정적분석 대응서

June 12nd 2014

Team 3

200913215 이인구

201013275 강태호

201013760 기세파

적정분석에서 나타난 문제의 거의 대부분은 OOP 가 이루어지지않았기 때문에 발생한것으로 생각되며 그렇기 때문에 Dependency 분석에 대한 대응만 이루어지면 나머지 문제는 해결될 것이라 생각하였다.

1. Dependency 검사 관련 : OOP 프로그래밍 실력의 부족으로 static 변수를 많이 넣었고 이는 심각한 클래스 상호의존과 함께, 메모리 누수를 낳았다. 이에 따라 객체지향 프로그래밍을 적용 하였다. static 변수는 GUI 관련을 제외하고 사용할수 있도록 변경하였다.

```
1⊕ import javafx.application.Application;
5
6
7  public class Fluxvator extends Application{
8      static Elevator elev1;
9      static Elevator elev2;
0      static SimulatorController sc;
1      static QueueAlgorithm queueAlgorithm;
```

```
4  public class QueueAlgorithm {
5      int emergencyDestination;
6      static QueueNode elev1DestinationNode;
7      static QueueNode elev2DestinationNode;
8      static QueueNode elev1NDestinationNode;
9      static QueueNode elev2NDestinationNode;
10     static Queue selectQueue1, selectQueue2, aboardQueue;
11     Elevator elev1=Fluxvator.elev1;
12     Elevator elev2=Fluxvator.elev2;
13     Iterator itr1, itr2, itr3;
```

수정 전 코드 : static 변수를 직접 참조할수있도록하였고, 이로 인하여 심각한 의존성 문제와 메모리 누수 문제가 발생하였다.

```
7  public class Fluxvator extends Application{
8      Parent root;
9      Scene scene;
10⊖    public void start(Stage primaryStage){
```

```
7  public class QueueAlgorithm extends Observable implements Observer{
8      int emergencyDestination;
9      QueueNode elev1DestinationNode;
10     QueueNode elev2DestinationNode;
11     QueueNode elev1NDestinationNode;
12     QueueNode elev2NDestinationNode;
13     Queue selectQueue1, selectQueue2, aboardQueue;
14     Elevator elev1;
15     Elevator elev2;
16     Iterator itr1, itr2, itr3;
```

수정 후 코드 : 메인 클래스에 static object들을 생성하던 예전과 달리, 시퀀스 다이어그램에 순서

진행도에 맞게 object들이 생성되도록 조정하였으며 static변수들이 사용하지 않도록 하였다. Dependency를 줄이기 위하여 QueueAlgorithm 클래스와 Elevator 클래스가 has-a 관계를 가지도록 조정하였으며, 이를 위하여 Observer pattern을 적용하였다. Observer pattern은 자바 기본 라이브러리에서 가져왔으며, Elevator클래스는 이를 위해 기존에는 Thread를 상속하던것에서 Runnable 인터페이스를 구현하는것으로 바뀌었다.

```
7  public class QueueAlgorithm extends Observable implements Observer{
8      int emergencyDestination;
9      QueueNode elev1DestinationNode;
10     QueueNode elev2DestinationNode;
11     QueueNode elev1NDestinationNode;
12     QueueNode elev2NDestinationNode;
13     Queue selectQueue1, selectQueue2, aboardQueue;
14     Elevator elev1;
15     Elevator elev2;
16     Iterator itr1, itr2, itr3;
17     private final ReentrantLock criticObj=new ReentrantLock();
18
19     public QueueAlgorithm(){
20         elev1=new Elevator(1);
21         elev2=new Elevator(2);
22         elev1.addObserver(this);
23         elev2.addObserver(this);
24         Thread elev1Thread=new Thread(elev1);
25         Thread elev2Thread=new Thread(elev2);
26         elev1Thread.start();
27         elev2Thread.start();
28         selectQueue1= new Queue();
29         selectQueue2= new Queue();
30         aboardQueue= new Queue();
31         elev1DestinationNode=null;
32         elev2DestinationNode=null;
33         elev1NDestinationNode=null;
34         elev2NDestinationNode=null;
35     }
```

