# Problem 1 – Marketplace

Once you know the city of Kermen, you will certainly become aware of one of the shining stars in the city – the local software developer, Mateyko, who is one of the prowd owners of an AloneYoung() home. He was once assigned a task to write a marketplace statistic system, where customers of three predefined store types can create, read and update products.

Mateyko was familiar with separation of concerns and wanted each event to be handled by a Controller. The controller asks the database and eventually returns the desired result to the customer. So he started to write his own framework which does the work magically.

He has heard that annotations are one core thing in Java where you can extract meta-information in order to create dynamic mapping. All of these local “new” operators are annoying and testing the user input manually whether it matches certain method or not – very verbose.

You are given Mateyko’s framework which is an annotation parser library and some annotations created by him. Here’s what he has written (or at least what he wanted to write, because there might be some bugs in his code):

1. **@Controller** – this annotation denotes that certain class is a controller. A controller is a class that is invoked upon user input, depending in the Request Method and the URI that the user has sent in its request.
2. **@RequestMapping** – this annotation denotes that certain controller’s methods should be invoked in the user’s input. It contains two parts:
   1. **value() –** this is the URI that the user has sent. It might contain placeholders e.g. “/users/{id}”. When the user sends “/users/4” it triggers that method
   2. **method() –** the request method that the user has sent. It can be: **EDIT, ADD, GET** or **DELETE**. The combination of request method and URI should match the user input in order to invoke the method
3. **@Component –** this annotation denotes that this class implements an interface which will be injected as a field dependency somewhere else. (e.g. **@Component public class UserImpl implements User**)
4. **@Inject –** this annotation denotes that the field should be populated with the value of the implementation of the interface which has **@Component** over it (e.g. **@Inject private User user**)
5. **@UriParameter –** this annotation denotes the name of the Controller->Method()’s argument in the @RequestMapping.value(). (e.g. **@RequestMapping(value = “/users/{lastName}”) public User getByLastName(@UriParameter(“lastName”) String lastName)) { … })**

There is an **AnnotationParser** class that scans these annotations and fills some in memory cache for later usage, exposing some public methods that you can use in order to create the logic of invoking controllers, methods and so forth.

### Refactor the Framework

The framework is very poorly written. It violates a lot of the principles of good object oriented design and general coding guidelines. It also might have some bugs.

### Use the Framework

You will need to use the public API of the framework in order to invoke let’s say controllers and give them the data they expect.

### Implement Core Logic

The core logic of the task is as follows:

There are three kinds of markets in the lifecycle of the application: **store**, **bazaar** and a **mall**. There are also products – a **big product** and a **small product**. Each product has a **name** and a **size** and can be **registered** either in the **application** or in **one of the markets**. The markets have predefined fixed **capacity** that they can hold, except the **Mall** which is **infinite**. The store has a **capacity** of **15**. If the product cannot fit in the store it must be sent to the **bazaar**. The bazaar has a **capacity** of **30**, if the product cannot fit in the bazaar it must be sent to the **mall**. The **mall** is the ultimate market **with unlimited capacity**.

Each product is registered from the customer by **name**, **size** and **type**. The **big** product **doubles** its **size** when registered and the **small** product **halves** its **size**.

Sending **products** to the **markets** is done by **ID**, which is **automatically** **incremented** and **assigned** when the product is **registered** in the application. The first product is with **ID=1** and the **N’th** with **ID=N**.

By accessing certain URI’s with certain request methods (let’s say commands) customers can **add** a product by all of its three characteristics, extract a list of products by some of the characteristics and edit a product by its **ID** and providing all of the characteristics.

### Qualify Core Logic

Make your application code meet the standards of the object oriented programming and object oriented design, by the well-known principles and patterns such as loose coupling, strong cohesion, opened for extensions, closed for modification and so on.

### Tests

The application needs some testing. Create certain type of tests for certain functionality comprising the border cases:

1. Unit tests – Editing a product. As your application might be multi-tier, write down unit tests for each layer which is involved in the editing product chain.
2. Integration tests – Editing a product. The same functionality again. This time it must be tested as a whole. Make sure you do not depend on external modules.

