<Fines Payment Manager>

Analysis and Design Document

Student: Plesa Gabriel

**Group: 30235**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| <13/04/2018> | <1.2> | <details> | <Plesa Gabriel> |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4

2.2 Package Design 4

2.3 Component and Deployment Diagrams 4

III. Elaboration – Iteration 1.2 4

1. Design Model 4

1.1 Dynamic Behavior 4

1.2 Class Design 4

2. Data Model 4

3. Unit Testing 4

IV. Elaboration – Iteration 2 4

1. Architectural Design Refinement 4

2. Design Model Refinement 4

V. Construction and Transition 5

1. System Testing 5

2. Future improvements 5

VI. Bibliography 5

# Project Specification

The project specification is to design and implement a client-server application for managing the payment of traffic fines. The application consists of two components which can be accessed using a username and a password. One component is used by the post office employees to register the persons that have paid a traffic fine. The other component is used by the police employees to do the following operations: (1) add/update/delete the drivers’ information (name, address, driving license details, identity card details), (2) add/update/delete fines, (3) create reports. When a post office employee registers a person that has paid a traffic fine, the application notifies the police employees from the corresponding police station. All the information about users, drivers and fines is stored in a database

# Elaboration – Iteration 1.1

# Domain Model

**User:**

*Attributes*: contains all the necessary data for a User, as shown in the diagram above

*Methods*: this class have the getters, setters and SQL methods for User data

**Fine:**

*Attributes*: contains all the necessary data for a Fine, as shown in the diagram above

*Methods*: this class have the getters, setters and SQL methods for Fine data

**Driver:**

*Attributes*: contains all the necessary data for a Driver, as shown in the diagram above

*Methods*: this class have the getters, setters and SQL methods for Driver data

**DrivingLicense:**

*Attributes*: contains all the necessary data for a Driving License, as shown in the diagram above

*Methods*: this class have the getters, setters and SQL methods for Driving License data

**UserActivity:**

*Attributes*: contains all the necessary data for driver data, as shown in the diagram above

*Methods*: this class have the getters, setters and SQL methods for User Activity data

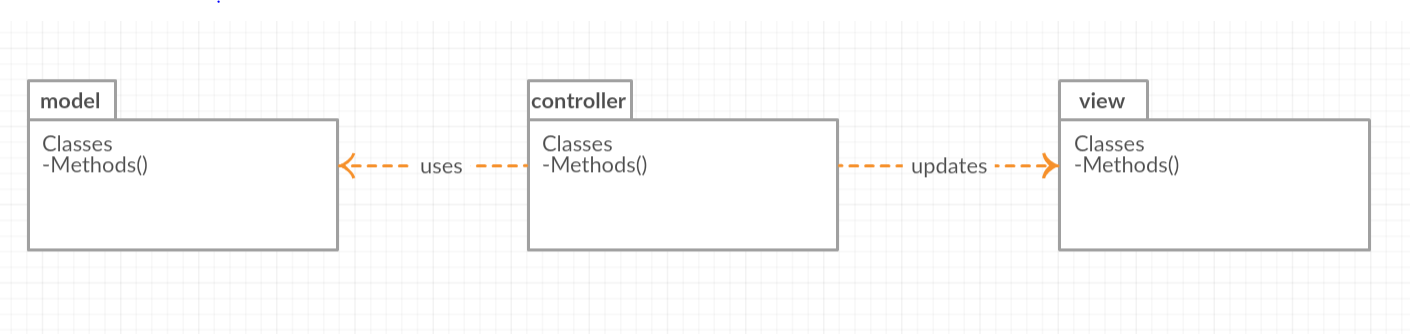
# Architectural Design

## Conceptual Architecture

For this project I chose to use:

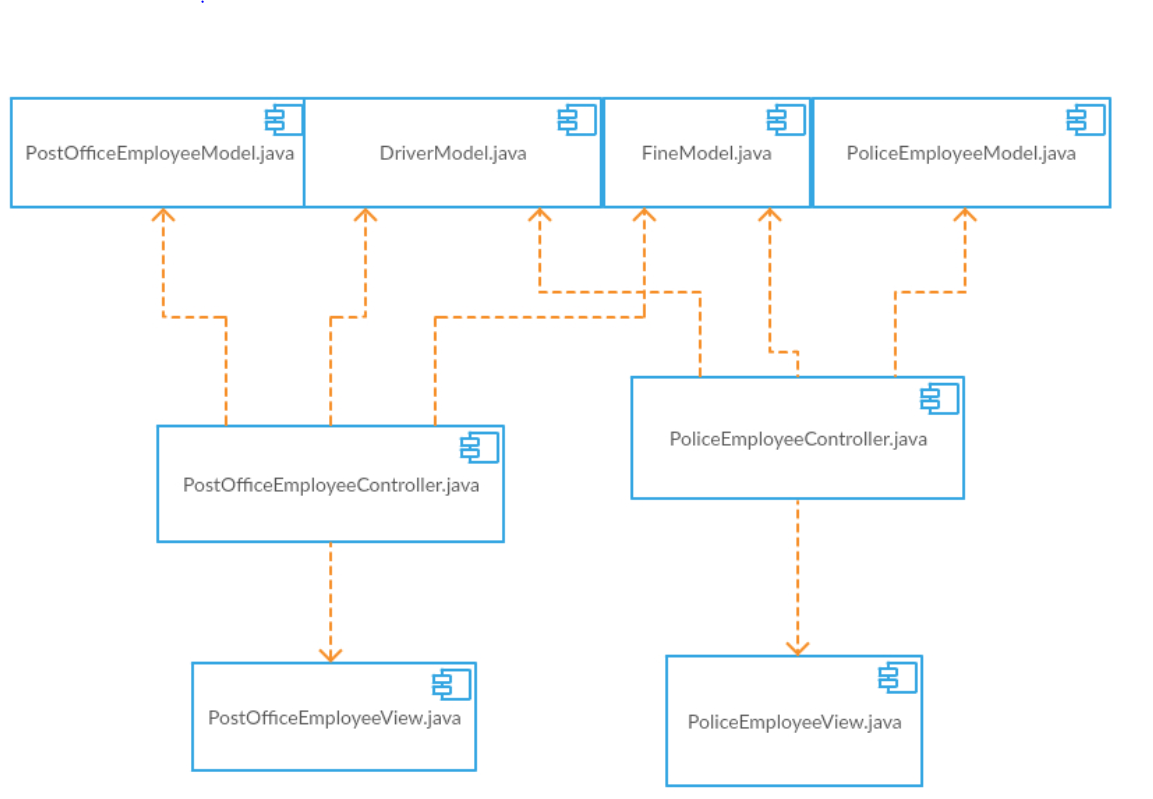
* Model – View – Controller Pattern
* Domain model in modeling the classes;
* Factory Pattern to generate the reports in .pdf and .csv format

