Linnæus University
School of Computer Science, Physics and Mathematics

Degree Project

Towards a Classification of Design Patterns for Web Programming Based on Analysis of Web Application Frameworks

Joanna Juziuk
2011-06-21
Subject: Computer Science
Level: Bachelor
Course code: 2DV00E

Abstract

The evolution of WWW leads to continuous growth of demands that are placed on web applications that results in creating sophisticated web architectures. To minimize the complexity behind their design, software frameworks were introduced. There are hundreds of web frameworks, hence the choice of the right framework can be seen as searching for the holy grail.

This thesis investigates the possibility of creating and validates usefulness of a classification scheme which organizes well-known object-oriented design patterns found in popular web frameworks: Apache Struts, Ruby on Rails, CakePHP and Zend Framework. The proposed classification scheme is based on two criteria: purpose and scope. The classification of such patterns that capture design rationale behind the decisions and best practices, is potentially important for building or restructuring a generic web framework, for capturing expertise knowledge and for orientation purposes in the problem domain - web engineering.

The methodology used in this thesis is based on case studies and the identification of design patterns in web frameworks uses manual approaches. The results revealed popular design patterns in web frameworks and that the proposed classification scheme in a form of a 2D matrix must be refined, because relationships among design patterns in web frameworks are important and have a tendency to be formed as complex hierarchies. It is proposed to use a classification scheme in a form of a map or a tree when refining the scheme.

Keywords

design patterns, web frameworks, Rails, Zend Framework, Struts, CakePHP, web programming, classification

List of Figures

List of Tables

Contents

1 Introduction

Chapter 1 highlights the problem and its root causes to the reader. It also presents research questions, aim of the thesis and its scope.

1.1 Problem

In general, the problem is rooted in the information overload problems that were intensified by the rapid and chaotic grow of the WWW (World Wide Web), especially in the Web 2.0 era (Bawden and Robinson, 2009).

Information overload problems occur when the amount of the information is greater than a human can handle, so it negatively affects decision making when selecting right sources of information (Bergamaschi et al., 2010). Moreover, this "paradox of choice", as Bawden and Robinson (2009) characterize it, can make people feel confused, overwhelmed and powerless. Bawden and Robinson (2009) point out that those issues are considered as fundamental in Human-Computer interaction and Information Management domains and they will be challenged over the next decades. In the field of software engineering and web development, the information overload problems concern the quantity and the quality of information created with overproduced technologies and software frameworks that are used to built web applications and solve common web programming problems.

The evolution of WWW leads to continuous growth of demands that are placed on web applications and creation of complex constructions behind their designs. Because web applications, especially designed as B2B (business-to-business) have come to be more sophisticated and software developers have become less experienced and qualified, the Web 2.0 movement has popularized the concept of rapid development using software frameworks - abstractions in form of software libraries or classes providing generic functionality to minimize the problem (Jazayeri, 2007). Nevertheless, the panacea of this problem is problematic itself since there were developed more than a hundred web applications frameworks (fig. 1.1). Hence the choice of the right framework can be seen as searching for *the holy grail* in the web engineering (Shklar and Rosen, 2009).

Figure 1.1: Web frameworks examples

Instead of trying to find the best framework, it would be more reasonable to focus on creating a generic, environment and language independent web framework or a framework of web framework that would be the solution of the overload information problems in this area. Since a web framework is built from components and those are built using best techniques described as design patterns, a generic framework could be built on patterns found in popular web frameworks. The amount of identified patterns could potentially arise information overload problems, so that to estimate which patterns are essential to build a generic framework, they must be first classified and a proper classification scheme must be proposed. As Bergamaschi et al. (2010) claim, classifying is among the useful techniques in overcoming the information overload problems, like ranking, filtering or grouping.

Moreover, the problem is associated with the existing classifications of software design patterns. There are many different classification schemes that are not standardized and strongly dependent on the author's viewpoints. Many of the classifications are focused on desktop-based programs and were not adapted to the web application's problems. Therefore, building such classification scheme could be a significant input in adapting, organizing and simplifying current schemes.

To conclude, the overall problem concerns overcoming the information overload problems by logically classifying and organizing information. The classification of design patterns is potentially important for building or restructuring a generic web framework, for learning and understanding complex software designs, for capturing expertise knowledge and for orientation purposes in the problem domain - web programming. It can be compared in terms of achieving simplicity, usability and comprehension to Carl Linnaeus biological taxonomy created 275 years ago.

1.2 Purpose and research questions

The goal of the thesis is to identify, then validate classification of best techniques used when for recurring problems in web programming. The study addresses the following questions:

- Which web application frameworks are popular?
- Which design patterns can be found in the popular web frameworks?
- Is the classification when applied to the pool of found patterns, useful for learning purposes (simple and easy to understand)?

1.3 Scope

This thesis is focused on the classification of design patterns found in server-centric web frameworks modeled with object-oriented paradigm and designed to support dynamic page development. Due to the rapid grow of web frameworks and limited time of the research, the report is restricted to: Ruby on Rails, CakePHP, Zend Framework and Apache Struts. The choice was supported by current trends that are described in more detail in the section 4.1.

The study does not cover Rich Internet Applications and technologies, like Adobe Flash or Microsoft Silverlight as they are proprietary, non-native to browser's environments and require sandbox plugins to work. In addition, those vendor-specific technologies, extending current web standards, will be replaced by HTML 5 in the future according to World Wide Web Consortium plans (Vaughan-Nichols, 2010).

1.4 Outline

The thesis is organized by six chapters. Chapter 1 highlights the problem and its root causes to the reader. It also presents research questions, aim and its scope. Chapter 2 offers the literature survey and is divided into three sections: first section describes background theory behind the design patterns, existing approaches to the classification, documentation and identification of design patterns in software engineering, the second defines a web application and the third section outlines the concept of software framework. Chapter 3 focuses on the description of the methodology. Chapter 4 contains the realization part that includes four sections: the first section contains results from searching for popular web frameworks, in the second section, the proposed classification scheme is described. Architectures of chosen frameworks and results from identification of design patterns are covered in section 3. Chapter 5 presents validation of the classification scheme on case studies, discussion and comparative analysis of the results. Chapter 6 ends the thesis with a discussion on future research.

2 Background

Chapter 2 offers the literature survey and is divided into three sections: first section describes background theory behind the design patterns, existing approaches to the classification, documentation and identification of design patterns in software engineering, the second defines a web application and the third section outlines the concept of software framework.

2.1 Design patterns and pattern languages

2.1.1 Definition

Designers in every domain are faced with recurring problems. One way of solving such problems is to use design patterns that contain descriptions of problems with all documented way of solving them. This manner of perceiving problem solving was originated in 1970s by an architect - Christopher Alexander.

Alexander (1979, p.28) started to treat patterns as design concepts that should have a *quality without a name* (QWAN) - an unexplainable quality which gives some sense of rightness in the design. According to Alexander (1979) patterns should consist of three layers (fig. 2.1). First layer embodies a recurring and arising problem. A problem arises in a situation known as a context that is the second layer. Third layer is the solution that is a well-known and proven solution to the problem.

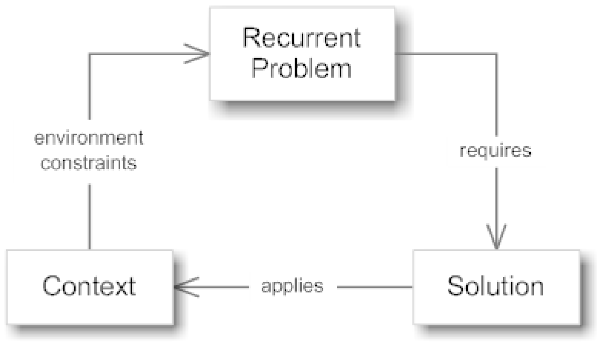


Figure 2.1: Pattern anatomy

A pattern language is a mean to organize design patterns and as natural languages it should have its own grammar and vocabulary. The main function of such language is to show grammatical and semantic relationships among patterns and to depict the decisions that were taken during the design phase (Alexander et al., 1977). Pattern languages have forms of hierarchy trees, maps or graphs that group related patterns. For example, in the fig. 2.2, the pattern language map is in a form of a directed acyclic graph where: A and B patterns must be applied before C pattern and applying D pattern shows that C pattern should be applied before.

The concept of pattern languages and design patterns was adapted to many disciplines, particularly the idea was promising to software architects, because it could improve communications among developers and they could make better decisions that would avoid time wasting on re-inventing existing solutions. In the context of software engineering, a pattern can be defined as:

"(...) a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves" (Gabriel, cited in Alur et al., 2003, p.10).

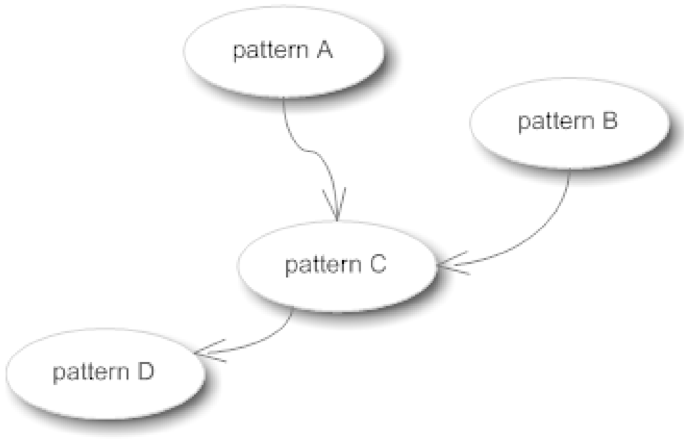


Figure 2.2: Pattern map

Software patterns are strongly bounded with the levels of abstraction and development phase, thus at least seven categories can be distinguished:

- design patterns
  - software design and architecture patterns
  that have focus on subsystems and components,

- process patterns
  - patterns concerning processes and structures
  in organizations involved in software development,

- analysis patterns
  - patterns that occur when looking for requirements,

- re-engineering patterns
  - patterns that are used when changing software
  in order to get better quality and maintainability,

- performance patterns
  - patterns that focus on improving software performance,

- domain-specific patterns
  - patterns that belong to the specific domain;

- anti-patterns
  - patterns that are counterproductive.

This study is focused mainly on design patterns which are always pragmatic, recurring and generative. Those types of software patterns are always connected with real, concrete solutions that were once adapted, not with new, not implemented nor tested ideas. The main characteristic of design patterns is also that their implementation can vary, because they are not meant to be finished designs, but generic.

Furthermore it is not possible to describe every problem with patterns that are not applicable to every context. In addition, it is important to emphasize that patterns are not *silver bullets* for software development as Brooks (1995, p.177) states:

> "there is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity".

However, cataloged patterns as a huge library of expertise and a carrier of history of successful solutions, can be effective tool for learning and teaching.

2.1.2 Example

An example of a design pattern can be the Vistor pattern. It is stated that the purpose of this pattern is to "represent an operation to be performed on the elements of an object structure" (Gamma et al., 1995, p.331). Diagrams: fig. 2.3 and 2.4 depict the mechanism behind the pattern. Following objects are created: a ConcreteVisitor object that implements a visit method with declared as argument, a ConcreteElement, and the ConreteElement with an accept operation that matches the visit method in the Visitor. Thus when an element has a visit, it calls the Visitor's method that corresponds to its class.

