UNIVERSITY OF GOTHENBURG

Understanding Patterns in Software through Reverse Engineering

Karl Annerhult
karl.annerhult@gmail.com

Bachelor of Applied Information Technology Thesis
Report No. 2010:059, ISSN: 1651-4769

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, May 2010

Abstract

Software patterns are common solutions to common problems. The key difference in making the most of such patterns lies in understanding what patters are actually used and how an organization or individual may improve their ways of designing software based on them. This paper presents the results of software pattern evaluations performed on five projects within a small game development organization. The aim of the study has been to reverse engineer the five projects, and to use the models and diagrams produced in the process as a foundation for the conversations and interviews with the developers. Additionally the analysis is looking at how design recovery in a number of different projects in house of one organization can support them in understanding their own patterns. The main contribution of this paper lies in the discussion around how reverse engineering and design evaluation can support organizations understanding of patterns, and how the understanding of software patterns can aid organizations in future development.

Keywords

Software Patterns, Design Patterns, Reverse engineering, Reverse Architecting, Design Recovery, Action Script 3.0

1. Introduction

In the field of construction, a pattern describe a problem which occurs over and over again in an environment and then describe the core of the solution to that problem [Alexander et al., 1977]. This should be done in such a way that the solution can be used a million of times over, without ever doing it the same way twice. This is also true for software-, and design patterns.

In the fall of 1994, Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides, also known as the Gang of Four (GoF) published their seminar book *Design Patterns - Elements of Reusable Object-Oriented Software*. This was the first catalogue of well described design patterns for Object-oriented (OO) software. Since then design patterns have become an essential part of OO software design. Design patterns represent well-known solutions to common design problems, or as defined by GoF: "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [Gamma et al., 1995]. They are the time-honored, battle-tested best-practices and lessons learned [Appleton, 1998]. They have also been proposed as techniques to encapsulate design experience and aid in reuse. Preserving collections of individual design techniques useful for the organization as a whole is a first step towards creating an institutional memory of design techniques, which can be reused in an organization-wide context [Shull et al., 1996]. Another benefit of design patterns is the increased resilience to changes they bring, avoiding system evolution to cause major re-design [Aversano et al., 2007]. Making changes to a complex system can be simpler if the system i built in good OO programming with appropriate use of design patterns, reducing change or global problems [Sanders and Cumaranatunge, 2007].

While using design patterns for forward engineering has obvious benefits, using reverse engineering techniques to discover existing patterns in software artifacts can help in areas such as program understanding, design-to-code traceability, and quality assessment [Antoniol et al., 2001]. Reverse engineering is the process of analyzing a system to identify components and their relationships and representing them at a higher level of abstraction. The objective of reverse engineering is to gain design-level understanding to aid maintenance, strengthen enchantment, or support replacement [Chikofsky and Cross II, 1990]. Design pattern detection and/or recovery techniques have since Gamma *et al.*(1995) emerged from manual processes to automatic and semi-automatic processes. An example of a manual reverse engineering process to recover design rationale is the BACKDOOR technique proposed by Shull et al. (1996). The output of their process is a knowledge base that describes patterns used by an organization [Shull et al., 1996]. More resent work focus more on automatic or semi-automatic techniques to support recovery of design rationale, such as the SPOOL technique used by Keller *et al.*(1999), or the tool proposed by Tsantalis *et al.* (2006) using a graph-matching based approach also tested by Aversano *et al.* (2007). Bergenti and Poggi (2000) takes their design pattern detection techique one step further. The tool presented in their study does not only detects instances of patterns but it also implements a system of critique to the patterns detected [Bergenti and Poggi, 2000].

This paper outlines a reverse engineering analysis of software patterns used in five OO software systems at a small game development organization. The goal is to combine the interest of a specific organization with formalized reverse engineering and design pattern recovery. Subsequently, the main contribution is to illustrate how theory and practice stand to gain from enriching the largely theory-driven related literature of reverse engineering and design pattern recovery, while at the same time deliver suggestions for improvement to the organization.

This rest of the paper is organized as follows: Chapter 2 positions the paper to related research. Chapter 3 outlines the research approach used. Chapter 4 presents the data collected. Chapter 5 presents the analysis of the data, and chapter 6 concludes the paper.

2. Related Research

2.1. Software Architecture and Design

In the 1970s Software Design acquired a great deal of attention from research. This attention arose as a response to the problems of developing large-scale programs in the 1960s. In the 1980s research in the field of Software Engineering moved away from a design specific approach towards integrating designs and design processes into the broader context of the software process and the management of it [Perry and Wolf, 1992].

The architectural design process concerns establishment of a basic structural framework that identifies major components of a system and the establishment between the components [Sommerville, 2007]. Design means to create and through this to give meaning and order to the environment [Stolterman, 1999].

The Software design process is about making decisions regarding the logical organization of software. This logical organization is sometimes presented in different kind of models such as Unified Modeling Language (UML) diagrams or informal notations and sketches. The process of designing Object-Oriented architectures concerns designing objects and classes and the relationships between them [Sommerville, 2007]. The objects in an OO design are directly related to the solution to the problem.

Although the benefits of good architectures and designs are clear to most software producing companies, some neglect proper documentation of such. This can bring two issues. The first concerns how companies can preserve design knowledge and experience, why did we do what we did. The information exists mainly in the minds of the people and in the source code. The second concerns how to more effectively integrate newcomers into the company's design culture. In such cases where little focus are put in documenting design, ways to preserve design rationale can be useful. Designs can be made more reusable and self explaining by documenting the patterns used [Odenthal and Quibeldey-Cirkel, 1997].

2.2. Software Patterns

In software architecture and design, patterns can exist on different levels and cover various ranges of scale and abstraction. Buschmann *et al.* (1996) groups patterns into three categories, *architectural patterns*, *design patterns*, and *idioms*. Architectural patterns express the fundamental structural organization for software systems and represent the highest level of abstraction in this pattern system. Bass et al. (2003) describes an architectural pattern as *"... a description of elements and relation types together with a set of constraints on how they may be used"*. The authors distinguishes an architectural pattern from the two concepts *reference model* and *reference architecture* but show the link betweens them. Their description of a reference model is *"... a division of functionality together with data flow between the pieces* [Bass et al., 2006]. The reference architecture is described as ... *a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them* [Bass et al., 2006].

One of the best known architectural patterns is the Model-View-Controller pattern (see figure 1).

