Ingegneria del software
Elaborato

Salvatore Baglieri

1 Introduzione

Consideriamo un sistema industriale la cui configurazione è un'aggregazione ricorsiva di Configuration Items. La configurazione fisica ha una rappresentazione digitale realizzata da un insieme di Digital Twin, ciascuno connesso ad un componente fisico.

Gli elementi del sistema sono soggetti a fallimenti transienti (possono fallire e ripararsi), che vengono notificati ai rispettivi digital twins.

Il metodo evaluate() offerto dal sistema valuta lo stato attuale di quest'ultimo e, a seconda della tipologia di Failure, avvia una strategia di risoluzione.

2 Intento

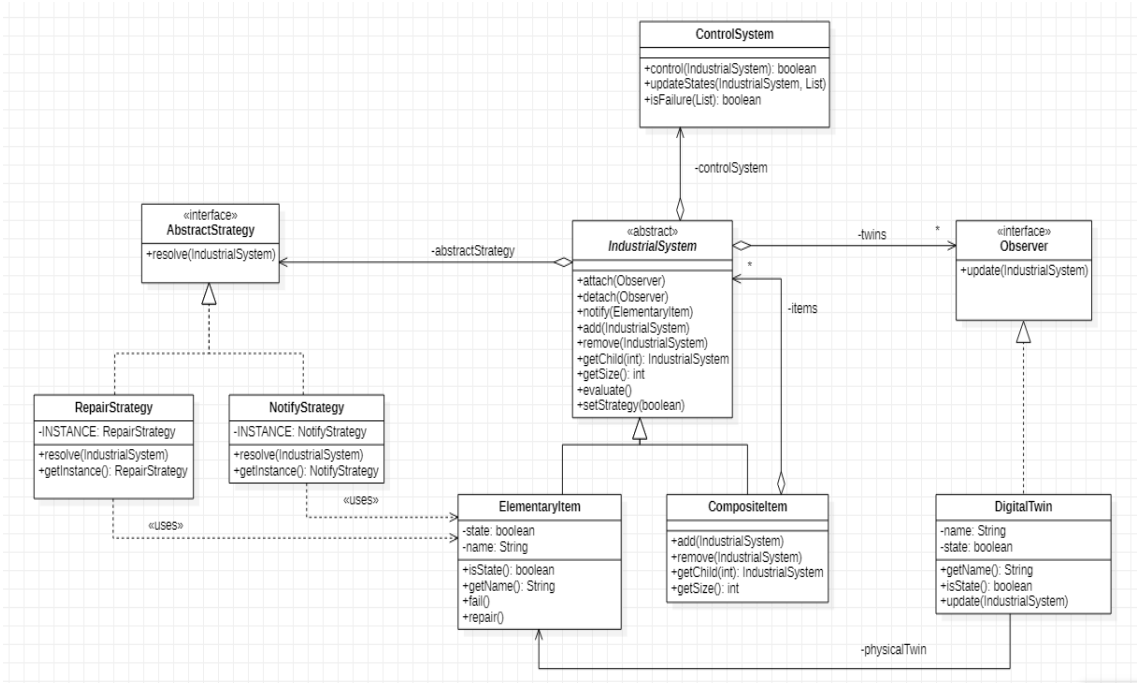
Simulare un sistema industriale, valutarne le condizioni in uno specifico istante di tempo, attraverso un metodo speciale offerto dal sistema, e avviare una strategia di risoluzione.

3 Implementazione

Per implementare quanto descritto sopra utilizzamo 3 design pattern:

- **Composite**: per la rappresentazione del sistema industriale, dove le Leaf sono Items elementari, e i Composite sono aggregazioni di Components;
- **Observer**: cosicché i digital twin ricevano notifica del cambiamento di stato del corrispondente componente fisico;
- **Strategy**: in modo tale da dare al sistema la possibilità di scegliere diverse strategie di risoluzione tra quelle a disposizione. Questi, non avendo stati, sono stati implementati come Singleton per risparmio di risorse, in quanto basta una sola istanza condivisa per ogni classe.

