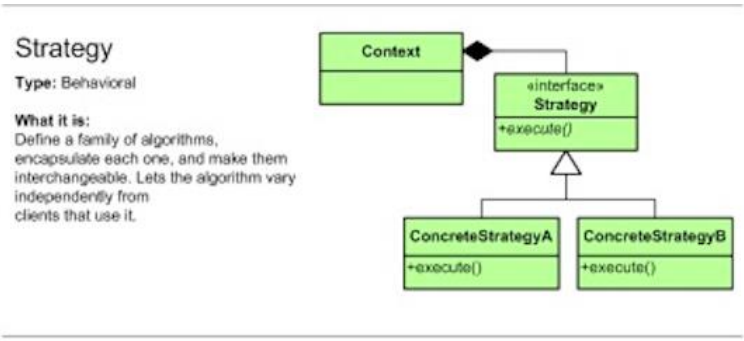
Exercicis de Patrons de Disseny

1. Es tenen diferents maneres de fer una ordenació: el mètode de la bombolla, el mètode d'inserció, el mètode de QuickSort i el mètode de MergeSort. Es vol modelar una classe que permeti ordenar un vector d'enters segons el mètode anomenat arrange. Per a dissenyar aquest problema, quin patró podries fer servir? Com l'aplicaries? Escriu el diagrama de classes que dissenyaries sota aquest patró, identificant les diferents classes del patró amb el teu problema concret. Dona l'exemple d'un programa principal per a veure com utilitzaria el teu disseny. Analitza si vulneres algun principi de disseny amb la teva solució i explica per què.

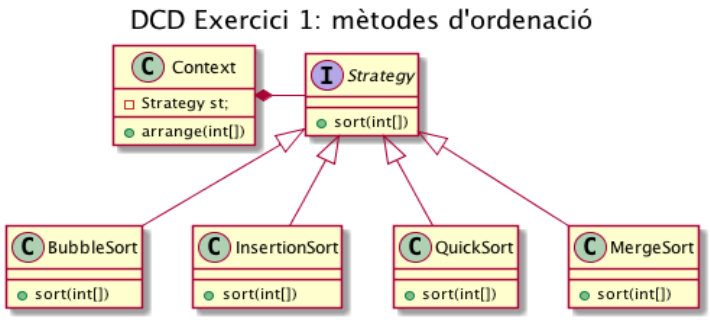
1. Problema identificat a solucionar:

El fet de tenir diferents estratègies d'ordenació que han de ser implementades des del mètode arrange() porta a un problema de modelar aquests diferents comportaments en una interfície on els diferents mètodes d'ordenació siguin les diferents implementacions.

2. Patró a aplicar: STRATEGY



3. Aplicació del patró:



4. Anàlisi del patró aplicat. (si s'escau)

En aquest cas, la classe Context té el mètode arrange() que, segons el seu atribut de tipus Strategy, cridarà a un mètode o un altre. La forma d'inicialitzar l'atribut st de la classe Context, es mitjançant injecció de dependències en el seu constructor. Així s'evita la vulneració del principi Dependency Inversion Principle, tal i com es mostra en el llistat de sota.

```
public class Context {
        private final Strategy strategy;

        public Context(Strategy strategy) {
          this.strategy = strategy;
        }

        public void arrange(int[] input) {
           strategy.sort(input);
        }
}
```

Serà el Client del Context qui farà la tria de quin mètode cal utilitzar i en tot cas, serà ell qui vulnerarà el principi Open-Closed Principle.

5. Programa principal que mostra l'ús del patró utilitzat.

```
public class Test {
  public static void main(String args[]) {
        // we can provide any strategy to do the sorting
        int[] var = {1, 2, 3, 4, 5 };
        Context ctx = new Context(new BubbleSort());
        ctx.arrange(var)
        // we can change the strategy without changing Context class
        ctx = new Context(new QuickSort());
        ctx.arrange(var);
  }
}
```

Aquí només se'n dóna un exemple d'ús sense tenir en compte tots els casos.