Design patterns represent the middle level of abstraction

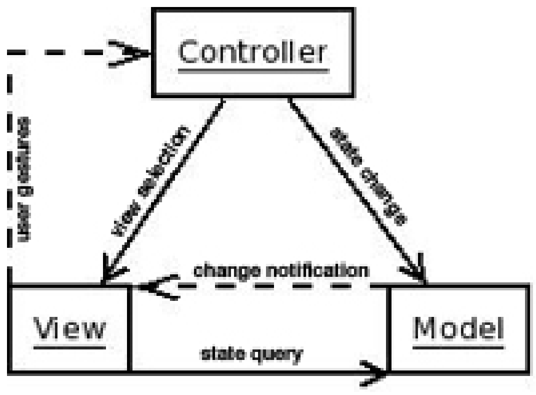

Figure 1. Example of the Model-View-Controller pattern.

of the Buschmann *et al.* (1996) system of patterns. They are smaller in scale then architectural patterns, but are higher abstractions then idioms. Working with design patterns help developers communicate design or architectural decisions. They provide a common vocabulary and remove the need to explain a solution to a particular problem with a lengthy and complicated description [Buschmann et al., 1996]. Design information can be communicated more quickly and accurate through design patterns [Vlissides, 1995]. A unified understanding of common design techniques can therefor also be helpful for developers to comprehend programs and code produced by others.

Another key aspect of design patterns is their ability to ease the task of making changes in a complex software program. Sanders and Cumaranatunge authors of the *Action Script 3.0 Design Patterns* summarizes design patterns as *"...tools for coping with constant change in software design and development"* [Sanders and Cumaranatunge, 2007].

Odenthal and Quibeldey-Cirkel (1997) stress the dual nature of design patterns: that they are both *generative* and *descriptive*. The attribute generative refers to a pattern's content: the objects constituting a micro-architecture. The attribute descriptive refers to a pattern's form: the way we capture and articulate this thing [Odenthal and Quibeldey-Cirkel, 1997]. The interplay between the form and the content promotes, according to the two authors, the principle of documenting by designing.

GoF divides design patterns into three classifications: *creational*, *structural*, and *behavioral*. Creational patterns are concerned with object creation. Structural patterns are concerned with capturing the composition of classes or objects. Behavioral patterns are concerned with the way which classes or objects distribute responsibility and interact [Gamma et al., 1995]. According to Antoniol *et al.*(2001) the complexity of extracting information from a design or source code is not the same for the different kind of patterns. Structural patterns information is explicit in their syntactic representation; the purpose of the pattern is visible on a abstract level by just looking at the relationships between components.

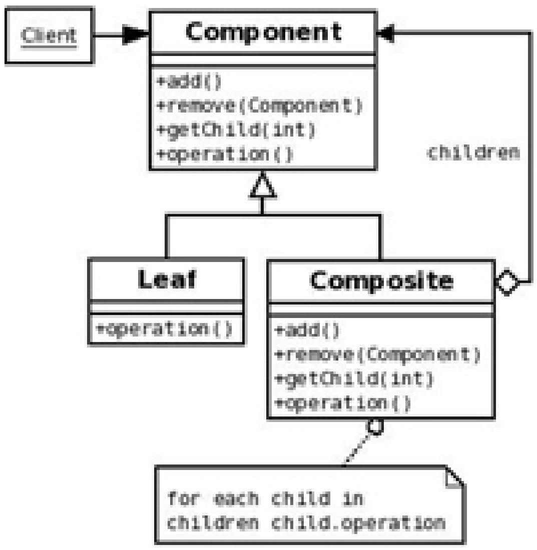

Figure 2. Example of the Composite pattern, a structural pattern included in the GoF collection.

This is not the case for the other two categories. In their cases information must be recovered more deeply within, which may involve the analysis of messages exchanged and the code of class methods [Antoniol et al., 2001]. In other words one must look inside the components and analyze their functionality in order to comprehend the purpose of the pattern.

Further, Gamma *et al.*(1995) divides a single pattern in different elements. Among them there are four essential elements: The *pattern name* is a handle of a word or two used to describe a design problem, its solution, and consequences. The *problem* describes when to apply the pattern. The *sulotion* describes the elements that makes up the design, their relationships, responsibilities, and collaborations. The *consequences* are the results and trade-offs of applying the pattern.

An idiom is a low-level pattern that describes how to implement the particular aspects of components and their relationships using the features and syntax of a specific language. They are at the lowest level of the pattern system, and specific in their representation [Buschmann et al., 1996]. Because of their low level of detail idioms are less portable between different programming languages then design patterns.

Since some idioms describe the concrete implementation of a specific design pattern [Buschmann et al., 1996], it can in certain cases be hard to draw a clear line between the two. To exemplify an idiom embedded in a design pattern we can use the *Singelton* pattern. The solution and example of the design pattern description would look quite different in a language as Smalltalk then it would in ActionScript 3.0. The following example of a Smalltalk implementation is taken from Buschmann et al., (1996):

Solution - Override the class method `new` to ensure only one instance of the object is created. Add a class variable `TheInstance` that holds a single instance. Implement a class method `getInstance` that returns `TheInstance`. The first time the method `getInstance` method is called, it will create the single instance with `super new` and assign it to `TheInstance`.
Example

```
new
  self error: 'cannot create new object'

getInstance
  The instance isNil ifTrue: [TheInstance
  := super new].
  ^ TheInstance
```

A description of the singleton design pattern implemented in ActionScript 3.0 could look like:

Solution - Add a variable `_instance` that holds a single instance of the `Singleton` class. Apply a `PrivateClass` as parameter to the `Singleton` class construct method used to assure that no other object can instantiate the `Singleton`. Implement a method `getInstance()` that returns a `PublicClass`. When the `getInstance()` method is called it first checks if the variable `_instance` points to an instance of the `Singleton` class and if not instantiates a new `Singleton` and points the `_instance` variable to it. Finally the method returns the `_instance`. Close the `Singleton` class and the package. Add a class `PrivateClass` with a constructor method `public function PrivateClass()`. This is the class used to assure that only the `Singleton` class can call its own constructor method.
Example

```
package {
public class Singleton
{
 private static var _instance:Singleton;
 public function Singleton(pvt:PvtClass){
 }
 public static function getInstance():
 Singleton
 {
  if(Singleton._instance == null)
  {
   Singleton._instance = new Singleton(
   new PrivateClass());
  }
  return Singleton._instance;
 }
}
}
class PvtClass
```

```
{
 public function PvtClass() { }
}
```

The underlying programming language is important to consider when designing for the use of design patterns since some patterns are not applicable in all cases [Gamma et al., 1995] or may look different for different languages [Buschmann et al., 1996]. ActionScript 3.0 is based on ECMAScipt [Sanders and Cumaranatunge, 2007], a scripting language standardized by Ecma International in the ECMA-262 specification and ISO/IEC 16262 [Ecma International, 2009]. Since the ECMAScript specification does not support private classes, the implementation of the Singleton pattern in ActionScript 3.0 utilizes a technique of creating a class in the same .as file as the Singleton class but outside the package. This way only the Singleton class can instantiate the class and use it as parameter for the call to its constructor method.

2.3. Reverse Engineering

Reverse engineering is defined by Chikofsky and Cross II (1990) as "the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction"

The goal of reverse engineering is to develop a more abstract or global picture of the subject system [Keller et al., 1999]. Generally reverse engineering concerns extraction of design artifacts and to build or compound abstractions that are less implementation-dependent. The process in, and of, it self does not include changing the system or creating a new system based on the reverse engineered subject system. Thus, reverse engineering is a process of examination, not a process of change or replication. The general purpose of reverse engineering is by increasing the overall comprehensibility of a software system to aid both in maintenance and future development [Chikofsky and Cross II, 1990].

