Linnéuniversitetet
Kalmar Växjö

Report

JHotel

Design and Refactoring

Författare: [Författare]
Handledare: [Handledare]
Examinator: [Examinator]
Termin:
Ämne: [Ämne]
Nivå: [Nivå]
Kurskod: XX999

Index

Index **2**

Bilagor **2**

Introduction **3**
Methodology 3

Analysis **3**
Previous Analysis 4
Metric Category: Code Analysis 4
Metric Category: Cohesion/Coupling 4
Metric Category: Cycle 4
Metric Category: Size 5
Personal pre-analysis 5
Cycles 5
Solution 6
Coupling vs. cohesion 6
solution 6
Lines of Code and number of Statements 7
Solutions 7
Method signature 7
Solution 7
Architecture 7
Solution 7
Patterns 8
Solutions 8
Language 8
Solution 8

Final Analysis **10**
Introduction 10
Final System Metrics 10
Metric Category: Size 12

Conclusion **12**

Bilagor

Skriv in eventuella bilagor manuellt

Introduction

For this assignment we received an application to manage a hotel, but the code is legacy and we have to improve it. As a personal appreciation, this code looks more “early stage” than legacy, or perhaps both things at the same time.
By making a small refactor exploration, the code shows to be not documented at all. There is no package division and all the classes, frontend or backend, are mixed together in the main package. Not only this but there seems to be a second application with no clear meaning, called “Album”. The same lack of documentation makes the application confusing. The code had a big cycle that included 20 classes of 33 files, three of them are part of the “Album” application, so more than 60% of the classes are a cycle. The application follows all the typical design problems, the length of the code is too big, lack of comments and when there are, sometimes are in english, german and even spanish (unless `//nada` is a slang word in german and not a translation of “nothing” from spanish) what it looks like many different hands touched the code and left no documentation. Other typical problems of legacy code found were deep code, with many repetitions, lack of automation, absolute no use of abstraction.

This was my first occasion of working with legacy code and I went through all the phases found on “*Finding your starting point”*, with fear and frustration galore. My first reaction was trying to find some sense on code with none documentation and no easy naming conventions. After many mistakes I tried my best, in the time left, to improve the code, but at least I improved my skills and even if I still feel unexperienced in this area, the fear and frustration were less than at the beginning and I was able to see with my own eyes why it is so important to clean code and the importance of following software patterns. Surprisingly enough, even when the fear and frustration were slightly lower, the main feeling was “when to stop”. This assignment made me learn that sometimes you need a point to stop, code may never be perfect.

Methodology

In order to create this report, I used sonargraph to make an analysis before and after the refactor, with eventual check outs during the process to help me focus on the next target.

Analysis

The analysis was made with SonarGraph version 9.10.0.545. I made a run in SonarGraph with the code without any refactor or addition, below referenced as “Previous Analysis” and a final analysis when the time was out and I decided I should stop, below referenced as “Final analysis”.

Previous Analysis

Metric Category: Code Analysis

Name	Value
Component Dependencies to Remove (Components) (System)	23
Component Dependencies to Remove (Java Packages) (System)	0
Number of Ignored Threshold Violations	0
Number of Threshold Violations	23
Parser Dependencies to Remove (Components) (System)	109
Parser Dependencies to Remove (Java Packages) (System)	0
Structural Debt Index (Components) (System)	339
Structural Debt Index (Java Packages) (System)	0

Metric Category: Cohesion/Coupling

Name	Value
ACD (System)	17.97
CCD (System)	575
Highest ACD (System)	17.97
Maintainability Level (System)	78.66
NCCD (System)	4.28
Propagation Cost (System)	56.15
RACD (System)	56.15

Metric Category: Cycle

Name	Value
Biggest Component Cycle Group (System)	20
Biggest Java Package Cycle Group (System)	0
Cyclicity (Components) (System)	409
Cyclicity (Java Packages) (System)	0
Number of Component Cycle Groups (System)	2
Number of Cyclic Components (System)	23
Number of Cyclic Java Packages (System)	0
Number of Cyclic Modules (System)	0
Number of Ignored Cyclic Components (System)	0
Number of Ignored Cyclic Java Packages (System)	0
Number of Java Package Cycle Groups (System)	0
Relative Cyclicity (Components) (System)	63.20

