NORTH-KARELIA UNIVERSITY OF APPLIED SCIENSES
Degree Programme in Information Technology

Leppänen, Väinö

SOFTWARE MAINTAINABILITY

Thesis
June 2011

THESIS
June 2011
Degree Programme in
Information Technology
Karjalankatu 3
FIN 80200 JOENSUU
FINLAND
Tel. 358-13-260 6800

Author

Väinö Leppänen

Software Maintainability

Commissioned by Blancco Oy Ltd

Abstract

The purpose of this thesis was to elevate the maintainability of a certain software product. The focus of analysis for the object-oriented software in question was its classes and the relationships among them.

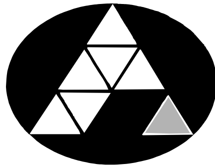
This thesis was operational by nature. The first phase was to find and familiarize myself with the pertinent literature, after which the software was analyzed. After each class was analyzed, it was evaluated and maintainability problems were fixed. The second phase of the project was to analyze each class individually and the third was to analyze the relationships between the classes. The project began in January and was finished in June 2011.

The primary objective was reached to a certain degree. The primary enhancements were related to the classes' clarity of abstraction and cohesion. The structure of the software's class hierarchy was also improved while the final goal of improving the relationships between the classes was left unimplemented. The solution was nonetheless designed. During the project it was noticed that object-oriented solutions might have been overly used.

Language	Pages 27
English	

Keywords

Software development, object-orientation, Python



OPINNÄYTETYÖ
Kesäkuu 2011
Tietotekniikan koulutusohjelma

Karjalankatukatu 3
80200 JOENSUU
p. (013) 260 6800

Tekijä
Väinö Leppänen

Software maintainability

Toimeksiantaja Blancco Oy Ltd

Tiivistelmä

Tämän opinnäytetyön tarkoitus oli kohottaa erään tietyn ohjelmiston ylläpidettävyyttä. Työ käsittelee kyseistä olio-pohjaista ohjelmistoa pääasiallisesti luokka-tasolla.

Kyseessä on toiminnallinen opinnäytetyö, joka alkoi kirjallisuuteen tutustumalla. Vuoden 2011 ensimmäisen puoliskon aikana ohjelmiston ylläpidettävyyttä kohotettiin käymällä sen rakennetta läpi samalla analysoiden sen kelvollisuutta. Rakenne käytiin läpi aloittaen luokkatasolta ja päätyen luokkien välisiin riippuvaisuuksiin. Ongelmalliset rakenteet korjattiin työn edetessä.

Tavoite ylläpidettävyyden kohottamisesta täyttyi tietyissä määrin. Keskeisimmät parannukset koskivat luokkien eheyttä ja abstraktioiden selkeyttä kuten myös ohjelmiston hierarkisen rakenteen ominaisuuksia. Viimeinen tavoite ohjelmiston luokkien välisistä riippuvaisuuksista jäi toteutusta vaille, mutta suunnitelma sille valmistui. Ohjelmiston tarkastelun aikana nousi myös epäilys että olio-pohjaisuus ei välttämättä ollut optimaalisin keino tiettyjen ohjelmiston osion rakentamiseksi.

Kieli

englanti

Sivuja 27

Asiasanat
Ohjelmistokehitys, oliokeskeisyys, Python

Contents

1 Introduction 6

2 Background 6

2.1 Banshee 6

2.2 Blancco Management Console 2 7

2.3 Software engineering 7

2.3.1 Abstraction 7

2.3.2 Cohesion 8

2.3.3 Information hiding 9

2.3.4 Service 9

2.3.5 Coupling 9

2.3.6 Law of Demeter 10

2.3.7 Fan-in and fan-out 10

2.3.8 Empirical verification 10

2.4 Maintenance 11

2.5 Maintainability 12

2.5.1 Understandability 13

2.5.2 Modifiability 14

2.6 Python 14

2.7 Design patterns 15

2.7.1 Façade 15

3 First iteration 16

3.1 Analysis and evaluation 16

3.1.1 Client 16

3.1.2 Licenses 17

3.1.3 Menu 17

3.1.4 Report 17

3.1.5 Group 18

3.1.6 Role 18

3.1.7 User 19

3.1.8 McSel 19

3.2 Results 21

4 Second iteration 24

4.1 System overview 24

4.2 Fan-in and fan-out 25

4.3 Summary 27

5 Discussion 28

References 29

1 Introduction

Software called Banshee was developed during 2010 at Blancco Inc. In December it seemed that its maintainability could be improved. Accordingly, the primary objective of this thesis was to elevate Banshee's maintainability by examining it and comparing the design choices made to advices given in literature. The examination is thus restricted to the design level, leaving out both the code and architecture levels.

2 Background

2.1 Banshee

Banshee's purpose is to facilitate the testing of Blancco Management Console 2 (MC2) by using Selenium-tools. It has been developed since the summer 2010. The goal is to create "commands" with which the testing engineers can write test scripts/cases which can then be run automatically simulating the actions of a real life user. This way regression tests can to some extent be automated and also some other types of tests, such as stress tests, become feasible. While at this point the detailed final use cases for the testing engineers are still somewhat unclear, the purpose is to engineer the needed low-level functionality which will be needed regardless of the specific type of use, which includes but is not limited to: clicking links, filling HTML-forms and reading responses from dialogs.

