Web Application Re-engineering

The MemoryLane Case Study

Daniel Kinneryd
2014

Master of Science in Engineering Technology
Computer Science and Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Web Application Re-engineering  
- The MemoryLane Case Study 

Daniel Kinneryd 

Master Thesis in Computer Science and Engineering  
Luleå University of Technology  
Department of Computer Science, Electrical and Space Engineering 
February, 2014
Abstract

No software program within a changing environment is immune to change. This change increases its complexity. This complexity decrease the maintainability and the understandability of the program. Through the use of re-engineering a developer can make a software program cope with the increasing complexity and therefore make it easier to understand for future developers. Reusable software components like software modules and application frameworks can be of great use for re-engineering purposes.

This thesis presents a case study of the life logging web application MemoryLane. The purpose of the case study was to re-engineer the application so that it is easier understand, update and expand in the future. Theories like re-engineering, model-view-controller (MVC), software frameworks, object relational mapping (ORM) and code complexity are presented.

These theories are then used to implement a less complex prototype of the application. It uses existing technologies like Zend Framework 2 and Doctrine 2 ORM, as well as a new MVC-architecture. Special care is taken to implement the existing activity recognition algorithm used for building life stories.

Lastly it is shown through code complexity analysis that the new application is indeed less complex than the old.
Preface
This thesis is the final step of a degree in Master of Science in Computer Science and Engineering at Luleå University of Technology (LTU).

The prototype discussed in this thesis was implemented during the spring of 2013 as part of a project at The Department of Computer Science, Electrical and Space Engineering at LTU. The goal was to re-engineer an existing application for life-logging called MemoryLane and turn it into a theoretical supported, easy to update, well designed and modularized platform for handling activity recognition algorithms. The task turned out to be a greater challenge than I first anticipated as the original application lacked all these requirements. Even so, it has been a very interesting project to be part of and I have learned a lot. The new application lives up to the requirements set at the start of my work and the project group seems happy with the results.

I would like to thank my supervisor Josef Hallberg (LTU) for his outstanding support and undying patience. The same goes for my wife Hanna, who understands that writing a thesis can sometimes be more important than doing the dishes.
Contents
1. Introduction ........................................................................................................................................... 1
  1.1 Background ........................................................................................................................................ 1
    1.1.1 Program evolution ......................................................................................................................... 1
    1.1.2 Software components and reuse .................................................................................................... 1
    1.1.3 Software Re-engineering ............................................................................................................... 1
  1.2 Case Study - MemoryLane .................................................................................................................. 2
    1.2.1 Problem definition ......................................................................................................................... 2
    1.2.2 Life Logging .................................................................................................................................. 3
    1.2.3 Activity Recognition ..................................................................................................................... 3
    1.2.4 Ethics ............................................................................................................................................. 4
  1.3 Research purpose ................................................................................................................................. 4
  1.4 Delimitations ....................................................................................................................................... 5
  1.5 Outline of the thesis ............................................................................................................................. 5
2. Theoretical work ....................................................................................................................................... 6
  2.1 Software Re-engineering Theory ......................................................................................................... 6
    2.1.1 Re-engineering Methodology ......................................................................................................... 6
    2.1.2 Reverse engineering steps ........................................................................................................... 8
    2.1.3 Notes on Refactoring .................................................................................................................... 9
  2.2 Model-View-Controller ....................................................................................................................... 10
  2.3 Software Frameworks .......................................................................................................................... 11
    2.3.1 Inversion of Control .................................................................................................................... 11
    2.3.2 Dependency Injection ................................................................................................................ 12
    2.3.3 Service Locator .......................................................................................................................... 12
  2.4 Object Relational Mapping ................................................................................................................ 13
    2.4.1 Advantages of using an ORM ....................................................................................................... 13
    2.4.2 Disadvantages of using an ORM .................................................................................................. 13
1. Introduction
First I start with a brief introduction. This chapter gives you a short background, presents my case study and gives you the purpose of this thesis. Delimitations and thesis overview are also here.

1.1 Background

1.1.1 Program evolution
No software program within a changing environment will ever have all the requirements for the future nor will it be immune to change. Rather, it will become more and more complex to cope with the changes of the outside world. This is reflected in two of the basic laws of software evolution [2]:

- **The Law of Continuing Change** — A program that is used in reality must change, or become progressively less useful.

- **The Law of Increasing Complexity** — As a program evolves, it becomes more complex. This will continue until work is done to maintain or reduce the complexity.

Example of ways to cope with continuing change and increasing complexity are components, reuse and re-engineering.

1.1.2 Software components and reuse
Software systems cannot and should not be replaced as a whole when they grow old. Therefore it is recommended to design the system with individual components that can be replaced or changed as the system evolves. These components have individual interfaces, they encapsulates internal details and they have separate documentation. [7]

One key advantage of components like these is software reuse. Software reuse is when existing software is used to construct new software [9]. Examples of software reuse are design patterns and software frameworks. A design pattern is a reusable software design. They are not code in themselves, they rather express a commonly tried and tested approach in software [8]. A framework uses these design patterns to form a skeleton for an application. Frameworks are always code, and they always have a particular purpose for the applications they support. A framework shows the architecture of the application [8]. Since a framework has its area of expertise, its domain, multiple frameworks may be needed by an application [10].

A special kind of component is the software module. In this thesis a software module is an implementation unit. Modules are independently developed and do not need each other to function. They are instead linked and combined by, for example, a framework.

1.1.3 Software Re-engineering
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." - Fowler et. al. [1]

Re-engineering is the analysis and modification of a software system. The goal with software re-engineering is to understand the existing software and then to improve the system. Re-engineering can consist of reverse engineering, refactoring and forward engineering [12][28]:

1
• **Reverse engineering** is a process where one analyze an existing software and creates a representation of this system on different abstraction levels.

• **Refactoring** or restructuring is the transformation of a system at the same abstraction level. It keeps the external behavior of the system.

• **Forward engineering** is the traditional way of designing systems. It usually starts from high abstraction and ends in an actual physical implementation of the system.

1.2 **Case Study - MemoryLane**
MemoryLane is a life-logging application, which stores media- and context-data, filters data, and segments data into daily activities. The purpose of this application is to support people with dementia by providing an overview of the activities the user performed during the day. The user is able to go into the application and review the data, look at the pictures, select significant pictures for the activity, add descriptions and other data important to the activity.

![Figure 1.1. Part of the old MemoryLane application. Source: [4]](image)

1.2.1 **Problem definition**
The application has already been tested and some shortcomings have been analyzed. The main drawback is the lack of flexibility as many settings are hardcoded in the software, and the algorithms for filtering and segmenting data are not interchangeable. This inhibits the possibility of adding third-party algorithms to the platform. The problem with opening up for third-party algorithms is the formatting of data, adding new types of data, and interfaces for making integration seamless and
simple. There is therefore a need to create a more flexible platform by centralizing the configuration and modularizing the code.