In cases where a system documentation is scarce, code can be one of the few reliable sources of information about the system. In response to this reverse engineering has focused on understanding the code [Müller et al., 2000]. However, not all information needed to fully understand a system can be found in the source code. Knowledge about architectural decisions, design trade-offs, engineering constraints, and the application domain exist in the minds of the Software Engineers [Bass et al., 2006].

John Vlissides (1995), one of the GoF authors, introduces a concept of reverse architecting as a contrast to reverse engineering. He argues that, since reverse engineering software focus rather on recovering design then implementation, the term reverse architecture would be more appropriate. He extends his concept reverse architecting with reverse architecture which would apply to the art of science behind the activity. Reverse architecture then refers to an analyze of many software systems in an effort to recover recurring designs and the rationale behind them [Vlissides, 1995].

According to Chikofsky and Cross II design recovery is a subpart of reverse engineering. In addition to just examining the system design recovery adds domain knowledge, external information, and deduction or fuzzy reasoning to the observations [Chikofsky and Cross II, 1990]. Ted Biggerstaff argues that design recovery must reproduce all of the information required to fully understand what a program does, how it does it, and why it does it [Biggerstaff, 1989]. Recovering design patterns may well serve Biggerstaffs argument as part of their purpose is to answer the questions what, how, and why. Yet, Keller *et al.*(1999) argue that most of the reverse engineering tools ignore to recover the rationale of the design decisions that have led to the shape of the programs [Keller et al., 1999]. One reason for this could well be because, even if patters are a suitable language for humans, they are are a rather complex language for automated systems [Bergenti and Poggi, 2000].

3. Research Approach

3.1. Action Case

The study follows an Action Case approach for research method. Vidgen and Braa, (1997), suggests the Action Case approach as a supplement to other approaches falling in between more traditional research methods. The Action Case approach then positions between an intervention approach and an interpretative approach. Thus we strive to gain increased understanding of the research domain while at the same time acquire purposeful change.

The characteristics of the Action Case study approach, as proposed by Vidgen and Braa, (1997), can been seen as follows: First, the scope of the study is restricted such that small scale interventions are made. This to gain understanding of IS used and at the same time achieving a rich understanding of the context. This was important in this study since we wanted to be able to enrich the theory of reverse engineering and design pattern evaluation, while at the same time deliver and test suggestions of improvements for the organization. Second the time frame for the study will typically be short to medium. This was suitable since the time frame for the study is short. Third, the intervention should be focused and direct so that the change related effects can be studied in detail. Fourth, the action case study will take from the action research a concern with building the future through purposeful change while still maintaing an interest in the historical conditions in which the study is set.

3.2. Research Setting

The study took place at a small game development company in Gothenburg. Most projects at the company are relatively small and developed within a, from a IS production view, short period of time. To save time in every project, minimum such is spend on designing architectures for each one of them. Instead, reuse of architectural structure and design have gained high priority. As an aid they have built large portions of their software based on reference models, architectural patterns, and design patterns. This gave a rich setting for looking for, and analyzing software patterns as we knew they were there, but got no design documented telling us where to find them. Many of the patterns they use comes along with the different external frameworks they follow, and it was expected that not always had they fully comprehended all of them before integrating them into their projects. This was important since it meant we knew that for certain part their understanding of the design could grow and there by opened up room for interventions.

3.3. Research Process

A pilot-study marked the starting point of this research project. The goal was to establish a firm starting point so that related literature could be identified that would be likely to have particular relevance to the setting.After the pilot-study, the research continued by performing a literature review on areas connected to the research topic. This was important for me as a researcher to gather in depth information since my experience in the field was limited. The literature study is not reported as a whole in this paper. Rather, the core elements identified are what make up section two of this paper. Furthermore, techniques for data treatment identified during the literature review are used and presented in section four. The data was collected using three different techniques, Direct observations of technical artifacts, Conversations with the developers, and a Formal Interview.

3.3.1 Direct observation of technical artifacts

The technical artifacts were studied in-context at the organization. This provided a rich setting for investigating the work, management, and technology issues associated with IS research [Braa and Vidgen, 1999]. Since no documentation existed, the source code and the developers were the only sources of information regarding architectural and design decisions taken in the past. This process of collecting technical data followed the BACKDOOR design recovery technique [Shull et al., 1996] but with modifications to better fit the Action Case method. The process was then in accordance with the technique divided in six smaller steps. Although these steps are defined in a sequent order here, they where really iterated within and across steps.

During the first step the five projects was briefly been studied. This was done for me to gain a general understanding of the overall architectural structures and features of the systems. The architectural similarities between the five different projects, immediately clear to a trained eye, served as a good starting point to where to start looking for instances of patterns. Also this is helpful to get a rough idea about the level of complexity of the source code and thereby the level of difficulty to reverse engineer.

The second step concerned automatically reverse engineering the projects into Class diagrams according to the Unified Modeling Language (UML) notation. The software Architect Enterprise was used as a tool to aid this process. It was an important step since it allowed for a more clear overview of the components and constructed diagrams to use in later phases of the study.

During the third step I took a deeper look at the code implementing the classes, trying to figure out more about how the components communicate with each other. This process was based on the diagrams from step two and served as an aid to clarify them. The process was a manual activity of analyzing the code where the diagrams could not clearly show the communications and relations between the objects. Therefor it also served as a link between step two and four to enable more accurate detection of pattern instances.

The forth step was to manually trying to detect pattern instances in the projects. A couple of different sources was used to aid in this process. First the he automatic reversed engineered diagrams from step two was used to give a more clear overview of the components. Secondly the formal descriptions of the patterns from the reference set was used as a help to know where to start looking. When looking for structural similarities Shull *et. al* (1996) defined a couple of indicators can serve as a useful aid.

- Classes that serve as the receiving end of communication links from many other classes could play a mediating role in the interactions between these other classes.
- A class positioned at the sending starting point of links with many other classes may act as an interface to those classes.
- Classes that have parallel inheritance are likely to be working together closely.
- An object which link two clusters of classes with high coupling may be sophisticated communication link between two subsystems.

This, the forth step, did not aim at finding any ad hoc design solutions the could make a possible candidate for becoming a pattern. There were simply no time for such activity, but this might be an interesting future task.

The fifth step concerned analyzing the patterns detected. To aid in this process the detected patterns where compared to the formal descriptions of their original purpose and structure to see how well they match or deviate. There was a need during this step for a technique for assessing the potential pattern matches found. This system of measure was developed based on the ranking metric used by Shull

et. al (1996), but since the intended purpose of pattern detection in this study differ from their purpose the measuring system needed to be modified in some aspects. To be able to do so the patterns were divided into three main components, and then each component was compared to the corresponding pattern in the reference set:
Structure - The structure of the pattern refer to the classes, objects, and relationships that builds the pattern. How well does the pattern match the *structure* of the corresponding pattern in the reference set.
Purpose - What effect will the pattern have on the system. Is the pattern trying to solve the same *problem* as the corresponding pattern in the reference set, and are the *consequences* the same.
Implementation - How well does the pattern match the *solution* aspect of the corresponding patter in the reference set.

