Study of Patterns in an Agile Web Development Environment

Radek Štěpán

MSc in Advanced Computer Systems:
Internet Systems 2010

Study of Patterns in an Agile Web Development Environment

submitted by Radek Štěpán

for the degree of MSc

of the University of Bath

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in the Department of Computer Science. No portion of the work in this thesis has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Acknowledgments

I am grateful to my supervisor, Dr. Claire P. Willis for her perfect guidance, to my adviser Theo for his critical eye and interest. Thank you both for keeping me down to Earth and not letting my “abstractions” run wild.

I would also like to acknowledge and thank my typist, Mr. LyX, particularly for his figures positioning.

Abstract

In the realm of web application architecture, the Model-View-Controller pattern is the predominant model for dividing labor between the system's business, application and presentation needs. Unfortunately, developers cannot precisely agree as to where to draw the lines separating these three concerns.

Our project develops a reusable, generally applicable, framework consisting of architectural and design patterns that clearly demonstrates how to solve the split in application concerns. Alternative solutions and approaches are evaluated according to their merit in the agile web development process.

An example is given that clearly illustrates the findings of the dissertation. The work is concluded with a suggestion for a deeper study and categorization of pattern variants, their combinations and qualitative effects on the system architecture.

Contents

List of Figures

Biases Declaration

The author is a junior web developer with his own PHP MVC/MVP Framework called Fari[1]. His interest wanes when a concept cannot be grasped simply and therefore loves the agile development approach, and lean approach to life in general. Disheartened with his own coding quality in a company setting he has decided to get a firmer grounding by studying a postgraduate degree in Internet Systems and tackling a project under Dr. Claire Willis.

[1] `http://radekstepan.com/fari-framework/`

Part I

Introduction

Chapter 1

Project Background

The aim of this dissertation is to research methods by which web developers can better realize the architecture of a web application to meet specific business needs. In particular the focus is on clearly determining which parts of the application correspond to which concerns[1]. This will be achieved through the following objectives, in order:

1. *Identify* the concerns a web application architecture has to handle and the different forces put on it.

2. *Identify* approaches used to describe web application architectures.

3. *Evaluate* critically the patterns, models and approaches relevant to a development of a web system that adhere to sound software engineering principles in the web domain.

4. *Explore* the merit of the different approaches identified in 3. and their synergic collaboration and effect.

5. *Formulate* architecture recommendations for industry web developers and web engineering academics alike.

Web applications, coming from their humble beginnings in the mid-90's, have seen a spectacular rise to ubiquity in our everyday lives. Wikipedia statistics show that Gmail, a web based email client, is used by about 176 million users monthly, while Facebook, a social networking site, by about 400 million. Other web applications help with daily project management, for example, Basecamp, tasks (Remember the Milk), blogging (Wordpress), listening to music (Last.fm), solving programming problems

[1] A concern may be defined as "a matter of consideration in a software system" [Sutton and Rouvellou, 2002]. We discuss concerns in detail in Chapter 4 on page 45.

(Stackoverflow) or shopping (eBay). Academics greeted the first web based systems with a call for a new discipline of "web engineering" dismissing traditional software engineering discipline as ill-suited for the web [Lang and Fitzgerald, 2007].

As web developers in the industry were getting accustomed to the newly gained abilities of server-side languages at the turn of the century, authors such as Gellersen and Gaedke [1999] and Taylor et al. [2002] were reporting that the development practice is largely unsystematic and unreliable. Boosted by such claims, the emerging field of web engineering has started to gain dominance in the academic world. As such its aims are "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of web-based applications" [Kappel et al., 2004].

In comparison with traditional desktop applications, web applications have to deal with the processing of the request and response cycle of each client-server request. Furthermore, this model is muddled with asynchronous requests to the server that fetch and update content as needed and client-side storage. Consequently, web applications respond to requests in various formats such as HTML, XML or JSON and may connect to other servers through Application Programming Interfaces (APIs) for information. Such applications need to be systematically split and ideally accomplish a separation of concerns.

The concerns that one is separating pertain to the different responsibilities that the code corresponds to. For example, there is the presentation concern that deals with the presentation of application data to the user or the business concern that contains the logic of the system, often connecting to a persistence concern - the database. An appropriate separation of these responsibilities leads to a more systematic and reliable software that is easier to maintain an update, an important requirement in web applications.

It has been suggested that reusable libraries of code, frameworks, could represent a solution supporting a systematic software development. As such, frameworks capture knowledge of a particular domain and represent a skeleton application, ready to be customized. They provide an opportunistic design reuse which in turn results in higher-quality, structured applications as a side effect.

The academic web engineering field has come up with their own frameworks such as OOHDM, WebML, WARP or WebActions. To understand whether such approaches are being used in the industry, a study was conceived by Taylor et al. [2002] testing real world practice in web development across 25 UK companies. The result was that

none of the IT practitioners interviewed saw academic literature as a useful source of website development guidance [*ibid.*]. Jeary et al., in their 2009 analysis of 23 projects, found only one case where a developer continued to use an academic web development method throughout the life-cycle [Jeary et al., 2009]. More generally Lang and Fitzgerald [2005] have concluded that there is a communication gap between academia and (web development) industry.

Meanwhile, two landmark books, by Fowler [2002] and Alur et al. [2001], were published applying formal software engineering, patterns based, principles to the web domain. The industry has responded to these books with an array of, often open-source, frameworks such as Zend, CodeIgniter, Ruby on Rails, Struts or Django and uses these extensively. To put it simply, industry has managed to produce successful web-based systems without the recourse to academic intervention [Barry and Lang, 2003; Lang and Fitzgerald, 2007]. One might therefore suggest that there is no need for the use of more formal, academic ideas in web development. However, this is a short-sighted view. As domains become more mature they tend to increase in complexity, and control of complexity becomes a major issue. This has already become obvious in general software development. However, as we will see later, the applications we describe are often not designed to be complex and focus more on business agility and future updateability of the system.

1.1 Problem

The web software development field is at a stage where we see an array of approaches, models and frameworks each shouting for our attention. On one hand, the Model-View-Controller (MVC) [Reenskaug, 1979*b*,*a*] pattern is the main architectural approach used in a vast array of open source web frameworks with each product stating that "their application of the pattern is the best one". However, a discussion of the alternatives and reasoning behind a particular approach has not been published. Nor do we often see a discussion addressing how to solve a problem that does not neatly fit into one "concern", one layer. Furthermore, we have identified a number of approaches that break solid principles of object-oriented development or conversely result in concerns being split among multiple components. For example, the Data, Context and Interaction (DCI) paradigm [Reenskaug and Coplien, 2009] breaks a single responsibility principle with its mixing of behaviors and presentation while Dispatcher View MVC has business behavior split among models and Business Helper objects.

Academically, web engineering, in the examples of Whitehead [2002] and Deshpande [2004], is poised to be an approach to web development for the industry. However, this

idea is worrying as Jeary et al. [2009] warns us that there has been little empirical work to understand the unique requirements of web development methods in the industry that is reflected in CS graduates being at loss to understand why much of the material they have diligently studied seems to be irrelevant in the "real world" [Lang and Fitzgerald, 2007]. The hypermedia - web engineering approach is centered around a navigation layer concern, what merit does this idea hold?

It is not clear how requirements of a business problem translate to code in the different tiers of a web application. The solution needs to be simple, after all, we do not discuss safety-critical software or enterprise level development, yet we need to understand *why* we are structuring our code in a certain way and what *alternatives* there are. So often the answer to this problem has been to "just use MVC", an answer that is unsatisfactory at best, and wholly naive and untenable at worst!

1.2 Thesis Guide

The problem we will research is, by necessity, very broad. One risks getting side-tracked mid-way through the discussion or forgetting what idea has been proposed a couple dozen pages ago. At the same time, we need to touch on multiple topics to get ideas from. Therefore, the thesis is divided into parts which address the key areas required by the objectives (Part I on page 14). Each part is then subdivided into individual chapters and sections.

Part II, Web Application Architecture

Addressing objectives #1 & #2.

Chapter 2 on page 23 **Web Applications Background** introduces the field of web development and specifies what web applications are.

Chapter 3 on page 28 **Layers** discusses a *4 layer model*, a dominant approach to splitting the different concerns of an application into its distinctive parts. It also comments on the *navigation layer* used in hypermedia - web engineering frameworks, offering alternative approaches. This chapter would not be complete without a discussion of state management and subsequently the *state layer*.

Chapter 4 on page 45 **Concerns** describes what happens to software quality as concerns in different layers are mixed and tangled.

Chapter 5 on page 50 **Web Application Architecture** discusses advantages of writing reusable code and what limits its use among developers. Then, the discussion will introduce to us approaches used by developers when architecting web software. We will start with good principles of object-oriented development and discuss, among others, patterns and frameworks that work as application skeletons ready to be reused.

Part III, Patterns, Methods & Architectures

Addressing objective #3.

Chapter 7 on page 68 **Patterns** introduces the way we will discuss these solutions to commonly recurring architecture problems. We will position patterns as documentation mediums.

Chapter 8 on page 71 **Model-View-Controller** will discuss a pattern that is found in most web application frameworks. The chapter is lengthy as many other patterns and architectures reference and improve upon this pattern, thus we need to understand it clearly.

Chapter 9 on page 95 **Data Transfer Object** describes an approach that illustrates how data between business and presentation layers can be transferred.

Chapter 10 on page 99 **Model-View-Presenter** is a chapter describing an improved MVC pattern that focuses on encapsulating more complex presentation layer logic in a controller component.

Chapter 11 on page 110 **Model Adapter** supports the MVP pattern by presenting a couple of "hard" scenarios in web architecture.

Chapter 12 on page 115 **Django Model Template View** focuses on an alternative approach to working with presentation logic - through generic view components. We also discover the advantages of validating data in a business layer.

Chapter 13 on page 119 **Naked Objects Architecture** further forces us to understand good principles behind Domain-Driven Design of which Naked Objects is an example. This chapter also takes the extreme point in making it easier to generate a presentation layer.

Chapter 14 on page 123 **Data, Context and Interaction** describes the latest offering from Trygve Reenskaug, author of MVC. It supports a view that

business objects need to be set in a certain context by an application layer logic.

Chapter 15 on page 128 **Other Reviewed Approaches** discusses patterns that were evaluated but for one reason or another are not particularly relevant to our argument.

Part IV, From Requirements to Code

Addressing objective #4.

Chapter 17 on page 141 **Agile Methodologies** introduces a software development methodology that will make it easier to envisage a web architecture in our next example.

Chapter 18 on page 143 **Blog Example** applies patterns discussed in Part III on page 66 through a lean development process.

Part V, Project Conclusions

Addressing objective #5.

Chapter 20 on page 167 **Conclusion** brings together what we have learned and suggests future directions for research.

Part II

Web Application Architecture

In this part of the dissertation we are going to introduce the field of web development and specify what web applications are. Then, in the later sections, we depict the way web development is done now. We present this introduction to the field so that the reader can appreciate the limitations and specifics of the environment. We can then better discuss the pressures a web application architecture needs to handle in subsequent parts of the dissertation.

Chapter 2

Web Applications Background

This chapter answers how we define web applications, the subject of our study. Then we present the specifics of the client - server environment which web applications are part of.

2.1 Classification & Definition

The purpose of our study is web applications. The name is obvious and everyone seems to intuitively know what "web applications" are. But do they?

Christodoulou et al. [2000] describes three distinct types of applications that use web for their delivery, they are:

1. *Web hypermedia application*s - these applications are used to publish information on the web with users navigating through nodes, links and anchors linking to the content.

2. *Web software application*s - are conventional applications that use web for their execution.

3. *Web application*s - are applications combining the previous two concepts.

On the other hand, Conallen [2000] differentiates web applications based on where the data served are put together:

1. *Thin web client*s - where all the processing happens on the server-side.

2. *Thick web client* - the client requesting data processes them through enhancements such as JavaScript.

3. *Web delivery* - are fully distributed systems where client and server maintain a state through protocols such as IIOP, DCOM and RMI.

However, both of these definitions do not give us, in 2010, the correct and unambiguous answer. For example, thin clients can be mistaken for the term referring to computers running on a networked OS and JavaScript is no longer a language used just for enhancements. A web application can also offer its content through various channels (APIs) and in various formats, such as JSON, XML or HTML.

Hence, can we get a clear answer from the environment web applications run in, client-server? This model is usually split into a commonly defined three-tiered approach, consisting of:

1. *Client front-end* - displaying content served by the server and making requests to it.

2. *Middle layer* - this is the domain of content processing and generation through application server technologies, platforms.

3. *Database layer* - a database server managing data.

But again, what about exhaustive processing on the client front-end? Do we have to split the client front-end, the browser, into a front and a back side? Also, data on the server can be managed by a flat-file system such as SQLite, XML or even plain text files.

Thus, for the purposes of this dissertation, web applications are defined as:

> *Online client-server applications executed on and served by an HTTP(S) web server technology accessed through URL schemes.*

This leaves the definition open for and allowing:

1. Applications that run primarily on a client-side technology such as JavaScript, storing client data in cookies[1] or in an HTML5 JavaScript database class (see Section A.1 on page 175).

2. Applications that use RFC3875 Common Gateway Interface (CGI) programs [Robinson and Coar, 2004] launched by web servers such as Apache, IIS, nginx or lighttpd to execute technologies such as PHP, ASP, .NET, Python, Ruby on the server-side.

3. Everything in between, using various means of storage and either browser or server processing[2].

[1] An example of which could be the various "to-do list" applications.

[2] A special case would be offline web applications that run from local source files only. We are interested in applications that require a web server for their execution.

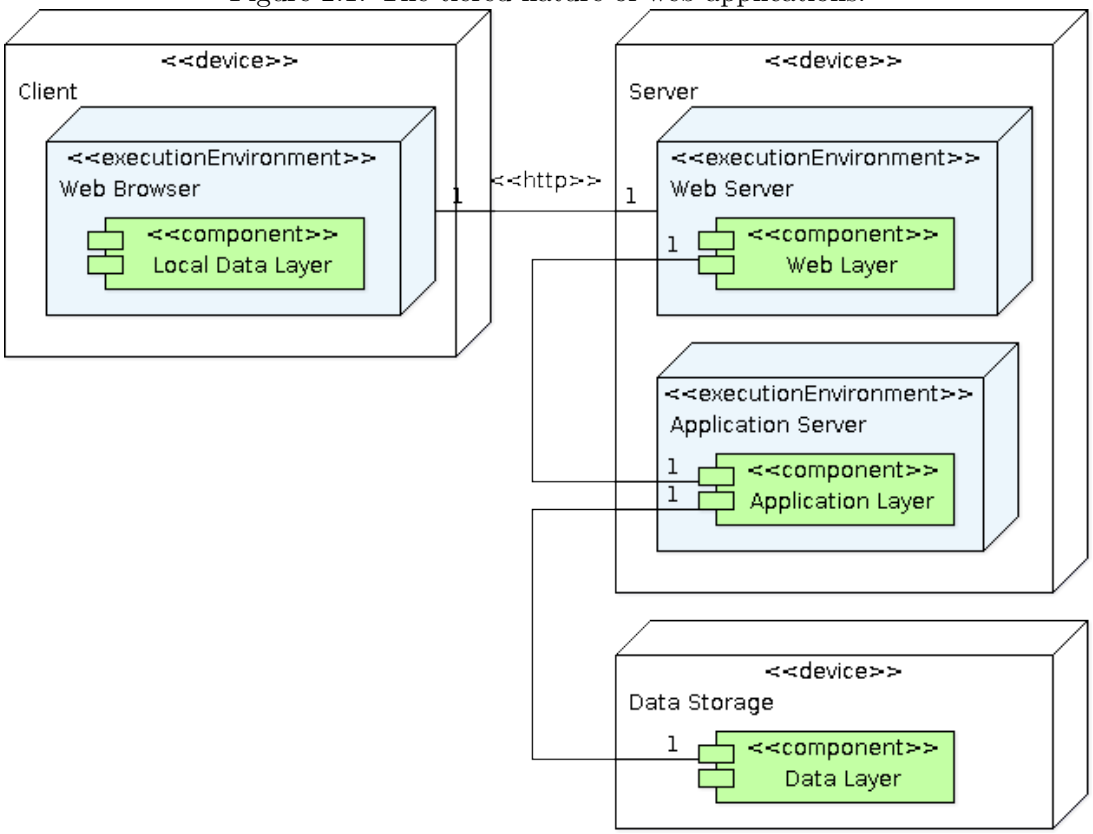
In the following section we are going to look at the environment web applications run in to better understand their requirements and specifics.

2.2 Web Application Environment & Specifics

Having defined and covered the technologies, applications and "sides" involved, let us turn to the domain of web applications.

The client-sever nature has been described in the previous section and can be seen in Figure 2.1. Of course, even this simple client-server model can be extended by using external servers. For example, an application running on the server can fetch data from yet another server, a service, using APIs. But from the view of the client, a request is made to the server via a URL scheme and it is up to the server to deal with it.

Figure 2.1: The tiered nature of web applications.



2.2.1 Request & Response Cycle

Originally, web servers were only handling static HTML content[3], but now only 29% of websites do so. The majority use CGI extensions that we use to build dynamic

[3]The name HyperText Transfer Protocol tells us that the protocol is for the transfer of hypertext. However a web application extends this original purpose by running in a dynamic environment.

applications for the web with. This approach[4] means that web applications have their specifics. Take for example the PHP technology which leads the way with 29% of the market[5]. PHP[6] stands for PHP: Hypertext Preprocessor and was meant as a way for code scripts mixed throughout HTML to be executed by the server. We will later see how mixing scripts with HTML is not good programming practice, but this approach is simple and straightforward. This simplicity of server-side languages means that modern web applications need to:

1. Handle HTTP request methods, most commonly GET and POST, and their processing. Examples: form input, file uploads, different user agents, technologies used such as Ajax, different server configurations.

2. Handle different HTTP server responses. Examples: status codes, content types (HTML, XML, JSON), cookies, caching, character set encoding.

3. The client should ideally remember its state as all rendered information is lost during a standard, synchronous, request.

Therefore, the request and response model represents a web layer in our application and this channel also means that the data transfer time to and from a server is greater than on a desktop.

The ease of distribution of new versions of applications means that software can be released earlier and more often and easily deployed to a central server. The maintenance cycle changes to a matter of days or even hours as researched by Offutt [2002] (for further discussion see Section A.2 on page 175). On the other hand, as Offutt [2002] continues, the application's users rely on the server to keep the software running; any downtime, no matter how short, can be harmful.

This changes the typical scope of projects, making them gain more from the design of an architecture rather than from code reuse, such is the specificity of web development environment.

2.2.2 JavaScript and Ajax

We have already seen that web applications can process various amounts of data on the client and on the server (or both). Sometimes we cannot choose and an incompatible environment forces us to fallback onto a method of processing supported by the client, the web browser. Take the example of a contact form in a generic Content Management

[4]Native to the whole way Internet works.

[5]In its purest form, the various PHP Frameworks were not calculated in, `http://trends.builtwith.com/framework` from August 22, 2010.

[6]PHP originally stood for personal home page.

System (CMS) distribution. We might use JavaScript to check the validity of input on the client and launch an Ajax request so that the response from the client is faster, but the browser might have this technology switched off. Furthermore, relying on security checks only on the client is dangerous. Therefore, a form needs to be sent even with JavaScript turned off and the web application has to accommodate that.

2.3 Discussion of Web Application Background

JavaScript, Ajax and caching break the traditional approach of generating a new static HTML for each request to the server made. For instance, the client is empowered to paginate or sort data internally[7] and an application running in an offline mode can update its state once back online.

The more frequent usage of external services through APIs and heavier client processing that is afforded by faster JavaScript engines means that web applications will only get more complex with parts of code scattered across different tiers or layers. And we have warned in the introduction that dismissing this complexity can be shortsighted (Chapter 1 on page 15). This will influence the way we look at web application development in the next chapter on layer. What is also apparent is that a page no longer neatly is the "unit" on the web. Asynchronous requests presented in this chapter show that a request to a server can serve as a communication medium akin to passing of messages that can be used to form a whole page that we see in the browser. We now access resources and not whole pages. This will have an effect on how we view the layers we discuss in the next chapter.

[7] Using a JavaScript library such as jQuery, Prototype in a browser.

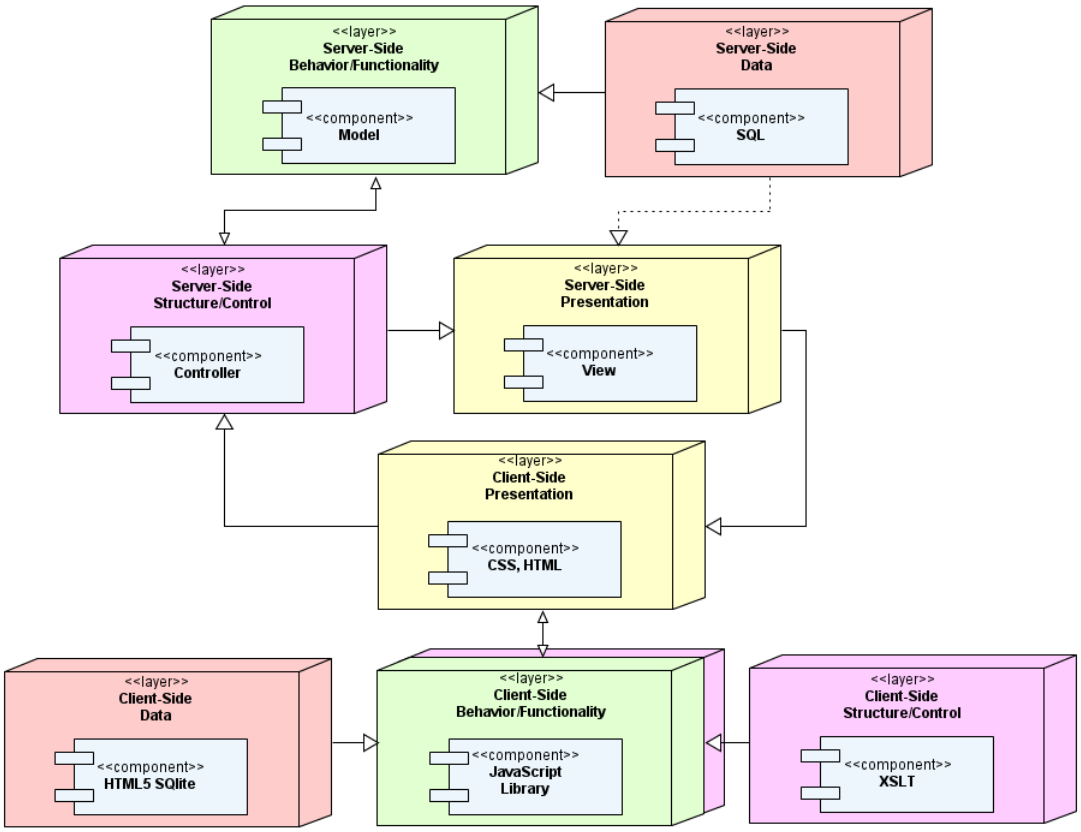
Chapter 3

Layers

Web application architecture is discussed in terms of a few generic layers, based on which responsibilities the different parts of the system handle, like presentation or business logic. We will introduce these *layers* now. Then, we will judge the validity of a *navigation layer* found in hypermedia web applications. Furthermore, the discussion will lead us to a *state layer* concern with an exemplary case. After seeing which layers are needed in a web application, we will be better equipped to evaluate the different approaches to web development later.

The 4 layer model or pattern is a dominant approach to splitting the different concerns of an application into its distinctive parts. A diagram can be seen in Figure 3.1 on the next page that illustrates a fairly broad picture of how a web application can utilize these layers. We see that these 4 layers concern themselves with the behavior, management of data, structure and presentation of an application. The diagram further shows that these layers do not just exist on the server but "live" in the browser as well. In fact, a JavaScript client-side framework might completely reside there. We will introduce the 4 layers as they relate to the different concerns of the application as we will continuously discuss them in the patterns part (Part III on page 66). The diagram itself is an example of the MVC pattern split.

Figure 3.1: 4-Layers Pattern.



Persistence Layer

Web servers do not maintain a state of an application as they are intended to serve large amounts of clients connecting from different locations. However, web applications do work with data and thus we need a layer that, indirectly, maintains them between requests. This is what the persistence layer takes care of. Usually it is in a form of a database, but it does not have to be.

Business or Functionality Layer

The functionality layer, as the name suggests, concerns itself with the functionality of the application, sometimes called the business layer. Here we would, for example, find the classes that calculate the price of our order in a web shop. Usually it is the business layer that makes connections to the persistence layer where it saves or retrieves the data it uses.

Control Layer

This layer is responsible for high-level flow control in the application. As an example, this layer would decide what to do when a product requested in an e-commerce ap-

plication is unavailable. As we will see, there are real pressures between this and the business layer as to which does what. This is exacerbated by the fact that the control layer often times receives requests from clients and has control over what happens next. One then risks having too little of a functionality in the control layer leaving the business layer "anemic".

Presentation Layer

This is the look and feel of the application, the front for the business objects that represent the application. But more appropriately, it represents the layer that communicates with the user and thus can, for example, represent a class that sends the user an email. The email, then, is an extension of the business logic of the application and the presentation layer is where it should reside.

However, as we will see, some authors view this differently and suggest the business objects themselves take care of the presentation as well. These will all be the issues that one needs to address and differentiate between appropriate approaches.

3.1 Navigation Layer

We will later examine frameworks from the hypermedia - web engineering domain. For now, we will focus on one of their common aspects, the use of the navigation layer and subsequent concerns. This, Schwabe and Rossi [1998] would argue, is the separating piece between traditional software engineering and web application "engineering". As we are discussing the concerns and layers a web application is split into, we need to discuss the validity of such claim.

Daum and Merten [2003] discuss the navigation layer as a concern that means, in effect, that when a user visits a topic on a web page, she should not have to jump to a different page to understand the topic. To put it simply, the *topic has to be contained in one page.*

3.1.1 National Gallery of Art

One example of a web application using a hypermedia approach is a National Gallery of Art website. We are going to now discuss this and an alternative approach which will lead to a discussion concerning the merit of using a navigation layer in web application architecture.

Figure 3.2: National Gallery of Art demo site.

Discenza [1999] have applied the navigation concern concept to a National Gallery of Art website[1] that they call a "web application". Their approach, using OOHDM framework (in more detail discussed in Chapter 5.7.1 on page 60), allows the developer to describe the interrelatedness of topics and pages so that the framework generator can generate the "application" as a set of pages - or nodes that the user travels through. In effect this means that the visitors to the NGA website can take "exhibitions" that consist of collections of paintings or art pieces with back and forward arrows to move between the collection much like one would do in a brick and mortar gallery. One can click for a detail of the art piece or visit a page discussing the author. Is the navigation layer a valid approach?

We have attempted to find out by designing the very same gallery as if it were an

[1]Still "live" today.

"ordinary" e-commerce store. The "products" in this case would be the paintings, the category the era we are exhibiting, much like in a web shop where one can click on a detail of a product or see a description of the manufacturer.

Soprano Framework

Therefore, we have designed a simple framework in PHP (example of which can be seen in Figure 3.2 on the preceding page) that would let us focus on the control and business concern in a straightforward fashion thus illustrating how we "solved" the navigation concern.

Figure 3.3: Soprano framework execution flow diagram.

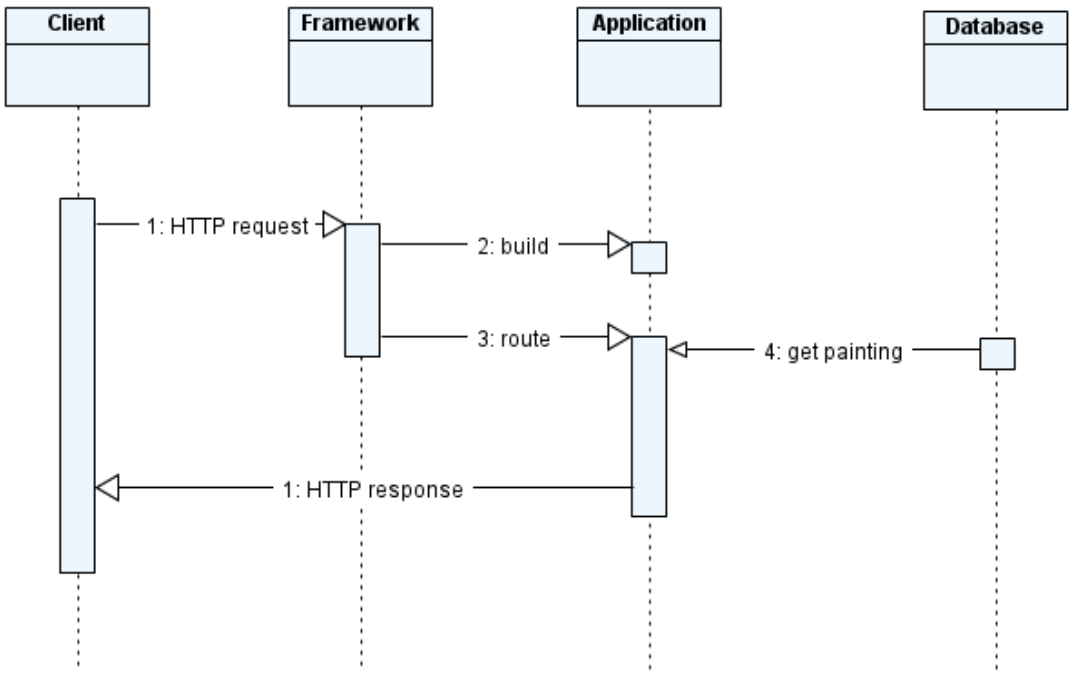


Figure 3.4: National Gallery of Art application.

```
<?php
include('../lib/soprano.php');

class Application extends Soprano {

    // painting listing
    public function getIndex() {
        $this->template->paintings = $this->dbSelect("SELECT *
           FROM paintings");
        $this->template->render('index');
    }

   // detail view of the painting
    public function getDetail($painting) {
        // ideally we would filter the query in the router and
           nevertheless the DB layer would throw exceptions based
            on failed validations
        if ($result = $this->dbSelect("SELECT * FROM paintings
           WHERE id=$painting")) {
            $this->template->painting = $result;
            $this->template->paintings = $this->dbSelect("SELECT
               * FROM paintings");
            $this->template->render('detail');
        } else {
            $this->template->render('no-detail');
        }
    }

}

$app = new Application();

$app->get('/', 'getIndex');
$app->get('/detail/:painting', 'getDetail');

$app->run();
```

Figure 3.3 on the previous page shows the flow chart through a request/response cycle, while Figure 3.4 shows the actual code of the application. Let us go through the request/response cycle step by step.

1. First the request from a client is received by the framework (through an irrelevant step of bootstrapping).

2. The framework instantiates the class that corresponds to our application, it has two methods and accepts two routes from the client. Let us say that a user's request will ask for `/detail/6`. This route corresponds to a pattern that we match for, `/detail/:painting` and a corresponding method `getDetail()`, accepting one attribute, `painting` that is extracted from the request. In our

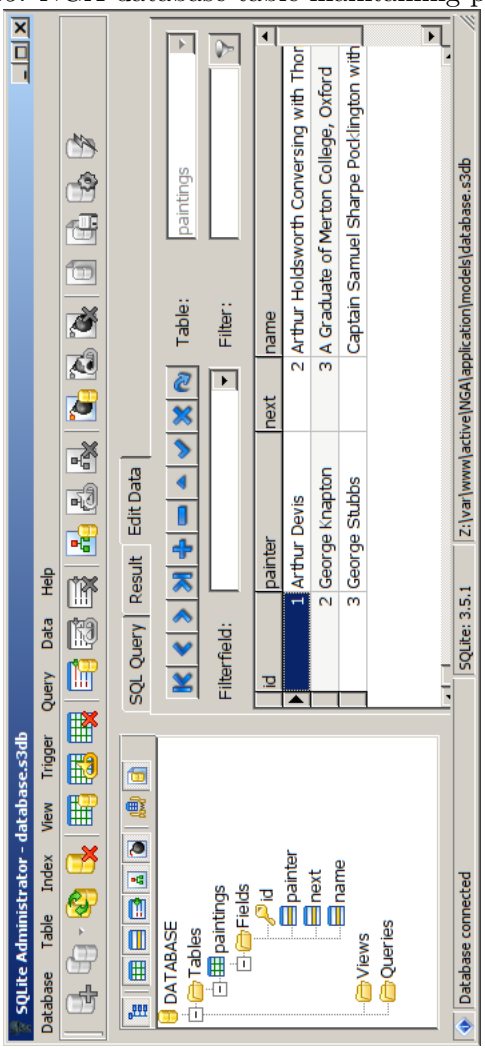
case, it is the string `6` [2].

3. The method `getDetail()` is thus called. This method makes a search for the painting with an `id` of 6 from a database. If found, it will respond to the client with the appropriate painting detail, and if not, it displays an error message built into a template.

Business Concern

The interesting part is how the "navigation" is solved. We would argue that what makes an art gallery different from a (random) stack of paintings is that the paintings are in a set with one preceding another, it is a *business concern*. Therefore, the business of moving from one painting to another is part of the business and persistence layers of the application. The database, in particular, specifies the order.

Figure 3.5: NGA database table maintaining paintings.



[2] At this point we have to clarify that PHP is dynamically typed.

Figure 3.5 on the previous page shows a database structure for a table that "holds" the paintings. Each painting is referenced by an `id` primary key that corresponds to the `painting` parameter in `getDetail()`. You will notice that we have another column, called `next` that simply has a reference to the next painting in a set, again referring to the `id`. Every time we display a painting, we get the link to the next painting as well, thus giving the *illusion of a navigation layer*.

3.1.1.1 Resources ≠ Pages

In Section 2.2 on page 25 we have introduced the use of Ajax, or asynchronous requests [Mahemoff, 2006], in web applications and this leads to our second point discrediting the navigation concern as not entirely valid. First, we have to repeat that resources in the web context are different from actual pages. A painting is a resource we are trying to retrieve from the application. If we wanted to, for example, make the paintings in a gallery load faster, we could use Ajax to retrieve the next image in the series behind the scenes (a predictive fetch [Mahemoff, 2006]) and still be on the same "page". If the request that gives us the initial layout we see in the browser is called a page, what is called the response that contains only the image?

Take another example. A user wants to edit a document in a web based CMS. She knows she will spend quite a while writing the text and we know that she needs to first login to an administrative area - she has a role. The way we know a user has already logged in between requests is to use a session[3]. A session usually lasts around 30 minutes and is refreshed on each new request from the client. However, in our example, the user is editing the document for longer than 30 minutes only to find out that when she tries to save it, she is met with a login screen asking her to authenticate herself again - the session has expired. A solution is to either alert the user that her session is about to expire or make an asynchronous request (Ajax) to the server purely to keep the session alive. The only thing we will receive from the server is a "200 OK" HTTP protocol response. Surely, this is one of the concerns in our system, while this concept does not render itself to be called a "page".