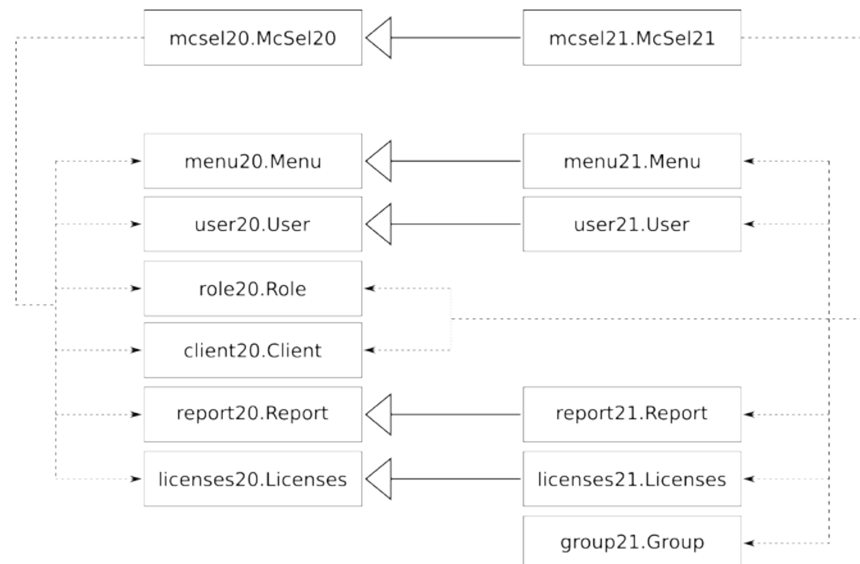


Figure 1. The classes, their dependencies and inheritance-relationships before the first iteration (UML).

Banshee consists of classes which can roughly be split into two hierarchical levels. At the first and higher level are the two classes *McSel20* and *McSel21*, of which the latter inherits the former. At the second and lower level are the classes which model MC2's user interface (UI), although a small part is modeled in the classes at the higher level (see next chapter for further information). The objects of the type *McSel20* or *McSel21* instantiate classes at the second level and while the classes at the lower level model MC2's functionality (e.g. user creation), the ones at the higher merely act as glue since the clients only use either of the two. In this thesis the clients refer to the test scripts that utilize Banshee.

2.2 Blancco Management Console 2

Management Console 2 is a software product created by Blancco Inc. Its purpose is to operate, manage and follow up on the erasure process (Blancco 2011).

Its UI is a web interface and it can be split into three primary parts:

1. Header. It is located at the top of the UI and it contains the software's name and version as well as language selection and busy-notification.
2. Menu. It is located on the left hand side of the UI and it is split into sections which contain links to pages. For example, 'Users' sections contains 'list users', 'create user' and a few other links to user related pages.
3. Work area. It displays the software's actual functionality when items on the menu are clicked. Each menu link leads to an initial page from which actions in the work area might lead to other pages and dialogs to which there are no menu links.

The three parts exist in all versions of MC2, but some changes have been made during the product's lifetime. For example, certain menu link texts have been changed and functionality has been moved from one page to another.

2.3 Software engineering

2.3.1 Abstraction

An abstraction can be defined in the following way: "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus

provide crisply defined conceptual boundaries, relative to the perspective of the viewer" (Booch 1991, 39). Abstractions serve an important role in helping people understand complexity (McConnell 2004, 89). Ideally the abstraction should represent something in the real world and in the problem domain (Booch 1991, 40; Coad & Yourdon 1991b, 53; McConnell 2004, 152). Obviously the classes should also be named using the problem domain semantics (Coad & Yourdon 1991a, 90).

There are various ways to implement abstractions, creating a class being the primary one in object-oriented programming. Some designchoices are considered mistakes as abstraction is concerned. For example, when a class really consists of two classes, the abstraction'sclarity and cohesion suffers and the class should be broken into two (McConnell 2004, 137). The class should preferably represent a real world concept and only one, in large part because it becomes easier to understand and thus, maintain. For the same reason it shouldn't contain anything that is not related to the represented concept. There is some correlation between clear abstraction and high cohesion as design with clear abstraction tends to have high coherence (McConnell 2004, 138).

2.3.2 Cohesion

Cohesion describes how well elements within a given module work together. Each module has certain functionality and a measure of cohesion is how well the elements within that module work together for the functionality, the module's "singleness of purpose" (Card & Glass 1990, 31).

High cohesion is achieved by having modules with limited functionality (Pressman 2001, 353) and all elements within the module striving for it. Cohesion is a desirable quality because it makes the modules good candidates for reuse and because they are easier to understand and thus, easier to maintain. The best kind of cohesion is considered to be functional cohesion where in all the parts of the class work toward certain behavior (Booch 1991, 124).

2.3.3 Information hiding

As a technique, information hiding is one of the few to have been proven valuable. David Parnas first introduced the idea to the public in 1972 in his landmark paper "On the criteria to be used in decomposing systems into modules". The basic idea is to create interfaces for modules and hide the details of implementation. Any changes within therefore cannot ripple past the interface since the clients are only dependent on the interface and class and its implementation can change freely. As little as possible should be revealed about the implementation (McConnell 2004, 92-93, 96.)

Encapsulation is there to make sure that information hiding is respected and in certain programming languages (e.g. Java, C++), it is not possible to access object's private attributes.

2.3.4 Service

Service is a public operation and in object-oriented programming typically a method in a class. The maximum amount of services per class tends to be limited to six or seven. The purpose of keeping the number of services low is to keep the classes simple. (Coad & Yourdon 1990a, 144-145.) I consider this to be more of a guideline than a rule and if the design is carried out correctly the outcome should lean towards this recommendation.