The following scale have been applied to the three categories on all projects:
1 = Not relevant
2 = Similarities, but not close match
3 = Parts match close to perfectly
4 = Near perfect match

The sixth and final step was to clarify design issues in discussions or interviews with the developers. As the founders of BACKDOOR puts it: *"Such interviews can sometimes be the only way to gain an understanding of what context issues influenced a particular design, or how various subsystems interrelate"* [Shull et al., 1996]. This sixth step was included in the following two techniques for gathering of data.

3.3.2 Conversations with the developers

The second technique were conversations with the developers. While analyzing the projects and reverse engineering the code informal conversations were held with the creators of the code to better understand how and why certain solutions have been chosen for certain problems. This was considered important since not all aspects the design of the architectures was expected to me clear to me as an external viewer. This then aided me in understanding the reasoning behind technical solution not directly clear. The conversations were held informal and took place as questions arose.

3.3.3 Formal interview

The third technique was a formal interview. When enough technical data had been collected an interview was held to aid the analyze of their understanding of the patterns used. The interview followed a semi-structured interview technique to allow for reflection while still having a clear direction. The interview was mainly held for me as a researcher to understand more of the reasoning behind their design. Additionally during the interview the technical analysis was presented to be able to discuss and reflect with the interviewee regarding the findings and thereby increase the understanding of the patterns under study.

4. Empirical findings

4.1. Presentation of Architectural Patterns

As representation for architectural pattern the Model-View-Controller was chosen. This decision was taken upon a few different factors. The pre-study conducted at the company showed that all the projects under study had high potential of showing similarities with the pattern. Additionally the foundation of the micro architecture Cairngorm, which some of the projects followed, builds on a mvc structure.

The pattern description in the reference set is a compilation of the general descriptions of the pattern taken from Buschmann *et al.* (1996), and Sanders and Cumaranatunge (2007), and was defined as follows:

The Model-View-Controller is a compound micro architecture built by multiple design patterns [Sanders and Cumaranatunge, 2007]. The pattern consists of, as the name implies, three main elements, *model*, *view*, and *controller*. The model contains the core functionality and data to manage the state of the application. The view presents the state of the application to the user. The controller handles user input. A change propagation mechanism should ensure consistency between the user interface and the model [Buschmann et al., 1996]. The diagram modeling the graphical representation of the pattern looks as figure 1.

The following table shows the scores for how well each of the projects match the MVC pattern from the reference set:

Table 1. MVC Architectural Pattern measurement

Project	Structure	Purpose	Implementation
IFS	4	3	3
Magic 5	4	3	3
Intersport	4	3	3
Puma	3	2	2
Volvo	3	2	2

Structure Three of the projects match the structure of the reference pattern relatively well. These are also the three pattern that heavily builds on the Cairngorm framework. In fact these three projects follows a very similar architectural structure, and will hence forth be refereed to as the three Cairngorm projects. The other two projects have a foundational structure that matches well to the reference pattern, but specific object are missing and some relationships are also missing to be a complete match.
Purpose It seems like in all projects the MVC pattern has been integrated for the purpose of giving an architectural structure to the projects, but not really to solve the main problems the original pattern aims to solve. Still the three Cairngorm project matches parts of the patterns purpose close to perfectly. But all projects seems to miss, or use

in a way not as intended by the original pattern, one major part, namely the change propagation mechanism.
Implementation As with the purpose, none of the projects have a perfect match in implementation of the pattern. Still the three Cairngorm projects come close. They have a very sofisticated implementation, which seems to deviate from the original purpose for the sake of working around some unwanted consequences of the pattern.

4.2. Presentation of Design Patterns

As representation of Design Patterns the Commands pattern was chosen for the reference set. The pattern represent a major part of the Cairngorm framework used in some of the projects. This is important since it gives a great hint of where to start looking, but also ensures that there will be some matches. The class diagram of the pattern is presented in figure 3. The pattern description in the reference set is based on the general description of the pattern from GoF [Gamma et al., 1995], and follows:
Intent To encapsulate an object, and thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. These are the participants of the pattern:

Command Declares an interface for executing an operation

ConcreteCommand - defines a binding between a Receiver object and an action and implements Execute by invoking the corresponding operation(s) on Receiver.

Client - Creates a ConcreteCommand and sets its receiver.

Invoker - asks the command to carry out the request

Receiver - knows how to perform the operations associated with carrying out a request. Any class may serve as a receiver.

Motivation Sometimes it is necessary to be able to issue requests to objects and still not know anything about the operation being requested or the receiver of the request. The key to the Command pattern is an abstract Command class, which declares an interface for an abstract Execute operation.

Table 2. Commands Design Pattern measurement

Project	Structure	Purpose	Implementation
IFS	4	4	4
Magic 5	4	4	4
Intersport	4	4	4
Puma	1	1	1
Volvo	1	1	1

The Commands patter is not utilized in the projects Puma and Volvo and therefor not relevant.
Structure The Commands structure of the three Cairngorm projects differ in some minor aspects to the original pattern.

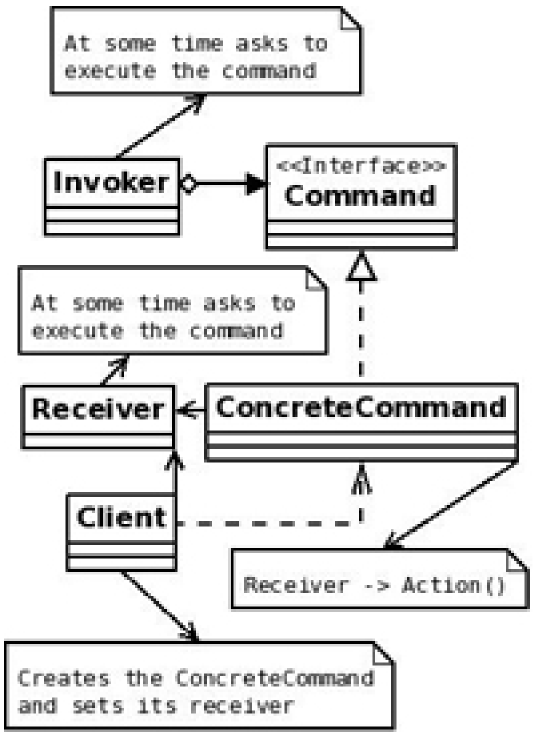
At some time asks to execute the command
Invoker
<<Interface>>
Command
At some time asks to execute the command
Receiver
ConcreteCommand
Client
Receiver -> Action()
Creates the ConcreteCommand and sets its receiver

Figure 3. The graphical representation of the Commands Design Pattern from the reference set.