3.1 Class Diagram



3.2 Discussione delle classi principali

- **IndustrialSystem**
  E' definita come classe astratta. Funge inoltre da Observable per il design pattern Observer, da Component per il design pattern Composite e da Context per il design pattern Strategy. Questa infatti ospita i metodi attach(Observer), detach(Observer) e notify() per il patter Observer che è implementato in modo pull; i metodi add(IndustrialSystem), remove(IndustrialSystem) e getChild(int) per il pattern Composite. Il metodo evaluate() ha lo scopo di valutare le condizioni del sistema, utilizzando il metodo control(IndustrialSystem) della classe ControlSystem; in base al valore ritornato da questa setta lo strategy, attraverso il metodo setStrategy(boolean), e chiama il metodo resolve().

- **ElementaryItem**
  Questo ha uno stato che può essere funzionante (true) o guasto (false). Ha la responsabilità di creare il proprio digital twin e fare in modo che quest'ultimo si colleghi ad esso. Espone infine i metodi fail() e repair() che cambiano lo stato e notificano il cambiamento al digital twin corrispondente.

- **RepairStrategy**
  Implementata come Singleton, quindi esiste una sola istanza di essa che può essere richiesta tramite il metodo statico getInstance(). Il metodo resolve(IndustrialSystem), invece, ha lo scopo di individuare i componenti guasti del sistema e ripararli.

- **NotifyStrategy**
  Implementata come Singleton anche questa; ma, a differenza del RepairStrategy, questa non ha lo scopo di riparare i componenti guasti, bensì di fornire un elenco di questi.

3.3 Codice e Testing

3.3.1 IndustrialSystem

```
public abstract class IndustrialSystem {

    private List<Observer> twins = new ArrayList<>();
    private ControlSystem controlSystem = new ControlSystem();
    private AbstractStrategy abstractStrategy;

    public void attach(Observer obs) { twins.add(obs); }

    public void detach(Observer obs) { twins.remove(obs); }

    public void notify(ElementaryItem ei){
        for(Observer obs:twins){
            obs.update(ei);
        }
    }

    public void add(IndustrialSystem is) throws IllegalAccessError{
        throw new IllegalAccessError( s: "You cannot access this method for the specified object");
    }

    public void remove(IndustrialSystem is) throws IllegalAccessError{
        throw new IllegalAccessError( s: "You cannot access this method for the specified object");
    }
      public IndustrialSystem getChild(int i) throws IllegalAccessError{
          throw new IllegalAccessError( s: "You cannot access this method for the specified object");
      }

      public int getSize() throws IllegalAccessError{
          throw new IllegalAccessError( s: "You cannot access this method for the specified object");
      }
```

```
    public void evaluate(){
        boolean systemState = controlSystem.control( industrialSystem: this);
        if(!systemState){
            System.out.println("Numero di componenti guasti superiore al 40% \nAvvio startegia di riparazione...");
        }else System.out.println("Numero di componenti guasti inferiore al 40% \nAvvio startegia di notifica..." +
                "\nElenco dei componenti guasti:");
        setStrategy(systemState);
        abstractStrategy.resolve( industrialSystem: this);
    }

    public void setStrategy(boolean state){
        if(!state){
            abstractStrategy = RepairStrategy.getInstance();
        }else abstractStrategy = NotifyStrategy.getInstance();
    }



}
```

3.3.2 ElementaryItem

```
public class ElementaryItem extends IndustrialSystem {

    private boolean state;
    private String name;

    public ElementaryItem(String name){
        this.name = name;
        this.state = true;
        new DigitalTwin(name,  state: true, ei: this);
    }

    public boolean isState() { return state; }

    public String getName() { return name; }

    public void fail(){
        state = false;
        System.out.println("Il componente "+getName()+" adesso è guasto!");
        notify( ei: this);
    }

    public void repair(){
        state = true;
        System.out.println("Il componente "+getName()+" è stato ripatato!");
        notify( ei: this);
    }

}
```