Our point is that web applications use client-side scripting through JavaScript more and more and this means that a "page" we see in the browser *cannot be neatly explained by one request/response cycle*.

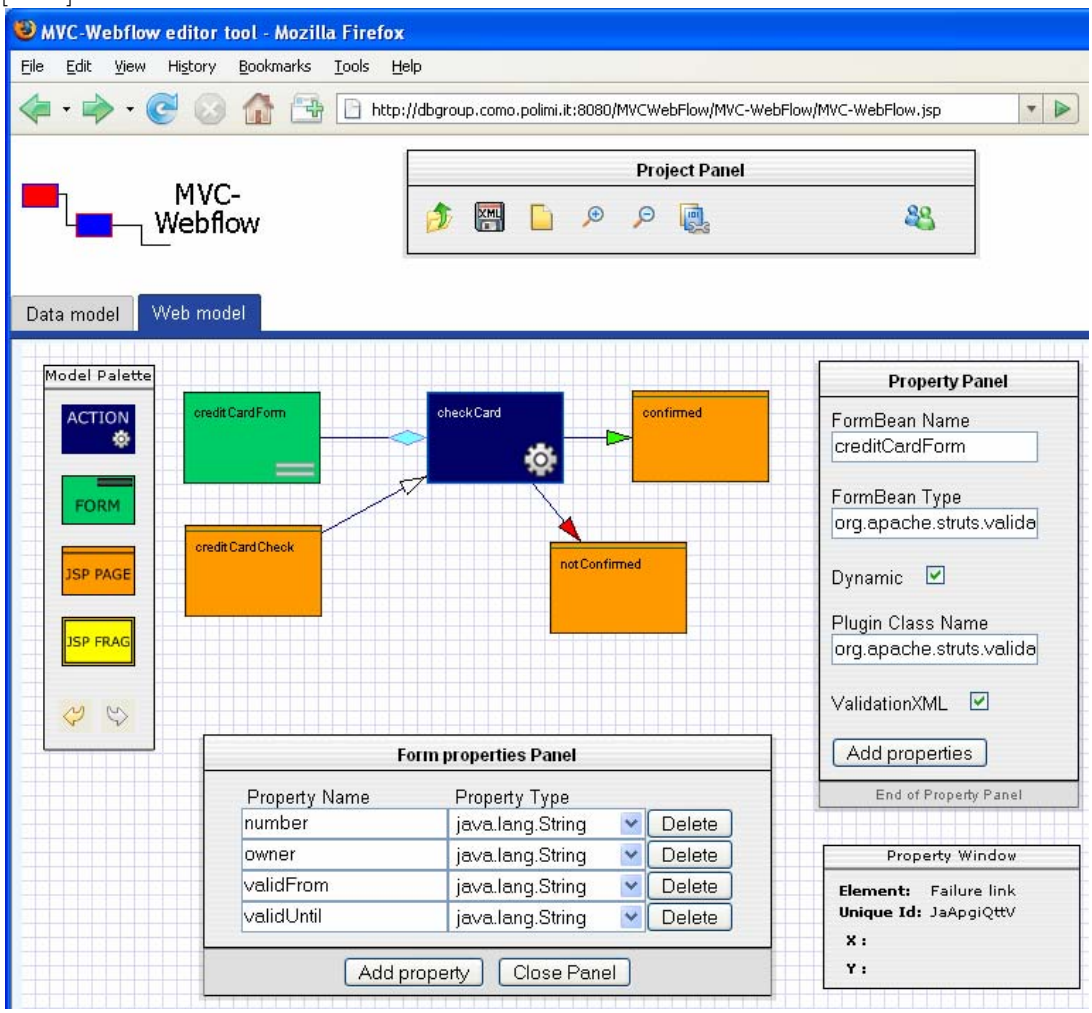
Now if we go back to the OOHDM framework generating separate pages for each painting (resource) the application holds, how can we make it more dynamic? What if we want to, again, use Ajax? How do pages now map onto requests that return

[3] A session uniquely identifies a client.

resource data instead of complete pages[4]?

Brambilla and Origgi [2008] have built a web environment for building web applications according to web engineering approaches. Specifically, the environment, MVC-Webflow, lets users graphically design (Figure 3.6) a navigation layer. But on closer inspection the "navigation" looks more like a use-case. Or rather, why use a "navigation" concern, when software engineering succesfully uses use-case diagrams [Booch et al., 2007] that describe a *flow of events*? We will spend more time discussing application flow in Part III on page 66.

Figure 3.6: MVC-Webflow navigation design, reproduced from Brambilla and Origgi [2008].



3.1.1.2 Semantic Web

What the navigation layer should instead be called in the OOHDM approach is an attempt to describe semantic relationships between resources that a web application manages. There is a concept of an "artist", "a theme" in a gallery as well as a concept

[4]They do not.

of a "painting" that can be clustered underneath "art pieces". The different artists and their paintings are then interrelated together. This is what semantic web does well [Antoniou and vanHarmelen, 2008].

Perhaps, rather than describing business concerns in terms of a navigation layer, a semantic map of interrelated concepts could be generated from well described business models. Then, users could use their dynamic Ajax presentation while computers would parse this semantic map, ontology, instead.

3.2 State Layer

By definition the HTTP protocol is stateless. This nature allows many concurrent clients to connect to a server and not hog its resources, mainly its memory [MacDonald and Szpuszta, 2005]. It is the persistence layer - the database that makes sure that data used in a web application are maintained between requests and among multiple clients. However, there are also ways that one maintains a state between the application and one client, these approaches differ whether they use a technique on a server or a client [Zimmermann et al., 2008].

3.2.1 State on the Client

Alur et al. [2001]; Morgan [2006] present approaches one can use when saving state of objects or the whole application on the client. It involves embedding information about the application objects into the response that gets returned to the client.

1. *HTML hidden fields.* This is the technique where a form is created with hidden fields that contain state information. The disadvantage is that a lot of information is saved on the client and thus this negatively affects the performance. An approach used in ASP .NET 2.0 is to encrypt this information on the client[5] and then validate the checksum when the data are unserialized[6] by the application, this means that invalid/modified data are discarded [MacDonald and Szpuszta, 2005].

2. *Cookies.* Cookies are similar to hidden fields but are easier to implement, however they might be blocked by the browser and are only of limited size. They are valid though, as MacDonald and Szpuszta [2005] note, for personalization as cookies can remain on the client for longer.

[5] With a directive `<%@ Page EnableViewStateMAC="true" %>`.

[6] Serialization is a process of translating objects into a series of bits or characters with a prescribed encoding so they can be transferred between systems.

URL Query

The problem with above mentioned approaches is that all of the data is maintained on the client and thus has to be transferred back and forth with each request. Furthermore, cookies can be blocked and a hidden form is harder to implement. Ideally, we would only like to maintain a sort of "pointer" that identifies our data on the server, only having to maintain this object - this is what *session id* does. Therefore a third, lightweight, option is to transfer this pointer as a parameter of a URL query. However, this makes it easily visible, thus open to modification and does not look appealing where the rest of the routing mechanism uses some sort of URL rewriting.

An interesting approach, where a framework helps the developer make use of "state saving" on a client can be seen in Nette Framework [*Nette Documentation*, n.d.]. There, one can simply specify which variables does one want to see transferred between requests in a URL query by way of using a PhpDoc[7].

Let us imagine an example where we are playing a simple game in a browser called "Fifteen". In this game one shuffles parts of an image to get their ordering right. The order in which each image appears can thus be kept in a variable called `fifteenOrder`. If we want this variable to persist between requests then we simply write the following syntax above its declaration `/** @persistent int */`.

We can then see the content of the variable being transferred between requests like so:

```
/temp/NetteFramework-0.9.3-PHP5.2/examples/fifteen/document_root/?fifteen-x=
1&fifteen-y=3&fifteen-order=1.4.5.2.8.7.14.10.12.13.6.3.9.15.0.11&fifteen-round=
1&do=fifteen-Click
```

3.2.2 State on the Server

MacDonald and Szpuszta [2005] discuss the robust architecture for maintaining state of a web application as provided by ASP .NET 2.0 technology.

1. *Session State.* Used for e.g. shopping baskets and is secure especially if SSL is being used. This represents saving data into a session that is accessed through the ID of the client.

2. *Application State.* An approach specific to ASP .NET where data are saved for all clients in one repository that represents the application. Such data are "saved" until the application or the server are restarted. As clients can overwrite each others saves to the data store, two functions, `Lock()` and `Unlock()`,

[7] A standard for commenting code.

are employed to explicitly allow only one client to access the application state collection at a time. Unfortunately this means the rest of the clients have to wait for the lock to be released [MacDonald and Szpuszta, 2005].

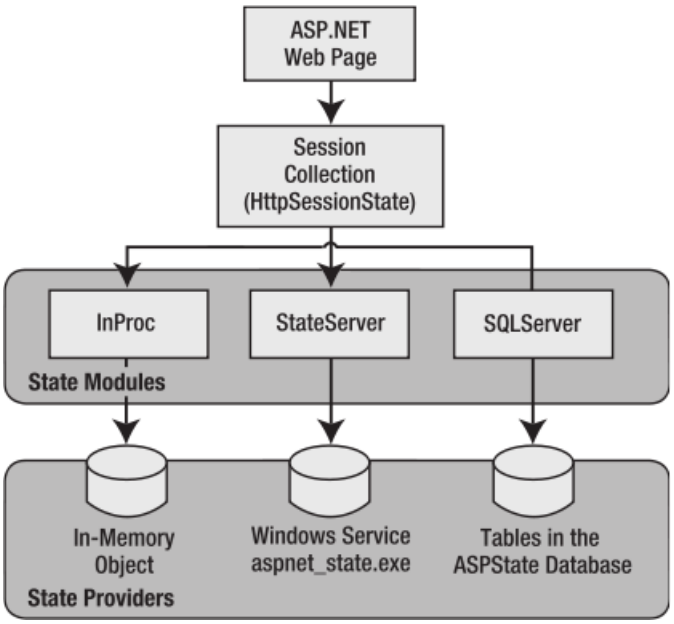
3. *Profiles.* An ASP .NET specific approach with data being serialized into a database. Therefore all other applications can access the data as well but this approach results in an overhead during the saving and retrieval.

4. *Caching.* An extension of profiles where data in the database can be optimized to perform a cleanup.

The diagram shown in Figure 3.7 represents the flexible nature with which one can save data on the server.

However, these approaches are platform specific and thus we have created an exercise where the user or the developer would not have to specify which data to save for the client but rather, see the resources the client uses as being stateful or not and if so then in which state they are. This is an approach an abstraction above simple saving and retrieving of data and as we will see has a lot in common with how desktop applications work.

Figure 3.7: ASP .NET session state architecture, reproduced from MacDonald and Szpuszta [2005].



3.2.2.1 ModelState

At the beginning of this chapter, we have discussed the business layer that holds the logic of an application. This layer usually consists of classes that we call models. The

approach we will illustrate here takes a model class of a Hangman game where one guesses letters that make up a solution word, but we will use no database connections. Instead the class has a declaration of variables (see Figure 3.8 on the following page) as if we were programming for the desktop. We see the use of variable `word` that contains the correct answer or `guess` that stores a collection of used letters etc. The code example also has two methods, one for starting a new game and thus getting a random word from a dictionary, and a guess method that is passed a letter.

Figure 3.8: Hangman game “business” logic.

```
<?php

class Hangman {

    private
        $word,
        $guess,
        $left,
        $round;

    public function startNewGame() {
        // parse a random word
        $words = file_get_contents(APP_DIR . "/models/words.txt")
            ;
        $words = explode("\n", $words);
        $this->word = strtoupper(trim($words[rand(0, count($words
            ))]));

        // initialize our word to guess
        $this->guess = array();
        foreach (str_split($this->word) as $letter) {
            $this->guess[] = array($letter => '_');
        }
        $this->left = $this->getLength();
        $this->round = 1;
    }

    /**
     * Make a guess.
     * @param  $letter
     * @return bool true if we have won
     */
    public function guess($letter) {
        foreach ($this->guess as &$g) {
            // if we have a match and are not guessing the
                guessed again...
            if (key($g) == $letter && current($g) == '_') {
                $g = array($letter => $letter);
                $this->left--;
            }
        }
        $this->round++;
        return ($this->left <= 0);
    }

    public function getLength() {
        return strlen($this->word);
    }

    public function getGuessed() {
        return $this->guess;
    }

    public function getRound() {
        return $this->round;
    }

}
```

What we are doing extra, is writing another class with the same name as our main class, but with a suffix `State` that extends a main state class (Figure 3.9). What this class does is simply provide us with a declaration as to how an initialized `Hangman` class looks like. As the code suggests, it calls the `startNewGame()` method introduced earlier. We have modelled this approach after a client-side framework SproutCore [*SproutCore Documentation*, n.d.] that, as it runs in the browser, needs to maintain the application in "some" state.

Figure 3.9: Hangman game "state" logic.

```
<?php

class HangmanState extends State {

    public function getObjectName() {
        return 'Hangman';
    }

    public function initialize() {
        $this->startNewGame();
    }

}
```

In order to put the functionality together we are using a Factory pattern [Gamma et al., 1995] that loads models with a state, if they are present, and optionally calls the state we want the model (and subsequently the application) to be in, if we pass it a custom state as a parameter (see Figure 3.10 on the following page). We will now describe the StateModel in more detail.

Figure 3.10: Generic “state” factory.

```
<?php

final class Factory {

    /**
     * Build a model object or its stateful adapter.
     * @static
     * @param  $model
     * @param  $state
     * @return Object
     */
    public static function build($model, $state=null) {
        $stateFile = APP_DIR . '/states/' . ucfirst($model) . '
            State.php';

        // is model stateful?
        if (is_readable($stateFile)) {
            include_once $stateFile;
            // database access
            include_once "db.php";
            // name convention
            $model .= 'State';
            // build it
            return new $model($state);
        } else {
            // return a new vanilla model
            if (is_readable($modelFile = APP_DIR . "/models/{
                $model}model.php")) {
                return new $model();
            } else Application::exception('model not found!');
        }
    }

}
```

In our example, we have used either a database or a session to store the model class in. The process works as follows:

1. Ask for a `Hangman` class maintained in a session on the server (serialized or hashed into a string).

2. `Hangman` class does not exist, but a `HangmanState` class does, therefore create a new `HangmanState` class that has `Hangman` as its inner class. This is an example of a Decorator or Proxy Adapter pattern [Gamma et al., 1995], with `HangmanClass` “wrapping around” `Hangman` and directing all method class to it.

3. Once we have processed the request, just before we are responding to the client

all classes are destroyed by the server. Therefore we have a hook in a PHP magic[8] method `__destruct()` that lets us define what happens to a class we work with just before it is destroyed. The `HangmanState` class extends `State` that has `__destruct()` defined as a method that serializes the inner `HangmanClass`.

4. Repeat, in the next step though, we are going to unhash a class in a particular state this time.

3.3 Discussion of Layers

First, we have introduced the persistence, business, control and presentation layers that form a 4 layer pattern model describing what each layer is responsible for. However, as we will see, such a nice delineation in an application is rarely possible, and that is why our dissertation will discuss harder problems where we have, for example, a presentation concern, that needs to make calls to a persistence concern. If we were not keeping, in such a scenario, a good separation of the different functionalities, we would end up with crosscutting application concerns, that are discussed in the next chapter.

We have also challenged the navigation layer seen in hypermedia frameworks stating that it does not provide many benefits in web applications that are dynamic and see the web as a collection of resources rather than pages. The closing part of this chapter discussed ways of maintaining state in a stateless environment of HTTP. We have seen how one can persist data in a database or maintain state with a client through the use of server and client based approaches. Our example of maintaining state of a whole class showed how a framework can be of good use to a developer, simply allowing her to plugin code that she has learned to use on a desktop. However, we need to note, that the example presented only works as an example. In a world of web applications that communicate with thousands of clients saving the whole class would not be a good idea storage and speed-wise and resorting to using a database is usually the way to go. The same can be said of Application State method from ASP .NET that results in quite the overhead.

[8] Official term, see `http://www.php.net/manual/en/language.oop5.magic.php`

Chapter 4

Concerns

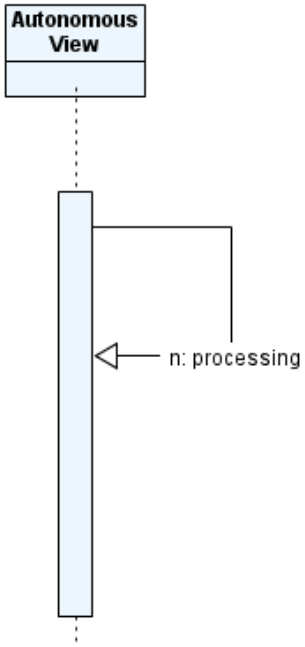
In the previous chapter, we have commented on the 4 layers a web application usually uses through the process of handling a request from a client and responding to it. In addition, two layers - state and navigation have been demonstrated with appropriate discussion. What the layers represent is the concept of concerns, introduced in this chapter. We will see what effect the mixing of concerns has on software quality and thus understand what scenarios to avoid, or conversely, aim for.

Atkinson et al. [2002] tell us that as web applications have a greater requirement for distinct levels of abstraction and interface wrapping concepts, traditional ad-hoc reuse methodologies will not work. These distinct levels of abstraction and wrapping of concepts are referred to as the concerns.

Autonomous view

In relation to concerns, let us introduce a first "architectural" pattern, the Autonomous view. In this approach all the application logic (control and business layers) and any database connections (persistence layer) are contained in a single class or file (see Figure 4.1 on the next page).

Figure 4.1: Autonomous view.



This approach is, however, beneficial only when mocking or creating small applications as one cannot test such a solution well [Williams, 2007] and one cannot work on one concern at a time without fearing that other concerns will break their connections to the rest of the application.

4.1 Separation of Concerns

The last point is a converse product of separation of concerns, the concerns are intermingled. What this represents is that a developer cannot handle one problem at a time and the application is hard to reuse [Schwabe et al., 2001; Batenin, 2006]. Figure 4.2 on the following page shows an example of such an application that parses a website and according to some pattern-matching rule sends an email notification, writes to a file and displays a result on a screen.

Figure 4.2: Intermingled concerns in a web application - an autonomous view.

```
<?php

$url = "http://www.specialtip.com/"; # url
$input = @file_get_contents($url) or die('Could not access file:
    $url'); # load

# regex
mb_internal_encoding("UTF-8"); # set encoding
if (preg_match("/Aktivní .* (tip|tipy)./", $input, $matches)) {

    # file
    $fh = fopen("status.txt", 'r'); # open file (read only)
    $d = fread($fh, 200); # read file

    if ($d != $matches[0]) { # file is different (new status)
        echo '<span style="font-family: calibri">new status<br>';
        $fh = fopen("status.txt", 'w'); # delete original
            contents
        echo 'file cleared</span><br>';

        fwrite($fh, $matches[0]); # write new status to file

        # send emails
        $msg = $matches[0]."\n\n".'http://www.specialtip.com/tipy
            ';
        $to = array("email@gmail.com", "email@fbi.gov"); #
            recipients
        foreach ($to as $key => $recipient) {
            mail($recipient, "Nový tip od SpecialTip.com ".date("
                D dS M, Y h:i a"), $msg,
                 "From: admin@tipsfortune.com"); # send mail
        }
        echo '<span style="font-family: calibri;background:#
            cefcd2">notifications sent</span>';
    }
    else echo '<span style="font-family: calibri;background:#
        f8f9cc">no new picks, some are active</span>';
    fclose($fh); # close file
}
# no new tips (clear the file)
else {
    $fh = fopen("status.txt", 'w');
    fclose($fh);
    echo '<span style="font-family: calibri;background:#fed9dd">
        no active picks</span>';
}
```

For example, if we wanted to change how the output message in the example looks like, we would have to first find the line that says the following (Figure 4.3 on the next page) and edit the style applied.

```
echo '<span style="font-family: calibri;background:#cefcd2">
   notifications sent</span>';
```

Figure 4.3: Notification output message.

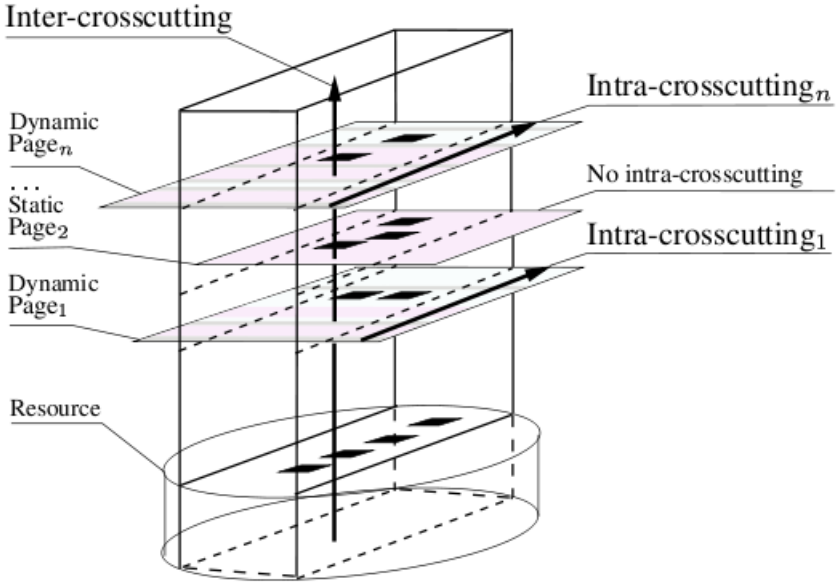
4.2 Crosscutting Application Concerns

Kojarski and Lorenz [2003] have compiled two approaches that differentiate between how a concern is intermingled in an application, a concept they call crosscutting.

Take the example of a table that needs to be drawn on the client's screen. As we are dealing with the act of displaying an object, the code should rest in a presentation layer that would have a logic pertaining to how the table will look and how it is filled with data. However, we might find a piece of such functionality in a business layer, a layer where it does not belong. We would call this an *intra-crosscutting* code, autonomous view being a prime example of such a concept by design.

The other variant, an *inter-crosscutting* concern, Kojarski and Lorenz [2003] find, is where a functionality is in a correct layer but is spread across multiple files. For example, the beginning of a `for` loop of the code for displaying the table, in our example, will be in file A, while the rest of the code will end in file B. Figure 4.4 shows a diagram representing layers of code of a web application with crosscutting concerns present.

Figure 4.4: Crosscutting concerns in a web application, reproduced from Kojarski and Lorenz [2003].



We can see 4 layers that the application is split into. Each layer should represent only one application concern but the different colors correspond to the use of code

belonging to more than one concern. At the same time, the "black" boxes show a concern that should be kept in one class or a file but is split between numerous units.

4.3 Discussion of Concerns

The purpose of this chapter was to show that code that is crosscutting concerns is hard to modify, update and find, a recipe for disaster in terms of code reuse and business agility were there to be updates to the codebase. On the other hand, a small application can exhibit crosscutting if we know that it is not going to be updated further, or if it serves as a mockup. In such a case we do not need to split code into layers.

Now that we have introduced the web architecture, the layers and concerns it consists of, we can discuss several approaches, in an increasing level of abstraction, that are used to develop a web application that exhibits separation of concerns, is easy to update and contains reusable code.

Chapter 5

Web Application Architecture

In the previous chapter we have determined what concerns a web application consists of and arrived, towards the end, at a discussion about separation of concerns and code reuse. In this chapter, we will first discuss the advantages of having code that is reusable and well separated and then discuss the approaches one uses when developing a web application. The discussion will take us from good principles of object-oriented programming through patterns to frameworks that work as application skeletons ready to be reused.

5.1 Reuse and Associated Issues

Reuse broadly means that software components are written once and then "reused" as they have common parts that should be assembled together rather than written from scratch [Garlan et al., 2000]. In a web development market with so many technologies and ways of transferring and processing data, this seems like a good call, why does it not happen more often as Baskerville and Pries-Heje [2004] have determined, especially given it helps in cost-effective quality software construction [Schmidt and Buschmann, 2003]?

Batenin [2006] suggests that *cost* is an important factor determining the extent of reuse of a web application (system). Someone has to be paid for putting extra effort into coding a reusable component Krueger [1992] adds. Therefore Batenin [2006]; Lang and Fitzgerald [2007]; Lang [2009] all find that as effort costs time, developers will, in order to meet a schedule, *ignore good practices to save time.*

As it is hard to imagine future requirements, and time lost designing and coding today is time needed tomorrow, Extreme Programming (XP) [Beck and Andres, 2004] advises not building flexible components on purpose. After all, in this agile methodology, working on things for the future is outside of the contract the developer has with

the customer. This can be viewed as reasonable, as why should the current customer pay for cost savings for the next customer? Or should the next customer pay off the original one as proposed by Schmidt [1999]? The customer cannot reuse the software either, as Lynex and Layzell [1998] find that companies will keep their code to keep their competitiveness rather than become benevolent with their intellectual property.

Unfortunately, the non-technical problems do not stop there and thus for more discussion on the barriers to reuse see Schmidt [1999].

However, there is a cost effective way of making applications more reusable. It is through the separation of concerns where applications become more reusable as a consequence of improved modularity [Batenin, 2006], a concept introduced in the previous chapter. Therefore the rest of the chapter describes the various approaches to code separation. Before we discuss the architecture of web applications and how it influences the separation of concerns, however, we need to understand what represents an "architecture" and how to evaluate it.

Parnas [1979], a classic in the field, has stated that when one builds up a system, one should decompose it into modules so that difficult design decisions are hidden from others. These modules encompass the system's components, connections and constraints [Gacek et al., 1995]. A collection of stake-holder's need statements and a rationale demonstrating that these statements will be met, represent a software system architecture [*ibid.*]. Such a system can then be reviewed and go into full development. Shaw and Garlan [1994] have stated that such a design and specification of the overall system structure is becoming increasingly more important then the choice of end algorithms or data structures. Their finding was corroborated by Clements [1996] that suggested that designing an appropriate architecture with appropriate connections between its parts allows for an economical production without sacrificing speed or program correctness.

5.1.1 Architecture

In this chapter we are discussing design decisions that impact how a web application behaves. Bass et al. [2003] conveys that the way we discuss the design decisions has "a profound impact on all software engineering work that follows". In order to gain insight into the architecture, we need to ask two questions Bass et al. [2003] debates:

1. We need to determine what is the flow and transfer of *control* between parts of the system and in what hierarchy these parts or components exist.

2. Consider the communication of *data* between the various components. Are the data shared, perhaps through an intermediary? Or are they globally available?

These questions are rather informal and this fact is mentioned in Pressman [2001] where the author concludes that at the moment, there are no quantitative measures that would allow us to evaluate architectures. Even the work by Asada et al. [1992] attempting to rectify this situation is pseudo-quantitative in nature [Pressman, 2001]. That is why we will describe the approaches in this project in qualitative terms [Kazman et al., 1998].

5.2 Programming Paradigms

Let us begin with the most generic programming paradigms, first with the mainstream, then with the alternatives.

5.2.1 Object-oriented Programming

Object-oriented programming is the mainstream programming paradigm on the web, working with the concept of objects consisting of parameters, methods and their interactions.

In particular, let us cover a couple of principles we need to keep in mind when evaluating a development approach from an architecture viewpoint. The following principles[1], collated by Robert [2002], can avert poor dependency management that would normally lead to code that is hard to change, fragile, and non-reusable.

1. *The Single Responsibility Principle* - "A class should have one, and only one, reason to change [DeMarco, 1978; Page-Jones, 1988]."

2. *The Open Closed Principle* - conveys we should be allowed to extend class behavior, without modifying it [Meyer, 1988]. This principle is especially useful in web development as applications have a shorter release cycle and thus are expected to change often [Jacobson et al., 1992].

3. *The Liskov Substitution Principle* - posits that a subclass should behave like the super-class. Let us illustrate that this principle does not apply only to inheritance. We could extrapolate it to mean that if a request is made to the server, the code representing an abstract response to the client should be capable to handle the request in a similar way. For example, it should not matter whether we ask for data in format X or Y or whether we request page A or B.

4. *The Interface Segregation Principle* - suggests the use of fine-grained client specific interfaces; then the clients using them only know about the methods they are

[1] Or patterns, but let us not confuse them with patterns discussed later on.

going to use [Robert, 2002].

5. *The Dependency Inversion Principle* - advises we depend on abstractions and not on concretions.

These are the general meta-concepts upon which less abstract approaches, discussed in the proceeding sections, are built.

Aspect-oriented Programming, Subject-oriented Programming

These paradigms try to improve the modularity of cross-cutting concerns. In practice this means that supporting methods in a program are isolated from the main business logic. An example of which would be a "logger" that has to be called at the start and end of each method call.

Class reflection[2] and possibly the Intercepting Filter pattern might at least partly solve the problem of automatically calling helper methods. On the other hand, an Observer object can be employed to notify its dependents of state changes [Gamma et al., 1995].

Mainstream web development is presently still dominated by OOP and thus we further discuss approaches purely relating to that programming paradigm.

5.3 Inheritance

In this part we address the use of inheritance, an OOP concept, from a design perspective in web development. It is a concept achieving code reuse where an object hierarchically inherits behavior from another object, called super-class.

The constant extensions and iterative nature of web application development means that developers often use inheritance for fear of breaking existing classes by modification, describes Batenin [2006]. However, this approach, Knoernschild [2001] finds, is one of the most catastrophic mistakes that contribute to the demise of an object-oriented system.

Composition and multiple inheritance represent alternatives to the single inheritance approach. Let us look at them in turn.

5.3.1 Object Composition

We cannot discuss inheritance without mentioning the *composite reuse principle* and a subsequent Strategy pattern [Gamma et al., 1995] that emphasize we should take

[2] A method where a program can observe the classes and methods it has at its disposal, thus allowing it to modify its behavior.

advantage of *object composition*[3] *over inheritance* to achieve code reuse and separation of concerns.

Namely, the landmark book by Gamma et al. [1995] has warned us that inheritance can represent problems when we are trying to reuse a class. Chiefly, with inheritance, any unfit object method for our problem domain has to be replaced or the parent class has to be rewritten which limits flexibility and ultimately reusability.

On the other hand, composition, as argued by Gamma et al. [1995], requires us to carefully design interfaces which leads to fewer dependency problems, smaller classes and thus greater separation of concerns that aids design reuse as a consequence. The composition also leads to a flexibility when designing a class, thus leading to concepts, or concerns, that are separated from the rest and only used together as part of a lightly coupled object produced from an object factory.

As we have discussed in Chapter 2 on page 23, web applications increasingly make extensive use of API calls. Therefore, as we have to rely on a third-party service, Venners [2005] describes composition as the favorite approach to tackle this scenario. Sutter and Alexandrescu [2004] also find weakest relationship between components to be advantageous to method overwriting found in inheritance.

As an illustration, this weakest relationship is supported by a concept of autoloading of classes, such as the one found in PHP. A class is only instantiated when needed, just before an exception would have been thrown. We therefore delay the creation of objects until they are actually required, a beneficial approach concludes Venners [1998].

5.3.2 Multiple Inheritance

Multiple inheritance is a concept where a class inherits features and behavior from multiple classes, usually by following a left-to-right rule. As of 2010 it is not implemented in the mainstream web server languages. In any case, we believe that in such a case where we would like to inherit from multiple classes, we should reconsider the way we have designed our application and instead of relying on hard-coded compilation rules assemble objects through *composition* as such approach is more flexible.

5.3.3 Structured Inheritance Relationships (SIRs)

Gardner [2001] has discovered that problems associated with inheritance can be resolved by applying and using inheritance in a structured way. Therefore, she came about developing structured relationships that use inheritance:

[3] A class contains other, inner, classes.

1. *Variant relationship* - serves to introduce an operation in a super-class with implementation variants in its subclasses. However this relationship can be replaced using a Template Method pattern discussed in Section 5.7.3 on page 62.

2. *Construction relationship* - similar to *variant* but we are not necessarily looking for code substitutability. Should we then not use object composition if the relationship is not an exact "is-a" instead? Object type can be enforced through interfaces.

3. *View relationship* - achieves code reuse such as "a view of a target is-a target instance looked at in a particular way [Gardner, 2001]". These could be roles that an abstraction plays during communications with other abstractions. Therefore, as an alternative, the Role Object pattern [Bäumer et al., 1997] suggests objects to have separate *role objects* that are dynamically attached and removed. While, if we want to simply provide a further set of operations for use in certain contexts, we can use the Decorator pattern that allows us to broaden an object's functionality dynamically [Gamma et al., 1995].

4. *Evolution relationship* - a useful relationship in which a complex abstraction is built up over time. Alternatives are a Template Method pattern if the core logic stays the same. An Adapter pattern, on the other hand, can be used to convert an interface of one class to be what another class expects. Such would be a case of a new version of a class extending the original one, an evolution.

The problem with SIRs is that they attain code reuse, but no separation of concerns. This is understandable as the comparison between inheritance and composition using patterns has not been discussed by Gardner [2001]. But the author's message is clear: developers feel lost in the sea of patterns and approaches available.

5.3.4 Traits

Language designers have proposed various forms of multiple inheritance, as well as mechanisms such as *mixins* that compose a class from a set of features as an evolution of the single inheritance concept, yet these approaches have not been accepted widely in the past, Taivalsaari [1996] notes.

The technique of traits has the benefit of allowing us to apply a common code at arbitrary places, Schärli et al. [2003] set forth, thus achieving reuse. As of 2010, this technique has not reached a wide acceptance in the web development community. As we will see in later chapters, one architecture approach, Data, Context and Interaction

(DCI), predominantly uses mixins which makes the approach less applicable in different languages.

5.3.5 Proper use of inheritance

It would seem that the proper place for inheritance is only in a subclass "is-a" superclass relationship, called specialisation SIR in Gardner [2001]. In such a case, subclasses follow the Liskov substitution principle stated in Section 5.2.1 on page 52.

But even then we can come up with patterns that offer an alternative such as the Decorator pattern[4] where an object has a role rather than an "is-a" relationship, Venners [1998] observes. See Fowler [1999], Genßler and Schulz [1999] and Opdyke [1992] for discussions on refactoring inheritance into either delegation [Grand, 2002] or composition and aggregation using patterns, discussed in the next section.

5.4 Patterns

Patterns are simply approaches, found in practice, that solve a recurring problem. The first pattern we have mentioned was the 4-layer architecture. It solves the problem of having multiple concerns in an application by splitting them into 4 layers. Authors such as Dong et al. [2005] believe that system design will effectively become the composition of many design patterns. Patterns are extremely popular, perhaps as they are discovered in practice, not invented. Schmidt and Buschmann [2003] say they provide time-proven solutions to commonly occurring software problems for a given context or a domain. Appleton [1997], investigating their advantages, states they improve maintainability, reusability, communication commonality and encapsulation variation.

