HW4 User's Manual

writer : Juyoung Ha (21700760)

1. Visitor Pattern

본 프로그램에서 visitor pattern 은 모든 동작의 Main 이 되는 pattern 에 해당합니다.

```
public static Component[] list
        = { new Wheel( location: 1), new Wheel( location: 2),
        new Wheel( location: 3), new Wheel( location: 4),
        new Motor( location: 1),  new Motor( location: 2),
        new Motor( location: 3),  new Motor( location: 4),
        new Solar_panel( number: 1,  capacity: 20 + 2 * random.nextInt( bound: 4)),
        new Solar_panel( number: 2,  capacity: 20 + 2 * random.nextInt( bound: 4)),
        new Solar_panel( number: 3,  capacity: 20 + 2 * random.nextInt( bound: 4)),
        new Solar_panel( number: 4,  capacity: 20 + 2 * random.nextInt( bound: 4)),
        new Solar_panel( number: 5,  capacity: 20 + 2 * random.nextInt( bound: 4)),
        new Solar_panel( number: 6,  capacity: 20 + 2 * random.nextInt( bound: 4)),
        new Robot_arm( location: 1), new Robot_arm( location: 2),
        new Camera() };
// execute the each job
System.out.println("The First Visitor for Checking Components\n");
for (Component comp : list)
    comp.accept(first_visitor);
System.out.println("The Second Visitor for Maintaining Components\n");
for (Component comp : list)
    comp.accept(second_visitor);
System.out.println("\nThe Third Visitor for Rover Movement\n");
pageMaker.buildHtmlPage(third_visitor, list);
System.out.println("Operation History Being Recorded in HTML file");
System.out.println("Rover_Running.html is made");
```

위의 코드는 제가 작성한 main function 의 일부 입니다. 위의 코드를 통해서 볼 수 있듯이 기본적으로 필요한 object 들을 생성하여 list 에 저장하되, 각 object 에 대한 모든 동작의 처리는 vistior 에 의해서 수행됩니다. 그래서 위의 main function 일부 코드를 통해 볼 수 있듯이 list 에 저장하여 유지하고 있는 각 object 들에 대하여 accept method 를 통해 visitor 를 전달하는 것을 볼 수 있습니다. 이 프로그램에서 정의된 모든 class 와 그에 대한 object 에 대한 접근은 모두 vistor 에 의해 수행됩니다. 본 프로그램에서 vistor 는 총 3 개(EventCheck_Visitor, EventHandle_Visitor, Html_Visitor)가 존재하며 각각 각 component 들에 대한 이벤트 확인, 발생한 이벤트에 대한 처리, 정상 동작에 대한 html 문서를 형성하는 동작을 수행합니다.

2. Singleton Pattern

본 프로그램에서는 발생한 이벤트에 대한 기록을 유지하는 BlackBox class 의 Object 에 대해서 Singleton Pattern 을 사용하였습니다. 이는 EventCheck_Visitor, EventHandle_Visitor 가 BlackBox Object 의 자원을 공유하여 사용할 수 있도록 하기 위함이며 아래의 코드를 통해서 알 수 있듯이 constructor 가 private 을 사용하고 있고, 대신 해당 class 에 대한 object 를 얻기 위한 getInstance method 를 정의하여 사용하고 있는 것을 확인할 수 있습니다. 그리고 실질적인 Object 의 생성은 static keyword 를 사용하여 blackbox 라는 Object 를 정의함과 동시에 생성하는 것을 확인할 수 있습니다.

```
public class BlackBox implements Component, Cloneable {
    private int num_of_record; // the number of record
    private static BlackBox blackBox = new BlackBox();
    private ArrayList<String> history;

    // constructor
    private BlackBox() { history = new ArrayList<String>(); // generate storage list }

    // return the instance of Blackbox
    public static BlackBox getInstance() { return blackBox; }
```