3.3.3 CompositeItem

```
public class CompositeItem extends IndustrialSystem {

    private List<IndustrialSystem> items;

    public CompositeItem(){
        items = new ArrayList<>();
    }


    @Override
    public void add(IndustrialSystem is) throws IllegalAccessError {
        items.add(is);
    }

    @Override
    public void remove(IndustrialSystem is) throws IllegalAccessError {
        items.remove(is);

    }

    @Override
    public IndustrialSystem getChild(int i) throws IllegalAccessError {
        return items.get(i);
    }

    @Override
    public int getSize() throws IllegalAccessError {
        return items.size();
    }
```

3.3.4 Observer

```
public interface Observer {

    void update(IndustrialSystem is);

}
```

3.3.5 DigitalTwin

```
public class DigitalTwin implements Observer {

    private String name;
    private boolean state;
    private ElementaryItem physicalTwin;

    public DigitalTwin(String name, boolean state, ElementaryItem ei){
        this.name = name+"Twin";
        this.state = state;
        physicalTwin = ei;
        ei.attach( obs: this);
    }

    public String getName() { return name; }

    public boolean isState() { return state; }

    @Override
    public void update(IndustrialSystem is) {
        if(is==physicalTwin){
            System.out.println(name+" ha ricevuto notifica!");
            this.state = physicalTwin.isState();
        }
    }

}
```

3.4 ControlSystem

```
public class ControlSystem {

    public ControlSystem(){}

    public boolean control(IndustrialSystem industrialSystem){
        List<Boolean> status = new ArrayList<>();
        updateStatus(industrialSystem,status);
        return isFailure(status);
    }

    public void updateStatus(IndustrialSystem industrialSystem,List<Boolean> status){
        for(int i=0; i<industrialSystem.getSize(); i++) {
            if (industrialSystem.getChild(i) instanceof ElementaryItem){
                ElementaryItem ei = (ElementaryItem)industrialSystem.getChild(i);
                status.add(ei.isState());
            }else updateStatus(industrialSystem.getChild(i),status);
        }
    }
```

```
public boolean isFailure(List<Boolean> status){
    int failure = 0;
    for (Boolean aBoolean : status) {
        if (!aBoolean) {
            failure++;
        }
    }
    if(failure==0){
        return true;
    }
    return ((failure * 100) / status.size()) < 40;
}
```

3.4.1 AbstractStrategy

```
public interface AbstractStrategy {

    void resolve(IndustrialSystem industrialSystem);

}
```

3.4.2 RepairStrategy

```
public class RepairStrategy implements AbstractStrategy {

    private static RepairStrategy INSTANCE = null;
    private RepairStrategy(){}

    public static RepairStrategy getInstance(){
        if(INSTANCE==null){
            INSTANCE = new RepairStrategy();
        }
        return INSTANCE;
    }

    @Override
    public void resolve(IndustrialSystem industrialSystem) {
        for(int i=0; i<industrialSystem.getSize(); i++) {
            if (industrialSystem.getChild(i) instanceof ElementaryItem){
                ElementaryItem ei = (ElementaryItem)industrialSystem.getChild(i);
                if(!ei.isState()){
                    System.out.println("Riparo componente "+ei.getName());
                    ei.repair();
                }
            }else resolve(industrialSystem.getChild(i));
        }
    }
```

3.4.3 NotifyStrategy

```
public class NotifyStrategy implements AbstractStrategy {

    private static NotifyStrategy INSTANCE = null;
    private NotifyStrategy(){}

    public static NotifyStrategy getInstance(){
        if(INSTANCE==null){
            INSTANCE = new NotifyStrategy();
        }
        return INSTANCE;
    }

    @Override
    public void resolve(IndustrialSystem industrialSystem) {
        for(int i=0; i<industrialSystem.getSize(); i++) {
            if (industrialSystem.getChild(i) instanceof ElementaryItem){
                ElementaryItem ei = (ElementaryItem)industrialSystem.getChild(i);
                if(!ei.isState()){
                    System.out.println("-"+ei.getName());
                }
            }else resolve(industrialSystem.getChild(i));
        }
    }

}
```

3.4.4 Testing

Di seguito sono riportati i test eseguiti, sia nella forma di Main, sia nella forma di Unit Test, utilizzando JUnit.