## Package Design

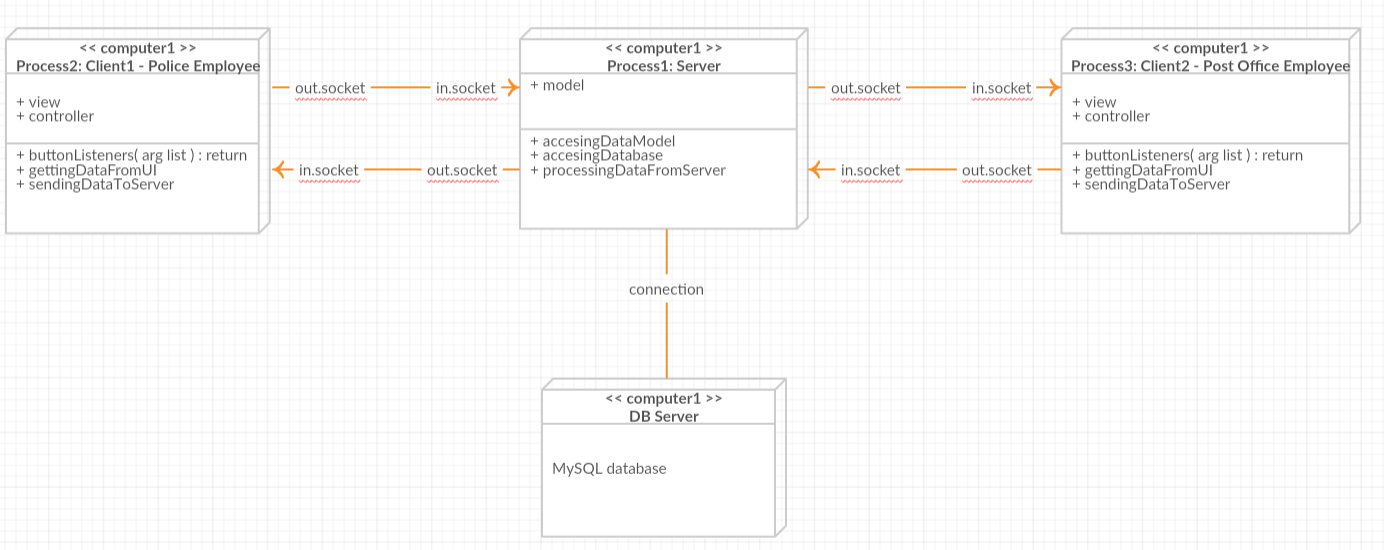


## Component and Deployment Diagrams

***Component Diagram:***



**Deployment Diagram:**

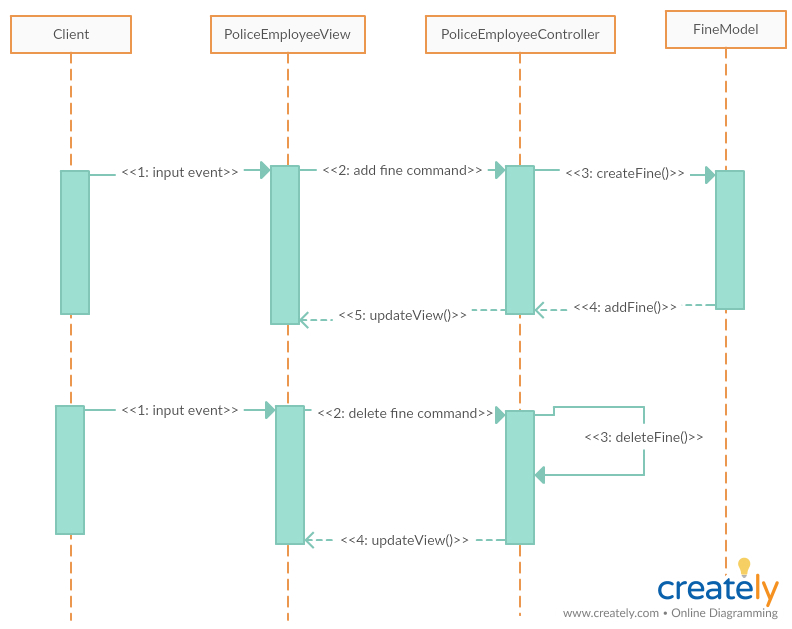


# Elaboration – Iteration 1.2

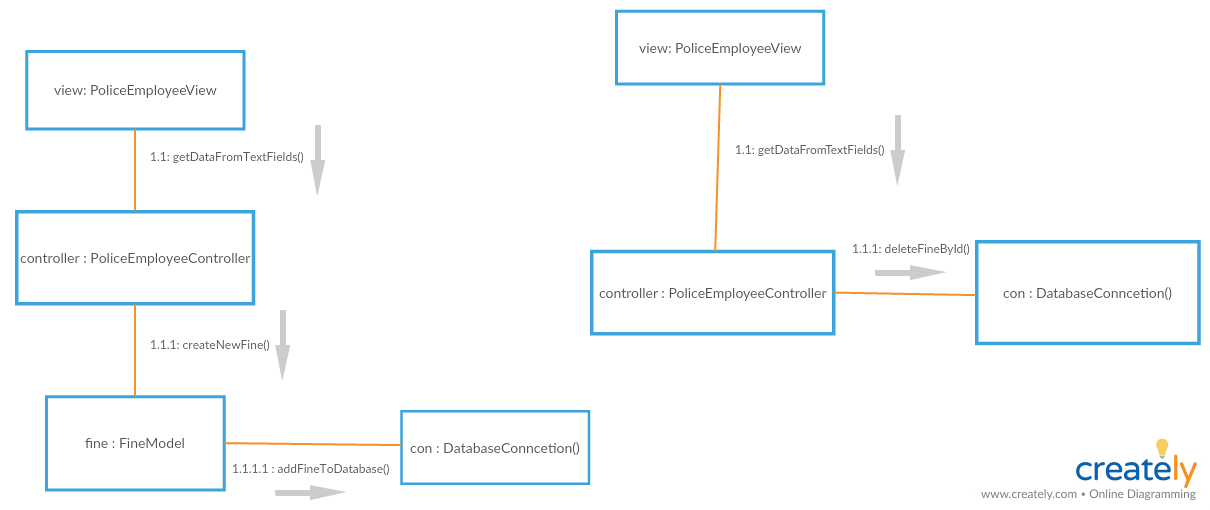
# Design Model

## Dynamic Behavior

**Sequence Diagram**



**Communication Diagram**



## Class Design

**PostOfficeEmployeeModel:**

*Attributes*:

* **idEmployee** – contains the post office employee id from database
* **name** – contains the post office employee name from database
* **age** - contains the post office employee age from database
* **sex** - contains the post office employee sex from database
* **address** - contains the post office employee address from database
* **paymentRegisterDate** – contains data of on the payment of a fine by a driver

*Methods*: this class have the getters, setters and SQL methods for Post Office Employee data

**FineModel:**

*Attributes*:

* **idFine** – contains the fine id from database
* **dateFineCommited** – contains data on which the fine was given
* **crime** – contains the fine crime information from database
* **price** - contains the fine price from database
* **paymentDeadlineDate** - contains the fine deadline date payment from database

*Methods*: this class have the getters and setters methods for Fine data

**DriverModel:**

*Attributes*:

* **idDriver** – contains the driver id from database
* **name** – contains the driver name from database
* **age** - contains the driver age from database
* **sex** - contains the driver sex from database
* **drivingLicenseInfo** – contains the driver driving license info from database
