test content type creation	On the create content type page, enter *test type* in the name field, select an arrangement and theme from the dropdown and select *page* from the type dropdown. Add any attribute and click save	The content type list page is shown, the new content type is present
8d	Test content type creation constraints	Repeat test 8c, although do not enter a content type name	Content type name is mandatory error is shown
8e	Test content type creation constraints	Repeat 8c, although do not select an arrangement	Arrangement is mandatory error is shown
8f	Test content type creation constraints	Repeat 8c, although do not select a theme	Theme is mandatory error is shown
8g	Test content type creation constraints	Repeat 8c, although do not select anything from the type dropdown	Type is mandatory error is shown

8h	Test content type creation constraints	Repeat 8c, although do not add any attributes	A content type must have at least one attribute message is shown
8i	Test duplicate content type name	Create a content type called *duplicate*. Repeat test 8c although enter the name *duplicate* instead	A content type with that name already exists message is shown
8j	Test content type attribute addition	On the content type crate page, enter *test type* in the name field, select *page* from the type field and choose any arrangement and theme. Add the text attribute and enter the name *text*, click save. Navigate to the content type list page, click on edit alongside the created content type	The content type should be created successfully. When returning to the content type editor, the text attribute will still be present, with the correct name
8k	Test content type attribute addition	Repeat 8j for each attribute type in turn	The content type should be created successfully each time. When returning to the content type editor, the attribute will still be present, with the correct name
8l	Test content type attribute editing	Repeat test 8j. When returning to the content type editor, click the edit icon alongside the content type, alter the name to *changed* and click save. Navigate to the content type list page, click on edit alongside the edited content type	When returning to the content type editor for the second time, the attribute name reflects the alteration to *changed*
8m	Test content type attribute editing	Repeat test 8l for each attribute type in turn	For each repetition, when returning to the content type editor, the name of the attribute reflects the change
8n	Test content type attribute reordering	Create a new content type, adding one of each of the attributes. Set name, type, arrangement and theme as required. Drag and drop the first attribute and move it to the last position. Click save and navigate to the content type list page. Click edit next to the created content type	The content type is created successfully. When returning to the content type editor, the attribute order reflects the reordering that was performed
8o	Test content type attribute reordering	Repeat test 8n for each attribute in turn	Each time, the content type is created successfully and, when returning to the content type editor, the order reflects the reordering that was made
8p	Test content type attribute deletion	Create a new content type, adding one of each of the attributes. Set name, type, arrangement and theme as required. Delete the first attribute. Click save and navigate to the content type list page. Click edit next to the created content type	The content type is saved successfully and, when returning to the content type editor, the attribute that was not deleted is not present
8q	Test content type attribute deletion	Using the same created content type, repeat 8p for each added attribute in turn	Each time, the attribute will no longer be present when returning to the content type editor. After deleting the final attribute, a content type must

			have at least one attribute error message is shown
8r	Test content type editing	Create a new content type named *edit*, adding an attribute, arrangement and theme as required. Repeat test 8j to 8r although, instead of starting with a new content type, use the *edit* content type	The test should perform as previously expected
8s	Test content type deletion	Create a new content type with any name. Navigate to the content type list page and click delete alongside the created content type	The content type disappears and successful deletion message is shown
8t	Test content type deletion constraints	Create a new content type with the *page* type chosen. Create a new page using the content type. Navigate to the content type list page and click delete alongside the created content type	The content type cannot be deleted because it is in use error message is shown
8u	Test content type deletion constraints	From 8t, delete the created page. Navigate to the content type list page and click delete alongside the created content type	The content type disappears and successful deletion message is shown
8v	Test content type visibility	Create two page content types. Navigate to the create pages page	Both content types appear in the content type dropdown
8w	Test content type visibility	From 8v, navigate to the create post page	Neither of the content types appear in the content type dropdown
8x	Test content type visibility	Create two post content types. Navigate to the create post page	Both content types appear in the content type dropdown
8y	Test content type visibility	From 8x, navigate to the page for create page	Neither of the content types appear in the content type dropdown

Test Block 9: Pages

Test	**Description**	**Action**	**Expected**
9a	Navigate to pages	On the main navigation menu, under content, click pages	The page list is shown
9b	Navigate to create page	On the main navigation menu, under content, click create page	The create page is shown
9c	Navigate to create page	From the page list, click create page	The create page is shown
9d	Test create page with valid data	On the create page, enter *test page* in the name field, select a content type from the dropdown, click continue	The insert content form is shown
9e	Test create page with invalid data	Repeat 9d, although do not enter a page name	The name field is required error message is shown
9f	Test create page with invalid data	Repeat 9d, although do not select a content type	Content type is required message is shown
9g	Test duplicate page name	Create a page called *duplicate*. Repeat test 9d, although enter *duplicate* as the page name	A page with that name already exists message is shown
9h	Test page saving	Create a page content type with one of each attribute. Repeat test 9d, selecting the created content type. On the insert page that appears, fill out all of the content with random, but valid data,	The page is created successfully and is present on the page list. When editing the page, the name and content fields show the correct, inserted data

		and click continue. Navigate to the page list and click edit alongside the created page	
9i	Test parent page	Create a page called *parent*. Repeat test 9h but select *parent* from the parent page list on the first step	The page is created successfully and is present on the page list; it appears immediately below *parent* and is indented slightly. When editing the page, the name, parent page and content fields show the correct data
9j	Test SEO title addition	Repeat test 9h but also enter *test title* in the SEO title field	The page is created successfully and is present on the page list. When editing the page, the name, SEO title and content fields show the correct, inserted data
9k	Test meta description addition	Repeat test 9h but also enter *test description* in the meta description field	The page is created successfully and is present on the page list. When editing the page, the name, meta description and content fields show the correct, inserted data
9l	Test meta keywords addition	Repeat test 9h but also enter *test keywords* in the meta keywords field	The page is created successfully and is present on the page list. When editing the page, the name, meta keywords and content fields show the correct, inserted data
9m	Test invalid content insertion	Repeat test 9h, however, leave the first content field blank	The field is mandatory error message is shown
9n	Test invalid content insertion	Repeat test 9m for each content field in turn	The field is mandatory error message is shown each time
9o	Test page editing	Create a new page with the name *edit*. Repeat test 9e to 9n, however do not start on the page for create page, start on the edit page for the created *edit* page	The tests perform as previously described
9p	Test page deletion	Create a new page called *for deletion*. On the page list, click delete alongside the created page	The page disappears and a deletion successful message is shown
9q	Test homepage nomination	Create a new page called *homepage*. On the page list page, click set as homepage alongside the created page	A home icon appears next to the page title. The page was set as the home page success message is shown
9r	Test homepage nomination	From 9q, create a new page called *new homepage*. On the page list page, click set as homepage alongside the created page	A home icon appears next to the page title. The page was set as the home page success message is shown. The homepage icon is removed from the previous homepage
9s	Test page deletion constraints	Create a new page called *delete*. Set it as the homepage. On the page list page, click delete alongside the created page	The page cannot be deleted because it is the homepage error is shown
9t	Test page deletion constraints	From 9s, create a new page called *home*. Set it as the homepage. On the page list page, click delete alongside the *delete*	The page disappears and a deletion successful message is shown

		page	
9u	Test page reordering	Create 4 pages named: *page1, page2, page3, page4*. They should not have a parent page. Navigate to the page list	All four pages are shown, in the order they were created, with no indentation
9v	Test page reordering	From 9u, drag *page1* to the end of the list, after *page4*. Reload the page list	*Page1* appears at the end of the list
9w	Test page reordering	Repeat 9v for each page in turn	Each time, the moved page still appears at the bottom after the reload
9x	Test page reordering	From 9u, drag *page2* to the right so it becomes a child of *page1*. Reload the page list	*Page2* appears indented beneath *page1*
9y	Test page reordering	From 9x, drag *page3* to the right so it becomes a child of *page2*. Reload the page list	*Page3* appears indented beneath *page2*
9z	Test page reordering	From 9y, drag *page4* to the right, so it becomes a child of *page3*. Reload the page list	*Page4* appears indented beneath *page3*
9aa	Test page reordering	Repeat tests 9x-9z, dragging to the left however not the right to un-parent the pages	Each time, the moved page, and any child pages beneath it, appear on the far left after reloading the page

Test Block 10: Posts

Test	**Description**	**Action**	**Expected**
10a	Navigate to posts	On the main navigation menu, under content, click posts	The post list is shown
10b	Navigate to create post	On the main navigation menu, under content, click create post	The create post page is shown
10c	Navigate to create post	From the page list, click create post	The create post page is shown
10d	Test valid create post	On the create post page, enter the name *test post* and select any content type, click continue	The insert content page is shown
10e	Test invalid create post	Repeat test 10d, however, do not enter a post name	Post name is required message is shown
10f	Test invalid create post	Repeat test 10d, however, do select a content type	Content type is required message is shown
10g	Test duplicate name constraint	Create a post called *duplicate*. Repeat test 10d, however, enter the name *duplicate*	A post with that name already exists message is shown
10h	Test post saving	Create a post content type with one of each available attribute. Repeat test 10d. On the insert content page, enter random but valid content and click save	The post list is shown and the new post is present
10i	Test invalid content insertion	Repeat test 10h, however, leave the first content field blank	The field is required error message is shown
10j	Test invalid content insertion	Repeat test 10i for each of the content fields in turn	For each repetition, an error is produced indicating that the blank field is mandatory
10k	Test edit post	Create a new post called *edit*. Repeat tests 10d-10j, however, instead of starting on the create post page, start on the edit page of the *edit* post, by clicking	The tests perform as previously described

		edit alongside the post name on the post list page	
10l	Test delete post	Create a new post called *delete*. On the post list page, click delete alongside the created post	The post disappears and a deletion successful message is shown

Test Block 11: Front End

Test	Description	Action	Expected
11a	Test page rendering	Create a layout with 1 of each type of block. Create an arrangement for that layout, inserting one of type of widget across the 3 blocks. Create a theme, using the default settings. Create a page content type with 1 of each attribute, and choosing the created arrangement and theme. Use the created content type to create a page. Set the page as the homepage, navigate to the page	The web page is shown, the widgets appear as they were organised in the arrangement editor into the correct blocks. The theme is applied correctly and the content is shown within the content widget
11b	Test page arrangement override	Create a new arrangement with the widgets in different positions than before. From 11a, alter the page to use the new arrangement. Navigate to the page	The web page is shown as before, although the new arrangement is used. The content and theme remain the same
11c	Test page theme override	Create a new theme, using distinctly different attributes. From 11a, alter the page to use the new theme. Navigate to the page	The web page is shown as before, although the new theme is used. The content and arrangement remain the same
11d	Test theme and arrangement override	Using the arrangement and theme from 11b and 11c, alter the page created in 11a to use both the new theme and arrangement. Navigate to the page	The web page is shown as before, however it is using the new theme and arrangement. The content remains the same
11e	Test post page rendering	Create a new post content type with 1 of each attribute, applying the theme and arrangement created in 11a. Create a new post using the created content type. Navigate to the post	The post page is shown, the widgets appear as they were organised in the arrangement editor into the correct blocks. The theme is applied correctly and the content is shown within the content widget
11f	Test widgets	Using the arrangement created in 11a, repeat test block 4 in entirety however, instead of checking the widget on the arrangement editor, navigate the page using the arrangement	The widget is displayed on the web page, as expected and described in test block 4
11g	Test form submitting	Create a new form, with 1 of each field, adding options to any as required and choosing to show a message when submitted. Select the created form as the source form for the form widget on the arrangement from 11a. Navigate to the page, fill out the form and click the submit button	The message is shown
11h	Test form submitting	From 11g, alter the form to direct to a page when submitted. Reload the page	The selected page is shown

		and submit the form again	
11i	Test invalid submissions	Alter the form so that all fields are mandatory. Navigate to the page where the form widget is shown and fill out the form, leaving the first field blank. Submit the form	The field is required error message is shown
11j	Test invalid submissions	Repeat 11i for each field in turn	On each repetition, the field is required message is shown
11k	Test invalid submissions	To the text field on the form from 11g, add a minimum text validator with a minimum length of 5. Navigate to the page where the form widget is shown and fill out the form, entering *data* into the text field	The value of the text field must be greater than 5 message is shown
11l	Test invalid submissions	Repeat 11k for each type of validator available, using the minimum length of 5 and the maximum length of 10. For each validator repeat 6 times using the following values: *data*, *correct*, *farfartoolong, 2, 6, 23*	The text field reports an error when using a numerical validator for the first 3 inputs, and reports an error on the second 3 inputs only if the input does not meet the minimum and maximum values. The text field reports an error when using a text validator on all of the last 3 inputs, but only on the first 3 if the length of the input word does not meet the minimum and maximum length

Test Block 12: Robustness and Performance

Test	**Description**	**Action**	**Expected**
12a	Test page load times	Create a page using an arrangement which includes each type of widget, a content type that includes each attribute and any theme. Make an HTTP request to the page 200 times	The page should load within 1 second
12b	Test page load times	Repeat 12a although using a post	The post page should load within 1 second
12c	Simultaneous request handling	From 12a, repeat the 200 requests, although send them simultaneously	The response should be produced within 1 second
12d	Simultaneous request handling	From 12b, repeat the 200 requests, although send them simultaneously	The response should be produced within 1 second

A5. Framework Design

This appendix is a continuation of the framework design from chapter six. Class diagrams of the major framework classes are documented, as well as a specification for the structure of the XML configuration file. The initialisation, request handling and component rendering processes are also explained in more detail.

A5.1 Initialization Process

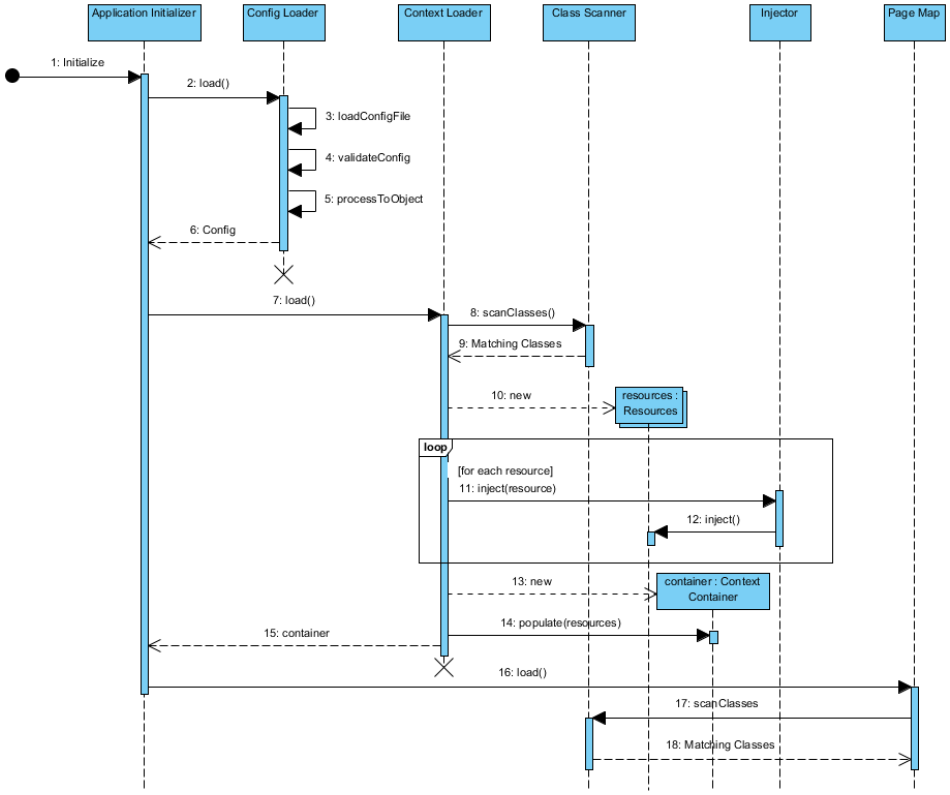

Figure 36 Sequence diagram showing the Picon application initialisation process

The application initialisation begins with the loading and validation of the XML configuration file, followed by parsing into a framework defined domain object. Although PHP includes functionality to create a dynamic domain object for this process, I believe it is beneficial for the framework to use its own as it will allow for custom method definitions, enabling more complex accessors and mutators.

Secondly the context, consisting of an associative array of resources, is loaded. The resources are located by a class scanner which detects classes declared as resources through class level

annotation. Each class is then instantiated and placed into the context array. Each resource is passed to the injector, which will search for any resource dependency declarations, also achieved through annotation, and provide the resource with any of the resources it is dependent upon.

The class scanner requires all of the user defined classes to be declared; consequently all PHP class files must be located and included beforehand through a recursive directory scan. Not the most efficient method, but there is no other way of performing a full class scan. The class scanner is rule based; it uses reflection to analyse each declared class and determines if it matches the criteria specified by one or more added rules. The class scanner and rules are shown in Figure 37.

Thirdly, the page map is initialised. The page map is responsible for storing all of the page names, the page classes that they equate to, and any stateful page objects. The page map will use the class scanner to locate webpage extension classes. The matching classes will be placed into an array with the page name which, by default, is the name of the class. The *Path* class level annotation may be utilised to alter the default name, by specifying a path annotation argument. The page map will be a singleton, and persist itself within the session so that stateful pages are available in subsequent requests.

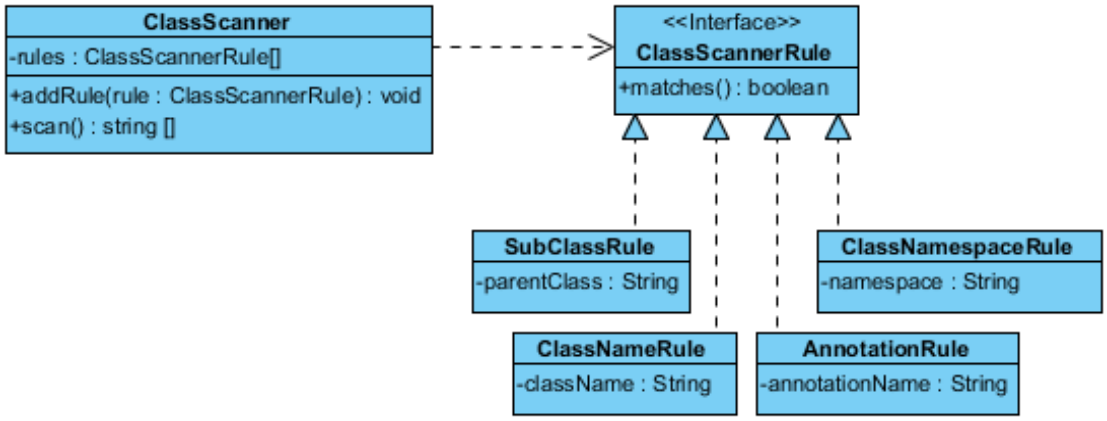

Figure 37 Class diagram showing the class scanner

A5.2 Request Processing Scenarios

The requirements in appendix two list several types of request scenario. Although the request cycle always uses the same process of request resolving and iterative request target processing, the overall process is somewhat different for each of the defined request scenarios. Each scenario is described in this section.

Standard Page Request

The request URI is expected to contain a path which corresponds to the name of a page in the page map. This path is used to locate the webpage extension class which is instantiated and then rendered, producing the HTML response. If the page is stateful, it is placed into the page map for persistence. Figure 38 illustrates the process. If no path is specified, the resolver will look up the default page name in the application configuration.

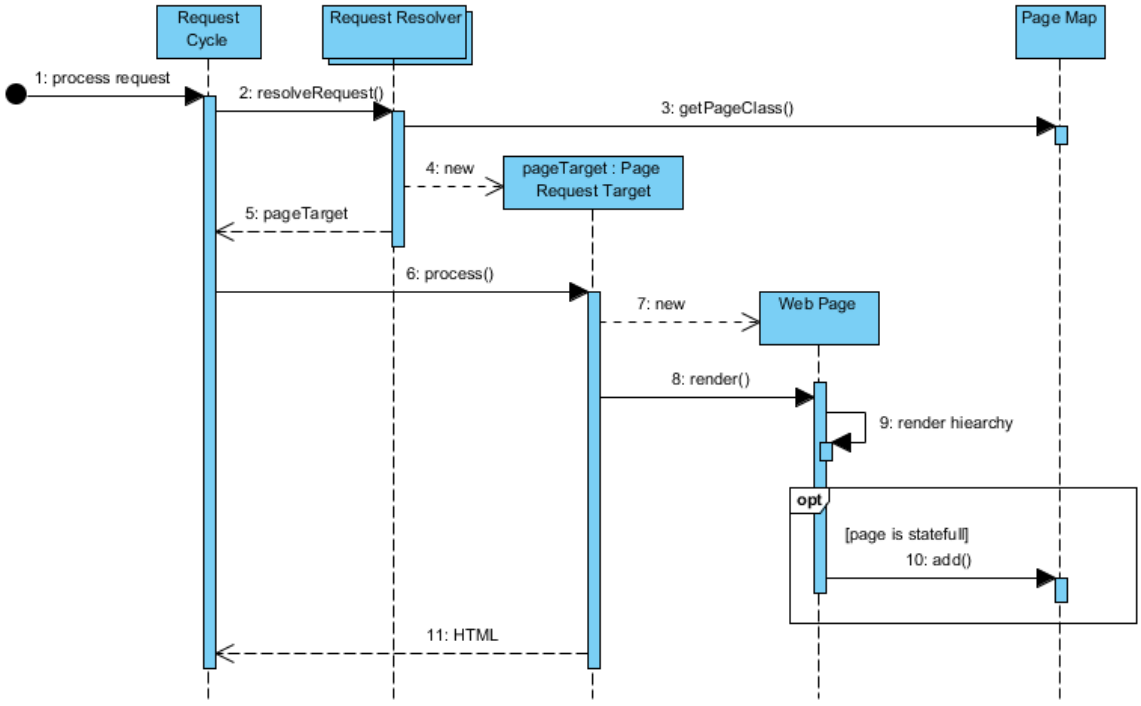

Figure 38 Sequence diagram showing the standard page request process

Stateless Callback Request

Any components with an associated callback can have that callback invoked during a callback request. As the callback is to be executed in a request subsequent to that in which it was defined, it

must be persisted. Persistence is not required however if the page is stateless and can be reconstructed, identically to the original, in the subsequent request. For a stateless callback request, the page class is located in the same way as the standard request; however, before rendering the page, the callback component is located through recursive iteration of the component hierarchy and then invoked. Figure 39 illustrates the process. The rendered page is the page on which the callback component was located; however, the callback is permitted to add a new request target to the stack, allowing redirection to a different page.