Booch [2007*a*] has mentioned that to him the architecture is the implementation of the "warp & woof" of patterns. This shows how the field has now moved to describe architecture in terms of patterns used. Since the publication of a landmark book by Gamma et al. [1995] on design patterns, our understanding of how to best present patterns has matured considerably, we in fact see patterns about patterns [Booch, 2007*b*] and pattern languages [Coplien and Schmidt, 1995; Zamani et al., 2008]. This has been allowed by the publication of handbooks [Booch, 2007*b*] by authors such as Alur et al. [2001]; Fowler [2002] that have looked at enough real systems to reveal these. They describe "ordinary solutions" that help software projects from failing when they are lacking [Coplien and Schmidt, 1995]

Shaw and Clements [2006] have described 1996-2003 as a phase in software engineer-

[4]Decorator pattern changes the interface of an object without modifying it [Gamma et al., 1995].

ing called an "internal enhancement and exploration phase". This phase was marked by the shift to architectural patterns as a means of describing the design of software [Shaw and Clements, 2006].

In the words of Redwine and Riddle [1985] it typically takes 15-20 years for a technology to enter widespread use. Shaw and Clements [2006] suggest we are now in a golden age of software architecture where it gains widespread adoption. This was 4 years ago. We would argue that because of this widespread adoption, many ideas and advancement has been made by the layman that has not necessarily been published well or up to a standard. Thus our work is intended to fill the gap, to bring these ideas, particularly pertaining to architectural patterns, back to the sphere of academic research where they can be further advanced and scrutinized. Citing from Shaw and Clements [2006], we need to now organize the architectural knowledge to create reference materials ideally finding the right language to represent architectures.

Buschmann et al. [1996] categorize patterns into three general groups[5], as follows:

Architectural Patterns

This patterns group represents a fundamental structural organization schema for software systems providing a set of predefined subsystems, specifying their responsibilities and including rules and guidelines for organizing the relationships between them [Buschmann et al., 1996]. An architectural pattern would be a solution to a problem as to how the different concerns are handled in the layers and how the layers communicate with each other. For example, one pattern might suggest we accept client request in a controller component that then calls models only to display the result in a view. Thus, we have an overarching solution that incorporates all layers of the web architecture paradigm. The situation has certainly changed since Garzotto et al. [1999] expressed that only a few effectively usable patterns have been proposed for the web, as we will show later. But we also need to state, that, as we will find later (Part III on page 66), many architectural patterns do not solve all concerns a web application has.

Design Patterns

Design patterns are an abstraction below architectural patterns and discuss and refine the connections between components of subsystems. These communicating components solve a general recurring problem in a given context [Gamma et al., 1995].

For example, we have earlier introduced the Factory pattern. Such a pattern would be useless as an architectural pattern as it does not handle the interconnections between

[5]These are, however, set in stone [Blewitt, 2006].

the different concerns of a system. However, it can be applied as a tool that gives us varying combinations of components without us having to write multiple constructors for a class. In our discussion of the state layer a few sections ago, we have seen how a Factory created a Proxy for neatly wrapping the concerns of one class into a component with increased functionality, a Decorator [Gamma et al., 1995].

Idioms

Idioms are patterns specific to a given programming language used, therefore the features of the given language are used [Buschmann et al., 1996]. For example a Template Method pattern in PHP language is easily developed using an *abstract class* concept. Thus we are using a concept that is specific to one or more languages to derive to a solution described by a pattern.

Other Patterns

Martelli [2007] reminds us that the question is not whether or not patterns are useful, but when, how and in which language are we implementing them.

One of their domains is Web Interaction with examples, inter alia Breadcrumbs, Autocomplete or Site Map. However, such patterns do not represent the overarching ability to solve problems in all the layers of a web application and thus will not be discussed further.

5.5 Modifiability Tactics

Different authors have different ways of describing and documenting patterns, in particular describing the characteristics a pattern resolves. This means that one cannot easily compare two pattern alternatives or styles. Bachman et al. [2007] have recently attempted to rectify the situation and described tactics. An architectural tactic is a design decision that discerns how software architecture addresses a particular quality attribute [Bachman et al., 2007]. For example, there is a "reducing coupling tactic" that makes use of patterns to achieve exactly that, reducing a coupling between components so they can be reused. We will discuss the tactics that patterns exhibit in the next chapter that is devoted to the different patterns in detail.

5.6 Algorithms

While patterns primarily address concerns like maintainability, algorithms Appleton [1997] tells us, address issues of *computational complexity*. For example, fewer database

queries or a better array sorting technique will have a profound impact on the speed of a web application. Refactoring into patterns, on the other hand, might actually slow down the application. We have discussed the pressures a web development team is under at the beginning of this chapter, a main one being the business agility, therefore we will not discuss algorithms further, as they come second to making a reusable application that is easy to update and work with.

5.7 Frameworks

Frameworks are ancillary application skeletons, we can think of them in two ways:

1. As *semi-complete applications* consisting of reusable and extensible sets of components [Appleton, 1997]. These domain specific components, patterns [Fayad, 2000], form the architecture of the application and leverage the domain knowledge of experienced developers as a consequence [Schmidt and Buschmann, 2003].

2. As *reusable designs* built from sets of classes and models of domain-specific object collaborations [Schwabe et al., 2001].

Thus we understand that frameworks are half-baked applications, utilizing a particular *domain knowledge*. Sure every framework developer would like her framework to be the "magical wand" solution to web application development, but frameworks are most useful when they match our (business) domain as found by Fayad [2000]. Our task will be to extract the collections of objects, architectural patterns, that form frameworks so we can better understand what domains they are best suited for. How are they useful?

Specifically, Fayad [2000] believes that these micro-architectures leverage capital-intensive software investment through reuse with modularity, reusability, extensibility and inversion of control being their greatest benefits. As frameworks decide how classes collaborate and the thread of control, they achieve design reuse rather than code reuse, Gamma et al. [1995] remind us.

Moreover both Fayad [2000] and Schmidt and Buschmann [2003] back the notion that frameworks will be at the core of web applications and technology in general.

For a more detailed background on frameworks, we direct the reader to Fayad [2000]. For now let us introduce two common types of frameworks that stand before the software developer:

1. *White-box frameworks* - require us to understand the internals to use them effectively while usually taking advantage of inheritance, Kegel and Steimann [2008] state.

2. *Black-box frameworks* - require no deep understanding and behavior is usually extended by object composition [Wake, 1998].

The difference between these two approaches - inheritance and object composition - is discussed in the previous sub-sections, for now let us take at look at frameworks from the academic and industry domain.

5.7.1 Web engineering approaches

Academic frameworks such as Object Oriented Hypermedia Design Method (OOHDM), WebML (Web Modeling Language), HDM2000, WARP or WebComposition represent a class of "hypermedia" frameworks, considering web applications as navigational views over an object model, providing basic constructs for navigation and for UI design as defined by Schwabe et al. [2001]. Their purpose often is to accomplish *navigation reuse*, such as was the case with National Art Gallery site project [Discenza, 1999] that we have disputed in Section 3.1 on page 30. However, let us still briefly introduce them to the reader.

Jeary et al. [2009] have studied 23 web development projects by IT practitioners over the course of 2 years and found that only 1 project used an academic framework throughout the life-cycle. To put it simply, application of these frameworks is temporal, the industry does not use them citing cons such as their incompleteness, no practical guidance, terminology and formality issues [Barry and Lang, 2003; Lang, 2004; Brambilla and Origgi, 2008; Jeary et al., 2009]. Examples of such are:

1. *WARP* [Bochicchio and Fiore, 2005] - has a schema editor (HDM2000), a content manager (XML) and a publishing editor (XSL). Alternatively this problem could be approached as one of document publishing and versioning resulting in a CMS. For example, SilverStripe CMS with its Sapphire Framework allows us to determine our own schemas and an admin interface is automatically generated for us, all from one application. Django is another representative of an admin building web framework built in Python. We will discuss these further in a section of Naked Objects architecture (Chapter 13 on page 119).

2. *OOHDM* [Schwabe and Rossi, 1998; Rossi et al., 2000] - the reasoning behind the Abstract Data View architecture used is to allow the application to generate data according to the context. Unfortunately, alternative, MV* patterns based approaches, such as Model-View-Presenter (MVP)[6] are not evaluated by Rossi

[6] We evaluate this pattern in Chapter 10 on page 99.

[*ibid.*]. Rossi has since moved on to name *web design improvements*, a catalog of navigational and presentation web application models.

3. *Web Actions Framework* [Knight and Dai, 2002] - the authors are describing an MVC pattern, yet do not mention this fact anywhere in the literature.

4. *Atomic Section model* [Offutt and Wu, 2010] - rather than representing a framework per se, ASM constitutes an approach of splitting a page into atomic parts called a component interaction model (CIM) with a fixed application transition graph (ATG) that directs the user through nodes - pages. A finite state machine - state transition graph aids in testing, and such approach has its merit [McDonald and Welland, 2001], but this solution does not deal with the separation of concerns problem and results in code being mixed throughout HTML tags.

For an overview of academic frameworks and approaches see Jeary et al. [2009].

Web Engineering discipline

All approaches have one thing in common, they try to support the emerging field of web engineering. Kappel et al. [op. cit.] characterize this discipline as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of web-based applications". This field has its merits, such as improving on testing, analysis and quality evaluation of web applications. And testing has been gaining prominence. Our approach is, rather, invigorating the research on architectural patterns than building a beeline process for "engineering" a web application. Barry and Lang [2003] in their survey comparing traditional and multimedia development have remarked that much of the literature on web and multimedia development fails to appreciate the legacy of experiences in traditional information systems development and other root disciplines. Pressman [1998] too recommends not forgetting the lessons of the past commenting that web development should be carried out in the same manner as traditional systems development. Warned by these statements our project will focus on successful software engineering principles, reviewing confusing or poorly understood concepts in an academic way and in a web application domain.

5.7.2 Architectural patterns-based approaches

Architectural patterns, discussed in the previous sub-section, represent the basis for highly popular frameworks such as:

- *Struts, Grails*, *JSP* - aimed at Java developers

- *ASP .NET MVC* - based on Microsoft's ASP .NET
- *CakePHP, CodeIgniter, Nette, SilverStripe, Symfony, Zend* - PHP offerings
- *Django* - a Python framework that automatically generates views, the presentation of our application, and leaves us to primarily define the business logic of the application
- *Ruby on Rails* - a Ruby framework

Thus we see the wide application of architectural patterns and can, in our project, analyze them in a real-life setting.

5.7.3 Discussion

Authors such as Reinhartz-Berger et al. [2002] especially claim that "current modeling methods support design of generalized frameworks or patterns but do not support *open reuse* - the ability to develop partially specified components and refining them in the target application". We challenge this notion and suggest a Template Method [Gamma et al., 1995] where a high level algorithm is encoded in an abstract base class and makes use of pure virtual methods to implement its details. Such approach results in a lighter coupling yet we specify the required methods, achieving separation of concerns and subsequently reuse. As an example, an abstract `AuthenticatorTemplate` class in Fari Framework defines the action flow for user authentication, while the `AuthenticatorSimple` class implements the details, in this case the database connection and the columns we need to check credentials with. This is to say, we can *reuse generic models in different contexts.* This point will be reiterated and expanded upon in a latter chapter of this project.

5.8 Middleware

Middleware leverages frameworks and patterns to make the application's functional requirements bridge to the underlying operating systems, network stacks and databases [Schmidt and Buschmann, 2003]. They are the platforms such as Common Object Request Broker Architecture (CORBA), Java 2 Enterprise Edition (J2EE), or .NET.

We will review Microsoft's .NET as it is a middleware as well as a "framework" thus representing a collection of web domain patterns such as Model-View-ViewModel (MVVM).

5.9 Discussion of Web Application Architecture

This chapter first showed that a reusable application code in a system leads to a cost-effective quality software construction. However, the amount of reuse is hindered by numerous non-technical issues in the industry such as which customer pays for code that will be used again. We have then seen that ones applications can become more reusable as a consequence of improved modularity through the separation of concerns that we discussed in Section 4.1 on page 46. Thus in the rest of the chapter we discussed various techniques one can use when developing an application. We began with programming paradigms first to introduce OOP and in particular its SOLID principles (Section 5.2.1 on page 52). They will be remembered when evaluating patterns, models and architectures in the next part of the dissertation as their application results in more reusable, clear, well separated code. Then we saw that inheritance in terms of architecture should be replaced by object composition (Section 5.3 on page 53). This lead us to a section on patterns that represent solutions to commonly recurring problem (Section 5.4 on page 56). It is there we differentiated between architectural and design patterns and idioms as they are applied. In conjunction with a chapter on frameworks we saw how a combination of patterns into a framework, or a collection can best result in a complete application architecture.

We also found out that architectures will need to be discussed in terms of qualitative not quantitative benefits they offer (Section 5.1.1 on page 51).

Chapter 6

Summary of Part II, Web Application Architecture

The topic of our study is web application development. We started off with a definition of web applications and in what environment they work. Thus, we have highlighted the fact that, in comparison with coding applications for the desktop, web applications need to process the request and response cycle of each client-server request with application components being loaded anew each time. Furthermore, this model is muddled with asynchronous requests to the server that fetch and update content as required and client-side storage using HTML5 that will become a standard. Consequently, web applications need to respond to a request in various formats such as HTML, XML or JSON. In essence, the specificity of web development stems from the various technologies employed and the ease with which one can code a server side script, often encouraging cut-and-paste code reuse.

Therefore we concentrate on the qualitative aspects of web development, finding that web applications often lack best-practices delivering solutions that are sub-optimal, something Lang [2009] calls "pragmatic satisficing". This leads us to the finding that a systematic reuse, be it design or code, results in a better quality software. The reason reuse is not more heavily applied is due to the *costs* associated with employing reuse approaches like middleware, frameworks, various patterns, inheritance etc. We especially draw the conclusion that architectural & enterprise patterns, when properly understood, can accomplish the biggest cost/benefit ratio when utilized in frameworks. To conclude, we take the approach that proper understanding of core patterns leads to a better understanding of frameworks. And in turn, these frameworks represent a systematic basis for development, resulting in more reusable web applications that have their concerns well separated.

Let us underscore our summary with a quote by Buschmann et al. [1996]:

> "Patterns expose knowledge about software construction that has been gained by many experts over many years. All work on patterns should therefore focus on making this precious resource widely available. Every software developer should be able to use patterns effectively when building software systems. When this is achieved, we will be able to celebrate the human intelligence that patterns reflect, both in each individual pattern and in all patterns in their entirety."

Therefore, in the next part of the dissertation, we exclusively discuss patterns, methods and architectures that describe the composition of web applications. This will allow us to critically evaluate the different approaches and determine what parts of the application architecture should exhibit which behaviors.

Part III

Patterns, Methods & Architectures

We started the previous part of the dissertation with a general understanding of what web applications are and in what environment they operate. In particular, we noted that modern web applications work in synchronous and asynchronous mode, fetching and passing information from and to the server as needed, without necessarily needing a page refresh.

We continued with an introduction to the 4 concerns layers that we need to differentiate from and that find their place in the majority of web applications today. Thus, we understood the need for their appropriate separation that results in reusable, updateable code that leads to business agility of the developed software. This agility has been highlighted in a discussion about software reuse. The social and economical factors that influence how much a developer "does reuse" are directly correlated to how easy and fast she can do it. Again, the focus on speed of development is paramount, while speed of execution is not necessarily a primary concern.

This led us to a discussion of the different design approaches, be it inheritance and their SIRs or patterns and frameworks that are all used (in various combinations) during a development process. It is here that we understood that if we want to encompass all 4 concerns of a web application, we need to pay attention to *architectural patterns that in combination with their design counterparts, form frameworks*. Frameworks are, then, reusable skeletons of future applications ready to be tweaked and extended. This is where this section of the dissertation comes in. In here, we will present to the reader a number of architectural patterns, methods and whole architectures that have various opinions on how we structure code in the 4 layers often repeated.

In each chapter we will, in turn, understand why we need to understand this approach, then define it and primarily discuss it. Each approach will present something good (be it just by showing something inappropriate that needs our attention) that we will use in the next part of the dissertation (Part IV on page 139). We need to understand all of the following presented approaches, their history and variants as not doing so would lead us to a rediscovery of already discussed problems. We do so as too often we see a discussion about web architectures recommending the Model-View-Controller (MVC) pattern but not showing its numerous variants and alternatives.

Chapter 7

Patterns

Shaw and Clements [2006] have debated that we are in a "golden era" of software architecture where patterns are widely used in the industry. We would argue that this allowed for numerous approaches and variants of the said patterns to be invented, and now it is time to look at them all, and decide what benefits they have, especially in the field of web development. This would lead to a reference text that might last a while, until it becomes obsolete again in a few months time and in need of updating again. This chapter introduces patterns as a communication medium and represents an introduction into the review of current patterns and approaches.

7.1 Knowledge Vaporization

The need for reference materials is underscored by a major problem in the field, namely, architectural knowledge vaporization [Harrison et al., 2007]. This problem occurs when a knowledge about a program exists only in the heads of its architects or other stakeholders. These are decisions that are not recorded during the development cycle and cannot be inferred from the model designed [Harrison et al., 2007]. Eventually, this knowledge dissipates and future developers are left scratching their heads when trying to evolve a system such that the requirements match the implementation. Harrison et al. [2007] suggests taking a patterns approach where these dictate a particular system decomposition into modules and their implementation resolves certain challenges having consequences on the system. The aim of this part of the dissertation, is to resolve these challenges through the use of patterns as a communication medium.

7.2 Pattern Documentation

Patterns, as a communication medium, give advice about a particular topic [Fowler, 2006*d*], being often described as a combination of solution to a problem in a context.

We can find multiple ways of describing and talking about patterns, but there are a few forms that stand out; we will briefly introduce them now.

7.2.1 Pattern Form

Alexander et al. [1977]; Evans [2004] advanced what is now called an *Alexandrian* form of discussing patterns - a pattern for documenting patterns. It consists of an example use of the pattern, its context, background, different ways one can see the pattern manifested, solution, an instruction and a diagram describing the pattern's main components. An important part is that patterns are related to smaller patterns from a set that usually complements the pattern in a higher hierarchy. We could say that this is akin to design patterns (Section 5.4 on page 56) complementing their architectural counterparts.

Gamma et al. [1995] were the first ones to write a book about patterns in software development and subsequently, their structure is called a *Gang of Four* form. In it we can find separate chapters for each pattern discussed describing the intent, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses and related patterns of the pattern described. This form is in contrast with *Portland Pattern Repository (PPR)* form that describes a problem the pattern is solving followed by a couple of paragraphs describing the solution to a problem the pattern addresses, thus serving as a simple way of discussing the topic.

As our aim is not in making a new repository of patterns[1], we will introduce each pattern with a summary of why the pattern needs to be reviewed, the problem - solution tuple standard for all patterns followed by a discussion, as appropriate, of related patterns, the pattern's web form & its inner working, concluding with a discussion addressing the advantages, disadvantages of the solution proposed with a conclusion fitting the pattern into the overall topic discussed. The reader will find that the pattern summary is similar to the *Pattern-Oriented Software Architecture (POSA)* form [Buschmann et al., 1996] with examples being included to illustrate a problem much like in the *Patterns of Enterprise Application Architectures (PoEAA)* book [Fowler, 2002].

[1] But it will be our suggestion in the conclusions.

During the patterns review we often found authors describing them in a set. Some authors try to extend this idea and cluster patterns into "languages" [Coplien and Schmidt, 1995; Zamani et al., 2008]. On the other hand, the original authors of the patterns argue that their work does not compromise a language [Fowler, 2006*d*]. Our work does not attempt to develop a language. Languages and clusters of interlinked patterns can come later after we introduce the patterns initially. And, if we have done our research right, the patterns we have researched and will present next have never appeared in the same document being discussed together. Our contribution comes in a form of a recipe describing pattern combinations.

When we discuss them, it is important to keep an open mind. In the words of Booch [2007*b*], patterns are legitimate solutions to a given problem simply because they exist and they *resolve the forces on the system* where they are applied. As we have identified the forces that influence the web application environment earlier (Part II), we are ready to discuss the patterns themselves now.

7.3 Discussion of Patterns

This chapter re-introduced the topic of patterns, in particular how they are used to "document" an architecture of a system. We saw that *knowledge vaporization*, where information about why and what in a system is lost, is a big problem in the field. The use of patterns then represents a form of documentation as they describe solutions to problems commonly found. These problems are then the concerns and *forces* that the architect had to juggle.

We followed with a brief review of a number of pattern forms used in the field by experts. None of them represented a "golden standard" of how patterns should be discussed which leads us to a conclusion that we will use an informal approach when discussing patterns too, for now, making sure that we identify a problem - solution tuple each pattern has, followed by a discussion of benefits and disadvantages complete with sample diagrams to complement this abstract topic.

The following chapters each discuss an important pattern that will highlight a particular beneficial solution to a problem we will use in our example architecture constructed in Part IV on page 139.

Chapter 8

Model-View-Controller

We will start our discussion with a pattern that is found in most web application frameworks [Singh et al., 2002], the Model-View-Controller or MVC. Our discussion needs to first focus on this pattern as many other patterns and architectures reference it and improve upon it. We will need to also overview this pattern so that a useful design patterns such as Data Transfer Object can be discussed in the future. They will effectively provide a solution to the issue of accessing data from a presentation layer of the application and complement MVC.

Among MVC's claimed benefits belongs the separation of design and business concerns, reduction in code duplication, centralizing control in an application and making the application easier to modify overall [Singh et al., 2002]. Singh et al. [2002] further comments that by splitting an application into different concerns and centralizing control, developers with different skill sets can focus on different parts of the [web] software.

However, we claim that if the split between the concerns was really done appropriately then we would see the industry reusing old business classes, models, and only updating them and their presentation, with ease. But rather, web applications are being built from scratch, even though this pattern has been used for awhile. Why is it so? We believe it is because the pattern is not well understood and developers then code systems that result in tangled code (Section 4.1 on page 46).

Fowler [2006*a*] agrees that people understand different architectures and ideas under the umbrella team "MVC". Kukola [2008] adds that this pattern is perhaps the most misunderstood (out of user interface models) and that trying to understand the many MVC adaptations leads to confusion as multiple and conflicting definitions of the pattern are given.

Our own research is based on a finding that when one tries to read more about the pattern, one only finds a reference to the original MVC pattern idea, irregardless of whether the architecture being built is on the desktop or the web. No papers are published describing alternatives of the pattern and the web engineering field goes a different way. These statements are underscored by a finding by Deacon [1995] that much of the information regarding the idiom is only available in the form of folklore (now blogs) rather than in textbooks, a fact that still holds true today.

We will attempt to first describe the pattern in a usual problem - solution manner. However to properly understand it we will need to look to the roots of the pattern, its desktop origins, then its web application only to end with a healthy discussion about the pattern, its consequences and alternatives. As we will find out, the statement that MVC is one of the most applied pattern on the web has a twist and a discussion about MVC has been long overdue.

8.1 Problem

Avgeriou and Zdun [2005] define a problem as follows:

> "A system may offer multiple user interfaces. Each user interface depicts all or part of some application data. Changes to the data should be automatically and flexibly reflected to all the different user interfaces. It should be also possible to easily modify any one of the user interfaces, without affecting the application logic associated with the data."

8.2 Solution

The solution to the previous problem is to separate a system into three parts, or layers:

1. *Model* component that encapsulates application data and the logic for manipulating the data. This should happen independently of the user interface(s) used.

2. As has been mentioned, the system has to have a user interface. This part is the concern of a *view* component.

3. The third component is a *controller* that is associated with views, encapsulating more or less extensive application logic.

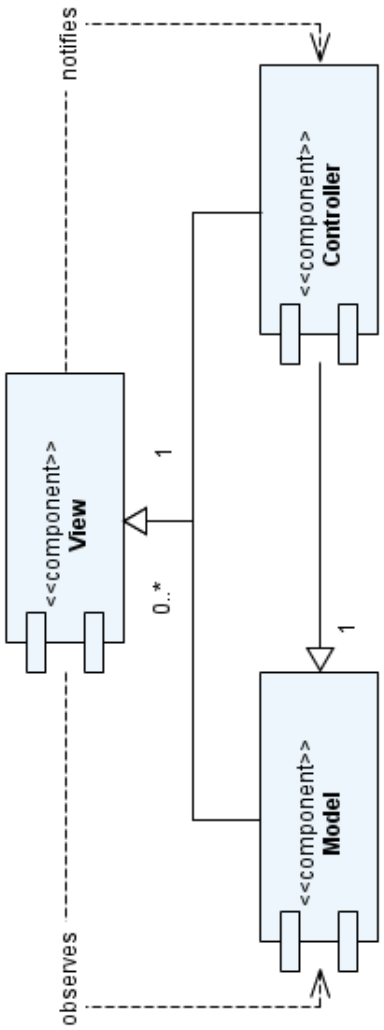
We have tried to limit the solution definition as the different versions of MVC, as we will see, behave differently.

8.2.1 Observer and Event-Listener Patterns

It is obvious that all parts of MVC have to have clearly defined communication channels. This is also a good place to introduce two patterns commonly used in desktop MVC. First one can make use of an *Observer* pattern. Gamma et al. [1995] define an observer as an inter-object one-to-many relationship. If one object changes its state, all dependent objects are automatically notified and updated [*ibid.*]. Its implementation will make sure that the model registers with all its views and sends them notifications about its serviced data. All views then interact with the model asking for changes and refreshing what they display. As an example, the model layer manages messages in a chat-room program. The model notifies the views that its data have changed so the views call the model back refreshing their output with new messages in a room.

The other communication pathway we need to sort out is the transfer of user requests from the presentation layer to appropriate controllers. An Event-Listener also known as Publisher-Subscriber pattern is often used. Being related to the Observer pattern, Publisher-Subscriber not only notifies its subscribers (or listeners) but sends them the actual data [Spell and Gongo, 2000]. In our example the user clicks on a button that creates an object wrapping this event, this object is then sent along to an appropriate controller. A component diagram of a generic MVC architecture is shown in Figure 8.1 on the next page. In the diagram we see the focus on the view that observes the model and notifies the controller to make changes to it when an event is received. Bear with us for a second as the exact flow of events will be described in the next section dealing with the different variants of the pattern on the desktop.

Figure 8.1: Component diagram of a generic Model-View-Controller architecture.



8.3 Desktop MVC

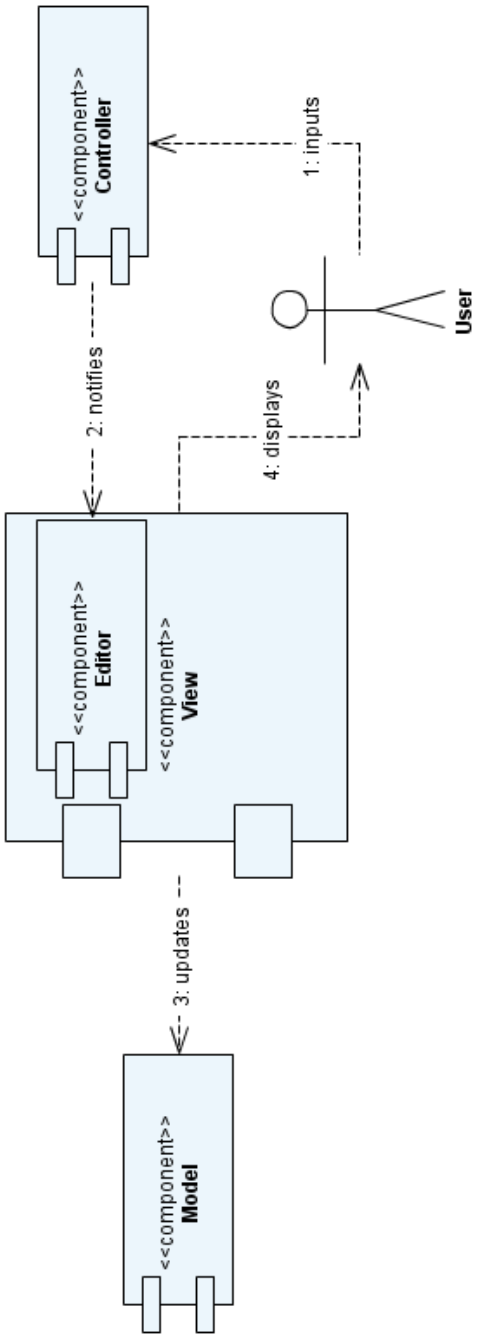
We will now look to the origins of MVC and its historical versions before we move on to how MVC works in the web applications domain.

8.3.1 MVC/79

MVC/79 was devised by Trygve Reenskaug when working at Xerox PARC. He has based his vision around his experience when building a system for managing shipyards. MVC is an improved upon term based on an an initial ideal called Thing-Model-View-Editor [Reenskaug, 1979*b*]. MVC/79 was never implemented and we base our findings only on a few less then detailed technical reports from that time.

We can discern that user input is intercepted by controllers that manage and create views. *Editors* represent a special component, similar to a controller that provides an interface for model handling. Kukola [2008] believes that these components apply commands through queries applied via the view. A sequence diagram showing a flow of events is shown in Figure 8.2.

Figure 8.2: MVC/79.



In the diagram we see the importance of a user as an integral part of the system.

The user launches an event by clicking for example on a submit button in a form. This event is passed as a notification from the controller to the view. The controller does not do any processing, it is the editor that is part of the control concern, deciding which model - a business layer, to update and subsequently displayed updated to the user through the view.

8.3.2 MVC/80

MVC/80 is an implementation of the MVC concept in Smalltalk-80. User input is handled through a controller that is an implementation of a *Strategy* pattern [Gamma et al., 1995]. Controllers also deal with interactions scheduling [Krasner and Pope, 1988] and "own" multiple views. The views concern themselves only with the output - visual aspects of the application and leave application decisions to the controller [*Mac OS X Reference Library*, n.d.]. We see an implementation of the Observer pattern discussed before where multiple view-controller pairs observe the model as dependents. The main idea behind MVC/80 is to separate the business layer, the input and the output with its models, controllers and views respectively. The view layer is further split into a composite of individual views - such as a menu or buttons, therefore [*Mac OS X Reference Library*, n.d.] refer to MVC/80 as a Compound Design pattern of Composite, Strategy and Observer patterns. A sequence diagram is shown in Figure 8.3 on the next page.

Figure 8.3: Traditional version of MVC as a compound pattern.

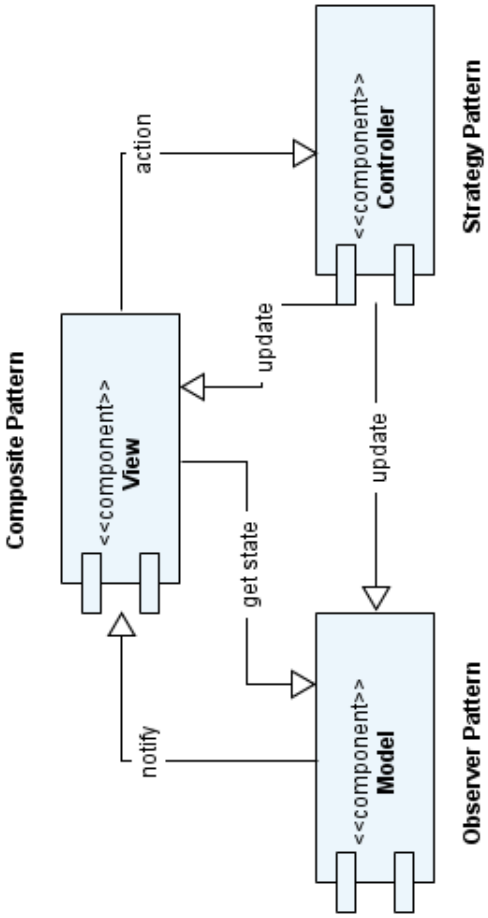
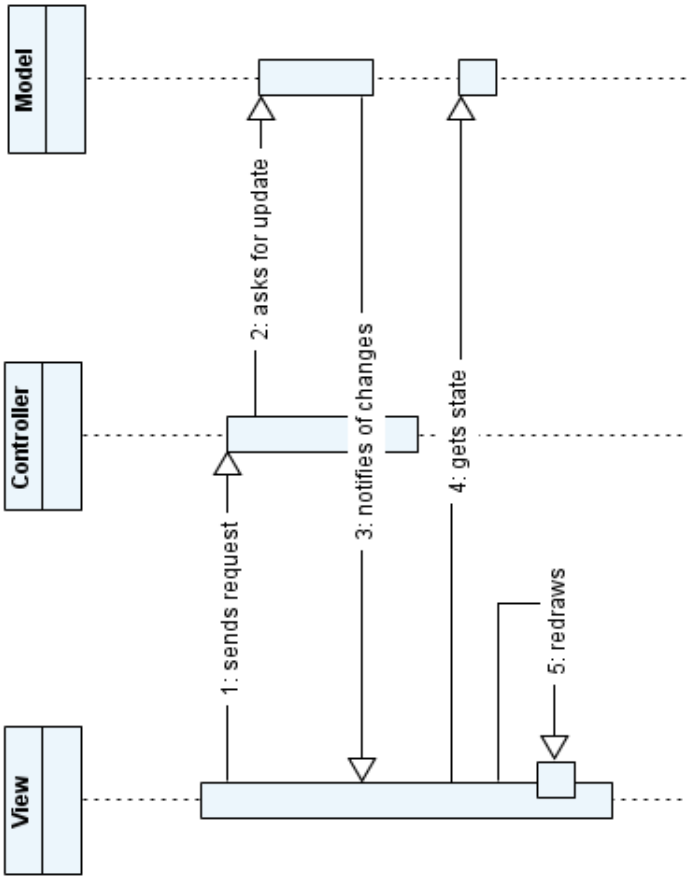


Figure 8.4: Sequence diagram in a class Model-View-Controller architecture.



As this version of MVC was implemented, we can clearly show the flow of events through the different layers of the system:

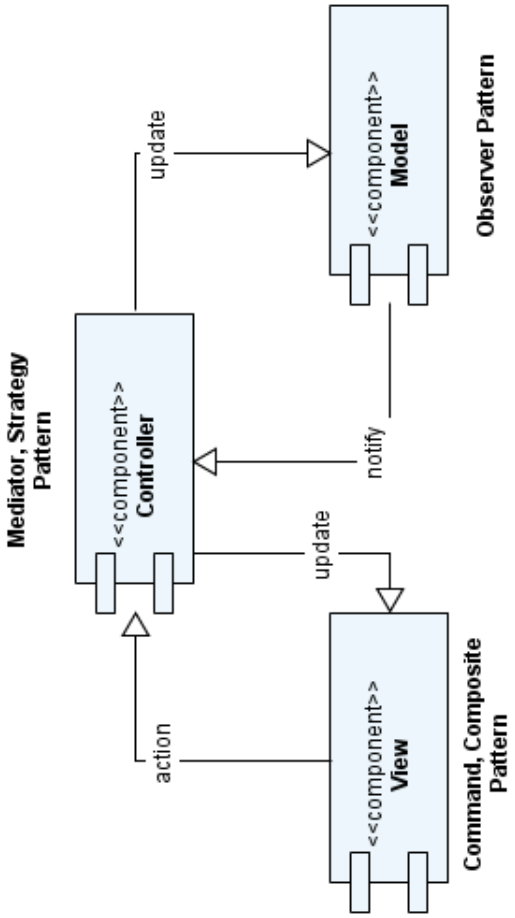
1. A controller is notified of an event triggered by the user on a view component.
2. The model is subsequently updated with the new information contained in the notification. This is an example of a strategy employed on part of the controller.
3. The Observer pattern is employed to notify the view of a model state change.
4. The view updates its state. We see that this is an example where a user has a view of the model and both need to stay synchronized.

