Refactoring Project

Design Refactoring Documentation  
Prepared by Team f262-01p2d:

* Gareth Daunton <gxd5116@rit.edu>
* Daniel Roberts <djr9478[@rit.](mailto:jdoa@rit.se)edu>
* Mitchell Derby <mxd5415@rit.edu>
* Quang Vu <qdv2130@rit.edu>

[Product Overview](#h.30j0zll)

[Analysis of Original Design](#h.1fob9te)

[Design Weaknesses and Strengths](#h.3znysh7)

[Use of design patterns.](#h.2et92p0)

[Subsystem and Class Structure](#h.tyjcwt)

[Sequence Diagrams](#h.3dy6vkm)

[Metric Analysis](#h.1t3h5sf)

[The Refactored Design](#h.4d34og8)

[Refactored Class Structure](#h.2s8eyo1)

[Sequence Diagrams](#h.dgjle05x8l9t)

[Implementation](#h.26in1rg)

[Metric Analysis](#h.lnxbz9)

[Reflection](#h.35nkun2)

# Product Overview

The problem facing our team was a slightly ill-conceived project with a number of design flaws and problems. Our job was to find problem areas in the code base and apply appropriate fixes to improve design quality and maintainability. A complete overview of the changes made will be listed below but a few of the fixes we applied were: class complexity, static access points and weighted methods.

# Analysis of Original Design

There was a number of problems we identifies within the original code base. Due to the large scale of the project and the limited time we had to complete the refactoring, we decided to chose a couple of key problems to implement a fix for. Based on simply looking at file names and how packages are split up, it was clear the project was intended to utilize an model view controller (MVC) architecture.

## Design Weaknesses and Strengths

The original design was very well commented and organized in a logical way with packages. The original design also followed the MVC architecture and incorporated several design patterns such as Observer and Strategy. One major weakness of the original design was the lack of a formal design document.

## Use of design patterns.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name: Observable Set** | | | **GoF pattern:Observer** |
| **Participants** | | | |
| **Class** | **Role in pattern** | **Participant's contribution in the context of the application** | |
| CalenderController | ConcreteObserver | Is notified when the calenderModel changes | |
| CalenderModel | Subject | Lets its observers know whenever it changes or an AppointmentTemplate or RefAppointment instansation is modified | |
|  |  |  | |
| **Deviations from the standard design pattern: N/A** | | | |

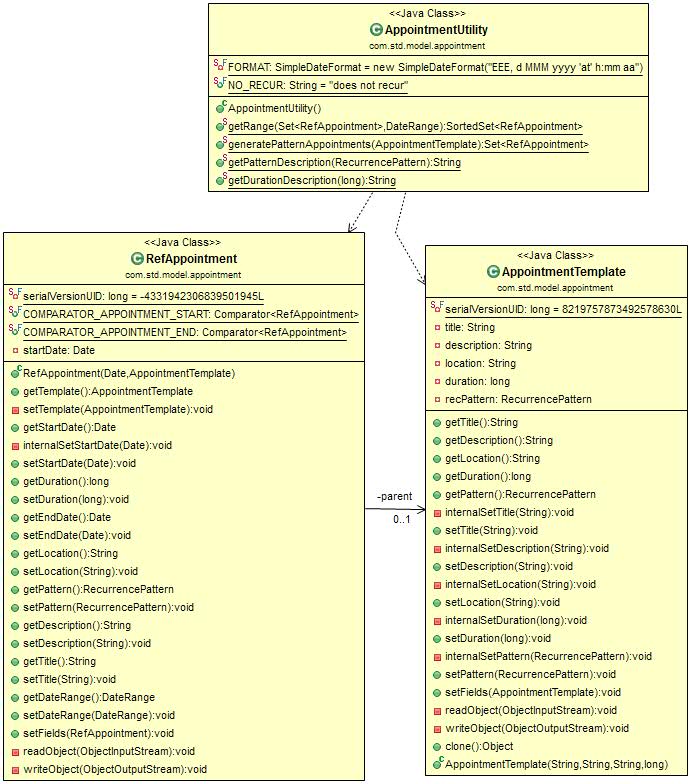
|  |  |  |  |
| --- | --- | --- | --- |
| **Name: Recurrence Pattern** | | | **GoF pattern: Strategy** |
| **Participants** | | | |
| **Class** | **Role in pattern** | **Participant's contribution in the context of the application** | |
| RecurrencePattern | Strategy | Parent class to the Concrete Strategies | |
| NDaysPattern | Concrete Strategy | Provides functionality specialized to a NDaysPattern | |
| DayOfWeekPattern | Concrete Strategy | Provides functionality specialized to a DayOfWeekPattern | |
| **Deviations from the standard design pattern:** None | | | |