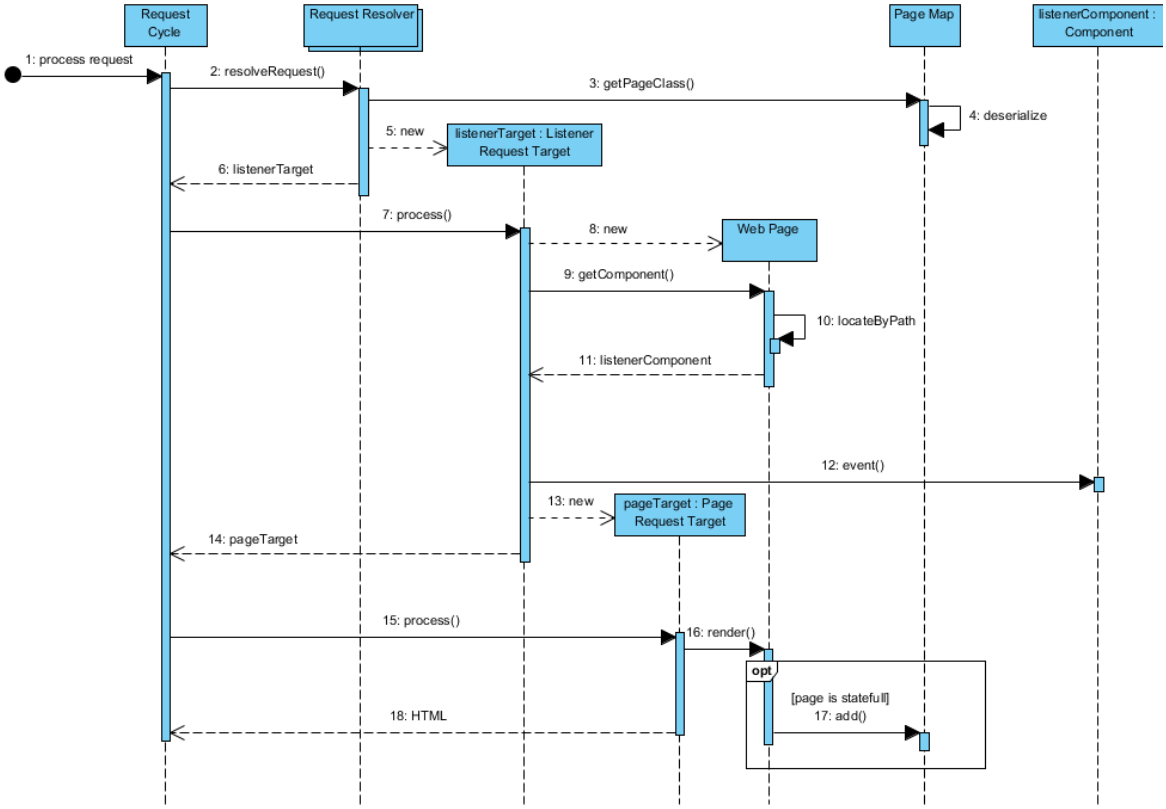

Figure 39 Sequence diagram showing the stateless callback request

Stateful Callback Request

Like the stateless request, the stateful request occurs to invoke an associated callback. Unlike a stateless request however, the page contains volatile data that cannot be re-retrieved, thus the webpage cannot be identically reconstructed. Consequently, the page is serialised within the page map against a page ID. The page object is retrieved in the callback request; the callback component is located and invoked, and the page is re-rendered. Figure 40 illustrates the process.

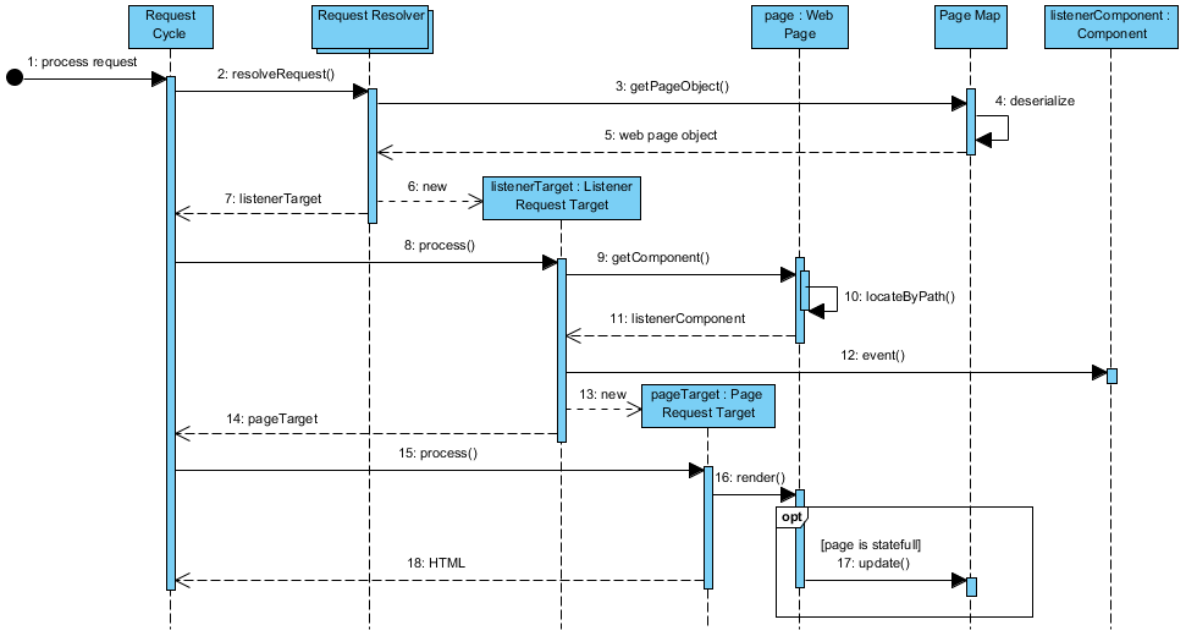

Figure 40 Sequence diagram showing a stateful callback request process

Ajax Request

Chapter five explained how a callback function may wish to update a page and re-render it, rather than directing to a new one – eliciting the statefulness requirement. Ajax is the epitome is statefulness – rather than generating a new page, an existing page is simply updated with a new state. Any page with an Ajax component in its hierarchy will be stateful and thus persisted in the page map. On receipt of an Ajax request, the persisted page object is retrieved and the callback component located and invoked. The callback however will be provided with a reference to the Ajax request target, so that as well as updating component states, it may also pass those components to the target – marking them for re-render. The Ajax request will render each of the components separately, and encode the HTML into a JSON string which is returned as the HTTP response. Any component that is to be updated in this way must, in its HTML output, specify a unique mark-up ID. The framework will generate a random string to achieve this. The mark-up ID of a component will appear in the JSON string against its corresponding HTML. Figure 41 illustrates the process. Client side JavaScript is responsible for initiating Ajax requests on DOM actions and acting upon the response to update the required tags.

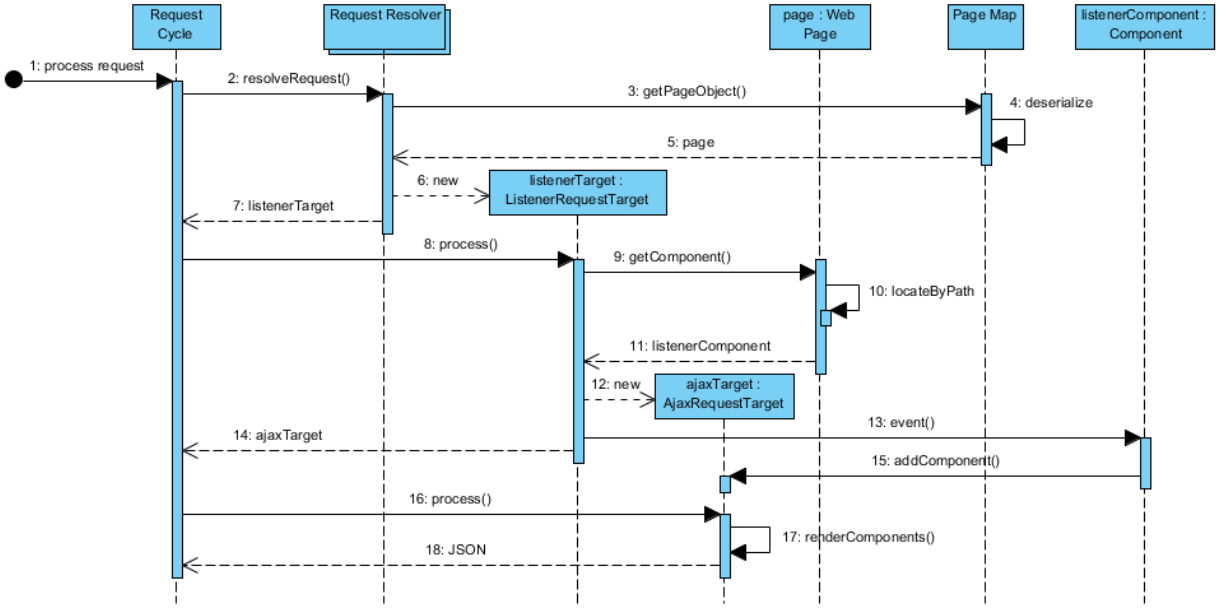

Figure 41 Sequence diagram showing the Ajax request process

Resource Request

The resource request allows the framework to act as a proxy for web protected resource files like CSS. Each resource file is to be located in the same directory as a class. The class and resource file name are required in the resource request URI. When received, the class directory is located with reflection; the resource file is then located and read; its contents are returned forming the HTTP response. Using the response object, the content type will be set based on the type of resource requested. The resource request process is shown in Figure 42.

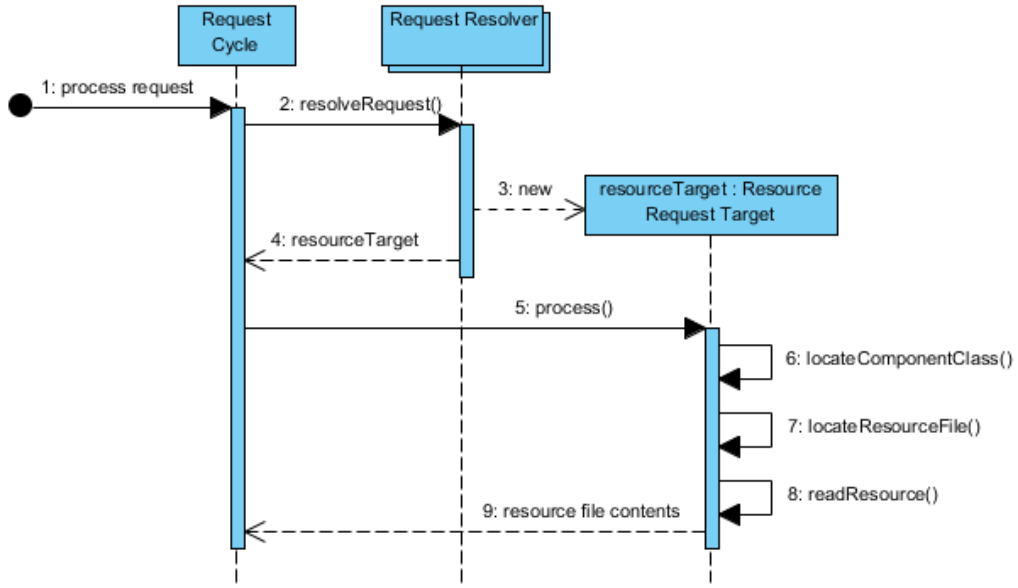

Figure 42 Sequence diagram showing the resource request process

URL Formatting and Generation

As well as analysing and responding to URLs, the framework is also capable of generating them. This is achieved through the request resolver chain, although the process of generating a URL is inverted compared to the process of matching a URL to a request target. Table 4 shows URL patterns for each request types and the input required to construct them. As components are composed into a hierarchy, a callback component can be used to locate the webpage component it belongs to, meaning the webpage component need not be separately specified.

Table 4 Table showing request types and their URL patterns

Type	URL Pattern	Required Input
Standard	http://example.com/pagename	Webpage component
Stateless callback	http://example.com/pagename?listener=path:to:component	Callback component
Stateful callback	http://example.com/pagename?pageid=45& listener=path:to:component	Callback component
Ajax	http://example.com/pagename?ajax=ajax&pageid=45& listener=path:to:component	Callback component
Resource	http://example.com/?resource=namespcae/classname:myfile.css	Class, resource name

Classes

Each of the request resolvers and request targets that have been described are separated out into their own classes, making them highly cohesive and reusable. The request cycle creates request and

response objects, as described in chapter six; only the interfaces are presented here. The interface allows for two implementations of each: one to use as a full implementation for actual request processing, and a second to represent a “dummy” request to enable testing. The request resolver collection is composed of the request resolvers. To the request cycle it is a single resolver, but it internally iterates through each of the other resolvers, creating the chain of responsibility.

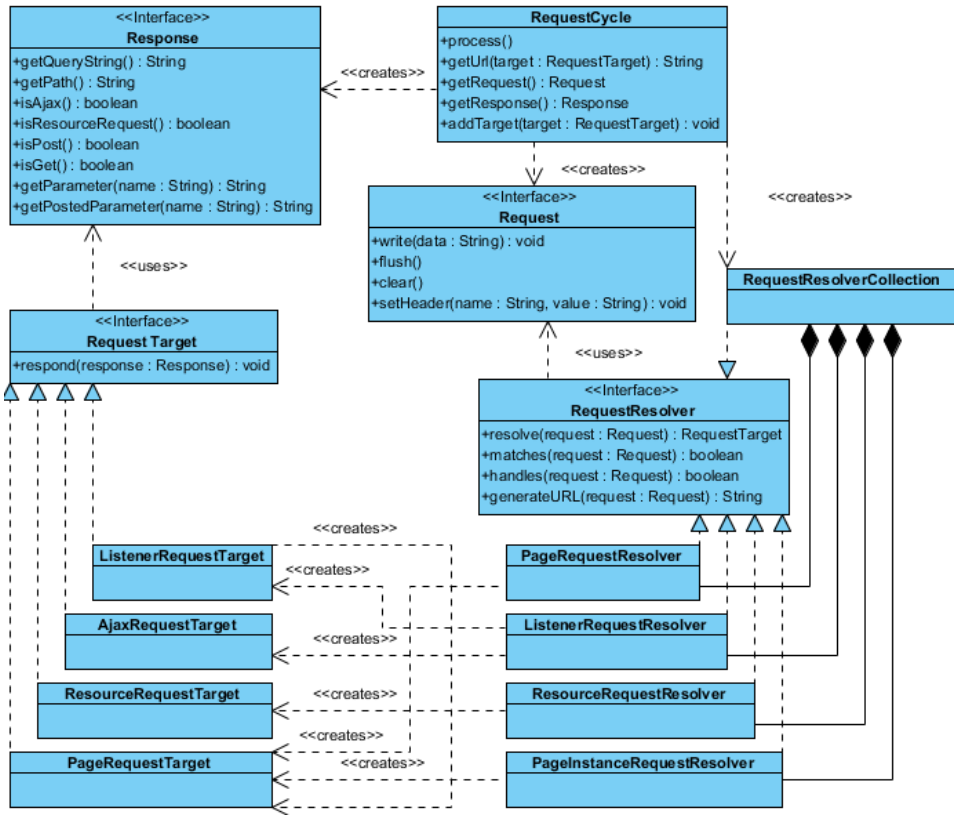

Figure 43 Class diagram showing the request target, request resolver and request cycle classes

A5.3 Component Lifecycle

After instantiation, a component is usually required to instantiate any child component it requires and add them to the hierarchy. Alternately, child components may be instantiated and added by another component, usually a parent. The component is provided with its identifier as a constructor argument; this will be used at a later stage to locate the associated mark-up.

Next, the component is initialised; this will be invoked by the root component when the hierarchy construction is complete, enabling any parent component dependent actions to occur. The next step is pre page render, signalling the whole webpage, and consequently all of its child components, are about to be rendered.

The internal render process is the next step and begins with mark-up retrieval. The mark-up tag which the component is associated with is located by requesting it from the parent component. It is expected that the parent mark-up will contain a tag with an identifier that matches the identifier of the component. If not, an exception is thrown. If the component does not have a parent, and is therefore the root component, the mark-up loader is invoked to locate and parse the associated HTML file. Once the mark-up is located, the component may alter it as required and then render it by writing plain HTML to the response object.

The mark-up loader will automatically assign an identifier to an HTML head tag. This means when the tag is rendered, a corresponding object in the hierarchy will be expected for it. This corresponding object will be added automatically by the framework. When this head component is rendering, it will locate and invoke the render head step on every component in the hierarchy, so any component is capable of rendering a header contributor. A header contributor could be inclusion of a CSS or JavaScript file, or the addition of a block of inline JavaScript.

Once the mark-up for a component has been sourced, it begins to actually render; this is achieved by writing to the response object. First, the opening tag is rendered. Each child tag is then iterated through, rendering each one. This iteration is done recursively so the entire child tag is rendered. In the event that one of the child tags has an identifier, a child component with a corresponding identifier is located and has its render method called – creating the recursive render cycle. As the child tag with an identifier, and thus all of its children, will be rendered by the child component, the recursive iterative render process ignores the tag entirely. Once the child component has rendered,

the iterative rendering will resume on the next tag. If no matching child component is found, an exception is thrown. Finally, the component renders its close tag.

Surrounding the internal render process are steps for pre component render, and post component render. Unlike the pre and post page render they do not invoke their namesakes on any child components. For the most part, pre and post, page and component render steps are for sub classes to perform lifecycle specific actions. The only action performed by every component on these steps takes place in post component render, which validates the render. This consists of an iteration through all of the child components to ensure that they have been rendered, if not, an exception is thrown. Typical actions for these steps in sub classes are usually model resolution in a pre page render, or clean up in post page render. The page component, in post page render, checks the component hierarchy for statefulness and persists the webpage component, and thus all its child components, in the page map if required.

Figure 44 illustrates the entire lifecycle of a component. In the case of a webpage component, the found messages will have come from a request target. A component being independently rendered for an Ajax response will also receive found messages from a request target. In most cases however, the found messages will originate from a parent component; in each, the component will pass those messages onto each of its child components which in turn become its found messages. The action of a component receiving a method invocation from a parent component and invoking the method namesake in its child components creates a recursive process.

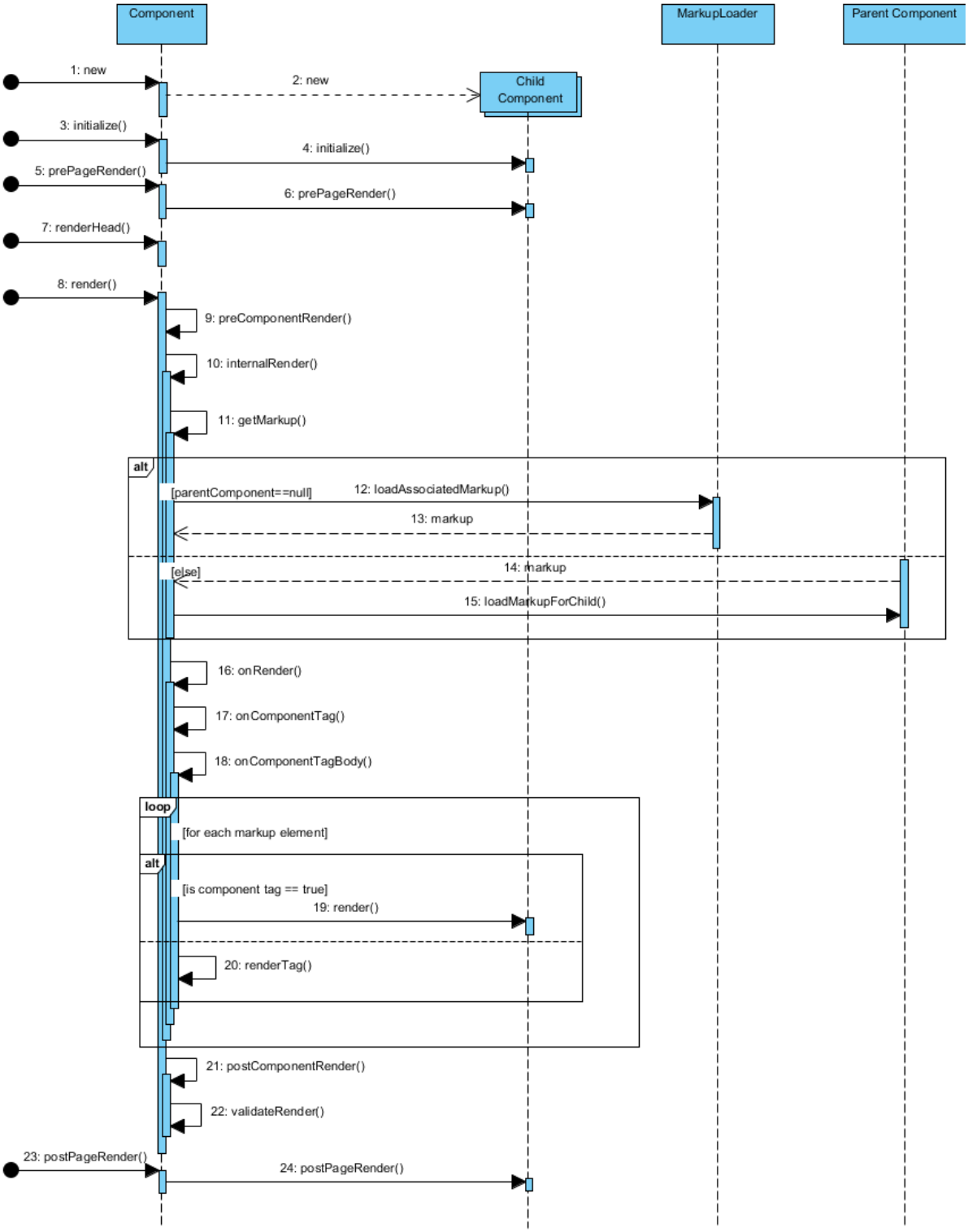

Figure 44 Sequence diagram showing the component lifecycle

The *Component* class is abstract and specialised by two classes, *WebComponent* and *MarkupContainer. WebComponent* is used for components that must be the last child node in the hierarchy; *MarkupContainer* is permitted to have child nodes. These classes and their associations are shown in Figure 45.