8.3.3 Cocoa MVC

Apple Inc. saw a problem with the MVC/80 version of the pattern arguing that views and models should be most reusable in an application [*Mac OS X Reference Library*, n.d.]. Views need to look and behave consistently in an operating system and therefore they need to be highly reusable. On the other hand models need to be well separated

from the presentation layer as the concern is completely different - the logic. Therefore we see the application of MVC in Mac OS X called Cocoa MVC [*Mac OS X Reference Library*, n.d.] that uses controllers as objects implementing the *use an intermediary* and *restricting communication paths* tactics, explained below. It means that notifications of model state changes are communicated to views through controllers and vice-verca, a Mediator pattern [Gamma et al., 1995] application. In this sense, the Web MVC approach will have more in common with Cocoa MVC than with MVC/80, as we will see in the next section. We conclude this part with Figure 8.5 showing the application of Mediator pattern in Cocoa MVC.

Figure 8.5: Cocoa MVC using Mediator and Strategy pattern in the controller.



1. We see that the controller is now the mediating part allowing for more functionality. A Command pattern, in this case, represents a specific method in a controller being invoked as a response to an event in the view. We call the specific method an "action". As the model is no longer directly connected to the view, we call the controller as performing a *restrict communication path* tactic a special case of *use an intermediary* that removes a dependency between a called and calling

object.

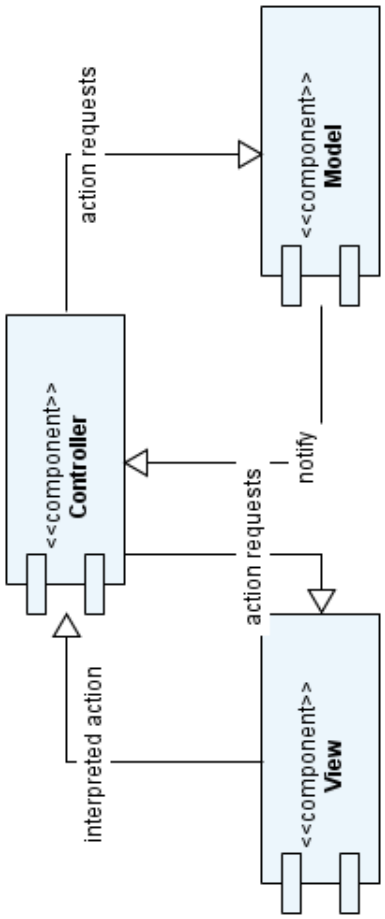
2. The controller action decides which models need to receive the new information.

3. Finally the controller is notified of a state change to the model and thus updates the views that depend on it.

8.3.4 Model-View-Controller++

This version of MVC can be seen as an in-between step between MVC/80 and Cocoa MVC. In comparison with MVC/80 the controller acts as a Mediator and is no longer contained in the view handling input for the view (Figure 8.6) [Jaaksi, 1995].

Figure 8.6: MVC++.



Still though, there is a main controller that handles dependencies between sub-controllers deciding which actions to take. This approach and a 1-to-1 relation between a controller and a view means a tight coupling that gets worse as changes propagate through sub-controllers [Jaaksi, 1995].

We will get ahead of ourselves and mention that in Web MVC it is common to

see an application controller and an action controller that inherits from it. But the application controller only gets involved when a global action should happen on each request, working as a form of Intercepting Filter pattern[1].

8.4 Web Model-View-Controller

In this section we will present the classic architectural pattern Model-View-Controller as applied in the specific environment of web applications. First we need to discuss two communication strategies used in Web MVC.

When it comes to frameworks on the web (and most of them are based on MVC), we discern between two types of strategies used, we present them now as they separate the way this pattern is applied.

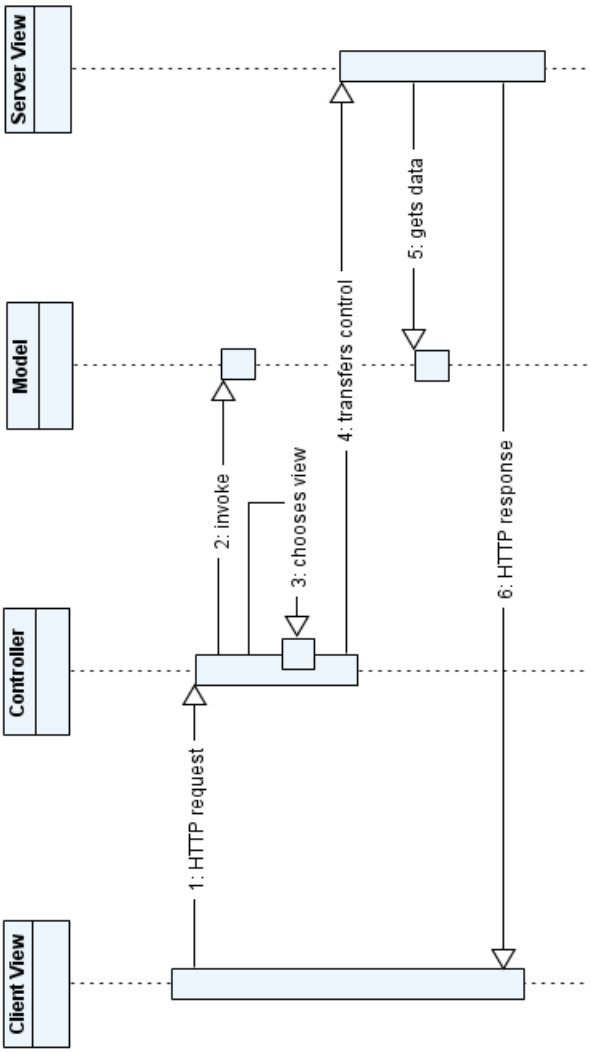
8.4.1 Push Based / Service to Worker Strategy

This is the strategy most web frameworks follow. The controller layer processes a request and then "pushes" the result to a presentation layer - the view. Framework examples are: Zend, Ruby on Rails, Django[2], Struts, Spring MVC or author's own Fari MVP. This approach restricts the communication path and gives focus to the controller layer, thus is similar to how Cocoa MVC (Section 8.3.3 on page 78) on the desktop has been devised and Figure 8.7 on the following page attests to that.

[1] This means that a request has to pass through a number of components - filters on its way through the system [Alur et al., 2001].

[2] We will discuss its MVC variant, Model-Template-View, in Chapter 12 on page 115.

Figure 8.7: Service to Worker.



In the Service to Worker frameworks as the processing focus is put on the controllers layer it is common to see the use of a Front Controller pattern [Alur et al., 2001; Fowler, 2002]. This pattern centralizes all requests from the client and decides which controller - associated with one or more models, to call. An example follows.

Ruby on Rails

Web framework Ruby on Rails [Alameda, 2008; Ruby et al., 2009] is a Ruby based MVC approach that uses the Service to Worker strategy to make it easy and automatic to create CRUD (Create-Read-Update-Delete) applications. Such applications are specific with their focus on databases. As this framework has support for all parts of the development process, and subsequently all parts of the MVC pattern, it is sometimes called a DSL (Domain Specific Language). One can, for example, independently of

other parts of the application, communicate and operate with the business models through a terminal[3]. A request processing in Rails MVC can be described as follows:

1. All requests are routed through a Front Controller that is a specialisation of an `ApplicationController` class. It decides which controller and action to launch. For example requesting `/paintings/delete/5` will instantiate a `Paintings` controller and call its method `delete` with a parameter `5`.

2. The controller than makes a connection to an appropriate number of models to process the request.

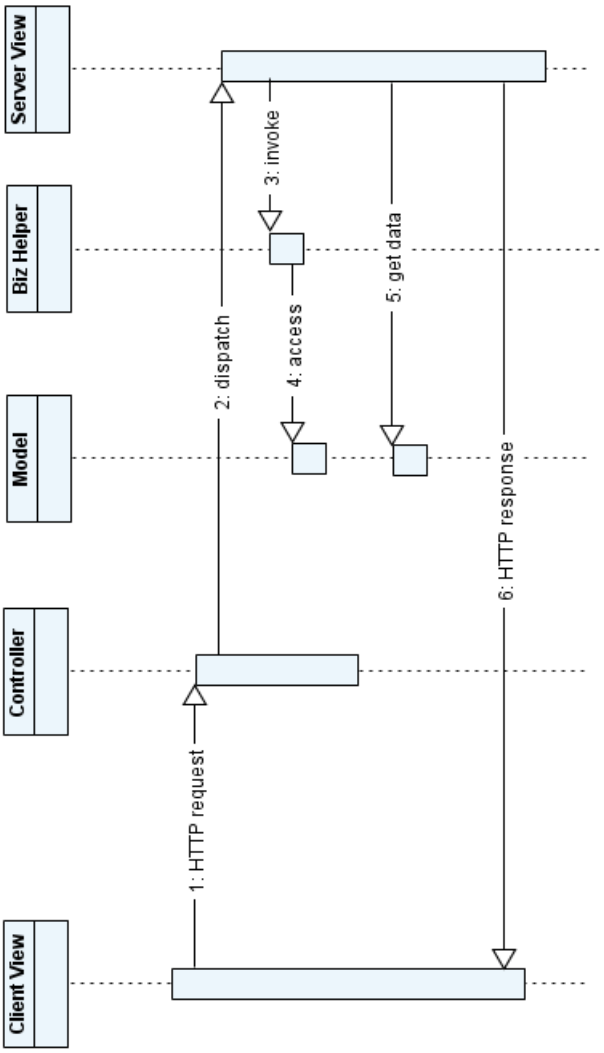
3. Finally, a view associated with that controller action of the same name will be invoked traversing received data as appropriate and sending them back as a response to the user.

8.4.2 Pull Based / Component Based / Dispatcher View Strategy

In this strategy, the views are commonly associated with many controllers with the views "pulling" required data from them. This approach is used in the Wicket, Struts 2 or JSF frameworks and is similar to how MVC/80 is organized. Figure 8.8 on the next page shows a flow diagram through a request, explained on an example of JSF that follows.

[3] A Command-Line-Interface.

Figure 8.8: Dispatcher view.



JavaServer Faces

JavaServer Faces or JSF represents an official specification of Sun (now Oracle) for web frameworks based on Java EE[4]. Thus it forms a combination of rules and approaches that one has to employ to create compatible frameworks and subsequently applications based on the technology. Sun has provided their own JSF Reference Implementation while one can also, for example, find an open-source solution called Apache MyFaces [Kummel, 2010]. JSP specifies an advanced Dispatcher View inspired MVC platform, trying to take a component oriented approach to the development of the user interface - the presentation layer. The lifecycle of a JSF HTTP request consists of 6 phases as follows [Mann, 2004; Mukhar et al., 2005; Zambon and Sekler, 2007]:

1. *Restore view.* Based on the HTTP request the view that dispatched the request

[4]Java Enterprise Edition (Java EE) is a platform for server programming.

is identified. The view consists of a tree structure of components that consists of fields and, where appropriate, attached *validators* to these fields. Some fields also act as *event handlers*. The structure is built-up based on how the components looked like at the end of the previous request.

2. *Apply request values.* In this stage, the fields are updated based on new values received from the client. As they know about their previous state, they can notify of their change.

3. *Process validation.* In this phase, the attached validators are activated. If at least one component fails its validation, we continue to the last step - responding to the user with a view showing where the validation has failed.

4. *Update model values.* If the validation of components has succeeded, we can continue with this step. We know the data are valid in terms of their attached validators but we still need to contact the model to verify whether they are valid from a "business" perspective. For example a field with a user name might have a validator stating that one has to input at least 5 characters while from a business perspective the user name might have been used in the system already and needs to be unique (thus requiring a call to the persistence layer). Business Helper component is therefore employed to validate the request.

5. *Invoke application.* We now have validated data in a request, therefore a method associated with the form that invoked the whole request is called on the model component.

6. *Render response.* The model has finished processing and the result of the request has been received. A transfer of control is given to the view that directs the response to the user.

What is interesting about this approach is that it shows that the view in fact has its server and client part. Microsoft, as we will see later, has taken a similar component-based approach and used a Model-View-ViewModel pattern (Section 15.5 on page 132), an advancement of a code-behind approach.

Both of these approaches are specific with their focus on building *form based web applications* that have a helper component sitting between the view and the model that validates and collates data together. Great solution for form based application that do not have to call a model component if a simple form validation has failed. However, we end up with two types of validation happening. We would discourage from using this approach if one is expected to have a dynamic web application that does not rely heavily

on forms. Instead, as the validation of a form field is a business concern, we would like to see its validation happening in the model as then we only have to update one part of the application to have valid data in an application. Consider a case where one type of business logic is reused multiple times throughout an application, we know we will eventually always have to contact the associated model code, then why not include the validation logic there? Some approaches we see using Service to Worker MVC validate data as they are received in the controller, trying to be leaner than Dispatcher View but still validating data twice. We will demonstrate a preferred solution that relies on a domain model in a future chapter (Chapter 13 on page 119), complete with an expanded discussion of what it means to be a business model.

8.5 Merged Variants

In this section we discuss two (or three) sub-variants of the MVC pattern. They come from the desktop implementation of the pattern and merely document these approaches offering web based counterparts.

8.5.1 Model-Controller & View-Controller

Both model-controller and view-controller merge the functions of two tiers or layers in MVC. They are discussed in [*Mac OS X Reference Library*, n.d.] as "acceptable designs" for some applications.

First the *model-controller* approach is where the controller "owns" the model. We have found the use of this pattern in micro-frameworks on the web like Sinatra. In this framework one is expected to make calls to the database and do all the data processing in a controller itself without relying on external classes, models, representing some business logic.

In the *view-controller* approach the controller "owns" the views. This pattern on the web is quite similar to Model-View-Presenter (Chapter 10 on page 99) as there, like here, the Presenter (or controller) is empowered to manage the presentation. But we run at a risk of getting ahead of ourselves and so let us discuss the MVP pattern when the time is due.

8.5.2 Document-View

The Document-View variant of the MVC pattern could just as well be called *Model-View* to keep the wording similar to what we have presented already. In this approach one sacrifices the exchangeability of controllers [Buschmann et al., 1996] and their

function is merged with the view [Bachman et al., 2007]. This is an application of the *maintain semantic coherence* tactic as all events are handled in a coherent fashion, describes Bachman et al. [2007]. This tactic is a result of making sure that semantically related responsibilities are bundled together. Doing so binds responsibilities that are likely to change together [*ibid.*].

This approach is commonly found on the desktop but it should not be hard to apply this approach on the web. Web application frameworks make use of router - Front Controller components that decide which controller to call anyways. We could perhaps specify a portion of the application where models will be called directly instead of communicating with controllers.

8.6 MVC Evaluation

We begin our discussion by reiterating what Fowler [2006*a*] has said: "Different people reading about MVC in different places take different ideas from it and describe these as 'MVC"'. Kukola [2008] also finds that MVC is a misunderstood model and trying to understand it leads to confusions among the varied and conflicting adaptations of the pattern.

8.6.1 Observer vs Mediator

We have first observed that the function of the controller has changed over the years. The original (and implemented) MVC/80 uses controller to mediate a communication between the end user and the application. In many Web MVC implementations and Cocoa MVC, the controller is used in a *Mediator* pattern fashion while in "traditional" MVC/80 it is the *Observer* pattern that is used to separate the views between the models. Perhaps we should not be surprised at this rift as Fowler [2006*a*] argues that Observer behavior is hard to understand and debug because of its implicit behavior. To put it simply, you do not understand what is going on by looking at the code [Fowler, 2006*a*].

The move to a mediating controller also has the perhaps undesired effect, we would argue, that more code, more application logic is put into a controller. By looking at many Web MVC architectures we can say that the controllers get bigger and the application logic is no longer just in the models. This has an impact onto how we understand the pattern(s) discussed and what qualitative attribute(s) they exhibit. Models should, states Deacon [1995], know absolutely nothing about the connection to the outside world. But what if we no longer encapsulate the logic into the models?

Then they are disconnected and incomplete.

In the view of Deacon [1995], the better acronym for the architecture would be M_dM_aVC with the first "M" representing a *domain model* - logic that encapsulates the essence of the problem such as `Users`, `Files` etc. The other "M", Deacon [1995] goes on, represents an *application model*, that part of the application that knows about views, the presentation and communicates with it. However we already know that in Web MVC, the models do no such thing, they do not and can not[5] notify views of its changes. But Deacon [1995] at least shows us the distinction between the part of the application that communicates with the outside world (user) and the part that does not. Is, therefore, the controller filling the role of an application model in Web MVC?

For now at least, we should always mention whether a controller discussed is of the observing or the mediating type. The difference is profound and quoting MVC/80 in a paper about Web MVC[6] is misguided at best and unfortunately all too common.

Nevertheless we will attempt to discuss the MVC patterns through architectural primitives and quality attributes first, before discussing the issues and consequences of applying these patterns. Our focus will be on the Web MVC variant and how it differs from the "traditional" MVC/80 view.

8.6.2 Architectural Primitives

Kamal et al. [2008] have identified a number of behavioral abstractions in architectural patterns, they call them architectural primitives.

1. First, Kamal et al. [2008] identify a *Push-Pull* primitive in the MVC pattern commenting that the model pushes data to the view and the view can pull data from the model. In the Web MVC variant using the mediating controller, it is the controller that does the pushing and the pulling.

2. Next on the list is a *Virtual Callback* that involves a called component keeping a reference to the calling object [Kamal et al., 2008]. Kamal et al. [2008] state that the view and model components may communicate with each other through a callback operation, the Web MVC variant does no such thing. However, as we will discuss later, in some implementations of the MVC pattern on the web, multiple controllers are used in one call, calling each other recursively. However, this is out of the scope of the discussion presented here.

3. *Forward-Request* abstraction represents a situation where an internal "system" is decoupled from the external "objects" [Kamal et al., 2008]. We would state that

[5] Technically it would be possible with Comet.
[6] And we are calling it **Web** MVC at least.

the Front Controller or router commonly used in Web MVC represents an object that translates an external formal request in HTTP onto internal application logic calls.

4. *Command* abstraction identifies a case where calling an object will invoke a specific method [Kamal et al., 2008]. The definition does not make it clear, whether the Command abstraction is the same as the Command pattern [Gamma et al., 1995]. Kamal et al. [2008] state that this abstraction is found in MVC. As Controllers are invoked on their actions[7] by an external request, we find the same thing in Web MVC too.

5. *Asynchronous Message* abstraction or asynchronous communication is to be found in the Client-Server pattern scenario [Kamal et al., 2008]. As the Web MVC, especially its implementation in Ruby on Rails [Ruby et al., 2009] often relies on Server-Side Generation [Mahemoff, 2006] of asynchronous requests[8], we can find this abstraction in Web MVC too.

6. *Passive Element* is an abstraction where an object cannot invoke any operation [Kamal et al., 2008]. It can be found in MVC [Kamal et al., 2008] and it can be found in Web MVC in a form of an HTML page that contains no links or buttons or in a form of an email. Here, we should point out that although views are often graphical, they do not have to be [Deacon, 1995]. As was mentioned earlier, an email class represents a presentation as well. It is a way of presenting data to a user.

7. In a *Control* abstraction invoking a method on an object may involve a transfer of control and is found both in MVC/80 [Kamal et al., 2008] and Web MVC.

8.6.3 Modifiability Tactics

Modifiability tactics are approaches that are associated with certain quality attributes. It is a design decision that affects parameters based on coupling, cohesion or the ability to reduce steps in processing [Bachman et al., 2007]. We have mentioned them earlier, and not it is time to define them properly.

Increasing Cohesion & Reducing Coupling

According to Bachman et al. [2007] the model, containing the functional core of the application, uses a *maintain semantic coherence* and *use an encapsulation* tactic. How-

[7] Action is a common name for a method in a controller.
[8] An example is generating JavaScript code for the client view.

ever, as we have mentioned earlier, the mediating controller in Web MVC often takes on more responsibilities and thus not all "concepts" are located within a model. We will remind and reiterate this point later in an overall discussion.

Reducing Coupling

The controller works as an *intermediary* between the input device and the model and the view as an intermediary between the model and the output device [Bachman et al., 2007]. As we have seen a mediating controller acts as an intermediary between a model and a view. This increases coupling and we will later argue (Part IV on page 139) that a controller should therefore contain a minimum amount of code.

Run-time

Views can be opened and closed dynamically, being bound to data at different times [Bachman et al., 2007]. For us this presents a problem. Do we call the code in a view template that is going to be sent to the browser or do we maintain a presentation layer - a view in the browser? As model data can be bound to a view at different times from a controller and a view in a client's browser can be dynamically updated, using JavaScript for example, we would argue for the *use of run-time binding* too.

8.6.4 MVC Consequences and Issues

This part of the section discusses the consequences of applying the pattern in different scenarios, both good and bad, and the issues related. The points will be taken from Buschmann et al. [1996].

1. *Multiple views of the same model.* On a desktop a model is associated with one or more views, while if we use a mediating controller, then one calls multiple models to bind data to multiple views. Thus the mediating controller solution in Cocoa MVC/Web MVC is more flexible.

2. *Synchronized views.* This point relates to the fact that views work like observers of models a fact that *only holds true* to the desktop version of MVC. In Web MVC the advantage of having synchronized views does not exist as the user always initiates a request, be it synchronously or behind the scenes - asynchronously. The last option would give the user the "illusion" of being always synchronized with the data, by using a Distributed Events pattern [Mahemoff, 2006], that represents a client constantly checking for model updates. The only exception would be a Comet or "reverse Ajax" technology that is still in its infancy. In the

chapter on maintaining state (Chapter 3 on page 28) we have discussed that web servers do not maintain state to not become overloaded with numerous requests.

3. *Pluggable views and controllers.* Be it desktop or Web MVC, one can exchange view, controllers or models. Ideally, one would replace controllers and views behind a model easily in Web MVC, but we often see whole projects being started, re-written from scratch and this is symptomatic of a logic spread throughout models, controllers and sometimes, unfortunately, views too.

4. *Exchangeability of "look and feel".* A point made referring to the way a desktop application can change its user interface. In the Web MVC implementation the look & feel is often "set" using a cascading style sheet (CSS) or extensible stylesheet language transformations (XSLT) [Mahemoff, 2006]. A declaration to a CSS file is done in a head of an HTML page and this part of the presentation logic is usually encapsulated in a *layout* component that is set for a view. Thus the exchangeability holds true. An interesting approach is taken by SproutCore framework that extends the MVC model with Server Interface, Display and Responders components [*SproutCore Documentation*, n.d.]. The Display part specifically empowers the browser and associated JavaScript frameworks to repaint, resize and animate content. With increasing browser optimization and power of JavaScript frameworks a step in a right direction as it highlights code in a presentation layer that is often not described in web applications. What we mean is that a presentation behavior is often dismissed and left out from various flow diagrams and focus rests on the server components.

5. *Framework potential.* The popularity of the pattern on desktop and on the web attest to that [Singh et al., 2002]. A Wikipedia entry listing and comparing web application frameworks[9] lists 55 out of 67 as using the Model-View-Controller pattern. The pattern is popular perhaps due to its simple three layer split that, in the web domain, does not impose any rules as to where application functionality should find its place and a fact that it has been selected as the pattern for application architecture in Alur et al. [2001]. But as no rules and restrictions are imposed, this also means that junior developers can develop a working system now that will be less reusable later. An often repeated example is including too much business logic in controllers where it does not belong.

6. *Potential for excessive number of updates.* As the views observe a model in

[9] `http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks`, accessed 17th August 2010.

MVC/80 this is a problematic fact. In Web MVC with no such behavior a single request is responded with a single reply. A potential for excessive updates happens in Dispatcher View MVC when users asynchronously refresh a content they are viewing as each time the framework has to understand the request, where it is authorized and build all the components required anew (Section 8.4.2 on page 83). Therefore, the simpler Service to Worker strategy is preferred.

7. *Intimate connection between view and controller.* Buschmann et al. [1996] have argued that these two components are closely related and they would unlikely be used without each other. In Web MVC a controller does not have to call a model and does not even have to call a view. We should perhaps note that one can return data to a client without using a view. What we are arguing is that one does not have to reply to a client apart from a standard "200 OK" response. For example, one can increasingly find a Periodic Refresh pattern [Mahemoff, 2006] which represents a recurring asynchronous call, simply to keep a client/browser session alive. In such a case the controller will (has to) be involved. Using a controller without a view in Web MVC or Cocoa MVC is problematic considering the controller acts as a mediator between the view and the model, especially in Web MVC where the processing starts with a controller. A solution, already mentioned, would be to setup the framework/application router to directly call views from a repository skipping controller processing. But such a solution is unlikely to find prevalence as it would minimize the execution time of a framework only a little with network transfer still being the Achilles [Homer, 1998] heel.

8. *Close coupling of views and controllers to a model.* Buschmann et al. [1996] mentions that both views and controllers make calls to a model so a change to its interface will break the code. In Web MVC it is usually possible to call a model directly[10], but the framework designers can discourage such behavior by using a custom templating language in the view that has no concept of calling a function or a class from a template, such is a case of Sapphire Framework of SilverStripe CMS (see Figure 8.9 on the following page for an example of a custom templating language). A different problem is how one binds model data to a view. Such a case will be better illustrated in our future discussion of the Model-View-Presenter (Chapter 10 on page 99) and Model Adapter (Chapter 11 on page 110) patterns. MVC is not flexible enough to address this concern on its own.

[10]For example in PHP, one can just include a class file and then use it.

Figure 8.9: SilverStripe template.

```
...
<div id="studies" class="span-40 last">
    <% control Page(studies) %>
        <% control Children %>
            <div class="span-19 <% if Odd %>last<% end_if %> study">
                $CalloutImg
                <p>$Content.LimitWordCountXML</p>
                <a class="orange-text" href="$Link">Click here to
                    find out more ></a>
            </div>
            <% if Odd %>
                <div class="span-2"> </div>
            <% end_if %>
        <% end_control %>
    <% end_control %>
</div>
...
```

9. *Inevitability of change to view and controller when porting.* The beauty of web frameworks is that most of their application code is being written in one language[11]. Therefore, one cannot port from a PHP environment to a .NET environment. At the same time, all browsers "should" handle the HTTP protocol and HTML language the same way. They do not do so and one solves this problem by using a custom CSS code/hacks or JavaScript frameworks on the desktop as introduced in point #4 in this account. All in all, usually a tweak is required to the view code if there is a problem with the application's presentation in a particular browser.

10. *Observer behavior is hard to understand and debug* [Fowler, 2006*a*] is a point we have mentioned earlier. Luckily the mediating controller in Cocoa MVC/Web MVC approach is more linear and thus easier to understand.

11. Holub [1999] would suggest that the MVC approach is not maintainable at all. It cannot be called object-oriented because "there is just too much data flowing around" [*ibid.*]. This point will be better addressed in a chapter on Naked Objects architecture (Chapter 13 on page 119) that will compare Naked Objects with MVC. By the time we will have presented the reader with project conclusions (Part IV on page 139), we will actually see that one can create a maintainable system that addresses Holub's concern.

[11] However a haXe language attempts to become a "universal language" being able to translate into Flash, PHP, C++ and JavaScript.

8.7 Discussion of MVC

The original author and inventor of MVC, Trygve Reenskaug, spoke in a foreword of a thesis by Pawson [2004] on Naked Objects architecture disputing the trend MVC has taken. He advanced that the purpose of MVC was to bridge the gap between the "user's mind and the computer-held data" and that the focus on the user's mind has been forgotten. Dispatcher View based frameworks (Section 8.4.2 on page 83) have a focus on the presentation, making it easy to work with and validate forms while Service to Worker based frameworks like Rails (Section 8.4.1 on page 81) make it easy to work with the databases. Interestingly, neither of these has been the original purpose of MVC that was trying to automatically produce a default graphical interface for any business object [*ibid.*].

As business concerns should be part of models, we discourage from the use of Dispatcher View frameworks like JSF that use Business Helpers, unless one relies heavily on forms. This is an example of concerns being split between classes. On the other hand, Service to Worker MVC can be applauded for its simplicity, building an application from three components. However, a generic MVC model is too restrictive in terms of presenting data to a user. In particular, the 1-to-1 relation between controller actions and views seems too constricting. Some MVC frameworks have addressed this issue and evolved into a pattern that is going to be addressed next, the Model-View-Presenter.

Also, while consisting of only three components we often see too much logic being handled by the controller, logic that should find its place in the model. On the other hand, Holub [1999] saw MVC as having two components too many. A view we do not share as having a presentation logic in a class handling business layer would result in breaking of the *single responsibility principle*, a concept we will revisit in the future when discussing Naked Objects architecture (Chapter 13 on page 119).

Chapter 9

Data Transfer Object

In the previous chapter we have introduced the landmark pattern in desktop GUI and web architecture, Model-View-Controller. We understood its history, motivation, variants and described two strategies used in respect with this pattern that differentiate between the transfer of control between the three components in MVC. These were the Dispatcher View and Service to Worker strategies. For web architecture, we have argued for the simpler Service to Worker, after all a doubled communication from a view to the model in MVC/80 and Dispatcher View increases the load on a system and makes the flow of events harder to understand.

The reason for introducing the Data Transfer Object (DTO) is to illustrate how data from a model can be accessed inside the view without calling the model directly thus exposing the business logic of the application inside the presentation layer. This concept will illustrate a *use an intermediary* tactic that decreases coupling in a system, a beneficial approach in an agile environment of web application development.

9.1 Problem

In the Service to Worker Web MVC strategy we need to solve the question as to how does the view communicate with the model while maintaining an overall decoupling of the components. While in the MVC/80 the view accesses the model directly after being notified of a change, in Web MVC this is harder to accomplish as the model does not "live" on the server and has to be built each time during a new request. The Dispatcher View solves this problem by first calling the model from a view to instantiate the application logic, then the view calls the business layer again to fetch new data, but this leads to more redundant operations and a slowing down of the system.

9.2 Solution

A fitting solution is to actively make use of the aspect of Service to Worker MVC, that is the focus of control on the controller component that directs the flow in an application, much like in desktop Cocoa MVC. Thus we can reverse the communication between the view and the model and instead of calling the business layer actively, we can use an injection of data. We call this paradigm the *inversion of control* [Martin, 1997].

Using this solution means that the view has all the needed data when executed and this implies it does not have to call the model layer again. In order to transfer or inject data to the view we can use the DTO [Alur et al., 2001; Zandstra, 2007].

9.2.1 Example in Zend Framework

The Zend Framework [Allen et al., 2008] uses a `Zend_view` object to encapsulate the server view while the controller component is defined by the `Zend_controller_Action` class. It is a framework that uses the Service to Worker MVC paradigm. In the example (Figure 9.1), we see the function `assign()` that accepts data from the controller and injects them to the view for further processing.

Figure 9.1: Assigning data to the view through a DTO object in Zend Framework.

```
<?php

public class IndexController extends Zend_controller_Action {
    ...
    public function indexAction {
        // create a DTO object based on data from a model
        $dto = new customDTO($model->getArticle());
        // create a view
        $view = new Zend_view();
        // assign model data to the view
        $view->assign('article', $dto);
        // call the view to render itself
        $view->render('article.phtml');
    }

}
```

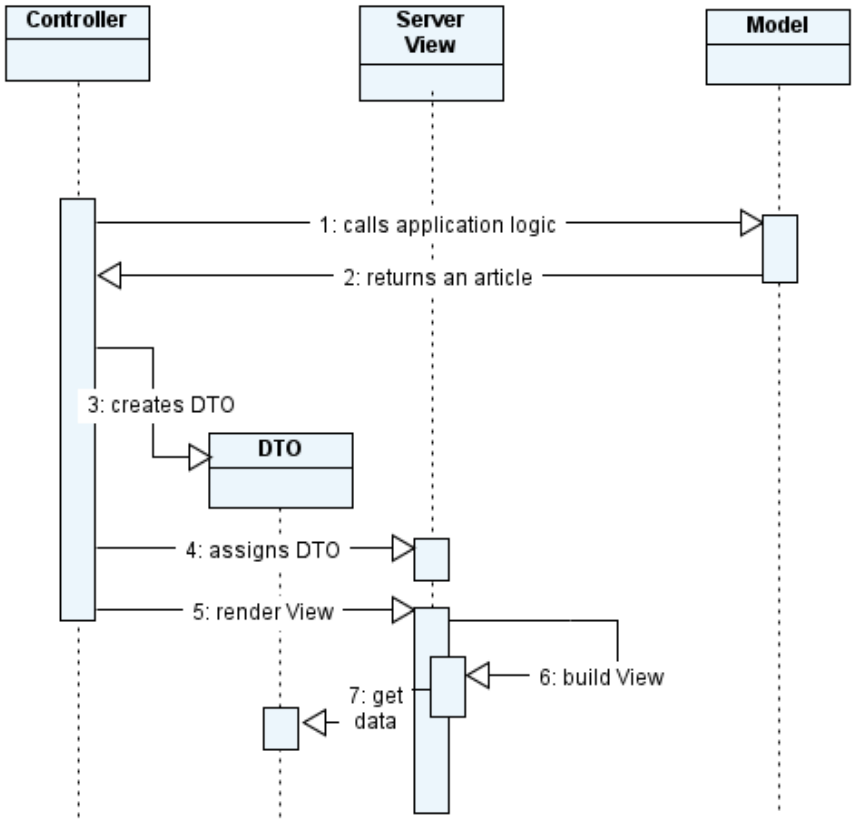
In our example, we are calling the `articles.phtml` template file that will traverse the data assigned. In Figure 9.2, we see the data from a DTO object being traversed.

Figure 9.2: Traversing data from a DTO in a view.

```
...
<h1><?php echo $this->article->getTitle(); ?></h1>
<p><?php echo $this->article->getText(100); ?></p>
...
```

What the example nicely illustrates is that the DTO object can work as a helper component doing presentation level processing of data. In the example we see the `getText()` method being called with a parameter `100`. This represents us requesting a text of the article we are retrieving, but only displaying 100 characters. This is, for example, used in blog article listings that need to show only a partial of the full text. It is a presentation concern and thus instead of creating numerous getters[1] in the model each customizing the output of data, the DTO works as a helper component marshaling data together for various presentation.

Figure 9.3: Data Transfer Object in Zend.



The Figure 9.3 shows a flow diagram of events using the DTO pattern:

1. The controller calls the application logic (model) asking for an article.
2. Model responds with an object, perhaps an row from a database table, that represents the article entry.
3. Controller instantiates a new, custom, DTO.
4. DTO is assigned to the view.