## 

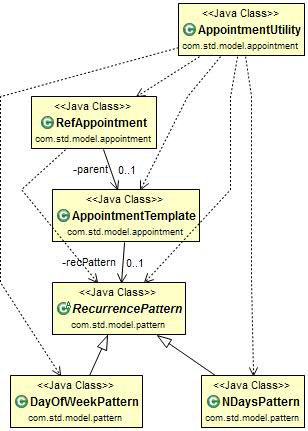
## Subsystem and Class Structure

Below are class diagrams of the areas refactored ONLY. Since this system is rather large with many packages and subsystems only the relevant and affected areas are displayed below.

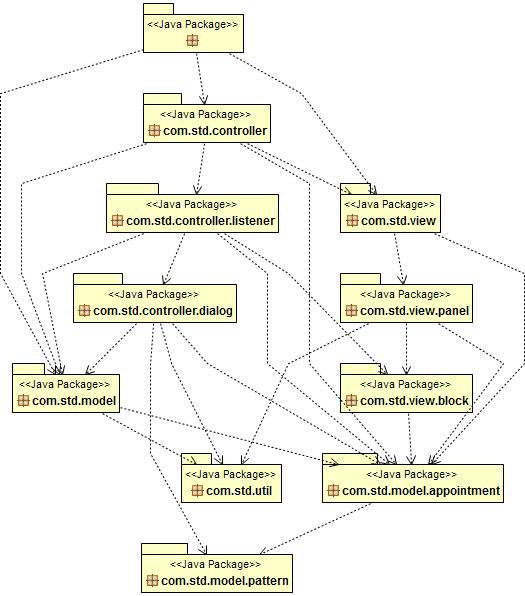
**AppointmentTemplate Before Refactor**



**Complexity Metric Before Refactor**

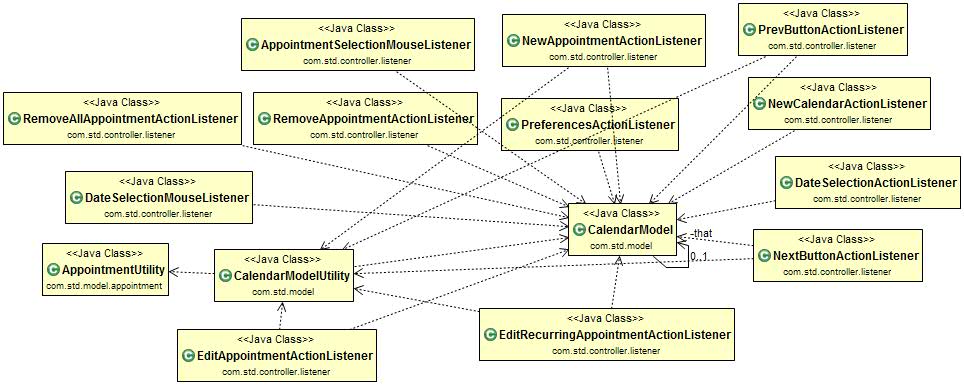


**Top Most Level Design**



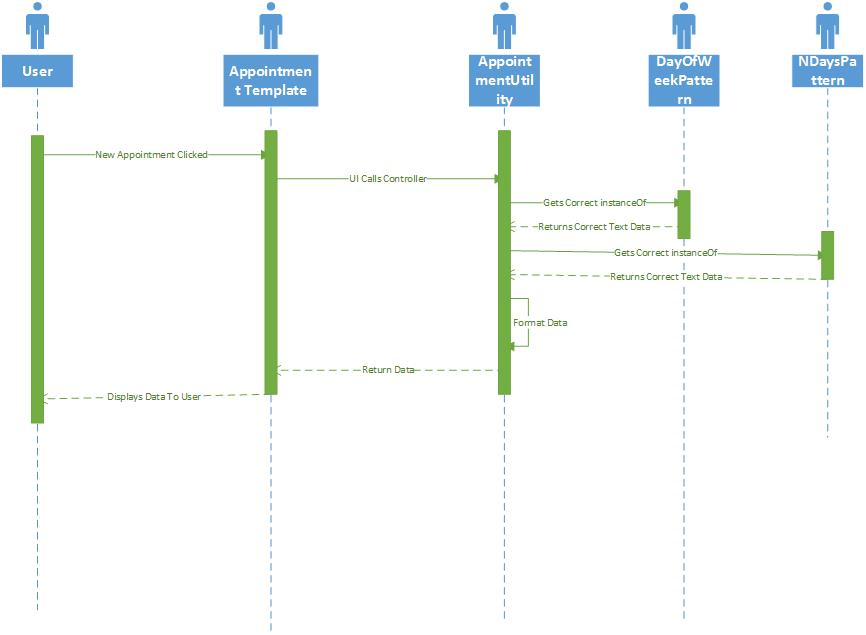
**Listener Package Before Refactor (Incomplete)**

*Note*: This diagram is not complete as the listeners are created in almost every class in the system.

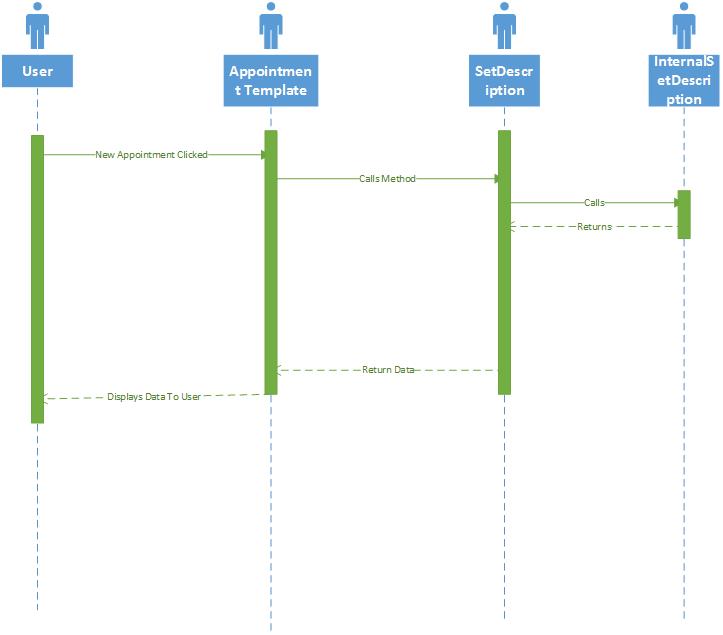


## Sequence Diagrams

Adding a recurring appointment



Setting Description for an appointment



## Metric Analysis

Our four primary concerns are below but all of them led us to look into one or two packages where most the problems were apparently occurring. In many MVC architectures the controller can become bloated and in this design we can see that very few classes in the controller handle the majority of tasks.

**Cyclomatic Complexity:**

Running initial metrics identified a single class to have double the value of complexity as the next highest value. This was later found to be caused by a large chain of “if” statements within “if” statements and the use of class instantiation within each statement.