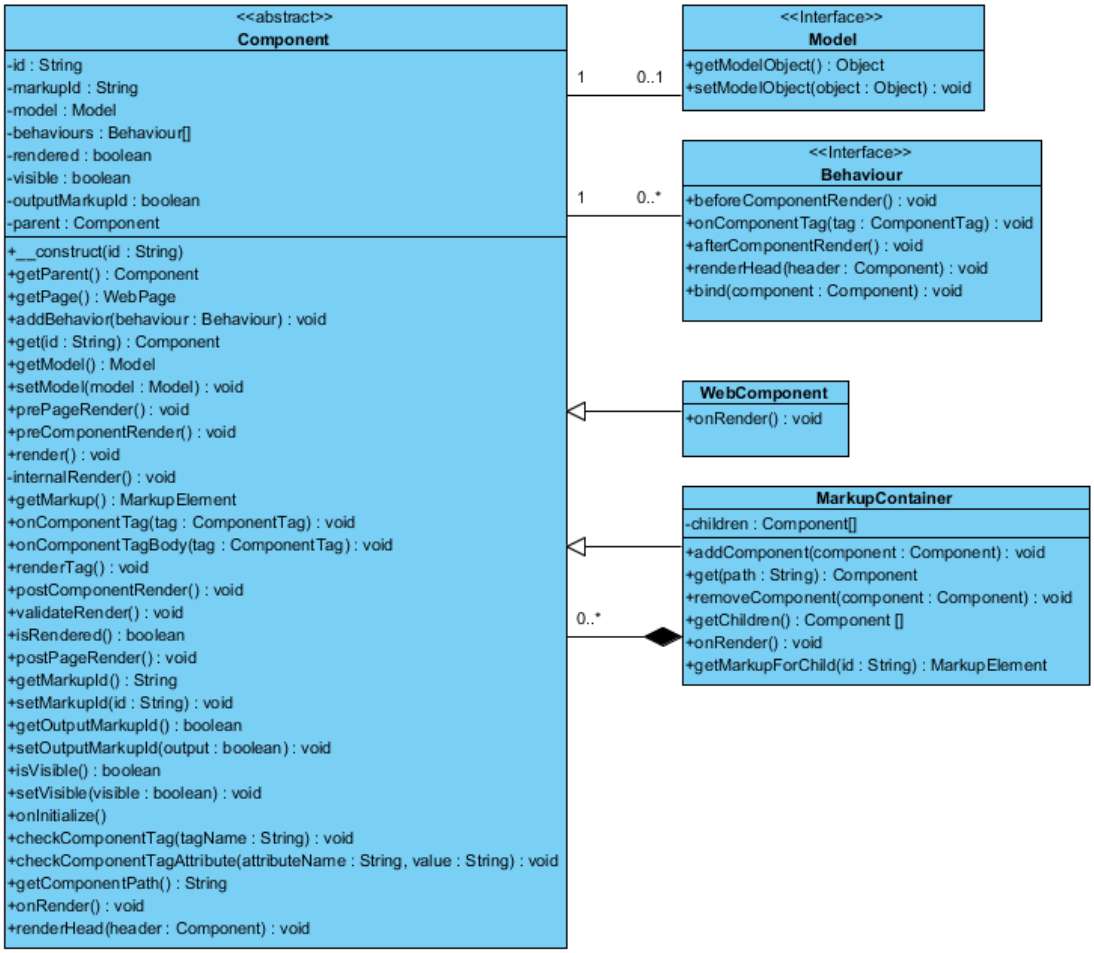

Figure 45 The main component, model and behaviour classes

As described previously, *onInitialize()*, *prePageRender()*, *render()*, *preComponentRender()*, *postComponentRender()* and *postPageRender()* form the various lifecycle steps for a component. Pre and post component render are not invoked by found messages, but instead are part of the internal render process. During this internal render process, the *onRender()* method is called; this is abstract in *Component* thus enabling the implementation to be different across sub classes. The *onRender()* method performs the physical rendering process of writing to the response object. It is different however in *MarkupContainer* than in *WebComponent* as the former renders child tags, the latter does not. The main manipulation methods are *onComponentTag()*, which is called when the component is about to write its open tag to the response. The *ComponentTag* object (see A5.4) is passed so it may altered if required. Once the open tag has been rendered, *onComponentTagBody()* is called. It is *onComponentTag()* that invokes *onRender()*.

Components are able to validate the tag they are associated with using the *checkComponentTag()* and *checkComponentTagAttribute()* methods. These would typically be called from *onComponentTag()* and throw exceptions if the component tag object does not contain the required values. This facilitates, for example, a text field component to specify that it may only be associated with an input tag with a type attribute value of text.

Components may be specified and retrieved within the hierarchy through *get()* and *getPath(). getPath()* returns a string representing the components position in the hierarchy; it is formed of all of the identifiers of the components in the hierarchy above it, followed by its own, separated with a colon. For example, *content:box:link* represents a component with an identifier of *link*, which is a child of a component with an identifier of *box*, which is itself a child to a component with an identifier of *content.* Similarly, the *get()* method performs the inverse; it accepts a component path string and locates the component that matches it. It is these methods that enable the component path to be retrieved for insertion into a URL, and for that component to be located in a subsequent request using the path string so that its callback may be invoked.

Behaviours are stored within an array, internal to the component they are added to. When a behaviour is added, its *bind()* method is invoked, passing along the component it is bound to thus allowing the behaviour to alter the components state if required. Within the behaviour, *beforeRender()* and *afterRender()* are called at pre component render and post component render respectively. The methods *onComponentTag()* and *renderHeader()* within the behaviour are called by their namesakes within *Component*. To permit this to occur, any component sub classes overriding any of the lifecycle methods are required to invoke the parent implementation.

A5.4 Associated Mark-up

The mark-up loader will locate an HTML file for a component class by using reflection, to determine the directory in which the class is located and then searching for an HTML file with the same name as the class. The contents of the file will be run through an XML parser, creating for it an object representation; like the component hierarchy, the mark-up object will be compositionally hierarchical, as shown in Figure 46. The class structure has been excessively abstracted to allow the same parsing and object structure to represent any XML.

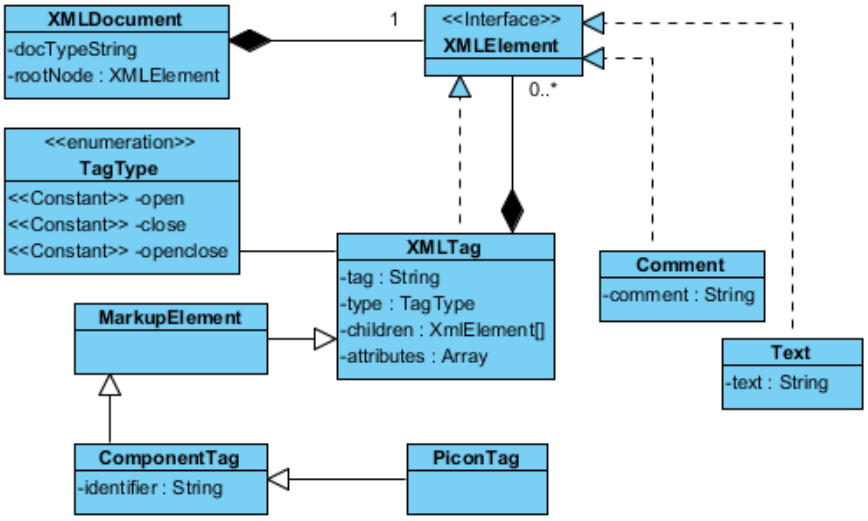

Figure 46 Class diagram showing the mark-up classes and their relationships

The parser will create, for most HTML tags, a *MarkupElement*, *Comment* or *Text* object. In the event an identifier is found on the tag, a *ComponentTag* object is created instead. *PiconTag* is created to represent special tags.

An identifier on a tag will take the form of an additional attribute, *picon:id,* Figure 47. The attribute of course does not exist, and is therefore invalid syntax. This can be rectified through the use of an additional DTD specification, which will be used in conjunction with XML namespace prefixing.

```
<a picon:id="link">Click here</a>
```

Figure 47 HTML tag with an identifier

```
<html>
<head>
<title>Page</title>
</head>
<body>
<h1>Title</h1>
<picon:child />
</body>
</html>
```

```
<html>
<head>
<title>Sub class Page</title>
</head>
<body>
<picon:extend>
<h2>sub heading</h2>
</picon:extend>
</body>
</html>
```

Figure 48 HTML showing the special tags for mark-up inheritance

The use of this prefix also enables special tags to be created. To facilitate mark-up inheritance, the parent HTML will use a *picon:child* tag, whilst the child HTML will surround its extension mark-up with *picon:extend* tags, Figure 48. This stipulates the manner in which the mark-up is to be merged; it also allows both HTML files to be valid, containing *html* and *body* tags. When merging the two sources, only the tags within the *picon:extend* tags will be placed into the position of the *picon:child* tag, thereby not duplicating the *html* or *body*.

Panels and borders, which are to be used for extension classes, also have associated HTML files. As these are merged with the HTML from the webpage associated mark-up, the same wrapping tags (although in this case *picon:panel* and *picon:border*) are used to specify the mark-up for the component, leaving behind tags that would otherwise be duplicated, Figure 49. The associated mark-up of both panel and border is rendered into the tag from the parent mark-up that the component is associated with. The panel replaces any pre-existing content from that parent tag. The border on the other hand wraps around it, achieved by initially replacing the original content but inserting it back in the location of the *picon:body* tag.

```
<html>
<head>
<title>Sub class Page</title>
</head>
<body>
<picon:panel>
<p>Panel contents</p>
</picon:panel>
</body>
</html>
```

```
<html>
<head>
<title>Sub class Page</title>
</head>
<body>
<picon:border>
<p>Border contents
<picon:body/>
</p>
</picon:border>
</body>
</html>
```

Figure 49 HTML illustrating the panel and border tags

A5.5 Component Classes

There are far too many component classes to list and document in entirety. This section looks at some of the more important classes. The diagrams in this section in many cases show only a subset of methods.

Main Components

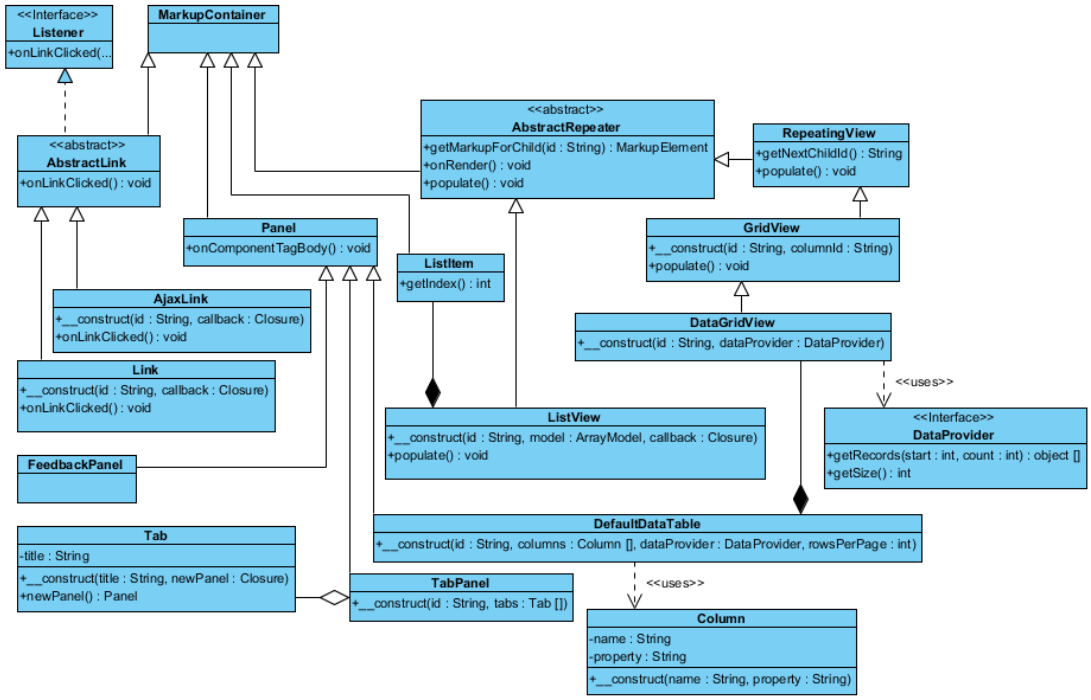

Figure 50 Class diagram showing the main component classes and their relationships

The functionality for lists and repeaters has been abstracted into *AbstractRepeater*, Figure 50. This overrides *getMarkupForChild()* causing any child components to be passed, when attempting to locate their associated the mark-up, the same mark-up associated with the parent repeater component. This causes the mark-up to be repeated for each child of a repeater. *ListView* instantiates a *ListItem* for each iteration of its array model, and adds the list item as a child of itself. It invokes a callback each time it does this, to populate any child components within its associated mark-up. By extending *AbstractRepeater*, *RepeatingView* children obtain their mark-up in the same altered way; these children are added by the developer though rather than model object iteration. *GridView* is an alternate *RepeatingView* that creates a repeating view inside of each child that is

added, causing the creation of a grid rather than a list. Consequently, *GridView* requires two IDs for associated mark-up tags, one for the row and one for the column. *DataGridView* goes a step further by creating rows and columns from a data provider. It is this *DataGridView* that forms the *DefaultDataTable*, which is a panel comprising the *DataGridView*, to show the table, and also a number of other components that are not shown in the diagram which display column headings and pagination controls.

Both *Border* and *Panel*, also shown in Figure 50, have their own *onComponentTagBody()* implementation, so that they may render their own associated mark-up instead of that inherited from the parent. *Panel* is extended to create *TabPanel*, which consists of *ListView* of *Link*s to show tabs and a single panel which is changed using the *Tab* method *newPanel()*. This component enables the creation of a tabbed interface.

The *Listener* interface, shown in Figure 50, is implemented by any component which needs to have a callback function invoked on a callback request. As the callback function exists only at runtime, it cannot be directly referred to; instead any implementing classes should invoke the associated callback when the *onEvent()* method is called by the request target during the callback request.

Form Components

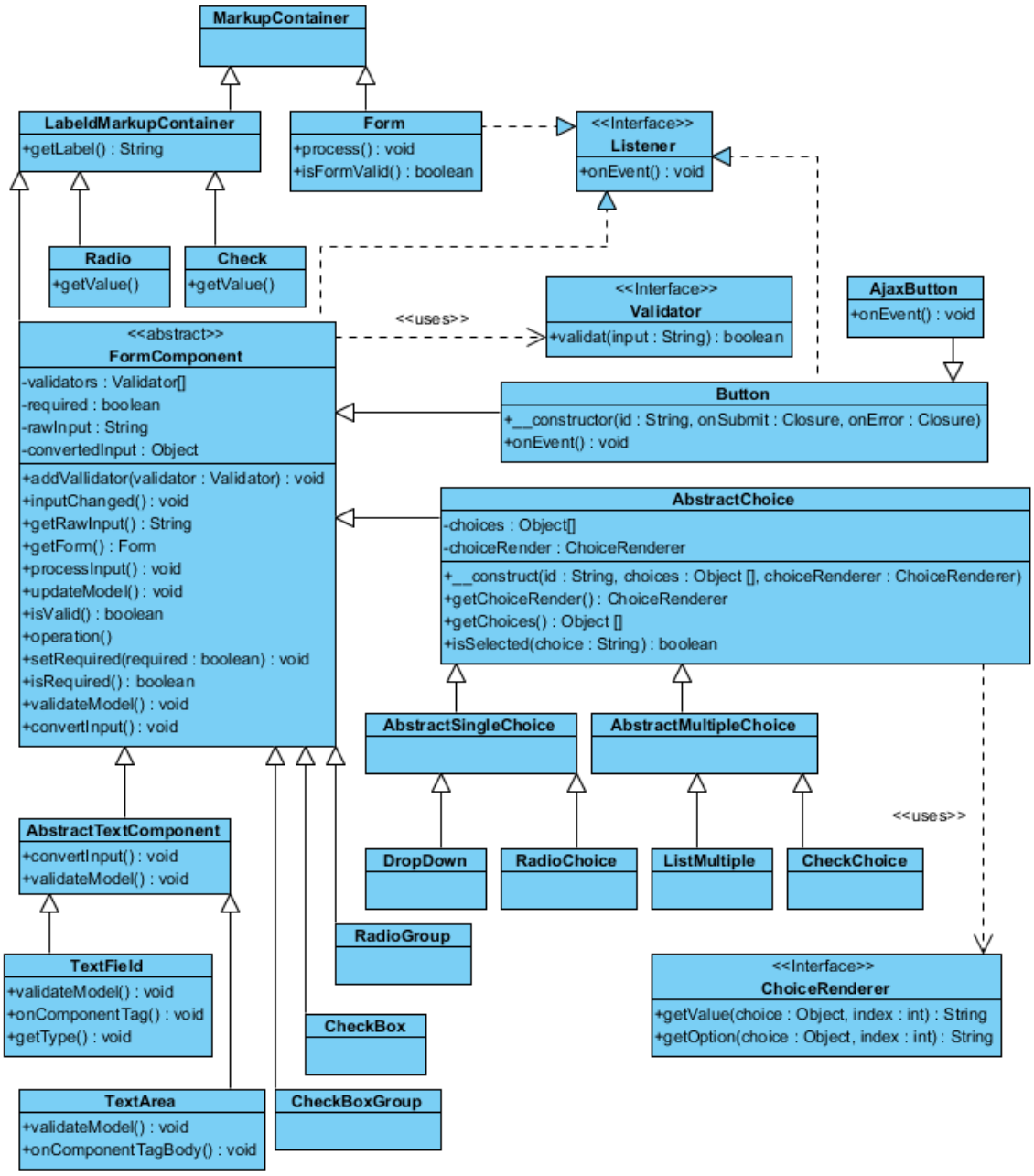

Figure 51 Class diagram showing the form component class, their relationships and dependant classes

All form components have had their functionality abstracted to an abstract *FormComponent* class, which deals with processing and validating input, and updating models. A form is a listener and when the *onEvent()* method is invoked, it locates any form components in the hierarchy below itself, invoking the *inputChanged()* method. This method extracts the posted parameter from the request that corresponds to the form component and stores it in *rawInput*. Next, the form invokes *convertInput()* which uses the raw input to extract the actual value for the form component. In the

case of a text field for example, if the data type was set to integer, the raw string will be converted into an integer, which will become the form component value. In the case of a choice component, which accepts a list of choice objects, the string input will be matched with the object it corresponds to and the corresponding object will become the form component value. Next, the form will invoke any validators on the form components, testing each one afterwards to ensure it is valid. If all of the form components on a form are valid, the *updateModel()* method is called, to push the form component value into the model object. The form then invokes the *onSubmit* associated callback, which is expected to deal with persisting the newly updated model objects. In the event that one or more of the form components is not valid, the form will not call for the model objects to be updated and will simply call the *onError* associated callback.

Two components have been defined for a checkbox, *Check* and *Checkbox*. *Check* is not actually a form component, so does not process data. This is for use solely inside a *CheckBoxGroup*, as it is the group that will deal with input processing and model updating. *Checkbox* is a form component and is used when a checkbox needs to exist independently. Initially I planned to implement these as one class, but they are functionally quite different and need to be separated to remain cohesive. As a radio cannot be used independently, it does not have a duplicate form component version and exists only for use in a *RadioGroup.*

The *AbstractChoice* provides an abstraction for any component which generates automatically based on a passed array of choices. As this has the potential to be an array of objects, a *ChoiceRenderer* is required to generate a string which is to be used for the option and value of the choice when rendered on the resulting page.

Each of the form components implements a *validateModel()* method. This is required to ensure that the model the form component is using is of the correct type. A *CheckBox* for example, can only use a boolean model.

A5.6 Models

A model works like a proxy, providing a wrapper for a domain object and offering an interface to retrieve and set the internal object. The domain object is wrapped in this way to permit additional functionality to be included. Only the *BasicModel* and *ArrayModel* actually wrap a physical object, and are only to be used in circumstances where model object duplication is not of concern. *PropertyModel* and *CompoundPropertyModel* use a target object and a property string to retrieve the domain object through reflection. They do not actually store the domain object; this is important for serialisation, as it enables a domain object to be stored in only one place but be accessed by many other objects by proxy.

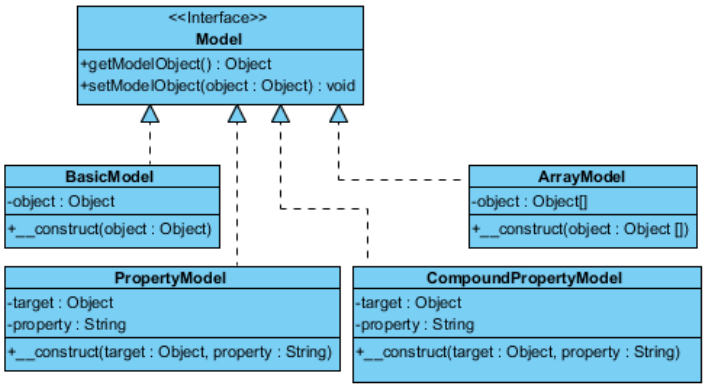

Figure 52 Class diagram showing the model classes

A5.7 Context Container

A class will declare itself as a resource with class level metadata in the form of either a *Service* or *Repository* annotation. The name of the resource will be the name of the class, unless specified as an annotation argument. Resources dependencies will be declared through the use of the *Resource* annotation at property level.