2.3.5 Coupling

When two modules communicate in any way, they are coupled. If they communicate only through simple interfaces using simple data arguments (primitives), the coupling is said to be "loose" or low (Pressman 2001, 354). If they directly access each other's implementation and the principle of information hiding is not respected, they are "tightly" coupled. Coupling is thus a relative measure which measures how dependent the objects are of each other. Tight coupling is not desirable for several reasons. Firstly, reuse becomes very difficult since the modules depend on each other and separation is arduous. Secondly, and sometimes more importantly, high coupling leads to ripple effects across the software system when changes are made in the implementation. A tightly coupled system is thus more difficult to understand, modify and maintain. Loose coupling is considered beneficial

and tight coupling is not (Booch 1991, 124; McConnell 2004, 100; Myers 1978, 42; Hunt & Thomas 2000, 140).

Circular dependencies are a form of coupling in which dependencies between modules are formed in a circle. A case in point is that there are three modules A, B and C. Module A depends on B, B depends on C and C depends on A. Circular dependencies should be avoided because they hinder testing and complicate the connections between modules (Fowler 2003, 49; McConnell 2004, 84-85, 95).

2.3.6 Law of Demeter

Law of Demeter for functions is a rule formulated in order to reduce coupling. It basically says that an object A can call routines in itself, routines in B if it instantiated it and also any methods belonging to objects passed to it as a method parameter but it should not call routines carried by objects contained within B. Following this rule will lead to building systems which tend to have fewer errors (Hunt & Thomas 2000, 140-141; McConnell 2004, 150.)

2.3.7 Fan-in and fan-out

Fan-in is a count of how many other modules use/call a certain module. It therefore describes how widely a certain module is used. Fan-out, on the other hand, describes to what extent a given module is dependent on other modules. Having high fan-in at the bottom of module hierarchy is considered good design as it indicates that the design utilizes general purpose modules to a high degree. Having high fan-out (more than approximately seven) predicts higher error rates and is therefore an undesirable design quality, which should be avoided (Card & Glass 1990, 36; McConnell 2004, 80-81.)

2.3.8 Empirical verification

There are some empirical studies made on certain design characteristics in object-oriented software systems. A study has confirmed that overall, following Coad & Yourdon’s design principles lead to a higher maintainability. They conclude:

> The result of our experiment show that the system designed according to Coad and Yourdon’s object-oriented design principles was significantly easier to maintain, i.e., complying with Coad and Yourdon’s object-oriented design principles has led to a design which is easier to understand and perform impact analysis upon. This result is further supported by qualitative evidence provided by the experimental subjects in the form of information from their debriefing questionnaires. This is consistent with a previous experiment where we tested the same hypothesis … This provides us with a high degree of confidence in the finding that the maintainability of object-oriented designs are sensitive to poor design practices, as defined by Coad & Yourdon. (Briand et al. 2001, 526.)

To name a few examples, Coad & Yourdon advocate loose coupling, high cohesion and discourage excessive attributes in a class (Coad & Yourdon 1990a, 129, 134, 144).

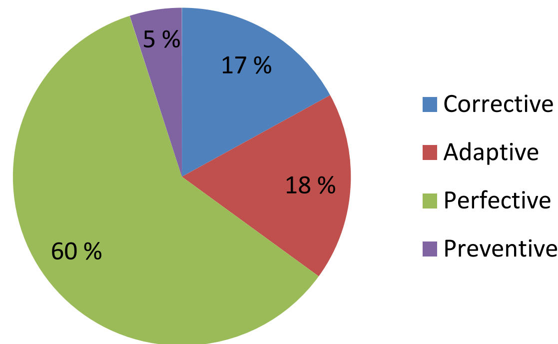


Figure 2. Distribution of effort during software maintenance, categorized by type.

2.4 Maintenance

As a part of the software life cycle, maintenance is the most expensive phase. It is estimated that maintenance covers about two thirds of the total cost of developing software (Haikala & Märijärvi 2004, 57). Maintenance can be divided into several types (Figure 2):

- Corrective maintenance: correcting the already existing features and components of the software system. It is standard practice to achieve this with bug reports and the following debugging leading to the corrections in the source code. This is estimated to cover approximately 17% of maintenance costs.

- Adaptive maintenance: making the necessary changes to enable the software to operate in a new environment, for example in a new operating system. It covers about 18% of the costs.
- Perfective maintenance: covers most of the costs with approximately 60% share. It is essentially about further developing the software by adding new features and also improving the software's non-functional features such as performance.
- Preventive maintenance: performed namely to avoid problems in the future, that is, to improve maintainability. (Keyes 2003, 247-248.) Recently this has also become known as refactoring (Glass 2003, 118).

Most of the maintenance is in fact enhancing the existing product during perfective maintenance, which bares similarity to developing software from scratch with one important distinction: the maintainers probably are not the people who originally developed the software. Thomas (1984 cited by McConnell 2004, 842) found that an average program is rewritten only after being maintained by ten generations of maintenance programmers so it is likely that the maintainers have to start their work with little or no understanding of the software. The maintainers therefore spend a lot of time trying to understand the intention of the original programmer, which is the most difficult task in software maintenance (Glass 2003, 121). Parikh and Zvegintzov (1983, cited by McConnell 2004, 842) estimated that maintenance programmers spend approximately 50 – 60% of their time trying to do just that while Fjelsted and Hamlen (1979, cited by Glass 2003, 121) estimated 30%. Furthermore, it should be noted that in order to execute any type of maintenance it is vital to understand the system being modified or perfected.