Visitor	Abstract class of interface that defines visit operation for each class of ConcreteElement in the object structure
ConcreteVisitor	Concrete class that provides implementation of every method declared by Visitor abstract class
Element	Abstract class that defines an Accept operation which should take a Visitor as an argument
ConcreteElement	Concrete class that provides implementation of an Accept method that takes a Visitor as an argument
ObjectStructure	Provides an interface that helps the visitor to enumerate a collection or composite of elements to be visited by the visitor

Table 2.1: Design pattern example - Visitor - elements

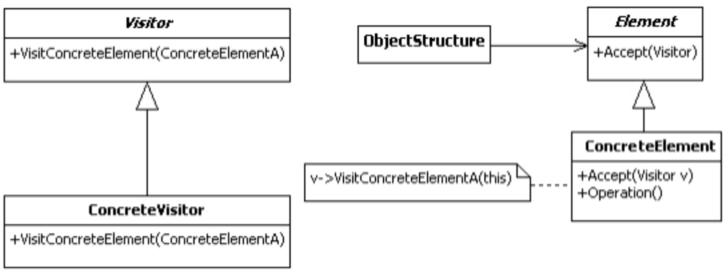

Visitor
+VisitConcreteElement(ConcreteElementA)
ObjectStructure
Element
+Accept(Visitor)
ConcreteElement
+Accept(Visitor v)
+Operation()
v->VisitConcreteElementA(this)
ConcreteVisitor
+VisitConcreteElement(ConcreteElementA)

Figure 2.3: Design pattern example - Visitor - class diagram

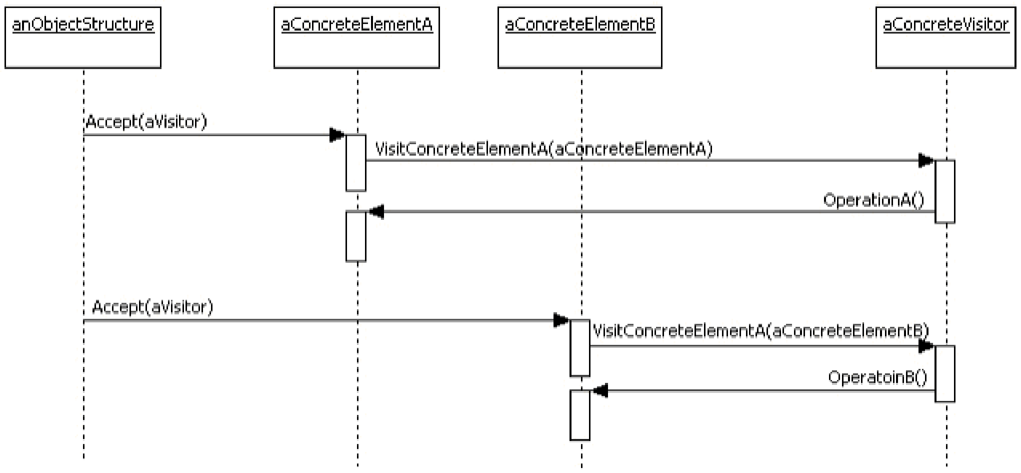
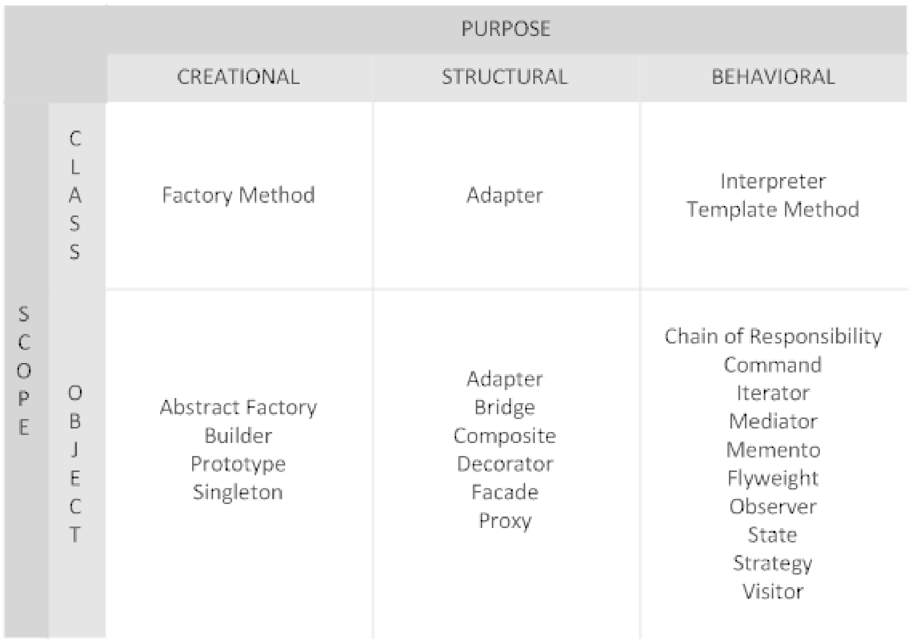


Figure 2.4: Design pattern example - Visitor - sequence diagram

One of the main advantage of the Visitor pattern is that it allows to define a new operation without changing the original classes of the operated on elements. For example, it can be useful in situation when we want to perform a set of test cases on the given set of classes without their modification. Hence among Visitor's positive consequences is ease when adding new methods or operations - a new visitor is only needed when extending functionality over object structure. Although Visitor pattern can have many highlights, it can have some negative effects. For example, it breaks encapsulation that is indispensable for object-oriented design.

2.1.3 Approaches to classifications of design patterns

Since the number of design patterns increased, there has been an urgent need to develop efficient techniques and methods to organize them.



		PURPOSE		
		CREATIONAL	STRUCTURAL	BEHAVIORAL
SCOPE	CLASS	Factory Method	Adapter	Interpreter Template Method
	OBJECT	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Figure 2.5: GoF matrix

GoF (Gang of Four) classification is the first and widely-known classification of design patterns. The pattern catalog contains 23 design patterns that were previously undocumented (fig. 2.5). Gamma et al. (1995) proposed to use two dimensional classification based on criterion: scope and purpose. Purposes concern the problem categories. Three types of purposes were distinguished: creational that concerns object creation issues, behavioral that is related to managing responsibilities in classes and structural that covers structures or compositions created by objects/classes. Pattern's scope is limited to class (class inheritance) or object (object composition) representations. In this classification, a pattern might have two different scopes, for example Adapter.

Zimmer (1995) proves that the same pool of patterns - the GoF catalog, can be organized using other approaches. Instead of two dimensional matrix, there is a pattern map that reflects relationships among design patterns. In this classification there are three types of connections: "uses", "is similar to" and "can be combined with". For example, in the fig. 2.6, Model View Controller uses Composite pattern, Iterator pattern is similar to Visitor and Composite pattern can be combined with Visitor.

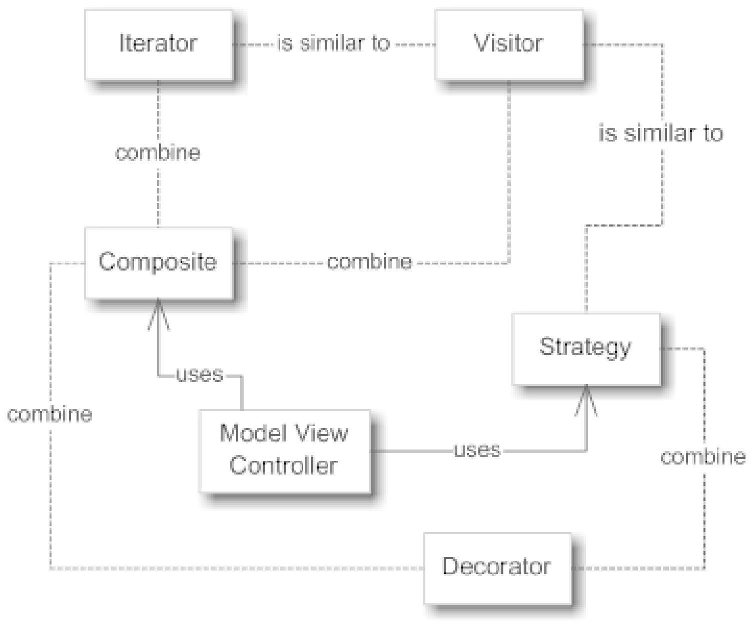


Figure 2.6: Zimmer classification example

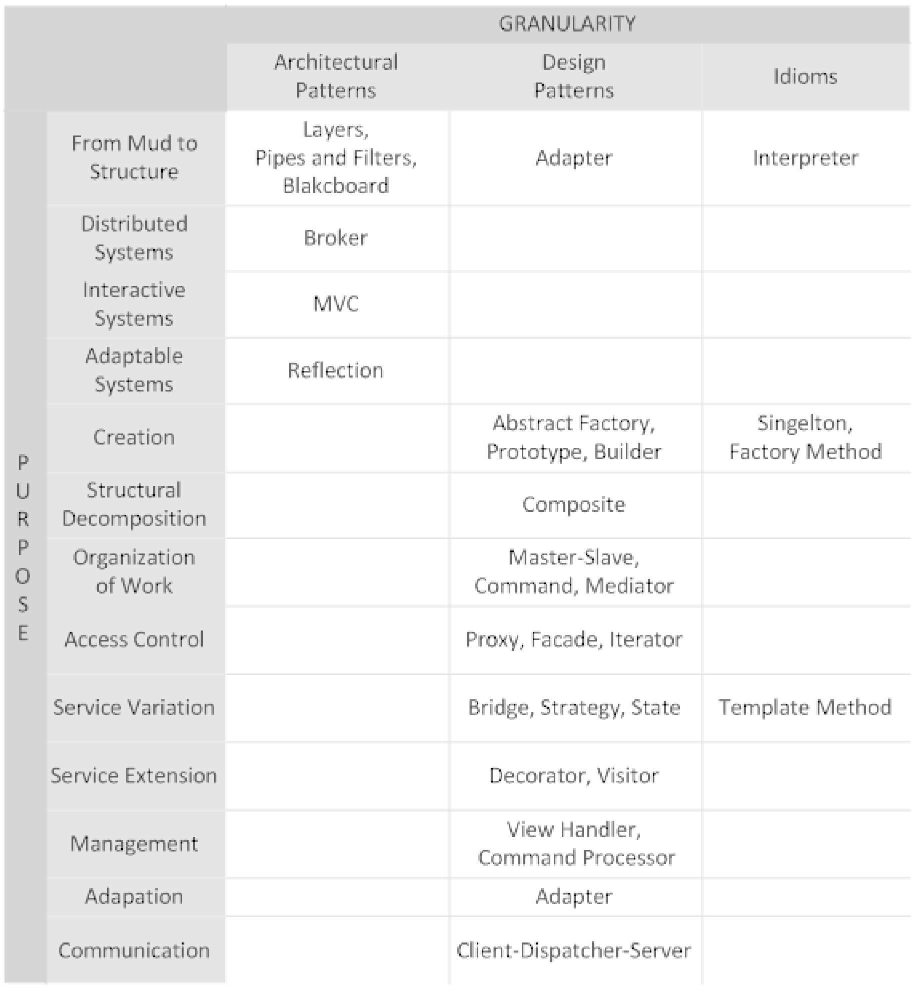
Granularity, functionality and structural principles are key issues to organizing software patterns by Buschman. As GoF matrix the Buschman classification is also two dimensional (fig. 2.7), but uses granularity and purpose criteria.