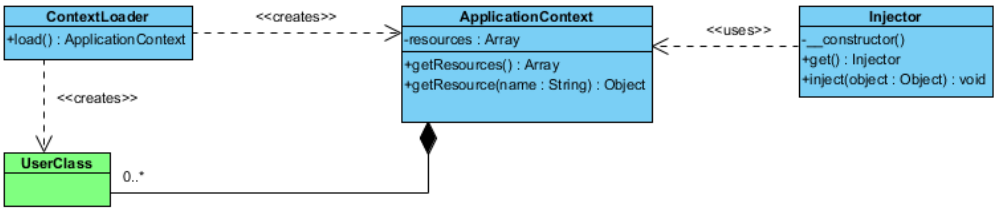

Figure 53 Class diagram showing the context container classes and their relationships. Framework classes are blue, user classes are green

The injection process is run individually on each resource. Reflection is used to access the object properties of the resource and detect the dependency annotation. If one exists, the corresponding resource is located and set as the value of the property. The injector is called by the context loader, however, it can be manually called for any object, enabling all objects to declare and have resource dependencies satisfied.

As resources are automatically instantiated and made available to the entire application, there is only ever one instance of each. Consequently, all resources will be stateless and therefore singletons by convention.

Dependency injection enables layering of resources, whereby each resource will be a dependency of resources in lower layers and will have dependencies itself of resources in higher layers. It is through these stateless, reusable, layered resources that service orientation is achieved. Any class written by the developer and declared as a resource is also expected to be autonomous, abstract and loosely coupled, fulfilling the true properties of a service.

A5.8 Database API

Using the data access pattern as a basis, and working with the pretence that a data access object (DAO) will be a resource, the framework breaks up database interaction into three layers, shown in Figure 54. *DAOSupport*, for which a developer will write extension classes, provides an interface to access the template. *DatabaseTemplate* provides methods to run a variety of different operations (insert, update, query etc). The template accepts not just a query string, but also arguments to insert

into those strings using the *sprintf* function. The argument insertion forms the query preparation. The template passes the prepared query to the *DatabaseDriver*. The driver actually runs the query on the database using PHP DBMS specific functions. The driver may return a result reference to the template if applicable. Insert and update operations return the inserted ID or the number of affected rows respectively. A query option requires a *RowMapper*. The mapper implements a single *mapRow* method which the template will invoke for each record of the result set. The method is expected to map the result set onto an object and return it. All the returned objects are placed into an array and returned by the template – facilitating fully customised object relational mapping.

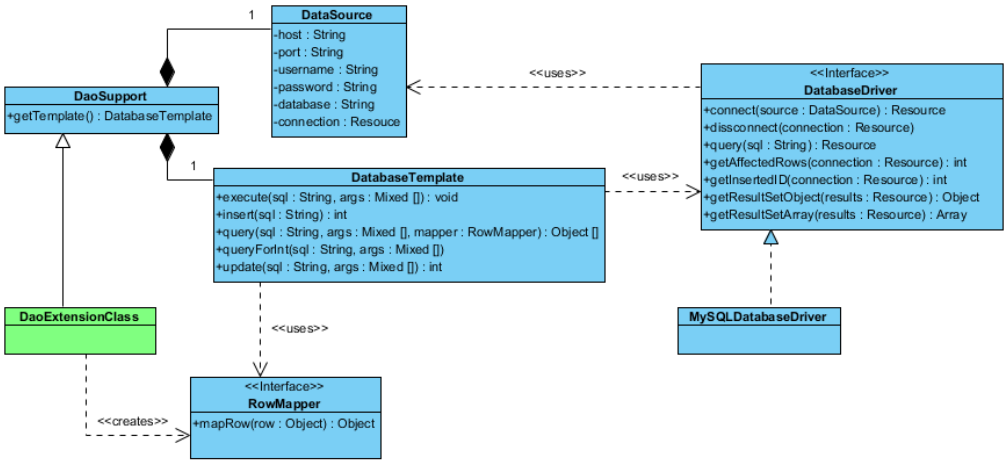

Figure 54 Class diagram showing the database classes and their relationships, framework classes are blue, extension classes are green

A5.9 Aspect Orientation

The formation of an aspect will be done through an extension class of *Aspect*. The extension class will override one or more of the advice methods and set the value of the point cut property. To instantiate a woven class, the new operator is not to be used, but instead the *ClassFactory* is called. The class factory locates (through the class scanner) extension classes of *Aspect*. The requested class is instantiated by the factory, and the *before*, *after* or *around* methods are invoked as needed for the constructor. The class instance is placed into a new instance of *Proxy* which is returned by the factory. The *Proxy* class intercepts all method calls with the *__call()* and *__callStatic()* magic

methods, and invokes the original target method of its internal object, but runs, at the appropriate time, any advice as defined by a point cut. Figure 55 shows the classes and relationships.

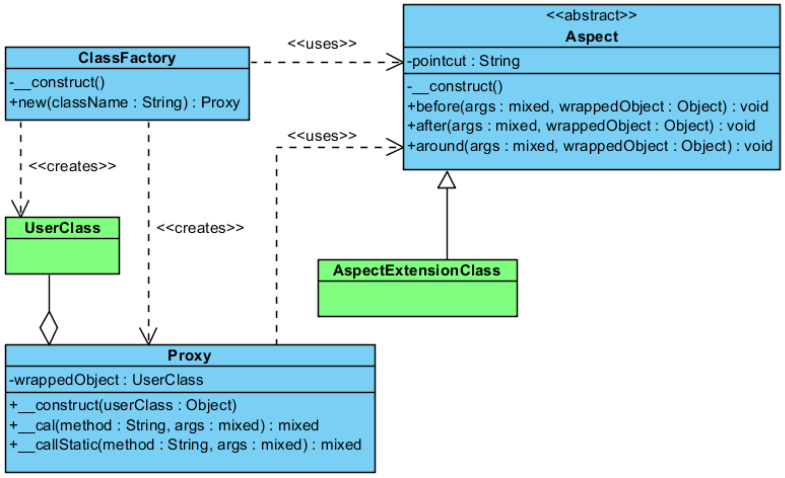

Figure 55 Class diagram showing aspect related classes. Framework classes are blue; user/extension classes are green.

A5.10 XML Configuration Structure

Within the scope of this project, the configuration file is required to contain settings for the homepage, application mode, start-up mode and data source. A sample configuration is shown in Figure 56.

```
<?xml version="1.0" encoding="UTF-8"?>
<piconApplication>
    <homePage>FrontPage</homePage>
    <mode>developddment</mode>
    <startUp>auto</startUp>
    <dataSource name="podiumDB">
        <host>localhost</host>
        <username>user</username>
        <password>P@ssword1</password>
        <database>mydatabase</database>
        <type>MYSQL</type>
    </dataSource>
</piconApplication>
```

Figure 56 Sample XML configuration for the Picon framework

The home page is the webpage extension class which is to be shown in the event the path of the URI does not contain a page name. The homepage is expected to be the name of a page in the page map.

The mode specifies in which environment the application is running in; for example, development, testing, staging or production. Although out of the scope for this project, this setting will be useful in the future for altering the applications behaviour for these different deployment scenarios.

The start-up specifies the initialisation process of the context contain. Although out of the scope for this project, I believe that the framework should enable a more manual approach for resource declaration detection specification or even manual declaration of resources in the configuration. This start-up value will facilitate this manual customisation although, for this project, the framework will support only auto start-up.

Finally, the data source value specifies the database that the framework should connect to. During the initialisation process, a data source object, as shown in Figure 54, will be instantiated for use by *DAOSupport*. Unlike the other options, multiple data sources may be specified.

A6. CMS Design

This appendix is a continuation of the CMS design in chapter six, and contains the business type model, system services, business services and service interactions, database design and user interface design.

A6.1 Business Type Model

This business type model (Figure 57) shows the information required by CMS, their attributes and relationships.

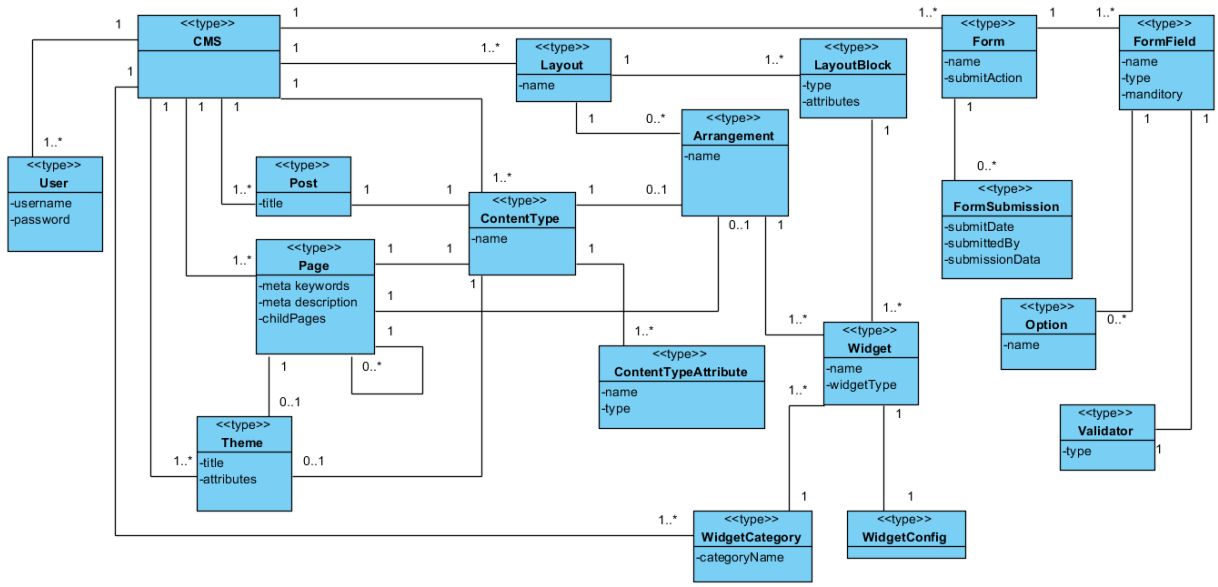

Figure 57 Business type model for the CMS

A6.2 Services

The business and system services, extracted from the business type model and use cases respectively, show the services the system will provide to perform the processes defined in use cases, and the CRUD operations for the business types. These services and their anticipated methods are shown in Figure 58, the service dependencies have been previously illustrated in Figure 15.

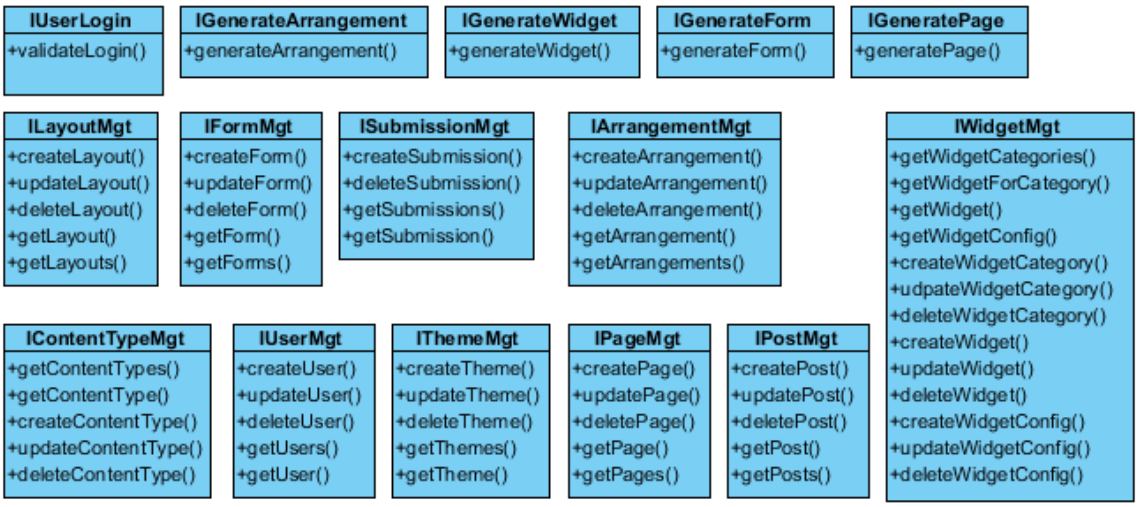

Figure 58 Class diagram showing the business and system services

A6.3 Service Interaction

The full service interaction is too extensive to show in entirety. This section illustrates some of the more important interactions. The sequence diagrams in this section are service sequence diagrams and are concerned, not with the inner workings of the services, but take a black box view of how the services interact with each other.

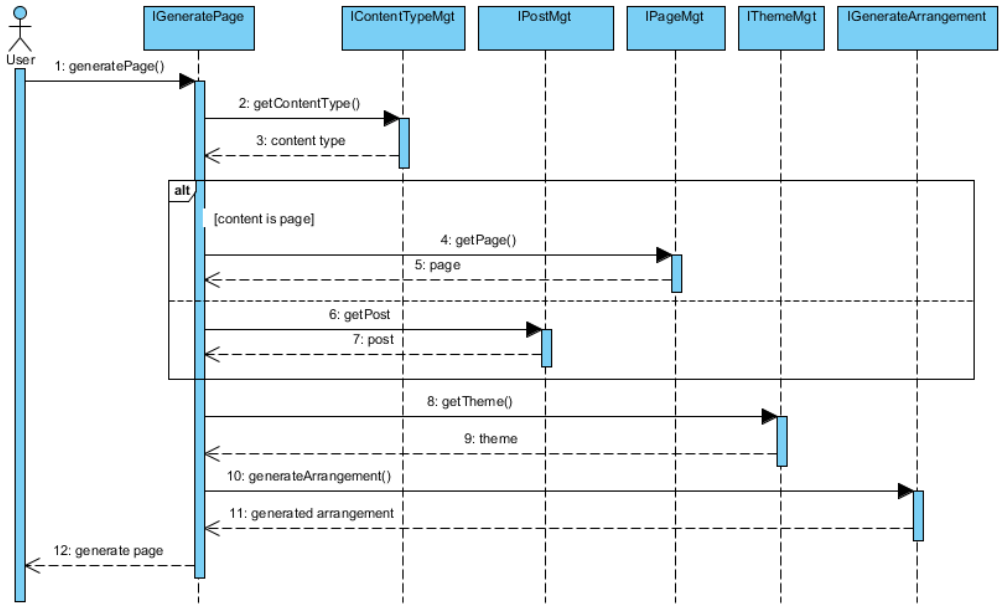

Figure 59 Service sequence diagram showing the process of generating a page

The process of generating a webpage begins with the retrieval of the pages content type. Depending on the type of content, the page or post is retrieved. The values from either the page or post are then used to populate the arrangement. This arrangement is retrieved in the same way, regardless of whether the arrangement was specified by the page or content type. The same retrieval process is also used for the theme. Once all these elements have been combined, the generated webpage is returned. The process is shown in Figure 59.

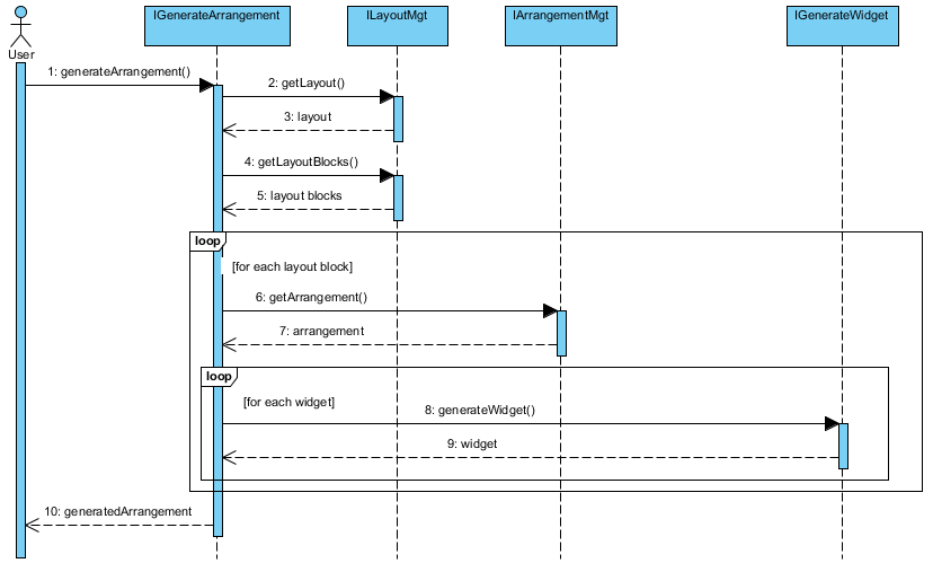

Figure 60 Service sequence diagram showing the process of generating an arrangement

The process of generating an arrangement begins by retrieving the layout, followed by the retrieval of the blocks that make up that layout. For each of the layout blocks, the arrangement is retrieved. This arrangement comprises the widgets added to the block and their order. Each of these widgets is then generated, producing the actual widget panel to output. Finally, the generated arrangement is returned. The process is shown in Figure 60.

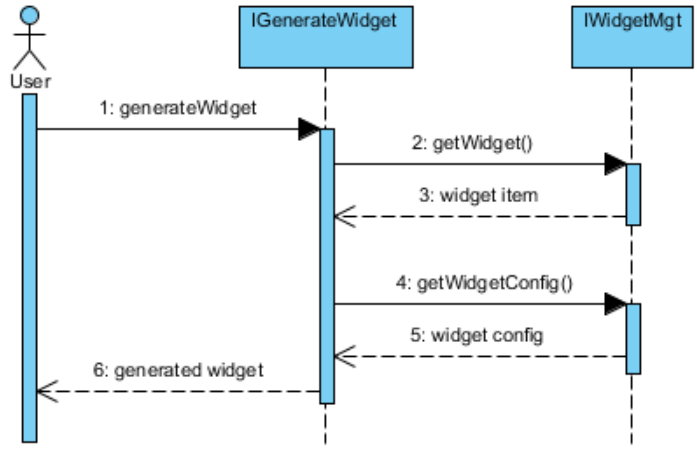

Figure 61 Service sequence diagram showing the process of generating a widget

The process of generating a widget begins with the retrieval of the widget item which will contain the information necessary to create the widget panel. The panel also requires a configuration to specify any widget instance specific options, this is also retrieved. Finally, the generated widget is returned. The process is shown in Figure 61.

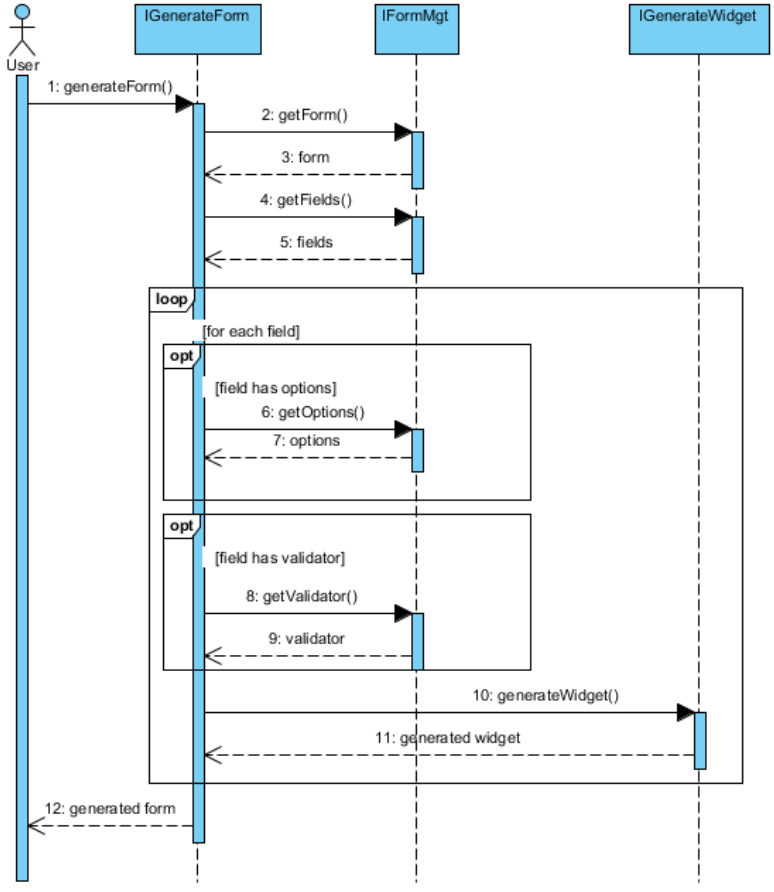

Figure 62 Service sequence diagram showing the process of generating a form

The process of generating a form begins with the retrieval of the form type and its fields. For each field, if options are required, they are retrieved. Similarly, if a validator is required, it is retrieved. The generate widget service is used to create a widget panel for each form field. These widgets are internally defined by the system. Finally, these widgets are placed into a form panel and returned. The process is shown in Figure 62.

A6.4 System Service Specification

The system services have been extracted from the use cases defined in appendix three. The services defined capture the functionality and processes of the system that is required to complete each of the use cases.