6. Observacions addicionals

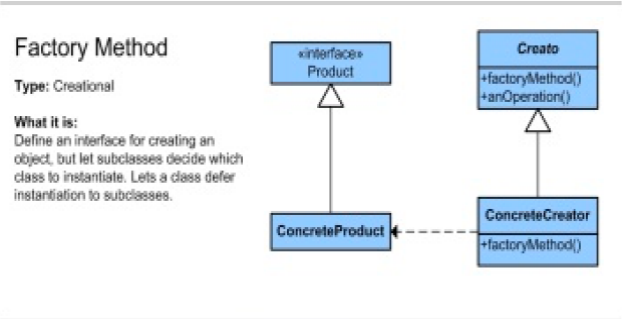
Si es tingués el Client que contempla tots els casos:

```
public class Client {
    public void order (SortType type, int[] var) {
      Context ctx;
     // SortType is an enum {BubbleSortType, InsertionSortType, QuicksortType,
MergeSortType}
      switch (type) {
       case BubbleSortType:
          ctx = new Context(new BubbleSort());
          break;
       case InsertionSortType:
          ctx = new Context(new InsertionSort());
          break;
       case QuickSortType:
          ctx = new Context(new QuickSort());
          break;
       case MergeSortType:
          ctx = new Context(new MergeSort());
          break;
       default:
          break;
      }
      ctx.arrange(var);
}
}
```

Es tindria el problema de vulnerabilitat de l'Open-Closed Principle, ja que si es volguessin afegir més mètodes, aquí caldria canviar el codi.

Cal observar que s'està davant d'un problema llavors de construcció de les classes i podríem aplicar el patró de FactoryMethod per a solucionar-ho. De fet el Product és la classe Strategy i

els concreteProduct seran les diferents classes de les ordenacions. Per a acabar d'aplicar bé el patró de FactoryMethod caldria definir una classe abstracte addicional (Factory) i la seva filla StrategyFactory.

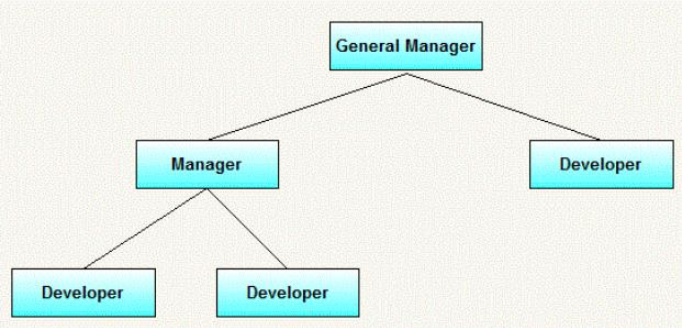



```
public abstract class Factory {
   public void order (SortType type, int[] var) {
            Context ctx;
            Strategy st  = createStrategy(type);
            ctx.arrange(var);
            }
            public abstract Strategy createSetrategy(SortType type);
}
```

```
public class StrategyFactory implements Creator {
            public Strategy createStrategy(SortType type) {
             switch (type) {
            case BubbleSortType:
            ctx = new Context(new BubbleSort());
             break;
          case InsertionSortType:
             ctx = new Context(new InsertionSort());
             break;
          case QuickSortType:
             ctx = new Context(new QuickSort());
             break;
          case MergeSortType:
             ctx = new Context(new MergeSort());
```

```
            break;
        default:
            break;
      }
    }
}
```

2. Es vol modelar un software que permeti definir una organització on hi ha managers que tenen sota les seves ordres empleats. Els managers poden estar a les ordres d'altres managers. Per a dissenyar aquest problema, quin patró podries fer servir? Com l'aplicaries? Escriu el diagrama de classes que dissenyaries sota aquest patró, identificant les diferents classes del patró amb el teu problema concret. Dona l'exemple d'un programa principal que construeixi l'estructura general següent:

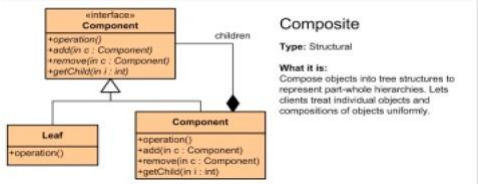


Analitza si vulneres algun principi de disseny amb la teva solució i explica per què.