The purpose criterion have 14 categories that are less abstract and more pragmatic than those proposed in GoF classification. For example, "Interactive systems" category contains patterns that are related to human-computer interactions. The granularity criterion is defined by a pattern category that is connected with the pattern's abstraction level. Buschmann (1996) defines three pattern levels. First level is for architecture where the author distinguished architectural styles - patterns for organizing the entire system. As Buschmann (1996, p.11-12) states "an architectural pattern expresses a fundamental structural organization schema for software systems; it provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them". Table 2.3 exemplifies architectural styles.

Repositories	databases, programming environments
Process Control	AirCraft/SpaceCraft flight control systems, power stations
Pipes and filters	UNIX shell commands, compilers
Layered Systems	virtual machines: JVM, OSI model, operating systems

Table 2.3: Architectural styles

Second level of pattern's abstraction concerns design patterns that are patterns which are principle, paradigm dependent and that are mainly described in GoF classification. The third level of pattern's abstraction is related to implementation and covers idioms that are low level patterns which are language, technology dependent. Among examples of idioms are: CORBA (Common Object Request Broker Architecture) patterns or J2EE (Java Platform, Enterprise Edition) patterns.



		GRANULARITY		
		Architectural Patterns	Design Patterns	Idioms
PURPOSE	From Mud to Structure	Layers, Pipes and Filters, Blakcboard	Adapter	Interpreter
	Distributed Systems	Broker		
	Interactive Systems	MVC		
	Adaptable Systems	Reflection		
	Creation		Abstract Factory, Prototype, Builder	Singelton, Factory Method
	Structural Decomposition		Composite	
	Organization of Work		Master-Slave, Command, Mediator	
	Access Control		Proxy, Facade, Iterator	
	Service Variation		Bridge, Strategy, State	Template Method
	Service Extension		Decorator, Visitor	
	Management		View Handler, Command Processor	
	Adapation		Adapter	
	Communication		Client-Dispatcher-Server	

Figure 2.7: Buschmann matrix example

For large scale Enterprise applications, we can distinguish two classifications - catalog proposed by Martin Fowler in the book: ”Patterns of Enterprise Application Architecture” and J2EE blueprints. Both classifications are one-dimensional. Fowler

(2003) proposes a purpose as a criterion and classifies patterns into categories: Organizing Domain Logic, Mapping to Relational Databases, Web Presentation, Concurrency, Session State, Distribution Strategies etc. J2EE blueprints contains classification that categorizes patterns using architectural tiers: presentation, business and integration. Moreover, this catalog contains a pattern map depicting relationships among patterns, that is similar to Zimmer map of GoF patterns. PoEAA patterns are generally technology independent and directed to different platforms in difference of J2EE patterns were originally adapted to J2EE technology (Alur et al., 2003). Although there are some patterns in J2EE blueprints that are strictly domain dependent, for example Value Object, many of them can be adapted in other technologies, like: Front or Application Controller. Those patterns are usually built on patterns found in GoF classification, e.g. Session Facade is built using Facade pattern.

To sum up, a wide range of classification schemes has been proposed, but none of them is used in correlation with some official standard.

2.1.4 Approaches to document design patterns

The purpose of formal documentation of design patterns is to provide simple graphical and textual descriptions that would make patterns traceable in the design and possibly automate all actions related. Specifications of patterns can be produced either manually or automatically and placed in a source code, in a separate design document or in a diagram depicting system's design. In separate documents we can distinguish to main document formats: using plain text and figures or documents based on markup languages.

Documentation in the source code can be in a form of comments or annotations (metadata). To improve traceability of patterns in comments, Meffert (2006) and Torchiano (2002) propose to use special extensions to standardized documenting frameworks, like Javadoc. Patterns properties could be described using specially defined tags, for example: @intent, @problem, @drawback, @need (Meffert, 2006), or @pat.name, @pat.role, @pat.task, @pat.use (Torchiano, 2002). Similar approach, but more automatic, is suggested in DPDOC that is a prototype CASE (Computer-aided software engineering) tool for APPLAB and supports documenting patterns via annotating the source code, rule checking and automatic role derivation. DP-DOC can be adapted to any grammar, but only supports seven GoF patterns which it treats as language constructs (Cornils and Hedin, 2000).

Gamma et al. (1995) introduce in GoF catalog separate documentation of patterns that was done manually and uses plain text with Object Modeling Technique (OMT) charts. This documentation consists of following sections: 'Pattern Name and Classification', 'Intent' (purpose of the pattern), 'Also Known As', 'Motivation' (Forces), 'Applicability', 'Structure', 'Class and Interaction Diagram', 'Participants', 'Collaboration', 'Consequences', 'Implementation', 'Sample Code', 'Known Uses' and 'Related Patterns' (Gamma et al., 1995).

PIML (Pattern Information Markup Language) is an example of a way of documenting patterns that is expressed in SGML (Structured Generalized Markup Language) which is a XML-based markup language that was standardized as ISO 8879:1986 (Ohtsuki et al., 1997). A document describing a design pattern consists of three parts: plain text explanations, structure information as class diagram, behavior information as pseudo code that would support OMT charts (fig. 2.8).

```
<pattern name= "Memento">
   <intent> </intent>
   <motivation> </motivation>
   <applicability> </applicability>
   <consequences> </consequences>
   <implementation> </implementation>
   <sample_code> </sample_code>
   <known_uses> </known_uses>
   <related_patterns> </related_patterns>
   <structure> </structure>
</pattern>
```

Figure 2.8: PIML structure example

Dietrich and Elgar (2005) select OWL (Web Ontology Language) as another markup language that can be used to document design patterns. OWL is a markup language that has semantics based on description logic. It is an extension to RDF (Resource Description Framework) which is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata data model. According to RDF Working Group (2004), RDF "integrates a variety of applications from library catalogs and world-wide directories to syndication and aggregation of news, software, and content to personal collections of music, photos, and events using XML as an interchange syntax". In this approach design patterns are represented as resources identified by uniform resource identifiers (URI) which makes the documentation more open and extensible.

LePUS (LanguagE for Patterns Uniform Specification) is a visual language based on defined symbols and Prolog logic constructs. According to Eden et al. (1998), LePUS supports formal documentation of design patterns, specifications' validations and proving relations among design patterns.

Another visual language for modeling design patterns is DPML (Design Pattern Modeling Language) created by Mapelsden et al. (2002). Its main advantage is that is compatible with UML (Unified Modeling Language) that is a standard modeling language for object-oriented software. DPML links design patterns participants with UML diagrams elements.

UML lacks support for visualizing design patterns. The closet diagram to depict design patterns is a collaboration diagram which role is to illustrate relations and interaction between objects. Hence extensions to UML must be introduced to visualize fully design patterns. Dong and Yang (2003) introduce UML Profile that defines new stereotypes, constraints, and tagged values that would support specification of design patterns.

Another mean to describe design patterns is Balanced Pattern Specification Language (BPSL) (Taibi and Chek Ling Ngo, 2003). BPSL notation is simple and uses First Order Logic (FOL) for structural aspects and Temporal Logic of Actions (TLA) for behavioral properties. BPSL can specify rules that are behind UML meta-model.

Automatic or semi-automatic documentation of patterns can be obtained when a developer uses a design-decision-detection tool while designing the system. Such tools allow to pick certain patterns and glue it with the design. An example of such tool is POD (Pattern-Oriented Design) tool that supports visually pattern composition using three view levels: Pattern-Level, Pattern Interfaces, Detailed Pattern-Level. The outcome of the tool is an UML class diagram (Yacoub et al., 2000).

Automatic approaches can guarantee high level of quality of documentation when compared to manual approaches, because every change in the code will be automatically updated in the documentation, therefore the design can never be outdated. Furthermore, manual documenting is always a standalone task for developers and it can be not prioritised when the project is close to the deadline.

2.1.5 Approaches to identify design patterns

The goal of identifying design patterns is to improve understandability and maintainability of the software. Information about the patterns can be found directly from the software documentation if a designer included detailed design decisions or used special CASE tools for documenting design patterns. In case of open source software that often lacks standardized artifacts, the information about used design patterns can be also found on official forums or users' groups etc. The limitations in pattern's detection in documention are poor quality of source code documentation or outdated design artifacts. If the documentation makes patterns visible and traceable in the code, then pattern detection can be done automatically.

Design patterns can be also extracted from the source code manually or semi-automatically or automatically using reverse engineering techniques. Manual pattern identification can be characterized by tedious source investigation activities, like looking for classes that follow certain naming or coding conventions, for example an adapter's class can be named DBAdapter. Semi-automatic approaches to detect design patterns include mainly rule-based mechanisms, for example Niere et al. (2002) present an approach to detect patterns based on static analysis of the source code that is represented as ASG (Abstract Syntax Graph) and graph transformation rules. As for the automatic approaches, Dori et al. (2005) name four ways: identifying key structures, searching for class structures, fuzzy logic based search and metrics based search.

First approach is characterized by creating a database with knowledge about a pattern that includes pattern's characteristics and features that this pattern should not manifest - positive and negative criteria. En exemplary prototype using identifying minimal structure approach with positive and negative criteria, that includes all GoF patterns, is presented by Philippow et al. (2005). Dori et al. (2005) propose to create a knowledge base using OPM (Object Process Methodology) that will hold meta-models of GoF patterns with positive and negative criteria. Second approach compares class structures with a pattern class structure. Third approach incorporates data mining, artificial intelligence, tree or graph algorithms. Most of the algorithms are based on traversing and filtering a graph that represents program structure. The limitation of using these algorithms is incapability of sufficient extraction or interpretation of the purpose from the code. Apart from static analysis of source code, dynamic analysis of software runtime can be performed to detect design patterns. Wendehals (2003) introduces dynamic analysis as a validation of static analysis results. The last approach concerns software quality metrics, like depth of inheritance tree or number of children, that can be calculated for a given pattern. Results and given values are checked if they closely match (Antoniol et al., 1998). Gueheneuc et al. (2004) call this kind of algorithm - fingerprinting of design patterns where a fingerprint is a value of specified metrics.

Using reverse engineering techniques in identification of design patterns is more time- and cost-efficient than using manual approaches. However automated approaches cannot guarantee accuracy and exact pattern identification, because some patterns are very similar in structure or in behavior.

2.2 Web application

2.2.1 History

In the beginning of WWW that was launched as a CERN (The European Organization for Nuclear Research) project, it resembled a static bank of articles and other texts that allowed to gain access, gather and associate information by scientists. The information was presented as HTML (HyperText Markup Language) documents that were requested using HTTP (Hypertext Transfer Protocol). The requests were made by client computers using web browsers, sent to the server computers that stored the documents and sent back responses to the clients. As the popularity of the WWW was quickly growing, new non-scientific users put demands on extra requirements, like communication, entertainment, shopping etc, resulting that new server side scripting technologies were created, for example ASP (Active Server Pages), JSP (JavaServer Pages) or PHP (PHP: Hypertext Preprocessor) that allowed previous requests of static text pages became invocations of concrete applications on the server side - web applications (Jazayeri, 2007).

2.2.2 Overview

A web application or a webapp is a client-server application that operates in a distributed environment, for example in Internet and which client is a web browser. Physical implementation of a webapp architecture is multi-tier.

Typically it consists of three layers: client (presentation), business logic and data access tier. On the client tier, following technologies are used: HTML, XML, CSS (Cascading Style Sheets), Flash and JavaScript. On the business logic tier, server-side scripting technologies, like PHP, JSP, ASP.NET, CGI and ColdFusion Markup Language, are responsible for programming the logic. On the database tier, there are database management systems (DBMS) or relational database management systems (RDBMS) (Shklar and Rosen, 2009).

