**DESIGN REVIEW: ECOBIKE APPLICATION – GROUP 6**

**1. Coupling**

**1.1. Content coupling**

|  |  |  |
| --- | --- | --- |
| Related modules | Description | Improvement |
| No related module | All modules must be accessed via getter. Some special attributes are closed and can only be set via constructor and logics inside the modules | No improvement |

**1.2. Common coupling**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| No related module | None | No improvement |

**1.3. Control coupling**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
|  | All functions are implemented by its functionalities, without being depended on the input params. Abstract classes and interfaces, with Factory pattern are used to resolves the problems of many kinds of bikes in the system | No improvement |
| FunctionalUtils | One static function needs to switch based on tag – however this is one part of the design pattern (Factory pattern) | No improvement |

**1.4. Stamp coupling**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| RentBikeService controller | In module RentBikeServiceController, the Bike entities was used as an argument for the calculateFee method, which only need bikeType and totalRentTime as arguments | Fix the method to accept only needed arguments instead of the accepting Bike entities as the argument |

**1.5. Data coupling**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| Controllers and Boundaries modules | Boundaries need data to render GUI, which is acceptable | No improvement |

**2. Cohesion**

**2.1. Coincidental cohesion**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| No module | The only visible coincidental cohesion in our project might be the class Configs, which contains some constant share between some controllers and entities | No improvement |

**2.2. Logical cohesion**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| DBUtils | All the functions related to DB are put here | Divide the functions related into Bike and Dock classes so that they can perform their work with the database |

**2.3. Temporal cohesion**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| None | None | No improvement |

**2.4. Procedure cohesion**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| None | None | No improvement |

**2.5. Communicational cohesion**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| No module | None | No improvement |

**2.6. Sequential cohesion**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| Mostly modules | Output of one function of this module is input of another module | No improvement |

**2.7. Information cohesion**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| DBUtils  JSONUtils | All methods are to perform database queries or manipulate json string | No improvement |

**2.8. Functional cohesion**

|  |  |  |
| --- | --- | --- |
| **Related modules** | **Description** | **Improvement** |
| Most of the modules | Elements in the modules are needed for its functionalities | No improvement |

**3. SOLID consideration**

* Single responsibility: some functions with different purposes are put in the same class, which make the design not good. These functions should be separated into different smaller class, which is OK since we have already had a base class for controller to extends
* Open-closed: Use interfaces and abstract classes to design the basic entities so that further requirements for adding bikes or methods of calculating fees can be easily adapted
* Liskov substitution: all types of Bikes (Normal bike, Ebike) can replace Bike type
* Interface segeration: No specific modules
* Dependency inversion: Communicate between subsystem by interfaces, which are implemented by different boundaries so that the changes in one system has minimal affect on other modules

**4. Pattern**

***Singleton***

* Used for UI: for better updating data when information about the entity changes
* Used for controller: This may cause the common coupling (?) but will make it easier for making a center head to process data, reduce the cost for resolving conflicts about concurrency. IRL, each center for processing will create handlers for connecting and receiving requests from clients, and threads are implemented to process them, instead of creating clones of servers to serve the customer.

***Observer***

* Docks are observer of Bikes, and the information controller is observer of Docks
* Each time a Bike is created, it registers a dock as an observer if it is in a dock
* Each time a bike changes its state, this will be notified to the dock
* Any changes of the dock (including the number of bikes in the dock) will be notified to the information controller, so that the controller can update its data
* This reduces the time application query to the DB, increase running time of application

***Factory***

* Using one static function to create different instance of bike based on its type passed to the function
* These kinds of bike are of one parent class (Bike), and all other class only need to operates on Bike instance (no need to care about its real type)

***Façade***

* Used to implement the boundary class for communicating between subsystems
* This make the design more flexible with the extension of subsystems when minimize the affects on other classes as much as possible