Relative Cyclicity (Java Packages) (System)	0.00

Metric Category: Size

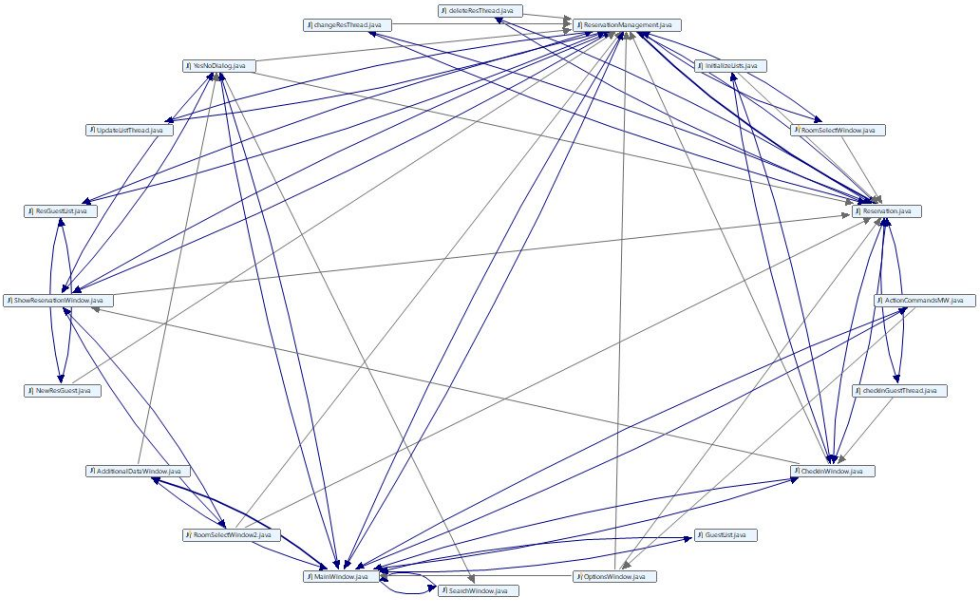
Name	Value
Byte Code Instructions (System)	69,980
Code Comment Lines (System)	1,615
Comment Lines (System)	2,032
Lines of Code (System)	15,785
Number of Components (System)	32
Number of Components (Ignoring Issues) (System)	0
Number of Java Packages (System)	0
Number of Modules (System)	1
Number of Statements (System)	11,787
Source Element Count (System)	14,172
Total Lines (System)	21,411

Personal pre-analysis

The code shows the next issues:

Cycles

The analysis shows that 63.20% of the classes (20) are part of a cycle.
This means that classes have dependencies that link them in a closed cycled. However that is not the worse part but the many dependencies that link two classes between them. In Sonargraph dependencies are show in a graph as gray arrows and 2-classes-cycle are shown in blue arrows,a s shown in the image below.



Solution

In this particular app, a big problem is the classes that work as Frame and are part of a frontend but that at the same time, include logic that could be easily implemented by a *controller class* in the back. In the particular case of the class `Reservation.java`, there is some logical code that, appart of being badly implemented with deep methods, it is only logical and is not extremely related to the class itself. However those methods (for instance methods that calculate dates) are used in many other classes just to make calculations and the rest of the class is unused. This kind of methods could be easily implemented in a *different class*, leaving `Reservations.java` for the real uses of the class.

Coupling vs. cohesion

The application suffers of high coupling and low cohesion, specially stamp and content coupling, where the complete structure is passed from one to another module, that is able to change content. For instance, the class `MainWindow.java` is passed to a high number of classes to be able to control its visibility, but it is not the only class. It is common in this application that classes are passed to different modules without any kind of hiding information.

solution

This is a big problem and the one that could make developers to rethink about recreating the application from scratch. There is no software pattern followed, so a solution and a must should be to create an architecture for the system, for instance MVC, and follow the patterns needed. For instance, the class `YesNoDialog.java` (that should be urgently renamed) has dependencies with all the classes that use a user confirmation, what makes no sense, something as common as user confirmation may be reused in the future. This class should be totally refactored following, for instance, a factory or decorator pattern and abstracted, so it could be used in many different classes, not only the ones with dependencies. It will make much easy to maintain and expand if necessary.