1.2.2 Life Logging
Life logging is to digitally record aspects of one's life. Most often the purpose is to remember specific events. It can be done by for example notes or the use of a diary. This has become more and more of a trend. More advanced life logging has surfaced. Nowadays online blogs are everyday life on the web. But life logging is about to become even more advanced. Scientists have long dreamed of automatically storing all that is important in our life digitally. The next stage towards this is to constantly take pictures during the day with a digital camera and store (with a GPS) the exact location and time of the picture being taken. The classic diary becomes visual. [4]

This automatic visual diary can be used for other things than the obvious. People with memory problems, for example mild dementia, can use this diary to remember their day and therefore have a richer life. [5]

Figure 1.2. Camera for life logging. Source: [4]

1.2.3 Activity Recognition
Activity recognition is to collect data about people's actions and behaviors. This data is collected from different sensors (such as pictures taken or GPS). The task at hand is then to sort these sensor data into meaningful activities; activities that can be understood and have meaning to humans [4]. This can be a small activity such as "picking up the pencil" to large activities like "vacation trip in Norway". One way of clustering the data into activities is by places visited and people met [6].
1.2.4 Ethics
Personal data is sensitive. Not everyone should have access to it, especially not without the permission of the individual whom the data belongs to. In the MemoryLane application only the users themselves have access to the pictures taken and the activities formed. In the current application prototype the pictures are stored locally and not open on the Internet. This may however change in the future, see the chapter about the future of the application at the end of the thesis.

According to Swedish law it is allowed to take pictures in public areas. If you take pictures standing on private property, the owner of that property can disallow you taking any pictures. It is also allowed to take pictures of people without their consent. You can take pictures of pretty much everything. However, if the picture is shown openly, for example on a web page, you need consent. Considering that pictures are needed in the MemoryLane application this is a good thing. It allows pictures to be taken for most of the time and these pictures can then be used to help people with mild dementia improving the quality of their life. [4][31][32]

1.3 Research purpose
The purpose of this project is to re-engineer the MemoryLane application into a prototype that follows good system design. This prototype will then be used as a foundation for future work on this life-logging application. The prototype must therefore make it easier for future programmers of the application to understand, update and expand the application; i.e. the maintainability$^1$ and understandability$^2$ of the application must be improved. This will be done in the following steps:

- Transforming the static implementation to an open, modularized, and flexible platform.
- Changing all configurations, such as user customization and language, to be centralized and made easy to change.

$^1$ Maintainability is the ease with which a software can be understood, corrected, adapted and enhanced. [27]

$^2$ Understandability is the ease with which an application can be understood. This is determining what a program does and how it works by reading the source code and the documentation. [27]
• Extracting the existing activity recognition and implement it as a separate module.

• Specifying a serialized method of adding third-party activity recognition modules to indicate in which order filtering and segmentation will be performed.

• Investigating how data storage can be made flexible enough to support new types of data to be added upon request from third-party modules.

1.4 Delimitations
The application of this project is a prototype. The fully fledged application would have to take more aspects into consideration. These aspects are important but not part of this thesis. These are:

• Activity recognition algorithms. The inner workings of these algorithms are not part of this thesis. Only an already existing algorithm is used.

• Multi-user environment. Since the application is web based it should naturally support multiple concurrent users. But the prototype does not address this.

• The camera and other media collecting devices. These are not part of this thesis. How data is collected is out of this scope.

• Security and privacy. Any web application should have a plan when it comes to security and user discretion. This is however out of scope.

1.5 Outline of the thesis
Chapter 2 handles the theoretical work. Already well-founded facts and accepted theories in the fields of re-engineering, model-view-controller, software frameworks, object relational mapping and code complexity are presented.

Chapter 3 handles the actual implementation; how the purpose is met by implementing the case study application presented above. This chapter shows the actual re-engineering methodology used, what technologies that realize the theory, the architecture of the implementation as well as how the application is built. It also shows how activity recognition and modules works together as well as how resources are reached within the application. Lastly the data for the code complexity analysis is presented.

Chapter 4 constitutes a discussion about the theories and the implementation. The work done in chapter 3 is also evaluated. It is about the re-engineering, the technologies used, analysis of the code complexity plots as well as some future work needed.

Chapter 5 concludes this thesis.
2. **Theoretical work**

For my project I will need to know how to best change an application into something new. I will also need to find one or several design patterns that can support me in building a flexible web application. A suitable architecture for the application must be chosen so that the new application is open, modularized and customizable. On top of that it also has to support new data types. A specific requirement is also that it supports modularization and methods for filtering and segmentation of activity recognition algorithms. Thus this chapter goes through some key points of already established theories and principles that can be used to build flexible, open and modularized applications that support maintainability and understandability.

2.1 **Software Re-engineering Theory**

Software re-engineering is the examination, analysis and alteration of an existing software system to reconstitute it in a new and better form [12]. The goal is to understand the existing software and then to improve the system (functionality, performance or implementation).

Re-engineering always has a goal. The goal can be to change the architecture of the system, for example to extract the business rules or to enhance the performance. When changing the architecture of the system one does not need to understand the small details. [28]

Software re-engineering can be done in different ways. It can be done incremental, one piece of the old system at a time. It can be done evolutionary, one piece at a time but based on functionality instead of only structure. The final way is called the "Big Bang Approach". Here, the whole system as a whole is replaced by a new system. Advantages are that interfaces and environments do not need to be shared. Disadvantages are that it can take a lot of time to do, that the new system might be a monolith that is not really suited for the needs at hand, and that new requirements may surface before the re-engineering is completed. [12]

Re-engineering becomes handy when the software system is broken. "Broken" in a software system context can be defined as "can no longer be maintained" [11]. When it comes to more coarse-grained architectural problems these are common [11]:

- **Insufficient documentation.** Documentation either does not exist or is not up to date.
- **Improper layering.** Missing or improper layering decreases portability and adaptability.
- **Lack of modularity.** Strong coupling between modules hinders evolution.
- **Duplicated code.** It is quick to copy-paste code, but it is really bad for maintenance.
- **Duplicated functionality.** Similar functionality exists in several places of the code.

2.1.1 **Re-engineering Methodology**

Re-engineering can encompass one, two or all of the following sub-processes [12]:

- **Reverse engineering.** The process to analyze a system to identify its components and their relationships. After that one creates representations of the system in another form or at a higher level of abstraction.

- **Refactoring.** The process to restructuring a software system in a way so that it does not change the external behavior of the system [1]. Instead it improves the internal structure. When you refactor you improve the design of the code after it has been written. According to [14], this is at the same relative abstraction level.

- **Forward engineering.** The process to move from abstract or high implementation-independent design to a physical implementation of a system. This is the traditional way of designing systems [28].

