Assignment 1

PG4100 - Jonas Jensen – [jenjon13@student.westerdals.no](mailto:jenjon13@student.westerdals.no)

# Preface

The following document represents the documentation for the included application, being my take on the first assignment of the PG4100 course. This document replaces the standard video demonstration. Additionally, the code is documented through the usage of Javadoc headers.

# Running the application

The included files represent a Maven project. Install the project’s dependencies by running “mvn install” in a command prompt with the current directory set to the root of the project. Then run the application by executing the main method located within Controller.java. This displays the main user interface, as described below. Enter a minimum of five customer names in the input field and the application will start the threads.

# Execution flow

As customer names are added in the form, each entered name is assigned to a Thread.

As the fifth customer name is added, each associated Thread is started. The lifecycle of each thread, as defined by customer.CarCustomer, is simple. When running, the CarCustomer instance will sleep for 1-3 seconds before requesting a lease. Having acquired a lease, the instance will sleep for 1-10 seconds whilst in ownership of a lease lock securely assigned by the CarLeaser singleton instance, before finally ending the lease. This cycle is repeated for as long as the thread is running.

# Purpose

The program’s functionality demonstrates a thread-safe approach to the *consumer/producer*-problem. The goal of this application is to simulate a threaded program where the various threads continuously acquire and release hold of resources from a shared resource pool. The inherent challenge presents itself in the form of race conditions derived from the concurrent nature of the program’s many threads. This challenge is dealt with by proper usage of synchronization locks in critical areas, as seen in leaser.CarLeaser.

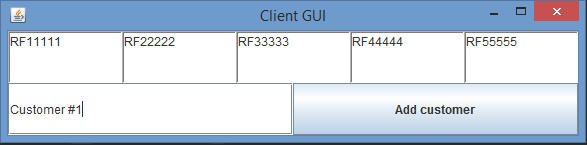
# Structure

The application is structured in accordance with the MVC pattern, with Controller representing the main controller of the application. It implements the Observer interface, as the delegation of concerns is enhanced using the Observable pattern. Being an *Observer* of gui.View that makes out the presentation layer, Controller receives updates about the values that a user enters in the view, and requests new Customer threads made from the CustomerFactory accordingly.

Various design patterns, such as Factory, Model-View-Controller and Observable, have been utilized to achieve a better architecture and a more solid and extensible codebase. Whilst not necessarily providing a tremendous impact for a task of such limited scope, design patterns are nice practices which benefit developers in the longer run.

# Presentation layer

The View, implementing both Observer and extending Observable, is set by the Controller to listen for updates to the LeaseCars held by the CarFactory. This allows the view to be as detached as possible from the logic in the business layer.



The view consists of simple input and output fields. The output fields contain information about the LeaseCar instances, displayed as text boxes containing the license plate identifiers and various other info made available by the respective cars.

The input fields, a textbox and a button, allow the user to add “customer” threads to the application. The *Enter* key may be used instead of the button to add a customer.

# Testing

The included tests ensure the integrity of the application code. The tests cover the core functionality of the application, ensuring the business logic works as intended.

Whilst reported code coverage appears to be mediocre, all essential business logic functionality is covered. When testing for code coverage, it is important to keep in mind that testing is a tool for the developer, and not vice versa – in the end, there must exist a purpose in testing of any given snippet of code. Hence, workarounds for covering snippets of code reported to be untested (private constructors of static classes and other such cases) have not been applied. Given the limited scope and size of this application, the presentation layer is not tested either.