QueueAlgorithm 클래스에게는 옵저버 속성을 구현하고,

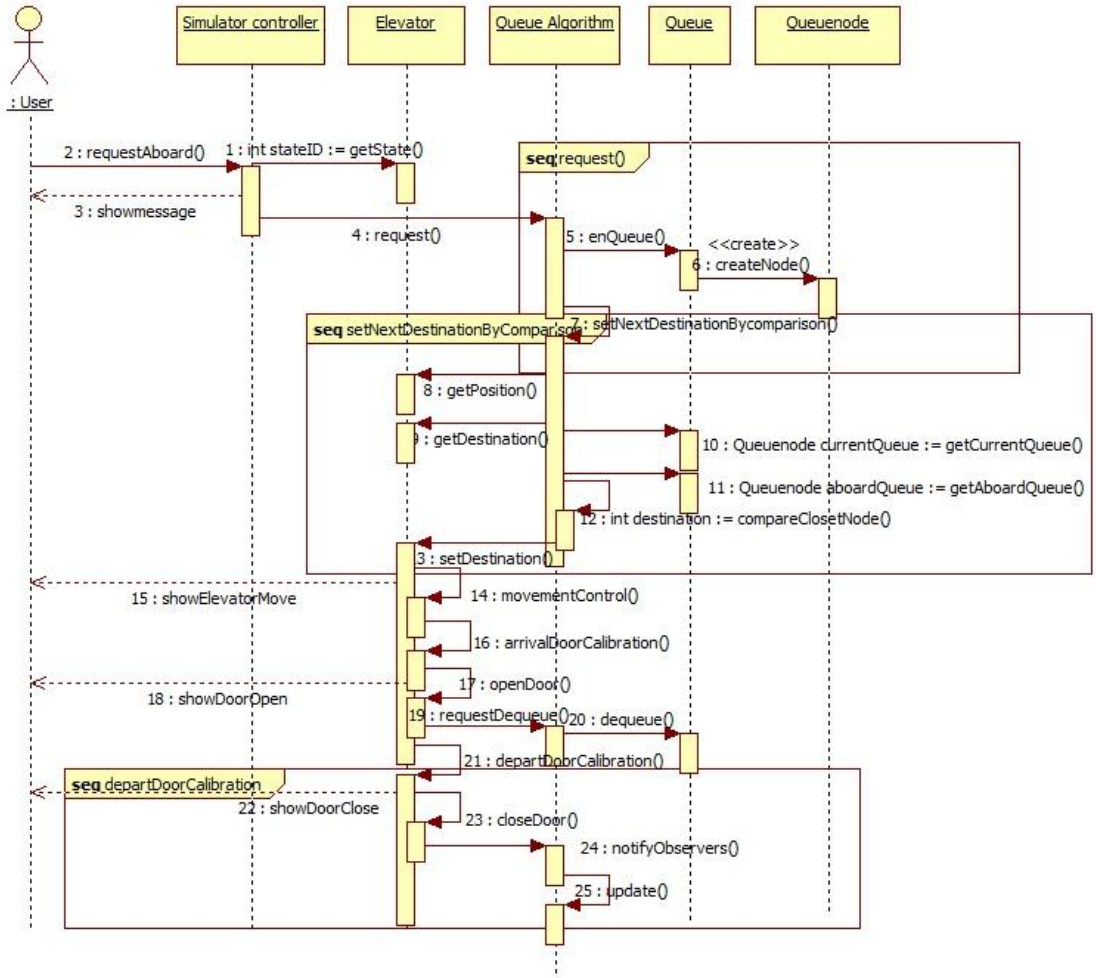
```
6  public class Elevator extends Observable implements Runnable {
7      final int elevatorID;
8      private int stoppedOrMoving;
9      private int doorState;
10     private int elevatorPosition;
11     private int elevatorDestination;
12     private int maxLoad;
13     private int currentLoad;
14     private int directionDelta;
15     private int currentState;
16     MovementControl job1, job2;
17     Timer jobScheduler1, jobScheduler2;
18     public Elevator(int elevatorID){
19         this.elevatorID=elevatorID;
20         stoppedOrMoving=0;
21         doorState=0;
22         elevatorPosition=1;
23         elevatorDestination=1;
24         maxLoad=1300;
25         directionDelta=0;
26         currentState=1;
27         currentLoad=0;
28     }
```

Elevator 클래스는 Observable을 상속, Runnable을 구현하도록 바꾸었다.

Observer pattern을 적용하면서 엘리베이터는 Queue와 QueueNode에 대한 접근이 원천적으로 금지되었고, QueueAlgorihtm에 대한 접근이 필요 없어졌음으로, Dependency문제가 해소되었다.

그리고 이로 인하여 기존의 시퀀스 다이어그램이 바뀌었다.
기존의 시퀀스 다이어그램에서 존재하던 addLoad부분은 QueueNode를 참조해야하기 때문에 Elevator클래스에서 삭제되어 QueueAlgorithm이 Observer로써 Elevator로부터 바뀜을 통보받으면 수행하도록 바뀌었다. 또한 기존의 시퀀스 다이어그램에서 엘리베이터가 QueueAlgorithm 클래스의 setNextByComparison()메소드를 요청하던 것 역시, Observer를 Notify시킴으로써 수행하도록 바꾸었다.

다음 페이지의 그림은 requestAboard()의 새로운 시퀀스 다이어그램이다.




```
public void departureCalibration() throws InterruptedException{
    closeDoor();
    setChanged();
    notifyObservers();
}
```

departureCalibration() 코드의 변화이다.

2. Code Pro 관련 : JavaDoc, Annotation관련 부분은 이미 프로그래밍이 완성단계이기 때문에, 중요도가 낮다고 판단하여, 추가하지 않았다.
   간단한 수정들 가령, Debug관련한 System.out.print류의 메소드, this, final 사용 등은 code pro의 지적대로 수정하였다.
   그러나, Synchronized는 아직 스레드에 대한 이해도가 부족하기 때문에, 어쩔 수없이 삭제하지 못하였다.
   여타 중요하지 않다고 생각되는 지적들 import의 알파벳순 정렬, String관련 부분등은 수정하지 않았다.

정리 : 정적분석에서 보고되었던 문제는 클래스에 대한 객체지향적인 프로그래밍이 되어있지 않았기 때문에 생겼던 문제였다. 그러므로 기존 시퀀스 다이어그램에서 파악한 순서대로 class가 생성되도록 조정하였으며 Dependency문제를 해결하기 위하여 Observer pattern을 구현, Elevator클래스가 다른 클래스를 직접 참조하는 것을 방지하였다. Code Pro에서 프로그래밍의 실행속도에 심각한 영향을 끼치는 부분은 없으므로, 간단히 수정할 수 있는 부분만 수정하였다.