Below is a figure showing my view on the re-engineering process. In this model one always start with an old system to be re-engineered. First step is to analyze the source code. This source code is then presented in another form. At this point, one can immediately refactor the source code (by following the refactor arrow to the right). This could be done when porting to a new language or changing the internal structure of a piece of code. Most often, though, one would take the description and move up an abstraction level, the source code can be represented as a function. In an object-oriented environment this can be a class for example. Refactoring can be done at this level as well. The interface of the function can change or the internal algorithms updated. Remember, refactoring does not change the external behavior of the system, but it can change its components. Most often one would move up another level again, to the architecture level. At this level refactoring is changing types of components, interactions between components and putting functions into modules. If it is still hard to refactor at the architecture level, one can move up to the intentions level. This level represents the requirements or the intentions of the system. It is what the system is supposed to do. If we fail to understand the architecture of the system, how it is built, we can always return to this abstraction level that captures the essence of the application. If refactoring is done at this level it can be seen as a start of a new architecture, but it does not change the external behavior of the system.

Throughout the steps from source code, to functions, to architecture and to intentions one always try to extract knowledge of the system to the new abstraction level. This is reverse engineering. Once the refactoring is done one might need to forward engineer to finally land on the new source code of the new application, i.e. use traditional software engineering methodology.
2.1.2 Reverse engineering steps

In order to understand a software system, sufficient information must be extracted from the system for the re-engineering goal to succeed. Reverse engineering does just that; it creates and abstract a description of the system. [28]

Reverse engineering can be done at different levels. If the code quality is very poor in the old system, one can choose to extract and restructure only a few valuable components only. If the old system structure is so bad that no useful piece of code can be reused, extraction of knowledge imbedded into the old system can help to specify the new system. Since reverse engineering is so coupled with program understanding it is also coupled with code complexity, the more complex a program is the harder it is to reverse engineer. Reversed engineering can be done in the following steps [28]:

- Scope: Firstly one must decide which part of the system should be refactored/restructured and why. All parts may not be important or critical to restructure.
• Domain: Identify business tasks that the system should support. These tasks can then be divided into use-cases. These should be documented and should be mapped to their data structures (entity, table column etc.).

• Architecture recovery: Cluster software elements together that support the business tasks. This is done by tracing which software elements connect to certain tasks. This can be done by seeing which data structure is connected to what software element. In a system where there is no separation between views and models this is hard.

An alternative to business tasks is to work with features instead. A feature is a realized functional requirement for a system. It can be seen as an observable behavior of the system that can be triggered by a user. If the user wants to invoke a feature of a system, he needs to provide the system with input to trigger the feature. Sequences of user inputs that trigger actions in the system are called scenarios. The scenarios can then be mapped to a set of features that in turn can be mapped to computational units (for example blocks of code or a class) [29].

2.1.3 Notes on Refactoring
Refactoring can also be called restructuring. If refactoring is done on a high level of abstraction it can even be called to re-architect [28].

It is claimed that object-oriented programs are easier to refactor than conventional non object-oriented programs. This is due to the abstraction, encapsulation and inheritance that the object-oriented approach automatically induce [13].

When doing refactoring one should apply all the good system design choices available. Examples are increased cohesion, decreased coupling, modularize system parts, decreased class sizes, better class- and methodnames and so on [3].
2.2 Model-View-Controller

"Model View, Model View, Model View Controller

MVC's the paradigm for factoring your code,

into functional segments so your brain does not explode.

To achieve reusability you gotta keep those boundaries clean,

Model on the one side, View on the other, the Controller's in between."

-Model-View-Controller Song. Lyrics by James Dempsey. Source: [17]

Interactive systems are systems where interaction between system and users are central. The user interface design often needs to be updated to fit specific user needs. The underlying data model often need to be unaffected by these changes to the user interface. A special architecture that supports these kinds of systems is needed. [16]

The Model-View-Controller (MVC) is a compound design pattern that divides an interactive system into three components [16]:

- **Model.** Contains the core functionality and data. It also provides functions to access this data. It implements the Observer Pattern (see [8]) to notify the view or the controller of changes to its data.

- **View.** Displays information to the user. Different views present information to the user in different ways. Each view has a controller.

- **Controller.** Handles user input. These inputs are handled as events. These events are then translated into service requests to the model or display requests to the view.

This pattern makes it easier to maintain an application since [15]:

- The application's "look" in the view can change a lot without changing the business logic in the model.

- The application can easily support multiple languages or multiple user permissions.

This makes the MVC architectural design pattern very well suited for web applications. An overview of the MVC pattern is shown in the figure below.
2.3 Software Frameworks

A software framework dictates the architecture of an application that uses it. That is: the different components, how they work together, the flow of control between them as well as the design parameters that are available to the developer that uses the framework [8]. This makes the developer able to focus on the specifics of the application at hand and not have to worry about the details of general domain functions.

Domains are application areas. Examples are many: reservation systems, financial systems, software development tools, user interfaces etc [7]. A framework for applications in a specific domain is called an application framework [16].

A good framework uses design reuse rather than code reuse. They capture the design decisions that are common to their domain. Design patterns can be seen as building blocks for a framework. A framework can be seen as a pattern for a complete software system. A developer using the framework can then change certain aspects of the framework to their needs. The parts that can be changed are commonly called hot spots, while the parts that are hidden from normal development are called frozen spots. Hot spots are adaptable to the specific implementation that uses the framework. [16]

2.3.1 Inversion of Control

One very strong feature of a framework is the Inversion of Control (IoC). Another name of IoC is the Hollywood principle - "Don't call us, we'll call you!". Inversion of Control prevents dependency rot. This is when high-level components depend on low-level components that depend on high-level components again. It is very hard to understand a system when the rot happens. Instead, with IoC, we only allow high-level components to determine when low-level components are needed. The high-level components give the low-level ones a "don't call us, we'll call you"-treatment. [18]
Inversion of Control is what really sets a framework aside from a software library. In a library you call its functions, but then control is returned to your calling software [19]. In a framework, you insert your behavior code in the hot spots, and then the framework calls your code when it is time. Often, the framework knows how to call your code since you implement a certain interface that the framework knows of.