[1]Getter is a common name for a public method that is used to get data from an object. By the same token a "setter" is used to set new data.

5. Controller asks the view to render itself.

6. While traversing the page template, the view calls the DTO for data.

7. The DTO has methods in place that return the article data.

9.3 Discussion of DTO

In this chapter we have presented a helper component that lives somewhere between the presentation and a business layer. We use it to delegate presentation responsibility from the model layer through the concept of inversion of control.

This pattern is an extension of a simple injection of data in web frameworks. For example, in Zend or Fari, one can assign data as they come from the models to the view, instead of wrapping them all in one object. Such a solution is beneficial in simple cases, however, as the application increases in complexity, an in-between/code behind components are used to wrap around certain aspects of the presentation. One might be poised to move a logic, such as getting a custom sized article text into the model or even a presentation, however neither of these result in concerns that are decoupled. In the former example, one has to ask why a business layer needs to know how big a text has to be on a blog home page, while in the latter, the presentation template is usually designed to be as simple as possible. For example, in Sapphire MVC Framework of SilverStripe CMS [Schommer and Broschart, 2009], one has to use a custom, limited, templating logic in the views (Figure 8.9 on page 93). The framework and subsequently the CMS is designed to be easy to work with for designers that are not necessarily proficient in web development. Therefore the system uses a custom language that is parsed by the framework. One can, therefore, not use a PHP language and has to move the logic elsewhere. The framework, for each model object generates a set of accessors that work much like DTOs. For example, if one defines a field in the model that holds a text, the designer will be able to display the text and customize its length based on the number of words or characters, all behind the scenes.

In the next chapter, we will look at the Model-View-Presenter pattern that will further enhance the way one works with the presentation aspects of a web application. We will also revisit the DTO pattern and extend its discussion, showing how one can maintain a separation of concerns in an application using an Adapter pattern.

Chapter 10

Model-View-Presenter

In the previous chapter we have introduced the Data Transfer Object pattern that serves as a repository of data for a view to use. We understood its usefulness in relation to the Service to Worker strategy of the Model-View-Controller paradigm. The approach allowed us to marshal data from a model together, to inverse the control, for the view. This has the effect of decoupling concerns between the presentation and business layers.

The reason we need to look at the Model-View-Presenter pattern, discusses herein, is that it will further show a systematic decoupling of concerns in a web application. In particular, we will see the focus being put on a Presenter component that will allow for easier testing and streamlined processing of data in a system.

The Model-View-Presenter pattern (MVP) does not render itself well to a classic "problem - solution" description. The pattern, we would argue, exists in web architecture more often than some would believe, misattributing it as a Model-View-Controller approach. A quick, and rather informal albeit informative, search on Google Scholar for the amount of research on MVP differs to MVC in a 1:80 ratio. The way MVP differs from MVC is by using two patterns - Supervising Controller and Passive View that facilitate control much like Observer pattern is used in MVC/80 pattern. The Presenter, a special component variant in role to a controller varies by the way it coordinates changes in the domain model [Fowler, 2006*a*]. But first, let us look at the pattern's history.

10.1 Taligent Model-View-Presenter

If you name your company Taligent, a portmanteau of Talent and Intelligent, you know something good will come out if it. Such was the case with Mike Potel who has stepped

back and thought about a new unified programming model that would be used across the company's major object-oriented language environments [Potel, 1996].

Potel [1996] discusses that there are two questions a programmer has to address in GUI architectures. First there is *data management* an umbrella term for the "How do I manage my data?" question, second is *user interface* that answers "How does the user interact with my data?" [Potel, 1996], reminding that even though we use a user interface, the presentation of the application need not be graphical.

Data Management

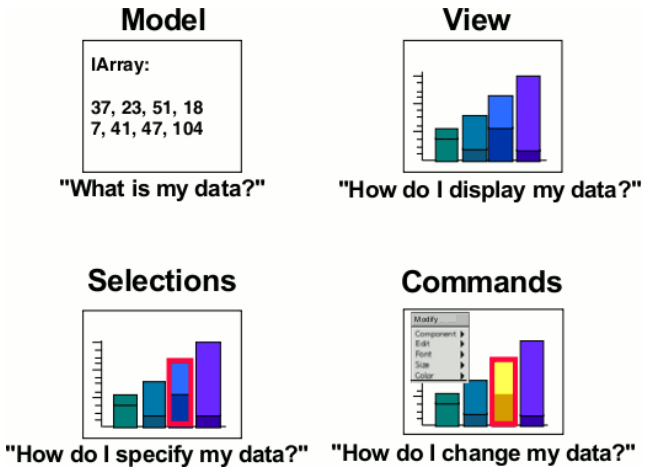
The question "How do I manage my data?" is further broken down into questions that are solved with counterpart components (see Figure 10.1) as follows:

1. *What is my data?* This is the responsibility of the model component, same as in MVC, that handles the data and the core business logic/functionality.

2. *How do I display my data?* Handled by a view component that we know from MVC is responsible for the displaying of the data on a screen or "displaying" the data by sending them in an e-mail, for example.

3. *How do I specify my data?* Selections components handle the task of selecting which parts of the model data we will operate upon in the view. In MVC/80 this task is handled in the model, but in Web MVC is often done in the controller layer - let us remember this fact for later.

Figure 10.1: Taligent asking "How do I manage my data?", reproduced from Potel [1996].



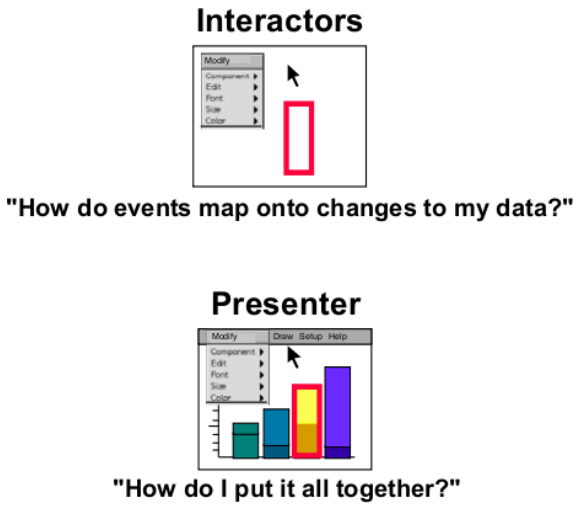
User Interface

These User Interface questions (see Figure 10.2) explore how does the user interact with the data as follows:

1. *How do events map onto changes to my data?* This is the link that connects the user events such as a mouse click onto the models, in Taligent solved using Interactor components.

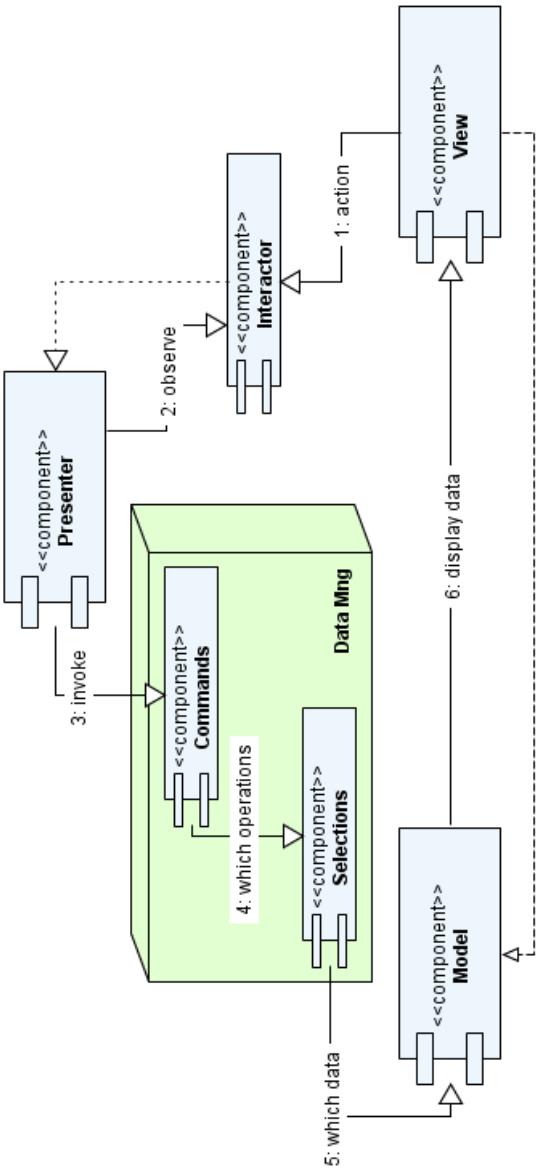
2. *How do I put it all together?* A broad question with a simple answer - use a Presenter. This component creates and maintains all the components introduced herein and above, it directs the workflow within the application.

Figure 10.2: Taligent asking "How does the user interact with my data?", reproduced from Potel [1996].



A diagram showing the collaboration of components in Taligent MVP is shown in Figure 10.3 on the following page.

Figure 10.3: Taligent MVP.



What we see is a triangle of communication between the different components, with each side of the triangle focusing on one aspect, concern, of the system. The system processes an event as follows:

1. The view component is where all events are launched. As we are discussing a desktop environment pattern, we do not differentiate between the client and the server view.

2. The focus is put on a Presenter component that observes an Interactor where all events are received. The view and Interactor form a UI layer.

3. The Presenter is a component that decides what commands to invoke on the Data

Management layer, the left side of the triangle.

4. The Commands component knows of actions that change data.
5. While the Selections component decides which portion of the data to edit and/or update.
6. The model defines the data themselves.
7. The base of the triangle represents a notification framework between the view and the model. The view gets updated by the model.

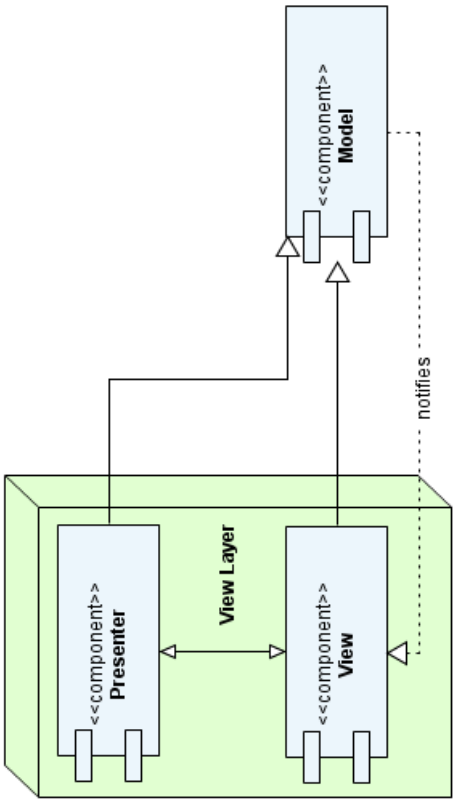
In relation to the MVC/80 pattern, Taligent MVP focuses more on how one works and selects the data we work with. Especially the data management part is split into 3 components, each representing the different aspects of updating a model. The Presenter also gains on responsibility, deciding which models to update and observing the state of the view. In MVC/80, the controller was more passive and it was the view that was deciding which models to update. The vision for Taligent MVC in a client-server environment was to leave the left side of the triangle, this means the Commands, Selections and model on the server with the view and Interactors living on the client [Potel, 1996]. The Presenter would then live both on the server and the client and maintain the flow. We have seen a controller component mediating the flow already, it was with Cocoa MVC (Section 8.3.3 on page 78).

10.2 Dolphin Smalltalk Model-View-Presenter

Much like with MVC, Smalltalk has served as a springboard for an architectural pattern. It improves on MVC/80 and simplifies Taligent MVP. In MVC/80 it is the application model that is being observed by the views while the model has no way of contacting the view. But, as Bower and McGlashan [2000] discuss, the model sometimes needs to access the interface and such a thing is discouraged. Fowler [2006*a*] thus describes a step towards a more "presentation aware" application model in VisualWorks that is a partial development of a *presentation model*.

The Dolphin Smalltalk team have further enhanced (Figure 10.4 on the following page) the functionality of the presenter component and left out Interactors, Commands and Selections from the Taligent MVP system [Bower and McGlashan, 2000] advancing their initial MVC version.

Figure 10.4: Dolphin Smalltalk MVP.



The diagram (Figure 10.4) shows the Presenter and the view both being part of the "view layer" with the former being a mediating part between the lattern and the model. In this Smalltalk version, the Presenter works much like an application model would in MVC/80 but the component is directly associated with a view mediating updates with the model on behalf of the view [Bower and McGlashan, 2000]. Its primary purpose is to update the model. Bower and McGlashan [2000] nicely declutter the motivation between the controller in MVC/80 and Presenter in Dolphin Smalltalk MVP, where the former primarily intercepts user input and updates the model as its byproduct, the situation in the latter is *reversed*.

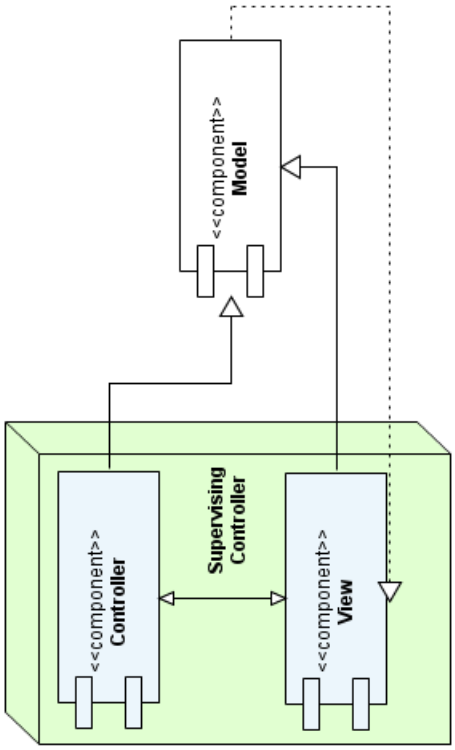
10.3 Generic Model-View-Presenter

Much like in our discussion of MVC discussing facilitating (some would say design) patterns (Section 8.2.1 on page 73), Fowler [2006*a*] identifies two "work in progress" patterns that distill the behavior of MVP in general. Let us discuss them now.

10.3.1 Supervising Controller and Passive View patterns

In the Supervising Controller (aka Presenter) pattern the presentation logic is split between the view and the Presenter with the latter encompassing more *complex behavior*. This, Fowler [2006*c*] informs, then allows the Presenter to be easily tested without getting involved with widgets and the GUI while testing. Ideally, we should be able to test the presentation logic without using layout components. This is what Supervising Controller, Presenter, achieves.

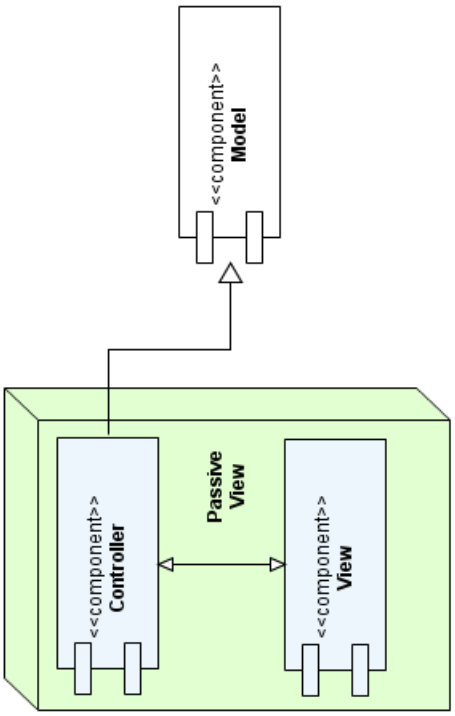
Figure 10.5: Supervising controller.



The other pattern identified in the MVP architecture is the Passive View (aka Passive Screen aka Presenter First) pattern. It is closely linked to the previous one with the only difference being that the views maintain no link with the model [Fowler, 2006*b*,*a*]. In effect the Presenter works as a mediator between model and a view containing all of the presentation logic (Figure 10.6 on the next page). Alles et al. [2006] introduced a Presenter First pattern in their work which is, in essence, a Passive View pattern.

We have seen this before in Cocoa MVC (Section 8.3.3 on page 78) or MVC++ (Section 8.3.4 on page 80). There the controller is a mediating part, directing the flow between the model and the view, but in MVP the Presenter is empowered with more application logic to make the presentation layer more testable.

Figure 10.6: Passive view.



10.4 Web Model-View-Presenter

In the previous sections we have seen the evolution of the MVC pattern into MVP. Mainly, the Presenter component has received more application logic to make the presentation more testable and served as a mediating part between the view and the model. The controller is no longer passively passing messages from the view to the model.

We have presented the desktop version(s) of MVP patterns and what sub-patterns it consists of. The important question is how it relates to Web MVP.

1. We have already stated that in Dolphin Smalltalk the views delegate requests to the Presenter when model needs to be involved. If we do not count a presentation logic on the client in form of a JavaScript then the views cannot really be "smart" and in Web MVP should contain minimum of application logic [Bower and McGlashan, 2000].

2. The previous point is extended by the fact that the application should be *testable*, thus moving complex logic from the views into a Presenter (a Supervising Controller) [Fowler, 2006*c*].

3. A Presenter directs the *workflow* within the application [Potel, 1996].

4. From the description of Dolphin Smalltalk MVP the Presenter mediates updates the model on behalf of the view [Bower and McGlashan, 2000].

5. In Taligent MVP it was the Selection components that would select which parts of the model data we will operate upon in the view [Potel, 1996]. In Web MVC this task is often done in the controller layer.

Given these five points, what we are trying to say is that:

> *Given there are no formal definitions of Web MVC or Web MVP, the former is closer to the Model-View-Presenter pattern than to the Model-View-Controller pattern that it stems from. To put it simply, a vast amount of "MVC frameworks" on the web are actually MVP.*

This holds true for the Service to Worker MVC frameworks that contain more complex presentation logic in the controller.

10.4.1 Nette Model-View-Presenter

This situation has been realized by David Grudl a developer of a mature web framework - Nette. The documentation for the project [*Nette Documentation*, n.d.] sees the different components of the pattern ideally having the following responsibilities:

1. The *model* is limited to operations with data and/or a database.

2. The logic contained in *views* is limited to simple if/else statements.

3. The logic in the *Presenter* builds up an overall component tree and queries the model.

This description is similar to how the Cocoa MVC pattern works. We see an ideal level of separation of concerns. The Presenter is discussed as a component building the overall component tree and making queries to the model. The view template is limited to traversing the data that the Presenter provides.

In the example of Sauter et al. [2005], the authors have created a Web MVC framework that aims to serve multiple devices/clients. It uses "extensions" to the pattern arguing that in its original conception both view and controller components would have to be rewritten for each device. Instead, we can use the MVP pattern that gives us the flexibility to have a different presentation logic for each device. The internals of such approach will be discussed in the architecture example part (Part IV on page 139).

10.5 Discussion of MVP

In this chapter we have introduced the Model-View-Presenter pattern. The idea behind MVP is to have the controller/Presenter *mediate updates* between the view and the model. Furthermore, the Presenter is empowered to contain more *complex presentation logic*, making it more testable. Such solutions can be found in Web MVC frameworks which leads us to a conclusion that many MVC frameworks are in fact MVP.

Consider a case outlined in Figure 10.7 of a badly designed view that contains more complex presentation logic. In this case, the order details are calculated in the view as they are different from those found in the business logic of a model. However, had we used a Presenter, all of this functionality could have been abstracted away and be easily testable through, for example, unit testing [Martin, 2008].

Figure 10.7: Shop template with complex logic.

```
...
<tr>
    <th class="align_right" colspan="4">Total cost of products in
        basket</th>
    <td class="align_right" colspan="2"><span>&pound;<?php echo
       money_format('%.2n', round(($this->cartTotal - $this->
       deliveryCharge) * $this->vat, 2)); ?></span></td>       </
       tr>
<tr>
    <th class="align_right" colspan="4">Postage and packing</th>
    <td class="align_right" colspan="2"><span>&pound;<?php echo
       money_format('%.2n', round($this->deliveryCharge * $this->
       vat, 2)); ?></span></td>
</tr>
<tr>
    <th class="align_right" colspan="4">ORDER TOTAL</th>
    <td class="align_right" colspan="2"><span>&pound;<?php echo
       money_format('%.2n', round($this->cartTotal * $this->vat,
       2)); ?></span></td>
</tr>
<tr>
    <th class="align_right" colspan="6"><p>Your purchases have
       raised <span>&pound;<?php echo money_format('%.2n', round
       (($this->cartTotal-$this->deliveryCharge) * $this->vat *
       $this->commission, 2)); ?></span> for <?php echo $this->
       school ?></p></th>
</tr>
...
```

What we like about MVP is that it extends on the Service to Worker MVC pattern allowing the developer to write more complex presentation logic in the Presenter component. This is useful in the web environment as applications do not persist between calls and an application tree of components has to be built each time anew. This means

there is a place for a component, a Presenter, to be *empowered* and decide which views and which models to call. In Part IV on page 139 we will return to this feature on an example and show how useful it is to have a Presenter deciding which view to render based on the model data.

While we tout Web MVP as a useful pattern, the lack of documentation does not show us exactly how to solve "hard" problems that do not neatly delineate themselves into the three components, concerns. Therefore, the next chapter will discuss an approach that cuts across concerns in such a way that the solution is both *elegant and reusable*.

Chapter 11

Model Adapter

In the previous chapter we have introduced the MVP pattern. Its web form builds on the Service to Worker MVC pattern, further allowing the developer to put more presentation logic into the Presenter (controller in MVC). This is beneficial as on the web, a user event can be responded and tailored to the target client or device.

We need to look at the Model Adapter pattern, as a further extension of the Data Transfer Object pattern, to see how one can manipulate data in a Presenter, while still preserving a separation of concerns.

11.1 Problem

In a well separated Web MVC/MVP solution, one should not make calls from the view to the model so that the two do not end up coupled and so that the model does not know too much about the presentation layer. But, sooner or later there will be an issue where we need to *paginate* a selection of data in our application. In Taligent MVP this UI task is handled by Selectors. In Dolphin MVP this piece of logic is handled by the Presenter. In Web MVP we would like to achieve the same in a Presenter without cluttering the logic in the view that should be kept simple.

11.2 Solution

We can use a component that serves as an adapter of the model, providing a presentation-only related accessors to data. This component, a Model Adapter, can then be associated with and processed by the view. This approach is similar to how the Data Transfer Object, discussed earlier, was organized. But with Model Adapter, one can work and call methods on the object instead of using it for a simple data wrapping. It is an application of the Adapter pattern [Gamma et al., 1995] tailored to the needs of

working with business objects and databases.

11.2.1 Dibi DataSource Example

Dibi [*Dibi API Documentation*, n.d.] is a database abstraction layer for PHP5. It manages calls to the persistent layer - the database and also contains a concept called "DataSource". Let us illustrate how this concept works on an example (Figure 11.1).

Figure 11.1: Dibi DataSource as a Model Adapter.

```
<?php

// the presenter
class AnyPresenter extends presenter {
    ...
    public function renderDefault($page) {
        $articles = $this->model->getArticles();
        // extra condition
        $articles->where('category = %i', $this->category);
        // bind articles to the template
        $this->template->articles = $articles;
    }
}

// the model
class model {
    function getArticles() {
        return dibi::dataSource('SELECT ... FROM table WHERE ...'
            );
    }
}

// the view
<h1>Article listing</h1>
<?php foreach ($articles->select(array('url', 'name', 'id'))->
   orderBy('date') as $article): ?>
    <h2><?php echo $article->name; ?></h2>
    <p><a href="<?php echo $article->url; ?>"><?php echo $article
       ->id; ?></a></p>
<?php endforeach; ?>
```

In the article retrieval example we see three components corresponding to the MVP pattern, the Presenter, the model, and the view template. A request is received on the Presenter, we only show the rendering method, `renderDefault()`. In it we ask the model for an article listing. The model responds with a DataSource that turns a query into an `iterable` and `countable` collection, the model does not care how we display the articles, it only maintains a connection to the database through an SQL query. The Presenter modifies the returned result by specifying which category we want to display. Again, the model should not care which category we need to display, it is

a *presentation concern*. The query is further modified in the view by specifying the order of the result set. This is a presentation concerns but is not really complex and as such should be relegated to the view. If the user needed more power as to how to order the results, we could move this condition to the Presenter as well. Just consider that perhaps on a different device one could have data paginated in a different form. Having such logic in a model would result in us having to modify business logic of the application each time we are changing how the data should look on a client.

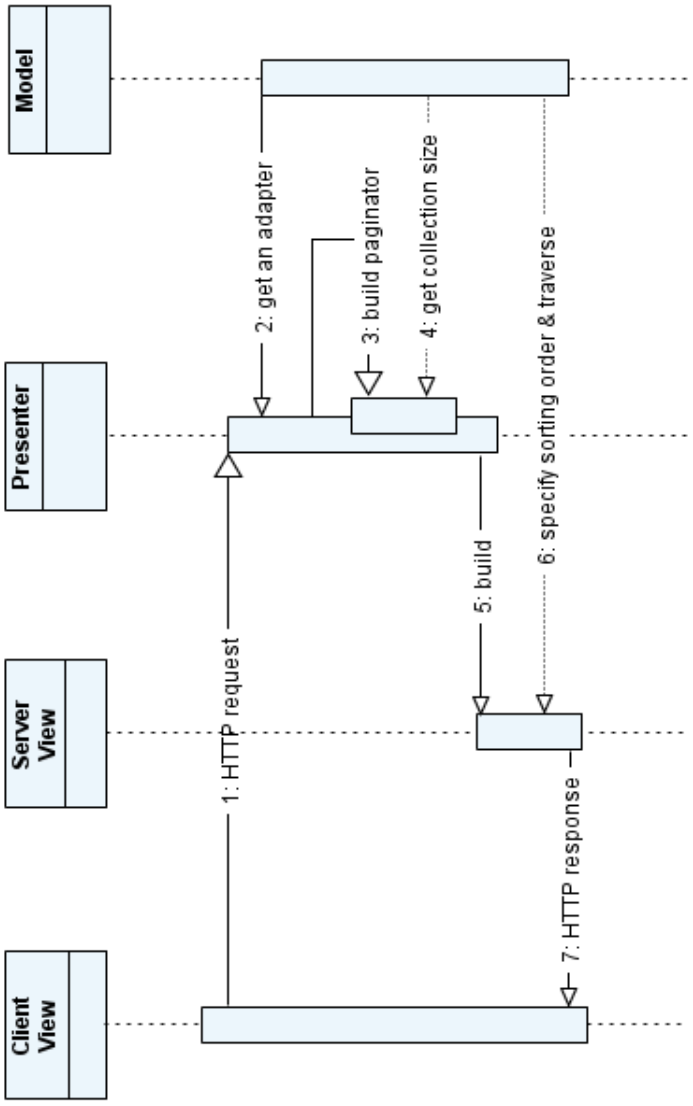
11.3 Discussion of Model Adapter

The Dibi DataSource shows the power that one can gain from a well designed component. The developer using the Dibi framework does not have to learn any new syntax or create new classes corresponding to Model Adapters. Instead, we are provided with raw data, that can be modified only by conditions that are related to the presentation concern. In our example, we might have passed the model two arguments specifying the category and order direction, so let us show a more complex example that should convince the reader that a combination of well designed Adapters can achieve better separation of concerns while seeing the benefit of Presenter, that is the ability to encapsulate more complex presentation behavior and make it testable.

In Figure 11.2 on the following page we see a flow diagram between components in a problem where one needs to paginate a data set. The steps are as follows:

1. An HTTP request is received on the Presenter that decides which model to invoke.
2. We request a Model Adapter that represents a *collection of items* we want to paginate over.
3. The Presenter builds a paginator. A paginator is purely a presentation concern. The model does not care whether the data it provides is displayed on 90 pages or just one. By building a paginator we see a benefit in having a Presenter. One can test the business logic of the application, without testing the paginator itself, which is not, in itself, a business concern.
4. While building the paginator we need to know the total size of the collection. Instead of making another query to the model to ask for the size of the items or more wrongly requesting all the items in the collection and counting them in the code, we simply call a `count()` function on the DataSource that translates the request into an SQL query to the database. It seems we are working with a

Figure 11.2: DiBi Model Adapter flow of events in MVP.



collection of data, but the adapter works as an *interface* for our queries making them only when needed.

5. We build the view, asking it to render itself.

6. The view again makes use of the fact that DataSource is an iterable component and translates the call for traversing of data into an SQL query into the database.

7. We respond to the user with a built view.

What this approach illustrates is that there is a need for more complex presentation behavior in web applications, and we are all for elegant solutions. But mainly, it shows how one can indirectly make calls to the database from the view through a finalized interface of a Model Adapter. This component acts as if we were already working with an array of objects, but *translates our calls* into SQL queries behind the scenes.

The generic MVC pattern does not solve such solutions but we can use components such as Model Adapters or Data Transfer Objects to maintain a clean separation of concerns with each layer only working with concerns over data each layer needs to know about.

The original MVP invention, Taligent MVP, was solving the issues illustrated herein through Selections and Commands components but was too complex to use. *Interestingly, by using Adapters we accomplish the same behavior without increasing the complexity of the overall system.* Thus we will recommend their use in Part IV on page 139 of this work as an integral part of a web application architecture. The only downside in using the Model Adapter is designing it appropriately. After all, one needs to translate method calls into SQL queries. But as we expect this component to be finalized in a target framework, the *end developer* does not need to know this components' internals.

Chapter 12

Django Model-Template-View

In the previous chapter we have discussed the Model Adapter pattern and how it is useful when used with a Presenter pattern found in MVP. Especially, we draw on the conclusion that one needs to access a database, **indirectly**, from Presenter and view components. This was underscored by the example of a data paginator that needs to find out the size of a collection of data in the Presenter; data that are ordered and traversed only in the view component.

We will now discuss the Model-Template-View (MTV) pattern as it shows the use of generic framework components when displaying and working with data. Thus it represents an alternative to the Presenter pattern, providing many presentation functions that a developer would have to code by hand.

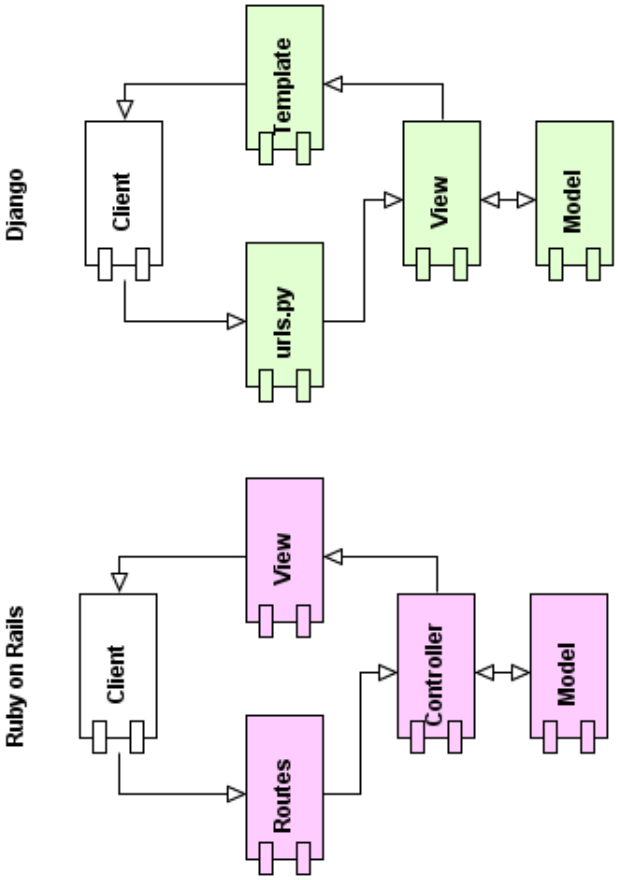
The MTV approach presented here is not dissimilar from what Service to Worker MVC frameworks look like (Section 8.4.1 on page 81), however the naming convention attests to the way its authors view the responsibilities of the three web tiers. They are:

1. The *model* layer is still in charge of data and persistence. The data get validated, processed, accessed and related to other data.

2. The *template* component is part of a presentation layer and decides how something should be displayed on a web page or other type of a document [Holovaty and Kaplan-Moss, 2009].

3. To make things a bit more confusing the *view* component is referred to in Holovaty and Kaplan-Moss [2009] as a business logic layer. This layer bridges the models and templates by calling models to fetch data and then displaying them in templates. The authors themselves note that this layer is commonly referred to as a *controller layer*, however in Django's interpretation of MVC the view de-

Figure 12.1: Ruby on Rails vs Django structure.



scribes the data that get presented and decides which to present to the user and not just necessarily how the data look.

12.1 Generic Views

The framework is often discussed as an alternative to Ruby on Rails MVC approach [Askins and Green, 2006; Linowes, 2007]. Figure 12.1 shows that Rails and Django are very similar in their structure.

One of the benefits [Askins and Green, 2006] that make it easier to build web applications in Django compared to Rails is their focus on automating the presentation of data through "generic views" [Holovaty and Kaplan-Moss, 2009; Bennett, 2009]. As views in Django encapsulate and represent the data that are presented to the users, these built-in and extensible components solve common patterns found in data presentation such as: providing an interface of a list or detail of data, archive pages, and a set of view for creating, editing, and deleting objects [Linowes, 2007; Holovaty and Kaplan-Moss, 2009]. We will discuss the generation of user interfaces in more detail in the next chapter, for now, let us discuss another interesting aspect of Django, its

Figure 12.2: Data validation in Django models.

```
from django.db import models

class Category(models.model):
    title = models.CharField(max_length=250)
    slug = models.SlugField(unique=True)
    description = models.TextField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
```

Figure 12.3: User authentication decorator, reproduced from [Holovaty and Kaplan-Moss, 2009].