* **identityCardInfo** - contains the driver identity card info from database
* **finePayd** – contains the data if the driver paid or didn’t paid the fine
* **address** - contains the driver address from database

*Methods*: this class have the getters, setters and SQL methods for Driver data

**PoliceEmployeeModel:**

*Attributes*:

* **idEmployee** – contains the police employee id from database
* **name** – contains the police employee name from database
* **age** - contains the police employee age from database
* **sex** - contains the police employee sex from database
* **address** - contains the police employee address from database

*Methods*: this class have the getters, setters and SQL methods for Police Employee data

**PostOfficeEmployeeView:**

*Attributes*:

* **GUI compoenents** – contains the components needed to design the GUI with java swing.

*Methods*: this class have the methods for add listeners, get data from text fields, show an error message when needed and add/delete a row from a table.

**PoliceEmployeeView:**

*Attributes*:

* **GUI compoenents** – contains the components needed to design the GUI with java swing.

*Methods*: this class have the methods for add listeners, get data from text fields, show an error message when needed and add/delete a row from a table.

**PostOfficeEmployeeController:**

*Attributes*:

* **view** – is used to display the current information from database or to update when a user perform a CRUD operation
* **model** – contains the information from current post office employee who is logged in and uses the controller
* **fines** – an array list that contains data from all the fines
* **drivers** -– an array list that contains data from all drivers