2.3.2 Dependency Injection
One way of introducing Inversion of Control in a object-oriented (OO) application is to use Dependency Injection (DI). Dependency Injection is about dependency management. Instead of letting the component take responsibility for instantiating dependencies with other components, another authoritative component inject any dependencies into the component. In an OO-perspective this is usually done by using the constructor or setters to wire together dependencies. This is very powerful since this can be done at runtime and not compile-time. [3] Below are some simple examples of no DI and DI.

```php
class Example {
    private $myInstanceVariable;

    function __construct() {
        $this->myInstanceVariable = new Variable();
    }
}
```

*Figure 2.3 A very simple example showing no Dependency Injection... (in PHP)*

```php
class Example2 {
    private $myInstanceVariable;

    function __construct(Variable $variable) {
        $this->myInstanceVariable = $variable;
    }
}
```

*Figure 2.4 ...and this example show Dependency Injection (in PHP).*

2.3.3 Service Locator
Another way of separating dependencies between components in a object-oriented application is using the Service Locator pattern. The basic idea of a Service Locator is to have an object that knows how to get hold of all the services needed by the application. This object then needs a method for every service it needs to give its callers. A service is in this case a component that also can be used remotely. Another approach is to have a dynamic Service Locator. This Service Locator can store any service you need into it and let other components use this service at runtime. [20]
2.4 Object Relational Mapping

Object Relational Mapping (ORM) is a technique for mapping the records of a relational database into the object-instances of an object-oriented program environment. This has great value since it decouples the relational schema and the domain object model using a mapping layer. [21] Building your own ORM can be very complex, so it may be advisable to use an already existing solution that suits your needs [22].

2.4.1 Advantages of using an ORM

Some advantages of using an ORM (tool) [22]:

- **Data portability**: One of the strongest points of an ORM is the data portability. The persistence layer can easily change with only minor changes to the code. The objects and their logic usually stay the same.

- **Developer performance**: An object-oriented programmer does not have to know a lot about relational databases. A developer can focus the programming tasks without worrying about the persistence layer.

- **Refactoring**: The ORM makes the database stay in its own layer. This makes it easier to refactor the application.

- **Time saving**: Once the ORM tool of choice is learnt it takes less time to implement new database structures. The ORM can implement the structures for you.

2.4.2 Disadvantages of using an ORM

Some disadvantages of using an ORM (tool) has been identified [22]:

- **Performance**: The queries to the relational database are auto-generated by the ORM. This is sometimes not as optimal as writing your own queries. This can drag down the performance of your application. However, in a good ORM you can state your own queries, but knowing when to do so can be hard.

- **Difficult to cover all use cases**: Today's ORMs cannot cover all use cases when it comes to database designs and object-oriented programming. In many cases some tweaking is needed.

- **A lot of objects**: Every row in the database if often reflected as an object. There can end up being too many objects, which can drag the application performance down.

2.5 Code Complexity

Code complexity can be defined as the difficulty of maintaining, changing and understanding an application [27]. Complexity can be seen as the effort needed to change a program. Code complexity is often seen as an important indicator of software quality and cost. The greater the complexity, the more fault-prone the software is. Many techniques exist that tries to measure the code complexity [26]. Some well-known techniques are McCabe's Cyclomatic Complexity (CC), Halstead's Measures and Lines of Code (LOC) [27].
CC is to state the number of linear independent paths through the program (flow graph). It can be stated as:

\[ v(F) = e - n + 2p \]

Where \( n \) = total number of nodes, \( e \) = total number of edges, \( p \) are the connected components or exit points (often 2). \( v(F) \) is the cyclomatic number. [26][27]

Halstead’s Measures focus on operands (variable or constants) and operators (*, /, +, and, etc.). Combining these into formulas gives for example program understandability and program effort. [27]

Lines of Code (LOC) can be defined as the count of lines of source code not including comment or blank lines [27]. It is very well used when determining the size and complexity of a program. Advantages of LOC are that it is easy to determine and corresponds well with effort and error density. The Disadvantage is that it is very much dependent on programming language (think Java vs. Assembler).

Understandability is the ease with which an application can be understood. This is determining what a program does and how it works by reading the source code and the documentation. Understandability has a reverse relation to complexity. [27]

Maintainability is the ease with which software can be understood, corrected, adapted and enhanced. There is no exact model determining maintainability, but complexity can be used as an indicator. [27]

Many complexity techniques correlate to each other. There exist for example a strong linearity between CC and LOC. This indicates that measuring one of them also measures the other. This holds true for any program, language, and even between the procedural paradigm and the object-oriented paradigm. [26]

2.6 Theoretical conclusions

By using these theories the desired refactoring of the application should be achievable. The re-engrining theory is needed to transform the application into something new. The MVC pattern seems to be a strong candidate when building a web application. A software framework seems to be a key point when trying to build open, modularized and flexible domain applications with focus on maintainability and understandability. If a framework that uses the MVC pattern could be found that would be even better. If it also supports Dependency Injection or a Service Locator this would make it even more flexible. Lastly an ORM tool is needed tool so that they flexibility of the application can be maintained while still having the option to add new data types to it. If all these theories and tools together can support the activity recognition algorithms it should form a solid foundation for the refactoring process.

Through the use of code complexity analysis it should be possible to determine if the new application has better understandability and maintainability than the old one; i.e. is easier to develop in the future.
3. Implementation
This chapter handles the implementation of the new application. The theories of the preceding chapter gave me insights into what to build and why. This chapter represents the how.

3.1 Use Case Recap
The final prototype developed in this thesis project is named *MLReview* after the old application *MemoryLane Review Client*. The purpose of the new application is exactly the same as the original application - to review and select any number of pictures to be saved from activities done during the day. These pictures are first sorted into activities by an advanced activity sorting algorithm, and then presented to the user for further selection. This activity sorting algorithms does not just simply choose pictures based on the time of when the picture is taken, but also on the target location (using GPS) and bluetooth recognition of known people nearby (through their devices).

The original application did not have the best architectural design, making it hard to update. The implementation in this thesis is based on the above Theoretical Work to transform the old MemoryLane Review Client into a flexible, reusable and extensible application. This was achieved by using a re-engineering methodology as well as a number of technologies discussed below.

3.2 My Re-engineering Methodology
The goal of re-engineering the old application was to refactor the whole system to fulfill the requirements. It has been done on a high abstraction level - maybe even higher than the architecture level at times. Immediately a number of problems were found: the system seemed to have all of the coarse-leveled architecture problems stated in the theory!

- **Insufficient documentation**: No relevant documentation was found at all.
- **Improper layering**: There was pretty much no layering at all. Portability and adaptability was near null.
- **Lack of modularity**: The different PHP-files called each other without thought to any modularity. This made the PHP-files very tightly coupled.
- **Duplicated code**: A lot of duplicated code was found. For example: database login with username and password was done in 59 different files.
- **Duplicated functionality**: Similar functionality existed in several places of the code. Some files were old and was not used anymore. Some had a lot of "dead code", but this was not obvious at first glance.

All in all: it seemed impossible to re-use or even refactor much of the source code of the old program. To fully understand the code, remove duplicated entries, spot dead code, group components and try to distinguish user interface from model based code would take weeks, perhaps even months. There was not enough time within the scope of this project. This landed the decision on making this re-engineering approach Big Bang; that is, replace the whole system with something new.
3.2.1 Reverse engineering
Presentation of the old system needed to be done. This was really hard following the standard method of indentifying components associated with for example business tasks. A good approach seemed to be working with features instead. Since the application is a web application, pretty much everything going on could be spotted on the web pages as a user. For example: Creating a new activity, editing an already existing person etc. All of these features were identified by triggering parts of the application, as a user. For example: if you clicked "upload image" you could add images to your activity. Scenarios (user actions triggering the features) were built and relevant data structures (tables in the database) were identified through exploration of the features.