2.5 Maintainability

As discussed, maintenance is an expensive phase in which most the costs occur during perfective maintenance. Furthermore possibly the most time-consuming activity is trying to understand the intent of the original developers. As maintainability is about trying to make maintenance as easy and cheap as possible, the aforementioned two facts shall serve as a guide. First, the software design should be as easy to understand as possible and second, it should be as easy to modify as possible. It would appear, though, that understandability is a prerequisite for any type of maintenance and therefore it will take precedence over

modifiability, at least in the initial analysis. I expect that there might be conflict between the two quality attributes at some point as, for example, the Visitor design pattern aims for certain easy modifications but research exists which indicates that it is hard to understand (Gamma et al. 1995, 335; Vokáč et al. 2004, 149).

It should be noted that whatever kind of maintenance is being performed, the skill level of the maintainers is a major factor.

2.5.1 Understandability

In order to achieve high understandability, design must be modular. The understandability works on several levels; single class level, subsystem level and system level. Creating clear abstractions results in high understandability on class level and also aids on higher levels. Documentation can aid the understanding on all levels. In object-oriented programming the basic building block is either a class or an object (Booch 1991, 75). No clear rules on class design have emerged over the years in software engineering (Briand et al. 2001, 524) and comparing one design to another remains subjective (Glass 2004, 79). Despite this, it appears that a class with clear abstraction, high coherence and low coupling is considered desirable and research also supports the notion to some extent (Briand et al. 2001, 523). There are also guidelines to the class' size; no more than five to nine data attributes and less if they are not primitives (McConnell 2004, 144).

One of the benefits of using abstractions is that it results in self-documenting software, but despite best efforts to create self-documenting design in which classes/modules are easy to understand, it sometimes cannot be achieved and certain modules end up being too complex to understand in a reasonable amount of time even on their own. The purpose and functionality of the class can then be further clarified with documentation. It can serve as a guide to ease the understanding of the system and thus reduce the costs of maintenance (Keyes 2003, 249.) There are experts who consider maintenance documentation worthless since most often it is not being kept up to date (Glass 2003, 123.)

2.5.2 Modifiability

The relative ease with which changes can be introduced to software is called modifiability. Some of the positive characteristics mentioned in the previous chapter on understandability also have an effect on modifiability as understanding the software's structure and behavior is essential when making changes. Class' size is a factor; the smaller the size, the easier it is to understand the effects that changes to be made will have.

As a technique, information hiding is one of the few to have been proven valuable (McConnell 2004, 96). Encapsulation in turn is the enforcer for the information hiding guaranteeing that modules hide their secrets behind their interfaces and any changes within will remain local to the extent that the interface remains the same. Encapsulation is thus the foundation for modifiability, after which the class' coupling is examined. As coupling describes the dependency between modules, the looser it is, the easier it is to make changes since other modules are unaffected. With both loose coupling and small class size the maintenance programmer's task is significantly easier.

As a special category of coupling there is the relationship between superclass and its subclasses. This coupling is tight and inherent in an inheritance relationship. This, however, is desirable (Coad & Yourdon 1991a, 132), but deep inheritance trees are considered problematic since in order to make changes the programmer should understand and take into consideration all the classes in the inheritance tree, both the ones from which it inherits and the ones that inherit from it, that is, its superclasses and subclasses, respectively. Empirical research also exists and it has confirmed that deep inheritance trees increase the fault rate of the classes in it (Basili et al. 1996, 757). Those findings speak of the rising complexity and the following difficulties in understanding the classes when making changes, which will lower modifiability. McConnell suggests limiting the depth of inheritance tree to two or three levels (McConnell 2004, 147).

2.6 Python

Python is a programming language developed by Guido von Rossum. It is an interpreted and high-level programming language with full support for object-orientation.

Python only has public attributes in the sense that they can always be accessed from outside the object. There is, however, a convention followed by most Python programmers that attributes which are named to start with an underscore are considered part of the implementation and not the interface. Such named attributes could be considered protected or private, though not in the sense traditionally understood in programming. As a result, Python offers no support for encapsulation i.e. information hiding cannot be enforced (Python, 2011.)

2.7 Design patterns

In 1995 Gamma et al. authored the book "Design patterns – elements of reusable object-oriented software". In it they describe 24 design patterns, each of which is a solution to a specific design problem or issue. The solutions aren't the naïve ones to which most programmers first end up with, but results of a long line of gestation, trial and error. To the problems described the solutions are elegant, flexible and reusable (Gamma et al. 1995,1.) The design patterns also give a certain solution a name, which has distinct advantages as communication is concerned. This enables professionals to talk about widely used design solutions with names and thus communication becomes very effective (Gamma et al. 1995, 3; McConnell 2004, 103.)

2.7.1 Façade

In the Façade design pattern a unified interface is provided for the clients to use instead of using the more complex modules in the subsystem. As the parts of the subsystem evolve and become more complex, the clients might not have the need for the customization that the complex interfaces in the subsystem provide. Façade provides a simpler and easier to use interface through which clients can use the modules in the subsystem. While the pattern does not restrict the clients from using the more complex modules directly, the ones that decide to use the façade are effectively decoupled from the functionality implementing modules in the subsystem. This way the system becomes layered and the subsystem becomes more reusable (Gamma et al. 1995, 185-186.) The decoupling naturally results in more independence and thus, elevated maintainability.

