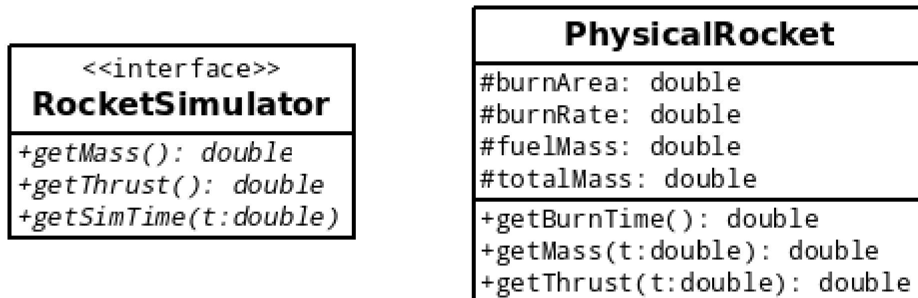
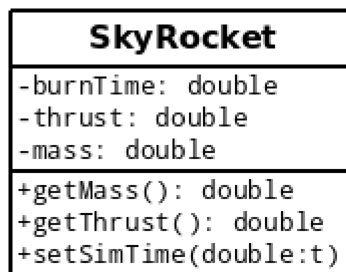
ESERCIZI DESIGN PATTERN

ADAPTER

1) Dato le seguenti classi e interfacce, completarlo con una classe OozinozRocket che faccia da Adapter tra l'interfaccia RocketSimulator e la classe PhysicalRocket.

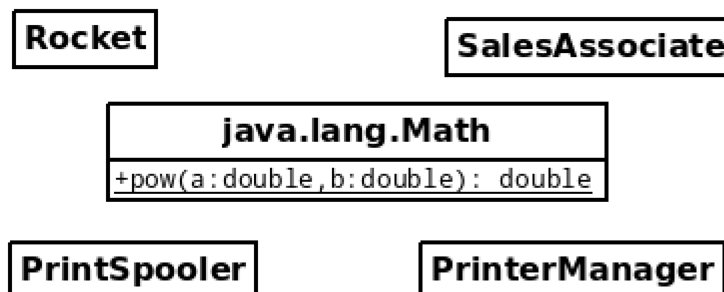


2) Scrivere il codice Java della classe OozinozRocket
3) Immaginare che si voglia usare SkyRocket al posto di RocketSimulator, la quale classe è la seguente, fare il diagramma delle classi creando un Adapter che sostituisca RocketSimulator.



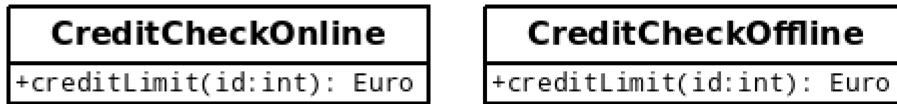
SINGLETON

1) Dato il seguente schema di classi, dire quali classi potrebbero applicare il pattern Singleton

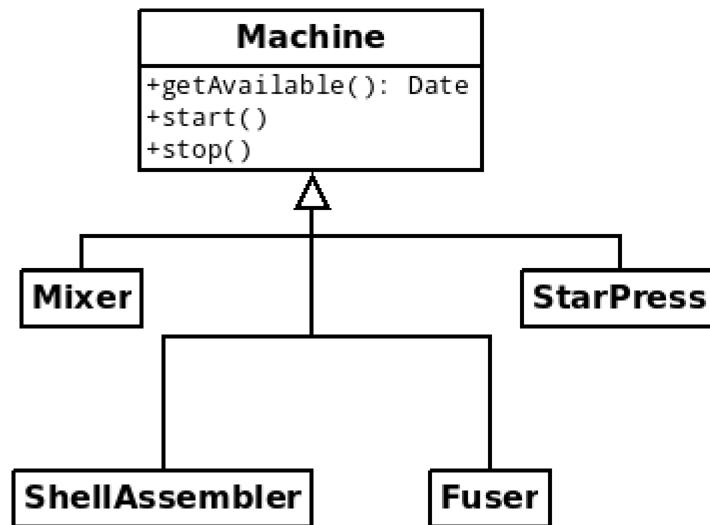


FACTORY

1) Dato il seguente schema di classi produrre una classe Factory unica per la creazione dei corrispondenti oggetti CreditCheck.