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='redirect_to')
def my_view(request):
    ...
```

models.

12.2 Data Validation

What Django (and other frameworks such as Rails, Fari) provides is an Object Relational Mapper (ORM). A Django model is the "single, definitive source of data about your data" [*ibid.*] containing the definition of fields and behaviors of the data as usually, a model maps to a database table. Defining attributes in a model means linking them to table columns. The example code in Figure 12.2 shows a couple of fields in a `Category` model. What we see is also a couple of rules that relate fields to other tables or decide how a field should be long.

12.3 Discussion of Django MTV

In this section we have introduced the MTV pattern that is similar in composition to MVC, but as the focus is on the presentation layer - the view in MTV, we would conclude that it is closer to the MVP pattern. This is underscored by the introduction of "generic views" that encapsulate common logic that one might need to code in a web application. This makes the development of a system more agile and streamlined. For example a view method that displays some data can be integrated with a user authentication system by using a `@login_required` "decorator" (Figure 12.3).

While such an approach makes it easier to build applications and include common functionality "out of the box", one has to learn the decorators and helpers provided by

the framework, approaches specific to Django (or other framework for that matter). We would also complain that using custom framework components has the disadvantage of behaviors being "lost" in pre-built components. Take the example of a use-case that needs to map to the code. How does one show a functionality in the code that is hidden away in framework libraries?

What we like about Django is its focus on defining and validating table fields in the models. For example, instead of validating form fields in a Business Helper component like a Dispatcher View MVC would do or checking input size of a field in a Presenter, leaving all this behavior in a model is highly beneficial. This means that one could test the business logic while using the model alone, without having to test a business logic that has "spilled" into a Presenter. One can then access the models, for example, from a command line, without having to rely on the Presenter or view components. This also means that if we need to use the same model from two different Presenters, one does not have to write an input validation code again in each of the Presenters. We will reiterate this point in the concluding parts of this dissertation.

Chapter 13

Naked Objects Architecture

In the previous chapter we have introduced Django's Model-Template-View approach. In particular we understood it as a way of providing common user interface functionality that works on top of business models. The models, often linked to database tables, specify the rules of the table fields - they represent an ORM layer. The approach discussed here, Naked Objects architecture, is an approach where the whole of user interface is generated automatically from business rules and objects. In this chapter, we will overview some principles of Domain-Driven Design (DDD), that Naked Objects implements. The reason for discussing this approach is that we see some aspects of DDD as beneficial and at the same time we will be able to discuss ways of structuring controllers in our proposed architecture. We will learn from a bad example.

Object-Oriented Interfaces

Naked Objects architecture is related to the concept of object-oriented interfaces. This approach, Nielsen [1993] discusses, in comparison with function-oriented interfaces, focuses on the operations on data and information objects that are represented graphically on a screen. Trygve Reenskaug in Pawson [2004] sees object-oriented interfaces as fulfilling the original motivation of MVC, that is, the generation of a user interface from objects and letting the user operate on them.

13.1 Use-Case Controller

Pawson [2004] begins his thesis by repeating the motivation behind MVC where the desire was to support multiple visual representations of objects, and multiple client platforms. Views and/or controllers were not supposed to have embodied any business behaviors [*ibid.*]. The reason behind this is that such an approach reduces the ability

to test an application. Furthermore, it then means that business objects cannot be used without their controllers and views.

The reverse is to embody business behavior in controllers which leads to a Use-Case Controller, a procedure by another name [Fowler, 2002], goes on Pawson [2004]. And if one implements behaviors in controllers, they are missing from the model layer.

Pawson [2004]; Holub [1999] argue that this goes against the motivation of OOP where objects need to be "behaviorally complete" meaning that all the functionality associated with an entity is encapsulated in it. Such a separation, of procedure and data, in object-oriented system leads to excessive coupling hindering agility of a system, thus making it harder to easily modify the system in a future [Firesmith, 1996].

13.1.1 Overall Flow ≠ Behavior

However, we do not share the view of Pawson [2004] that Use-Case Controllers are procedures that encapsulate too much behavior. Take the definition of a Use-Case Controller from EuroPlop conference where the pattern was discussed [Aguiar et al., 2001]. We find that the pattern manages a flow of events in a use-case with the controller providing "a uniform way of coordinating actor events, user interfaces and system services." We believe that in the domain of web applications, this nicely fits with the Presenter component that manages more complex presentation behavior. The Presenter can act as a use-case coordinator managing a *flow of events*, thus one can get a glimpse of the overall behavior of the system by looking at such a component without going through each and every business model.

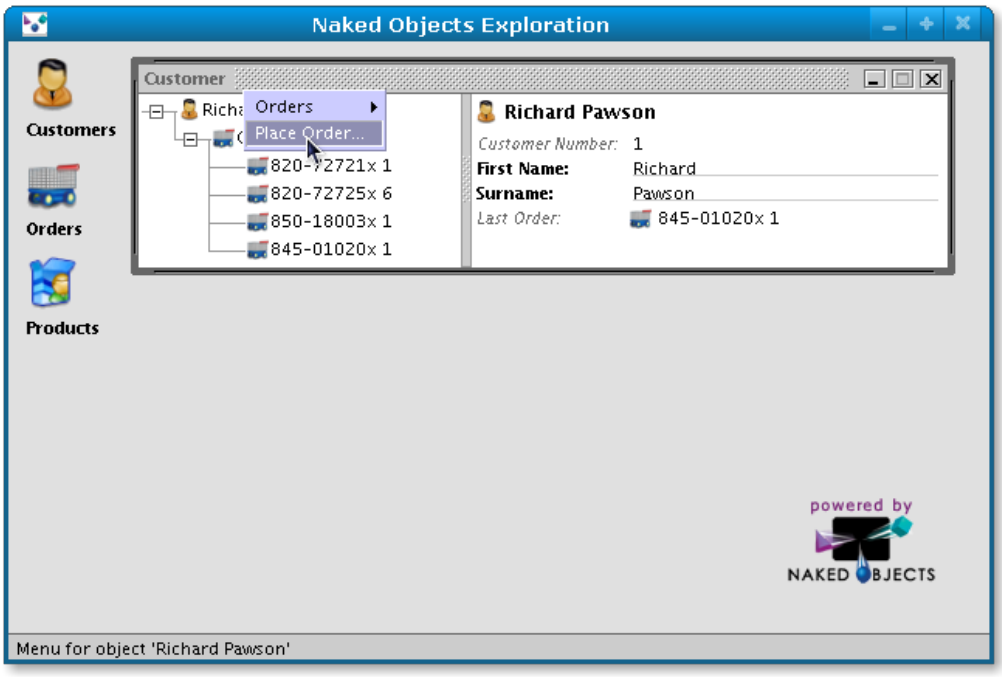
Transaction Script

What is perhaps Pawson [2004] worried about instead is that the controller will end up working as a Transaction Script pattern [Fowler, 2002]. In this pattern, a whole business behavior and computation is described in a single method. **That** would be a procedure by another name.

13.2 Domain model

Pawson [2004] takes the extreme of encapsulating all business logic and object coordination in model definitions, generating user interfaces purely from them. Thus he is revisiting the concept of Puerta et al. [1998]. The application is defined solely in terms of the model layer that the user acts upon through run-time generated [Läufer, 2008] views and controllers [Pawson, 2004]. See figure 13.1 on the following page for

Figure 13.1: Naked Objects, reproduced from Pawson [2004].



an example of a generated interface.

In order to understand development purely in terms of a domain model we need to look at the concepts behind a Domain-Driven Design (DDD) Naked Objects makes use of.

Domain-Driven Design

In DDD a user story that describes a functionality of a system is not described in terms of the user interface and its elements, but in terms of the underlying behavior that domain objects needs to exhibit. Fried et al. [2009] would take an opposing view suggesting that ultimately, the end user will evaluate the usefulness of an application in terms of the interface. Therefore, as Naked Objects automatically generate the whole "user experience" automatically, we will need to discuss the cases where such an approach is beneficial and where it is not.

Context and Use-Case Presenter

DDD defines for each domain model a bounded context, that is how users relate to the concepts defined in the model [Evans, 2004]. We believe that this is where a "Use-Case Presenter" can be useful, in setting a *context for the business logic that happens in models*. We do not want to define business logic in the Presenter and leave the models anemic. Instead we want it to take care of the interaction with the user, finding the

objects we need to work with and properly initializing them.

13.3 Discussion of Naked Objects

Among the benefits of using Naked Objects is the fact that domain models are behaviorally-complete. However, the objects are also limited in their usefulness. The reason why patterns such as MVC became so popular is that they differentiate between the different concerns of a web application. In Naked Objects there is no space for a presentation concern, if we wanted to add one, we would break the *single responsibility principle* (Section 5.2.1 on page 52).

The Naked Objects approach cannot be understood as useful in the type of web application we build as it does not provide us with enough flexibility in terms of presentation. The author himself is not, by his own admission, a usability expert and automatically generated interfaces are not useful in cases where the user of the system is a process-follower. Constantine [2002]; Pawson [2004]; Läufer [2008] comment that Naked Objects are beneficial in directly tracing user requirements into the implementation, however we see the solution as too rigid. Instead we have overviewed the Use-Case Controller as an alternative and saw that it has its validity when used properly. In particular, we have identified that *setting objects in a context* could be exactly what controller/Presenter would be useful for. We crystallize this idea further in the concluding sections.

For now, we would suggest using Django MTV (Chapter 12 on page 115) and its ability to generate admin[1] interfaces while giving the developer the power to modify them instead of using Naked Objects, end users will appreciate it.

[1] Administrator section of a web application often having the capacity to operate on business models, i.e. editing articles in a CMS.

Chapter 14

Data, Context and Interaction

In the previous chapter we have challenged Naked Objects arguing that it does not offer enough flexibility in terms of presentation and is best replaced by the Django MTV that offers helpful patterns that solve common presentation concerns. But we have appreciated the Domain-Driven Design approach Naked Objects are applying and in particular highlighted controller as a component that sets context for use-cases and subsequently business behaviors that reside in models.

This chapter is dedicated to a pattern invented by the original mind behind MVC, Trygve Reenskaug. We need to look at this approach as it represents the *latest* architectural paradigm to have sprung up. More importantly, it will show us again the need for setting up components in a context.

14.1 Problem

Reenskaug and Coplien [2009] discuss the the problems that current MVC architectures have, offering their own solution as an "advancement". They discuss MVC as an approach that separates the representation of information from user interaction, as the user is important in their view, they dub a term Model-View-Controller-User [1].

Reenskaug and Coplien [2009] describe the goal of this MVCU model to sustain an illusion that "the computer memory is an extension of the end user memory - that computer data reflect end user cognitive model". Therefore we see a move from presentation generation present in Naked Objects to an idea of an architecture that easily maps computer memory (also business objects - models) to the user's memory - and view.

[1] At this point it might be interesting to note that in their view, the controller is coordinating Views and Models, therefore perhaps indirectly dismissing MVC/80. Remember that mediating controller only appeared in MVC++ and Cocoa MVC (Section 8.3.3 on page 78).

However Reenskaug and Coplien [2009] see a problem with this paradigm asking where to put, for example, a functionality such as a spell-checker in a word processor in this model? The authors go on to argue that some of the solutions lead to poor cohesion of the objects while others increase the coupling between objects [*ibid.*].

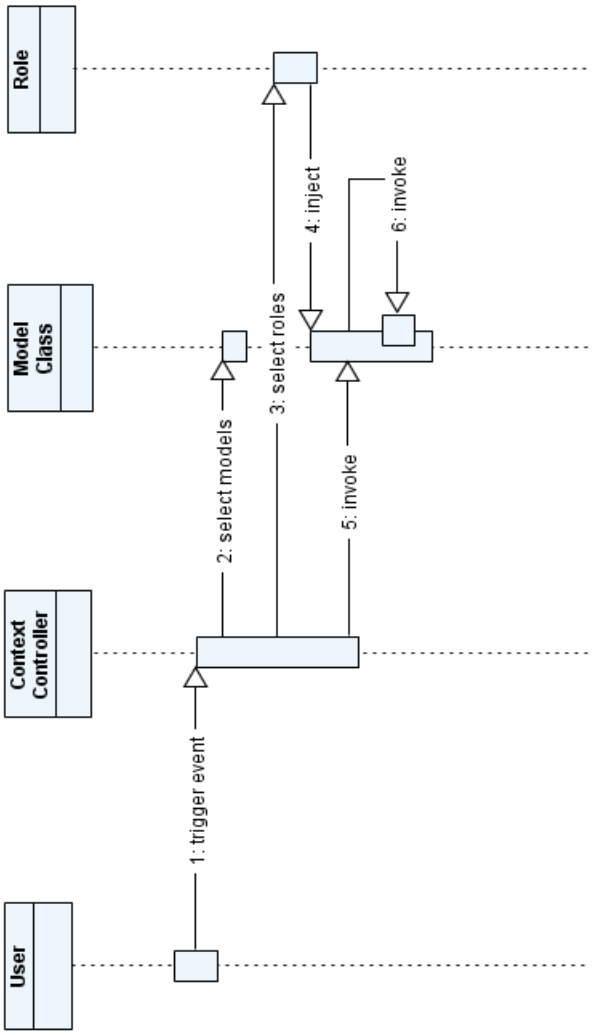
14.2 Solution

In MVP we would have suggested putting as much of business functionality into models, but Reenskaug [2009]; Reenskaug and Coplien [2009] take a different view instead extending the MVC model by three more components that support interactions with the model:

1. *Data*, that live in the domain objects that are rooted in domain classes.
2. *Context* that brings live objects into their positions in a scenario, giving context to objects through roles.
3. *Interaction*, that describes end-user algorithms in terms of the roles, both of which can be found in end users' heads.

Ultimately the goal is to decouple controller from the model and to communicate with it through Context and Interaction instead. Let us discuss the DCI paradigm through a flow diagram (Figure 14.1 on the following page):

Figure 14.1: DCI architecture flow diagram.



1. Much like in a "ordinary" MVC architecture, an event from a user is received on the controller (called a *Context*).

2. The controller selects the *models* that we will need to work while processing the request.

3. The controller further selects the *roles* that the objects will need to play during the process.

4. Roles are *injected*, using perhaps aspects[2], into the models. The idea is to represent only data in the models while the roles encapsulate behavior.

5. Now that the controller has injected behavior into data, it invokes the first method on a model.

[2]Refer back to Aspect-oriented Programming in Section 5.2 on page 52.

6. The models continue to interact with each other.

14.3 Discussion of DCI

Coplien [2009], a co-author behind DCI, describes the reason for using aspects as a technology injecting roles into classes as "interesting" functionality is bound to cut across objects. While Naked Objects were trying to encapsulate everything there were to be known about objects, both business and domain logic in Domain Models, DCI *embraces the opposite*.

Use-Case Controller

Evaluating the controller Contexts we see how each of these represent an "algorithm" for a use-case interaction. This is in line with our suggestion to use Presenter actions as high-level overviews of the functionality of the system without actually specifying a behavior, resulting in a Use-Case Controller [Aguiar et al., 2001].

The DCI framework is an approach at capturing algorithms that translate use-cases into code so that one can reason about the state of a system and not just state of one or the other object [Coplien, 2009]. This is exactly what we argued for in the previous chapter (Section 13.1 on page 119), having a way to see what behavior an application exhibits without traversing all relevant models. In DCI then, for each use-case there is a Context that casts roles to objects.

Presentation

From this we can again derive that while the MVC was aiming to bridge the gap between the user's mental model and that of the system, DCI attempts to translate business problems into code. However, a more complete consideration is then not given to presentation concerns. A PHP DCI framework called Wax represents the presentation of objects in roles. But do we then not end up with roles that either contain model behavior and roles that output information? This, in our mind, breaks the *single responsibility principle* (Chapter 5.2.1 on page 52) that such a component should have. We have already argued for the use of a Presenter as it allows one to encapsulate more complex presentation behavior. How do roles communicate with each other to combine data from multiple sources into one presentation, one response, on the web?

Traits

The use of traits has not become a commonplace and we are left wondering whether it is not because, much like in Observer behavior "there is too much stuff flying around to understand what is happening". The approach adds more components instead of simplifying.

The DCI paradigm forces us to use role objects and aspects, but why not use a Role Object pattern [Bäumer et al., 1997] that has been described 13 years ago, and use it only *when needed*? From Bäumer et al. [1997]:

> "Adapt an object to different client's needs through transparently attached role objects, each one representing roles the object has to play in that client's context. The object manages its role set dynamically. By representing roles as individual objects, different contexts are kept separate and system configuration is simplified."

What we like about DCI is that it forces us again to think about the role of a Presenter as a Use-Case Controller setting up objects in a certain context, a view that we are trying to advance. We believe that we have enough information from the patterns and approaches presented to take interesting aspects from each of them and form a simple architecture solution that clearly describes what are the different functionalities/concerns of the different tiers in a web application.

What we are left to do is to briefly discuss patterns reviewed that do not represent interesting solutions to problems identified and we can move to Part IV on page 139 that shows how our architectural solution translates business requirements into code in an elegant way.

Chapter 15

Other Reviewed Approaches

15.1 Presentation Abstraction Control

Presentation Abstraction Control (PAC) addresses an issue in user interface development where various functionalities need their own UI elements while still need the ability to communicate with each other [Avgeriou and Zdun, 2005].

The solution proposed by Coutaz [1987] is to provide a "tree-like" hierarchy of agents [Buschmann et al., 1996]. Each agent is responsible for its functionalities representing a part of the interface with the ones higher up in the structure having more responsibility [Buschmann et al., 1996; Avgeriou and Zdun, 2005]. The agents collaborate with each other through a control layer that also intermediates between presentation and abstraction layers with the former taking care of user interface while the latter represents the application data and logic that modify it [Avgeriou and Zdun, 2005]. When the agents communicate with each other, they apply the Composite Message pattern [Gamma et al., 1995] which allows the interfaces of each agent to be independent not representing or marshaling a specific data format etc. [Buschmann et al., 1996].

Drupal is a prime example of PAC architecture in current web systems. Each part of the interface is split into blocks that, after processing the request push the result onto a template canvas.

We believe that the PAC needlessly increases the complexity of a resulting system. In a web environment where the whole application needs to be loaded from ground up, we do not achieve a benefit in first splitting the logic into a multitude of agents and then marshaling the data back together for one unified response. It is interesting to note that as some Ajax web applications when first loaded initialize a layout and a frame for the application in the browser with subsequent asynchronous requests only making transfers to/from the server, we achieve the same benefits as PAC. The first synchronous request can be thought of as representing a "main" agent that sets up

communication and initializes a "page" with the other ones working as "data transfer" agents.

15.2 Hierarchical Model-View-Controller

The Hierarchical Model-View-Controller (HMVC) is an invention by Cai et al. [2000], presented in a JavaWorld magazine. It famously refers to the authors being unaware that they have re-invented a wheel 13 years old now, the PAC. HMVC, still based on MVC is a less strict version of the PAC in that the presentation layer (view) and the data layer (model) are allowed to talk to each other directly, much like in MVC/80 (Section 8.3.2 on page 76).

As HMVC is a variant on PAC the same, needlessly complex, conclusion applies here as well. Where we see an application for HMVC/PAC is in frameworks that want to exhibit looser level of coupling between components. For example the TYPO3 framework uses a "Tree of Plugins" approach where each plugin encompassing a functionality in a system represents a separate MVC triad [*TYPO3 wiki.Resource*, n.d.]. Thus the system is very loosely coupled, as the developer of a plugin cannot modify the behavior of the larger framework.

Open Reuse

This brings us back to the issue of open reuse (Section 5.7.3 on page 62). Reinhartz-Berger et al. [2002] would claim that frameworks do not exhibit open reuse, the ability to develop partially specified components and refining them in the target application. Then, it would seem, some framework architects design plugins to extend system behavior. To which we would repeat that one can design framework components through the Template Method pattern that allows one to define a general execution template of methods that themselves are specified in a subclass [Gamma et al., 1995]. A user authentication is a prime example, where we might want to include a user authentication class in a framework but the developer might need to tweak its functionality. Thus, instead of using plugins to inject code and intercept code in various points of the execution, we would inherit from the Template Method class *prescribing an execution order of events* in a library and define our own implementation. After all the goal of a framework is to work as a skeleton and prescribe some order yet be extensible as much as possible [Wake, 1998]. This has been with success used in Fari Framework, with Figure 15.1 on the next page showing how a Template Method definition looks like.

Figure 15.1: Template Method definition for user authentication.

```
/**
 * Authenticate user.
 * @param string $username
 * @param string $password
 * @param string $token optional form token
 * @return authenticationFail() or authenticationSuccess()
 */
public function authenticate($username, $password, $token=
    NULL) {
    // escape input
    $this->prepareUsername($username);
    $this->preparePassword($password);

    // if credentials provided and token is valid
    if (isset($username, $password) && ($this->validateToken(
        $token))) {
        // select a matching row from a resource
        if ($this->matchUser()) {
            // save user into a session
            $_SESSION[self::SESSION_CREDENTIALS_STORAGE .
                APP_SALT] = $this->credentialsString();
            // success
            return $this->authenticateSuccess();
        }
    }
    // fail
    return $this->authenticateFail();
}
```

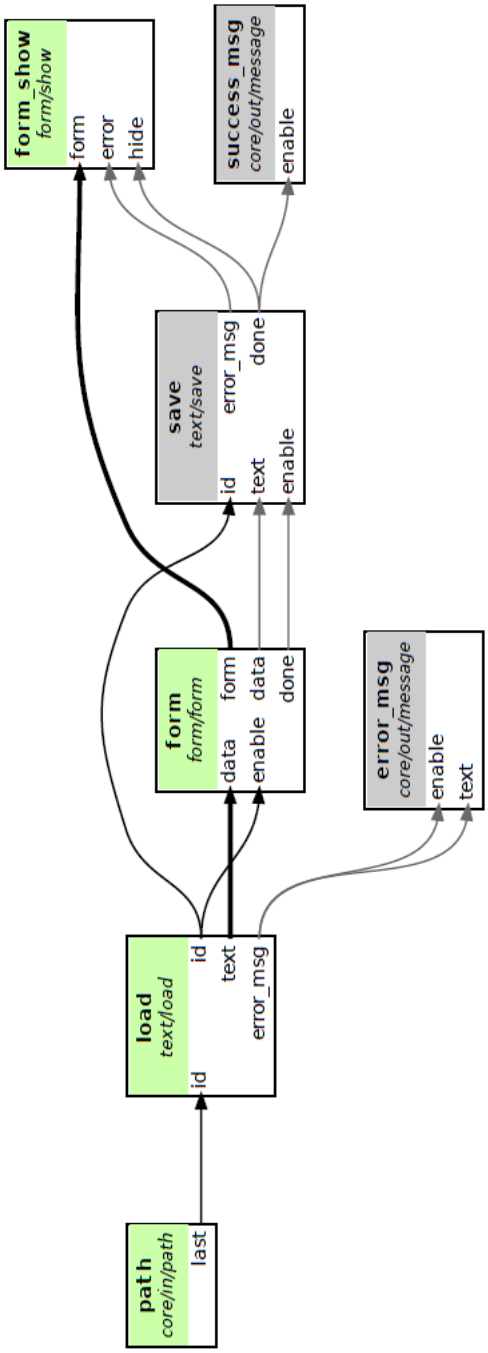
15.3 Dynamic Pipeline

The motivation for Dynamic Pipeline comes from UNIX shells and the Pipes & Filters pattern [Fowler, 2002]. This approach, developed by Kufner [2010], forces the developer to clearly define interfaces between components that are executed in an order like filters in a pipeline.

The advantage of such a solution is that one can easily test each component and they can be rearranged and easily swapped. This is a common benefit when one clearly defines interfaces, an application of *interface segregation principle* (Section 5.2.1 on page 52). As the flow of events is clearly given in a pipeline, the approach can produce a chart of its own components, showing each step in execution. Some filters work like adders and some work as decision trees deciding between components to call next (Figure 15.2 on the next page).

However, the solution does not clearly delineate between the different concerns an application has to handle. Furthermore, there is an overhead in the amount of data constantly being passed among the components. On the desktop, the benefit of having

Figure 15.2: Pipeline form and text edit in a database, reproduced from Kufner [2010].



Pipes & Filters is that a stream of data is processed as we often deal with large amounts of information that need to be processed quickly. On the web, however, such forces are less common and applications rather benefit from clear separation of concerns and good architecture design.

15.4 Chiron-2 (C2)

C2 architecture [Taylor et al., 1995] has, to our knowledge, not been used in web applications. It is similar to a Dynamic Pipeline, achieving the same benefits of a flexible component-based development. However, each component can have its own thread of control that may run in a distributed environment. While the solution allows one to communicate between different systems, the conclusions that apply to a Dynamic Pipeline apply to C2 as well.

15.5 Model-View-ViewModel

ViewModel is a specific application of a Presentation Model pattern [Fowler, 2004] used by Microsoft frameworks. It is a concept evolved from code-behinds where a piece of logic updating and loading data to the views would be placed [Smith, 2009]. We can compare a Presenter in MVP with a ViewModel in MVVM. While the former creates the view and fills it with data, the latter is bound in a reverse order [*ibid.*].

The three components found in MVVM (Figure 15.3 on the following page) are as follows:

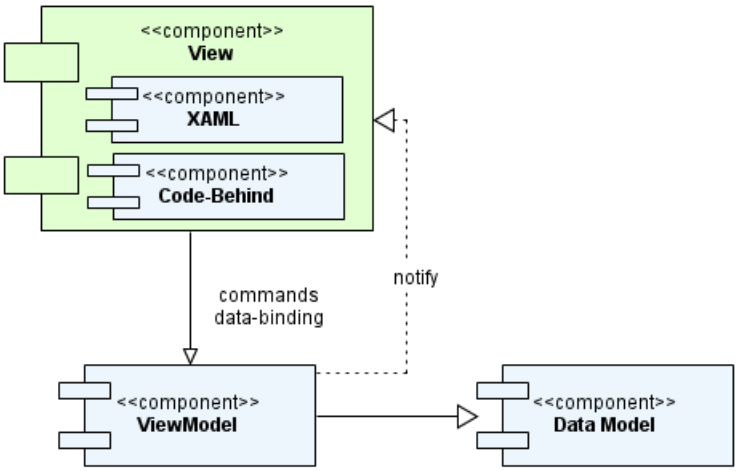
1. *Data Model*, a component analogous to a data/business layer found in MVC or MVP.
2. *ViewModel* represents a layer that manages the state of the application by being bound to the view and its events. It provides data to the presentation layer in an appropriate form.
3. *View* is the presentation layer that communicates with the user.

As MVP is a generic approach while MVVM is tied to a specific platform, we choose to not discuss the solution further.

15.6 Model Pipe View Controller

This approach, discussed at Cunningham's C2 Wiki extends the MVC model by another component, a Pipe. This component, [Atherton, 2008] suggests, would solve the

Figure 15.3: ViewModel.



problem as to how to sort data provided by a model in the view. As we will see, our architecture example will use a Model Adapter (Chapter 11 on page 110) to solve the same problem.

Chapter 16

Summary of Part III, Patterns, Methods & Architectures

In this part of the dissertation we have overviewed and commented on a number of patterns and approaches. We have chosen these artefacts of an architecture as they represent fundamental organization schema of software and subsequently of the whole frameworks they are part of.

Model-View-Controller

We have discussed the desktop variants of the MVC pattern to differentiate between its numerous versions and to better explain an often misunderstood model. This helps us to see two different strategies used in Web MVC applications - Dispatcher View (Figure 8.8 on page 84) and Service to Worker (Figure 8.7 on page 82). In the web domain, we conclude that Dispatcher View needlessly makes multiple calls to a business logic as it often features (in a case of JSF) a Business Helper component that means model validations are done twice. Such a solution is only advantageous in applications heavily relying on forms and even then it is not clear that easier development of the service means easier updateability in the future.

The MVC pattern, as we have commented, can be applauded for its simplicity in splitting concerns of a system into three parts. However true MVC controllers are too constrictive in their 1-1 relation to views and do not address complex presentation behavior. Some frameworks have addressed this issue and evolved into a Model-View-Presenter pattern.

Data Transfer Object

Data Transfer Object was discussed as a pattern resolving communication between view and a model. From MVC we understand the advantage of using the controller as a mediating component between these parts. Thus, DTO serves as a wrapper for model data, injected by the controller into the view, for it to use. Furthermore, DTO is seen as a simple data object that does not encapsulate any model behavior.

Model-View-Presenter

The MVP pattern is an MVC variant that uses a Supervising Controller, a Presenter. This component is designed to encompass more complex presentation behavior. Such a solution makes the end software more easily testable as views can be relegated to handle only simple data traversal. The Presenter builds up an overall application tree and decides which components to call, it directs the flow. This is beneficial in the web environment as the whole application needs to be instantiated again with each request, thus there is no need for any persistent parts to maintain links between each other, as would happen on the desktop. However, we have not yet seen how hard problems in concerns separation are solved by the MVC/MVP pattern, until describing the next approach, the Model Adapter.

Model Adapter

We use the Model Adapter pattern as an example of why using a Presenter is better than using a generic controller in a web architecture. We have considered a case where one needs to display a listing of articles. The "problem" was to sort the listing and filter it by a category. The solution was to use a Dibi DataSource, an applied Model Adapter. In this solution we see how a model is oblivious to sorting and filtering concerns and only returns a link to a persistence layer. The Presenter modifies the Adapter by adding a filtering condition by a category, while the view adds an ordering rule. We have confirmed the usefulness of such an approach by demonstrating its use when presenting data through a paginator. On the desktop, data are fetched from a database as needed, while on the web one has to build an appropriate component that will support pagination and filtering of data, while making as few calls to a database as possible. This is what Model Adapter in our example does.

These examples also show the need for a more complex behavior to be encapsulated in a Presenter, rather than using a controller, thus providing further confirmation of our support of the MVP pattern.

Django Model-Template-View

The MTV pattern was discussed as a variant of MVC where the view component is similar to an MVC controller, but determines *what* data are presented and *how* they look. This fits well with the view of the Django framework that offers many helper components - generic views, that take care of common presentation functionality. The problem we have with this approach is that such a solution is not generic to multiple platforms and one has to use Django and its way of building software. Functionality and behaviors are being lost in pre-built components by the framework.

Where the approach and framework shines is in its data validation. Much like in the case of Ruby on Rails, Django offers an ORM with rules in models specifying and validating database table columns. Such an approach is beneficial as one can use models without their controllers and views and still validate the data. Also, if one were to move validation of data into controllers, such functionality would have to be repeated with each controller and could possibly even be forgotten by the developer, leaving the business functionality in models "vulnerable".

Naked Objects Architecture

While discussing the Django framework and its ability to generate user interfaces and solve common functionality through generic views, we have looked at the Naked Objects approach. With it we have noted the difference between object-oriented and function-oriented interfaces. The former has the user working as a "problem solver" while the latter has her working as a process follower.

Naked Objects is an application of Domain-Driven Design ideas and can be applauded for increased clarity in models, the layer of business rules. We have looked at the Use-Case Controller approach and unlike Pawson [2004], we see a real benefit in having a controller/Presenter describe the overall use-case behavior. This, we argue, would not result in behavior existing in the controller where it should not belong; instead of behavior, it describes *flow*. Instead of visiting each and every model in the system to understand what it does, one can see the overall structure from the Use-Case Controller. This, however, means that one has to be clear in the model's definition and relegate all the business behavior to those components. Understanding the tenets of DDD, we see that the Presenter can direct the flow of the application and argue it should be responsible for setting up the *context* for the whole system.

Ultimately, we suggest using Django MTV in cases where one needs to put an interface together fast, instead of using Naked Objects that lacks in flexibility of the presentation layer.

Data, Context and Interaction

Our discussion ended with the latest development from Trygve Reenskaug, the inventor of MVC.

Data, Context and Interaction (DCI) supports our view in having a controller/Presenter component set up the application, the models in particular, in a certain context. But unlike DCI where one injects roles into behavior-less classes using traits, we argue for the use of a Role Object pattern instead. This means that one uses the concept of roles only when needed. After all, a composition using a Factory or Façade is an alternative that gives different interpretations of objects too. One then does not have to rely on traits, a still uncommon feature of programming languages as well.

The problem with DCI is that while it focuses on the transfer of functionality from requirements to code through roles, it does not handle the presentation concerns appropriately. We have shown that presentation behavior in DCI is encapsulated in roles, which breaks the *single responsibility principle* in our view, as roles should encapsulate behavior. We do not share the view that presentation is a business concern, a behavior of an object.

Overview

In order to see the beneficial aspects of the different approaches presented, we have constructed Table 16.1 on the following page that lists the points we are defending and that will be brought together, and extended in our final part of the dissertation to form an architecture that translates business problems into code.

Table 16.1: Overview of dis-/advantageous feature of patterns and architectures.

Approach	Disadvantages	Advantages
Dispatcher View Model-View-Controller	Multiple calls to models. Can result in broken dependencies when pulling data from the presentation layer.	Useful in heavily form-based applications.
Service to Worker Model-View-Controller	Controllers not flexible in terms of presentation logic.	Simple architecture split.
Data Transfer Object	Not complex enough, only carries data.	Commonly used in MVC and very simple to use.
Model-View-Presenter	Not described well enough, needs helper patterns.	More complex presentation logic in Presenters.
Model Adapter	Needs to be well designed by the architect.	Fits well into the MVP paradigm. Acts as a collection of data while making calls to a database behind the scenes.
Django Model-Template View	Behaviors "lost" in pre-built components.	Fast development of user interface. Data validation in models.
Naked Objects Architecture	Limited to a niche of applications. Not flexible at all in terms of presentation.	Domain-Driven Design and focus on well defined models that encapsulate all of the system behaviors.
Data, Context Interaction	Badly solved presentation. Forced use of roles. Use of traits.	Use of contexts to set up the environment of business objects.

Part IV

From Requirements to Code

The purpose of Part II on page 21 of the dissertation was to introduce the topic of web applications and understand what concerns and forces a reusable and extensible web application architecture has to handle. In Part III on page 66 our attention has turned to architectural patterns and approaches that one can use when devising such software. We have critically evaluated the different approaches, and taken something positive from each example.

We are now going to demonstrate how business requirements can be turned into code, based on our findings from the previous overviews. The end result will be an architecture that tackles the changing needs of clients and limited schedules of developers. Each part of the system will then clearly show what its responsibilities are, helping the industry and academic development efforts alike. Such a discussion in one place has been long overdue.

Chapter 17

Agile Methodologies

It was in Section 5.1 on page 50 that we discussed the pressures a web development team is under. In particular, we have seen how in order to keep costs low, developers will ignore good practices to save time. We saw that the deployment cycle (Section 2.2.1 on page 25) in web development is significantly reduced when compared to traditional software development. Clients do ask for changes more frequently, which easy deployment of software on the web allows.

Increasingly, teams are turning to project management and software development practices stemming from the field of agile development [Highsmith and Fowler, 2001]. The practices, such as Extreme Programming (XP) [Beck and Andres, 2004] or Scrum [Schwaber and Beedle, 2001], are significant with their focus on iterative development and customer involvement. This is beneficial as the longer the development team goes without the concrete feedback of a working software, the greater the danger that they are modeling things that do not reflect what stakeholder truly needs. No design is perfect from the start and all designs need to be modified as they are implemented [Monson-Haefel, 2009] as the initial design solution will likely be wrong and certainly not optimal [Glass, 2003].

17.1 Agile Model Driven Development

In our example, we are going to loosely follow the Agile Model Driven Development (AMDD) [Ambler, 2004] approach. While, XP says a lot about iterative development and Scrum a lot about project management, AMDD fits an "Architecture Envisioning" phase at the start of each project[1], to understand the overall build requirements.

At this point we need to stress the fact that one should not fit the solution to a problem. What we mean is that too often do we see developers asking "How do I fit

[1] Itself a 0^{th} iteration.

my project to MVC". Particular patterns solve particular problems. However, in our case, we need to demonstrate a generally applicable architecture and thus we invent a problem to fit our solution.

Chapter 18

Blog Example

We are choosing a development of a blog[1] as a problem that needs to be solved. We expect to have two user roles in the system, one for the editor and one for the reader, blog visitor. Editor can access a restricted interface allowing her to write and publish content - text. While the reader can filter through and view different articles that the editor has published.

18.1 Usage Model

We are choosing user stories to explore how users will work with the application. A user story is a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it [Ambler, 2004]. Thus, through these artefacts, we can initially envisage the architecture the software will need. Not only that, by looking at the structure of the story:

> *As a (role) I want (something) so that (benefit).* [Cohn, 2004]

we see that not only is the functionality we are implementing important, but also the context in which they appear - the role part. This fits well with our discussion of DCI (Chapter 14 on page 123) that has a specific component designed to set up the application in a certain context.

18.2 Architecture Envisioning

18.2.1 Model-View-Controller

From our discussion of the different patterns we have found that the Model-View-Controller (Chapter 8 on page 71) paradigm prescribes a simple split in application

[1] A website that allows a user to publish content in a chronological order with visitors to the web discussing it.

concerns into three components. As our problem is expected to evenly make use of the three layers (Chapter 3 on page 28), unlike say Naked Objects that has a focus on the business layer, we choose MVC. In particular, we choose the Service to Worker variant with a controller component *mediating* the flow of events. As we have discussed in Section 8.4 on page 81, this strategy minimizes the number of application calls and fits the web model where the whole system needs to be instantiated from scratch on each request well (Section 2.2.1 on page 25).

18.2.2 Presenter and Model Adapter

The AMDD understands that system testing, Test Driven Development (TDD) [Martin, 2008] in particular, is an important part of each iteration. Even without this reminder, we would like to make our blog solution testable. We understood from Chapter 10 on page 99 the Model-View-Presenter model, in particular how the Supervising Controller/Presenter acts. This component is designed to encompass more complicated presentation logic. We have validated the need for such a component in Chapter 11 on page 110 discussing the Model Adapter pattern. In particular, we saw that a web application needs to minimize the number of calls to a persistence layer - database it uses. As such, the Model Adapter creates a virtual collection of data that translates system calls to database calls. Such a collection can then, we have explored, be modified in each layer of the system, thus aiding the goals of the separation of concerns, a sound goal (Chapter 4 on page 45).

Therefore, as we need to make calls to a database and want to have a greater flexibility in presentation, we call Presenter and Model Adapter patterns to action.

18.2.3 Domain Model

Now we need to turn to the business layer. When discussing the Naked Objects architecture (Chapter 13 on page 119) we realized the benefits of having all system behaviors defined in a business layer, models. We will therefore not implement any behaviors in the controller component and make models, that could potentially operate on their own, without being tangled with controllers or views which would make the application harder to update in the future (Chapter 4 on page 45).

18.2.4 Context

We saw earlier how Naked Objects are an implementation of Domain-Driven Design (DDD) principles that focus on business behaviors of a system. DDD defines for each domain model a bounded context, that is how users relate to the concepts defined in

the model. This fits well with how user stories are defined, paying attention not only to the functionality an object needs to have, but also its role. Our system will have two types of users that, in itself, represent two roles. However, the context should not be limited only to the role of the users of the system, but should apply in general, to how one sets up components. Instead of being forced to use DCI (Chapter 14 on page 123) that uses traits, our architecture will likely make use of Role Object, Façade, Factory and Template Method patterns. They are the design patterns that will *setup business objects for us in different contexts.* We are using patterns that already exist in the field, when needed, instead of taking their behavior and making them into an architecture (DCI). We aim for flexibility.

18.2.5 Use-Case Controller

In order to make it easy for the developer to see the overall behavior of the system, we are going to implement a Use-Case Controller. Such a component maps onto business models implemented from essential use cases.

A fully documented essential use case is a structured narrative [Biddle et al., 2002], expressed in the language of the application domain and of users, comprising a simplified, abstract, technology-free and implementation-independent description of one task or interaction [Constantine and Lockwood, 1999].

A use-case, furthemore, replaces the need for a navigation layer as identified in Section 3.1 on page 30.

18.2.6 Validation

As we have decided on all of the behaviors to be contained in the models, we will implement database table validation rules in there as well. It was during our discussion of Django Model-Template-View (Chapter 12 on page 115) that we saw the benefits of having table validation rules as close to the database as possible. Namely, we do not need to write input validators in all of the controllers and instead, can write them once in the models and whenever we use them, we know that the data is valid.

18.2.7 Data Transfer Object

Data Transfer Object (DTO) has been discussed (Chapter 9 on page 95) as a behavior-less object that is used by the controller to inject data to the view. This fits well with our view of the controller as a component that manages flow of the application and makes data available for the presentation layer.

18.3 Initial Domain Model

The application architecture is the primary determinant of application performance and scalability. Those quality attributes cannot be miraculously improved with some silver-bullet switch of software brands, or infrastructure "tuning". Instead, improvements in those areas require the hard work of carefully-considered (re-)architecting [Monson-Haefel, 2009].

A domain model identifies fundamental *business* entity types and the relationships between them. Tools such as Class Responsibility Collaborator (CRC) cards allow us to design the architecture of the whole system and keep track of the domain model, while the architecture is refined.

18.3.1 Class Responsibility Collaborator cards

A card split into three parts; a class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities [Ambler, 2004].

Objects are created only under the demands of the moment, which ensures that a design contains only as much information as the designer has directly experienced, and avoids premature complexity [Beck and Cunningham, 1989]. CRC cards, then, allow us to envisage an architecture without outlining complex objects.

Figure 18.1: Model CRC card.

Model	
Provide business objects. Encapsulate system behaviors. Validate data. Abstract away database connection.	(Persistence layer)

Figure 18.2: Controller CRC card.

Controller	
Setup models in a context. Provide use-case flow. Encapsulate complex presentation behavior. Render response.	Model View

Figure 18.3: View CRC card.

View	
Provide a layout for a presentation. Paginate over data in a template.	

Figures 18.1, 18.2 and 18.3 on the previous page describe the overall functionality and collaboration of the components we are going to use in the architecture. The responsibilities each component has stem from our architecture envisioning phase (Section 18.2 on page 143).

18.4 Development

In a true AMDD process [Ambler, 2004] we would have gone through the Iteration Modeling and Model Storming phases, carefully planning the actual structure of the different classes that will contain the behavior of a system. We will rather put forward a user story scenario and mockup the solution to such a business problem, discussing the various approaches used as we go along. We will be refining the responsibilities of the different architecture components envisioned (Section 18.2 on page 143) and modeled (Chapter 18.3 on the preceding page).

1. *As a blog visitor I want to see the blog articles from a specific category in a chronological order.*

2. *As an editor I want the ability to edit an article in a listing.*

From the user stories provided we see that both cases have to do with the listing of articles in a certain way. The visitor needs to see them chronologically for a given category while the editor (owner) needs to be able to edit them as well.

18.4.1 Layout

We would start off with a layout that a given part of the blog will be using. A layout would represent a "frame" for the specific "picture" we are presenting to the user. It could be as simple as providing a valid HTML definition of all the "pages" we will be presenting (Figure 18.4 on the next page).

Figure 18.4: Layout example definition.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
   http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en
   ">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=
       utf-8" />
    <meta http-equiv="Content-Language" content="en" />
    <title>Blog</title>
</head>
<body>
    <?php echo $template; ?>
</body>
</html>
```