3 First iteration

The first analysis concentrates on the individual classes; their abstraction, coherence and possible inheritance. In the thesis plan encapsulation in particular was mentioned but as Python does not support enforcing it in any way, it is out of the scope of this paper. Instead of enforcement, the extent to which information hiding is respected in the code could be analyzed but it is outside the scope of this thesis as well.

3.1 Analysis and evaluation

During this chapter the classes are individually analyzed and evaluated in order to plan the necessary changes. The results of the changes to the design are reviewed in the last chapter.

3.1.1 Client

The *Client* class represents the ‘Client management’ page in MC2 and the dialogs it contains. As the class represents all the elements on the page and dialogs, it carries several abstractions:

- The page itself, the text on it.
- The table of clients on the page.
- The dialog used to configure the client detector.
- The dialog used to manually manage the clients.

Most of the class’ operations and attributes are related to the aforementioned two dialogs (approximately half of the data attributes and operations).

The class’ understandability and coherence could be improved to a large degree by separating the two dialogs into their own classes. Currently there is not too much functionality implemented on the table of clients, but it, too, should be abstracted if the functionality is implemented. The original class is too complex for its high attribute count (McConnell 2004, 143) and it also carries too many public operations (Coad & Yourdon 1991a, 145). After splitting up the dialogs into separate classes the situation should improve.

3.1.2 Licenses

There are two *Licenses* classes in separate modules: *licenses20.Licenses* and *licenses21.Licenses*. The latter inherits the former. MC2’s main menu contains links to pages License container and license management. MC2’s license container related functionality was implemented in the *licenses20* module and inherited *licenses21* module made no changes, while implementing the license management functionality. License container page in MC2 consists of only one table and license management contains two dialogs in addition to a table of licenses.

The two pages should not be abstracted in the same class or in the same class inheritance tree. They are related only by the fact that both of the pages deal with licenses. They should be split apart to classes *LicenseContainer* and *LicenseManagement*. From the latter the two dialogs should be extracted into abstractions of their own as the dialogs are not simple but form somewhat complex problem domain entities of their own.

3.1.3 Menu

The Menu classes (*menu20.Menu* and *menu21.Menu*) represent the left-hand side menu in MC2’s UI. They are both very simple classes merely containing the MC2’s UI’s menu links. There is no serious reason to change these classes for complexity reasons, but their inheritance relationship should be removed.

3.1.4 Report

The Report classes (*report20.Report* and *report21.Report*) cover several pages and other elements in the MC2’s UI. The inherited class *report21.Report* merely changes one data attribute so from here on; the two classes are discussed as if they were one. Report class covers the three pages behind main menu links ‘Reports’ and ‘Report import’ in addition to several menus, dialogs and a report table.

The class contains multiple abstractions, which should be separated to:

- Reports page.
- Each of the pages behind the ‘report import’ menu link.
- Report table/browser on the reports page.

There are a few elements which could be abstracted but are so simple that it might not be necessary. The decision whether to abstract those simple elements will be decided during refactoring.

The inheritance between the original classes is considered useless and the two classes should be merged.

3.1.5 Group

The *group* class covers a single main menu link ‘List groups’ and the two pages contained. One of the pages is for listing groups and the other is for creating and editing them. The problem domain objects covered are the two pages, a group listing (table like), a form to create/edit groups and the relevant dialogs.

The domain objects worth abstracting are basically the two web pages. The group listing and the form are not good candidates since they are the entire content of their respective pages. Abstracting them would make the page objects themselves rather hollow and as such, prime candidates for removal (McConnell 2004, 575). The decision to separate the two contained pages gives rise to another design problem. The one page object needs to be aware of the other in certain operations for reasons related to the website’s AJAX mechanisms which are not delved into here. The solution is using the façade design pattern. This means creating a class merely to control the two pages and to contain the higher level commands while the pages themselves would contain only low-level commands. This would result in very low coupling as the two objects are unaware of each other and the façade knows them both and directs them. The resulting low coupling elevates maintainability (Hunt & Thomas 2000, 140).

3.1.6 Role

As the name suggests, *Role* class covers the functionality behind MC2’s main menu links ‘List roles’ and ‘Create role’. The two links collectively contain two pages, one for listing existing roles and another one for creation. The listing page also contains UI elements to

edit and remove roles. The roles are edited essentially on the same page as they are created only with prefilled information.

This class is not high priority when elevating maintainability due to small size and limited responsibilities. There could be reason to abstract the two pages and possibly some of the elements contained if there was knowledge of change, which there is not. This class will be considered low priority for now; it will be refactored if further reasons to do so emerge.

3.1.7 User

This class covers the main menu links ‘Create user’ and ‘List users’. MC2’s functionality to create, edit and list users are contained in the pages behind the links. It abstracts the two pages and almost all of the user related functionality implemented in MC2.

The evaluation leads to the same conclusion as in almost all of the previously dealt with classes: the pages should be abstracted and possibly the contained major elements such as forms and tables as well. This class exhibits an identical design decision as *Group* class does and thus this one also results in the same design decisions. The inheritance between *user20.User* and *user21.User* appears to be something that should be removed by absorbing the minor differences into the code (McConnell 2004, 575).