*Methods*: this class have the methods needed to do the CRUD operations

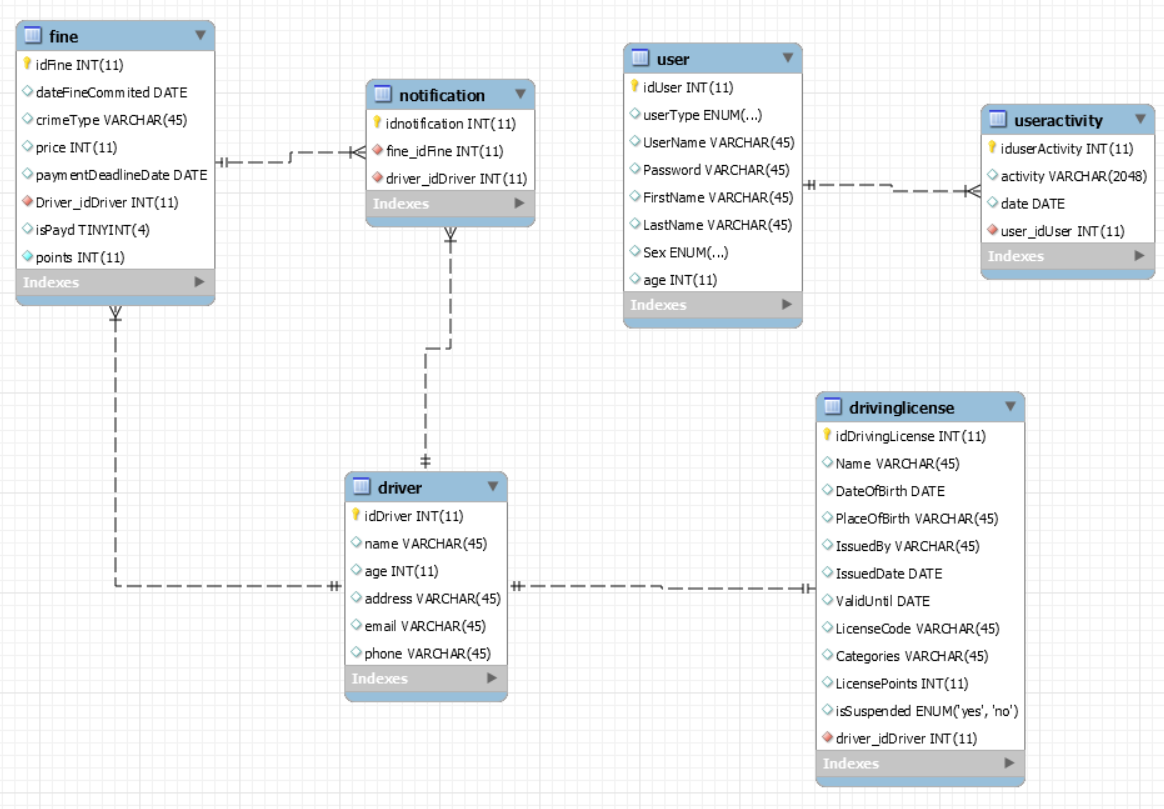
**PoliceEmployeeController:**

*Attributes*:

* **view** – is used to display the current information from database or to update when a user perform a CRUD operation
* **model** – contains the information from current police employee who is logged in and uses the controller
* **fines** – an array list that contains data from all the fines
* **drivers** -– an array list that contains data from all drivers

*Methods*: this class have the methods needed to do the CRUD operations

# Data Model



# Unit Testing

*[Present the used testing methods and the associated test case scenarios.]*

**-Tests will be performed after the implementation**

# Elaboration – Iteration 2

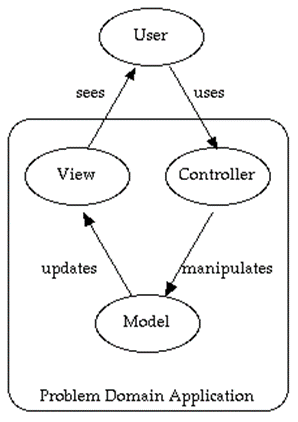
# Architectural Design Refinement

*[Refine the architectural design: conceptual architecture, package design (consider package design principles), component and deployment diagrams. Motivate the changes that have been made.]*

**Model–view–controller** (MVC) is an architectural pattern used in software engineering. Successful use of the pattern isolates business logic from user interface considerations, resulting in an application where it is easier to modify either the visual appearance of the application or the underlying business rules without affecting the other. In MVC, the *model* represents the information (the data) of the application; the *view* corresponds to elements of the user interface such as text, checkbox items, and so forth; and the *controller* manages the communication of data and the business rules used to manipulate the data to and from the model It is common to split an application into separate layers that run on different computers: the presentation/user interface (UI) (view), business logic (controller), and data access (model).