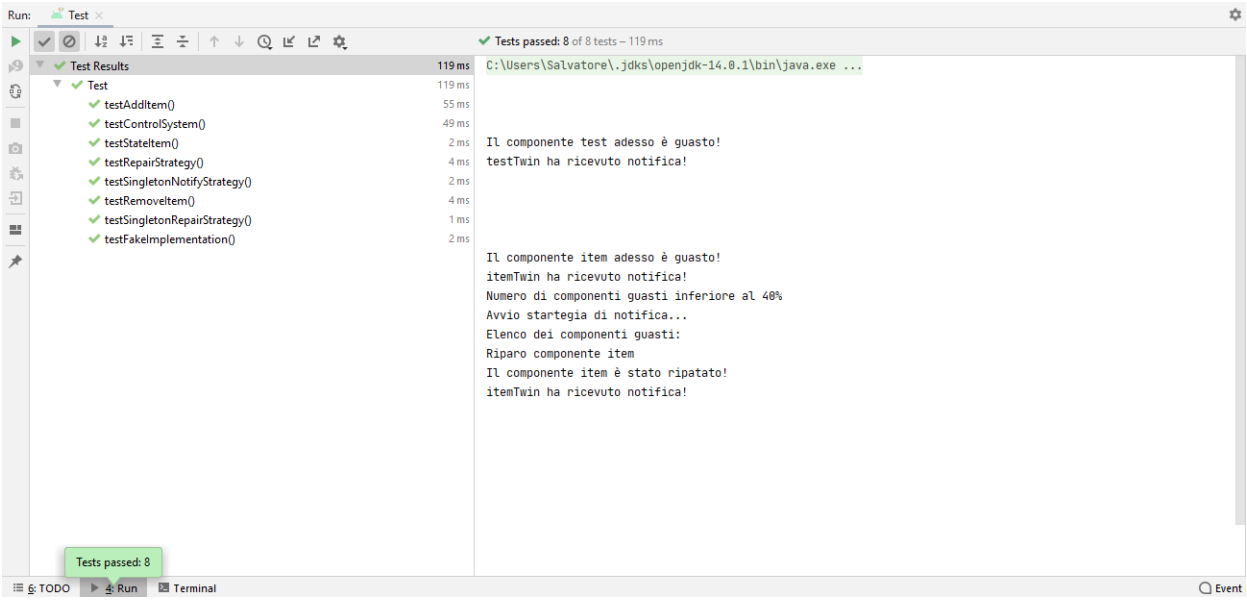
Main

```
public class Main {

    public static void main(String[] args) {

        ElementaryItem item1 = new ElementaryItem( name: "ItemType1");
        ElementaryItem item2 = new ElementaryItem( name: "ItemType2");
        ElementaryItem item3 = new ElementaryItem( name: "ItemType3");
        ElementaryItem item4 = new ElementaryItem( name: "ItemType4");

        CompositeItem compositeItem1 = new CompositeItem();
        CompositeItem compositeItem2 = new CompositeItem();
        CompositeItem compositeItem3 = new CompositeItem();

        compositeItem1.add(item1);
        compositeItem1.add(item2);
        compositeItem2.add(item3);
        compositeItem3.add(compositeItem2);

        IndustrialSystem industrialSystem = new CompositeItem();
        industrialSystem.add(compositeItem1);
        industrialSystem.add(item4);
        industrialSystem.add(compositeItem3);

        System.out.println("\n!!TEST REPAIR STRATEGY!!\n");
        item1.fail();
        item3.fail();
        industrialSystem.evaluate();


        System.out.println("\n!!TEST NOTIFY STRATEGY!!\n");
        item2.fail();
        industrialSystem.evaluate();

    }
}
```