Lines of Code and number of Statements

Some classes are redundant on their methods, not automatised at all and many times hardcoded. This problem is tied to the number of Statements, that it is too high. Lines of code is 15,785, the biggest threshold violation of lines of code is 3,047 for `Hotel_Translator.java` and the biggest number of statements is 446 statements for a single method `OptionsWindows.saveRooms()`. It is understandable that java.swing creates many variables for each item, button, textfield, etc, and those have to be initialised and that is the main reason that most of the frames have a threshold violation. An special case of both threshold violation on lines of code and statements is the class `Rooms.java` with a cyclomatic complexity of 81, a number of statements of 340 and its own methods appear also in the Threshold, `Rooms.getRooms()` has a violation in number of statements with 340 statements, so is not the class with goes beyond the threshold but even its methods has that same problem. The way it creates strings for each floor and each room it is poorly automatised.

Solutions

In the case of the frontend frame classes, as said before, a refactor with focus on *separation of responsibilities* with a controller class could help to improve the lines of

code and statements. For the case of classes that are not frames, but models, I would recommend to add *new models* for specific situations. For instance, the class `Rooms.java` could be renamed and instead of creating a string for each floor of the hotel and then count the rooms, creating a nightmare of deep methods filled with loops and conditionals, we could make use of a real-world room java class with logic methods to return the string needed. The minimum solution to this problem would be refactor `Rooms.java` to automatize the condition-loop nested to a reusable method.

Method signature

During the project, it is easy to find difficult naming conventions, parameters with different order for each class, more than three parameters, and different ways to declare variables makes the code too ambiguous. For instance, parameter declarations with confusing naming and same type made it difficult to reuse or refactor (Methods with three strings as parameters, those three with confusing naming).

Solution

The objective should be to avoid any ambiguity. Making an initial exploration while refactoring, changing the naming convention and ordering variables and methods will help to understand the code and make it more readable. Methods with more than three parameters should be listed to be refactored to be considered clean.

Architecture

To be clear, there is no architecture. I explained before that logic and user interface is mixed in between the same classes, with no division by responsability or view. This may been seen by simple code, with no complexity between packages, because there are none. The real problem is that this type of code is not readable nor can be used easily by different developers. Two persons trying to work on specific areas, as frontend and backend, cannot do it individually because both types are entangled.

Solution

By creating different packages that explain the main objective of each class, it can help developers to understand the code and improve it. If a developer find a class which main objective is user interface, it can place it in a view package and extract all the logic that is not related to the frontend and place it in a different class. In this specific case it would be much easier to create packages to follow the model-view-controller (MVC) or 4+1 view model.

Patterns

The application do not use any software pattern so reusing is really low. Much of the duplicated code is a fault of a lack of automation of code.

Solutions

Many patterns could be used in this project that would help to improve the interactions between the classes. For instance, using the Observer pattern to update fields, Decorating or Factory method to create different types of rooms, users and user confirmation depending of the class that calls, encapsulating algorithms in a template method, etc. There are many useful patterns that would help and easy the job.

Language

This application was created in 2004 with Java Swing, that is still in use but is not the latest way to create java applications.

Solution

It would be nicer to re-create the application in JavaFX. Its layouts are nodes that may content groups of nodes, that you do not need to associate like in Swing. JavaFX events are consistent and more human friendly, its user interface controls can be styled with css and has more controls than swing.

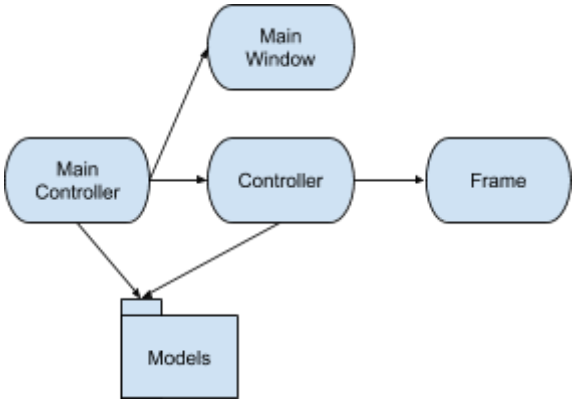
Re-engineering Plan

In order to start working, I created a list of objectives to consider the application as refactored.