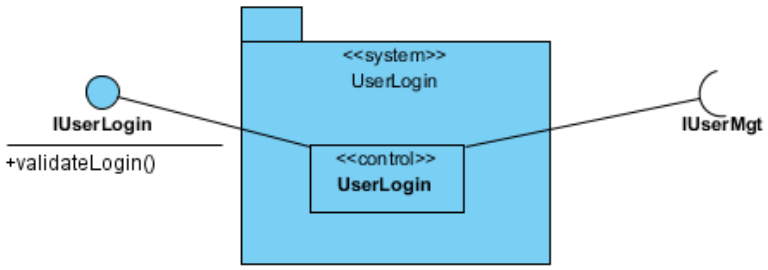

Figure 63 Component diagram showing the user login system service

The user login system service satisfies the login precondition for many of the use cases. It encapsulates logic to valid a user's credentials against users registered on the system, which are retrieved through the user management business service.

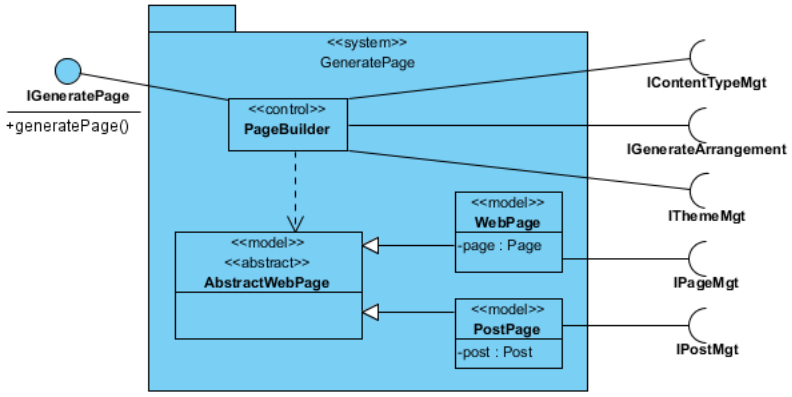

Figure 64 Component diagram showing the generate page system service

The generate page system service deals with the formation of a webpage for an end user to perceive. There are two kinds of webpage: one created to display the page type, another defined to show the post type. The content type business service is required to retrieve the content type. Although only a page type may have its own arrangement and theme, a post type does have a theme and arrangement; it is however inherited from the content type. Consequently the theme business service is required to locate the theme type. The generate arrangement system service is required, as it produces the actual arrangement to be displayed; however, generate page populates it with page specific content. Web page and post page require the business services page management and post management respectively.

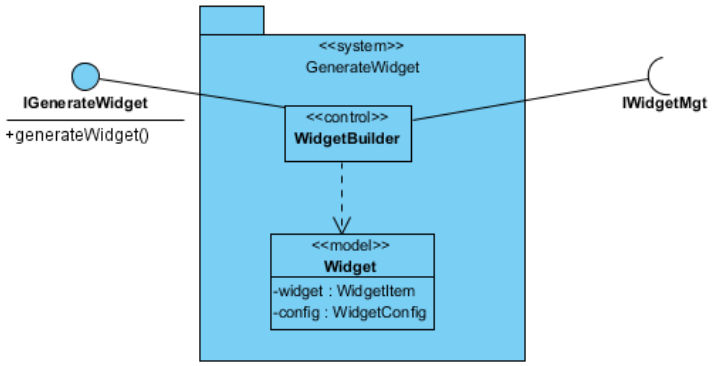

Figure 65 Component diagram showing the generate widget system service

The generate widget system service collates the widget and its configuration for rendering. Information is gathered about the widget type which is used to locate the required widget panel component; the panel component is then customised for the specific instance of the widget that is

to be shown with the widget configuration. Both widget information and its configuration are retrieved through the widget management business service.

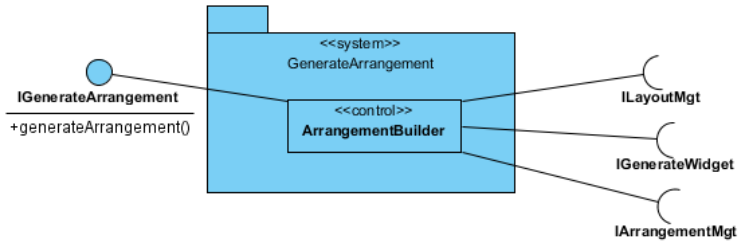

Figure 66 Component diagram showing the generate arrangement system service

The generate arrangement system service deals with the creation of a layout, and its population with an arrangement for output. The layout is retrieved via the layout management business service; the arrangement of widgets on that layout is retrieved using the arrangement management business service, and the widgets themselves are retrieved through the generate widget system service which actually produces the widget panel to be output.

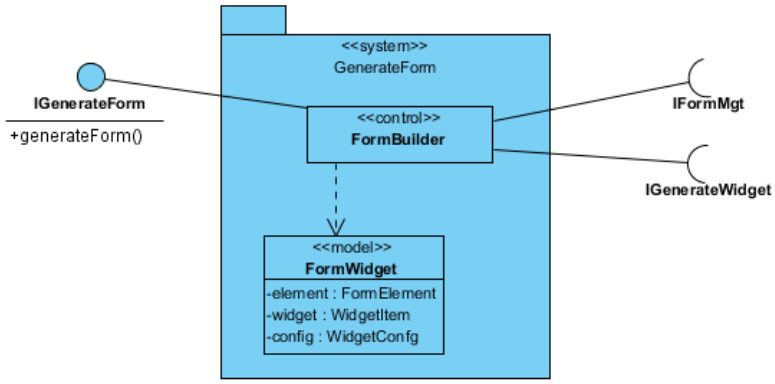

Figure 67 Component diagram showing the generate form system service

The generate form system service is responsible for producing a form for output. It uses the form management business service to retrieve form details, fields and details about those fields. It then utilises the generate widget service to generate the panel for each form component required. These form component widgets are defined internally within the system and are used only by the generate form service; they are not available for use as normal widgets.

A6.5 Business Service Specification

The business services have been extracted from the business type model. The functionality exposed by these services enables the information defined by the business type model to be retrieved, created, modified and removed.

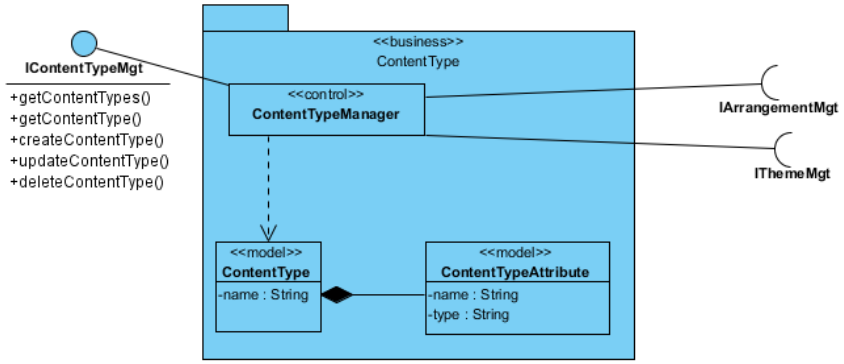

Figure 68 Component diagram showing the business service for content types

The content type business service deals with the CRUD operations required for working with content types. As a content type is associated with both a theme and an arrangement, it requires the use of the theme and arrangement management business services.

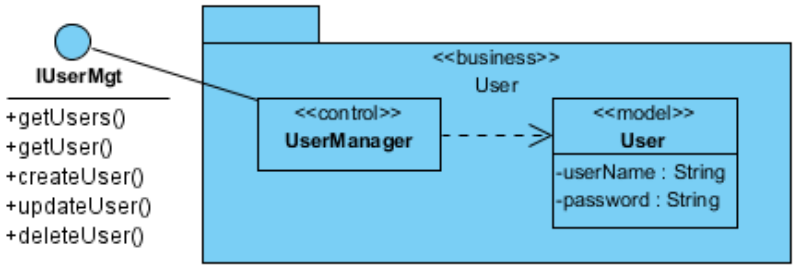

Figure 69 Component diagram showing the business service for users

The user management business service deals with the CRUD operations required for working with the user type. The user type consists of the username and password, as no other attributes have been defined. It is out of the scope of this project to deal with any more information or a greater granularity of users. It is due to this scope that the party pattern has not been employed.

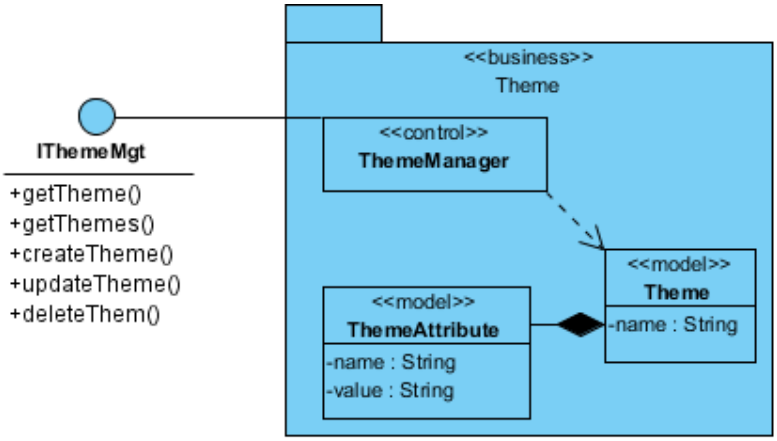

Figure 70 Component diagram showing the business service for theme

The theme management business services deals with the CRUD operations required for working with the theme type. The theme type has been normalised from the original type in the business type model, to separate out the name value pair attributes of which a theme is composed.

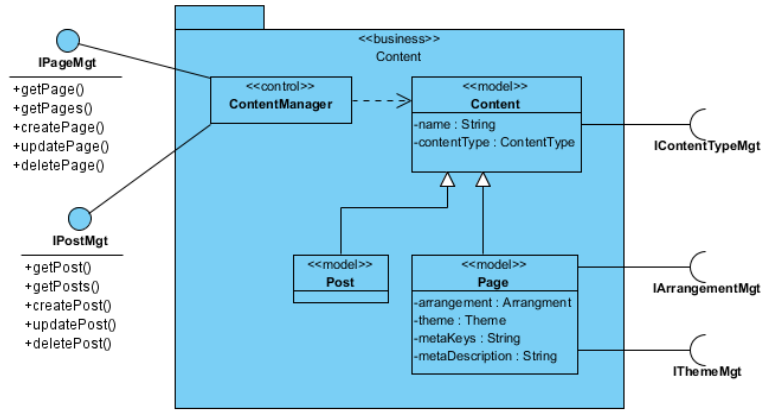

Figure 71 Component diagram showing the business Service for posts

The content manager realises both the page management business service and the post management business service. They have been combined as they would otherwise contain duplicate logic. Content management deals with the CRUD operations required for dealing with the page type and the post type. Both the page and post types contain similar attributes and have thus been generalised to create the new type: content. Content requires the content type business service, as it is applicable to both pages and posts, whereas only page requires the arrangement and theme business services, as only a page is permitted to have its own theme.

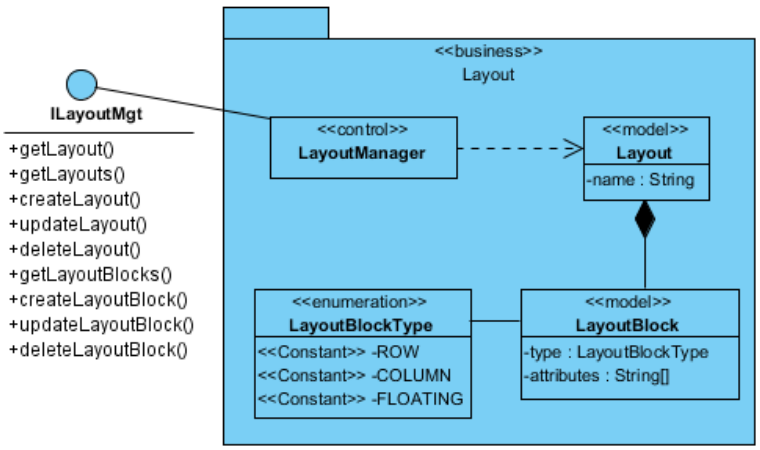

Figure 72 Component diagram showing the business service for layouts

The layout management business service deals with the CRUD operations required to work with the layout type. As there is a fixed quantity of layout block types available, a new enumeration type has been defined. Further to those defined in Figure 58, additional methods have been added for working with layout blocks separately.

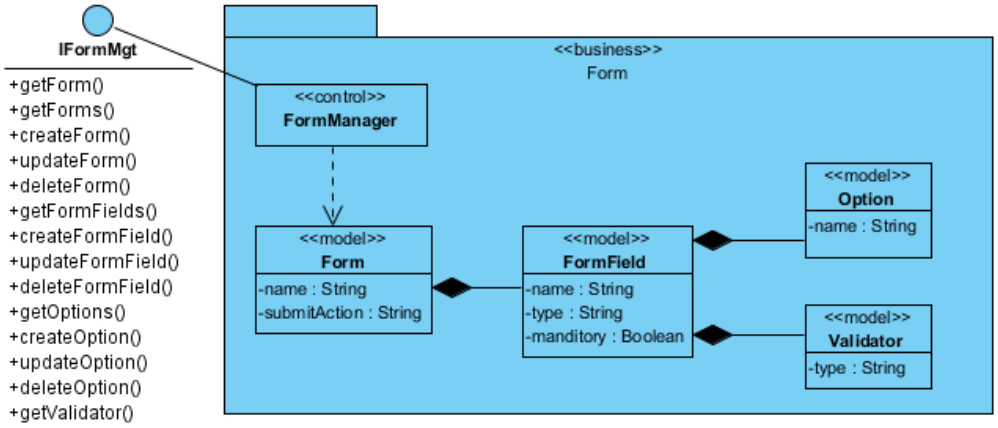

Figure 73 Component diagram showing the business service for forms

The form management business service deals with the CRUD operations required for working with the form type, its fields and their options or validator. Further to those defined in Figure 58, additional methods have been added for working with fields, options and validators separately.

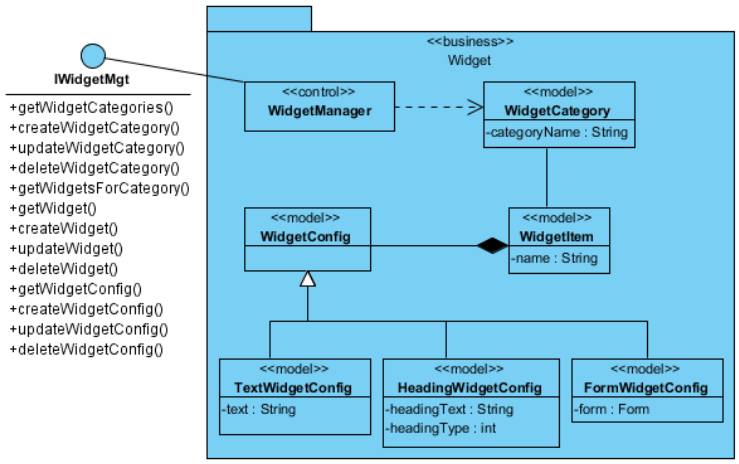

Figure 74 Component diagram showing the business service for widgets

The widget management business service deals with the CRUD operations required for working with widgets, widget categories and widget configurations. Further to the business type model, the widget categories have been normalised through specialisation, and three of the widget configuration types are shown. As more widget types are added, more widget configuration types will be added to configure them.

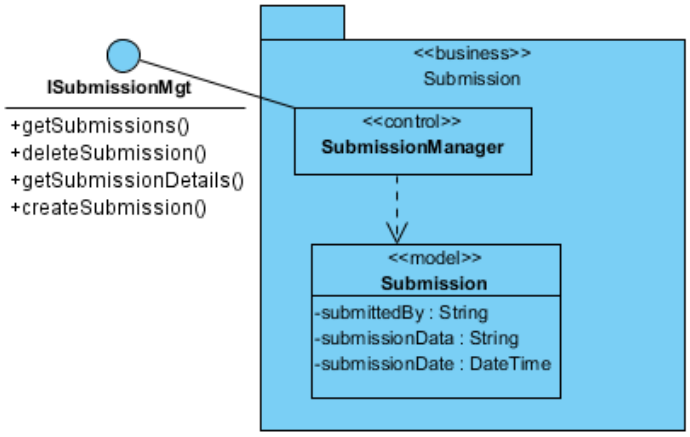

Figure 75 Component diagram showing the business service for submissions

The submissions management business service deals with the CRUD operations required when working with the submission type. The update process has been purposefully omitted as, once a submission has been made, it cannot be altered and may only be viewed or deleted.

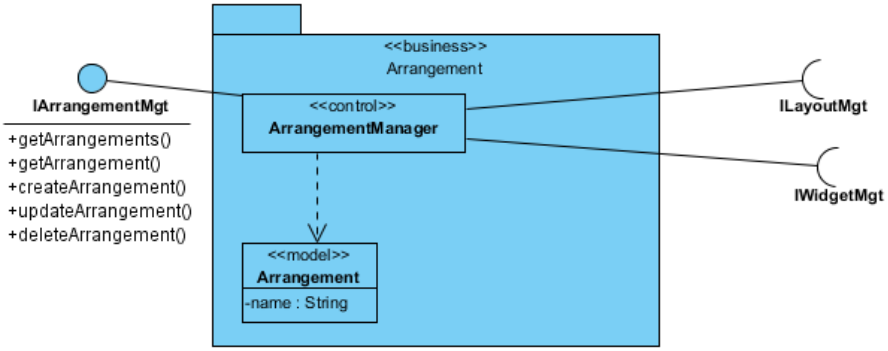

Figure 76 Component diagram showing the business service for arrangements

The arrangement management business service deals with the CRUD operations required for working with the arrangement type. Although seemingly a very simple type, arrangements are also associated with a layout and contain widgets, consequently creating the requirement for the use of the layout and widget management business services.

A6.6 Database Design

From the business type model shown in Figure 57, the information requirements can be extrapolated and used as the basis for an entity relationship diagram. The information structure is however quite complex and in some cases, to avoid data redundancy, it is desirable to utilise more advanced relationships such as inheritance and composition. Figure 77 shows an enhanced entity relationship diagram that illustrates these more complex relationships.

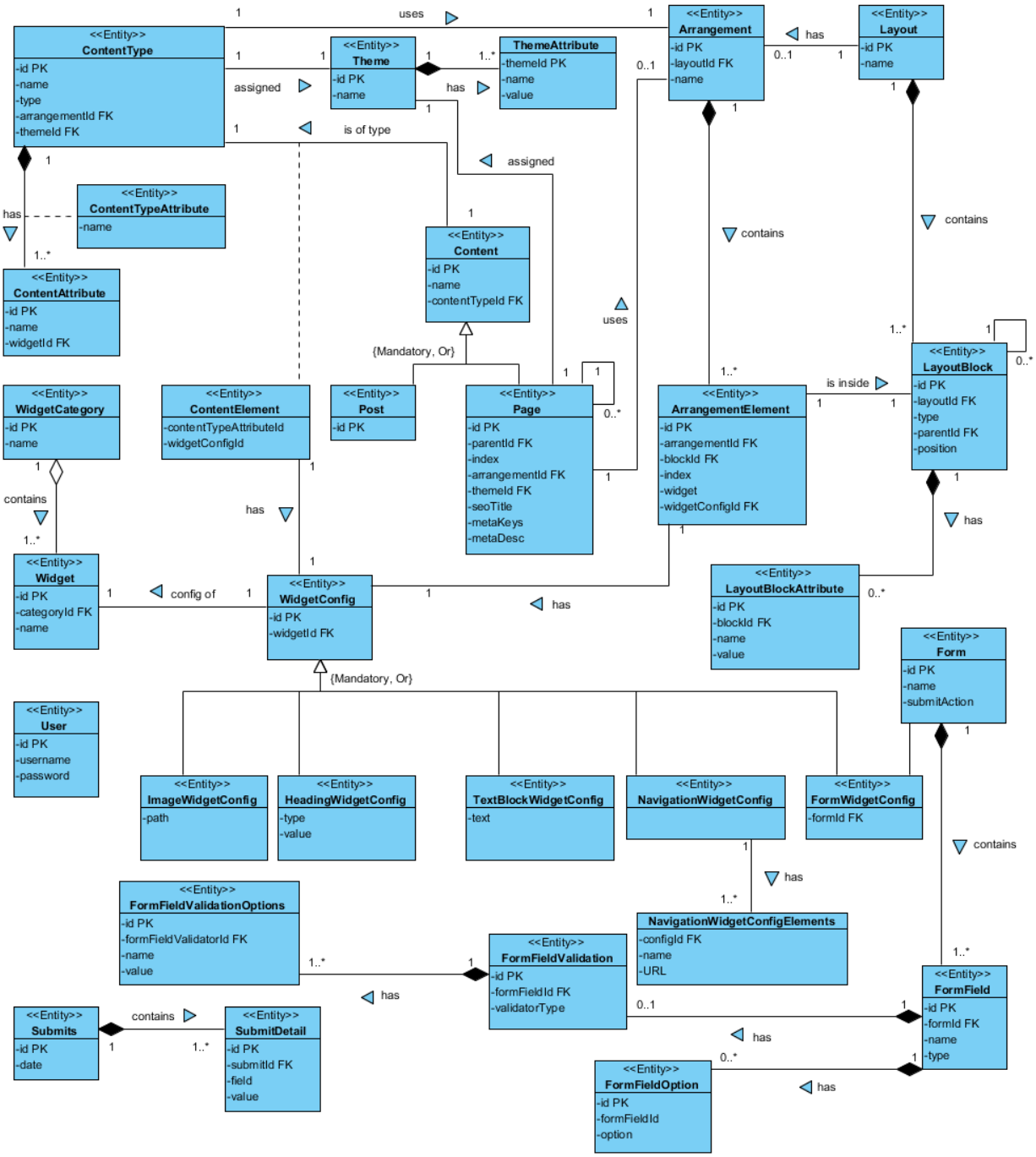

Figure 77 Enhanced entity relationship diagram showing the conceptual database structure

The structure present in the enhanced entity relation diagram cannot be implemented directly, as the complex relationships are unsuitable for the MySQL DBMS. Figure 78 shows a physical model, in which the entities have been broken up into several tables and constrained through primary and foreign keys, and multiplicities.