Unit Test

```
import static org.junit.jupiter.api.Assertions.*;

class Test {

    @org.junit.jupiter.api.Test
    void testFakeImplementation(){
        IndustrialSystem isTest = new ElementaryItem( name: "test");
        assertThrows(IllegalAccessError.class, ()->isTest.getChild(5));
        assertThrows(IllegalAccessError.class, ()->isTest.add(isTest));
        assertThrows(IllegalAccessError.class, ()->isTest.remove(isTest));
    }

    @org.junit.jupiter.api.Test
    void testSingletonRepairStrategy(){
        AbstractStrategy instance1 = RepairStrategy.getInstance();
        AbstractStrategy instance2 = RepairStrategy.getInstance();
        assertEquals(instance1,instance2);
    }
```

```java
@org.junit.jupiter.api.Test
void testSingletonNotifyStrategy(){
    AbstractStrategy instance1 = NotifyStrategy.getInstance();
    AbstractStrategy instance2 = NotifyStrategy.getInstance();
    assertEquals(instance1,instance2);
}

@org.junit.jupiter.api.Test
void testRepairStrategy(){
    ElementaryItem item = new ElementaryItem( name: "item");
    IndustrialSystem is = new CompositeItem();
    is.add(item);
    item.fail();
    is.evaluate();
    assertTrue(item.isState());
}

@org.junit.jupiter.api.Test
void testRemoveItem(){
    ElementaryItem isTest = new ElementaryItem( name: "test");
    IndustrialSystem is = new CompositeItem();
    is.add(isTest);
    is.remove(isTest);
    assertThrows(IndexOutOfBoundsException.class, ()->is.getChild(0));
}

@org.junit.jupiter.api.Test
void testAddItem(){
    ElementaryItem isTest = new ElementaryItem( name: "test");
    IndustrialSystem is = new CompositeItem();
    is.add(isTest);
    assertEquals(isTest,is.getChild(0));
}

@org.junit.jupiter.api.Test
void testStateItem(){
    ElementaryItem isTest = new ElementaryItem( name: "test");
    assertTrue(isTest.isState());
}

@org.junit.jupiter.api.Test
void testControlSystem(){
    ElementaryItem isTest = new ElementaryItem( name: "test");
    IndustrialSystem is = new CompositeItem();
    is.add(isTest);
    ControlSystem controlSystem = new ControlSystem();
    assertTrue(controlSystem.control(is));
    isTest.fail();
    assertFalse(controlSystem.control(is));
}
```


Run:
Test
Tests passed: 8 of 8 tests – 119 ms
Test Results 119 ms
Test 119 ms
testAddItem() 55 ms
testControlSystem() 49 ms
testStateItem() 2 ms
testRepairStrategy() 4 ms
testSingletonNotifyStrategy() 2 ms
testRemoveItem() 4 ms
testSingletonRepairStrategy() 1 ms
testFakeImplementation() 2 ms
C:\Users\Salvatore\.jdks\openjdk-14.0.1\bin\java.exe ...
Il componente test adesso è guasto!
testTwin ha ricevuto notifica!
Il componente item adesso è guasto!
itemTwin ha ricevuto notifica!
Numero di componenti guasti inferiore al 40%
Avvio startegia di notifica...
Elenco dei componenti guasti:
Riparo componente item
Il componente item è stato ripatato!
itemTwin ha ricevuto notifica!
Tests passed: 8
6: TODO
4: Run
Terminal
Event

3.5 Sequence Diagram

Il seguente Sequence Diagram mostra il semplice caso in cui il client valuta lo stato del sistema, composto da un singolo item elementare, utilizzando il metodo evaluate().