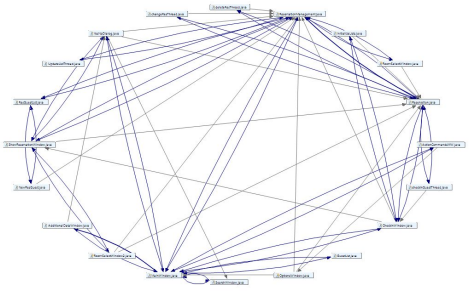
1. Exploration and refactor naming conventions.
    a. Test that changes of names does not break code.
    b. Use refactoring through IDE to change names.
2. Create packages and split classes on their main aim (MVC).
    a. Test that the IDE refactored the packages declarations.
3. Extract logic from View.
    a. Attack main cycles (Sonargraph blue cycles).
    b. Test that view still works properly.
    c. Test that view receive the logic from the backend.
4. Extract view from Controllers/Modells.
    a. Test that view still shows when called from controllers.
5. Abstract and reuse.
    a. Test that application still works as intended.
6. Implement patterns.
    a. Test that the application still works as intended.
    b. Test that the code can be scaled-up.

During the first days I explored the code while changing some naming conventions that made the code confusing. After that I decided to add git and started to refactor by using VCS. The first two days helped to understand the code but no many changes were done. After two branches of code trying to understand what happens with the classes I decided to attack the main problem, cycles.

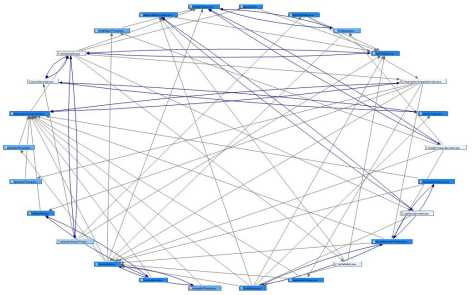
The objective was to extract the logic from View and try to remove as many endogamic cycles showed in Sonargraph as blue graphs. For this I thought about a way to improve the code:



In this way I would try to stick to MVC architecture. The problem was the handling of events in swing. The way the application sends the data to the frontend by event handling is not the ideal. Because of that I was able to remove many blue cycles from sonargraph but at the same time to create new ones from windows and their controllers, so the logic was able to flow from back to front.
During the days I worked so I was able to untangle many cycles but at the same time create new ones, mostly for I am inexperienced with swing and probably there is a better way to handle events and data flow. In the way I understood, I tied controllers to window frames so the information would flow from and to those couples. The objective was that controllers will flow data between them without the knowledge from the frontend and will only share data with their assigned frame. For instance, I created a main controller that would share data with other controllers, those controllers then will work with their correspondent frame. The problem was that there is at least 20 classes tied together. I was able to reduce the blue dependencies as show in the picture below, but there was still many hours of work to be able to create an horizontal code from this. The blue dependencies that are still on the picture come from controllers and their frames. Frames will trigger events, send data to their controllers, controllers will handle the logic and return the answer to the frame.



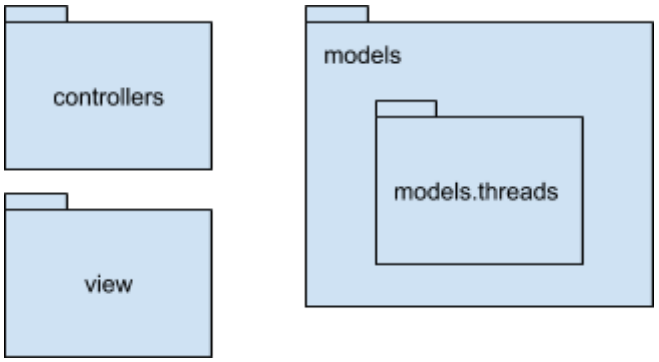
Before



After

The code still smell because in my opinion, I do not have the experience to apply techniques to improve the code so an extra pair of eyes of a partner would have been a great help. For instance, I created a branch to try to improve the user confirmation of the class YesNoDialog.java by using a factory pattern, but I spent too much time and I was unable to make it work properly.