The only thing extracted (except knowledge) to be refactored from the old application was the activity recognition algorithm used to automatically create activities. This part of the program was actually a module in some ways. It was developed as a separate part of the application and was therefore not as tangled as the rest of the code.

3.2.2 Refactoring
The only source code that actually survived the reverse engineering was the activity recognition algorithm. This was extracted, parts of it refactored and put into a separate module (see below). The interface of the module was made to fit into the framework paradigm, but the actual computations of the algorithm were left intact.

3.2.3 Forward engineering
All of the identified features of the web application were then tailored into the new architecture. The features were re-implemented and tested so that they exactly matched the old application. The new architecture consists of a framework using MVC and is supported by an ORM framework; all to meet the requirements.

The hot spots of the frameworks were implemented as the features needed them. The Model was the first layer built since it was the easiest to extract from the knowledge of the old application. It was built on top of the relevant data structures. Then Controllers were built to support the scenarios done. Lastly the View was implemented to exactly match the features identified. Resource location is very central to the new application, so the use of a Dependency Injected Service Locator was very helpful.

3.2.4 Additional refactoring of the new code
When forward engineering was done an additional refactoring of the new code and the new classes were also done. This was done as the mapping of the features into the new application went along. This shows that re-engineering can be done as an iterative process. The small refactoring steps mentioned in the theory were actually used here. Some code was restructured to be easier to read. Some central code was made into services located by the Service Locator. More on this in later chapters.
Figure 3.1 Visible high-level structure of the old and the new application.

Next, the figure 3.2 is the horseshoe model again, showing at what abstraction level work has been done. This is shown with a white arrow. Note that for example the activity recognition algorithm was refactored at the architecture level, but the rest of the application was remade at the intentions level.
3.3 Technologies

There are a number of technologies used in the new MLReview application. Some of the minor are discussed in the Architecture chapter, but two main technologies need their own presentation.

3.3.1 Framework - Zend Framework 2

Zend Framework 2 (ZF2) is an open source, object-oriented web application framework implemented in PHP. It is component based, fully object-oriented. It includes an extensible MVC implementation that supports different layouts. The loosely coupled architecture allows you to use whichever components you want or need without the need of implementing the whole framework. The makers call this "use-at-will". [23]

One of the key strengths of ZF2 for our purpose is the use of Modules to achieve modularization in ZF2 based applications. A Module is according to ZF2 is a stand-alone application that plugs into any other ZF2 application. This makes for a powerful foundation that other developers and projects can use to build their own modules or plug-in systems.

3.3.2 ORM - Doctrine 2

Doctrine 2 has a Object Relational Mapper (ORM) that sits on top of a powerful Database Abstraction Layer (DBAL). A DBAL is an application-programming interface that handles the communication...
between a database and an application program. Different database vendors can code their own
DBAL to have applications use their database. Doctrine 2 uses its own object-oriented database
query language (Doctrine Query Language). Doctrine also uses PHP classes known as "Entity"-classes
to map the database tables to PHP code. [24]

Another powerful feature of the Doctrine ORM is the custom data types. These data types are
defined in PHP-code, but map to any type (or types) of a relational database. This makes you able to
create data types that actually do not exist in standard PHP or databases. [24] Examples that I
implemented were a Point and a Polygon, two geometrical data types.

In the new application I use Doctrine 2 to almost seamlessly map PHP-objects (called Entity) to
relational database tables (done in MySQL). Once Doctrine 2 is configured correctly the mapping is
very straightforward and easy to use by any programmer.

3.4 Architectural overview

The MLReview uses the framework (Zend2) to create a Model-View-Controller (MVC) design to layer
the application. By layering the application, one separates the data (Model) from the presentation of
the data (View). This makes updating the application far more efficient. The Controller handles the
input logic between the View and the Model.

3.4.1 Model
My model uses a relational database as persistent storage (MySQL). Since my model uses a relational
database, I also need a way to translate between PHP objects and relational mapping. This is handled
by the Object Relational Mapper (ORM - Doctrine 2). Doctrine uses PHP classes known as "Entity"-
classes to map to the database tables to PHP code.

3.4.2 View
Since the application is a web application, the view is PHP-based html-pages (using the .phtml
extension). This makes for a powerful and flexible design using the strength of both languages. To
build even more dynamic web pages, Javascript is used to handle user interactions with web page
elements. Some advanced Javascript techniques like AJAX and tools like JQuery enhances the
developer possibilities further. The graphical layout is handled with Cascading Style Sheets (CSS),
making graphical web page changes easy to implement.

3.4.3 Controller
The controllers are ZF2 based classes that are designed to handle http-typed request messages
(POST/GET) and map these to the ORM (Doctrine2) or any other resource in the application.
3.5 How the new application is built

3.5.1 Overview

![Figure 3.3 Basic layout of the new application.](image)

Above is the basic layout of the new application.

**Config**

This is where the configuration of the application is. Zend2 uses array-based configuration elements specific to its different parts. This folder includes stuff like default images, navigation options, database connection, and last but not least configuration for the activity recognition algorithms (more on this later). This makes the configuration of the application centralized, easy to find and easy to update.

**Data**

Non-public data storage like caches and proxies, used by for example Doctrine.

**Database**

Files to build and populate the relational database. As for now this is a MySQL database. This could be switched to any database that have a DBAL supported by Doctrine.

**Docs**

The documentation for this project/application. This was lacking in the old application: but now extensive documentation for the installation, use and design of the application are here.

**Module**

The concepts of modules are very central to the new application, and therefore this part has its own chapter below.

**Public**

The files viewable by the web browser users. Ranges from images to Javascript to CSS-files.
Vendor

3rd part framework libraries like ZF2 and Doctrine2. This is a very large portion of the new application, but not all parts of the frameworks are used. The frameworks as a whole are there though in case more components are needed in the future.

### 3.5.2 Module

![Diagram of Module part of the application]

The "module/" folder holds the actual module source code. This folder is very central in the application. Currently there are two modules, the MLCluster and the MLReview.

Notice how both modules have their own configuration under the "config/" folder. This configuration holds module-specific things. This is later merged with the application wide configuration mentioned above.

Both modules also have a "src/" folder containing the actual source code that is not view-based code. That is, controller and model-based code. The view based code (phtml-files) are stashed in the "view/" folder.

**MLCluster**

MLCluster holds the source for the activity recognition algorithm refactored from the old source code. This module groups activities from a set of xml-files specified in the options in the configuration.

This module handles the sorting and grouping of activities. In the MLCluster case the view files are just a lot of Javascript code doing the calculation of the Clustering algorithm (activity recognition algorithm). This is not very much refactored due to its complexity. Only the integration with the rest of the MLReview application is refactored (changed). This makes the MLCluster view part not following the conventions of the rest of the Application very much.