In the example we are not worrying about applying CSS styling or forbidding outside access to the template file. What we do however see is a declaration outputting contents of a `$template` variable, the variable that will hold our specific presentation.

18.4.2 Template

In our templates we will need to output model data (an article) and embed them in HTML code. We are going to use a DTO pattern (Figure 9.3 on page 97) as we do not want to make direct calls from the view to the model, we want to reverse the flow and thus decouple the two components and responsibilities. In our example, DTO will be a "bag of values" that the template will know about and can work with. In PHP, we can for example use an `extract()` function that takes an array of values on input and extracts them to the current name space under the array keys[2].

Figure 18.5: Visitor template viewing article listing.

```
<h1>Article listing</h1>
<?php
foreach($articles as $article) {
    echo "<h2>$article['title']</h2>";
    echo "<p>$article['text']</p>";
}
```

Figure 18.5 shows a code we can use to display a template to visitors of our blog. We can see we are only resorting to article array traversal using the `foreach` construct, no complex logic will be contained. Speaking of which, we need to provide a similar template to the editor of the blog. The only difference is that the editor will be recognized as a certain user (hint, we are going to setup objects in a context) and will

[2]See `http://php.net/manual/en/function.extract.php`.

be provided with a link with each article to edit its text. In this simple example, we could resort to an `if/else` statement checking whether a user is an editor or not and if so, outputting an extra piece of information - the link. However, imagine a case where the editor does not want to see contextual ads when she is viewing the articles. Her layout might further be modified by adding an "administrator bar" that would contain links to common functions she can do in the system, like adding new users, creating articles etc. In such a case, it might be better to create a separate template for each role. The editor would then have a template similar to the one presented in Figure 18.6.

Figure 18.6: Editor template viewing article listing.

```
<h1>Article listing</h1>
<?php
foreach($articles as $article) {
    echo "<h2>$article['title']</h2>";
    echo '<a href="edit/' . $article['id'] . '">Edit</a>';
    echo "<p>$article['text']</p>";
}
```

We can see that the template is oblivious as to in which context it appears and is extended by a link to some function that would let us edit the article listed.

18.4.3 Model-View-Presenter

The application logic will be contained in a Presenter (Chapter 10 on page 99). We are using it as we will need to display different templates for each user (editor and visitor). In the future, it will allow us to contain more complex presentation logic there. But for now, we need to solve the concern of the different users. We could have a separate Presenter for each user, but why rewrite basically the same use case (view a listing of articles) for each of them? Therefore we will need to setup our business functionality in a certain context. In our simple example this is the context of two user roles. We are going to use the concept of filters that will be executed on the construction of the Presenter. Filter is just a fancy name for a function. But when the functions are executed in an order, we call this the Intercepting Filter pattern [Alur et al., 2001].

Figure 18.7: Presenter context.

```
<?php
final class BlogPresenter extends Presenter {
    public $contexts = array('listing' => array('user', 'blog'));

    public function userContext() {
        $this->user = new User();
    }

    public function blogContext() {
        $this->blog = new Blog($this->user->getRole());
    }

    public function listingUseCase() {
        ...
    }
}
```

The example in Figure 18.7 shows a context part of the Presenter. We see that the `BlogPresenter` extends a main framework class that would know how to handle input from the user and would have a constructor that guarantees the execution of filters before any work with models (through use-cases) can occur. The variable `$contexts` is an example of how one can define a list of contexts that need to be executed in an order for a given Presenter action. In our example, it will be the `listing()` function for which we need to setup the system components. First we will setup the user in a context, in her role. Then, we can setup the main blog class[3] and direct it to set itself up according to the user role that will communicate with it.

Factory, Façade, Template Method, Role Object

The main `Blog` class could be setup using a Factory, Façade, Template Method, Role Object or any number of design patterns. The Role Object pattern would suggest creating context specific views of business objects as separate roles and dynamically attaching/removing them from the core objects [Bäumer et al., 1997]. Façade is a more generic example of the Adapter pattern we have discussed earlier and is simply used when we need a simpler interface to an object [Gamma et al., 1995]. Template Method then is a behavioral pattern that defines a way - a method - one addresses a business concern with subclasses to it being the specific examples of a behavior [*ibid.*]. The specific are not concerns of our thesis. We simply want to show that we should expect a layer of design patterns that provide *interfaces and connections to our behaviors*. As

[3]We see a the user and blog models being saved in class variables that seemingly do not exists. But PHP offers us to override such call (magic __set and __get functions [Lavin, 2006]) and the main framework Presenter will know how to intercept such calls and keep all of the objects in one array.

they provide specific interfaces to their clients - controller use-cases - we are applying the *interface segregation principle* (Section 5.2.1 on page 52).

18.4.4 Use-Case Controller

For now, let us turn back to the Presenter and its listing functionality. As we want to get a high level overview of what the system does and do not want to encapsulate behavior in the application layer, we will decide for the Use-Case Controller functionality.

Figure 18.8: Presenter use-case.

```
<?php
final class BlogPresenter extends Presenter {

    ...

    public function listingUseCase() {
        $articles = $this->blog->getArticles();
        displayListing($articles);
    }
}
```

We have discussed what Use-Case Controller does in Section 13.1 on page 119. In particular we have tried to differentiate between the overall flow of events in a system and the system's behavior. If we were not careful, our Presenter could end up "knowing" too much. At the same time, one might be poised to define the category now, that we might need from the blog. But is that a use case concern? No! What is however a use-case concern is whether we display some data or not. Therefore we end the method declaration for the Use-Case Controller with a call to a method that displays the article listing. We will leave the use case controller functionality for now and keep it at a simple level where we fetch articles from a `Blog` object and then ask to display them (Figure 18.8).

18.4.5 Presentation behavior

Let us assume that we have fetched a listing of articles (perhaps in an array), now we need to display it in an appropriate way. This is where the Presenter becomes useful, allowing us to choose which template to show based on the user role. In a more general way, it is the context of the environment that will now decide what to display. The actual display logic, encapsulated in a Presenter, will then follow the *Liskov substitution principle* (Section 5.2.1 on page 52).

Figure 18.9: Presenter displaying views.

```
<?php
final class BlogPresenter extends Presenter {

    ...

    public function displayListing(array &$articles) {
        // inject data into a View through the Data Transfer
           Object
        $this->view->dto->articles = $articles;
        // choose which template to render
        switch($this->user) {
            case "editor":
                $this->view->render('editorArticleListingTemplate
                   .phtml');
                break;
            case "visitor":
                $this->view->render('
                   visitorArticleListingTemplate.phtml');
                break;
        }
    }
}
```

Figure 18.9 shows a two step process used in the presentation part of the Presenter. First, we save the article listing to a Data Transfer Object (`$this->view->dto` ; see Chapter 9 on page 95 for a discussion). This ensures that the view does not have to ask the model for data and the process is reversed. The second step is a simple `switch/case` construct choosing as to whether to display an editor or visitor template. We are using objects setup in a context at the start of the process. At the moment the presentation looks simple, but we are about to extend it now.

18.4.6 Model Adapter

An observant reader might have noticed that we are yet to specify the ordering of articles in which they appear or even their category. This is where the Model Adapter pattern (Chapter 11 on page 110) comes to play. When discussing this approach, we have seen an example using Dibi DataSource. It clearly allowed us work with database table as if it were an array collection. What we did not like is that Dibi uses a modified SQL language when creating the Adapter. We believe that this makes one vulnerable and tied up to one solution for the whole project. Just consider that the Adapter will be modified in the application and presentation layers as well as being created in the business layer. Therefore, if we decide to go with a different approach maintaining connection to a database, we have to modify code in all these layers! This is certainly not what we would want. Therefore, when designing the models in our solution, we

should expect that, beyond the classes that define model behavior, we will have a database abstraction framework - ORM, and in between *a Façade translating calls from the models into specific calls tied up to an ORM*. Then the ORM again translates these calls to a pure SQL, if that is the type of database we will use. This might look like an overkill and one might resort to not abstracting away an abstracted away ORM from SQL logic, but if a system is to be purely not tied up to a single framework, we might have no choice. An alternative is to write a Façade to an ORM when we need to replace it and thus create an abstraction layer when needed.

Therefore, what we would expect to find in the main `Blog` model, would be a data getter as mocked up in Figure 18.10. As a side note, consider the power that comes from using a context in a Presenter. The `Blog` model could have been instructed (in the context of the articles listing) to return to us a Model Adapter instead of a giving us a data collection in an array immediately. We do not have to clutter the Use-Case Controller part of the Presenter with special parameter passing asking for data in a certain way. The use-case does not care! The model would now know that in the context of this use-case conversation between the user (*As a role...*) and the system (*...I would like to do x*) it needs to give us the data in a specific format. What is more, because we have abstracted this behavior away using any number of design patterns, the `Blog` class can only specify the behaviors of the system. It would already be setup in a certain way using design patterns an abstraction above ours[4].

Figure 18.10: Blog model getter.

```
<?php
class Blog {

    ...

   /**
     * Return articles in the blog system.
     * @return array or an adapter based on the context!
     */
    public function getArticles() {
        return $this->giveMeData();
    }
}
```

By using the Model Adapter, the main business class is not cluttered with numerous getters each giving us data according to some condition. But this means we need to modify our Presenter and view and set the conditions for working with data there. Interesting how a Model Adapter works similar to the Selections component in Taligent

[4]This means the class signature might extend some other class and implement some interfaces, but this is not the point of this exercise.

MVP (Section 10.1 on page 99). We will remind ourselves that Selections would handle the task of selecting which parts of the model data we will operate upon in the view. This nicely fits with the *dependency inversion principle* (Section 5.2.1 on page 52) that advises we do not depend on concretions but instead work with abstractions. The Presenter and view work on abstractions of model data through its Adapters.

18.4.7 Presenter paginator

Just when we were to set an article category for the articles in the Presenter, our client rang and asks for the data to be displayed in a paginator on the blog. Apparently, having a 100 of her articles on one page is not a good business practice. Not to worry, we can fit that functionality right in and still keep our model intact. And why should we not? The model does not care about puny pagination. Therefore, we are going to extend our original Presenter, but beware, we will not touch the use-case controller part nor the context part.

Figure 18.11: Presenter pagination.

```
<?php
final class BlogPresenter extends Presenter {

    ...

    public function displayListing(array &$articles) {
        // first we need to specify a category that we are
           viewing
        $articles->where('category="' . $this->request->
           getCategory() . '"');

        // create a paginator component
        $paginator = new PresenterPaginator();
        // set size of articles per page
        $paginator->itemsPerPage = 10;
        // get the size of the collection
        $paginator->itemCount = count($articles);
        // set which page we are on now
        $paginator->currentPage = $this->request->getPage();

        // modify the adapter based on the paginator settings
        $articles->paginate($paginator->limit, $paginator->offset
           );

        // inject data into a View through the Data Transfer
           Object
        $this->view->dto->articles = $articles;
        // choose which template to render
        switch($this->user) {
            case "editor":
                $this->view->render('editorArticleListingTemplate
                   .phtml');
                break;
            case "visitor":
                $this->view->render('
                   visitorArticleListingTemplate.phtml');
                break;
        }
    }
}
```

Figure 18.11 shows a modified presentation logic after making use of a paginator. Let us discuss the solution in a bit more detail:

1. First we are specifying a category the user (be it editor or visitor) has requested. We are setting a condition on the Adapter returned to us from the model.

2. Next we are creating a framework component designed to deal with data pagination. Such a functionality is common and thus we should expect to have such component ready for use in the Presenter.

3. We are applying a maximum size on each page in the data collection. This is clearly a presentation concern.

4. Next we are making use of the functionality Model Adapter provides and counting the number of entries in the collection. It looks like we are working with an array, but the Adapter translates our request into an SQL query behind the scenes and gives us the result as needed.

5. We are then setting the current page we are on. This would come from the request.

6. Finally, the Adapter needs to be modified with conditions specifying which part of the collection we need.

7. The rest of the example stays the same.

Now let us imagine that all this functionality that has to do purely with presentation would be mixed with the use-case controller application functionality. Or even worse, what if the template contained all this code? Clearly, there is a need for a Presenter. We can, for example, run tests on this presentation logic, checking whether a paginator that we set for 10 items gives us more items or not.

We still have not specified an ordering status of the results the collection needs to appear in. But before that, let us answer what to do if the request, the user, has not provided a specific category.

18.4.8 Category case

Let us imagine a case where a user has not provided a category in the request. How should we solve this problem? The following are all valid solutions based on what the system should do in such a case.

18.4.8.1 Context

If not specifying a category has a profound impact on how objects in a use-case interact, their context has changed, then we should filter for the category parameter in the context section of the Presenter, Figure 18.12 on the following page shows such an example.

Figure 18.12: Category filtering in the context.

```
<?php
final class BlogPresenter extends Presenter {
    public $contexts = array('listing' => array('user', 'blog'));

    public function userContext() {
        $this->user = new User();
    }

    public function blogContext() {
        if (!$this->request->getCategory()) die('Failed to
           provide a category!');
        $this->blog = new Blog($this->user->getRole());
    }

    public function listingUseCase() {
        ...
    }
}
```

However, we need to note that if a parameter has not been provided in a request then perhaps the framework itself will not even execute the `BlogPresenter` because it will fail to match a rule requiring a category parameter to be specified in the request (Section 3.1 on page 30).

18.4.8.2 Use-Case Controller

It might be more feasible for the system to react to a missing parameter with a specific use-case that perhaps displays a totally different presentation in the end. If a user has not specified a category, we might present her with an apology that a category has not been found and show her a listing of all blog categories instead. Then, we would encompass this behavior in a model and subsequently the Use-Case Controller. Figure 18.13 on the following page shows an example of an extended use-case that displays listing of categories instead of articles. Consider that encompassing this logic in a `Blog` model does not make sense. If we ask `Blog` model for articles we should get articles and not "maybe articles, maybe categories".

Figure 18.13: Presenter use-case checking category.

```
<?php
final class BlogPresenter extends Presenter {

    ...

    public function listingUseCase() {
        // either display articles or give us a categories
           listing
        if (!$this->request->getCategory()) {
            $categories = $this->blog->getCategories();
            displayCategories($categories);
        } else {
            $articles = $this->blog->getArticles();
            displayListing($articles);
        }
    }

    ...

}
```

18.4.8.3 Presentation logic

The last example (Figure 18.14 on the next page) would then show the last scenario, where we do not need to change the template we are showing and the presentation logic is fairly similar in either case. This could mean that the system is to display articles from all categories and not a specific one. In the code we see a simple case where an extra `WHERE` clause is added to the listing logic if we have provided a category. If not, then the system will return a result set from all the categories in a database.

Figure 18.14: Presenter category filtering logic.

```
<?php
final class BlogPresenter extends Presenter {

    ...

    public function displayListing(array &$articles) {
        // are we viewing articles from all categories or just
           one?
        if ($this->request->getCategory()) {
            $articles->where('category="' . $this->request->
               getCategory() . '"');
        }

        // create a paginator component
        $paginator = new PresenterPaginator();
        // set size of articles per page
        $paginator->itemsPerPage = 10;
        // get the size of the collection
        $paginator->itemCount = count($articles);
        // set which page we are on now
        $paginator->currentPage = $this->request->getPage();

        // modify the adapter based on the paginator settings
        $articles->paginate($paginator->limit, $paginator->offset
           );

        // inject data into a View through the Data Transfer
           Object
        $this->view->dto->articles = $articles;
        // choose which template to render
        switch($this->user) {
            case "editor":
                $this->view->render('editorArticleListingTemplate
                   .phtml');
                break;
            case "visitor":
                $this->view->render('
                   visitorArticleListingTemplate.phtml');
                break;
        }
    }
}
```

18.4.9 Validation

The last point we would like to show are unexpected scenarios and different behavior on different results from business layer and/or a persistence layer. We are going to encapsulate this behavior in the models and we need a systematic way of event handling. We could use exceptions that Presenters would catch and act upon. But there is a problem. Consider a case where we use models separately from Presenters (and views),

for example in a Command Line Mode[5]. Who captures exceptions then? What is the exception class definition like? On the other hand, if we only were to return boolean values from method calls signifying whether the call has failed or not, how can we encapsulate more specific messages hinting *where* the problem was? And we are not even discussing cases where one has multiple solutions to a single fail answer.

Figure 18.15: Model class implementing event callbacks.

```
<?php
class Article implements Events {

   /**
     * Attach an event callback.
     * @param can be dummy object to switch events off.
     */
    public function attachCallback(&$object) {
        $this->callback = $object;
    }

   /**
     * Callback object.
     */
    public function eventCallback($code=null, $message=null) {
        $this->callback->raise($code, $message);
    }

    ...

    public function saveArticle($text) {
        // raise an event
        if (!isset($text)) $this->eventCallback('
           VALIDATE_PRESENCE_OF_FAILED', 'Article text has not
           been provided');
        ...
        $this->saveObject();
    }
}
```

A solution would be again to use contexts. In a *context of business models working as part of a whole framework*, our objects would be setup to use an event management class that would handle events perhaps through exceptions. On the other hand, callbacks could also be "switched off" by attaching a dummy callback object to them that never raises an exception for example. See Figure 18.15 for an example where we implement the `Events` interface that would enforce we implement at least the `attachCallback()` and `eventCallback()` methods. The former would attach an object that raises exceptions or calls a framework event management class or does nothing while the latter would be simply a way of raising new events.

[5]For example in Ruby on Rails one can access application models through a terminal.

Figure 18.16: Presenter use-case catching exceptions.

```
<?php
final class ArticlePresenter extends Presenter {

    ...

    public function saveUseCase() {
        try {
            $this->article->save(...);
            displayArticle(...);
        } catch (ArticleFormException $e) {
            displayForm(...);
        }
    }

    ...

}
```

Now that we have a way of notifying of event results, we still need to somehow act on them. Figure 18.16 is a mockup of a Presenter use-case that does just that. We can assume that the example uses exceptions for event notification and thus the use case is wrapped in a `try/catch` block that, either displays successfully saved article or shows a form asking for article input again. The use-case is not doing any complex validation, it only decides how to act on models and which, then, method to call that will render the result. The rest is up to the presentation logic to handle.

Chapter 19

Summary of Part IV, From Requirements to Code

We have, after a careful review of different patterns, models and approaches constructed a complementary architecture which achieves the same benefits as that of Reenskaug's and Complien's latest DCI. But, the end result is more flexible in its use of components, uses approaches that are more familiar to developers and addresses all concerns of a web project.

The Presenter/controller is by far the most important part of the architecture, directing flow of events. We have implemented the three components a controller should encompass - *context, use-case and presentation* in one class (Figure 19.1 on the following page).

However, in terms of reuse, we might find it useful to actually split the three functionalities into separate classes. Then the contexts would reside in `Contexts` class, the use-cases in a `Controller` class and the presentation logic in a `Presenter` class. In order to route these components together, we would keep the `Contexts` and `Presenter` as inner objects of the `Controller`, routing all data manipulation and method calls[1] to the `Controller` object[2] for increased clarity and reuse. This is an example of favoring composition rather than inheritance (Section 5.3 on page 53).

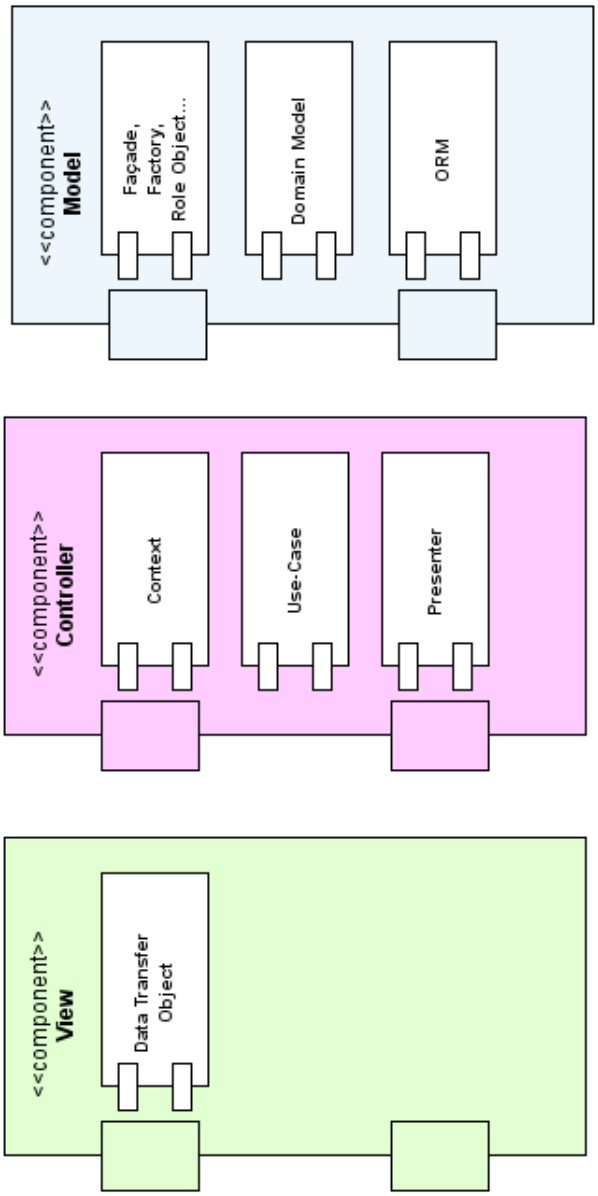
[1] In PHP through magic ___set, ___get and ___call methods, `http://php.net/manual/en/language.oop5.magic.php`.

[2] Technically, the Controller would add a reference to itself when constructing the Context and a Presenter, so the other two components can call it. This is a case of object composition [Gamma et al., 1995].

Figure 19.1: A complete controller containing separate context, use-case and presentation logic.

```
<?php
final class BlogPresenter extends Presenter {

    /********************* Context *********************/

    public $contexts = array('listing' => array('user', 'blog'));

    public function userContext() {
        $this->user = new User();
    }

    public function blogContext() {
        $this->blog = new Blog($this->user->getRole());
    }

    /********************* Use-Case *********************/

    public function blogUseCase() {
        try {
            $this->blog->someAction(...);
            $articles = $this->blog->fetchSomeData();
            displayListing($articles);
        } catch (SomeSortOfException $e) {
            displayForm(...);
        }
    }

    /******************* Presentation *******************/

    public function displayListing(array &$articles) {
        ...
        $this->view->dto->articles = $articles;
        // choose which template to render
        switch($this->user) {
            case "editor":
                $this->view->render('editorArticleListingTemplate
                    .phtml');
                break;
            case "visitor":
                $this->view->render('
                    visitorArticleListingTemplate.phtml');
                break;
        }
    }

}
```

Figure 19.2: Final architecture.



Looking at the architecture once more (Figure 19.2) we can identify three main components:

1. **View**; that is to contain only a traversal of objects passed to it through **DTO** (Chapter 9 on page 95), thus the template does not contain any logic calling models.

2. **Controller**; that will contain three steps a request goes through:

   (a) First, model objects will be setup according to the **context** in which the request appears.

(b) Second, we describe the overall flow of events in a system by writing a **use-case** method.

(c) Third, at the end of the use-case, we identify which logic to use to present the result. Then, the **Presenter** would work with logic functionality as paginators, possibly adding conditions to Model Adapters as needed.

3. **Model**; is a component encompassing behaviors and data validation. We can identify three layers in a model:

(a) First, we would expect to find Role Objects, Template Methods, Factories and other approaches used to construct objects in a certain context. These objects can be likened to Aggregate Root entities in Domain-Driven Design [Evans, 2004] that cluster domain objects that will be treated from the use-case as one unit for the purposes of data exchange. Larman [2004] would call this functionality as being a Creator pattern in general.

(b) Second, we have the actual **domain models** containing behaviors, and the business logic.

(c) Third, we would abstract away all persistence layer connections from domain models to a further layer that handles them, an **ORM**.

Part V

Project Conclusions

Chapter 20

Conclusion

The aim of this dissertation was to research *methods* by which web developers can better realize the architecture of a web application to meet specific business needs. In particular, the focus was on clearly determining which parts of the application correspond to which *concerns*. The specific research objectives were (Chapter 1 on page 15):

1. *Identify* the concerns a web application architecture has to handle and the different forces put on it.
2. *Identify* approaches used to describe web application architectures.
3. *Evaluate* critically the patterns, models and approaches relevant to a development of a web system that adhere to sound software engineering principles in the web domain.
4. *Explore* the merit of the different approaches identified in 3. and their synergic collaboration and effect.
5. *Formulate* architecture recommendations for industry web developers and web engineering academics alike.

This chapter will revisit the research objectives outlined above, summarize the findings of this research work and offer conclusions based on these findings. The previous part of the dissertation, Part IV "From Requirements to Code", was exhaustive as it tackled the issue of applying the approaches identified earlier, and hence this chapter offers a coherent conclusion. Success in tackling research objectives, recommendations for future research as well as recommendations and, importantly, knowledge contribution will all be discussed. We will conclude this chapter with a self-evaluation of the research process undertaken in this dissertation.

20.1 Research Objectives: Summary of Findings and Conclusions

Identify the concerns a web application architecture has to handle and the different forces put on it

We undertook the task of identifying what application concerns web applications must handle. This was based on our initial literature review that has concluded that academics and industry developers see merit in different concerns.

We came to the conclusion that application, business and presentation concerns need to be represented in the majority of web systems (Chapter 3 on page 28). On the other hand, we have discounted the navigation concerns approach, touted by hypermedia web engineering applications, as lacking in applicability today (Section 3.1 on page 30). We have based this conclusion predominnatly on a fact that requests to web servers are now responded to with resources that often do not represent whole "pages". Pages are no longer a "unit" that an application is conceptually based upon and navigation should rather be represented by a use-case.

In terms of our research undertaken into the merits of state concerns, we have discovered that, in particular, web frameworks based on .NET technology offer an array of useful approaches making web components stateful. Our own exploration into an applied case of using a state layer showed that maintaining whole components between requests from a client is hindered by a database access bottleneck (Section 3.2 on page 37).

Identify approaches used to describe web application architectures.

To determine which approach is best in terms of describing web application architectures, we have discussed aspects ranging from inheritance and SIRs, patterns, frameworks and middleware. Thus we could identify that patterns are a mechanism often used to describe architectures (Chapter 5 on page 50). This helped us focus on an area of knowledge that would best describe how web application architectures tackle the different constraints put onto them.

Evaluate critically the patterns, models and approaches relevant to a development of a web system that adhere to sound software engineering principles in the web domain.

The core of our work represented our study of architectural & design patterns, models and approaches that are used by academics and industry developers alike. The

conclusion of our evaluation is that the different approaches sometimes cover a whole development process (Naked Objects, Chapter 13 on page 119), while some describe only a part of the whole architecture and need to be extended (Data, Context and Interaction, Chapter 14 on page 123) and yet others are so broadly defined (Model-View-Controller, Chapter 8 on page 71) that we cannot assume anything about a system developed under these patterns.

An unexpected conclusion can be drawn from this research on the different architectures that, as the field of web architecting has moved to a stage where patterns are used by developers in the industry of all expertise levels, we are overloaded with a multitude of pattern variants that still, somehow, are supposed to be discussed under their historic pattern umbrella term. *This is clearly no longer feasible and helpful.*

Explore the merit of the different approaches identified and their synergic collaboration and effect.