Every web application conforms to the Separation of Concerns principle which states that presentation aspect of the application should be isolated from the logic, therefore it implements Model-View-Controller design pattern or its variants. Model contains the logic, View presents the Model, and Controller receives requests, instructs the View and updates the Model. MVC's benefit is reduction of the complexity and improvement of re-usability and maintainability.

Fig. 2.9 depicts the mechanism behind MVC. A client sends a HTTP request to the server where it is received by a Controller (1). Controller sends a request to the Model for the asked data (2). The Model sends a query to the database (3) which returns to the model requested data (4). The Model forwards the data to the Controller (5). It can add some logic to it if necessary. The Controller instructs the View to represent the data (6). Finally, the View sends the response to the client (7).

The main advantages of web applications over desktop applications are cross-platform compatibility, relatively small size since they require almost no disk space,

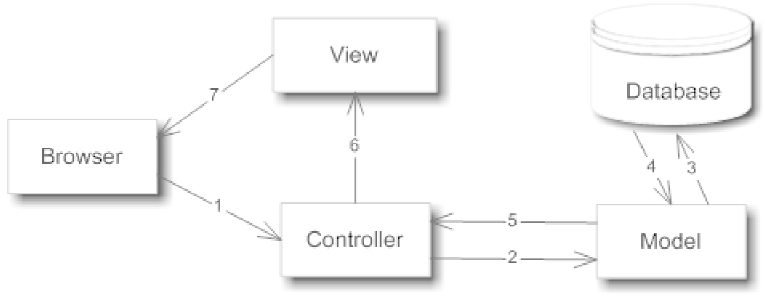


Figure 2.9: MVC architecture

no installation procedure and constant availability in the Internet. However, the Internet connection and high security risks are disadvantages. Furthermore, desktop applications have better means to provide better user experience in comparison with web applications that are limited by web browsers interfaces.

2.3 Framework

2.3.1 Definition

A software framework can be defined as:

> “a set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by sub classing and composing the instances of the framework classes” (Gamma et al., 1995, p.360).

Frameworks’ architectures can be considered using two views. Johnson (1997) claims that ”design patterns are the architectural elements of frameworks”. As design patterns, frameworks capture best reusing techniques and strategies for solving problems. In contrast to design patterns which are more conceptual, frameworks are focused on implementations and designs. Therefore they can be viewed as sets of domain specific, concrete forms of design patterns.

Pree (1994) looks at frameworks’ architectures from the perspective of hot and frozen spots. Frozen spots are those basic components of a framework that became frozen or unchanged while those parts that are extended by a developer adding new, project-specific functionality are represented as hotspots (fig. 2.10).

Frameworks tend to be ”extensible skeletons” of applications with code that has a default behavior and it is non-modifiable, providing only options to extend or override its functionality via inheritance, in difference of libraries that are self-contained, have well-defined operations and are called from a code. Furthermore, frameworks are the main coordinators that invoke the code. This characteristic is called Inversion of Control and is based on the Hollywood Principle (”don’t call us, we’ll call you”). As Fowler (2005) emphasis it is essential to the concept of frameworks. For example, when en event occurs, the flow control is inverted and only the inversion control in the framework calls back the classes or modules.

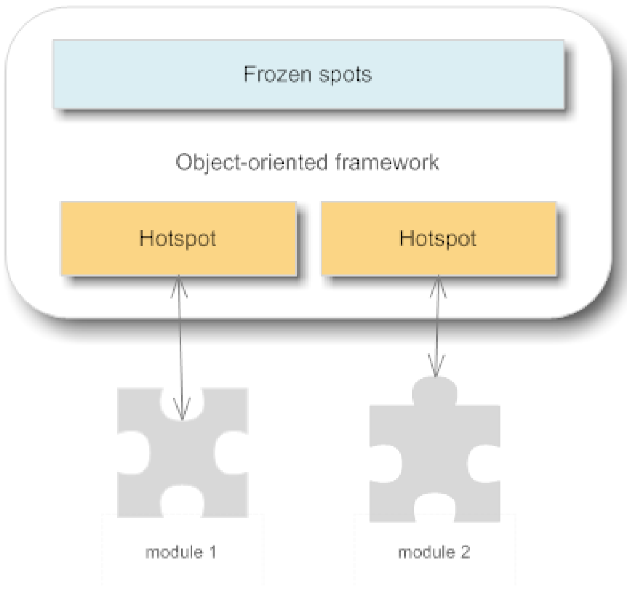


Figure 2.10: Framework architecture

Using a framework benefits to improved re-usability and enhanced modularity. It leads to better software quality and reduces the effort needed to maintain the software, because it localizes the effect of changes in the implementation and design. All frameworks' interfaces contribute to the re-usability of the software since their generic components can be defined and used to create other applications. Thus frameworks' re-usability improves developers' productivity and software's interoperability, quality and performance. However, frameworks have complex designs and using it requires time for learning their APIs by developers. In the case of abandoning the framework or changing the technology in the future development, time and cost invested in getting familiar with the framework would be wasted.

2.3.2 Web application framework

A web application framework is a combination of two terms - a web application and a software framework. It is a software framework that should contain reusable components associated with common web applications features. A typical web application framework should contain modules responsible for: templating system, security, caching, web services or database access.

Web application frameworks are related to web content management systems (WCMS) that are web applications which allow to manage and share documents collaboratively by many users. Some of web application frameworks and WCMS are focused on creating an API for a specific content management system therefore there can be defined as content management frameworks (CMF). An example of such hybrid is Drupal that is both CMS and flexible CMF (Buytaert, 2010).

3 Method

Chapter 3 focuses on the description of the methodology used in this thesis - strategies for literature study, classification and identifiaction procedures.

3.1 Overview

The method of problem solving used in this thesis has as a natural science approach. This approach is characterized by a pragmatic scheme: making observations, creating new theories on this, performing experiments, validating the theories and making conclusions.

3.2 Literature study

Observations are gathered from the literature study with a trusted-review strategy in the domain related to software engineering and web development. The most important contributions to those fields, found by keywords: design patterns classifications and software frameworks, would be included. Additionally, the scientific databases: ACM Digital Library, IEEE Xplore, Springerlink and DiVA portal will be searched. The relevance criterion is the fitness with the purpose of the study. Reviewing the literature will be a solid foundation for the further investigation which involves creating a proper classification scheme.

3.3 Classification procedure

Classification is a fundamental process that aims to organize things or concepts and it is a practice of taxonomy which is a science that includes identification, grouping, filtering and naming activities. The main benefit of a classification is that things can be easily found without knowing their details which is very helpful when people are learning. The most widely known is Linnaean taxonomy that is a biological classification of living organisms. Unlike biological classifications that are often hierarchical and tree-structured, classifications in computer science primarily resemble a multi-dimensional-matrix. Most of CS classifications are topical or subject-oriented. The concept of topical classifications is rooted in Aristotle's philosophy that stated that everything has characteristic properties that can be used to subjective categorization (Taylor and Joudrey, 2009).

The process of identifying a classification scheme used in this thesis is illustrated in the fig. 3.1. In order to create a good taxonomy, requirements for the ordering must be established and on this basis, the draft classification scheme should be designed and developed. The next step involves the application of the classification scheme on patterns found by observations in chosen frameworks. If the classification scheme is not useful, then it must be in the refinement phase until it satisfies the requirements.

The choice of the frameworks for the case studies will be supported by qualitative research. Quasi-statistics of the web frameworks popularity factor will be gathered and the data will be presented in a form of charts. The shortcoming of case studies, as Kitchenham et al. (1995) claim, is that case studies are detailed, therefore they may not represent general results and it cannot be verified if this classification scheme could be applied to every other web framework. Finally, the research will lead to comparing results that will be interpreted and will determine the basis for drawing conclusions.

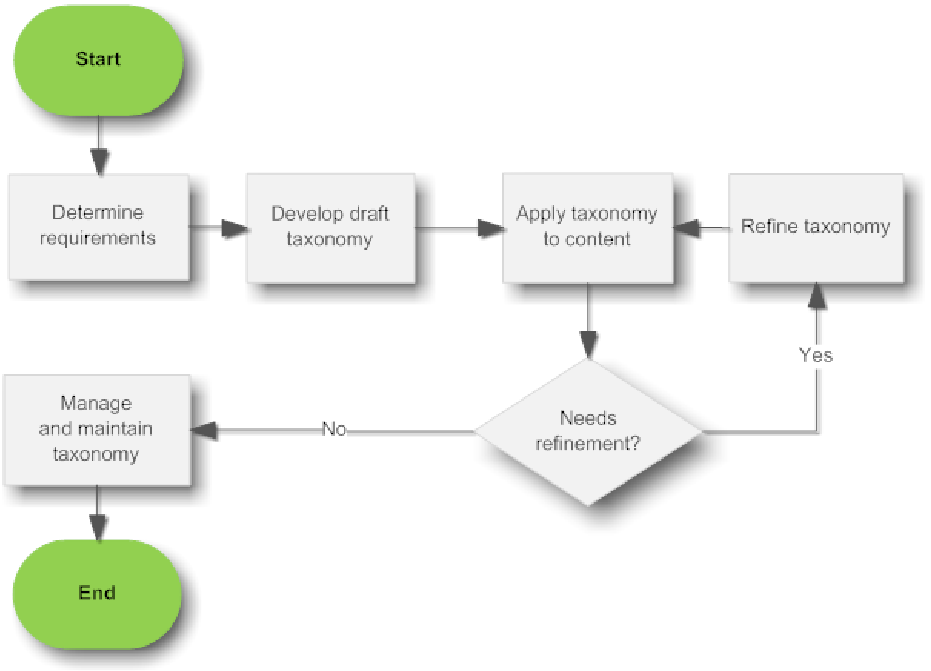


Figure 3.1: Flow chart of generic classification procedure

3.4 Identifying patterns

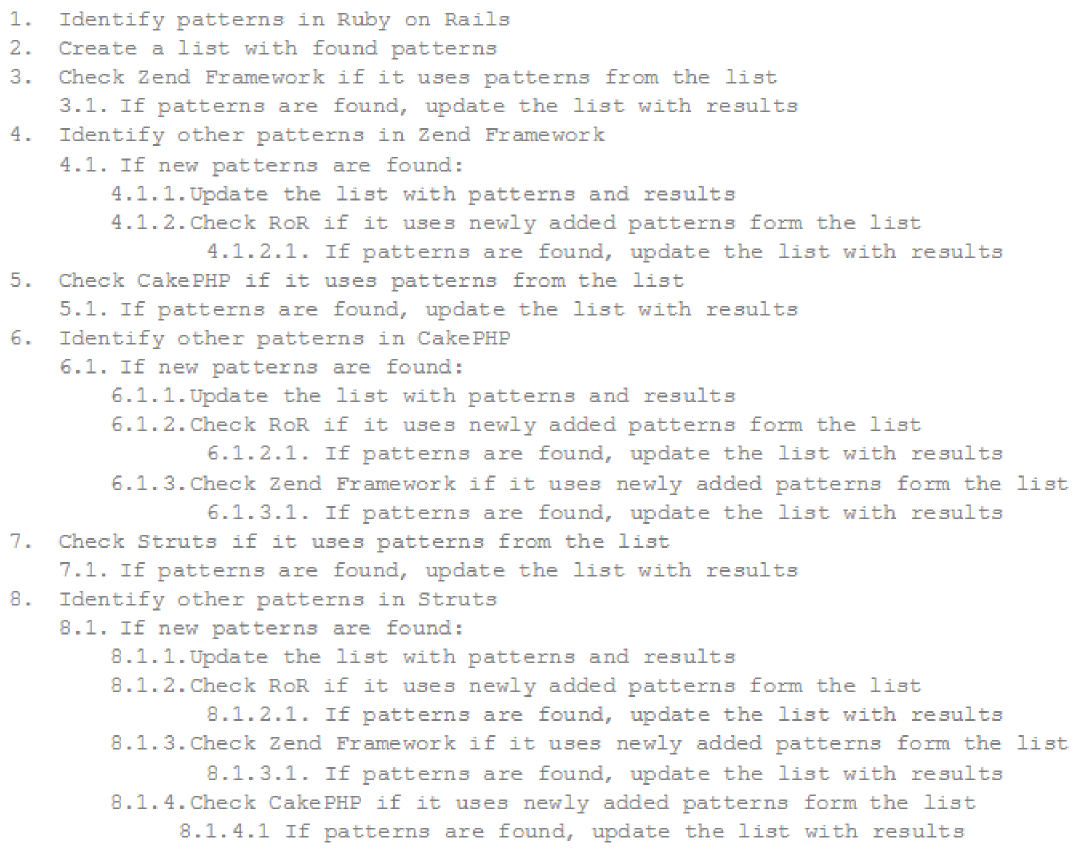
The search for the design patterns will only include well-known design patterns that were mentioned in the literature. Identifying patterns will concern only manual techniques since automatic approaches cannot guarantee sufficient accuracy in pattern detections. Other limitation is variety of design patterns in literature and programming languages, because it would make an automatic approach very complex to define.