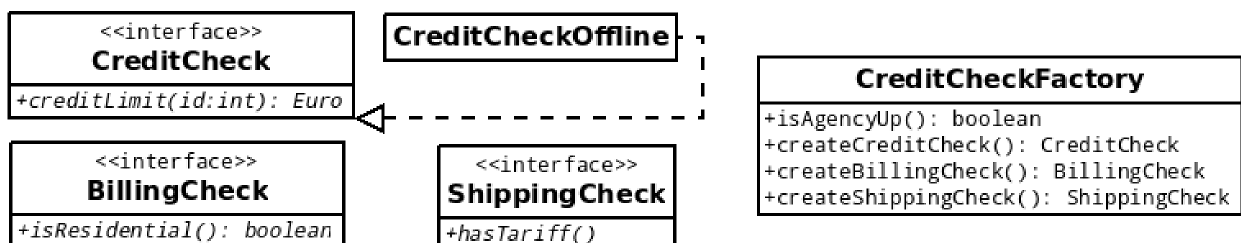
2) Modificare la classe CreditCheckFactory precedente aggiungendo il metodo isAgencyUp() che dice se si può controllare il credito online, scrivere il codice della classe.
3) Dato il seguente schema di classi, vogliamo creare una classe MachinePlanner che crea un planner per ogni tipo di Machine.



4) In base allo schema precedentemente creato, scrivere il codice delle classi StarPress e StarPressPlanner.

ABSTRACT FACTORY

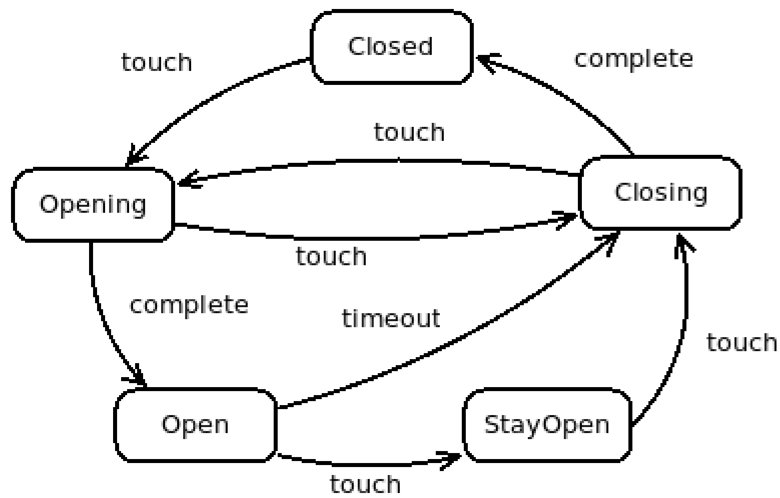
1) Abbiamo una serie di classi e interfacce generali per il controllo del credito, noi però vogliamo sfruttare queste classi in varie implementazioni, per esempio per l'Italia. Creare uno schema delle classi che mostra questa specifica implementazione.



2) Scrivere il codice della classe CreditCheckFactoryItaly dello schema risultante dall'esercizio precedente

STATE

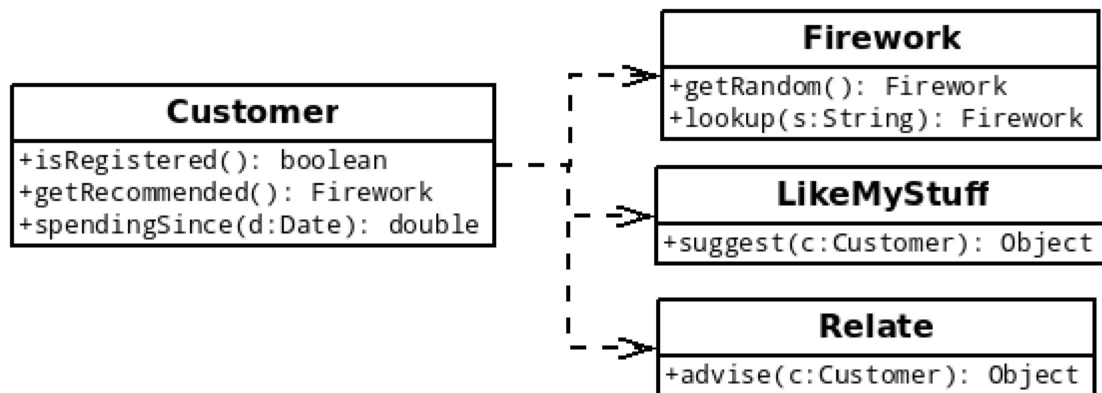
1) Dato il seguente grafo degli stati di una Porta, scrivere lo schema delle classi per gli stati



2) Basandosi sullo schema precedentemente ottenuto scrivere il codice delle classi Door, DoorState e Closing