그리고 두번째로 생성한 Solar Panel 들을 유지하는 class 인 "Panel_storage" class 도 아래의 코드에서 확인할 수 있는 것처럼 여러 Solar Panel 의 Object 들이 이를 공유하여 사용할 수 있도록 하기 위하여 Singleton Pattern 을 사용하였습니다. 아래의 코드를 통해서 확인할 수 있듯이 "Panel_storage" class 도 constructor 를 private 으로 지정하고 있고, object 를 얻기 위한 getInstance method 를 정의하고 있는 것을 확인할 수 있습니다.

```
public class Panel_storage {
    private Map<Integer, Solar_panel> storage = new HashMap<~>();
    private static Panel_storage panel_storage = new Panel_storage();

    // constructor
    private Panel_storage() {};

    // return the instance
    public static Panel_storage getInstance() { return panel_storage; }
```

3. Prototype Pattern

본 프로그램에서는 Robot Arm 에 대한 Class 인 "Robot_arm" class 에서 이를 사용하고 있습니다. 아래와 같이 Clonable 을 interface 로 inheritance 받고 있고, clone method 를 정의하여 사용하고 있는 것을 확인할 수 있습니다.

```
public class Robot_arm implements Component, Cloneable
```

```
public Object clone() {
    Object clone = null;
    try {
        clone = super.clone();
    }
    catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return clone;
}
```

실제로 Robot Arm 에 대한 이벤트가 발생한 경우 아래의 코드와 같이 clone method 를 사용하여 기존의 arm method 를 복제하고 이를 사용하여 새롭게 교체하는 동작을 구현한 것을 확인할 수 있습니다. (아래의 코드는 event 에 대한 action 을 담당하는 EventHandle_Visitor 에서 확인하실 수 있습니다.)

```
if(c.getStatus() <= 2) {
    Robot_arm newArm = (Robot_arm)c.clone(); // clone the new arm
    newArm.setStatus(10); // set the new arm
    c = newArm; // replace the arm
    System.out.println(event_message1 + c.getLocation() + event_message2
            + c.getLocation() + " Arm Clone " + c.getLocation() + event_message3);
    backup_message = "1" + "3" + c.getLocationCode();
    visit(blackBox, backup_message);
}
```

4. Flyweight Pattern

본 프로그램에서는 Solar panel 를 나타내는 Solar_panel class 에 Flyweight Pattern 을 사용하였습니다. 하지만 여기서 각각의 Solar_panel class 는 각각 하나의 Solar Panel 을 표현하는 class 이고 Flyweight Pattern 을 사용하기 위해서는 이들을 모아서 관리할 수 있는 하나의 또 다른 class 가 필요하기 때문에 별도로 "Panel_storage" 라는 class 를 정의하여 사용하였습니다. 그래서 실질적으로 Flyweight Patterndml "Panel_storage"에 사용되었으며 이로인한 혜택을 Solar_panel class 가 받는다고 보시면 되겠습니다.

```
private Map<Integer, Solar_panel> storage = new HashMap<~>();
private static Panel_storage panel_storage = new Panel_storage();
Solar_panel lookup(int capacity) {
    if(!storage.containsKey(capacity)) {
        storage.put(capacity, new Solar_panel( number: 0, capacity));
        System.out.println("Making a New Solar Panel : " + capacity + " KW Panel");
    }
    return storage.get(capacity);
}
```

위의 코드를 통해서 확인할 수 있듯이 우선 solar panel 들을 효율적으로 관리하기 위한 별도의 Hashmap 을 선언한 것을 확인할 수 있습니다. 그리고 그 아래에 lookup 이라는 method 를 정의하여 capacity(20W, 22W, 24W, 26W)를 기준으로 해당 solar panel 이 존재하는 지에 대한 유무를 확인하고 존재하지 않는다면 새롭개 생성한 후 Hashmap 에 이를 넣어준 후 반환을, 이미 존재하는 경우에는 새롭게 생성하지 않고 Hashmap 에서 이를 가져다 사용하는 모습을 확인할 수 있습니다.