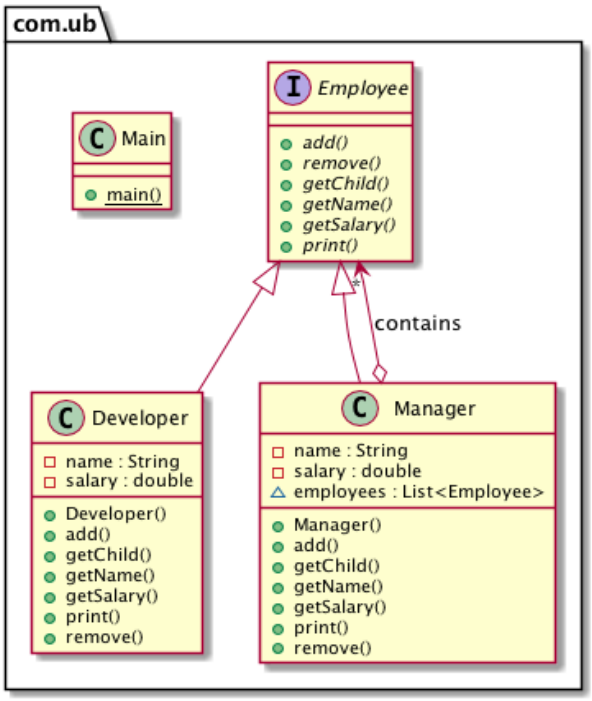
1. Problema identificat a solucionar:

El problema consisteix en estructurar informació jeràrquica de Manager i Developer de forma recursiva. Hi han classes compostes (General Manager) formades per classes Simples (Developer) i compostes (Manager). És clarament un problema d'estructura.

2. Patró a aplicar: Composite



3. Aplicació del patró:



4. Anàlisi del patró aplicat. (si s'escau) El codi generat aplicant el patró Composite compleix els principis SOLID, exepte el principi Liskov, on Developer no implemente els mètodes add, remove i getChild. Aquest és un problema inherent al patró Composite.

Employee.java (Component)

```
public interface Employee {
   public void add (Employee employee);
   public void remove (Employee employee);
   public Employee getChild (int i);
   public String getName();
   public double getSalary();
   public void print();
}
```

Manager.java (Composite)

```
public class Manager implements Employee{
   private String name;
   private double salary;
   List<Employee> employees = new ArrayList<Employee>();
   public Manager(String name,double salary){
      this.name = name;
      this.salary = salary;
   }
   public void add(Employee employee) { employees.add(employee); }
   public Employee getChild(int i) {return employees.get(i); }
   public String getName() {return name;}
   public double getSalary() {  return salary;}
```

```
public void print() {
      System.out.println("----------");
      System.out.println("Name ="+getName());
      System.out.println("Salary ="+getSalary());
      System.out.println("----------");
      Iterator<Employee> employeeIterator = employees.iterator();
      while(employeeIterator.hasNext()){
         Employee employee = employeeIterator.next();
         employee.print();
      }
   }
   public void remove(Employee employee) {
      employees.remove(employee);
   }
}
```