STRATEGY

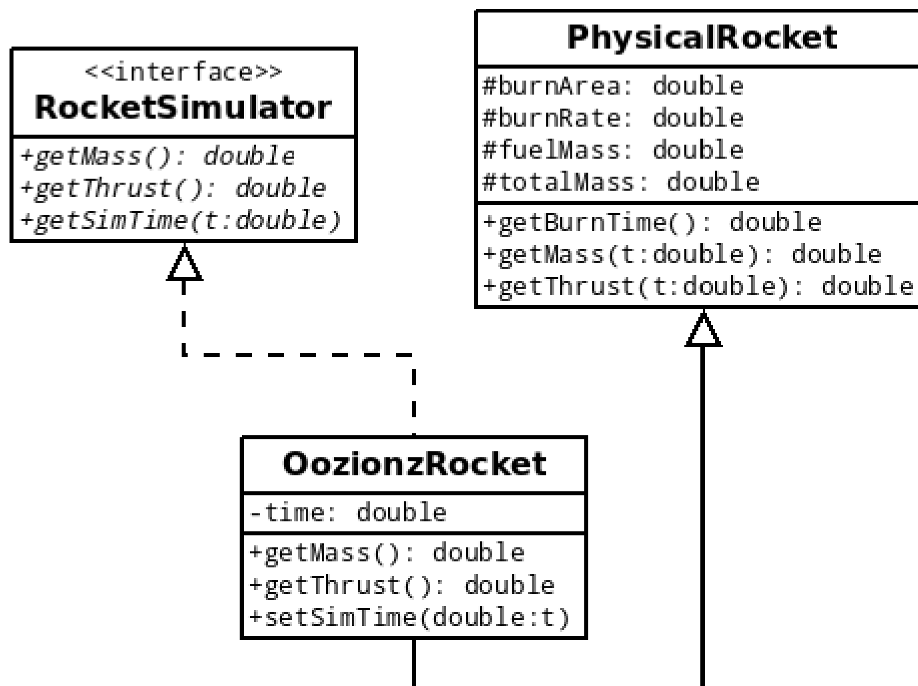
1) Date le due seguenti classi, applicare il Pattern Strategy modificando il diagramma delle classi nel modo opportuno.



SOLUZIONI

ADAPTER

1) La classe OozionzRocket avrà gli stessi metodi dell'interfaccia e userà quei metodi per richiamare quelli di PhysicalRocket.



2)

```
public class OozinozRocket extends PhysicalRocket implements RocketSimulator {

    private double time;

    public OozinozRocket(double ba, double br, double fm, double tm){
        super(ba, br, fm, tm);
    }

    public double getMass(){
        return totalMass;
    }

    public double getThrust(){
        return getThrust(time);
    }

    public double setSimTime(double t){
        time = t;
    }
}
```

3)

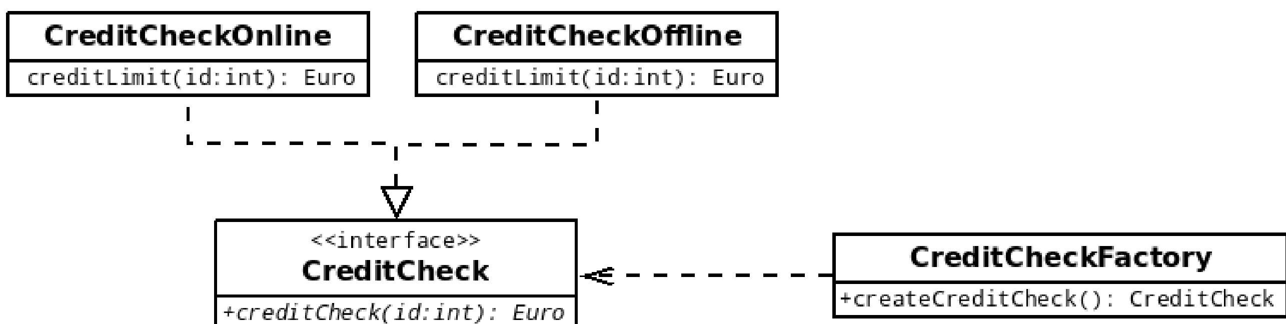
```java
public class SimulatorAdapter {

    private SkyRocket s;

    public SimulatorAdapter(double m, double t, double bt){
        s = new SkyRocket(m, t, bt);
    }

    public double getMass(){
        return s.getMass();
    }

    public double getThrust(){
        return s.getThrust();
    }

    public double getSimTime(double t){
        return s.getSimTime(t);
    }

}
```

SINGLETON

1) *Rocket*: non può essere un Singleton, è più probabile che sia una normale e comune classe.
   SalesAssociate: come per Rocket.
   java.lang.Math: ha metodi statici, dunque non può essere un Singleton.
   PrintSpooler: lo spooler di stampante è in ogni stampante, dunque non può essere un Singleton.
   PrinterManager: un gestore di tutte le stampanti è decisamente un singleton. Una sola classe per gestire tutte le stampanti di un ufficio, per esempio.