**MLReview**

Is the largest module controlling and showing all things associated with the Review Client. See details below.
3.5.3 Module/MLReview

![Diagram of Module/MLReview]

**Figure 3.5 The module MLReview.**

**Module.config.php**

A lot of things are configured at startup of this module in this configuration file. Routes are specified so that web pages can be found throughout the application.

- A Service Manager (Service Locator pattern) is specified so that internal services can be used throughout the app.

- A Translator is specified so that text can be translated in the view-files.

- There are a lot of controllers that handle the calls between web pages.

- A View Manager are configured to handle which layouts are available and how to find view-pages (.phtml-files).

- Some Doctrine-specific things are told.

- Finally, some session options are initialized.

**Language**

These are the files used by the Translator of the MLReview application. The Translator is specified in the module.config.php of module MLReview. By using this Translator one can translate words in the
View (by using $this->translate('test')) using the .mo-files under this folder. The .mo-files are generated from the .po-files. The tool used to edit the .po-files in this project is PoEdit.

Src

The source structure is as follow:

- **Controller**: The controllers in the MVC layering. Controls the flow from View to Model and vice versa. Each use case category (for example “handle Activity”) has its own Controller. Each Controller class has its Action-functions. These are for example addAction(), editAction() or deleteAction(). This indicates what kind of action this function is associated with. If we are to edit an activity for example, we look into the ActivityController and the editAction(). In the end an action returns a View Model. This View Model is then used to display data in the view-pages (.phtml-files).

- **Entity**: Our Model. Used by Doctrine ORM to "talk" to our database. Each database entity is mapped to a Doctrine Entity PHP-class. These Entity-classes all follow a distinct pattern and uses special annotations that tells Doctrine ORM how they should work. Here are the custom data types of Doctrine also created. In my case: Point and Polygon.

- **Form**: Forms are ZF2’s way of make html-based forms (<form>) into Objects. These objects can then be used both in the View-layer and in the Controller-layer. These Forms (often consisting of Fieldsets and Elements) are then used in the Controllers to ship data to and from an Entity.

- **Service**: Everything that has to do with external systems. This folder is intended as a placeholder for classes involving external services and external modules. Not to be confused with the internal services of the Service Locator. An adapter for the MLCluster module lies here and its abstracts. This is needed because MLReview uses lists of Entity/Activity to display activities. The MLCluster uses old fashion arrays. A conversion between the two is needed.

- **Util**: Some utility classes used by many other classes throughout the application. Some are made available through the internal Service Locator. This includes a utility for file uploads as well as a translator for dates.

- **View**: Here are some Helper classes for our view scripts. These are not regular view-pages, hence their location.

View

There are about 26 view-pages in the MLReview module. Here are all the .phtml-files. Each represent a web page displayed to an user. They are built using standard web application techniques like html, Javascript and PHP. They display the Forms created in the Controllers.

A layout folder holds the different layouts of the application. Layout controls how web pages are put out to the browser.
Module.php

The Module.php runs every time the module is loaded, and the module is loaded every time a web page of the application is shown. This means that Module.php should be as lightweight as possible. The Module.php of Module/MLReview is the heart of the application configuration.

3.6 Activity Recognition and Modules

Parts of the purpose of the thesis are:

- Extracting the existing activity recognition and implement it as a separate module.

- Specifying a serialized method of adding third-party activity recognition modules to indicate in which order filtering and segmentation will be performed.

The Module concept implemented by our framework gives us a great foundation to implement any separate activity recognition module. At the moment the MLReview client only has one additional Module named "MLCluster". This module contains the old activity recognition algorithm used by the old version of MemoryLane Review Client. It is written in Javascript code. Javascript might not be the best choice to implement these algorithms, but the Module system is flexible enough to handle even this.

3.6.1 New Activity Recognition Algorithms

Implementing a new activity recognition module can be done in the following steps:

1. Defining when and where it is called in the sequence of algorithm method calls. This is done within the configuration files. Easy and straightforward.

2. Implement an Adapter for the answers from the specific module. MLReview uses lists of Entity-objects called "Activity" to show the user any saved activities. The Modules data needs to be converted into this using an Adapter. Adapter-classes are defined under the "Src"-folder. They build on inheritance.

3. Define new Filters (if needed). A Filter is just a way of filtering already converted Activity-objects further. This makes the new Modules focus on the Activity recognition and skip all the details of "How many medias per activity?" or "How long should an activity be?". Filters and their orders are defined in the configuration files as well.

4. The actual call to any activity recognition Module is made from a central Controller. This Controller uses the configuration files to make calls to the right module, in the right order and using the correct adapters and filters.

3.7 Resource Location

How to find resources in the application? For example, how does the central Controller find the adequate adapters and filters for a specific activity recognition module, using only from names in a configuration file? How do we find the utility needed to handle image uploads anywhere within our application?
This is done by using Dependency Injection of a Service Locator into every class that needs it. The Service Locator and its services are loaded during initiation of the modules (run at every go of the PHP-program). This Service Locator can then help the application components find other components without inducing dependency within the internal class. This becomes especially important when using the central Controller class to look for activity recognition modules to call - we do not want to have to change any code for each and every module added, only the configuration file.

```php
/**
 * Form for adding and editing persons
 */

class PersonForm extends Form {

    /**
     * Constructor
     * @param \Zend\ServiceManager\ServiceManager $serviceManager
     */
    public function __construct(ServiceManager $serviceManager) {
        parent::__construct('person-form');
        $entityManager = $serviceManager->get('Doctrine\ORM\EntityManager');
    }
}
```

Figure 3.6. Example of Dependency Injection of the Service Locator within a Form-class. The Service Locator is then used to find the Entitymanager of Doctrine2.

### 3.8 Documentation

In addition to the extensive coding mentioned in previous chapters, a lot of documentation was also done for the new application. This documentation was stored together with the source code to be easily accessed by any future developers. The documentation consisted of a detailed installation guide followed by an even more detailed application guide. The application guide described the actual application: the architecture, how new activity recognition algorithms was to be implemented, how MVC was used to enhance the layering, a detailed source code walkthrough, how the different frameworks was used and explanation of different complex source code parts. It also described practical things like what versions of the frameworks were used. Also known bugs and suggestions on how to correct them was included. Another important part of the documentation was what future work that could be done to make the application into more than a prototype.

All in all the situation of lacking documentation has been addressed and it should be easy to pick up the continued work on the new application.