The information about design patterns in the frameworks will be gathered from source codes, documentation, APIs (Application Programming Interfaces) available in the official frameworks' websites, forums and other related books. Pattern detection in the source will consist of finding special keywords related to a pattern, for example a pattern name. The evaluation of information found in separate documents will be checking if they exist in the source code. Process of identification method will be conducted according to defined in instructions (fig. 3.2)

shortcut	referenced book
GoF	Design Patterns: Elements of Reusable Object-Oriented Software
PoEAA	Patterns of Enterprise Application Architecture
J2EE	Core J2EE Patterns: Best Practices and Design Strategies

Table 3.1: Reference shortcuts

It involves creating a list with found patterns. This list will consist of following sections: 'Pattern' - pattern's name, 'Ref' - referenced where in the literature it was mentioned (tab. 3.1), framework's name, 'Context example' - where in the source code it was identified, 'Original category' - category of the pattern from the referenced literature.



```
1.  Identify patterns in Ruby on Rails
2.  Create a list with found patterns
3.  Check Zend Framework if it uses patterns from the list
    3.1. If patterns are found, update the list with results
4.  Identify other patterns in Zend Framework
    4.1. If new patterns are found:
        4.1.1.Update the list with patterns and results
        4.1.2.Check RoR if it uses newly added patterns form the list
               4.1.2.1. If patterns are found, update the list with results
5.  Check CakePHP if it uses patterns from the list
    5.1. If patterns are found, update the list with results
6.  Identify other patterns in CakePHP
    6.1. If new patterns are found:
        6.1.1.Update the list with patterns and results
        6.1.2.Check RoR if it uses newly added patterns form the list
               6.1.2.1. If patterns are found, update the list with results
        6.1.3.Check Zend Framework if it uses newly added patterns form the list
               6.1.3.1. If patterns are found, update the list with results
7.  Check Struts if it uses patterns from the list
    7.1. If patterns are found, update the list with results
8.  Identify other patterns in Struts
    8.1. If new patterns are found:
        8.1.1.Update the list with patterns and results
        8.1.2.Check RoR if it uses newly added patterns form the list
               8.1.2.1. If patterns are found, update the list with results
        8.1.3.Check Zend Framework if it uses newly added patterns form the list
               8.1.3.1. If patterns are found, update the list with results
        8.1.4.Check CakePHP if it uses newly added patterns form the list
              8.1.4.1 If patterns are found, update the list with results
```

Figure 3.2: Identification steps

4 Realization

Chapter 4 contains the realization part that includes four sections: the first section contains results from searching for popular web frameworks, in the second section, the proposed classification scheme is described. Architectures of chosen frameworks and results from identification of design patterns are covered in section 3.

4.1 Choosing frameworks for the analysis

Although there are many aspects worth considering when choosing a web framework, for example, context and purpose of the application, personal preferences of the developers, hosting requirements, scaling or license, the popularity factor is the most important. Well-liked frameworks are known to have supportive communities that benefits to: proper documentation that leads to better quality, better support for novice users, more extensions and plugins to the core functionality, quicker validation and defect removal that is equivalent with better security. To estimate which web frameworks are the most popular, three sources were used: Amazon.com: Internet's biggest bookstore, Indeed.com: job search engine and Google.com: most popular search engine.

Amazon service was used to get the values of the metric - number of published books about the web frameworks. The results are presented in the fig. 4.1. The most described frameworks are: Ruby on Rails, Apache Struts and ASP.NET MVC. The least described frameworks are: Yii, Wicket and Tapestry.

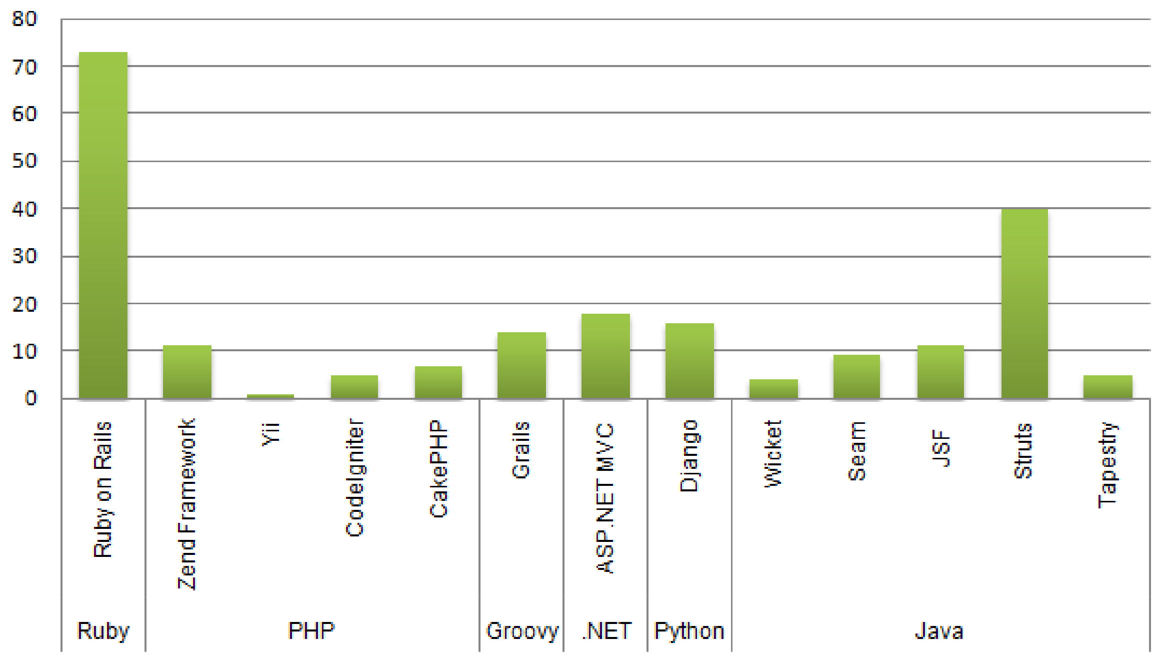


Figure 4.1: Number of books about web frameworks (Amazon.com, 17-02-2011)

Another metric that could provide deeper insights on popularity of web frameworks in the market, is percentage of jobs offerings gathered in Indeed.com. The absolute scale is presented in fig. 4.2. Results show that familiarity of Apache Struts, JSF, Ruby on Rails is most required. Figure 4.3 presents the data in the relative scale and therefore it depicts more accurate current trends in demands of web frameworks. It shows that very rapid growth in popularity have: Ruby on Rails, Zend Framework, Django and CakePHP.

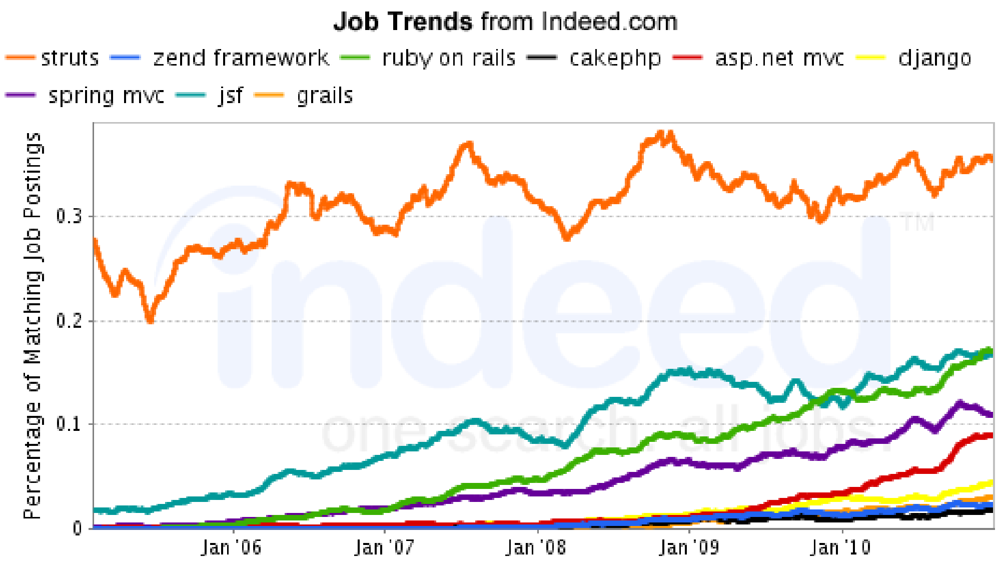


Figure 4.2: Job trends for web frameworks (scale: absolute)

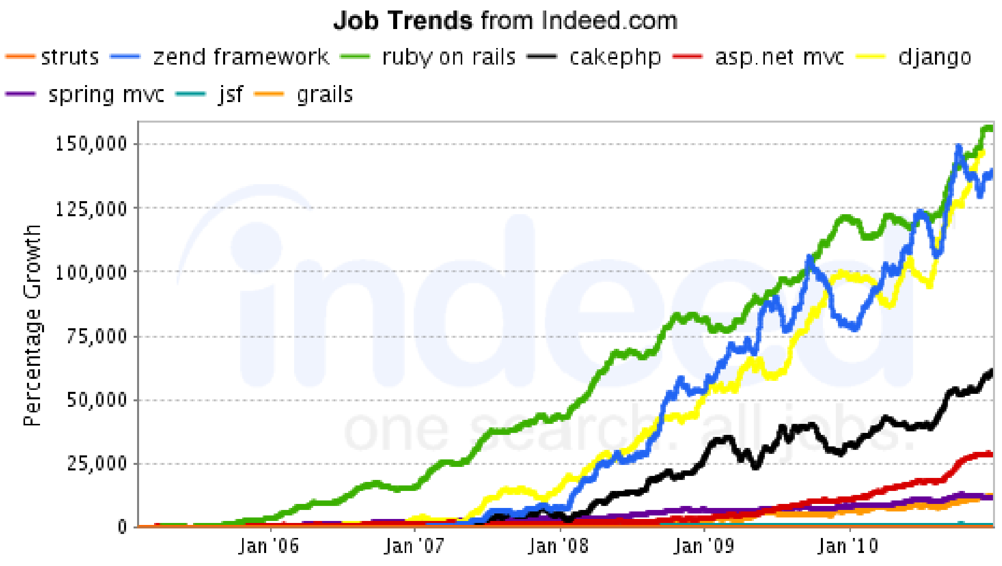


Figure 4.3: Job trends for web frameworks (scale: relative)

Overall search trends from Google.com served as a validation of job trends. In the figure 4.4 there is a comparison among the top four results from the jobs trends results in relative scale. It is noticeable that there is a significant interest in Ruby on Rails and recently it is equal to CakePHP's trends. However, the trend for Ruby on Rails has a falling tendency. As for the absolute scale (fig. 4.5), trends for all frameworks have been stabilizing since 2008. It is clearly seen that, Apache Struts is the most popular framework among Java frameworks.