Still all the components are represented and in most aspects the structure is close a perfect match. The *commands* is represented by the class ICommand. The *ConcreteCommand* is represented by the GeneralCommand class. The *Client* is represented by the FrontController class. The *Invoker* is represented by all the event triggers spread out in the projects but is not a part of the class diagram in figure 5. The *Receiver* is represented in some cases by the Model, in some cases by the different View classes, and in some cases other classes or even other Commands.
Purpose The intent of using the pattern in the three Cairngorm projects matches close to perfect the patterns original purpose.
Implementation The implementation of the pattern in the three Cairngorm projects is sophisticated and matches well to the original pattern from the reference list. The projects extend the pattern even further, implementing a pattern of chain where every pattern has a reference to an next command in the chain, or null if none.

4.3. Presentation of Idioms

As representation for Idioms the Singleton pattern was chosen. The pre study showed that there are different implementations of the Singleton pattern spread through out the projects which made it an interesting pattern to base the Idiom pattern match on. The pattern description in the reference set follows is based on GoF [Gamma et al., 1995] and

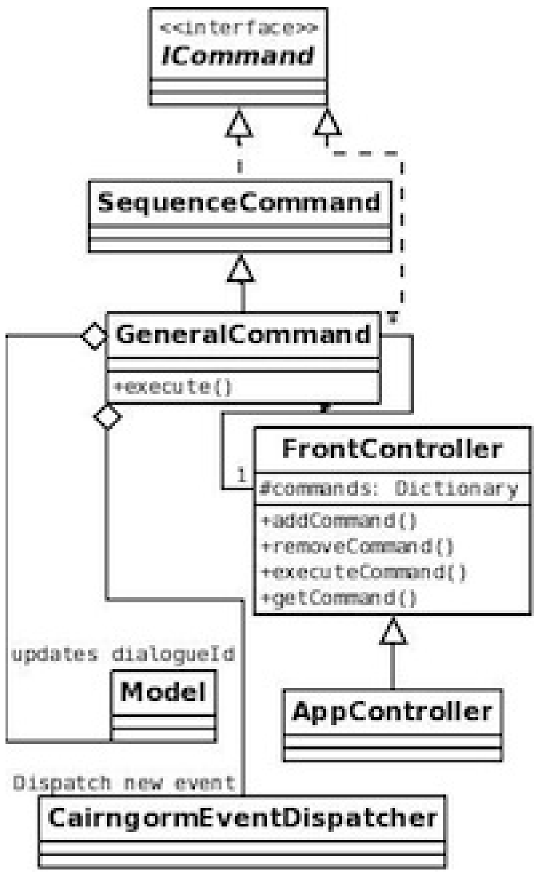

Figure 4. The graphical representation of the Commands Design Pattern as implemented in the three Cairngorm projects.

follows:

Intent Ensure that a class only has one instance, and provide a global point of access to it.

Motivation In some cases it is important that a class can have no more then one instance. The solution to assure this is to let the class itself be responsible for keeping track of its own instance. This class should ensure no other instances can be created, and should provide a global access to the instance reference.

The Singleton pattern exists in many occasions and in different implementations through out the projects. Therefor the name of the class is added to the table of Singleton Idioms. Also, since many of the Singleton object follow the same implementation, representatives from all projects are included but not every instance of the Singleton classes. In the projects where two, or more, different implementations of the pattern exists, representatives for each implementation will be included. Since Idioms are so code specific the category Structure has been removed.

Purpose The analyze of the patterns shows two groups of Singletons in the projects. In the first group are Singletons that score low on the measurement scale which will be referred to as the low group, and in the second are those that score high which will be referred to as the high group. The intended purpose of the low group seems to lie close to the reference pattern, the actual consequences are far from matching. The purpose of the high group match close to perfectly.

Implementation The implementation of the low group lies far from the reference pattern. It allows for multiple instances of the class, which violated the main intention of the pattern. Still it has a global access and it does hold it own reference, even though other classes can create and hold that reference. For the high group the implementation is close to a perfect match.

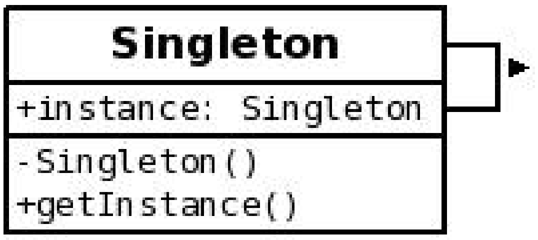

Figure 5. The graphical representation of the Single Design Pattern in the reference set.

Table 3. Singelton Idiom measurement

Project	Class	Purpose	Implementation
IFS	ApplicationModel	3	2
IFS	SoundController	4	4
Magic 5	ApplicationModel	3	2
Magic 5	SoundController	4	4
Intersport	ApplicationModel	3	2
Puma	Model	3	2
Volvo	Model	3	2

4.4. Interview

The interview was conducted following a semi structured interview technique. It was held in swedish, recorded on audio, and then transcribed from audio format into text in english.

The interview was performed for two main purposes. First it helped the study clarify certain aspects of the technical data. In the final step of BACKDOOR, as proposed by Shull *et al.* (1996), conversations or discussions can to be held with the developers of the software. How ever skilled the person performing the manual design evaluation is, one can only make qualified estimations or guesses around certain aspect of the design. Therefor this is an important step for the researcher to be able to test the technical data on the developers to increase the understanding of the rationale behind the designs. The second reason was to, by presenting the findings of the technical analysis, try to increase the developers awareness of their implementation of the three patterns.

The interview was divided into four topics, the first three being the patterns from and the reference set and the last regarding the future. However, all questions where either touching upon one of the two purposes for the interview. The discussions regarding the technical data will not be presented here, but rather as part of the analysis. The following section summarizes the discussion around how the analysis of the technical data could help the organization in the future.

The developer thinks the a greater understanding of design patterns and architectural structures would aid them in their everyday work. As he puts it:

As an example we have had a external developer in-house for six months who comes from a background as a educated programmer and understood well architectural patterns. This way she could easy grasp our structure and understand why it was designed this way. This is not the case for most of our developers, as they come more from a background as graphical designer. Thus an increased understanding can lead to them being more effective in their work.

Even though the developer had never seen any problem with their implementation of the Singleton pattern he did admit it could become a potential problem in the future. This has directly increased their own understanding of their pattern.

The developer believes that increasing their understanding of patterns, as has been done in this study, can increase their possibility to reuse code.