Developer.java (leaf)

```
public class Developer implements Employee{
   private String name;
   private double salary;
   public Developer(String name,double salary){
      this.name = name;
      this.salary = salary;
   }
   public void add(Employee employee) {
      //this is leaf node so this method is not applicable to this class.
   }
   public Employee getChild(int i) {
      //this is leaf node so this method is not applicable to this class.
      return null;
   }
```

```
public String getName() { return name;}
   public double getSalary() {
      return salary;
   }
   public void print() {
      System.out.println("----------");
      System.out.println("Name ="+getName());
      System.out.println("Salary ="+getSalary());
      System.out.println("----------");
   }
   public void remove(Employee employee) {
      //this is leaf node so this method is not applicable to this class.
   }
}
```

5. Programa principal que mostra l'ús del patró utilitzat.

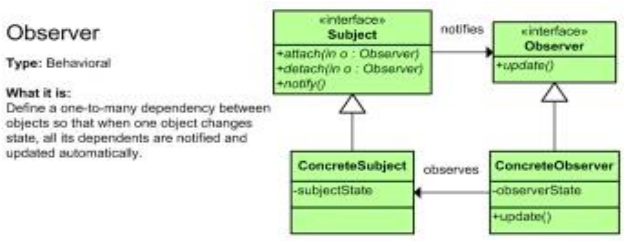
```
public static void main(String[] args) {
   Employee emp1=new Developer("John", 10000);
   Employee emp2=new Developer("David", 15000);
   Employee manager1=new Manager("Daniel-Manager",25000);
   manager1.add(emp1);
   manager1.add(emp2);
   Employee emp3=new Developer("Michael", 20000);
   Manager generalManager=new Manager("Mark-General Manager", 50000);
   generalManager.add(emp3);
   generalManager.add(manager1);
   generalManager.print();
}
```

3. Es té un sistema de préstec que defineix un tipus d'interès, Quan canvia el tipus d'interès, el sistema de préstec ho notifica a un diari o per internet per a visualitzar el nou tipus interès dels préstecs. puja el tipus d'interès. Per a dissenyar aquest problema, quin patró podries fer servir? Com l'aplicaries? Escriu el diagrama de classes que dissenyaries sota aquest patró, identificant les diferents classes del patró amb el teu problema concret. Dóna l'exemple d'un programa principal per a veure com utilitzaria el teu disseny. Analitza si vulneres algun principi de disseny amb la teva solució i explica per què.

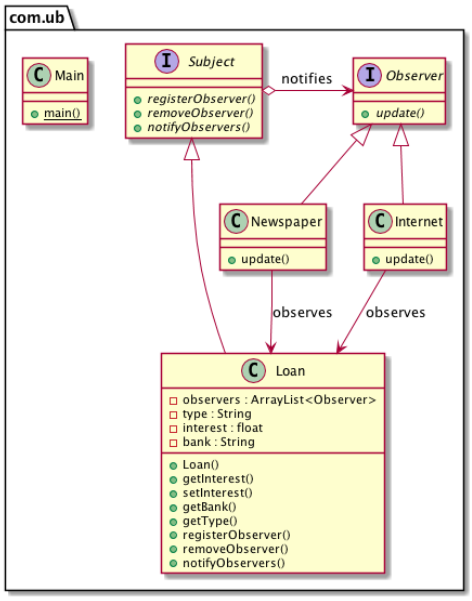
1. Problema identificat a solucionar:

En aquest problema quan s'actualitza el nou tipus d'interès dels prèstecs, s'ha d'actualitzar la informació en els diaris i per Internet. Això fa que tant els diaris com Internet hagin de ser notificats dels canvis.

2. Patró a aplicar: Observer



3. Aplicació del patró:



Per a dissenyar la solució s'ha fet servir una interfície anomenada **Subject** que conté mètodes per enregistrar, esborrar i notificar als Observadors. També s'ha dissenyat una interfície anomenada **Observer** que conté el mètode update(int interest), que serà cridat per la classe que implementi Subject quan el tipus d'interès canvïi.