3.1.8 McSel

The class *McSel21* is inherited from *McSel20* so while discussing their role they will be collectively referred to as *McSel*. Several roles can be identified:

1. The primary class which instantiates all the objects representing MC2’s implemented functionality.
2. Acts as a mediator between the classes at the lower level.
3. Provides a few general-purpose operations.
4. Represents the UI’s header section and offers related operations.

McSel passes a reference of itself to the constructors at the lower level in order to provide the created objects access to the other objects, the general purpose operations in it and the operations related to MC2’s header section.

Because the class contains several roles, it becomes overly complex and its size increases. By providing the classes at the lower level a reference of *McSel*, a circular dependency is created which results in strong coupling and hinders testability (McConnell 2004, 95), reusability, and maintainability due to the system becoming one step closer to a monolithic. In similar manner to the one used at the lower level classes, there are several abstractions which should be extracted from *McSel* to new classes:

- There dropdown menu related operations which should be extracted somewhere as they are in no way related to *McSel's* abstraction (McConnell 2004, 137).
- Header related operations can also be separated to its own class.
- There are certain other general purpose operations which should be extracted from *McSel* as well.

While the relationship and dependencies between classes were planned to be examined during the second analysis, some observations were made during the first. *McSel* instantiates Selenium and the entire system then use that instance. Currently Selenium has united with WebDriver (Selenium project 2010) and this creates a higher than normal probability for change in Selenium's future versions. In general and particularly in preparation for the possible changes, the selenium class should be abstracted (Hunt & Thomas 2000, 45). Some of the earlier mentioned general purpose operations can be included in the Selenium wrapper, as they are conceptually extensions to Selenium's functionality and thus the wrapper's abstraction still remains clear. The header related operations will be moved to a new class called *Header*.

The inheritance relationship between *McSel20* and *McSel21* seems to be useless since very little is changed in *McSel21*. There are some changes in data attributes and operations in addition to instantiating classes of "21" (classes ending in 21). All of those inheritance relationships are planned to be removed and thus the reasons for this inheritance decrease. As the *McSel* classes are refactored last, the decision to remove the inheritance will be delayed to refactoring.

3.2 Results

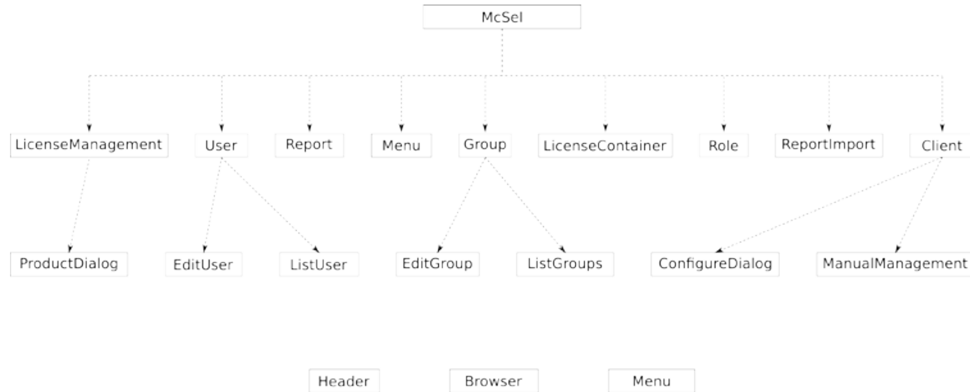


Figure 3. The classes and their dependencies after the first iteration.

The changes made according to the previous chapters are presented in this one. For the most part the process was executed without problems and for the most part it only consisted of refactoring the existing code. The result (Figure 3) does contain some new functionality, namely in the created facades *User* and *Group*.

The *Client* class was split up as planned. The much smaller and cleaner *Client* class became to represent the Client Management page in MC2 and the two major dialogs were created and the functionality was transferred accordingly. The *Client* class instantiates both of the created dialog classes, *ConfigureDialog* and *ManualManagement*.

As planned, the two classes concerning license related functionality were separated as there was no connection between their abstractions. The resulting classes were named *LicenseContainer* and *LicenseManagement* according to MC2's vocabulary and the links of the main menu. The two classes were related only by the fact that they are grouped in the same section in the main menu and thus represent a near perfect example of McConnell's advice about two classes in one. There was also a plan to abstract the two dialogs in license management, but during refactoring it was noticed that their implementation was so similar that one class, named *ProductDialog*, was enough.

The two *Menu* classes were merged into one in an operation which was almost as simple as cut and paste. The only reason for there being two classes seemed to be that the new methods were added in Management Console 2.1 and thus seemed to require an extending class which *McSel21* in turn would instantiate. It was only noticed later in the project that while this logically made sense at the time, it was only making the design more complicated without any practical benefits.

The *Report* class was split into two classes called *Report* and *ReportImport*, according to MC2's vocabulary. While there were plans to further separate the pages in *ReportImport* to new classes, this was not implemented as the code inside was considered to require redesign and possibly rewriting. Redesign was considered a lower priority and was delayed to be done at some later occasion. The inheritance relationship was removed according to plan.

The *Group* class was refactored as planned. The result was the creation of two new classes for the two contained pages: *EditGroup* and *ListGroups*. MC2 (from version 2.1 onward) contains the functionality to create, edit, list and delete groups, all of which were implemented in Banshee with the new classes. The new classes were positioned hierarchically under the class *Group*, which contains instances of the newly created classes. This class in fact is not necessary in order to implement the existing functionality but was added in order to first, contain the two objects and second, to later on contain higher-level functionality. An example of such could be the creation of arbitrary number of groups automatically or changing the attributes of a number of groups in a certain way. This functionality is more utilitarian in nature to save the testing engineers' time when they wish to test a larger setup, while it still was a part in elevating maintainability as the two contained classes attained low coupling.