Taking all statistics into consideration, the choice of popular frameworks for the case studies involves: Ruby on Rails, CakePHP, Zend Framework and Apache Struts. The overall choice is dictated by making the case studies diverse. Choosing frameworks that were written in different languages: Ruby, PHP and Java will support

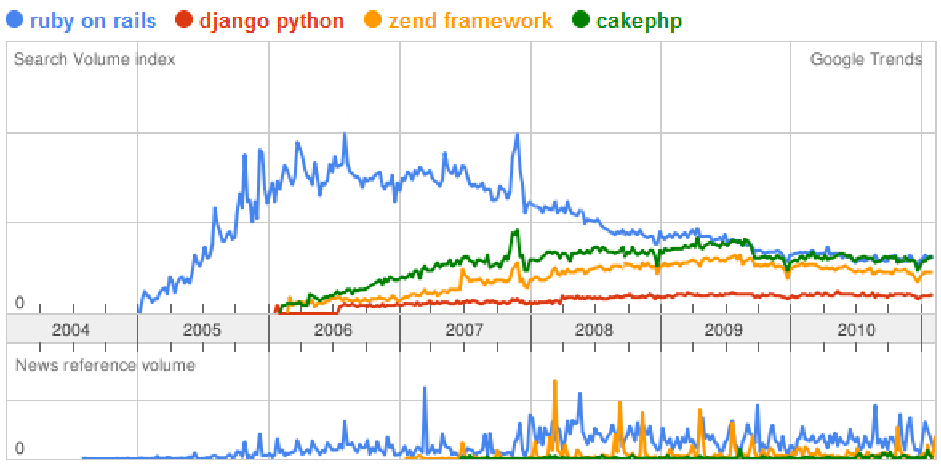


Figure 4.4: Google Trends chart for web frameworks (with regard to fig. 4.3)

better validation of the classification scheme. Furthermore, only open source frameworks were chosen, leaving popular, but based on the propriety platform, ASP.NET. The special motivation behind the choice of Apache Struts is that it is one of the oldest web frameworks and it is possibly the most well-described framework. As for the CakePHP, it is noticeable that there is an increasing trend in using this framework among other PHP frameworks. Reason in choosing two PHP frameworks is that PHP is the most used scripting language in WWW (TIOBE Software BV, 2011).

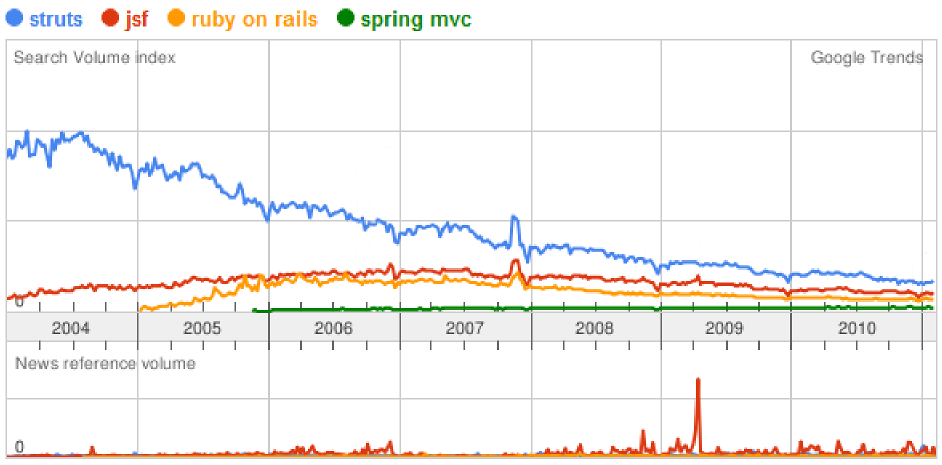


Figure 4.5: Google Trends chart for web frameworks (with regard to fig. 4.2)

4.2 Creating new classification scheme

First step in creating a classification scheme involves defining requirements which the classification scheme will fulfill. The requirements should reflect the need of usefulness, simplicity and high level of understandability of the classification scheme.

The classification scheme should have more than two problem categories, because multi dimensionality would increase usefulness of the scheme. Other requirement could be that each pattern could have many criteria to be ranked by. To avoid information overload problems, each pattern should have from two to three criteria.

Moreover as Buschmann (1996, pp.363-364) suggests the classification scheme must demonstrate main pattern properties, it should allow to add new patterns and should be a some kind of a roadmap that will help developers find a needed pattern quickly. This roadmap should reflect pattern's natural criteria, like: a pattern category or a problem category.

In the proposed classification scheme for web programming, the most useful criterion would be the purpose since it would reflect recurring web application problems. Following purposes can be defined (tab. 4.1):

purpose	problem example
user interface	templates, forms validation, localization
data persistence	database access, session management
safety and security	authentication, authorization
performance optimization	caching
producing resources	creating RSS feeds
extendending functionality	creating plugins

Table 4.1: Purpose criteria

Other useful criterion would be a scope. Since many patterns are built on other patterns, for example Active Record pattern can use Observer pattern, the scope could address those problems. If a pattern is a part of other pattern's construction, then its scope will concern a framework scope otherwise a pattern scope, so for example, the scope of ActiveRecord will be a framework scope and Observer's will be a pattern scope. The scope criterion would also validate how complex are relationships in web frameworks. It is assumed that hierarchies are simple and relationships among design patterns in web frameworks are closed.

Following matrix depicts proposed classification scheme (fig. 4.6):

	framework	pattern
user interface		
data persistence	Active Record	Observer
safety and security		
performance optimization		
producing resources		
extendending functionality		

Figure 4.6: Proposed classification matrix example

This proposed classification scheme is a draft which after application to the content, should be evaluated if it needs refinement.

4.3 Design patterns in web frameworks

4.3.1 Ruby on Rails

Ruby on Rails is an open source web framework based on the Ruby programming language. It was created by 37signals in 2004 and it available under the MIT (Massachusetts Institute of Technology) license. The latest version of the framework is 3.0. It is one of the most popular and supported open source projects with more than more than 1,600 contributors (37signals, LLC, 2011). Examples of web applications based on Rails are: Twitter (twitter.com), Github (github.com) or Groupon (groupon.com).

There are two main principles that are behind Rails design - DRY ('Don't Repeat Yourself') and Convention over Configuration. The first principle instructs a developer to avoid writing repeating code and states that: "every piece of knowledge must have a single, unambiguous, authoritative representation within a system"(Hunt and Thomas, 2000, p.26). The second principle, CoC, means that a developer only needs to use predefined rules that regulate everything that is unconventional.

The architecture behind Ruby on Rails is MVC. When a client sends a HTTP, RSS, ATOM or SOAP request to a web server, for example Apache, WeBrick, Mongrel, Lighttpd or Nginx, the server forwards this request to a dispatcher e.g., invokes FastCGI mod_ruby which loads ActionController that is Rails' Application Controller (fig. 4.7). ActionController's role is to decide what decisions and actions need to be taken to fulfill the client's request.

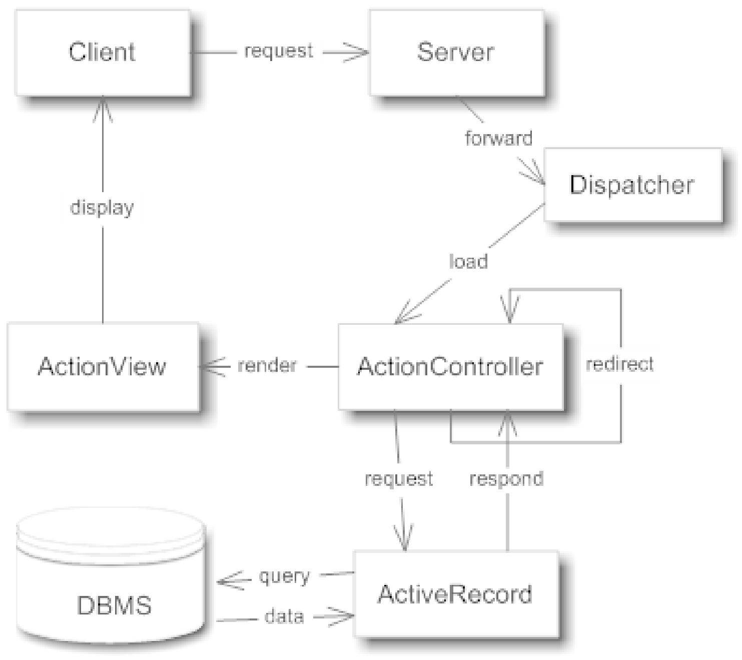


Figure 4.7: Architecture of Ruby on Rails

ActionView is the view of the MVC in Rails and it contains methods that render all templates, HTML pages, JavaScript and XML outputs. Helper modules that can be seen as implementing View Helpers patterns, support template behavioral extensions, for example AssetTagHelper is responsible for linking CSS style sheets to the templates.

ActiveRecord element in Rails architecture is a design pattern described by

Fowler (2003) in PoEAA book. It is responsible for object-relational mapping (ORM), connecting to the database and modifying the data. There are several patterns that are building ActiveRecord, for example, Adapter pattern is used to adapt the application to work with different database management systems, like MySQL or PostgreSQL and its interface is implemented in AbstractAdapter class.

Observer pattern is also part of the ActiveRecord. Observers receive callbacks from the ActiveRecord and allow to extend functionality of the model class without interfering Single Responsibility Principle and modifying its structure. Observers can be used for various, unrelated for the model class purposes, like managing cache or log events (Olsen, 2008).

Among other identified patterns (tab. 4.2) is for example Decorator pattern that can be found in the package ActiveSupport where it decorates methods. Naming convention and purpose suggest that Rails uses Builder patterns to create, e.g. atom feeds. Proxy design pattern is located in ActionContoller helpers where it is used to provide a proxy to access view helpers outside the actual views. Singelton pattern example is located in Inflections class that is a set of defined rules for plural words.