5. Analysis

5.1. Software Architecture and Design

For a smaller companies, such as the one in this study, where most projects are short, the process of documenting the design can have lower priorities then other processes more directly focused on the implementation of software. How then can such companies still build robust and correct software? The solution at the company has largely been dependent on external frameworks for how to structure projects such as theirs. One of these external frameworks is the Flex and Action Script specific micro architecture Cairngorm [Adobe, 2008]. Cairngorm can be seen as an architectural pattern as it is defined by Buschman *et al.* (1996), but it can also be refereed to as a reference model as defined by Bass *et al.* (2006). Their definition of a reference model follows: *"... a division of functionality together with data flow between the pieces"*. An architectural pattern and a reference model is, according to them, not the same. The fundamental structure behind the framework is the Model-View-Controller which is seen as an architectural pattern [Buschmann et al., 1996], but it also extends further and in its completeness gives deep information of how the data flows between the elements, and in that aspect could be seen as an reference model [Bass et al., 2006]. Therefor I refer Cairngorm to both a reference model and an architectural pattern dependent on the situation.

A reference model does not make an architecture, nor does an architectural pattern or any other software pattern for that matter. They are concepts that capture elements of an architecture [Bass et al., 2006]. Most software systems are to complex to be structured according to one single architectural pattern, and have requirements that can only be supported by different patterns [Buschmann et al., 1996]. For the organization studied in this research, most of the projects under study followed at least two architectural patterns, but also integrated smaller components and micro architectures from their home made in-house framework. When it comes to architecture the developer at the company gives some hints that they would like to be able to document the design, but they lack both skills and time to do so. During the process of reverse architecting the five projects, a basic architectural model over each of the projects where created. This gave us a clear view over the structure of the projects and their components. But it did not say much about the reasons behind the design decisions. The problem here lies in the complexity of automatically reverse engineer code. As an example from the diagrams produced in this study, no cross-package references are drawn. This became a big problem for understanding how the components of the architectures interact, especially for the projects based on Cairngorm where most parts of the reference model i placed in a different package [Flex-Cairngorm-Community,]. To get deeper understanding of the design we had to move further into the reasoning behind it; we had to look specifically at the software patterns building up the architecture.

5.2. Software Patterns

The reasoning behind the patterns used in the projects lies to some extent in the minds of the developers, but in the company's case it also lies a lot in the hands of the creators of the external frameworks and patterns used. As the interviewed developer puts it when I asked if some of the patterns in the project are there simply because they are a part of the external frameworks, rather then being picked by them for their specific purpose: *"Yes that is likely to be the case. For example it could be the case that we know how to use the Command pattern, but not why we should use it."*

The architectural structure of the Model-View-Controller pattern in the projects complies quite well to its original structure, especially in the Cairngorm projects, but when it comes to purpose and implementation of the pattern it does not score as high. This could be an effect of the reasoning behind using the pattern as a way to structure projects rather then the specific consequences of the pattern. As an example, one of the main benefit of the pattern is that you can easily have multiple views of the same model [Buschmann et al., 1996]. This was not something the company had considered a usefulness for their projects as they never thought of having different views of one model. This illustrates how a limited understanding of patterns original purpose may lead to a less sophisticated implementation. By reverse engineering the projects we have found that the potential usage of the pattern can be extended in future development.

Another example where there implementation suggests a Publisher-Subscriber pattern, also known as Observer, to implement the change mechanism of the MVC pattern. The company has decided to implement the pattern without having a change mechanism for the model to notify observing objects that it has updated. This clearly limits the pattern, since it will remove many beneficial effects from the list of consequences, such as multiple views of the same model and synchronized views [Bass et al., 2006]. On the other hand, the solution of the company also removes or eases some of the unfavorable effects such as, increased complexity, potential for excessive number of updates, and close coupling of views and controllers to a model. Thus, while the implementation clearly limit the pattern from many potentially beneficial effects, the limitations themselves are not simple mistakes during implementation, but rather conscious choices made to limit potentially negative effects.

Another reason for a looser coupling of the views and the controllers to the model is the application of the Commands design pattern to the Cairngorm projects. The pattern is a essential part of the framework and many beneficial consequences comes from using this particular pattern. As the pattern allows for clients to issue requests to objects without making assumptions about the request, or the receiving object [Sanders and Cumaranatunge, 2007], it decouples the sender from the receiver.

Two of the projects did not utilize the pattern, and therefor they have been excluded from this analyze. It could have been interesting to compare these two projects with those using the pattern and compare differences in how well we can understand them, but it lies beyond the scope of this study.

The other three projects that uses the pattern, does so in a sophisticated manner that complies well to the original purpose of the pattern. It seems like the Commands pattern was integrated into these projects, not only for being a part of Cairngorm, but the beneficial aspects of this specific design pattern was was taken into consideration by the company. This is how the developer reflected on why they use the Commands pattern: *"I wanted to break out things, get logic away from views."*

The pattern has also been a way for them to increase the understanding of the code for the developers. As the interviewee explains how the separation of logics from the views and graphical parts of the systems allowed for a more separate division of staff, letting those with higher programming skills work more on code, and those with less such skills work more on graphics.

The understanding of patterns can differ from individuals in the same organization, or even in the mind of one developer over time. As an example of this we can look at the Singleton data. It has been implemented in different ways through out the projects. Some instances of the pattern complies well to the original purpose, and some instances deviates much.

5.3. Reverse Engineering

The reverse engineering the five projects was a time demanding process. Action script 3 is a rather new language. Few softwares for automatic reverse engineering such code exists today. The software chosen in this study did support action script 3.

The diagrams produced in this process did comply with the Chikofsky and Cross II (1990) definition, as the components and the relationships where represented in a higher level of abstraction. One aim of the reverse engineering process for the company was to find out if it could aid them in future reuse of components and structure, as well as in maintenance of code. The ultimate goal for them was to have a system of synchronized architecture and code, where developers could create and change software either by writing code or by drawing components in a diagram connected to the software. The reverse engineering software used in this study does support this, but it lies beyond the scope of the study to investigate further into.

Though the architecture was remodeled in accordance to the formal definition of reverse engineering, it gave very little information about the rationale behind the design. To be able to capture the answers to the what, why, and how the systems where built I manually remodeled parts of the architectures in even more abstract diagrams. These new diagram gives a better overview of the design of the projects and thus a more stable foundation for detecting and analyzing patterns.

5.4. Key findings

This study shows us that the reasons for integrating patterns in software design can be different from what the literature of software pattern usually propose. As the developer at the company describes regarding their choice to start building their projects according to the reference model Cairngorm: "... if we at least could have a set foundational structure of how to build things, and thats when Cairngorm came into the picture as an MVC pattern. This led to that people new to a project could easier find where to start looking for the problems." Other common quality attributes, such as reusability or robustness, were no major issues for their choice. Even so, Bass *et al.* (2006) argues that a useful aspect of architectural patterns is that they exhibit quality attributes. This has led to a situation where the company unintentionally has built products with quality attributes inherited from the consequences of the patterns in the reference model. What we have found here is that by pattern detection and evaluation organization can discover new quality related attributes in their software systems. Thus reverse engineering and pattern evaluation, as the one performed in this study, can support organizations discover new or potential candidates of features of their own software.