**Static Methods:**

Metrics identified a single class with an over use of the static modifier. This can cause problems with classes knowing too much about each other and can increase class coupling.

**Weighted Methods:**

From analysis we identified that many methods within certain classes were duplicated, or had duplicated code within them.

**Afferent Coupling:**

Through metric analysis we determined that several packages/classes had too many other classes coupled to it and too many classes relied on a small number of classes which was violating the MVC architecture.

# The Refactored Design

Provide an overview narrative describing your refactored design. Describe how it addresses competing criteria such as low coupling, high cohesion, separation of concerns, information hiding, the Law of Demeter, extensibility, reusability, etc.

For each refactoring that you did, give it an identification and complete a copy of this table.

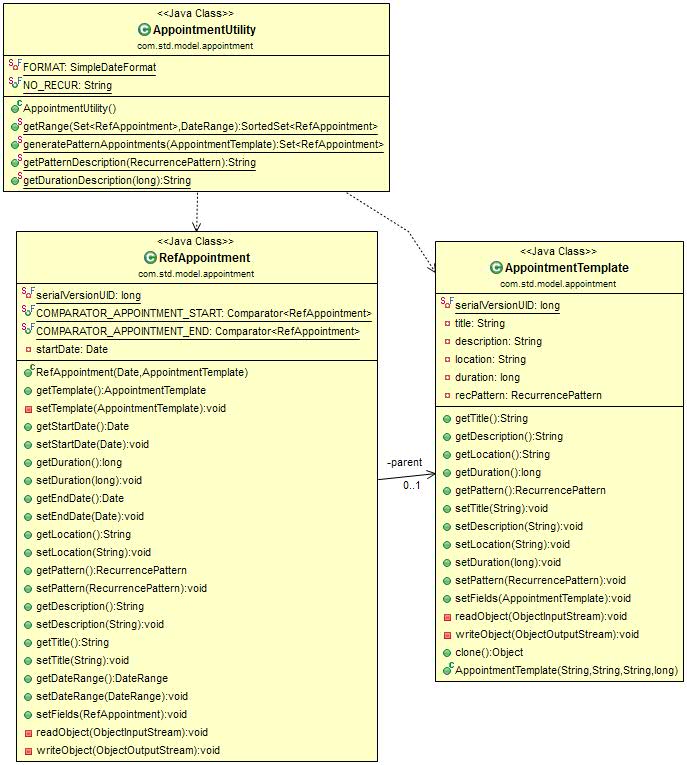
|  |  |
| --- | --- |
| **Refactoring identification** | Cyclomatic Complexity (VG) |
| **Metric evidence** | Value of 16, double the next value for complexity |
| **Other evidence** | Code Analysis |
| **Standard refactoring pattern (if any)** | Change conditionals to polymorphic types |
| **Description of the refactoring** | Reduce the complexity of the method by reducing the number of if statements and separating them into the appropriate class |
| **Classes involved** | 1. Method: getPatternDescription, Class: AppointmentUtility.java, Package: com.std.model.appointment 2. Class: DayOfWeekPattern.java, Package: com.std.model.pattern 3. Class: NDaysPattern.java, Package: com.std.model.pattern; 4. Class: RecurrencePattern.java, Package: com.std.model.pattern |

|  |  |
| --- | --- |
| **Refactoring identification** | Duplicated Code across Listener Package |
| **Metric evidence** | N/A |
| **Other evidence** | Code Analysis |
| **Standard refactoring pattern (if any)** | N/A |
| **Description of the refactoring** | Created a ControllerListener super class with a constructor that takes a CalendarController instance.  All other listener classes extend this super class and use the super’s calendar controller instead of instantiating a new calendar controller in each listener class. |
| **Classes involved** | Entire Listener Package |