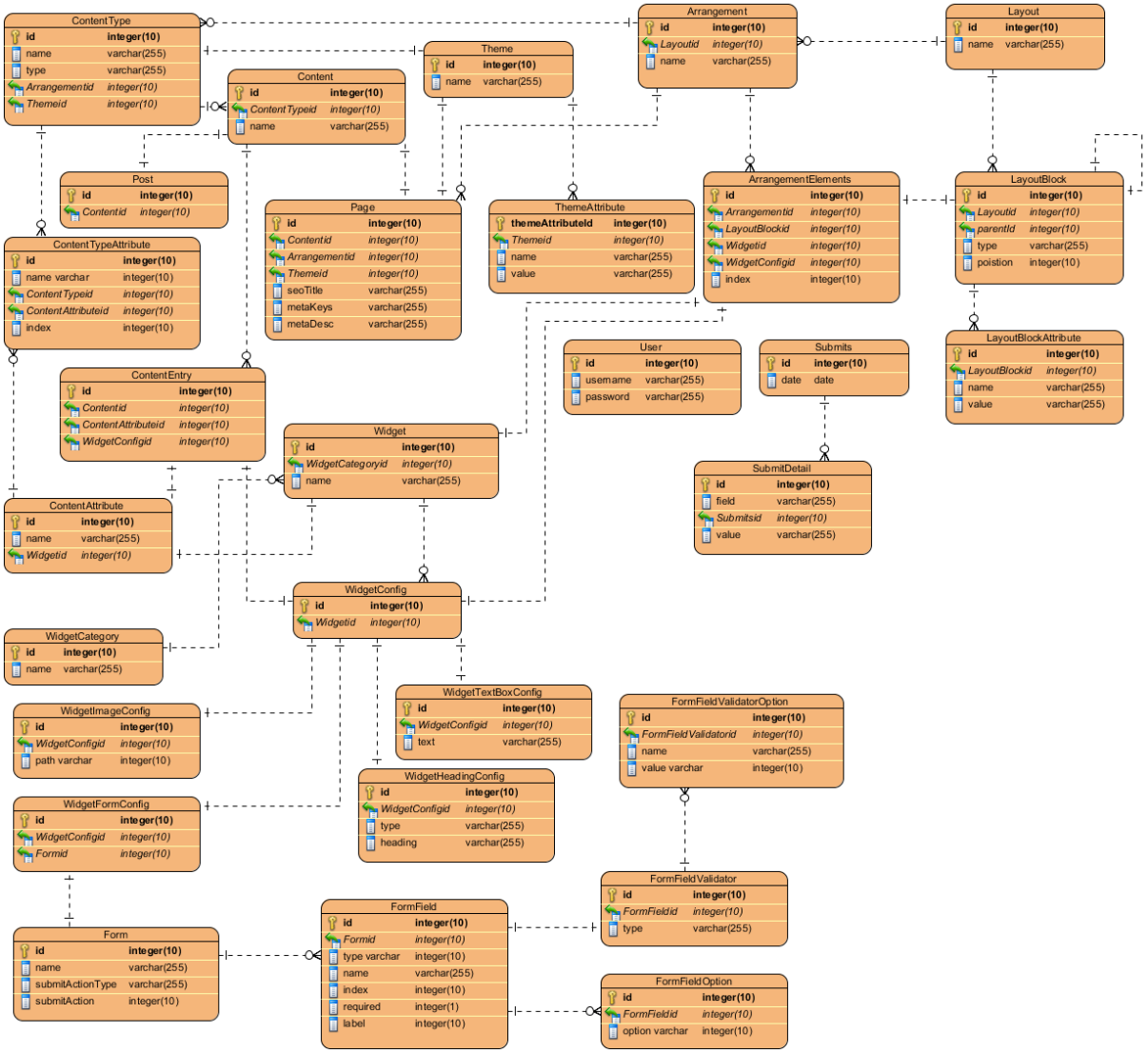

Figure 78 Entity relationship diagram showing the physical data structure

A6.7 User Interface Design

The login screen is the first that is displayed to a user when attempting to access the CMS administration portal, Figure 79. A user is required to enter their username and password. If incorrect, an error message is to be displayed immediately above the username field. This page uses the form, text field and button framework components.

Figure 79 Login page design

Although not specified by the requirements, the dashboard arose as a new requirement during implementation. It is the homepage for the CMS portal and is the first page displayed after logging in, Figure 80. It offers an overview of the system status, recent activity and links to common tasks. Although they are fixed, each of the blocks on the page is conceptually a widget which is constructed with a panel extension class.

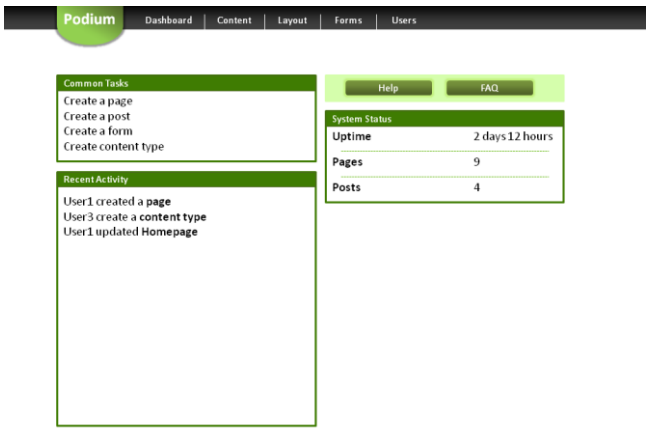

Figure 80 Dashboard design

The webpage list shows the organisational hierarchy of every page by indenting child pages, Figure 81. The blocks on the left side of each page entry indicate a drag handle. Each page entry can be dragged up, down, left and right to allow re-ordering within the organisational structure. This will be achieved using the *jQuery* nested sortable plug-in. Links are presented on the right side of each page entry for edition and deletion. A create page link is located at the bottom. Each hierarchical level consists of a panel containing a list component. The list iterates for each page on the hierarchy level, and also provides a placeholder where an additional instance of the list panel may be added if child pages exists, creating recursion.

Figure 81 Page list design

The post list works identically to the page list with the exception of the indentation, of which there is none, as there is no hierarchy, Figure 82. There is also no drag handle as posts may not be re-ordered. As a result of these differences, despite looking identical, the post page uses a data table and data provider.

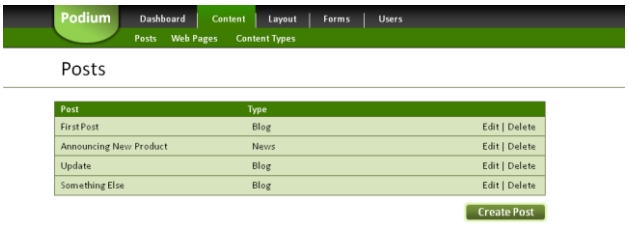

Figure 82 Post list design

Upon clicking the create page link, the create page setup is shown, Figure 83. The first step provides generic page options for title, parent page, content type, menu visibility, content type and metadata

for SEO. The parent page option allows for the creation of the organisational structure. The continue button does not create the page, but directs to the next step. If the form contains invalid data when submitted, an error message will appear above the first field via a feedback panel.

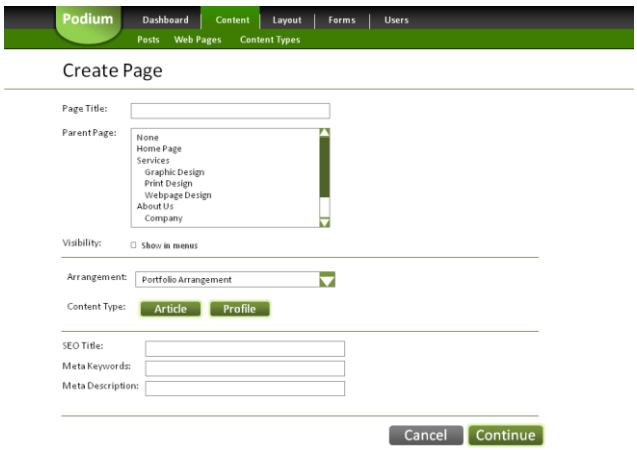

Figure 83 Create new page design, step one

The next step consists of a form generated from the content type previously chosen in step one, Figure 84. The create button will actually create the page and direct the user back to the page list. If the form contains invalid data when submitted, an error message will appear above the first field through a feedback panel.

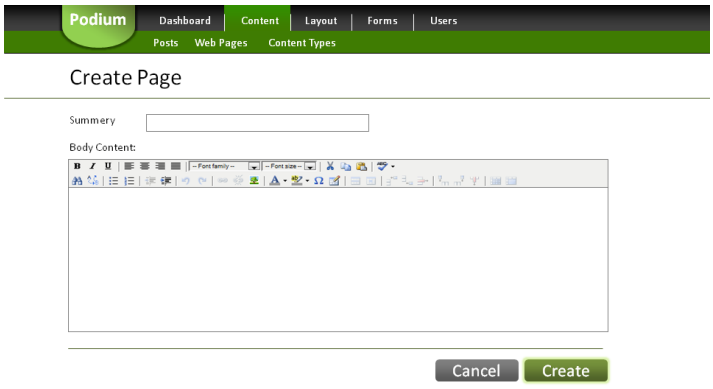

Figure 84 Create new page design, step two

Edit page is broken up into tabs, Figure 85. The first tab, page setup, shows the same options as those on create page step one, with the exception of the content type and arrangement fields which cannot be edited.

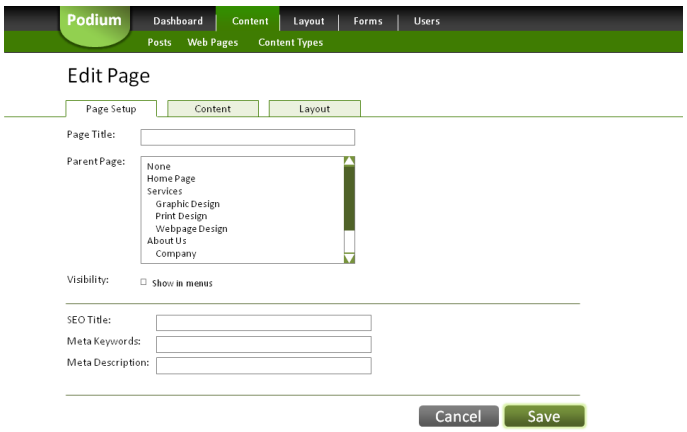

Figure 85 Edit page setup design

The second tab, content, shows the form fields generated from the content type, Figure 86. Although the content type itself cannot be altered, the values of the fields it generates can.

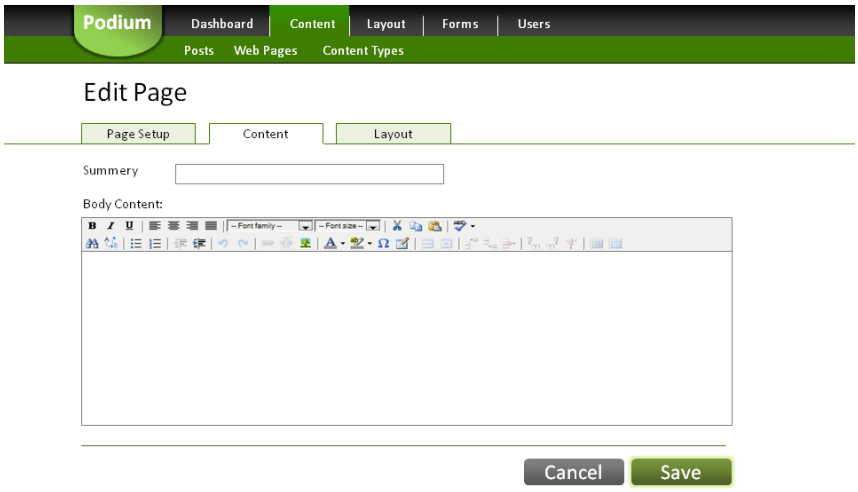

Figure 86 Edit page content design

The arrangement itself cannot be reselected; however the layout tab enables the widgets and their configurations to be edited, Figure 87. The pages existing arrangement is used to generate the default widget positions and configurations. The alterations here are applied only to the page being edited. This enables finely grained layout modifications at the page level. The watermarked layout blocks indicate the block has not yet been altered at the page level, and is currently inheriting from

the originally selected arrangement. A user must unlock the block in order to edit it and by doing so override the widget positions and configurations for that block on the page. At this level, the widgets that represent the content block are populated with the values set in the content type attributes on the content tab.

Figure 87 Edit page layout design

Layouts and arrangements are listed on the same page, illustrating their relationship to one another, Figure 88. There is only one level of indentation here, and there is no support for re-ordering as an arrangement cannot be moved from one layout to another. Edit and delete links are present for each layout and arrangement. Links for creating both layouts and arrangements are located at the bottom of the list.

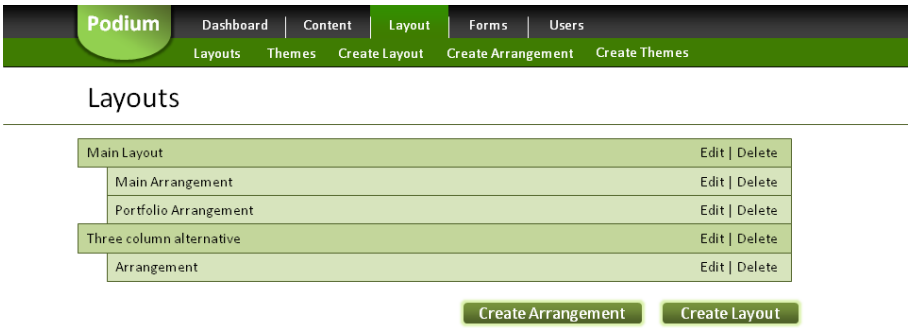

Figure 88 Layout and arrangement list design

The create layout page is directed to when a user clicks create layout, either at the bottom of the layout list or on the secondary navigation menu of the layout section, Figure 89. The page shows a

scaled wireframe view of the blocks of a layout. The toolbar below the title offers a button for each of the three types of layout block that can be created: row, column and floating block. When clicked, a new element will appear and the user must drag it into position. When the mouse hovers over a layout block its controls appear. On the left is a drag handle which enables re-positioning (floating blocks) or re-ordering (rows and columns). On the left is a delete button and also an options button. The options button opens an options modal window. Editing a layout will use precisely the same screen. Each layout block will be represented by a panel extension class to prevent any mark-up or code duplication.

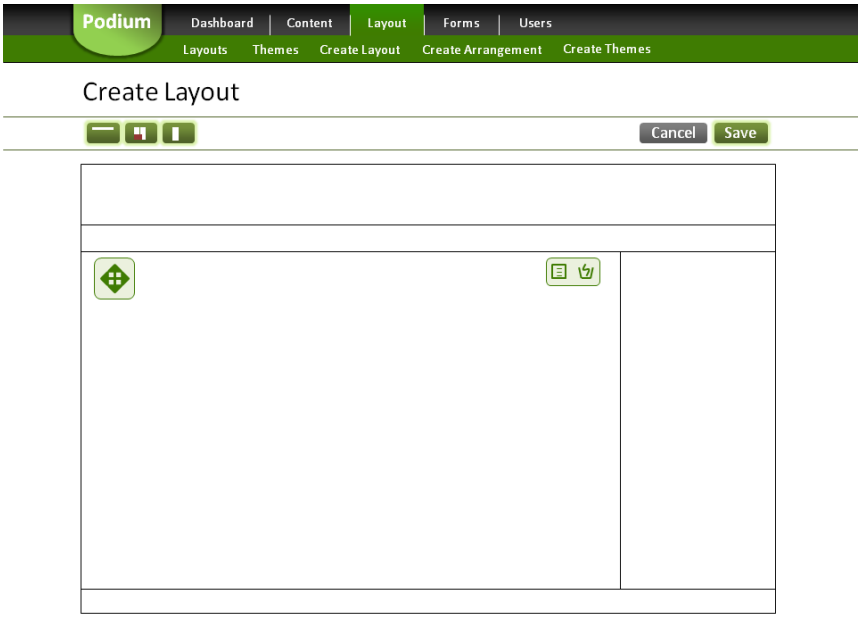

Figure 89 Create layout design

The options modal window shows various configuration options for the chosen layout block: background colour, border colour, border width, inner margin (CSS padding), outer margin (CSS margin), and a choice of horizontal or vertical stacking, Figure 90. The stacking option is applicable when creating arrangements for the layout and denotes the way in which widgets internal to the layout block are to be stacked.

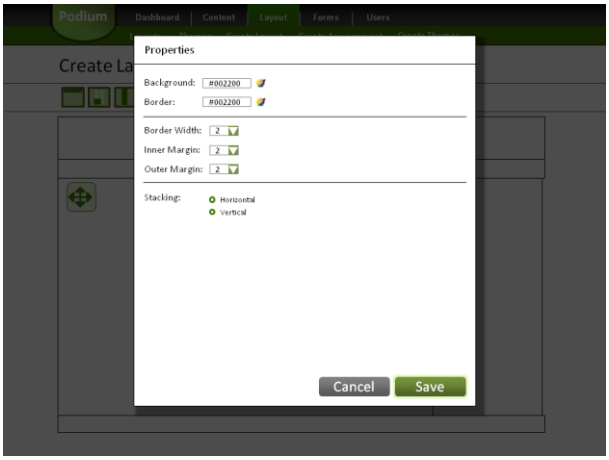

Figure 90 Edit layout panel design

Create arrangement uses the same toolbar layout as create layout, although it provides drop downs for each of the widget categories, Figure 91. When the mouse hovers over a widget category a dropdown appears below, showing all of the widgets within that category. The user clicks and drags a widget from the dropdown menu onto a layout block. The layout blocks are displayed in the same scaled view as on the create layout screen. Like the layout blocks, a widget has controls which appear when the mouse hovers over it. Firstly, there is a drag handle to re-order the widgets within a layout block or move to another block. There is also a delete button to remove the widget entirely and an options button which opens a configuration modal window. The modal window contains widget specific configuration options. Each widget will be a separate panel, as will its configuration options. The overall layout of the page will be generated with a nested repeater. The parent repeater will iterate for each of the layout blocks, adding a layout block panel for each iteration. These layout block panels will themselves iterate through each of the widgets they contain, adding the appropriate widget panel for each iteration.

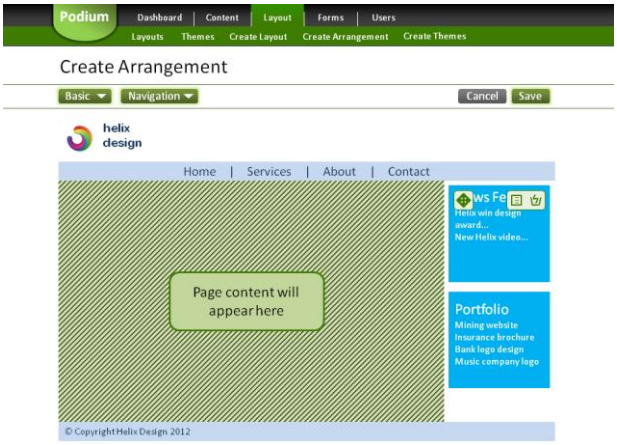

Figure 91 Create arrangement design

The theme list page shows all of the current themes available on the system, alongside edit and delete links, Figure 92. A create theme link is located at the bottom. This page will be constructed using a data table and data provider. The in use icon indicates whether the theme is currently applied to either a content type or page. As well as useful information, it also enforces the constraint that prevents the deletion of themes that are being used.

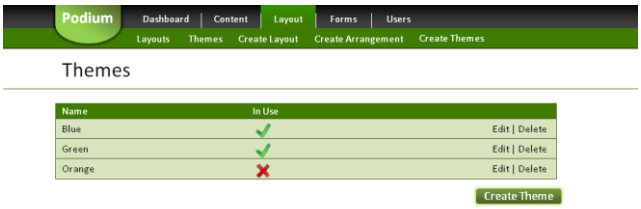

Figure 92 Theme list design

The create theme screen presents a tabbed layout to break up the available options into sensible groupings, Figure 93. It makes use of colour palette selectors, dropdowns, radios, and checks to hide the complexity that it actually configures and enforce the insertion of valid data. The edit theme screen is identical to the create theme screen.

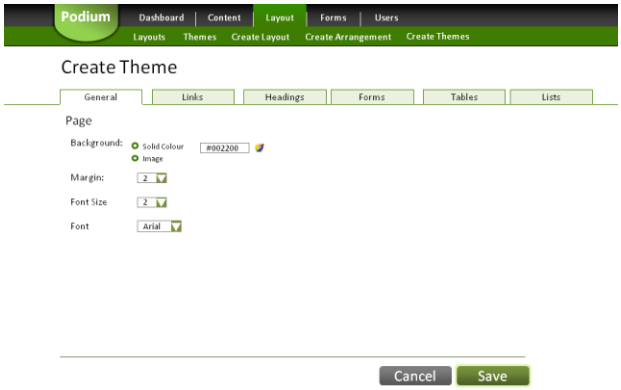

Figure 93 Create theme design

The content type list shows a list of all of the available content types, Figure 94. As with the other list pages, each entry has an edit and delete link and a create button at the bottom. The page will be generated with a data table and data provider. Once again, it enforces deletion constraints by indicating which of the content types are in use.

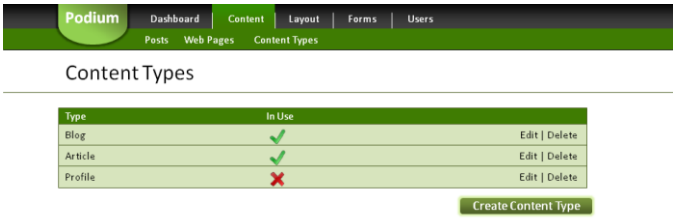

Figure 94 Content type list design

The content type create page gives name, arrangement and theme options, Figure 95. The primary interface, below these options, provides two panes: the left pane lists the available attributes, the right pane lists the attributes currently added to a content type. A user must drag available attributes from the left to the right pane to add it to the content type. As they do so, a modal window will appear so they may name the added attribute. This modal window does not disrupt the workflow as direction to another page would, but does disrupt the process enough to ensure that a user enters a name for the attribute. Each attribute in the right panel also includes a delete and edit option. Delete removes the attribute, and edit opens the same configuration modal window that was displayed upon initially adding the attribute.