But what is the effect of misinterpreted pattern purposes and implementations and are there any drawbacks. One finding of this study is the problem of reusing an architectural pattern. The interviewed developer saw a problem in that he thought that having a set architecture based on a reference model in their case have inhibited the creation of smaller reusable components. A reference model such as Cairngorm is general and abstract and the smaller project-specific components have to fit the architecture, instead of an architecture fit for the problems. Such architectures should be more customized and de-generalized for it to serve the complexity of software [Brooks, 1987]. Understanding the architectural patterns behind the systems increases our ability to specialize them for a more specific purpose.

The reverse engineering of the five projects show that the three projects following the Cairngorm architecture are very similar in structure. Even though the developer argues that the framework inhibits reusability of specific components, this study illustrated how it support reusability on an higher architectural level as the structure of the three projects were very similar. Thus reverse engineering can help organizations rethink their understanding of their own patterns.

5.5. Limitations and future research

The software used for reverse engineering the projects did not comply to the needs of the pattern detection. To manual detect and evaluate pattern put high demands on the person or group performing the process. Therefor, technical drawback are unwelcome and it is necessary to know the limitations of the softwares used in the process in beforehand. I as researcher propose that a study such as this one could gain in complexity, have more proper evidence, and thus greater findings should it have been a quantitative study performing automatic pattern detection for the technical gathering of data. As mentioned earlier the decision to use a manual technique was taken based on the researcher disability to understand and make use of the complex algorithms that comes with the automatic pattern detection techniques. On the other hand, the manual process have led to a great understanding of the organizations code for me as researcher, as I have been forced to dive deeply into their systems.

The patterns used in this study where chosen because we knew they would exist in some instances in the projects. The reason for this was that since we rather discuss how reverse engineering can help us understand software, then how well we can discover patterns. A future study could look for recurring ad hoc solutions that can make possible candidates for patterns, instead of having already well known pattern in a reference set. This is something the organization asked for, but time limitations forces us to push the idea in the future.

6. Conclusion

Software patterns represents the rationale behind software design. They are the well tested best practices to solve common software related problems. To understand the patterns used in a software can aid grasping the question for what, how, and why we have done something in a particular way. This study has illustrated how reverse engineering software code into abstract models to discover and analyze the design can help organizations better understand the software patterns used. Systems inherits quality attributes from the software patterns it holds, and therefor we have found that organizations can discover new non functional features of their systems through pattern detection and evaluation. This way an increased understanding of software patterns can aid organizations in future development by the new quality attributes discovered, such as changeability, maintainability, portability, etc. Manual design detection techniques, such as BACKDOOR, can serve as a great aid in understanding software where design documentation is sparse, though it puts relatively high demands on technical skill of the person or group of persons performing the manual pattern detection. Lack of a reverse engineering software that can comply to the need of in depth design information from the diagrams produced, limits the ability to analyze the rationale behind the projects. Thus high skilled designers with access to well working tools and techniques for pattern detection are key to best grasp the architectures and designs of software through reverse engineering.

References

[Adobe, 2008] Adobe (2008). Introducing cairngorm. Technical report.

[Alexander et al., 1977] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A Pattern Language: Towns Buildings Constructions*.

[Antoniol et al., 2001] Antoniol, G., Casazza, G., Penta, M., and Fiutem, R. (2001). Object-oriented design patterns recovery. *The journal of Systems and Software*, 59:181–196.

[Appleton, 1998] Appleton, B. (1998). Patterns and software: Essential concepts and terminology.

[Aversano et al., 2007] Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., and Di Penta, M. (2007). An empirical study on the evolution of design patterns. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 385–394, New York, NY, USA. ACM.

[Bass et al., 2006] Bass, L., Clements, P., and Kazman, R. (2006). *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Bergenti and Poggi, 2000] Bergenti, F. and Poggi, A. (2000). Improving uml designs using automatic design pattern detection. In *In Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000*, pages 343–336.

[Biggerstaff, 1989] Biggerstaff, T. J. (1989). Design recovery for maintenance and reuse. *Computer*.

[Brooks, 1987] Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Somerland, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture - A System of Patterns*, volume 1 of *Software Design Patterns*. WILEY.

[Chikofsky and Cross II, 1990] Chikofsky, E. J. and Cross II, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7.

[Ecma International, 2009] Ecma International (2009). Standard ecma - 262. Technical Report 5th edition, Ecma International.

[Flex-Cairngorm-Community,] Flex-Cairngorm-Community. http://www.cairngormdocs.org/.

[GaiaFramework,] GaiaFramework. http://www.gaiaflashframework.com.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Menlo Park.

[Keller et al., 1999] Keller, R. K., Schauer, R., Robitaille, S., and Pagé, P. (1999). Pattern-based reverse-engineering of design components. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 226–235, New York, NY, USA. ACM.

[Müller et al., 2000] Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R., and Wong, K. (2000). Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA. ACM.

[Odenthal and Quibeldey-Cirkel, 1997] Odenthal, G. and Quibeldey-Cirkel, K. (1997). Using patterns for design and documentation.

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52.

[Sanders and Cumaranatunge, 2007] Sanders, W. B. and Cumaranatunge, C. (2007). *Action Script 3.0 Design Patterns*, volume First edition. O'Reilly Media Inc.

[Shull et al., 1996] Shull, F., Melo, W. L., and Basili, V. R. (1996). An inductive method for discovering design patterns from object-oriented software systems. Technical report, Computer Science Department/ Institute for Advanced Computer Studies, University of Maryland, Computer Science Department, College Park, MD, 20742 USA.

[Sommerville, 2007] Sommerville, I. (2007). *Software Engineering*. Addison-Wesley, eight edition edition.

[Stolterman, 1999] Stolterman, E. (1999). The design of information systems parti, formats and sketching. *Information Systems Journal*, 9(1):3–20.

[Vlissides, 1995] Vlissides, J. (1995). Reverse architecture.

[Webster and Tanner, ting] Webster, S. and Tanner, L. (Adobe Consulting). Developing flex rias with cairnform microarchitecture.

7. Apendix A - BACKDOOR

7.1. Step one - pre-study of technical artifacts

During the first step in phase one the five projects have briefly been studied. This has been done for me to gain a general understanding of the overall architectural structures and features of the systems. The architectural similarities between the five different projects, immediately clear to a trained eye, serves as a good starting point to where to start looking for instances of patterns. Also this is helpful to get a rough idea about the level of complexity of the source code and thereby the level of difficulty to reverse engineer.

External frameworks. HelloThereFramework is a compilation of artifacts, libraries, and structures often re-used in HelloThere projects. Two parts of the framework, Gaia framework and Cairngorm, builds on external frameworks for best-practice structures and collections of design patterns.