All of the aforementioned steps were carried out for *User* class, too, as the problems were identical. The *User* classes were refactored to classes *User*, *EditUser* and *ListUsers* and the original inheritance was removed. As planned, *Role* class was considered low priority and it wasnot refactored.

McSel carried a lot of responsibilities. The general purpose operations were noticed to be expansions on Selenium's functionality. It was also noticed that Selenium itself was higher than likely candidate for replacement either by its next version (Selenium 2) or some other tool entirely. Thus the result was to create a wrapper for selenium and include the general purpose operations into it. By isolating the selenium class into a wrapper it is much easier to replace. The resulting new *Browser* class' coherence is high and *McSel's* coherence is also increased by removing the operations that were out of place.

The header related operations in McSel were also moved to a class of their own resulting in a clear abstraction and high coherence. The class is named *Header* and hierarchically it is at the bottom as indicated in Figure3. The class in fact becomes one with a high fan-in, which is considered desirable for classes at the bottom of the hierarchy (McConnell 2004, 80; Pressman 2001, 355-356).

After the aforementioned parts of *McSel* were moved, passing the reference of *McSel* was analyzed and it was noticed that mostly it was about having an access to single operations, most of which were in the *Menu* class. Passing the entire object when only one part of it is needed is not strictly bad practice (McConnell 2004, 179), but it does result in reduction of reusability (Plauger 1993, 148-149). In this case it was not strictly clear based on sources whether the entire object should be passed or only the reference to the needed services. Service references were chosen because the dependencies are that way clear and explicit in the definition and documentation of the classes dependent on *Menu*.

As planned, the decision to remove *McSel*-classes' inheritance was done when starting its refactoring. The decision was an easy one as the differences became very small. This was due to the fact the inheritance relationships elsewhere were removed and a substantial part of *McSel's* functionality was moved elsewhere as planned. Furthermore, for the clients who instantiate *McSel*, the situation was greatly simplified as there now is only one class.

4 Second iteration

In accordance to the plan, the second iteration of this process is focused on the relationships between the classes in the system. Primarily this is about the level of coupling in the existing system and the ways to decrease it. As it turned out, the first iteration in fact removed some of the coupling between the modules and it also removed some problematic relationships. The system after the first iteration is reviewed first in order to find the remaining problematic relationships.

4.1 System overview

A representation of the system after the first iteration can be seen in Figure 3. The goal was to elevate maintainability on the class level and the refactoring resulted in an increase in the total number of classes, which also have cleaner abstractions and higher coherence. The result was also that the class hierarchy was transformed from a “pancake” to a more distributed model with more levels vertically.

The current system is easier to understand when each class is considered but the system as a whole changed its style. The system’s style was essentially transformed from a central control to a more distributed style, which generally is preferable, though to some extent harder to understand for less experienced programmers (Arisholm & Sjøberg 2004, 533). I consider the decrease in understandability to be merely theoretical for several reasons. First, almost all of the classes’ names are directly from the problem domain and when looking at the system as a whole the roles of each of the classes should be self-evident to anyone familiar with the problem domain. Second, with a few exceptions the dependencies in the system flow from top to bottom and the total number of classes is small (20, see Figure 3).

Design patterns were used in the system to small extent. On the design level the design pattern Façade in particular was used on a few occasions. While the use of design patterns is considered beneficial to understandability, the reason for their use was strictly practical during the first iteration. I consider that the current use increases maintainability to some degree as it usually is considered to do so (Bass et al. 2003, 109-110; Gamma et al. 1995,

188-189), but cannot estimate the extent. This is in part due to the fact that the facades used do not restrict the use of the sub modules contained.

The first iteration was mainly about understandability but the results had an effect on modifiability as well. The distributed nature of the system in itself increases modifiability as the changes are more likely localized. Naturally clean abstractions guide the maintainers on where changes are needed when they become necessary. Furthermore, the more distributed style encourages information hiding to the extent that the developers use interfaces and avoid content coupling, which has been the case during this project.

4.2 Fan-in and fan-out

Having low to medium fan-out at the highest levels of module hierarchy and high fan-in at the lowest level is considered ideal. With this in mind, the situation has improved since starting this effort to increase maintainability. Before starting, there were only two levels and at runtime, the higher consisted of only one class which called all the other classes. This type of centrally controlled system has the worst features; high fan-out at the top and low fan-in at the bottom. When considering the current design, the levels have increased which in fact was needed in order enable low fan-out for the modules in middle of the hierarchy. During the process certain high fan-in modules were formed at the bottom of the hierarchy and currently the modules in the middle of the hierarchy are also in good condition; their fan-in is limited and more importantly, their fan-out is limited, always below three. There might be some work in increasing the highly utilized classes at the bottom but the only truly problematic part of the current design is fan-out of *McSel*. After the first iteration its fan-out in fact increased since many of the existing classes were refactored to several classes and some of them in a way that all of them ended up in *McSel's* containment. This was to be expected as the system is designed in this way; *McSel* is the one class that the clients instantiate and it acts as their main interface. But *McSel* in fact does not depend on the classes directly beneath it in any major way as it only instantiates them and provides the clients a way to access them. In reality it is the clients that depend on the interfaces of *McSel* and most of the system. This breaks the Law of Demeter and the choice to break it was made consciously as the clients needed to have the

low-level access in order to test certain features of MC2. In fact the only modules that are in some sense hidden from clients are the ones at the bottom of the hierarchy.