We have taken an extra step in evaluating patterns on their own and attempted to come up with a group of collaborating patterns and approaches (Part IV on page 139). This, we find, is a difficult task as the effect a pattern has on a system is hard to quantify, and thus one has to rely on qualitative measurements and evaluation of patterns *applied in specific circumstances.*

The conclusion is that the ways experts discuss patterns are broad and varied and rather informal (Chapter 7 on page 68). The multitude of forms have been reviewed and, lacking one best approach, we have used an informal discussion as well. More importantly, we can draw a conclusion that as one author tries to argue for a benefit of this one architecture and this one pattern, they often draw a comparison with another pattern that does not solve a certain part of a system architecture well. For example, the Model-Pipe-View-Controller was offered as a pattern that tries to handle the task of sorting data from a business model in a presentation layer. We however, found a better solution that consists of a combination of Model-View-Controller, Presenter and Model Adapter patterns. What we can conclude is that, as some patterns do not clearly describe what their aspects are, they *must be discussed in conjunction with other patterns that further clarify the system.*

However, we find a severe lack of discussion of such pattern pairings, or even a coherent categorization of patterns that correspond to and describe an overall architecture, and those that do not[1]. This situation is not helped by the fact that the various pattern variants are linked to their generic predecessor. An example from industry is

[1] MVC itself is considered an architectural pattern or a design pattern or a design style.

the common recommendation to "just use MVC" that is too broad, and one can imagine a multitude of applied MVC variants (see procedural MVC in Figure 20.1). While at the same time, from the academic sphere, we find researchers citing aging books that describe an MVC pattern in too general a setting. What is more worrying is when a desktop pattern is discussed in a web setting. The web works as a request/response system where application components on the server are trashed after each request and a network time represents different challenges on the web than on the desktop.

We conclude and stress that as one combines patterns to derive an actual coherent architecture, the qualitative attributes that the system exhibits need to constantly re-evaluated, particularly in the context of the programming language used.

Figure 20.1: Procedural MVC pattern.

```
<?php

switch($_GET['route']) {
    case "home":
        controller();
        break;
    default:
        die("Page not found");
}

function getData() {
    return 'Lorem ipsum dolor sit';
}

function controller() {
    $data = getData();
    view($data);
}

function view($data) {
    echo '<html><head><title>Template</title></head><body>' .
       $data . '</body></html>';
}
```

Formulate architecture recommendations for industry web developers and web engineering academics alike.

We set out to combine several approaches to describe clearly an architecture of a system. We have described, in an example of a blog system, how a web developer approaches the task of architecture envisioning up to the actual process of coding (Part IV on page 139).

We can draw the conclusion that one needs to know and apply a multitude of approaches in order to describe coherently a web architecture even for a simple web

application. That is, if one wants to adhere to sound object-oriented principles (Section 5.2.1 on page 52).

The solution offered is interesting in that it uses patterns and approaches that have existed for almost three decades, yet their combination is unique, to the best of our knowledge, and embodies a web development scenario well, guiding the developer through the development process, forcing her to *clearly separate concerns and prepare a system for future extensions and reuse*. However, these conclusions come with a caveat: as we looked deeper to the different approaches applied, we discovered niche areas of web development that tackle small problem areas in a painstaking detail. Therefore, the deeper the knowledge a software architect has, the more likely she is to apply better fitting solutions. However, this is the problem: as there is no accurate, general advice on web development processes, one has to build enough systems to gain experience to understand the problem at hand fully.

20.2 Recommendations & Future Work

We set out to combine several approaches to describe an architecture of a system, finding that knowing one pattern, one approach, is not sufficient. What we found is that, although there is no coherent way of discussing, describing and evaluating patterns, the amount of informal discussion on these topics is abundant, and developers produce web applications that do work. However, they often represent solutions that are not reusable in the future or that are tied to a particular domain or framework. Is this perhaps because the cost of web development is so low that a client can afford to *skimp on quality*?

We, of course, will not advocate that one can get away with poor coding based on the resources having to be "inexpensive". Instead, we advocate a higher level of expertise so that this advanced knowledge can be used even in "inexpensive" scenarios.

Our recommendations are as follows:

1. For researchers in the subject to not quote generic variants of patterns or variants that are applied in a different domain. Patterns are now so varied that they need to be described and cited in their *specific* version, not a generic one.

2. Furthermore, as one pattern is not enough to describe even an overall system architecture, combinations of patterns and approaches need to be studied. We would like to make a point that mentioning that "pattern X" can be often found and used in conjunction with "pattern Y" is insufficient, as the combination of these patterns creates and presents a unique set of qualitative traits. *A disad-*

vantage of one pattern can be leveraged by using another and vice versa. Serious discussion on this topic needs to start.

3. Following from the previous point, a web engineering field should focus on providing help for the developers in the field and represent "engineering" in the domain of computer science. What we mean is that the domain can build up to date repositories that describe *pattern variants and their collaborative effects.* We therefore suggest that the next steps should be as follows:

   (a) A first stage would be to find successful architectures in the field and describe them in terms of patterns (after all a pattern is a solution to a problem) used. Such projects could, in the first instance, be taken from open-source repositories that are plentiful[2]. At the moment, the task of pattern identification cannot be fully done automatically[3].

   (b) In a second stage, the *overall* architecture could be rated in terms of qualitative benefits derived. This would mean that one could see which approaches were used to tackle a particular problem, and would help developers in the field better gauge the good from the bad.

   (c) A third stage would reflect back on this knowledge base and would identify which patterns, in the long run, needlessly *increased the complexity of a system or represented a non-reusable solution.* Then, experts in the field could suggest alternative solutions. Perhaps, approaches such as artificial neural networks could be trained to recognize patterns in future systems based on a large sample which would be present at this stage.

We realize that the process of software architecting cannot be completely streamlined. But at the moment, too much knowledge is kept only in the heads of senior developers and the transfer of knowledge is slow as it lacks any formal guidance. Successful experts in the industry do not know what they know [Amin and Cohendet, 2004]. We should aim to extract this knowledge as how else can we comprehend the ever-increasing amount of approaches on offer?

This represents a problem not just for developers building web software in the industry, but for researchers like us as well. One has to get theirs hands on any possible little gem of information to piece together the puzzle that some patterns represent. The more one reads up about a certain approach, the more one feels lost.

[2]GitHub, Google Code, Bitbucket, Sourceforge, Launchpad, Snipplr etc.
[3]Blewitt [2006] presents a verification language for some design patterns implemented in Java.

Some researchers advocate building pattern languages, but that is not a completely beneficial approach either. The end result is a tightly controlled group of patterns, much as we have presented in the earlier parts of the dissertation. But what about other patterns that might be more applicable in different scenarios? As patterns simply solve concerns put on a system, we should not try to reduce their number first, but to understand what the concerns and thus solutions were. The web engineering field would be well advised to take this route.

20.3 Contribution to Knowledge

How has this research work contributed to the knowledge in the field? We are recommending a track that the web engineering field should take, to develop a "standards" repository cataloging pattern variants and their combinations. If this suggestion is applied, then one could reference exact pattern combinations and see their examples, descriptions and qualitative benefits in a coherent manner. Up until now, all (informal) research has focused on producing cookbook-style recipes to tackle various problems. But, as the different pattern variants and combinations applied in a particular language have different results on architecture qualitative measures, we should study them and understand them. In the end, the system uses a combination of patterns and not one pattern in isolation, then another etc.

It should not be so painstakingly difficult to understand every possible web architecture approach there is in order to build a web system well and offer alternative solutions.

20.4 Finally

We believe that a contribution has been made in this dissertation to enable a practitioner to understand the various strands of a development process and better realise the architecture of a web application. Suggestions have been made for the development of a suitable standards repository, encompassing pattern variants and combinations. Definitions in existing practice and pointers to good practice are highlighted.

Appendixes

Appendix A

Further Discussions

A.1 HTML5

HTML5 brings, apart from new HTML and CSS tags also new JavaScript APIs. These APIs provide client side storage, better communication and desktop experience:

1. A a basic **relational database** based on SQLite:

```
var db = window.openDatabase("Database Name", "Database
   Version");
db.transaction(function(tx) {
        tx.executeSql("SELECT * FROM test", [],
           successCallback, errorCallback);
});
```

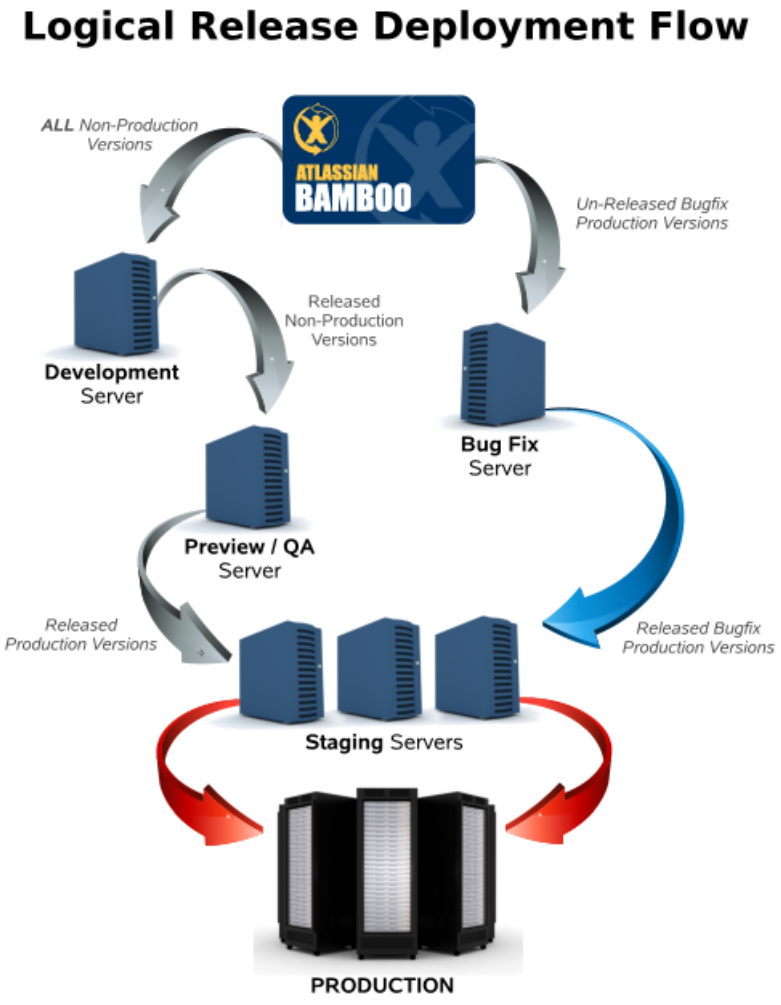
2. **Web Sockets** that allow the web server to push new data to the client browser.

3. **Notifications** that display a desktop notification rather than just a browser notification. Currently applications have to change the *<title>* of the browser window or play a sound to notify the user.

A.2 Web Applications Release & Deployment

Note that with a shorter maintenance and release cycle, bugs can creep in into the released software. Continuous Integration partly solves this problem by creating a staging server(s) where all updates and releases are pushed but are checked one last time before deploying onto the live servers, a diagram of which can be seen in Figure A.1 on the next page.

Figure A.1: Attlasian Bamboo Release & Deployment Flow.



```
[{
        "text":"Hello, this is a new message",
        "type":"text",
        "user":"Jerry S."
}]
```

Figure A.2: JSON response.

```
<tr>
        <td class="user text our">Jerry S.</td>
        <td class="body text our">Hello, this is a new message
                <div id="highlight_6" onclick="highlightMessage('6');return
                    false;" class="highlight"><a href=""></a></div>
        </td>
</tr>
```

Figure A.3: HTML response.

A.3 Response types

The first code example represents a JSON object consisting of a message in a chatroom program. The second then represents its HTML equivalent after being processed on the client with JavaScript. The server only needs to return the message in a generic format and it is up to the client to determine the context it sees the data. In our example, the message is from us and thus a *className* attribute is appended to the *<td>*, table column, elements. Unfortunately, the data has to be processed somewhere, and large objects, such as a 100 of such messages, can freeze the JavaScript engine in the browser and hinder the user experience none the less. The solution then is to preload the data on the server with a first request and only append a few processed messages at a time.

Glossary

Ajax **A**synchronous **Ja**vaScript and **X**ML is a group of interrelated Web development techniques used for creating interactive Web applications or rich Internet applications.

Apache A very popular Open Source, Unix-based Web server.

API An **a**pplication **p**rogramming **i**nterface is an interface that a software program implements to allow other software to interact with it.

CGI The **C**ommon **G**ateway **I**nterface is a standard that defines how webserver software can delegate the generation of webpages to a console application.

Extreme Programming is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements.

HTML5 is the next major revision of HTML, the core markup language of the World Wide Web.

JavaScript is an object-oriented scripting language used to enable programmatic access to computational objects within a host environment.

JSON short for **J**ava**S**cript **O**bject **N**otation, is a lightweight computer data interchange format.

ORM **O**bject **R**elational **M**apping is a way of translating programming language queries to calls to different relational databases.

Scrum an agile software development methodology with a focus on project management.

SQLite is a relatively small embedded relational database management system.

URL **U**nified **R**esource **L**ocator, the global address of resources on the web.

XML is a set of rules for encoding documents electronically and their interchange.

Index

Bibliography

Aguiar, A., Sousa, A. and Pinto, A. [2001], Use-case controller, *in* 'Sixth European Conference on Pattern Languages of Programs (EuroPLoP 2001)'.

Alameda, E. [2008], *Foundation Rails 2*, friends of ED (Apress).

Alexander, C., Ishikawa, S. and Silverstein, M. [1977], *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press.

Allen, R., Lo, N. and Brown, S. [2008], *Zend Framework in Action*, Manning Publications Co., Greenwich, CT, USA.

Alles, M., Crosby, D., Erickson, C., Harleton, B., Marsiglia, M., Pattison, G. and Stienstra, C. [2006], 'Presenter first: Organizing complex GUI applications for test-driven development', *AGILE Conference* **0**, 276–288.

Alur, D., Crupi, J. and Malks, D. [2001], *Core J2EE Patterns: Best practices and design strategies*, Upper SaddleRiver, Sun Microsystems.

Ambler, S. W. [2004], *The Object Primer: Agile Model-Driven Development with UML 2.0*, Cambridge University Press, New York, NY, USA.

Amin, A. and Cohendet, P. [2004], *Architectures of Knowledge: Firms, Capabilities, and Communities*, Oxford University Press.

Antoniou, G. and vanHarmelen, F. [2008], *A Semantic Web Primer*, second edn, MIT Press, Cambridge, MA, USA.

Appleton, B. [1997], 'Patterns and software: Essential concepts and terminology'.
URL: *http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html*

Asada, T., Swonger, R F Bounds, N. and Duerig, P. [1992], The quantified design space - a tool for the quantitative analysis of designs, Technical report, Carnegie Mellon University.

Askins, B. and Green, A. [2006], 'A Rails/Django Comparison'.
URL: *https://docs.google.com/View?docid=dcn8282p_1hg4sr9*

Atherton, B. [2008], 'Model pipe view controller'.
URL: *http://c2.com/cgi/wiki?ModelPipeViewController*

Atkinson, C., Bunse, C., Groß, H. and Kühne, T. [2002], 'Towards a general component model for web-based applications', *Annals of Software Engineering* **13**, 35–69.

Avgeriou, P. and Zdun, U. [2005], Architectural patterns revisited - a pattern language, *in* '10th European Conference on Pattern Languages of Programs', pp. 132–136.

Bachman, F., Bass, L. and Nord, R. [2007], Modifiability tactics, Technical report, Carnegie Mellon University, Pittsburgh, PA, USA.

Barry, C. and Lang, M. [2003], A comparison of traditional and multimedia information systems development practices, *in* 'Information and Software Technology', pp. 217–227.

Baskerville, R. and Pries-Heje, J. [2004], 'Short cycle time systems development', *Information Systems Journal* (14), 237–264.

Bass, L., Clements, P. and Kazman, R. [2003], *Software Architecture in Practice*, 2 edn, Addison-Wesley Professional.

Batenin, A. [2006], Subjectivity and ownership: A perspective on software reuse, PhD thesis, University of Bath.

Bäumer, D., Riehle, D., Siberski, W. and Wulf, M. [1997], The role object pattern, *in* '4th Conference on Pattern Languages of Programs'.

Beck, K. and Andres, C. [2004], *Extreme Programming Explained: Embrace Change*, 2nd edn, Addison-Wesley Professional.

Beck, K. and Cunningham, W. [1989], A laboratory for teaching object oriented thinking, *in* 'OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications', ACM, New York, NY, USA, pp. 1–6.

Bennett, J. [2009], *Practical Django Projects, Second Edition*, Apress, Berkely, CA, USA.

Biddle, R., Noble, J. and Tempero, E. [2002], From essential use cases to objects, *in* 'forUSE 2002'.

Blewitt, A. [2006], Hedgehog: Automatic Verification of Design Patterns in Java, PhD thesis, University of Edinburgh.

Bochicchio, M. and Fiore, N. [2005], WARP for re-engineering of web applications, *in* 'ACM Conference on Hypertext and Hypermedia', pp. 295–297.

Booch, G. [2007*a*], 'The irrelevance of architecture', *IEEE Software* **24**(3), 10–11.

Booch, G. [2007*b*], 'It is what it is because it was what it was', *IEEE Software* **24**(1), 14–15.

Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J. and Houston, K. [2007], *Object-oriented analysis and design with applications*, third edn, Addison-Wesley Professional.

Bower, A. and McGlashan, B. [2000], Twisting the triad: The evolution of the Dolphin Smalltalk MVP application framework, *in* 'European Smalltalk User Group (ESUG) 2000'.

Brambilla, M. and Origgi, A. [2008], Mvc-webflow: An ajax tool for online modeling of mvc-2 web applications, *in* 'ICWE '08: Proceedings of the 2008 Eighth International Conference on Web Engineering', IEEE Computer Society, pp. 344–349.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. [1996], *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Vol. 1, John Wiley & Sons.

Cai, J., Kapila, R. and Pal, G. [2000], '"HMVC: The layered pattern for developing strong client tiers', *JavaWorld* .
URL: *http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.html*

Christodoulou, S. P., Zafiris, P. A. and Papatheodorou, T. S. [2000], WWW2000: The developer's view and a practitioner's approach to Web Engineering, *in* '2nd ICSE Workshop on Web Engineering', pp. 75–92.

Clements, P. [1996], Coming attractions in software architecture, Technical Report CMU/SEI-96-TR-008, Carnegie Mellon University.

Cohn, M. [2004], *User Stories Applied: For Agile Software Development*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Conallen, J. [2000], *Building Web Applications with UML*, Addison-Wesley.

Constantine, L. L. [2002], The emperor has no clothes: Naked objects meet the interface.
URL: *http://www.foruse.com/articles/nakedobjects.htm*

Constantine, L. L. and Lockwood, L. A. D. [1999], *Software for use: A practical guide to the models and methods of usage-centered design*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Coplien, J. O. [2009], 'The DCI architecture: Supporting the agile agenda in your software architecture', *Øredev* .

Coplien, J. O. and Schmidt, D. C., eds [1995], *Pattern languages of program design*, ACM Press/Addison-Wesley Publishing Co.

Coutaz, J. [1987], PAC, an object oriented model for dialog design, *in* 'Human-Computer Interaction - Interact'87', Elsevier, pp. 431–436.

Daum, B. and Merten, U. [2003], *System Architecture with XML*, Morgan Kaufmann Publishers Inc.

Deacon, J. [1995], Model-view-controller (MVC) architecture, *in* 'Training', Vol. 1995, JDL, pp. 1–6.
URL: *http://faculty.washington.edu/hanks/Courses/560/s06/Handouts/mvc_observer.pdf*

DeMarco, T. [1978], *Structured Analysis and System Specification*, Yourdon Press Computing Series.

Deshpande, Y. [2004], Web Engineering curriculum: A case study of an evolving framework, *in* '4th International Conference on Web Engineering', pp. 526–530.

Dibi API Documentation [n.d.].
URL: *http://api.dibiphp.com*

Discenza, A. [1999], Design patterns for www museum hypermedia, Technical report, Politecnico di Milano.

Dong, J., Alencar, P. and Cowan, D. [2005], Automating the analysis of design component contracts, *in* 'Software - Practice and Experience', Wiley, pp. 27–71.

Evans, E. [2004], *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley.

Fayad, M. E. [2000], 'Introduction to the Computing Surveys' electronic symposium on object-oriented application frameworks', *ACM Computing Surveys (CSUR)* **32**(1).

Firesmith, D. G. [1996], *Wisdom of the Gurus: A Vision for Object Technology*, Cambridge University Press, chapter Use Cases: the Pros and Cons, pp. 171–180.

Fowler, M. [1999], *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

Fowler, M. [2002], *Patterns of Enterprise Application Architecture*, Addison-Wesley.

Fowler, M. [2004], 'Presentation Model'.
URL: *http://martinfowler.com/eaaDev/PresentationModel.html*

Fowler, M. [2006*a*], 'GUI architectures'.
URL: *http://martinfowler.com/eaaDev/uiArchs.html*

Fowler, M. [2006*b*], 'Passive View'.
URL: *http://martinfowler.com/eaaDev/PassiveScreen.html*

Fowler, M. [2006*c*], 'Supervising Controller'.
URL: *http://martinfowler.com/eaaDev/SupervisingPresenter.html*

Fowler, M. [2006*d*], 'Writing software patterns'.
URL: *http://martinfowler.com/articles/writingPatterns.html*

Fried, J., Hansson, H. D. and Linderman, M. [2009], *Getting Real: The smarter, faster, easier way to build a successful web application*, 37signals.

Gacek, C., Abd-Allah, A., Clark, B. and Boehm, B. [1995], On the definition of software system architecture.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. [1995], *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley.

Gardner, T. A. [2001], Inheritance relationships for disciplined software construction, *in* 'Distinguished Dissertations', Springer.

Garlan, D., Monroe, R. T. and Wile, D. [2000], Acme: Architectural description of component-based systems, *in* 'Foundations of Component-Based Systems', Cambridge University Press.

Garzotto, F., Paolini, P., Bolchini, D. and Valenti, S. [1999], "Modeling-by-patterns" of web applications, *in* 'Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling', pp. 293–306.

Gellersen, H.-W. and Gaedke, M. [1999], 'Object-oriented web application development', *IEEE Internet Computing* **3**(1), 60–68.

Genßler, T. and Schulz, B. [1999], Transforming inheritance into composition - A reengineering pattern, *in* '4th EuroPLoP'.

Glass, R. L. [2003], *Facts and Fallacies of Software Engineering*, Addison Wesley.

Grand, M. [2002], *Patterns in Java: A Catalog of Reusable Design Patterns*, Wiley & Sons.

Harrison, N., Avgeriou, P. and Zdun, U. [2007], 'Using patterns to capture architectural decisions', *IEEE Software* **24**, 38–45.

Highsmith, J. and Fowler, M. [2001], 'The agile manifesto', *Software Development Magazine* **9**(8), 29–30.

Holovaty, A. and Kaplan-Moss, J. [2009], *The Django Book*, 2.0 edn.
URL: *http://www.djangobook.com/en/2.0/*

Holub, A. [1999], 'Building user interfaces for object-oriented systems', *JavaWorld* .
URL: *http://www.javaworld.com/javaworld/jw-07-1999/jw-07-toolbox.html*

Homer [1998], *The Iliad*, Penguin Classics Deluxe edn, Penguin Classics.

Jaaksi, A. [1995], 'Implementing interactive applications in C++', *Software: Practice and Experience* **25**(3), 271–289.

Jacobson, I., Christenson, M., Jonsson, P. and Overgaard, G. [1992], *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison Wesley.

Jeary, S., Phalp, K. and Vincent, J. [2009], 'An evaluation of the utility of web development methods', *Software Quality Journal* **17**(2), 125–150.

Kamal, A. W., Avgeriou, P. and Zdun, U. [2008], Modeling variants of architectural patterns, *in* '13th European Conference on Pattern Languages of Programs (EuroPLoP 2008)', pp. 1–23.

Kappel, G., Michlmayr, E. and Pröll, B. [2004], Web Engineering - old wine in new bottles, *in* 'International Conference on Web Engineering', Springer-Verlag, pp. 6–12.

Kazman, R., Klein, M. H., Barbacci, M. R., Longstaff, T. A., Lipson, H. F. and Carrière, J. [1998], The architecture tradeoff analysis method, Technical Report CMU/SEI-98-TR-008, Carnegie Mellon University.

Kegel, H. and Steimann, F. [2008], Systematically refactoring inheritance to delegation in Java, *in* 'International Conference on Software Engineering', ACM, pp. 431–440.

Knight, A. and Dai, N. [2002], 'Objects and the web', *IEEE Software* **19**(2), 51–59.

Knoernschild, K. [2001], *Java Design: Objects, UML, and Process*, Pearson Education.

Kojarski, S. and Lorenz, D. H. [2003], Domain driven web development with WebJinn, *in* 'OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications', ACM, pp. 53–65.

Krasner, G. E. and Pope, S. T. [1988], 'A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80', *Journal of Object Oriented Programming* **1**(3), 26–49.

Krueger, C. W. [1992], 'Software reuse', *ACM Computing Surveys* (24), 131–183.

Kufner, J. [2010], Dynamic pipeline in a web application, Master's thesis, Czech Technical University in Prague.

Kukola, T. [2008], On user interface architectures and implementation, Master's thesis, University of Tampere.

Kummel, B. [2010], *Apache MyFaces 1.2 Web Application Development*, Packt Publishing.

Lang, M. [2004], A critical review of challenges in hypermedia systems development, *in* '13th International Conference on Information Systems Development', pp. 277–288.

Lang, M. [2009], The influence of short project timeframes on web development practices: A field study, *in* 'International Conference on Information Systems Development (ISD2009)'.

Lang, M. and Fitzgerald, B. [2005], 'Hypermedia systems development practices: A survey', *IEEE Software* (22), 68–75.

Lang, M. and Fitzgerald, B. [2007], 'Web-based systems design: A study of contemporary practices and an explanatory framework based on method-in-action', *Requirements Engineering* (12), 203–220.

Larman, C. [2004], *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3 edn, Prentice Hall PTR.

Läufer, K. [2008], 'A stroll through domain-driven development with naked objects', *Computing in Science and Engineering* **10**(3), 76–83.

Lavin, P. [2006], *Object-Oriented PHP: Concepts, Techniques, and Code*, No Starch Press, San Francisco, CA, USA.

Linowes, J. S. [2007], 'Evaluating web development frameworks: Rails and Django'.
URL: *http://www.vaporbase.com/var/uploads/File/rails-vs-django.pdf*

Lynex, A. and Layzell, P. J. [1998], 'Organisational considerations for software reuse', *Annals of Software Engineering* (5), 105–124.

MacDonald, M. and Szpuszta, M. [2005], *Pro ASP.NET 2.0 in C# 2005*, Apress.

Mac OS X Reference Library [n.d.].
URL: *http://developer.apple.com/mac/library/navigation/index.html*

Mahemoff, M. [2006], *AJAX Design Patterns*, O'Reilly Media, Inc.

Mann, K. D. [2004], *JavaServer Faces in Action*, In Action, Manning Publications Co., Greenwich, CT, USA.

Martelli, A. [2007], 'Python design patterns'.
URL: *http://code.google.com/intl/ja/edu/languages/index.html*

Martin, R. C. [1997], 'The dependency inversion principle'.
URL: *http://www.objectmentor.com/resources/articles/dip.pdf*

Martin, R. C. [2008], *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

McDonald, A. and Welland, R. [2001], Web Engineering in practice, *in* '4th WWW10 Workshop on Web Engineering', pp. 21–30.

Meyer, B. [1988], *Object-Oriented Software Construction*, Prentice Hall.

Monson-Haefel, R. [2009], *97 Things Every Software Architect Should Know: A Collective Wisdom from the Experts*, O'Reilly.

Morgan, D. [2006], 'Maintaining state in web applications', *Network Security* **2006**.

Mukhar, K., Zelenak, C., Weaver, J. L. and Crume, J. [2005], *Beginning Java EE 5: From Novice to Professional*, Apress, Berkely, CA, USA.

Nette Documentation [n.d.].
URL: *http://doc.nette.org*

Nielsen, J. [1993], *Usability Engineering*, Morgan Kaufmann.

Offutt, J. [2002], 'Quality attributes of web software applications', *IEEE Software* (19), 25–32.

Offutt, J. and Wu, Y. [2010], 'Modeling presentation layers of web applications for testing', *Software and Systems Modelling* **9**(2), 257–280.

Opdyke, W. F. [1992], Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois at Urbana-Champaign.

Page-Jones, M. [1988], *The Practical guide to structured systems design*, Yourdon Press Computing Series.

Parnas, D. L. [1979], *Classics in Software Engineering*, Yourdon Press, Upper Saddle River, NJ, USA, chapter On the criteria to be used in decomposing systems into modules, pp. 139–150.

Pawson, R. [2004], Naked Objects, PhD thesis, Trinity College, Dublin.

Potel, M. [1996], 'MVP: Model-View-Presenter - The Taligent programming model for C++ and Java'.
URL: *http://www.wildcrest.com/Potel/Portfolio/mvp.pdf*

Pressman, R. S. [1998], 'Can internet-based applications be engineered?', *IEEE Software* **15**, 104–110.

Pressman, R. S. [2001], *Software Engineering: A Practitioner's Approach*, 5 edn, McGraw-Hill Higher Education.

Puerta, A. R., Eriksson, H., Gennari, J. H. and Musen, M. A. [1998], 'Model-based automated generation of user interfaces', pp. 508–515.

Redwine, Jr., S. T. and Riddle, W. E. [1985], Software technology maturation, *in* 'ICSE '85: Proceedings of the 8th international conference on Software engineering', IEEE Computer Society Press, pp. 189–200.

Reenskaug, T. [1979*a*], Models-views-controllers, Technical Report 12, Xerox PARC.
URL: *http://heim.ifi.uio.no/ trygver/1979/mvc-2/1979-12-MVC.pdf*

Reenskaug, T. [1979*b*], Thing-Model-View-Editor: An example from a planning system, Technical Report 5, Xerox PARC.
URL: *http://heim.ifi.uio.no/ trygver/1979/mvc-1/1979-05-MVC.pdf*

Reenskaug, T. [2009], The common sense of object oriented programming, Technical report, University of Oslo.
URL: *http://folk.uio.no/trygver/2008/commonsense.pdf*

Reenskaug, T. and Coplien, J. O. [2009], 'The DCI Architecture: A New Vision of Object-Oriented Programming', *Artima Developer* .
URL: *http://www.artima.com/articles/dci_visionP.html*

Reinhartz-Berger, I., Dori, D. and Katz, S. [2002], Open reuse of component designs in OPM/Web, *in* '26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment', IEEE CS Press, pp. 19–26.

Robert, M. [2002], *Agile Software Development: Principles, Patterns and Practices*, Pearson Education.

Robinson, D. and Coar, K. [2004], 'Rfc 3875: The common gateway interface (cgi)'.
URL: *http://tools.ietf.org/html/rfc3875*

Rossi, G., Schwabe, D. and Lyardet, F. [2000], Abstraction and reuse mechanisms in web application models, *in* 'Conceptual Modeling for E-Business and the Web', pp. 76–90.

Ruby, S., Thomas, D. and Hansson, D. H. [2009], *Agile Web Development with Rails*, 3 edn, Pragmatic Bookshelf.

Sauter, P., Vögler, G., Specht, G. and Flor, T. [2005], 'A Model-View-Controller extension for pervasive multi-client user interfaces', *Personal Ubiquitous Computing* **9**(2), 100–107.

Schärli, N., Ducasse, S., Nierstrasz, O. and Black, A. [2003], Traits: Composable units of behavior, *in* 'European Conference on Object-Oriented Programming', pp. 248–274.

Schmidt, D. C. [1999], 'Why software reuse has failed and how to make it work for you'.
URL: *http://www.cs.wustl.edu/ schmidt/reuse-lessons.html*

Schmidt, D. C. and Buschmann, F. [2003], Patterns, frameworks, and middleware: Their synergistic relationships, *in* '25th International Conference on Software Engineering', pp. 694–704.

Schommer, I. and Broschart, S. [2009], *SilverStripe: The Complete Guide to CMS Development*, Wiley.

Schwabe, D., Esmeraldo, L., Rossi, G. and Lyardet, F. [2001], 'Engineering web applications for reuse', *IEEE Multimedia* (8), 20–31.

Schwabe, D. and Rossi, G. [1998], Developing hypermedia applications using OOHDM, *in* 'Workshop on Hypermedia development Process, Methods and Models', pp. 116–128.

Schwaber, K. and Beedle, M. [2001], *Agile Software Development with Scrum*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Shaw, M. and Clements, P. [2006], 'The golden age of software architecture: A comprehensive survey', *IEEE Software* **23**(2), 31–39.

Shaw, M. and Garlan, D. [1994], Characteristics of higher-level languages for software architecture, Technical Report CMU-CS-94-210, Carnegie Mellon University.

Singh, I., Stearns, B. and Johnson, M. [2002], *Designing Enterprise Applications with the J2EE(TM) Platform*, 2 edn, Prentice Hall.

Smith, J. [2009], 'WPF apps with the Model-View-ViewModel design pattern', *MSDN Magazine* .
URL: *http://msdn.microsoft.com/en-us/magazine/dd419663.aspx*

Spell, B. and Gongo, G. [2000], *Professional Java Programming*, Wrox Press Ltd.

SproutCore Documentation [n.d.].
URL: *http://wiki.sproutcore.com*

Sutter, H. and Alexandrescu, A. [2004], *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, Addison-Wesley.

Sutton, S. M. and Rouvellou, I. [2002], Modeling of software concerns in cosmos, *in* 'AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development', ACM, New York, NY, USA, pp. 127–133.

Taivalsaari, A. [1996], 'On the notion of inheritance', *ACM Computing Surveys (CSUR)* **28**(28), 438–479.

Taylor, M. J., McWilliam, J., Forsyth, H. and Wade, S. [2002], 'Methodologies and website development: a survey of practice', *Information and Software Technology* (44), 381–391.

Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, Jr., E. J. and Robbins, J. E. [1995], A component- and message-based architectural style for gui software, *in* 'ICSE '95: Proceedings of the 17th international conference on Software engineering', ACM, New York, NY, USA, pp. 295–304.

TYPO3 wiki.Resource [n.d.].
URL: *http://wiki.typo3.org*

Venners, B. [1998], 'Composition versus inheritance: A comparative look at two fundamental ways to relate classes'.
URL: *http://www.artima.com/designtechniques/compoinh.html*

Venners, B. [2005], 'Design principles from design patterns: A conversation with Erich Gamma'.
URL: *http://www.artima.com/lejava/articles/designprinciples4.html*

Wake, W. C. [1998], 'Growing frameworks in Java'.
URL: *http://xp123.com/wwake/fw/*

Whitehead, E. J. [2002], 'A proposed curriculum for a masters in web engineering', *Journal of Web Engineering* (1), 18–22.

Williams, P. [2007], 'Presentation patterns - Autonomous View'.
URL: *http://blogs.adobe.com/paulw/archives/2007/09*

Zamani, B., K, S. and Butler, G. [2008], A pattern language verifier for web-based enterprise applications, *in* 'MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems', Springer-Verlag, pp. 553–567.

Zambon, G. and Sekler, M. [2007], *Beginning JSP, JSF & Tomcat Web Development: From Novice to Professional*, Apress.

Zandstra, M. [2007], *PHP Objects, Patterns, and Practice*, 2 edn, APress.

Zimmermann, O., Zdun, U., Gschwind, T. and Leymann, F. [2008], Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method, *in* 'WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)', IEEE Computer Society, pp. 157–166.