Cairngorm is a lightweight micro architecture for Flex or AIR applications. It is described as an approach to organize and partition software code [Adobe, 2008]. Cairngorm build heavily on the well known micro architecture Model-View-Controller. All the five projects under study follows a Cairngorm structure. This also means that all the five projects inherently utilizes best-practices techniques integrated in the framework. Now, since the framework provides a guide, and not definite rules, with interfaces for how the architecture should be structured, variations from its intended purpose and the purpose of different patterns may well, and perhaps must, exist. The architectural framework provides a generic starting point for the applications [Webster and Tanner, ting]. Since the Cairngorm is built by a compilations of different design patterns this provides a good starting point for where to start looking for patterns in the five projects. Some of the patterns included in Cairngorm are: ValueObjects FrontController Commands Singleton Model (Strategy the view-controller relationship)

Gaia framework is a front-end for webpages developed in Action Script 2 and 3. It provides solutions to the challenges and repeated tasks that most Flash developers face when developing sites. Such challenges are navigation, transition, preloading, asset management, site structure, and deep linking [GaiaFramework,]. All the five projects integrates the framework for their view structure. HelloThere have interconnected the Gaia framework and Cairngorm so that the View structure of the Cairngorm framework is also implemented in a Gaia framework fashion.

In addition for these two parts there are a number of different micro architectures for supporting best-practice integrated in the projects, but due to the small size and timeframe of the study these part will not be investigated.

Three of the five projects seems to follow a very similar architecture, where all build on the external framework Cairngorm. The last two projects are also very similar in their architecture, and they seem to follow a MVC structure.

7.2. Step two - Automatic Reverse Engineering

The tool Enterprice Architect was used as an aid in this process. The pre-study on tools to automatically reverse engineer Action Script code shows that few exists. Enterprice Architect is the only tool I could find that integrates Action Script source as input. Other ways to automatically reverse engineer the code has been studied but rejected due to the complexity behind the processes. (Should I include an example here?).

Enterprice Architect builds diagrams for all classes and objects in the source. Since the process is automatic, it is extremely fast relative to manually reverse engineer the projects.

But there is one major downside to the diagrams produced. Due to the lack of possibilities to show cross package references between the components, relationships are not always clearly visible. To add to that problem, the specific source code of HelloThere builds on multiple packages to separate all the different components connected to the Cairngorm micro architecture. Many of the different patterns in the structure includes cross package references and are therefor quite hard to detect just based on these diagrams.

This is why the next step becomes so important, looking at how the objects communicate and the relationships among them.

7.3. Step three, Looking at the code implementing the classes and how the objects relate and communicate with one another.

This process will base on the diagrams from step two and serves as an aid to clarify them. It will be a manual activity of analyzing the code where the diagrams can not clearly show the communications and relations between the objects. Therefor it also servers as a link between step two and four to enable more accurate detection of pattern instances. Some of the patterns from the reference list concerns capturing the compositions of objects and classes, and some concerns the way in which classes and objects distribute responsibility and interact.

7.4. Step four, Manual Detection of Pattern Instances.

The manual detection of patterns uses mainly the diagrams from step two as input. During the process, parts of the diagrams will be remodeled manually to integrate the communications and relations discovered in step three into the models. The parts to be remodeled will be the potential matches of patterns.

7.5. Step five, Analyze of patterns detected.

The fit step as to analyze the patterns detected, comparing them to the patterns in the reference set. There was a need during this step for a technique for assessing the potential pattern matches found. This system of measure was developed based on the ranking metric used by Shull et. al (1996), but since the intended purpose of pattern detection in this study differ from their purpose the measuring system needed to be modified in some aspects.

7.6. Step six, Interview transcription

Model-View-Controller

K -Three of the projects follows a Cairngorm structure. Was there a specific problem that you tried to solve when you decided to integrate Cairngorm into your projects?

B -It depends how you look at it. Developers tend to do things ”their own way”. Which works fine for them, but when something happens (that was not planned for), and external developers needs to be added to the project, which always does, it is always a hell for the now comers to get into the projects. On my previous job we where eight developers, and the skills of us differed. And then I thought that if we at least could have a set foundational structure of how to build things, and thats when Cairngorm came into the picture as an MVC pattern. This led to that people new to a project could easier find where to start looking for the problems. But still developers solves things differently, but when when you know sort of where things are.

K -Can you say that after introducing Cairngorm, that it has worked as you wanted, and that new comers faster have been able to comprehend the projects? B -Well, we have used some external recourses, and they have not been used to working with structure. So quite opposite, they get surprised and lost because it's not they way they are used to doing things. Now, instead of just doing things and bloating stuff all over where they them selves know they would find it, they now have understand why they should place things in certain places. This has been a big problem which has led to them being slower then they would have been otherwise.

K -Would it have helped if these developers would have a greater understanding of Design Patterns, and Arcitectural structures?

B -Yes I think that would help. As an example we have had a external developer in-house for six months and she comes from a background as a educated programmer and understood well why MVC patterns are good and why and where to put things. But most often flash developers doesn't have such education, but usually are graphical designers originally, with low understanding for patterns.

K -Regarding the issue with your specific implementation of how the model can tell listeners that it has updated. Is there a problem in there as you see it?

B -It's a unfortunate must.

K -Have you ever considered an Oberver pattern?

B -No, we thought about this a couple yeas back and the solution still exists today as it did then.

K -The two project that do not follow cairngorm? What about them?

B -Well they kind of follow Cairngorm.

Commands

K -When you decided to use Cairngorm, did you reflect on why to use the Commands pattern?

B -I wanted to break out things, get logic away from views. When we started with this the skills of the developers differed a lot, and then by removing as much code as possible away from the binary graphical files into classes, some developer could work more in graphics and some more on code. So a structural division seemed good, but much more then that was not considered. The biggest problem with these commands, sometimes you don't know where to put things.

K -One of the grate features of the Commands is that it allows for a undo function. Have you thought about this?

B -Yes but it has not been utilized.

K -When you look for other solutions, do you ever look on an abstract level, or just action script forums?

B -Just action script.

Singleton

K -In some cases your implementation of the Singleton pattern deviates from its original purpose, namely you have made it possible to create multiple instances of the Singleton classes. Have this ever been a problem for you?

B -No. We always do things in a certain way. If we see the function getInstance() we know it's a Singleton.

K -But what happens if an external developer comes in and just creates the Singleton by calling it's constructor (new SingletonClass();)?

B -That could be a problem, yes.

K -But then you also have implementations of the Singleton that very much applies with its original purpose.

B -Well, this is probably because classes get copied from project to project.

K -Does the Design Patterns you use in some cases get integrated into your projects because of the frameworks you use rather then for their specific purpose?

B -Yes this is likely to be the case. For example it could be the case that we know how to use the Command pattern, but not why we should use it.

The Future

K -Do you think that an increased understanding of design patterns would increase the possibility to reuse code?

B -Yes, I think so. Then we would be able to build more reusable components.

B -The understanding of Patterns, lets you understand better where to use them. Using Architectural pattern can give you good structure, but in my opinion it can sometimes inhibit the construction of reusable components.

K -The architecture of three of the projects are very similar, which must mean that the architecture is very reusable?

B -Our architectural is very reusable and have been used in many projects.

K -But I get the impression the you think a very reusable architecture can inhibit the building of reusable components. What you take on this?

B -Yes, we have seen this in many cases in our projects.