The amount of both fan-in and fan-out are acceptable within the system. The problematic dependency between clients and almost all the parts of the system is unresolved. While the problem was identified during the development, enough time was not spent on solving it. There are, to my knowledge, two simple ways to solve this issue and both will be considered.

On the other hand, the traditional approach which honors the principle of information hiding is to merely create more interfaces. An example scenario could be one in which there are three classes a, b and c. A contains b and b contains c. A needs access to c's routines but for b to publically offer c is a violation of the information hiding principle. The solution is to write interfaces in b which in turn access the routines in c and thus a and c remain decoupled while b acts as a mediator. This is a simple but arduous solution as in a hierarchy with several levels the amount of routine interfaces can increase dramatically the higher the class resides and in the case of *McSel*, there are three levels in the hierarchy beneath it but currently I believe only two of them should be publically offered to clients. While this solution is arduous, it is simple enough so that less experienced programmers can successfully implement it. Taking into consideration both my experience and the upcoming maintainer's experience, this solution seems the right one. This in spite of the fact that after it is completed, *McSel* will have a somewhat high fan-out (nine).

The second way to solve the dependency problem is to create a mini-language for the clients to use. This effectively decouples the system and clients to the highest degree possible. This was in fact considered at one time during the development effort but it was not considered a viable solution by the management. The effort needed to design and implement the mini-language depends to a high degree on the experience of the programmers. I personally do not have any experience in designing or implementing mini-languages but fortunately there are utilities to aid in the effort, one of them being pyparsing library. As this option already has been rejected, it will not be delved into any further. I do

consider that if the decoupling of the clients and the system becomes important sometime in the future, this option should be reconsidered.

4.3 Summary

The second iteration brought no changes to the software. A decision was made to create the necessary interfaces to remove the coupling between clients and most modules in the system, but its implementation will be executed by the programmer taking over my duties in the near future as there simply is not time left for me to do it myself.

5 Discussion

The objective of this thesis was to elevate maintainability. I consider the effort mostly successful. This software project will be continued by someone else who does not have a familiarity with its structure. If I have elevated the system's understandability even moderately, I consider it worth the effort. In addition to aiding the maintainers I have learned a lot about object-oriented software design while writing about this very interesting topic.

The basic method used was to examine the software system thoroughly while reading the advices of the expert authors and attempt to apply them in practice. The method was a success and the most import advices were related to the importance of clear abstraction, high cohesion and low coupling. The principle of information hiding lies at the heart of object-oriented programming and it deserves recognition here as well.

This thesis looked into some of the basic design decisions used in this particular software, but there is one aspect in particular which I consider could become problematic sometime in the future. The chosen architecture was object-oriented and as the focus in this thesis was design level issues, architecture was not examined. It was nevertheless noticed that some of the classes are only instantiated once during an execution. In addition, the instantiated objects are sometimes passed around. This could indicate that more procedural style structure is more appropriate for this software.

References

Arisholm, E. & Sjøberg, D. 2004. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. IEEE transactions on software engineering, August. 521-534.

Blancco. 2011. Data Destruction Reporting and Auditing - Blancco Management Console. http://www.blancco.com/en/product-services/blancco-management-console. 29.5.2011.

Booch, G. 1991. Object oriented design with applications. Redwood city, California, United States of America: The Benjamin/Cummings Publishing Company, Inc.

Briand, L.C., Bunse, C. & Daly, J.W. 2001. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. IEEE transactions on software engineering, June. 513-530.

Card, D. N. & Glass, R. L. 1990. Measuring software design quality. Englewood Cliffs, New Jersey, United States of America: Prentice-Hall.

Coad, P. & Yourdon, E. 1991a. Object-oriented design. London, United Kingdom: Prentice Hall.

Coad, P. & Yourdon, E. 1991b. Object-oriented analysis. London, United Kingdom: Prentice-Hall.

Fowler, M. 2003. UML distilled, third edition. Boston, Massachusetts, United States of America: Addison-Wesley.

Glass, R. L. 2003. Facts and fallacies of software engineering. Boston, Massachusetts, United States of America: Addison-Wesley.

Haikala, I. & Märijärvi, J. 2004. Ohjelmistotuotanto. Helsinki: Talentum.

Hunt, A. & Thomas, D. 2000. The pragmatic programmer. Boston, Massachusetts, United States of America: Addison-Wesley.

Keyes, J. 2003. Software engineeging handbook. Auerbach.

McConnell, S. 2004. Code complete, 2nd edition. Redmond, Washington, United States of America: Microsoft Press.

Myers, G. J. 1978. Composite/structured design. New York, New York, United States of America: Van Nostrand Reinhold Company.

Plauger, P.J. 1993. Programming on purpose. Upper Saddle River, New Jersey, United States of America: PTR Prentice Hall, Inc.

Pressman, R. 2001. Software engineering, 5th edition. New York, New York, United States of America: McGraw-Hill.

Python, 2011. Classes – Python v2.7.1 documentation. http://docs.python.org/tutorial/classes.html. 29.5.2011.

Selenium project. 2010. Selenium 2.0 and webdriver – Selenium documentation. http://seleniumhq.org/docs/03_webdriver.html. 25.4.2011.

Vokáč et al. 2004. A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns. Empirical software engineering, 9. 149.