### User Input

1. **ADD /product/{size}/{name}/{type}**
   1. **Registers a product in the application with the given size and name, by the given type**
   2. **The {size} will be a valid positive integer**
   3. **The {name} will be a non-empty string**
   4. **The {type} will be either “SmallProduct” or “BigProduct”**
   5. **The expected output is in the format “Product {id} registered successfully”**
2. **GET /product/{size}/{name}/{type}**
   1. **Returns a set of products from the application and markets by the given size, name and product type**
   2. **The {size} will be a valid positive integer**
   3. **The {name} will be a non-empty string**
   4. **The {type} will be either “SmallProduct” or “BigProduct”**
   5. **The expected output for each product is in the format “{productType}: {id}. Size: {size}. Name: {name}” on a separate line**
   6. **Alternative output “No products by the given criteria”**
3. **GET /product/{size}/{name}**
   1. **Returns a set of products from the application and markets by the given size and name**
   2. **The {size} will be a valid positive integer**
   3. **The {name} will be a non-empty string**
   4. **The expected output for each product is in the format “{productType}: {id}. Size: {size}. Name: {name}” on a separate line**
   5. **Alternative output “No products by the given criteria”**
4. **GET /product/{id}**
   1. **Returns a single product from the application or markets by the given ID**
   2. **The expected output is “{productType}: {id}. Size: {size}. Name: {name}”**
   3. **Alternative output “Product {id} does not exist”**
5. **EDIT /product/{id}/{newName}/{newSize}**
   1. **Modifies existing product by given ID**
   2. **{newName} will be valid non-empty string**
   3. **{newSize} will be valid positive integer**
   4. **The expected output is “Product {id} successfully edited”**
   5. **Alternative output “Product {id} does not exist”**
6. **ADD /shop/{type}/{productId}**
   1. **Adds existing product to certain market by given ID**
   2. **{type} will always be either Mall, Store or Bazaar**
   3. **Expected output “Product {id} moved to market {type}”**
   4. **Alternative output “Product {id} does not exist” when the product does not exist**
   5. **Alternative output “Product {id} is already registered to a market {marketType}” when the product is not in the application, but already in any of the predefined markets**
7. **GET /shop/{type}**
   1. **Returns a set of products from the given market type**
   2. **{type} will always be either Mall, Store or Bazaar**
   3. **The expected output for each product is in the format “{productType}: {id}. Size: {size}. Name: {name}” on a separate line**
   4. **Alternative output “No products by the given criteria”**

### Input

* Each command will come on a new line
* The input ends when you receive the command “**ILIENCI**”.
* The input will be no more than **400,000,000** lines per each command

### Output

* Every listing is in order of appearance

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| **ADD /product/4/pesho/SmallProduct**  **ADD /product/4/pesho/BigProduct**  **ADD /product/1/pesho/BigProduct**  **GET /product/2/pesho**  **ILIENCI** | **Product 1 registered successfully**  **Product 2 registered successfully**  **Product 3 registered successfully**  **SmallProduct: 1. Size: 2. Name: pesho**  **BigProduct: 3. Size: 2. Name: pesho** |
| **Comment** | |
| **Small product with ID 1 halves its size twice becoming with size 4 / 2 = 2. Big product with ID 3 doubles its size so it becomes 1 \* 2 = 2.** | |

|  |  |
| --- | --- |
| **Input** | **Output** |
| **ADD /product/4/pesho/SmallProduct**  **ADD /product/4/pesho/BigProduct**  **ADD /product/1/pesho/BigProduct**  **EDIT /product/2/gosho/13**  **ADD /product/12/kofa/SmallProduct**  **ADD /product/11/condom/BigProduct**  **ADD /shop/Store/1**  **GET /shop/Bazaar**  **ADD /shop/Store/2**  **ADD /shop/Store/3**  **ADD /shop/Store/4**  **ADD /shop/Store/5**  **GET /shop/Store**  **GET /shop/Bazaar**  **GET /shop/Mall**  **ILIENCI** | **Product 1 registered successfully**  **Product 2 registered successfully**  **Product 3 registered successfully**  **Product 2 successfully edited**  **Product 4 registered successfully**  **Product 5 registered successfully**  **Product 1 moved to market Store**  **No products by the given criteria**  **Product 2 moved to market Bazaar**  **Product 3 moved to market Store**  **Product 4 moved to market Store**  **Product 5 moved to market Mall**  **SmallProduct: 1. Size: 2. Name: pesho**  **BigProduct: 3. Size: 2. Name: pesho**  **SmallProduct: 4. Size: 6. Name: kofa**  **BigProduct: 2. Size: 26. Name: gosho**  **BigProduct: 5. Size: 22. Name: condom** |