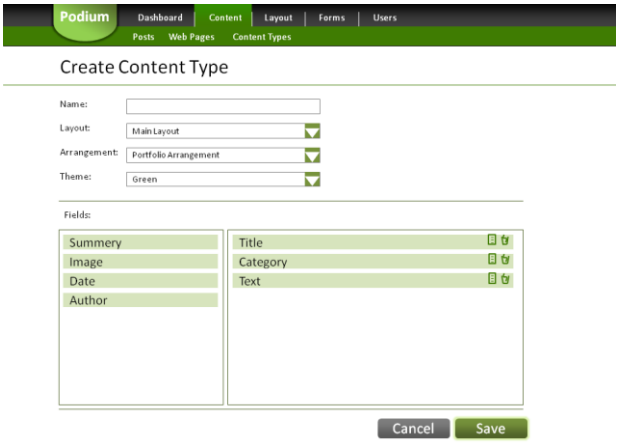

Figure 95 Create content type design

The edit content type page differs slightly by offering a tabbed layout, Figure 96. The layout and arrangement options have been removed as they can no longer be changed directly.

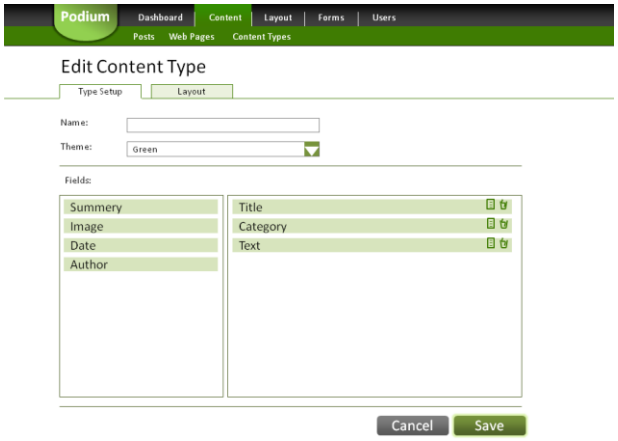

Figure 96 Edit content type design

The layout tab of the edit content type page enables the same arrangement edition as the page equivalent, Figure 97. Although the arrangement can no longer be directly changed, it can be altered. This operates in the same way as the arrangement editor at page level, enabling individual layout blocks to be unlocked permitting the reorganisation widgets and the alteration of their configurations. Furthermore, as the content type attributes are known at this level, they are displayed, although only as widget placeholders as, unlike the page arrangement editor, the value of the attributes is not known.

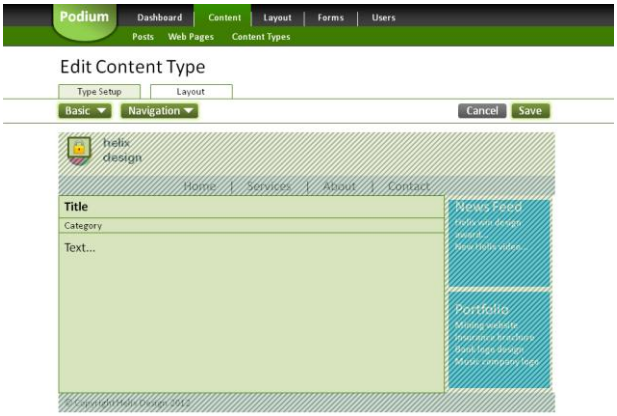

Figure 97 Edit content type layout design

The form list page shows a list of all of the forms that are available on the system, Figure 98. As with every other similar list page, this is achieved with a data table and data provider. Edit and delete options are presented for each entry on the table and a create button is present at the bottom. Again, to enforce the deletion constraint, an in use icon is utilised.

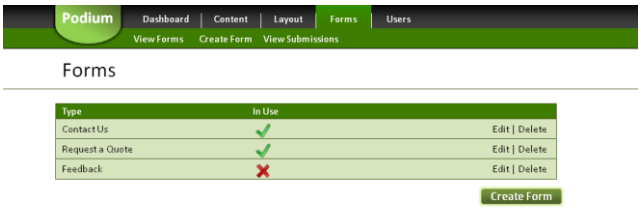

Figure 98 Form list design

In a similar manner to the content type create and edit pages, the create form page consists of two panes, one for available form field types and one to list the fields already added to the form, Figure 99. In a manner similar to the content type page, a modal window appears upon the insertion of a new field, forcing the user to configure it. The edit form page is identical to the create form page.

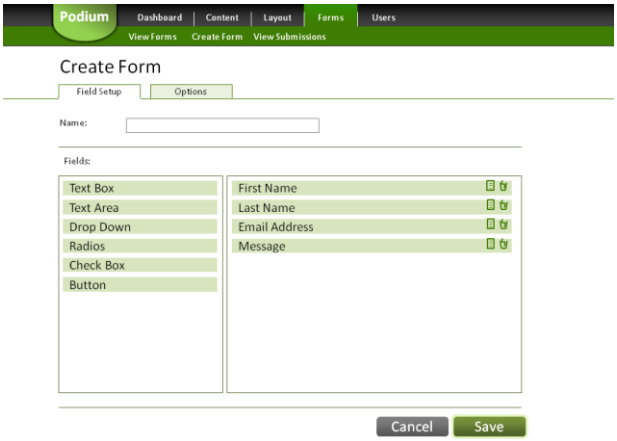

Figure 99 Create form design

The configuration modal window offers field type specific configuration options, Figure 100. A dropdown field for example will require the user to specify a list of options that are to be available to select from, whereas a text field will allow the addition of a validator. These validators will be derived directly from the framework.

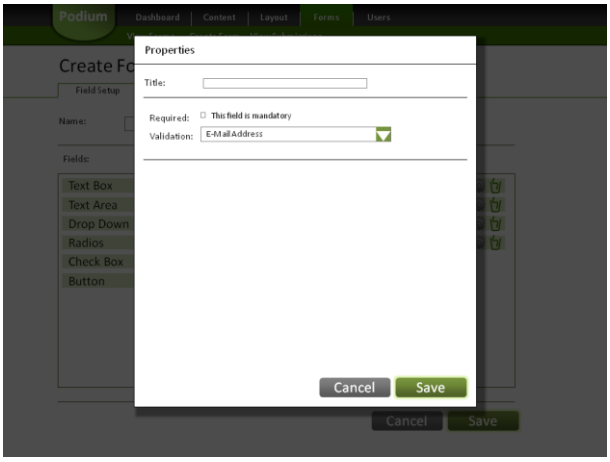

Figure 100 Form field properties design

The create form page uses a tabbed layout to separate out the submission actions so as not to disrupt the field editor. The second tab presents submission options, permitting the user to specify whether the end user is to be directed to a new page when they submit the form, or whether they are to simply be shown a message, Figure 101.

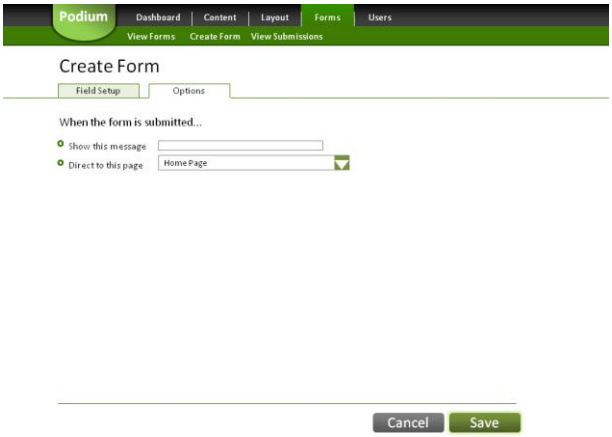

Figure 101 Create form options design

Any submission that an end user makes appears on the submission list page, Figure 102. This, as with other list pages, uses a data table and data provider. There is no edit or create option, as submission can only be created by submitting a form on the front end. Instead, a *read* link is provided allowing the user to view the details of the submission.

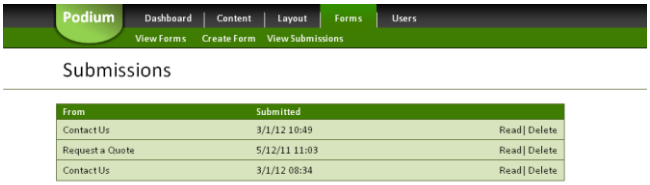

Figure 102 Submissions list design

There is no design for the front end as it is entirely the responsibility of the user to create it. It will however utilise precisely the same nested repeater approach, layout block panels and widget panels to display the page to an end user – the only difference being that the drag handle, and the options and delete buttons will not be present.

A7. Technologies

This appendix explores some of the technologies used during this project.

JIRA

JIRA provides issue and project tracking for teams, increasing the ease of project management and boosting productivity. (Atlassian Plc Ltd, 2011a) *JIRA* works through a web portal enabling users to create projects, and create various types of issues for those projects. A plug-in called *Green Hopper* enables Agile development in *JIRA*. *Green Hopper* allows for issues to be grouped into sprints and provides an array of tools for sprint planning and review. *JIRA* records massive amounts of statistics, which are collated into useful charts to monitor progress on a project or the productivity of an individual or team. In this project I used *JIRA* and *Green Hopper* to keep track of tasks and monitor my progress.

Subversion

Also known as *SVN*, Subversion is a version control system. (Apache Software Foundation, 2012) *SVN* enables historical versions of source codes to be preserved. *SVN* breaks up code storage into three main categories: the *trunk* which stores the current version of all files, *tags* which are used to mark releases or milestones, and *branches* which contain code branches enabling tangent development. Free *SVN* hosting is available on *Google Code* which I made use of for this project.

jQuery and jQuery UI

jQuery is a cross browser compatible JavaScript library which simplifies event handling, document traversal, event handling, animation and Ajax. (The jQuery Project, 2010) A separate project, jQuery UI, is built on top of *jQuery* and offers abstraction for interaction such as drag and drop, advanced animation and effects, and a number of themeable widgets such as buttons, modal windows and tabs. Both libraries have been used in this project for client side interaction, specifically the drag and drop on the layout and arrangement editors in the CMS, and the Ajax functionality of the framework.

Addendum

Addendum provides annotation support for PHP. It takes advantage of some recent improvements in the reflection API, by allowing annotations, syntactically identical to those in Java, to be placed in the doc blocks at class, method or property level. The annotation support provided by addendum was used in the framework to enable the use of class metadata. (Dilley, 2011)

JMeter

JMeter is an open source tool created with Java to enable performance and load testing for web applications. (Apache Software Foundation, 2012a) *JMeter* was used in this project for the testing of non-functional, performance related requirements. *JMeter* enabled the process of sending HTTP requests to both the framework sample application and the CMS to be automated, allowing large quantities of requests to be sent simultaneously to test performance when handling concurrent requests.

phpUnit and Code Coverage

The sooner that a mistake is detected, the faster it is to fix; *phpUnit* is a unit testing framework that facilitates this. (Bergmann, 2012) *phpUnit* allows the creation of unit tests that enable the white box testing of small fragments of code within an application to ensure that it functions correctly. *phpUnit* was used for the framework to test much of its core functionality. An additional tool, created by the same developer, *Code Coverage*, works in conjunction with *phpUnit* and detects the lines of code that are executed by the unit tests to determine how much of the code is being executed. Ideally, all code should be tested as part of a unit test; *Code Coverage* helps to identify the lines that are not, so that new tests can be created.

A8. Design Patterns

Factory

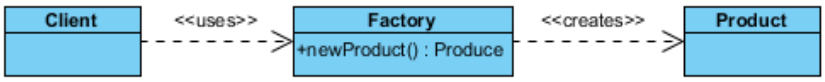

Figure 103 Class diagram showing the factory method pattern

The factory method pattern encapsulates logic that governs the creation of other classes so that the instantiation process may be centralised. In the example above, there may be many product subclasses; with criteria that determines which needs to be used in a particular circumstance. When the *newProduct()* method is invoked by the client, the factory determines, based on the circumstance, which product subclass to instantiate and return.

Proxy

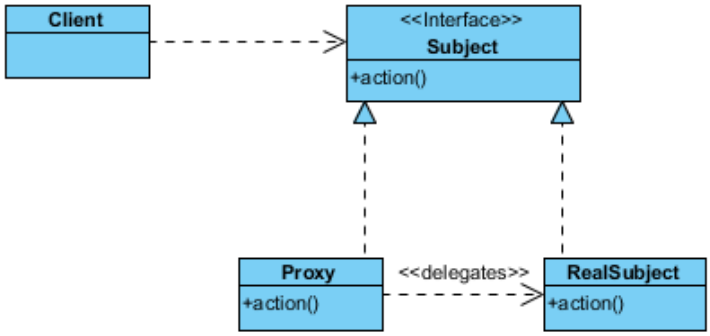

Figure 104 Class diagram showing the proxy pattern

The proxy pattern provides a surrogate for another object to control access to it. The proxy pattern enables access to remote objects through a network connection, facilitates usage of larger objects or other resources that would be inefficient to recreate or duplicate, or acts as an intermediary to incorporate additional functionality such as security.

Decorator

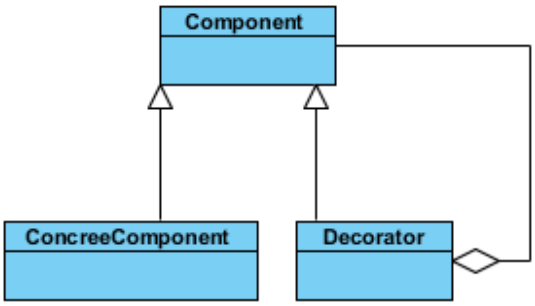

Figure 105 Class diagram showing the decorator pattern

The decorator pattern allows additional functionality or responsibility to be added to an object through a means other than sub classing. This new functionality is added by wrapping the decorator around the original class. Another decorator may be added by wrapping it around the previous decorator, enabling an unlimited number of decorators to be added through recursive wrapping. The decorator allows the behaviour of objects to be altered at runtime, unlike sub classing which alters behaviour at compile time.

Facade

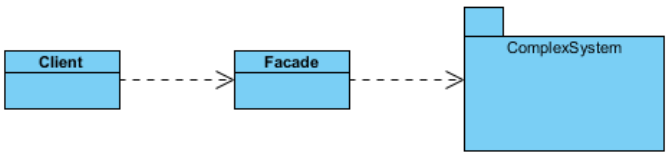

Figure 106 Class diagram showing the facade pattern

The facade pattern is a form of adapter. A facade is used to hide the complexities of a subsystem. A facade provides a single, unified and simplified interface to multiple more complex interfaces within a subsystem. Facades are useful for reducing the complexity of a complex system, such as a class library, keeping dependencies on the inner workings of other systems to a minimum and enabling a badly designed API to be wrapped by a well designed API.

Singleton

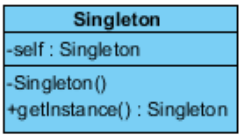

Figure 107 Class diagram showing the singleton pattern

The singleton pattern ensures that a class can only ever have one instance. The class diagram above shows a singleton class which has a private constructor. Classes wishing to use the singleton call the public static method *getInstance()* which, on the first call, instantiates the singleton, stores it in the static *self* property and returns it. On the second and subsequent calls the value of *self* is merely returned. Singletons are useful for thread pools, caches and any other situation in which only one object is needed and a single access point to that object is required.

Adapter

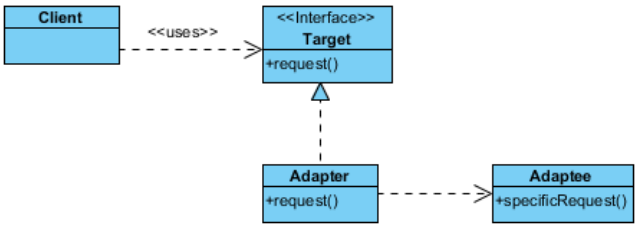

Figure 108 Class diagram showing the adapter pattern

The adapter pattern is used to convert the interface of one class into an interface that the client expects, enabling classes with previously incompatible interfaces to work together. The adapter works by providing a client compatible interface whilst using the original interface of the adaptee internally.

Observer

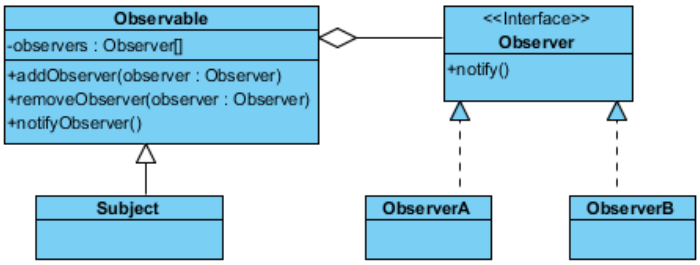

Figure 109 Class diagram showing the observer pattern

The observer pattern allows the definition of a one-to-many relationship between an object and a set of observing objects that need to be notified of a change in the state of the object they are observing. When the state of the observable object is changed in any way, each of the added observers is notified of that change. Although the observable object is aware of the added observers, it is not aware of their inner workings, thus observers and observables are loose coupling and can be reused independently of one another.

Visitor

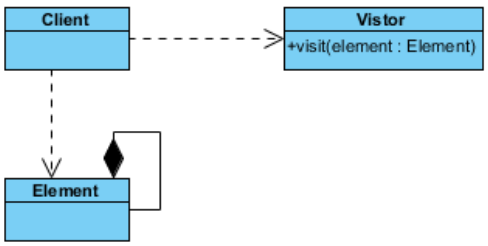

Figure 110 Class diagram showing the visitor pattern

The visitor pattern allows a client to work with all of the elements within a composite. When invoked, a visitor traverses the composition, visiting each and every element within it to retrieve or manipulate its state. This is a simple and elegant solution to perform actions on composites in a centralised way, but the process violates the encapsulation of each visited element.

Chain of Responsibility

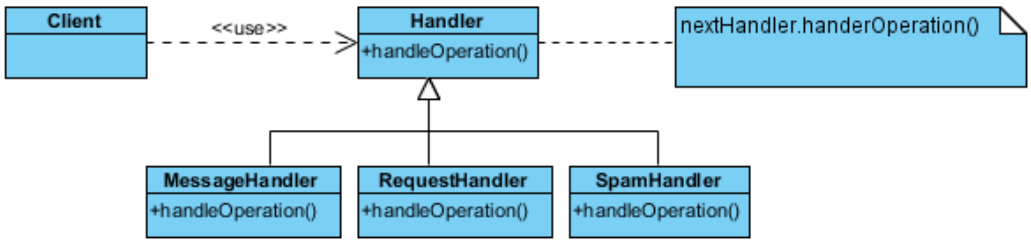

Figure 111 Class diagram showing the chain of responsibility pattern

The chain of responsibility pattern allows a client to pass a request to a single handler, which will in turn pass it along a chain of handlers until the handler capable of handling the request is reached. This pattern simplifies the client class which does not need to be aware of the complexities of the chain, and new handlers for new request types may be added without disrupting the client. The pattern does reduce runtime traceability however, and may causes a problem if a request reaches the end of the chain without being handled.

Builder

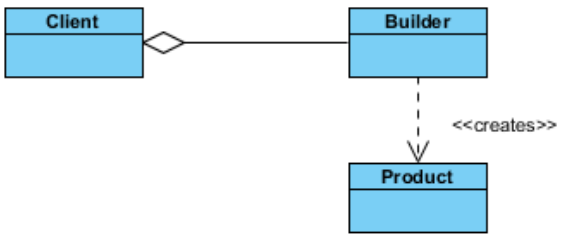

Figure 112 Class diagram showing the builder pattern

The builder pattern allows the logic for the construction of complex objects to be encapsulated. This is similar to the factory pattern, although the builder pattern facilitates the creation of products in multiple steps and processes.

A9. Release Notes

This appendix lists the release notes for each of the development iterations that were completed.

The information was generated automatically by JIRA.

Picon Framework Sprint 1

New Feature

- [PICON-16] - Mark-up inheritance
- [PICON-17] - Repeatable components
- [PICON-18] – Panels

Task

- [PICON-1] - Create the Application Initialiser
- [PICON-6] - Request Processor
- [PICON-7] - Request Resolve
- [PICON-8] - Rendering Process
- [PICON-9] - Page Map
- [PICON-10] - Simple page request
- [PICON-13] - Panels

Sub-task

- [PICON-2] - Require all needed PHP scripts
- [PICON-3] - Config Holder
- [PICON-4] - Resource Instantiation
- [PICON-5] - Injection of Resources

Picon Framework Sprint 2

Bug

- [PICON-46] - XML parser throws exception on HTML entities

Improvement

- [PICON-38] - Path separator should be constant
- [PICON-50] - Add full referencing support to the serialiser
- [PICON-52] - Form model object conversion

New Feature

- [PICON-24] - Form validators
- [PICON-25] - Data source creation from config
- [PICON-26] - DAO super classes
- [PICON-31] - Form field processing
- [PICON-32] - Compound models
- [PICON-33] - Model inheritance
- [PICON-64] - Create data table component and data provider to power it
- [PICON-65] - Implement component behaviours

Story

- [PICON-29] - Stateful pages are to be placed in the page map and stored by ID
- [PICON-30] - Authorisation and Authentication

Task

- [PICON-11] - Stateful page request target
- [PICON-12] - Forms and Form components

Sub-task

- [PICON-35] - Create authentication strategy
- [PICON-36] - Authenticated and Authorised storage

Picon Framework Sprint 3

Bug

- [PICON-60] - Picon Serialiser
- [PICON-77] - HTML entities are rendering as plain text.