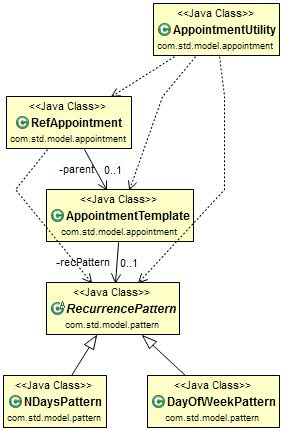
|  |  |
| --- | --- |
| **Refactoring identification** | Duplicate methods |
| **Metric evidence** | Weighted Methods per class (value of 29) |
| **Other evidence** | Code Analysis |
| **Standard refactoring pattern (if any)** | N/A |
| **Description of the refactoring** | Remove all private methods that are duplicated with the public methods  Instead of calling a private method in a public method, we simply perform its functionalities within the public method |
| **Classes involved** | AppointmentTemplate.java |

## Refactored Class Structure

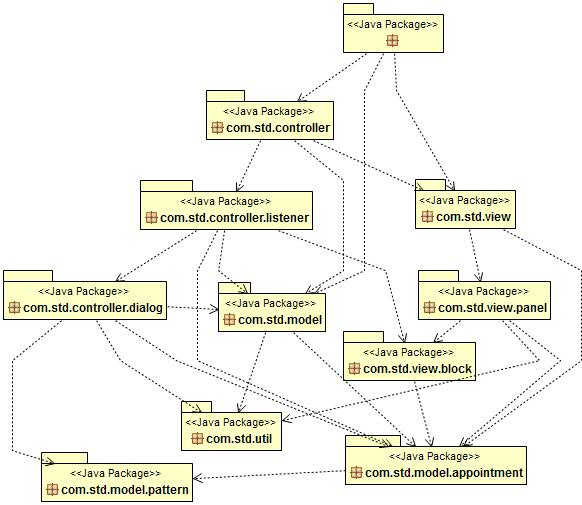
**AppointmentTemplate After Refactor**



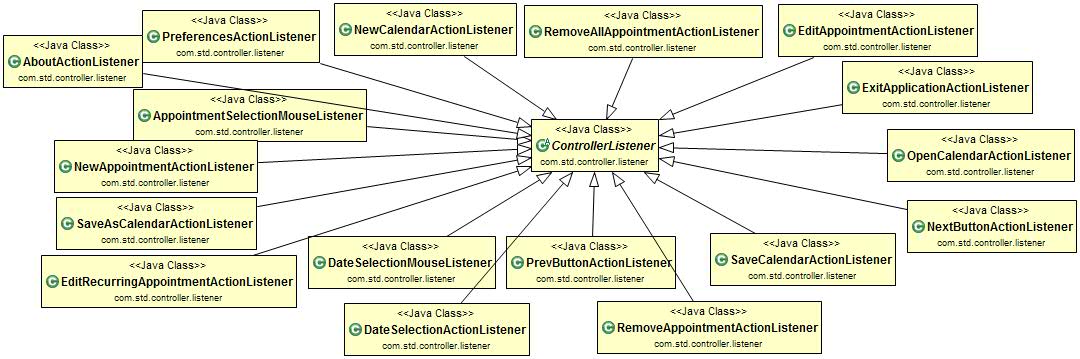
**Complexity Metric After Refactor**



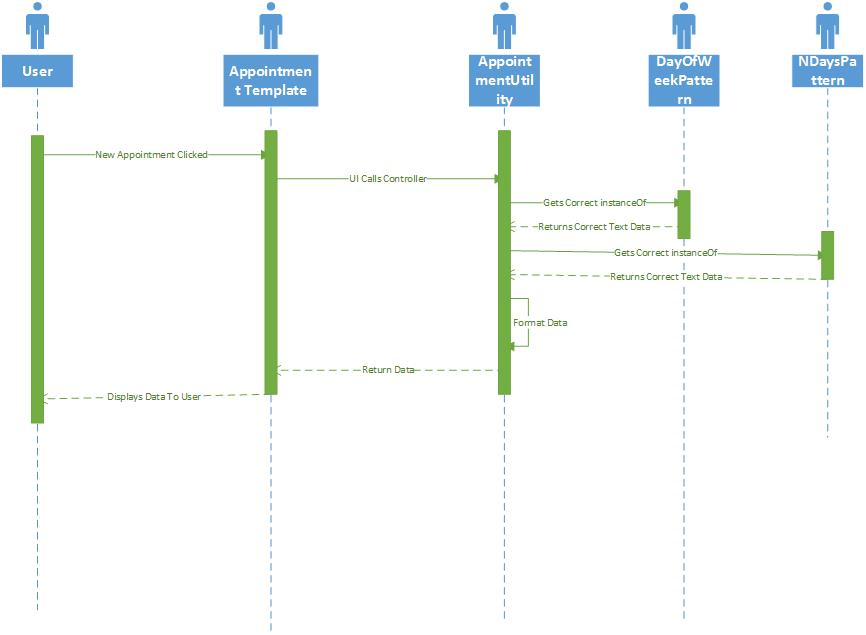
**Top Most Level Design After Refactor**



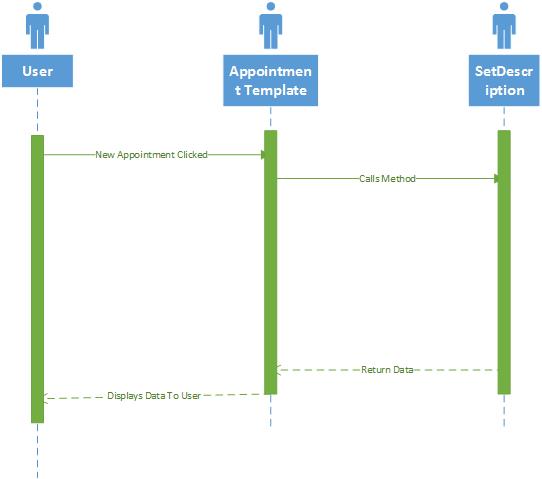
**Listener Package After Refactor**



## Sequence Diagrams

Adding a recurring appointment

Setting Description for an appointment



# Implementation

The sections of the project we decided to refactor was decided by the metric analysis we did as well as well as a quick read through of the code base. Based off of these two things we decided upon 3 sections of the code to refactor: The listener package, the ApppointmentTemplate class, and the getPatternDescription method in the AppointmentUtility class. For the Listener package duplicated code was removed by removing the creation of an instance of CalanderController in each listener class. This was accomplished using a parent class for the listener classes. For the AppointmentTemplate class, the original design had a public and private method for every action in the class, where the public method called the private method, which completed the action. This unnecessary code was removed and each action in the class has just a public method. For the AppointmentUtility class

## Metric Analysis

After the refactor was complete, the problem areas were completely resolved and the values were reduced significantly. Below is a complete overview of the new metrics for the areas refactored

**Cyclomatic Complexity:**

Value Before:16

Value After: 9

**Static Methods:**

Identified as problematic but not refactored.

**Weighted Methods:**

Value Before:29

Value After: 25

**Afferent Coupling:**

Identified as problematic but not refactored.

# Reflection

The most difficult part in any refactoring job is understanding the original code base and how all classes interact. If the original design deviates from the original design it also becomes extremely difficult to understand and make changes to because trying to identify certain pieces of the puzzle becomes next to impossible. Using a metric analysis and other software to help you in this process is a great feature that should almost always be utilized as it saves hours of trying to find hot spots as well as provides a more detailed look at what can/is causing problems with the current system. Once the hot spots have been identified through metrics and other analysis, the actual job of refactoring becomes a lot easier with only one or two small problems. Perhaps the most difficult aspect of the actual implementation stage is that the system must work in the exact same way as it did before the refactoring was even started. To achieve this testing at every stage is very helpful as well as not committing any changes to the master repository before there is assurance that it is actually functioning correctly.