FACTORY

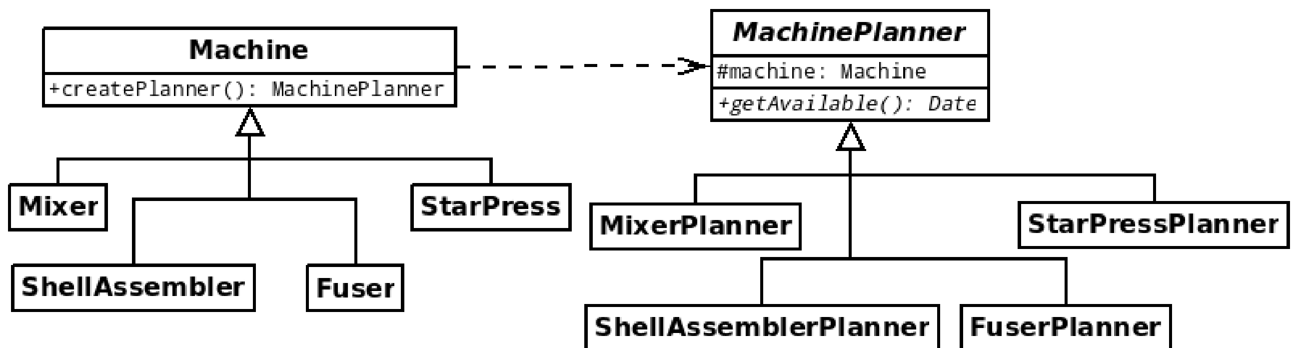
1) Per creare la classe Factory mi basta innanzitutto far implementare alle due classi un'interfaccia comune, ovvero CreditCheck che avrà il metodo creditLimit(). A questo punto creerò una classe CreditCheckFactory che creerà attraverso un proprio opportuno metodo uno dei due oggetti.



2)

```java
public class CreditCheckFactory{

    public CreditCheck createCreditCheck(){
        boolean flag = this.isAgencyUp();
        if(flag==true) return new CreditCheckOnline();
        else           return new CreditCheckOffline();
    }

    public boolean isAgencyUp(){
        //Metodo che controlla se la connessione è ok.
    }
}
```

3) 4) Per poter creare un Planner per ogni tipo di Machine devo dunque fare un parallelismo, ovvero per ogni Planner associo una Machine, e ad ogni Machine associo un Planner.
Per migliorare ciò posso rendere Machine il Factory e i vari Planner le classi create.

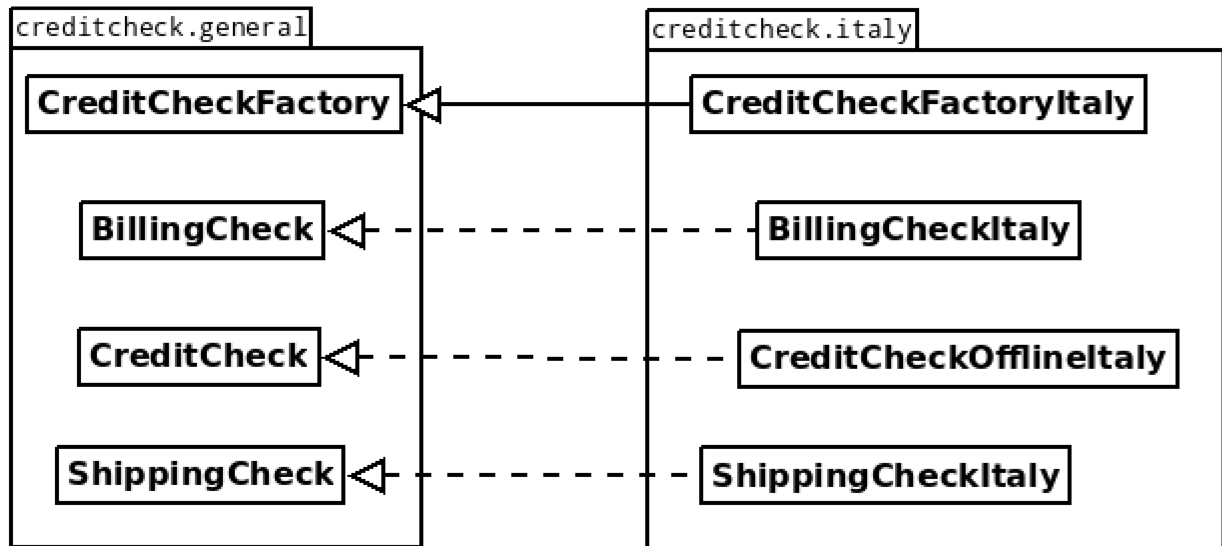



```java
public class StarPress extends Machine {

    public StarPress(){
        super();
    }

    @Override
    public MachinePlanner createPlanner(){
        return new StarPressPlanner(this);
    }
}
```

```java
public class StarPressPlanner extends MachinePlanner{

    public StarPressPlanner(Machine machine){
        super(machine);
    }

    @Override
    public Date getAvailable(){
        //Codice personale della classe
    }
}
```

