**Sample Result**

OUTPUT 결과는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

**User Manual**

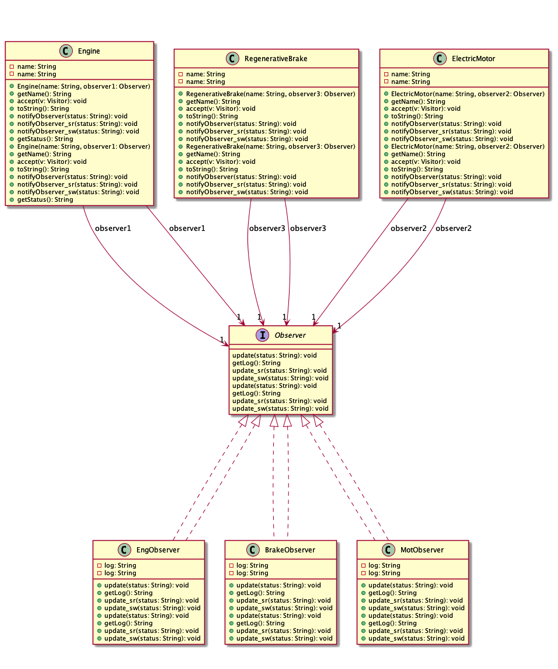
* 실행 방법

Javac \*.java

Java VisitorDemo.java

**Design Document**

이 프로그램에는 Visitor, Observer, Singleton, Abstract, Factory, Builder 패턴이 사용되었다. 먼저 프로그램이 시작되면 Motor, Engine, Brake 객체가 생성되고 그에 맞는 observer들을 생성된다. Visitor를 통해 실제로 각 장치들은 검사를 받도록 실행이 된다. 즉 각 visitor가 3개의 장치를 검사하는 방식이다.

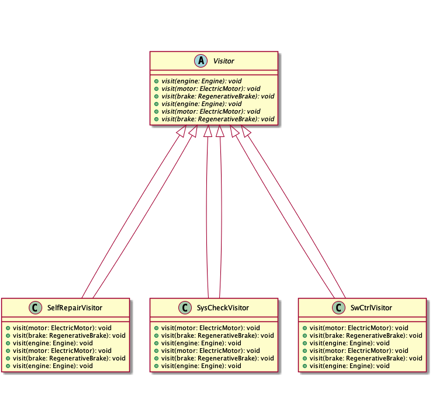


* System/Component Checking Visitor에서는 3개의 장치를 방문한다. 이때 singleton에 의해 생성된 각 부분의 status를 바탕으로 상태를 점검한다. Singleton은 Random한 숫자로 각 상태를 결정한 후 이를 3개의 Visitor에서 모두 사용한다. Singleton을 통해 알게된 state들을 바탕으로 visitor는 각 장치를 처리한다. Singleton UML은 다음과 같다.

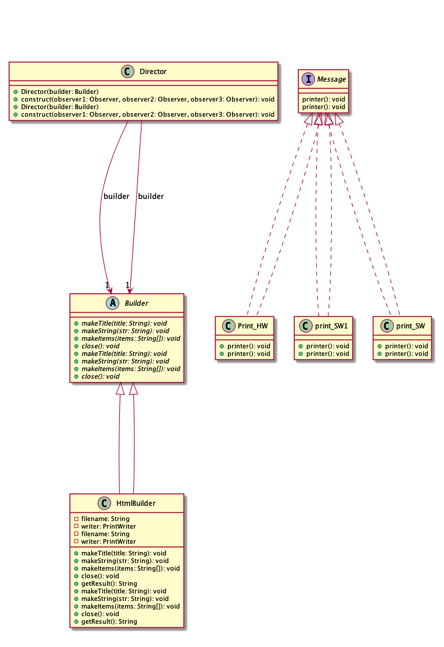
텍스트이(가) 표시된 사진

자동 생성된 설명

* 각각의 visitor는 총 3개의 Visit()으로 구성되어 있는데 이는 각각은 3개의 장치에 따라 다르게 호출된다. 그래서 각각 필요한 처리 과정은 visitor들이 호출될 때 어떤 장치에서 호출하는지에 따라 다르게 구성하였다. 밑에 UML은 3개의 visitor의 구조를 나타낸다.



* Self Repairing visitor와 software Control visitor 역시 3개의 visit()으로 되어 있고 각각은 다른 장치를 검사한다.
* Engine, Motor, Brake는 각각 자신의 상태를 Observer에게 알린다. 그래서 현재의 log를 observer들이 updatae하도록 한다. 또한 System/Component Checking Visitor에 의한 observer 결과는 후에 builder에게 알려줘 html 파일을 생성하게 된다. 이때 Builder Pattern이 사용된다. Director class에서 html에 대한 정보를 가지고 파일을 build 하면, Builder class에 의해 정의된 HtmlBuilder가 형식에 맞춰서 director가 준 정보를 바탕으로 실제로 파일을 생성하게 된다. 밑에 UML은 Builder pattern의 구조를 나타낸다.



* Html의 실제 결과는 다음과 같다. 첫번째 Visitor의 결과를 나타내도록 하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* 위에 UML에는 Message 클래스가 있는데 이는 Factory Pattern이다. 특정한 조건이 있을 경우, Factory에서 조건에 맞도록 정해진 문장을 출력한다. 이때에는 Abstract Factory Pattern이 사용되어, abstract class로 Factory의 구조를 정의한다. 해당 시스템에서는 3가지의 특정 조건에서는 Factory pattern이 필요하였기 때문에, HW와 2개의 SW로 각각 조건에 맞는 출력을 하도록 하는 class들이 정의되도록 프로그램을 구성하였다. 밑에는 이를 표현한 UML이다.