우선 초기에 6 개의 Solar Panel 을 추가할 때에는 capacity(20W, 22W, 24W, 26W)를 random number generator 를 사용하여 할당하고 이 초기에는 해당 panel 들을 hashmap 에 추가하지 않습니다. 그리고 solar panel 에 대한 이벤트가 발생하고 이를 Handle 하는 과정부터 주어진 capacity(20W, 22W, 24W, 26W)의 panel 을 생성하여 hashmap 에 넣어주고 Flyweight Pattern 을 적용하여 사용하게 됩니다.

5. Facade Pattern

본 프로그램에서는 정상적인 상태의 동작을 html 문서 형태로 기록하기 위해서 정의된 "PageMaker" class 에 Facade Pattern 을 사용하였습니다. 아시다시피 본 과제에서 요구되는 html 문서를 생성하는 코드를 작성하기 위해서는 상당히 많은 라인 수의 코드와 logic 의 구성이 필요합니다. 하지만, 이를 사용자 측면에서 바로 사용하도록 하기에는 다소 무리가 있습니다. 그래서 Html_Visitor 에서 실질적으로 html 문서에 내용을 기록하는 동작들을 수행하지만, 이를 논리적으로 잘 정리 및 구성해서 사용자 측면에서 더 높은 수준의 API 를 제공할 수 있도록 Facade Pattern 을 사용했습니다.

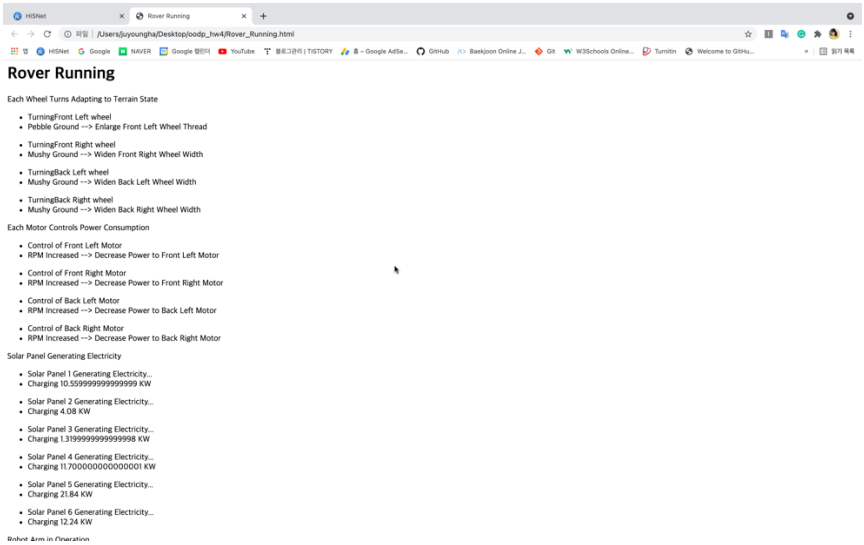
```
public void title(String title) throws IOException {
    writer.write( str: "<html>\n");
    writer.write( str: "<head>\n");
    writer.write( str: "<title>" + title + "</title>\n");
    writer.write( str: "</head>\n");
    writer.write( str: "<body>\n");
    writer.write( str: "<h1>" + title + "</h1>\n");
}
```

```
public class PageMaker {
    public void buildHtmlPage(Html_Visitor visitor, Component[] list) throws IOException {
        visitor.title("Rover Running");
        list[0].accept(visitor);
        list[1].accept(visitor);
        list[2].accept(visitor);
        list[3].accept(visitor);
        list[4].accept(visitor);
        list[5].accept(visitor);
        list[6].accept(visitor);
        list[7].accept(visitor);
        list[8].accept(visitor);
        list[9].accept(visitor);
        list[10].accept(visitor);
        list[11].accept(visitor);
        list[12].accept(visitor);
        list[13].accept(visitor);
        list[14].accept(visitor, (Robot_arm)list[15]);
        list[16].accept(visitor);
        visitor.close();
    }
}
```