Improvement

- [PICON-39] - Header contributor
- [PICON-40] - Support import of external java script and CSS resources
- [PICON-62] - Create a hierarchy of properties which are inherited through the class hierarchy with component followed by application at the top. Values from such propertY files to be accessible via a localizer.

New Feature

- [PICON-19] - Borders
- [PICON-48] - Auto loader cache
- [PICON-61] - Cache Utils
- [PICON-66] - Create Ajax behaviour, Ajax request target, and client side JavaScript to go with it.
- [PICON-67] - Ajax form submission
- [PICON-68] - Ajax buttons
- [PICON-69] - Ajax links

Picon Framework Sprint 4

Improvement

- [PICON-79] - Header contributor for Ajax response
- [PICON-82] - setPage() should add a redirect request target if added on a listener request
- [PICON-83] - Create an actual not found page, not just a placeholder

Task

- [PICON-70] - Model window component including JavaScript
- [PICON-71] - Create the sample application
- [PICON-72] - Server side jQuery integration

Podium CMS Sprint 1

Task

- [PDM-1] - Create the Podium logo
- [PDM-2] - Create the overall user interface graphics. HTML and CSS
- [PDM-3] - Page layout editor, HTML and JavaScript
- [PDM-4] - Arrangement editor, HTML and JavaScript

Podium CMS Sprint 2

New Feature

- [PDM-19] - Create the widget configuration modal window

Task

- [PDM-5] - Widget editor, HTML and JavaScript
- [PDM-6] - Create authentication strategy and login page

- [PDM-7] - Create user management pages (view/edit/delete/create)
- [PDM-8] - Create form manager
- [PDM-9] - Create the text widget
- [PDM-11] - Create the form widget
- [PDM-12] - Create the heading widget

Podium CMS Sprint 3

Task

- [PDM-10] - Create the image widget
- [PDM-13] - Create the content type editor
- [PDM-14] - Page create/edit/view
- [PDM-15] - Posts create/edit/view
- [PDM-16] - Create front end page composite from page/post
- [PDM-17] - Theme manager
- [PDM-18] – Dashboard
- [PDM-20] - Create the navigation widget

Glossary

Abstract Class

In object orientated programming, an abstract class is a class which cannot be instantiated. It will often contain abstract methods which must be implemented by sub classes unless those sub classes, are themselves abstract.

Accessors and Mutators

In object orientated programming, where data is encapsulated inside a class, methods are required to retrieve and modify that data. These methods are known as accessors (retrieve) and mutators (modify).

Ajax

An acronym for Asynchronous JavaScript and XML. Ajax is a set of techniques for allowing a website client side to send or retrieve data from the server side asynchronously.

Annotation

In software, an annotation is a special form of syntax that allows metadata to be added to a language construct. In Java for example, an annotation may be added to a class, a property, a method or a method argument. Annotations are usually detected using reflection.

Application Logic

The enterprise logic of an application which contains technology specific logic, and usually consists of utility services offering functionality for user interaction and presentation adaptation. (Erl, 2005)

Application Programming Interface (API)

An interface used by software components to communicate with other components.

Array

A collection of objects which are stored as a list, each entry of which is stored against a number, generated at runtime.

Associative Array

An array in which the objects are stored, not against a number, but any type of data such as a string.

Bean (See Java Bean)

Black Box Testing

A form of software testing which tests the functionality of the application without knowledge of how the application works internally. Tests are generated using use cases and requirements, they specify the input that is to be used and the output that is expected to be produced.

Bootstrap

A small process of a software application which triggers more complex processes. Typically a bootstrap performs some form of loading or initialisation.

Business Logic

The logic of an application, responsible for implementing business rules and constraints in an application independent manner. (Erl, 2005)

Business Service

A software component which implements the core business types, rules and transformations. (Cheesman & Daniels, 2001)

Business Type

Business types form the business type model. A business type model is a UML class diagram which represents the information that a system requires. This information is broken up into cohesive business types, which are represented as a class with the type stereotype. A business type could for

example be *Product* or *Customer*. Business types consist of attributes and associations with other business types. (Cheesman & Daniels, 2001)

Callback Function

A reference to another piece of code which is passed as an argument to other code. The callback is executed upon the meeting of some criteria within the code it is passed to, rather than being executed immediately. (Zandstra, 2010)

Called Class

A term prevalent in PHP since the addition of late static bindings in version 5.3. A called class is the class which is referenced when invoking a static method, even if the method is inherited from a parent class.

CheckStyle

A tool for automating the process of evaluating code for adherence to specified formatting standards.

Client Classes (see User Classes)

Code Abstraction (See Abstract Class)

Code Smell

A code characteristic that is indicative of a more serious underlying problem

Cohesion

Cohesion, within a class, is the measure of how well each piece of functionality within a class is related.

Colour Code

A hexadecimal representation of colour in the RGB format.

Coupling

The measure of how related or dependent a component of a system is to the other components of the system.

Create, Retrieve, Update, Delete (CRUD)

Creation, retrieval, updating and deletion are the four basic operations of data persistence.

Database Access Object (DAO)

An interface of an object which provides an abstraction for accessing a database, exposing some or all of the CRUD operations without exposing the details of the database.

Database Management System (DBMS)

A collection of programs which facilitate the creation, usage and maintenance of a database.

Dependency Injection

The process of injecting an object into a class, removing the need for the class to create the object itself.

Design by Contract

The process of stipulating code implementations that fulfil a specified set of pre-conditions, post-conditions and invariants. Pre-condition specifies what is required before an operation takes place, for example the supply of an argument to a method must be of a particular type. A post-condition is a condition that must be met after an operation has been executed. An invariant is a condition which must always be met in order for operations to be available. (Fowler, 2009)

Design by Primitives

The process of basing the functionality of a class on a small set of primitive operations which are left open for implementation by sub classes. (Riehle, 2000)

Doctrine

A PHP library focused on persistence functionality, offering database abstraction and object relational mapping.

Document Object Model (DOM)

A convention used for representing HTML or XHTML as objects. The objects are organised into a tree structure and provides methods for data access and manipulation.

Document Type Definition (DTD)

Similar to an XML schema, a document type definition declares the type of mark-up that is being used by a document and what elements and references the document may contain.

Domain Object

An object which stores data for representing a particular domain. The object should be autonomous and maintain any business logic required to enforce its validity. It will also frequently offer functionality to compare itself with other object instances and to copy itself.

DOMDocument

A core PHP class which is capable of processing and representing HTML and XML using the document object model.

Dynamic Object

An object instance which exists only at runtime and does not have a class definition.

Enterprise Java Bean (EJB)

A server side component which encapsulates the business logic of an application. EJBs exist within a container and are instantiated automatically by that container, based on a deployment descriptor.

Eval Function

A PHP function which evaluates and executes a string of code.

Extension Classes (Framework)

A form of user class, but with a specific inheritance relationship to the single framework class of which it is a sub class.

Framework

A model of a particular domain, which can be a technical domain or an application domain. The framework provides reusable design and implementation to clients (users).

Front Controller

A controller which acts as a single, centralised intercept point for all HTTP requests received by a web application.

Grey Box Testing

A hybrid testing approach, combining black and white box testing. Grey box testing is conducted in a black box manner, but with knowledge of the inner workings of the components being tested. Consequently, the tests are written and conducted to reveal any defects in the inner workings of the application caused by inappropriate or improper usage.

Hard Coding

The process of embedding configuration or input data within the source code of an application, rather than implementing code to retrieve the data from an external source.

Htaccess File

A configuration file supported by the Apache Web Server which allows directory specific configuration to be applied, such as: authorisation, rewriting, directory list permissions and cache control.

Information Architecture

The process of categorising and organising data for a website or other software application into logical groups to improve usability.

Injection (See Dependency Injection)

Interactive Development Environment (IDE)

A comprehensive application that allows programmers to develop and debug software in a variety of languages such as Java, C# or PHP.

Integration Testing

The process where the individual components of a system are brought together and tested as a group. The purpose of integration testing is to ensure that the components of system interact correctly.

Java Bean

A reusable Java component which follows a convention, stipulating that it is serialisable, has a default constructor, and implements getter and setter methods for the data it encapsulates.

Java Database Connectivity (JDBC)

An API for Java which provides an abstraction for interacting with relational databases.

JavaScript Object Notation (JSON)

A text based format, based on JavaScript, for representing simple data and associative arrays.

Locale

A set of parameters which define data such as time zone and language for a particular location.

Metadata

Data about other data. Metadata describes the structure or content of data.

Microformat

A semantic technique which uses XHTML tags to convey metadata. A Microformat allows software to automatically extract data from a website and use the metadata to identify its meaning.

Model View Controller (MVC)

An architectural design pattern which promotes the separation of concerns for presentation, event handling and data persistence, see chapter 4.

MySQL

An open source relational database management system.

Object Relational Mapping (ORM)

The process of converting data from a type defined in one system to a type defined by another. A common example is the process of mapping a database result set onto a domain object.

Orchestration Logic

The logic of an application which is responsible for invoking operations in the correct order and at the correct time to perform a function.

Party Pattern

A design pattern which describes the representation of information about people and organisations in abstracted manner, enabling general information common to all parties to be inherited rather than duplicated.

Plain Old Java Object (POJO)

A Java object which does not follow any convention, framework or model. It is merely a simple object which encapsulates some data and offers methods to access and modify it.

Plumbing Code

A slang term, also known as boilerplate code, referring to verbose code which does very little, although has to be called or included in many places as there is no alternative implementation.

Plug-in

A software component that can be incorporated into a larger component to supply it with additional functionality.

PMD

A tool which scans and evaluates source code for potential bugs, code smells, dead code, and over complicated or duplicated code

Portal

Also known as a web portal, a portal is a website which brings together information and functionality from many sources in a unified, consistent and seamless manner.

Query String

The name value pairs which appear after a question mark in a URL, allowing specification of additional parameters for an HTTP request.

Rapid Application Development (RAD)

A software development methodology that includes very little planning in favour of the iterative development of prototypes.

Refactoring

The process of reorganising source code to alter its structure but not its behaviour. Refactoring aims to improve code cohesion and minimise duplication, through the extraction of methods or the pushing up of methods or properties into parent classes.

Scriptlet

The name given to a small fragment of code embedded into another document for runtime execution.

Search Engine Optimisation (SEO)

The process of improving the semantics, metadata and keyword prevalence on a website to improve its rank in search engine results.

Selenium

A set of testing tools that enable browser automation. Selenium enables the creation of automated system tests which operate using a web drive to take control of a web browser, performing clicks, executing DOM events and evaluating response HTML.

Separation of Concerns

The process of breaking up an application into distinctive, cohesive parts with very little or no functional overlap.

Session

A semi-permanent mechanism for storing data concerning the current interaction between two systems. In web applications, a session is a mechanism for maintaining state across multiple stateless requests. The client is issued with a session ID by the server through an HTTP response header; the client will include the session ID in all subsequent HTTP requests, enabling the server to recognise it and retrieve any data stored against the ID.

SimpleXML

A core PHP library which enables the parsing, manipulation and output of XML using dynamic domain objects.

Sprintf Function

A function built into PHP for formatting a string containing one or more directives which are to be replaced with user provided parameters.

Stack Trace

A report of the active subroutines of a program at a particular point in its execution.

Stringly Typed

A slang term for a program which relies heavily on strings when programmer and refactoring friendly alternates are available.

System Service

The software component which represents the processes of a system. A system service uses business services to perform its processes. System services are a facade for the business services. (Cheesman & Daniels, 2001)

System Testing

A set of tests conducted on a completed and integrated system that determines whether it fulfil its requirements.

Tagging Interface

Also known as a marking interface, a tagging interface does not contain any methods but acts as a method to mark or tag a class in a manner that can be detected by other classes, so that they may perform additional or special operations on the tagged class.

This Keyword

An object orientated programming keyword used within the scope of a method, in order to refer to the instance of the class in which the method is defined.

Three-Tier Architecture

An architectural pattern in which the presentation logic, functional (business and system) logic and persistence logic are broken up into separate, distinct and loosely coupled layers.

Twig

A templating engine for PHP that reduces the amount of code required to process and prepare data for output.

Uniform Resource Identifier (URI)

A string that is used to identify a named resource enabling it to be interacted with over a network, typically the Internet.

Uniform Resource Locator (URL)

A string of characters that references an Internet resource.

Unit Testing

The tests that are conducted on individual pieces of source code, with suitable test data and parameters to ensure that they fulfil their purpose and that the internal logic is correct and robust.

User Acceptance Testing

A form of system testing completed specifically by users of a system rather than testers. User acceptance testing is conducted to ensure that the systems users are satisfied the system meets its specification. User acceptance testing is a form of black box testing.

User Classes (Framework)

A class external to a framework, written by a developer making use of the framework. The class is related to one or more framework classes with a use relationship.

User Extension Class (See extension class)

Web 3.0

The general name given to the semantic web which promotes a common semantic format for data on the web.

White Box Testing

A form of testing which tests the inner workings and internal logic of a system. White box testing requires the tester to have knowledge of the inner workings of a system, and have programming skills; both of which are used to design and implement test cases.

Widget

An element of a graphical user interface which, when displayed as part of a composition of widgets, displays and allow manipulation of the data and functionality of a system.

WYSIWYG Editor

What you see is what you get. The term defines any form of editing interface in which the text and graphics being edited are present in a form that is as visually close to the final output as possible.

XML

Extensible mark-up language. XML is a format and structure that enables the storage and transport of any data, in a format that is both human and machine readable.

XMLParser

A set of core PHP functions for processing and parsing XML. It can process documents automatically into dynamic objects; but also allows a user to attach callback functions to the parsing process, which are called on the discovery of each element, passing full details of the element as arguments, enabling the mapped objects to be customised.

YAML

A human readable standard for language independent formatting of serialised data.

References

Allen, R. & Lo, N., 2007. *Zend Framework in Action*. Manning.

Apache Software Foundation, 2012a. *Apache JMeter - Apache JMeter*. [Online] Available at: http://jmeter.apache.org/ [Accessed 23 March 2012].

Apache Software Foundation, 2012. *Apache Subversion Features*. [Online] Available at: http://subversion.apache.org/features.html [Accessed 23 March 2012].

Atlassian Plc Ltd, 2011a. *Bugs, Issues and Project Tracking for Software Development*. [Online] Available at: http://www.atlassian.com/software/jira/ [Accessed 26 March 2011].

Automattic, 2012. [Online] Available at: http://wordpress.org/ [Accessed 3 January 2012].

Basham, B., Sirra, K. & Bates, B., 2008. *Head First Servlets and JSP*. 2nd ed. O'Reilly.

Beck, K. et al., 2001. *Manifesto for Agile Software Development*. [Online] Available at: http://agilemanifesto.org/ [Accessed 27 February 2012].

Bergmann, S., 2012. *Chapter 1. Automating Tests*. [Online] Available at: http://www.phpunit.de/manual/3.6/en/automating-tests.html [Accessed 23 March 2012].

BigCommerce Pty, 2012. *Shopping Cart Software | Ecommerce Software - Interspire*. [Online] Available at: http://www.interspire.com/ [Accessed 2 January 2012].

Cake Software Foundation, Inc., 2011. *The Manual : 1.3 Collection*. [Online] Available at: http://book.cakephp.org/ [Accessed 4 December 2011].

Chan, K., Omokore, J. & Miller, R.K., 2009. *Practical Cake PHP Projects*.

Cheesman, J. & Daniels, J., 2001. *UML Components: A Simple Process for Specifying Component-Based Software*. 1st ed. Addison-Wesley.

Cooper, I., 2008. *The Fat Controller*. [Online] Available at: http://codebetter.com/iancooper/2008/12/03/the-fat-controller/ [Accessed 26 January 2012].

Dashorst, M. & Hillenius, E., 2009. *Wicket in Action*. Manning.

Davis, P., 2012. *Annotations Gotchas and Best Practices*. [Online] Available at: http://willcode4beer.com/design.jsp?set=annotations_gotchas_best_practices [Accessed 15 February 2012].

Dilley, B., 2011. *addendum - Annotations support for PHP - Google Project Hosting*. [Online] Available at: http://code.google.com/p/addendum/ [Accessed 23 March 2012].

DocForge, 2011. *Web Application Framework*. [Online] Available at: http://docforge.com/wiki/Web_application_framework [Accessed 14 November 2011].

Erl, T., 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall.

Fowler, M., 2009. *UML Distilled*. 3rd ed. Addison Wesley.

Golding, D., 2008. *Beginning Cake PHP*. Apress.

Gradecki, J.D. & Lesiecki, N., 2003. *Mastering AspectJ*. Wiley.

Johnston, M., 2010. *What is a CMS?* [Online] Available at: http://www.cmscritic.com/what-is-a-cms/ [Accessed 9 October 2011].

Jonnay, 2006. *Aspect Oriented Programming in PHP as a contrast to other languages*. [Online] Available at: http://blog.jonnay.net/archives/637-Aspect-Oriented-Programming-in-PHP-as-a-contrast-to-other-languages.html [Accessed 13 February 2012].

Maciaszek, L.A., 2001. *Requirements Analysis and System Design*. Addison Wesley.

Mall, R., 2004. *Fundamentals of Software Engineering*. Prentice Hall.

Marriott, J. & Waring, E., 2011. *The Official Joomla!™ Book*. Addison Wesley.

Open Source Matters Inc, 2012. [Online] Available at: http://www.joomla.org/ [Accessed 3 January 2012].

Reenskaug, T., 1979. *Models - Views - Controllers*. [Online] Xerox PARC Available at: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf [Accessed 17 December 2011].

Reenskaug, T., 2011. *MVC XEROX PARC 1978-79*. [Online] Available at: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html [Accessed 17 December 2011].

Riehle, D., 2000. *Framework Design: A Role Modeling Approach, Ph.D. Thesis, No. 13509*. [Online] Doctor of Technical Sciences, University of Hamburg. Available at: http://www.riehle.org/computer-science/research/dissertation/diss-a4.pdf [Accessed 9 October 2011].

Sametinger, J., 1997. *Software Engineering with Reusable Components*. Springer.

Sensio Labs, 2011. *Book - Symfony*. [Online] Available at: http://symfony.com/doc/current/book [Accessed 4 December 2011].

Shore, J. & Warden, S., 2007. *The art of agile development*. O'Reilly Media.

Stern, H., Damstra, D. & Williams, B., 2010. *Professional WordPress Design and Development*. Wiley Publishing, Inc.

Szyperski, C., 1998. *Component Software Beyond Object-Orientated Programming*. Addison-Wesley.

The jQuery Project, 2010. *jQuery: The Write Less, Do More, JavaScript Library*. [Online] Available at: http://jquery.com/ [Accessed 23 March 2012].

The PHP Group, 2012. *PHP: Hypertext Preprocessor*. [Online] Available at: http://php.net/ [Accessed 15 February 2012].

Wallace, B., 2010. *Existential Programming*. [Online] Available at: http://existentialprogramming.blogspot.com/2010/05/hole-for-every-component-and-every.html [Accessed 1 January 2012].

Walls, C., 2011. *Spring in Action*. 3rd ed. Manning.

Wel, A.v. et al., 2011. *Preferred php framework by wicketeers*. [Online] Available at: http://apache-wicket.1842946.n4.nabble.com/preferred-php-framework-by-wicketeers-td3341212.html [Accessed 15 February 2012].

Wikipedia, 2012. *Agile software development - Wikipedia, the free encyclopedia*. [Online] Available at: http://en.wikipedia.org/wiki/Agile_software_development [Accessed 27 February 2012].

Yinka, A. et al., 2009. *Wicket in Php*. [Online] Available at: http://www.mail-archive.com/users@wicket.apache.org/msg36920.html [Accessed 15 February 2012].

Zandstra, M., 2010. *PHP Objects, Patterns, and Practice*. 3rd ed. Apress.

Zaninotto, F. & Potencie, F., 2010. *A Gentle Introduction to Symfony 1.4*. Sensio Labs.

Bibliography

Anon., 2010. *RFC Annotations*. [Online] Available at: https://wiki.php.net/rfc/annotations [Accessed 16 February 2012].

Anon., 2012. *phpcheckstyle - Coding convention style checker for PHP - Google Project Hosting*. [Online] Available at: http://code.google.com/p/phpcheckstyle/ [Accessed 26 March 2012].

Apache Software Foundation, 2011. *Apache Wicket Index*. [Online] Available at: https://cwiki.apache.org/WICKET/ [Accessed 3 November 2011].

Apache Software Foundation, 2011. *Overview (Wicket Parent 1.5-SNAPSHOT API)*. [Online] Available at: http://wicket.apache.org/apidocs/1.5/ [Accessed 3 November 2011].

Connolly, T.M. & Begg, C.E., 2005. *Database systems: a practical approach to design, implementation, and management*. Addison-Wesley.

Freeman, E., Freeman, E., Sierra, K. & Bates, B., 2004. *Head First Design Patterns*. 1st ed. O'Reilly.

Kobryn, C., 2001. *Proceedings of OMG Workshops*. [Online] Available at: http://www.omg.org/news/meetings/workshops/presentations/embedded-rt2002/02-1_Kobryn_UML_Tutorial.pdf [Accessed 11 February 2012].

Lo, N., 2006. *Zend Framework - Nested Views and Layouts*. [Online] Available at: http://zend-framework-community.634137.n4.nabble.com/Nested-Views-and-Layouts-td636945.html [Accessed 30 October 2011].

New Digital Group, 2011. *PHP Template Engine | Smarty*. [Online] Available at: http://www.smarty.net/ [Accessed 29 October 2011].

Pichler, M., 2012. *PHPMD - PHP Mess Detector*. [Online] Available at: http://phpmd.org/ [Accessed 26 March 2012].

SpringSource, 2011. *Overview*. [Online] Available at:

http://static.springsource.org/spring/docs/3.0.x/javadoc-api/ [Accessed 4 November 2011].

Ullman, L., 2008. *PHP6 and MySQL5*. Peachpit Press.

Zend Technologies Ltd, 2011. *Zend Framework: API Documentation*. [Online] Available at:

http://framework.zend.com/apidoc/1.11/ [Accessed 30 October 2011].