Pattern	Ref	Ruby on Rails	Context example	Original category
Abstract Factory	GoF			Creational
Active Record	PoEAA	X		Data Source Architectural
Adapter	GoF	X	concrete database adapters, like: PostgreSQL adapter	Structural
Application Controller	PoEAA J2EE	X	ActionController	Web Presentation Presentation layer
Builder	GoF	X	AtomBuilder AtomFeedBuilder FormBuilder	Creational
Command	GoF			Behavioral
Composite	GoF			Structural
Composite View	J2EE			Presentation Layer
Data Mapper	PoEAA			Data Source Architectural
Decorator	GoF	X	ActiveSupport method: alias_method_chain	Structural
Dispatcher View	J2EE			Presentation Layer
Facade	GoF			Structural
Factory Method	GoF			Creational
Front Controller	PoEAA J2EE			Web Presentation
Iterator	GoF			Behavioral
Model-View-Controller	PoEAA	X		Web Presentation
Observer	GoF	X	ActiveRecord Observer ActiveModel Observer	Behavioral
Page Controller	PoEAA			Web Presentation
Proxy	GoF	X	ActiveRecord Associations ActionController Helpers	Structural
Registry	PoEAA			Base
Service Locator	J2EE			Business layer
Service To Worker	J2EE			Presentation Layer
Session Facade	J2EE			Business layer
Singleton	GoF	X	ActiveSupport Inflections Rake::Application	Creational
Strategy	GoF			Behavioral
Synchronizer Token	J2EE			Presentation Layer
Table Data Gateway	PoEAA			Data Source Architectural
Template Method	GoF			Behavioral
Transform View	PoEAA			Web Presentation
Two Step View	PoEAA			Web Presentation
ValueObject	PoEAA J2EE			Base Business Layer
View Helper	J2EE	X	ActionView helpers: FormHelper AtomFeedHelper ActiveModelHelper JavaScriptHelper NumberHelper TagHelper TextHelper TranslatorHelper UrlHelper	Presentation Layer

Table 4.2: Design patterns in Ruby on Rails

4.3.2 Zend Framework

Pattern	Ref	Zend Framework	Context example	Original category
Abstract Factory	GoF	X	Zend_Cloud's resources Zend_Pdf's resources	Creational
Active Record	PoEAA			Data Source Architectural
Adapter	GoF	X	Zend_Db_Adapter Zend_Translate Zend_Captcha_Word Zend_Http_Client Zend_Log	Structural
Application Controller	PoEAA J2EE	X	Zend_Controller	Web Presentation Presentation layer
Builder	GoF	X	Zend_Feed	Creational
Command	GoF	?		Behavioral
Composite	GoF	X	Zend_Log Zend_Form Zend_Validate_Builder	Structural
Composite View	J2EE	X	Zend_Layout	Presentation Layer
Data Mapper	PoEAA			Data Source Architectural
Decorator	GoF	X	Zend_Form's decorators	Structural
Dispatcher View	J2EE			Presentation Layer
Facade	GoF	X	Zend_TimeSync	Structural
Factory Method	GoF	X	Zend_Cache Zend_Log Zend_Barcode	Creational
Front Controller	PoEAA J2EE	X	Zend_Controller_Front	Web Presentation
Iterator	GoF	X	Zend_Feed_Abstract Zend_Ldap_Collection Zend_Config	Behavioral
Model-View-Controller	PoEAA	X	Zend_Db, Zend_View, Zend_Layout and Zend_Controller	Web Presentation
Observer	GoF	X	SplObserver	Behavioral
Page Controller	PoEAA	X	Zend_Controller_Action	Web Presentation
Proxy	GoF	X	Zend_Queue Navigation Helpers	Structural
Registry	PoEAA	X	Zend_Registry	Base
Service Locator	J2EE			Business layer
Service To Worker	J2EE			Presentation Layer
Session Facade	J2EE			Business layer
Singleton	GoF	X	Zend_Auth Zend_Front_Controller Zend_Loader_Autoloader	Creational
Strategy	GoF	X	Zend_Cache Zend_Form's validators Zend_Filter Zend_Validator	Behavioral
Synchronizer Token	J2EE			Presentation Layer
Table Data Gateway	PoEAA	X	Zend_Db_Table	Data Source Architectural
Template Method	GoF			Behavioral
Transform View	PoEAA	X	Zend_Layout	Web Presentation
Two Step View	PoEAA		Zend_Layout	Web Presentation
ValueObject	PoEAA J2EE	X	Zend_Json_Expr	Base Business Layer
View Helper	J2EE	X	Action View Helper BaseUrl Helper Currency Helper Cycle Helper Partial Helper Placeholder Helper Doctype Helper Gravatar View Helper HeadLink Helper HeadMeta Helper HeadScript Helper HeadStyle Helper HeadTitle Helper HTML Object Helpers InlineScript Helper JSON Helper Navigation Helpers Breadcrumbs Helper Links Helper Menu Helper Sitemap Helper Navigation Helper TinySrc Helper Translate Helper UserAgent View Helper	Presentation Layer

Table 4.3: Design patterns in Zend Framework

Zend Framework is an open source PHP web framework available under New BSD License that was created in 2006 by Zend Technologies. The latest version of this framework is 1.11.5 and it requires PHP 5. Among the features of the framework are object relation mapping, unit testing, ACL-based security, special framework for templates, caching and form validation. Moreover, Zend Framework is very complex and built with many components. It contains the support for integration with Adobe Air, popular APIs like Amazon or OpenID authentication. The architecture behind Zend Framework is MVC with Zend_Controller as the controller and Zend_View as the view. There is no Zend_Model since Zend does not directly support the Model but it implements via specialized components, like: Zend_Db_Table etc. Furthermore, Zend Framework contains a huge amount of design patterns from every presented pattern classification. The identified design patterns are illustrated in the table 4.3.

4.3.3 CakePHP

Pattern	Ref	CakePHP	Context example	Original category
Abstract Factory	GoF			Creational
Active Record	PoEAA	X		Data Source Architectural
Adapter	GoF	X	AclComponent CakeLog DboSource for SQLite 3	Structural
Application Controller	PoEAA J2EE	X	Controller	Web Presentation Presentation layer
Builder	GoF	X	ConsoleOptionParser	Creational
Command	GoF			Behavioral
Composite	GoF			Structural
Composite View	J2EE			Presentation Layer
Data Mapper	PoEAA	X	Model	Data Source Architectural
Decorator	GoF			Structural
Dispatcher View	J2EE			Presentation Layer
Facade	GoF			Structural
Factory Method	GoF	X	ConsoleOptionParser CakeFixtureManager AclComponent	Creational
Front Controller	PoEAA J2EE	X	/cake/app/webroot/index.php	Web Presentation
Iterator	GoF	X	RecursiveDirectoryIterator	Behavioral
Model-View-Controller	PoEAA	X		Web Presentation
Observer	GoF			Behavioral
Page Controller	PoEAA			Web Presentation
Proxy	GoF			Structural
Registry	PoEAA	X	ClassRegistry	Base
Service Locator	J2EE			Business layer
Service To Worker	J2EE			Presentation Layer
Session Facade	J2EE			Business layer
Singleton	GoF	X	Debugger ClassRegistry	Creational
Strategy	GoF	X	AclComponent	Behavioral
Synchronizer Token	J2EE			Presentation Layer
Table Data Gateway	PoEAA			Data Source Architectural
Template Method	GoF			Behavioral
Transform View	PoEAA			Web Presentation
Two Step View	PoEAA			Web Presentation
ValueObject	PoEAA J2EE			Base Business Layer
View Helper	J2EE	X	FormHelper HtmlHelper NumberHelper PaginatorHelper RssHelper SessionHelper TimeHelper TextHelper CacheHelper XmlHelper JavascriptHelper AjaxHelper	Presentation Layer

Table 4.4: Design patterns in CakePHP

As Zend Framework, CakePHP is also a PHP web framework, but it was designed using Ruby on Rails principles and models in 2005 by Cake Software Foundation. As Rails it is open source and available under MIT License. The latest version of CakePHP is 1.3.8 and it has support for internationalization, localization and object relation mapping. Other features involve a testing framework that supports, for example unit tests, object mocking and ACL-based security.

Table 4.4 illustrates patterns identified in CakePHP. Since this framework was modeled after concepts found in Rails, there are some similarities in found design patterns, like ActiveRecord and ViewHelper patterns.

4.3.4 Apache Struts

Pattern	Ref	Apache Struts	Context example	Original category
Abstract Factory	GoF			Creational
Active Record	PoEAA			Data Source Architectural
Adapter	GoF	X	Action	Structural
Application Controller	PoEAA J2EE	X	RequestProcessor	Web Presentation Presentation layer
Builder	GoF			Creational
Command	GoF	X	ActionServlet Action	Behavioral
Composite	GoF			Structural
Composite View	J2EE	X	Template taglib Tiles taglib	Presentation Layer
Data Mapper	PoEAA			Data Source Architectural
Decorator	GoF	X	ActionMapping	Structural
Dispatcher View	J2EE	X	ActionMapping ActionServlet Action ActionForward	Presentation Layer
Facade	GoF			Structural
Factory Method	GoF			Creational
Front Controller	PoEAA J2EE	X	ActionServlet Action	Web Presentation
Iterator	GoF			Behavioral
Model-View-Controller	PoEAA	X		Web Presentation
Observer	GoF			Behavioral
Page Controller	PoEAA			Web Presentation
Proxy	GoF			Structural
Registry	PoEAA			Base
Service Locator	J2EE	X	ActionServlet Action	Business layer
Service To Worker	J2EE	X	ActionServlet Action	Presentation Layer
Session Facade	J2EE	X	Action	Business layer
Singleton	GoF	X	ActionServlet, Action	Creational
Strategy	GoF			Behavioral
Synchronizer Token	J2EE	X	Action	Presentation Layer
Table Data Gateway	PoEAA			Data Source Architectural
Template Method	GoF	X	process() method of the RequestProcessor	Behavioral
Transform View	PoEAA			Web Presentation
Two Step View	PoEAA			Web Presentation
ValueObject	PoEAA J2EE	X	ActionForm ActionErrors ActionMessages	Base Business Layer
View Helper	J2EE	X	Action ActionForm ContextHelper tag extensions	Presentation Layer

Table 4.5: Design patterns in Apache Struts

Apache Struts is a web framework based on J2EE platform for server-based programming in Java. It is open source and available under Apache License 2.0. Apache Struts is one of the first web application frameworks and was created in 2000 by Craig McClanahan. The latest version of Struts is 2.2.1.1 and it supports ORM, internationalization, localization and unit testing. However security and caching frameworks are not included(The Apache Software Foundation, 2011). As other web frameworks, Struts architectural style is MVC. In Struts the model is the application logic and data, views are in forms of JSP pages and ActionServlet with Actions make the controller.

Since Struts uses J2EE technology, there are many design patterns described in J2EE blueprints (tab. 4.5). For example, Service to Worker is a macro-level pattern, because it consists of two patterns: FrontController and ViewHelper and it is responsible for generating templates. FrontContoller pattern manages views, navigation and security issues. Command pattern is a part of a Front Controller and is implemented through Action classes. The use of Singleton pattern is mainly in Action and ActionServlet since Struts creates only one Action per application which benefits to performance optimization (Franciscus and McClanahan, 2002).

5 Results and discussion

Generally, Zend Framework contains 69% of all patterns in the list, CakePHP - 41%, Ruby on Rails - 31% and Apache Struts 50% (number of types of design patterns identified in a framework by constant size of the list - 32 as seen in figures: 4.2-4.5). That makes Zend Framework, a web framework with best traceability of design patterns that leads to better comprehension of the source code.

As for the design patterns, there are easily noticeable outliners that are found in every web framework, for example MVC and ApplicationContoller pattern. Moreover, ViewHelper pattern is always present to support Separation of Contents principle. Decorator pattern is often used to extend functionality and Builder pattern to produce resources. Almost not present in web frameworks are: Mediator, Memento, Bridge, Flyweight and Visitor (tab. 5.1 - 5.4).