The final packages were:



Final Analysis

Introduction

The number of classes increased by creating controllers and the main cycle of declarations was not fixed. However, some of the cycles were broken and the readability is cleared (even when not fixed) and naming convention was changed in the classes that were attacked, factory pattern was attempt to be implemented. If this project of refactorization was an actual project with a longer deadline I would rather decide to recreate from scratch the application in a newer platform as JavaFX, but anyways I was optimistic with the plan though the actual mess. In the words of Marie Kondo “It gets worse before getting better”.

Final System Metrics

Metric Category: Code Analysis

Name	Value
Component Dependencies to Remove (Components) (System)	22
Component Dependencies to Remove (Java Packages) (System)	20
Number of Ignored Threshold Violations	0
Number of Threshold Violations	22
Parser Dependencies to Remove (Components) (System)	137
Parser Dependencies to Remove (Java Packages) (System)	108
Structural Debt Index (Components) (System)	357
Structural Debt Index (Java Packages) (System)	308

Metric Category: Cohesion/Coupling

Name	Value
ACD (System)	23.43
CCD (System)	984
Highest ACD (System)	23.43
Maintainability Level (System)	66.01
NCCD (System)	5.14
Propagation Cost (System)	55.78
RACD (System)	55.78

Metric Category: Cycle

Name	Value
Biggest Component Cycle Group (System)	26
Biggest Java Package Cycle Group (System)	4
Cyclicity (Components) (System)	685
Cyclicity (Java Packages) (System)	16
Number of Component Cycle Groups (System)	2
Number of Cyclic Components (System)	29
Number of Cyclic Java Packages (System)	4
Number of Cyclic Modules (System)	0
Number of Ignored Cyclic Components (System)	0
Number of Ignored Cyclic Java Packages (System)	0
Number of Java Package Cycle Groups (System)	1
Relative Cyclicity (Components) (System)	62.32
Relative Cyclicity (Java Packages) (System)	100.00

Metric Category: Size

Name	Value
Byte Code Instructions (System)	69,025
Code Comment Lines (System)	1,941
Comment Lines (System)	1,941
Lines of Code (System)	16,040
Number of Components (System)	42
Number of Components (Ignoring Issues) (System)	0
Number of Java Packages (System)	4
Number of Modules (System)	1
Number of Statements (System)	11,651

Source Element Count (System)	14,157
Total Lines (System)	21,680

There is not many changes shown in the general report. There is still work to do with the dependencies and by creating new classes that are not fully working yet, it created more average component dependency, but it looks like some light is at the end of the tunnel because the propagation went down from 55 points from 56 and the relative cyclicity when 1% down to 62.32%. The lines of code increased probably for the new controllers created.
As positive change, I improved the automatisation in `Rooms.java` and changed from 340 statements to 103 by fixing a single method and removed the modified cyclomatic complexity. If I had time, I would add more automatisation and reduce even more the number of statements. The next step would be to create an object that represented a single room to abstract the statements and improve the scaling. That kind of automatisation should be done through the project in other classes, for instance OptionsWindows.java and `JHotel_Translator.java`.

Conclusion

During this assignment I encountered frustration and fear but at the same time it gave meaning to why to write clean code is so important. It would have been nice to know more about refactoring before now, I feel that I made big mistakes and wasted time by having tunnel vision and getting obsessed with some mistakes that were beyond my knowledge. However I feel more comfortable now with the tools and strategies to attack legacy code to the point that I wished for more time to keep working on this area, just for the pleasure of seeing the puzzle finished.
What I learnt was to make a attack plan by defining the architecture and always stick to them, go directly to the biggest problem, do not delay because of fear, abstract as much as possible as soon as possible, automatise all the things.