### 3.9 Code Complexity

After the final application prototype was built, code complexity analysis was done for both the old application and the new one. There was a huge issue trying to determine the Cyclomatic Complexity in the old non-component based application. It was also silly to try to count the operators in a web application. The far best complexity measure deemed to be the Lines of Code (LOC). Since the different complexity techniques seem to correlate to each other, this should be a good choice. The weakness that it is dependent on the program language is not an issue for us since both programs are written in PHP.
To be able to accurately read the lines of code, a program named CLOC (Count Lines Of Code) was used [25]. CLOC can count the number of lines of code without the comments or blank lines. Below are the charts showing this comparison. The raw data resides in Appendix A.

The first one is a diagram showing the number of files for the two applications:

![Figure 3.7. The number of files in the old and in the new re-engineered application. The Y-axis shows the number of files, the X-axis the file types.](image)

The next one is a diagram showing the number of lines of code in the two applications:

![Figure 3.8 The number of lines of code in the old and the new application. The Y-axis represents the number of lines of code and the X-axis the file types.](image)
The next diagram shows the number of lines of comment in some of the program files:

![Diagram showing comment lines in old and new applications](image)

*Figure 3.9 The number of lines of comment in the old and the new application. The Y-axis represents the number of comment lines and the X-axis the file types.*

The last diagram shows what happens if the vendor frameworks that are used by the new re-engineered application are taken into account:

![Diagram showing code lines in old, new, and new with frameworks](image)

*Figure 3.10 The number of lines of code in the old, new and the new adding all the frameworks (Vendor). The Y-axis represents the number of code lines and the X-axis the file types.*
4. Discussion and evaluation

4.1 Re-engineering
My re-engineering methodology does not follow the theory on all points. My impression is that the basic methodology steps (reverse engineering, refactoring and forward engineering) are modeled after an application that have components, layers and blocks of code that can be traced and mapped to high abstract designs like an architecture. This is all good and fine, but when facing an application which severely lacks these things one have to skip a lot of the steps to present the old application (reverse engineer). I found the notion of extracting knowledge useful though. The Big Bang approach was also very good to be able to use. Finally: I found that my practical refactoring actually followed the theoretical steps, but only when used on my newly designed and layered components that at least had some structure to them.

The theory to extract and model features was very helpful; starting at the intention-abstract level to do a Big Bang.

4.2 Technologies
I was very impressed with the two main technologies.

4.2.1 Framework
Zend Framework 2 (ZF2) is a solid framework for the web application domain. That ZF2 builds on the object-oriented (OO) paradigm seems to be a huge benefit for future development as OO seem to be easier to refactor and almost "force" new developers to think about maintainability. The MVC pattern used in ZF2 really shines when it comes to different layouts and language selections. It also makes an application very well layered, differencing concerns of user layout apart from data handling. The Module concept is another part of ZF2 that was very useful. By implementing the activity recognition algorithm as a ZF2 module, the goal of modularization, openness and flexibility was implemented in my opinion.

By using Dependency Injection and the Service Locator of the ZF2, the application can grow and still stay very flexible even without using the concept of modules.

I did however run into some problems with the framework. Not the core ZF2 framework, but the modules translating between the ORM framework and the ZF2 framework. These were downloaded from the Internet and contained some bugs. The complexity of the framework structure was very complicated and it was a proper challenge to correct the bugs on my own (but I was successful in the end). The lesson I learned from this is that it is very important that the framework you are using is a solid piece of code that does not contain any serious errors. If not it will contradict the whole point of using a framework in the first place.

4.2.2 ORM
The Doctrine 2 ORM automatically handles all the transition from the relational world to the object-oriented one. Custom made data types can be implemented in the Doctrine ORM, and the mapping supports a lot of databases. This should make it a solid choice even for future implementations of the MemoryLane application. The speed with which new development could be done on the application
should be vastly increased with the data portability, the developer friendliness, and the time saving aspects of an ORM.

If the database-layer for some reason would need to be moved away from the ORM and into something else (spatial database?) the layering of the rest of the application should make this transition less painful.

My one concern when it comes to using ORM is the potential performance issues. Tests with a lot of simultaneous users (it is a web application after all) could not be conducted for this prototype application due to the delimitations. I would however recommend tests of this type in the future if a decision is made to continue using a ORM. The theoretical concerns to not cover all use cases or having a lot of objects does not seem to be a problem at first glance though.

### 4.3 Code Complexity

The only measurement that seemed to work was the Lines of Code (LOC). This was due to the unstructured character of the old application. Attempts to make some kind of relations between the high level PHP-files failed. Using operands and operators seemed a waste of time.

The results of the code complexity analysis follows.

Diagram 1 - The number of files:

![Diagram of file types](Image)

**Figure 4.1. The number of files for the old and the new re-engineered. The Y-axis represents the number of files and the X-axis the file types, same as in chapter 3.**

Not much to say about this one. There are about twice as many PHP-files in the old application as the new one, but to draw any conclusions from this seems wrong. Note however that the new application has split the PHP-files into standard PHP-files as well as PHTML-files. These PHTML-files are the View layer of the MVC in the ZF2 framework.

Diagram 2 - The number of lines of code:
This is however much more interesting. We see that the relevant source files (PHP and PHTML) have about 6 times more lines of code in the old application compared to the new one. This, I argue, shows that the old application is more complex and therefore is less understandable and maintainable. The Javascript- and the CSS-files have about the same LOC: not so strange since similar layout and View-functionality is used. The vast number of XML-LOC in the old application can be explained by the large number of test-files used in the old folders. The same goes for the SQL: it is backups of different database populations. These were not needed in the new one.

Does this mean that my new program is 6 times less complex and vastly superior when it comes to complexity? The answer is: probably not. What is not shown in the above diagram is the framework code. I also do not have an accurate model that says that 6 times more LOC means 6 times more complexity.

Diagram 3 - number of lines of comments:

Here too we see that the old application has a lot more lines. This is due to that version control was implemented by simply comment out pieces if code that was old and not used anymore. This is, of
course, not the case in the new application. Version control should also be implemented differently, but this is out of scope of this thesis.

Diagram 4 - number of lines of code including frameworks:

![Diagram 4](image)

**Figure 4.4. The number of lines of code in the old, the new, and the new adding all the frameworks (Vendor). The Y-axis represents the number of code lines, and the X-axis the file types, same as in chapter 3.**

Here we see that the introduction of frameworks makes the code far more complex, even more than the original application. This is however expected, as the frameworks used are vast and flexible. All of the frameworks are downloaded and installed, not just the parts implemented.

I still argue that the new application is less complex than the old one. If we see the frameworks as their own programs, not all of the parts of these programs are used and then add that the developer does not need to understand everything that happens in the frameworks, Diagram 2 is the one to draw conclusions about code complexity from.

### 4.4 Future work

The new application needs to evolve beyond being understandable and maintainable.