As for the classification scheme, 'User interface' category contains most of the patterns. 'Safety and security' problem category seems to be vague since it is difficult in many cases to state if a pattern conforms to this category. Lastly, the classification scheme in a form of a matrix has become not useful. When the scope criterion: pattern and framework was chosen, it was assumed that, in web frameworks, relationships among patterns would be rather closed and hierarchies among patterns would not be complex. However, it was revealed that patterns in web frameworks have a tendency to form a web of relationships. There are many patterns that are building materials for other patterns or are strongly related to one another. For example:

- Service to Worker is built on View Helpers and Front Controller that is built with Composite pattern (tab. 5.4);
- Strategy patterns or Proxies can be found implemented in View Helpers (tab. 5.2);
- View Helpers and Application Controller are stronlgy related to MVC, so it is diffcult to logically separate them (tab. 5.1 - 5.4);
- Front Controller can use Singleton as a base and can be confused with Application Controller (tab. 5.2);
- Two Step View (which is highly related to Transform View) and Composite View are parts of the View in MVC (tab. 5.2);
- Builders that often build form elements related to the View, often build Composites as well (tab. 5.2);
- Observers, Adapters and Proxies are often incorporated in building Active Record classes (tab. 5.1, tab. 5.3);
- Registry pattern is concerted with Singleton (tab. 5.2, tab. 5.3).

Therefore it becomes very difficult to spot *patterns within patterns* with flat, 2D scheme. The classification scheme in a form of a matrix loses hierarchical relationships that could be illustrated via a tree or a map. The map of realtionships could be similair to the classification by relationships proposed by Zimmer (1995).

To end, design pattern relationships play a bigger role in the web frameworks architectures than it was assumed and as Zimmer (1995) concludes: "applying design patterns requires a fair knowledge of both single design patterns and their relationships". Hence the classification scheme must be refined to resemble a pattern language.

	framework	pattern
user interface	MVC, ViewHelper, Singleton, Application Controller	
data persistence	Active Record	Observer, Adapter
safety and security	Application Controller	
performance optimization		Observer
producing resources	Builder	
extendending functionality	Decorator, ViewHelper	Proxy, Observer, Adapter

Table 5.1: Example of classification scheme for Ruby on Rails

	framework	pattern
user interface	MVC, View Helper, Front Controller	Two Step View, Composite View, Singelton, Proxy, Strategy
data persistence	Adapter, Registry, Table Data Gateway	Singelton
safety and security	Front Controller, Singelton	
performance optimization	Factory Method, Singleton, Strategy	
producing resources	AbstractFactory, Builder	Iterator
extendending functionality	Decorator, Factory Method, ViewHelper	Proxy, Strategy

Table 5.2: Example of classification scheme for Zend Framework

	framework	pattern
user interface	MVC, View Helper, Front Controller	Composite
data persistence	Active Record, Data Mapper	Adapter
safety and security	Strategy, Factory method	
performance optimization	Registry	Singleton
producing resources		
extendending functionality	ViewHelper	

Table 5.3: Example of classification scheme for CakePHP

	framework	pattern
user interface	MVC, Service to Worker	Front Controller, View Helper, Command, Composite View
data persistence	Service Locator	
safety and security	Session Facade	Front Controller
performance optimization	Singleton	
producing resources		
extendending functionality	Decorator, ViewHelper	

Table 5.4: Example of classification scheme for Apache Struts

6 Conclusions and further studies

This chapter ends the thesis and contains a discussion on future research. The *first question* was: ”Which web application frameworks are popular?”. The answer for this question is that popular web application frameworks are based on MVC pattern. More exhaustive answer would be if used popularity metrics were extended to other factors, like number or contributors, commits in the code repository, frequency of releasing new versions etc, so that the results could bring new insights.

The *second question* concerned design patterns that can be found in the popular web frameworks. It was revealed that there are some popular design patterns, for example: MVC, ViewHelper, Application Controller or Decorator. Design patterns, like: Mediator, Memento, Bridge, Flyweight or Visitor are not popular. Most of the patterns concern graphical user interface issues and can be very similar or strongly related to one another.

Finally, the answer to the *third question* is that proposed current classification scheme must be reformed and needs further validation. It would be more useful if it would look like a Zimmer classification map with strongly highlighted hierarchy relationships. Furthermore, cataloging effort should be conducted with the community or at least some developers must interviewed to ensure correctness in categorization of particular patterns.

In the *future research*, there are several aspects of the study to be addressed. For example, the aspect of client-side web frameworks was unexplored, therefore it could be fruitful to conduct such investigation on web frameworks that execute UI on the browser side and make use of JavaScript and AJAX. Moreover, developing a full taxonomy with a dictionary of the used elements could also be a step in developing a standard that could establish conventions of the construction of a generic web framework and would reduce information overload problems when choosing a web framework. Standard-complaint frameworks would ensure quality and promote interoperability among web applications. Such standard could also contain standardized descriptions of patterns. Moreover, the scope of such standard could also be extended to frameworks for testing and user interface in web applications.

Bibliography

37signals, LLC (2011). Ruby on rails, [Online] Available from: `http://rubyonrails.org/`.

Alexander, C. (1979). *The timeless way of building: which makes possible for men and buildings to become a part of natire and alive.* Oxford U.P., New York.

Alexander, C., Ishikawa, S., Silverstein, M., and Jacobson, M. (1977). *A pattern language: towns, buildings, construction.* Center for environmental structure series, 99-0150903-3 ; 2. Oxford U.P., New York.

Allen, R., Lo, N., and Brown, S. (2008). *Zend Framework in Action.* Manning Publications Co., Greenwich, CT, USA.

Alur, D., Crupi, J., and Malks, D. (2003). *Core J2EE patterns : best practices and design strategies.* Prentice Hall PTR, Upper Saddle River, N.J., 2. ed. edition.

Antoniol, G., Fiutem, R., and Cristoforetti, L. (1998). Using metrics to identify design patterns in object-oriented software. In *Proceedings of the 5th International Symposium on Software Metrics*, METRICS '98, pages 23–, Washington, DC, USA. IEEE Computer Society.

Bawden, D. and Robinson, L. (2009). The dark side of information: overload, anxiety and other paradoxes and pathologies. *J. Inf. Sci.*, 35:180–191.

Bergamaschi, S., Guerra, F., and Leiba, B. (2010). Guest editors' introduction: Information overload. *Internet Computing, IEEE*, 14(6):10 –13.

Brooks, F. (1995). *The mythical man-month: essays on software engineering*, chapter No Silver Bullet - Essence and Accident. Addison-Wesley, Reading, Mass., anniversary ed. edition.

Buschmann, F. (1996). *Pattern-oriented software architecture. [Vol. 1], A system of patterns.* Wiley, Chichester.

Buytaert, D. (2010, August). The Drupal overview, [Online] Available from: `http://drupal.org/getting-started/before/overview`.

Cornils, A. and Hedin, G. (2000). Statically checked documentation with design patterns. In *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*, pages 419 –430.

Dietrich, J. and Elgar, C. (2005, march-1 april). A formal description of design patterns using OWL. In *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, pages 243 – 250.

Dong, J. and Yang, S. (2003, oct.). Visualizing design patterns with a UML profile. In *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on*, pages 123 – 125.

Dori, D., Perelman, V., Shlezinger, G., and Reinhartz-Berger, I. (2005). Pattern-based design recovery from object-oriented languages to object process methodology. In *Proceedings of the IEEE International Conference on Software - Science,* *Technology & Engineering*, pages 77–82, Washington, DC, USA. IEEE Computer Society.

Eden, A. H., Hirshfeld, Y., and Yehudai, A. (1998). Lepus - a declarative pattern specification language.

Fowler, M. (2003). *Patterns of enterprise application architecture*. Addison-Wesley, Boston, Mass.

Fowler, M. (2005, June). Inversionofcontrol, [Online] Available from: `http://martinfowler.com/bliki/InversionOfControl.html`.

Franciscus, G. and McClanahan, C. R. (2002). *Struts in Action: Building Web Applications With the Leading Java Framework*. Manning Publications Co., Greenwich, CT, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1995). *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass.

Golding, D. (2008). *Beginning CakePHP: From Novice to Professional*. Apress, Berkely, CA, USA, 1 edition.

Gueheneuc, Y., Sahraoui, H., and Zaidi, F. (2004). Fingerprinting design patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 172–181, Washington, DC, USA. IEEE Computer Society.

Hunt, A. and Thomas, D. (2000). *The pragmatic programmer: from journeyman to master*. Addison-Wesley, Boston, Mass.

Jazayeri, M. (2007). Some trends in web application development. *Future of Software Engineering*, pages 199–213.

Johnson, R. E. (1997, October). Frameworks = (components + patterns). *Commun. ACM*, 40:39–42.

Kitchenham, B., Pickard, L., and Pfleeger, S. (1995, July). Case studies for method and tool evaluation. *Software, IEEE*, 12(4):52 –62.

Mapelsden, D., Hosking, J., and Grundy, J. (2002). Design pattern modelling and instantiation using DPML. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT '02, pages 3–11, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Meffert, K. (2006). Supporting design patterns with annotations. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:437–445.

Niere, J., Schäfer, W., Wadsack, J., Wendehals, L., and Welsh, J. (2002). Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 338–348, New York, NY, USA. ACM.

Ohtsuki, M., Segawa, J., Yoshida, N., and Makinouchi, A. (1997, aug). Structured document framework for design patterns based on SGML. In *Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International*, pages 320 –323.

Olsen, R. (2008). *Design patterns in Ruby.* Addison-Wesley, Upper Saddle River, NJ.

Philippow, I., Streitferdt, D., Riebisch, M., and Naumann, S. (2005). An approach for reverse engineering of design patterns. *Software and Systems Modeling*, 4: 55–70.

Pree, W. (1994). Meta patterns - a means for capturing the essentials of reusable object-oriented design. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, ECOOP '94, pages 150–162, London, UK. Springer-Verlag.

RDF Working Group (2004). Resource Description Framework (RDF), [Online] Available from: `http://www.w3.org/RDF/`.

Shklar, L. and Rosen, R. (2009). *Web application architecture: principles, protocols, and practices.* Wiley, Chichester, 2. ed. edition.

Taibi, T. and Chek Ling Ngo, D. (2003, July). Formal specification of design patterns - a balanced approach. *Journal of Object Technology*, 2(4):127–140.

Taylor, A. G. and Joudrey, D. N. (2009). *The organization of information.* Libraries Unlimited, Westport, Conn., 3rd ed. edition.

The Apache Software Foundation (2011). Apache Struts, [Online] Available from: `http://struts.apache.org/`.

TIOBE Software BV (2011, February). Tiobe programming community index for february 2011, [Online] Available from: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

Torchiano, M. (2002). Documenting pattern use in java programs. *Software Maintenance, IEEE International Conference on*, 0:0230.

Vaughan-Nichols, S. (2010). Will HTML 5 restandardize the web? *Computer*, 43 (4):13 –15.

Wendehals, L. (2003). Improving design pattern instance recognition by dynamic analysis. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA)*, pages 29 – 32.

Yacoub, S., Xue, H., and Ammar, H. (2000). Automating the development of pattern-oriented designs for application specific software systems. In *Application-Specific Systems and Software Engineering Technology, 2000. Proceedings. 3rd IEEE Symposium on*, pages 163 –170.

Zimmer, W. (1995). *Pattern languages of program design*, chapter Relationships between design patterns, pages 345–364. Addison-Wesley Publishing Co., New York, NY, USA.

Linnæus University
School of Computer Science, Physics and Mathematics

SE-351 95 Växjö / SE-391 82 Kalmar
Tel +46-772-28 80 00
dfm@lnu.se
Lnu.se