왼쪽의 코드는 Html_Visitor class 코드의 일부이고, 오른쪽은 이를 더 높은 수준의 API 제공하기 위해서 정의한 PageMaker class 코드의 일부입니다. 위에서 확인할 수 있듯이 PageMaker class 는 buildHtmlPage 라는 하나이 method 정의하고 이는 여러개의 Html_Visitor 의 호출을 동반하고 있는 것을 확인할 수 있습니다. 하지만 실질적으로는 이렇게 복잡하고 많은 라인의 코드로 구성되었다고 할지라도 사용자 입장 즉, main function 에서는 아래와 같이 buildHtmlPage method 에 사용할 vistor 의 종류와 object 들이 저장된 list 만을 전달함으로서 복잡한 html 문서작성 과정을 수행할 수 있습니다.

```
pageMaker.buildHtmlPage(third_visitor, list);
```

프로그램 실행을 마치면 아래와 같이 웹 브라우져에서 열람할 수 있는 형태의 html 문서가 생성된 것을 확인할 수 있습니다.



5. 추가정보

추가적으로 본 프로그램에서는 Blackbox 에 데이터를 유지하기 위해서 Blackbox 라는 class 를 정의하여 사용하고 해당 class 내에서 아래와 같이 하나의 ArrayList 를 선언하여 사용합니다. 그런데 여기서 String 을 사용하고 있는 것을 알 수 있는 데 여기서 정보 저장과정의 효율성을 높이기 위해 encoding, decoding 의 개념을 사용했습니다.

```
private ArrayList<String> history;
```

우선 아래와 같이 각 case 들에 대한 정보를 하나의 숫자문자로 표현하기 위해 다음과 같은 decoding 을 수행하였고, 실질적으로 blackbox 에 저장되는 값도 decoding 을 거친 3 자리 숫자문자열입니다.

BlackBox

- 저장해야하는 거

1. Alarm인지, Action인지 - 0, 1
2. 어떤 파트에 대한 것인지

- wheel - 0
- motor - 1
- solar panel - 2
- robot arm - 3
- camera - 4

3. 각 파트에 대한 세부정보

- wheel
  - front left - 1
  - front right - 2
  - back left - 3
  - back right - 4
- motor
  - front left - 1
  - front right - 2
  - back left - 3
  - back right - 4
- solar panel
  - 1 - 1
  - 2 - 2
  - 3 - 3
  - 4 - 4
  - 5 - 5
  - 6 - 6
- robot arm
  - front - 1
  - rear - 2
- 세부파트가 1개인 파트들은 모두 0으로 처리

명령어 사용 시 "/"를 입력하세요

<example>

003 - back left wheel에서 발생한 alarm

그리고 나중에 blackbox 의 내용을 사람이 이해할 수 있는 형태로 출력할 때에는 아래와 같이 encoder 라는 method 를 정의하고 이를 사용하여 사람이 이해할 수 있는 문자열의 형태로 변환하여 이를 반환해주게 됩니다.

```
public String encoder2(char c0, char c1, char c2) {
    if(encoder1(c0).equals("Alarm")) {
        if(c1 == '0') // wheel
            return "Mud and Dust on Wheel Axle of " + encoder3(c2) + "wheel";
        else if(c1 == '1') // motor
            return "Weak Power to " + encoder3(c2) + " Motor";
        else if(c1 == '2') // solar panel
            return "Low Electricity Generated to Solar Panel " + c2;
        else if(c1 == '3')
            return encoder4(c2) + " Robot Arm Not Working Properly";
        else // c1 == 4
            return "Camera Not Working Properly";
    }
    else { // encoder1(c0).equals("Action")
        if(c1 == '0') // wheel
            return "Removing Mud/Dust from " + encoder3(c2) + " Axle";
        else if(c1 == '1') // motor
            return "Supply more power to " + encoder3(c2) + " Motor";
        else if(c1 == '2') // solar panel
            return "Replacing Solar Panel " + c2;
        else if(c1 == '3')
            return encoder4(c2) + " Robot Arm Replaced by Prototype " + encoder4(c2) + " Ar
        else // c1 == 4
            return "Repair Camera";
    }
}
```