One thing that is not implemented and that should be a part of any refactoring is tests. Unit-tests should be a minimum, but these are not present to actually prove that the new application (or the old) does what it is suppose to do. This led us to another part that is missing: Requirements engineering. Requirements are the basis for every software project. It defines what stakeholders (users, customers etc.) need from a system, and what the system needs to do to satisfy them [30]. This vast area is not explored by this thesis. It is, however, crucial to any application. Without knowing what requirements exists for an application, it is hard to know that the application is actually useful. It is my experience that this is often overlooked in projects to save time, leading to less satisfactory end results. The completed software is technically good, but does not meet the user needs. Hence it might not be needed at all.

Other important aspects are ethical aspects, user privacy and security. In our particular application the life story of a user is stored. This private log should be protected in every way possible. User authentication, cryptology, transport layer security and other technologies applicable should be
used. Also: the consent from the user should be made and stored digitally. Since the application is aimed towards people with mild dementia it is even more important that it is controlled who gets access to the application data and why. The user often sits together with their caregiver when using the application. Who gets to see the life story and when? Perhaps some kind of log of user activities might be necessary? These are complex questions that once more call for more requirements engineering. This is however something for future research.

Finally I would like to emphasize that multi-user environment tests, like performance tests, are vital if this prototype is to become a real web application.

4.5 Final words
Using re-engineering I produced a final prototype of the MemoryLane application. This prototype was more understandable and more maintainable than the previous application. With the use of components, like the software module and frameworks built on design patterns, I made an application that is built of reusable parts. This lets the developer cope with the continuing change and increased complexity that affects all programs according to the Law of Continuing Change and the Law of Increasing Complexity.
5. Conclusions

The existing theories on re-engineering were not enough to re-engineer an application that lacked any obvious components or good architectural structure. This called for another re-engineering abstraction layer that I called "Intentions". By mapping the old application into so called features at this Intentions level, re-engineering could finally be done. By using the exact features of the old application, I argue that the new application does the same thing. The new application is however, according to the code complexity analysis, far less complex. This is due to the use of model-view-controller layering and frameworks.

The purpose of this project was to improve the MemoryLane application by re-engineering it. At the start of the thesis this goal was broken down into five steps. Below I will make a short summary on how each of these five steps have been met:

- Transforming the static implementation to an open, modularized, and flexible platform.
  - This was met by using an open, modularized and flexible platform using a model-view-controller layering and frameworks like the Zend Framework 2 and the Doctrine 2 ORM.

- Changing all configurations, such as user customization and language, to be centralized and made easy to change.
  - Configuration and language settings were centralized by using the Zend Framework 2 and other tools like a language translator.

- Extracting the existing activity recognition and implement it as a separate module.
  - The existing activity recognition algorithm was separated into a Zend2 module.

- To specifying a serialized method of adding third-party activity recognition modules to indicate in which order filtering and segmentation will be performed.
  - A serialized method for adding activity recognition modules was added. It uses the Zend2 module system. The module data is converted into data readable by the new application using adapter-classes. The data can be filtered using arbitrary number of filters attached to these adapters.

- Investigating how data storage can be made flexible enough to support new types of data to be added upon request from third-party modules.
  - Data storage was made flexible by using Doctrine 2 custom data types. Two types, Point and Polygon, were implemented in the new application. Doctrine 2 supports a wide range of relational databases.

Further: The code complexity analysis shows that the new application should have a better understandability and maintainability than the old application.
References


Appendix A - Analysis Raw Data

Raw data from the CLOC (Count Line Of Code) tool.

Raw data for the old application:

```
<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>93</td>
<td>27</td>
<td>0</td>
<td>74586</td>
</tr>
<tr>
<td>SQL</td>
<td>6</td>
<td>456</td>
<td>667</td>
<td>63381</td>
</tr>
<tr>
<td>PHP</td>
<td>129</td>
<td>27416</td>
<td>30312</td>
<td>47890</td>
</tr>
<tr>
<td>Javascript</td>
<td>53</td>
<td>5612</td>
<td>4939</td>
<td>15136</td>
</tr>
<tr>
<td>CSS</td>
<td>17</td>
<td>304</td>
<td>97</td>
<td>1936</td>
</tr>
<tr>
<td>HTML</td>
<td>4</td>
<td>475</td>
<td>108</td>
<td>1479</td>
</tr>
<tr>
<td>DOS Batch</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>SUM:</td>
<td>303</td>
<td>34290</td>
<td>36123</td>
<td>204410</td>
</tr>
</tbody>
</table>
```

Raw data for the new application without the vendor frameworks:

```
<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Javascript</td>
<td>52</td>
<td>4028</td>
<td>3451</td>
<td>22216</td>
</tr>
<tr>
<td>XML</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>12944</td>
</tr>
<tr>
<td>PHP</td>
<td>65</td>
<td>2318</td>
<td>4101</td>
<td>5546</td>
</tr>
<tr>
<td>PHTML</td>
<td>28</td>
<td>703</td>
<td>688</td>
<td>2838</td>
</tr>
<tr>
<td>CSS</td>
<td>11</td>
<td>154</td>
<td>164</td>
<td>2088</td>
</tr>
<tr>
<td>SQL</td>
<td>3</td>
<td>39</td>
<td>94</td>
<td>155</td>
</tr>
<tr>
<td>SUM:</td>
<td>178</td>
<td>7242</td>
<td>8488</td>
<td>45787</td>
</tr>
</tbody>
</table>
```
Raw data for the new application including the vendor frameworks:

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP</td>
<td>3631</td>
<td>63446</td>
<td>169193</td>
<td>263620</td>
</tr>
<tr>
<td>HTML</td>
<td>139</td>
<td>10032</td>
<td>0</td>
<td>31267</td>
</tr>
<tr>
<td>Javascript</td>
<td>54</td>
<td>4150</td>
<td>3641</td>
<td>23171</td>
</tr>
<tr>
<td>XML</td>
<td>69</td>
<td>156</td>
<td>3</td>
<td>14139</td>
</tr>
<tr>
<td>PHTML</td>
<td>30</td>
<td>716</td>
<td>698</td>
<td>2983</td>
</tr>
<tr>
<td>CSS</td>
<td>12</td>
<td>267</td>
<td>185</td>
<td>2562</td>
</tr>
<tr>
<td>YAML</td>
<td>32</td>
<td>20</td>
<td>0</td>
<td>771</td>
</tr>
<tr>
<td>XSD</td>
<td>1</td>
<td>54</td>
<td>0</td>
<td>466</td>
</tr>
<tr>
<td>Ant</td>
<td>3</td>
<td>22</td>
<td>51</td>
<td>191</td>
</tr>
<tr>
<td>SQL</td>
<td>4</td>
<td>39</td>
<td>84</td>
<td>156</td>
</tr>
<tr>
<td>Bourne Shell</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>61</td>
</tr>
<tr>
<td>DOS Batch</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>SUM:</td>
<td>3986</td>
<td>76911</td>
<td>173848</td>
<td>339415</td>
</tr>
</tbody>
</table>

http://cloc.sourceforge.net v 1.60  T=10.48 s (380.4 files/s, 56507.3 lines/s)