ABSTRACT FACTORY

1)

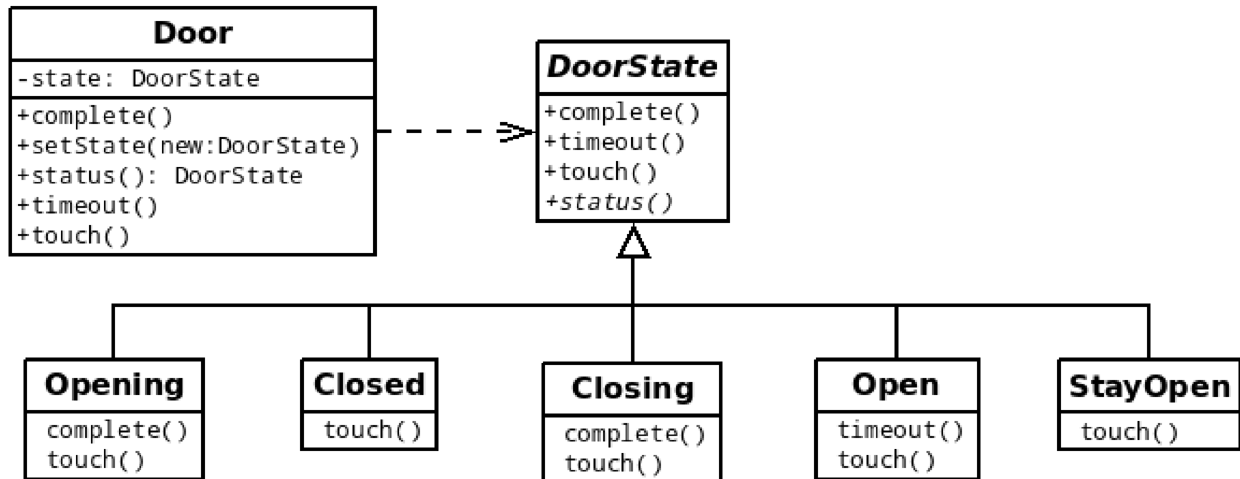


2)

```java
public class CreditCheckFactoryItaly extends CreditCheckFactory{

    public boolean isAgencyUp(){
        //Metodo che controlla se la connessione è attiva
    }

    public CreditCheck createCreditCheck() {
        if(this.isAgencyUp()) return new CreditCheckOffline();
        else new CreditCheckOnlineItaly();
    }

    public BillingCheck createBillingCheck() {
        return new BillingCheckItaly();
    }

    public ShippingCheck createShippingCheck() {
        return new ShippingCheckItaly();
    }

}
```

STATE

1) Creiamo una classe DoorState che rifletta i metodi di Door, ma non solo, questi metodi non devono essere abstract. Essi saranno poi usati dai vari stati re-implementandoli a loro piacimento.



2)

```
public class Door{

    private DoorState state = Closed.status();
    public void complete(){ status.complete(); }
    public void touch(){ status.touch(); }
    public void timeout(){ status.timeout(); }
    public DoorState status(){ return state; }
    public void setState(DoorState newState){ state = newState;}

}

public abstract class DoorState{

    protected Door door;

    public DoorState(Door d){ door=d; }

    public void complete(){ }
    public void touch(){ }
    public void timeout(){ }
    public static abstract DoorState status();

}

public class Closing extends DoorState {

    public Closing(Door d){ super(d); }

    public void complete(){ d.setState(Closed.status()); }
    public void touch(){ d.setState(Touch.status()); }

    public static DoorState status(){ return new Closing(d); }

}
```

STRATEGY

1) Innanzitutto dobbiamo dividere l'intero schema in 3 distinti gruppi di idee:
   - i consigli
   - i clienti
   - i fuochi d'artificio.

   I consigli non sono altro che “la strategia” da applicare al nostro cliente su quale fuoco d'artificio dargli.