MVC is often seen in web applications, where the view is the actual [HTML](http://en.wikipedia.org/wiki/HTML) or [XHTML](http://en.wikipedia.org/wiki/XHTML) page, and the controller is the code that gathers dynamic data and generates the content within the HTML or XHTML. Finally, the model is represented by the actual content, which is often stored in a [database](http://en.wikipedia.org/wiki/Database) or in [XML](http://en.wikipedia.org/wiki/XML) nodes, and the business rules that transform that content based on user actions.

Though MVC comes in different flavors, control flow is generally as follows:

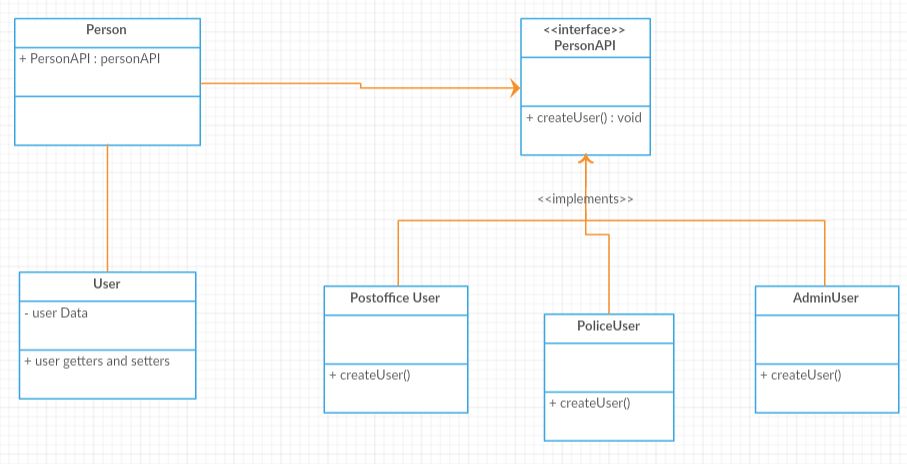
1. The user interacts with the user interface in some way (for example, click on a button)
2. The controller handles the input event from the user interface
3. The controller notifies the model of the user action, possibly resulting in a change in the model's state. (for example, the controller updates the user's [results](http://en.wikipedia.org/wiki/Shopping_cart_software) table).
4. A view uses the model indirectly to generate an appropriate user interface (for example, the view lists the result table's contents). The view gets its own data from the model. The model and controller have no direct knowledge of the view.
5. The user interface waits for further user interactions, which restarts the cycle.

By decoupling models and views, MVC helps to reduce the complexity in architectural design and to increase flexibility and reuse of [code](http://en.wikipedia.org/wiki/Source_code)

**Structural Design Pattern:** - *Bridge DP*

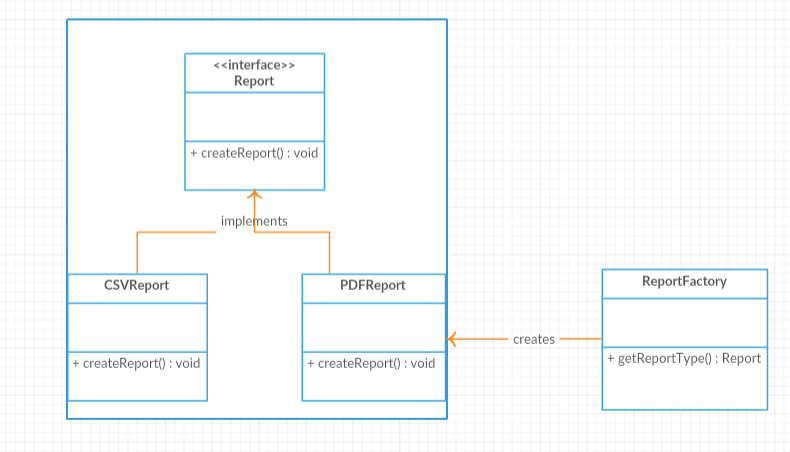
As a structural design pattern I have decided to use Bridge DP to create the users of my application. Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other. I will demonstrate the use of bridge design pattern via the following diagram:



**Creational design Pattern:** - *Factory DP*

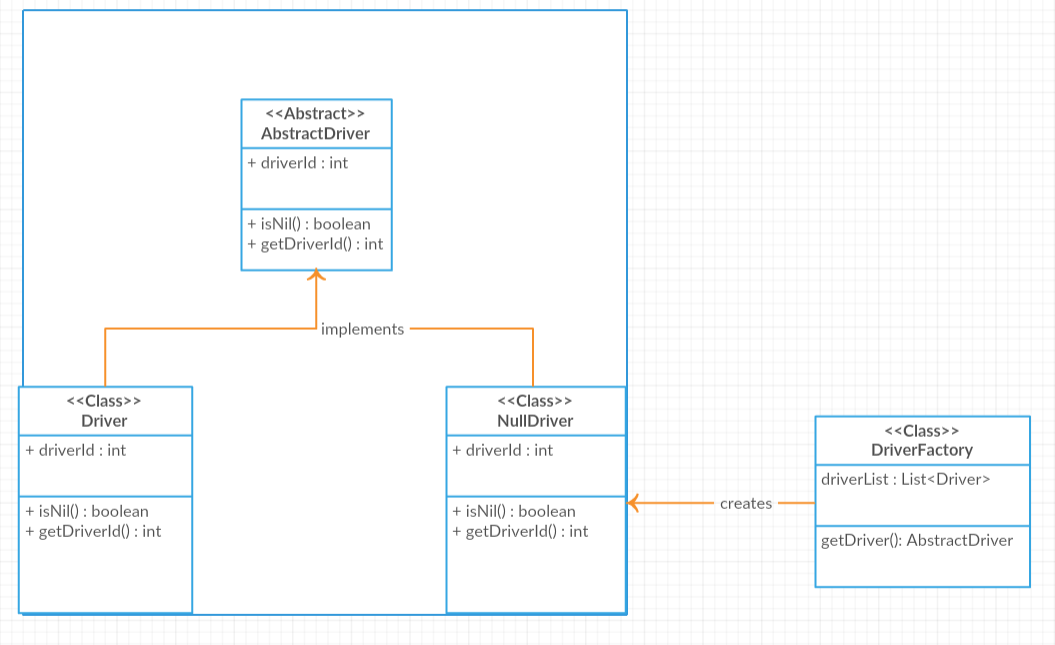
Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

**Behavioral Design Pattern:** - *Null Object DP*

As a behavioral design pattern I have decided to use Null Object DP. In Null Object pattern, a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do nothing relationship. Such Null object can also be used to provide default behaviour in case data is not available.

In Null Object pattern, we create an abstract class specifying various operations to be done, concrete classes extending this class and a null object class providing do nothing implemention of this class and will be used seemlessly where we need to check null value.



# Design Model Refinement

## *[Refine the UML class diagram by applying class design principles and GRASP; motivate your choices. Deliver the updated class diagrams.]*

# Construction and Transition

# System Testing

A unit test is a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state. The percentage of code which is tested by unit tests is typically called test coverage. A unit test targets a small unit of code, e.g., a method or a class. External dependencies should be removed from unit tests, e.g., by replacing the dependency with a test implementation or a (mock) object created by a test framework. Unit tests are not suitable for testing complex user interface or component interaction. For this, you should develop integration tests. Data-flow testing looks at the life-cycle of a particular piece of data (i.e. a variable) in an application. By looking for patterns of data usage, risky areas of code can be found and more test cases can be applied. There are four ways data can be used: defined, used in a predicate, used in a calculation, and killed. Certain patterns, using a piece of data in a calculation after it has been killed, show an anomaly in the code, and therefore the possibility of a bug.

# Bibliography

[1] https://github.com/kittyrad/laborator1

[2] https://en.wikipedia.org/wiki/Package\_diagram

[3] https://stackoverflow.com/questions/17390049/component-diagram-versus-class-diagram

[4] https://www.lucidchart.com/pages/uml-deployment-diagram

[5] https://www.ibm.com/developerworks/rational/library/3101.html

[6] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-communication-diagram/

[7] https://creately.com/

[8] https://www.tutorialspoint.com/design\_pattern/factory\_pattern.html

[9] https://www.tutorialspoint.com/design\_pattern/bridge\_pattern.htm