Interfícies:

```
public interface Observer {
    public void update(float interest);
}
```

```
public interface Subject {
    public void registerObserver(Observer observer);
    public void removeObserver(Observer observer);
    public void notifyObservers();
 }
```

Loan (prèstec implementació de Subject):

```
public class Loan implements Subject {
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    private String type;
    private float interest;
    private String bank;

    public Loan(String type, float interest, String bank) {
        this.type = type;
        this.interest = interest;
        this.bank = bank;
    }
   public float getInterest() {
        return interest;
    }
    public void setInterest(float interest) {
        this.interest = interest;
        notifyObservers();
    }
    public String getBank() { return this.bank; }
   public String getType() {return this.type; }
@Override
 public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
 @Override
    public void notifyObservers() {
        for (Observer ob : observers) {
            System.out.println("Notifying Observers on change in Loan
         interest rate");
            ob.update(this.interest);
        }
    }
}
```

Implementacions d'Observer (Diari i Internet):

```
public class Newspaper implements Observer {
    @Override
    public void update(float interest) {
        System.out.println("Newspaper: Interest Rate updated, new Rate is: "
                + interest);
    }
}
```

```
public class Internet implements Observer {
    @Override
    public void update(float interest) {
        System.out.println("Internet: Interest Rate updated, new Rate is: "
                + interest);
    }
}
```

4. Anàlisi del patró aplicat (si s'escau) En principi, la solució donada compleix els principis SOLID.

5. Programa principal que mostra l'ús del patró utilitzat.

```
public class ObserverTest {
    public static void main(String args[]) {
        // this will maintain all loans information
        Newspaper printMedia = new Newspaper();
        Internet onlineMedia = new Internet();
        Loan personalLoan = new Loan("Personal Loan", 12.5f,
                "Standard Charterd");
        personalLoan.registerObserver(printMedia);
        personalLoan.registerObserver(onlineMedia);
        personalLoan.setInterest(3.5f);
    }
}
```

4. Seguint l'exemple il·lustrat al campus sobre la construcció de Vehicles de diferents tipus Problema OpenClosed (Simple Factory), intenta dissenyar el problema de les transparències de teoria on es volen crear figures representades per Imatges o representades per Punts. Com et quedaria el codi? Hauries de modificar el disseny? S'aconsegueix no vulnerar el principi Open-Closed? Dóna les principals idees per a aconseguir no vulnerar-lo.

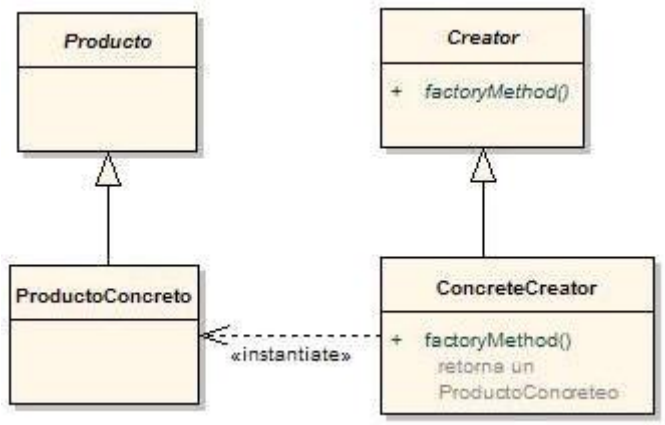
5. Es tenen tres tipus de Triangles: Equilàters, Escalens i Isòsceles que hereten de la classe Triangle. Es vol crear diferents tipus de triangles. com dissenyaries el seu constructor per a que es pogués crear un triangle cridant a crearTriangle(float costat1, float costat2, float costat3). On aniria aquest mètode? Per a dissenyar aquest problema, quin patró podries fer servir? Com l'aplicaries? Escriu el diagrama de classes que dissenyaries sota aquest patró, identificant les diferents classes del patró amb el teu problema concret. Dona l'exemple d'un programa principal per a veure com utilitzaria el teu disseny. Analitza si vulneres algun principi de disseny amb la teva solució i explica per què.

1. Problema identificat a solucionar:

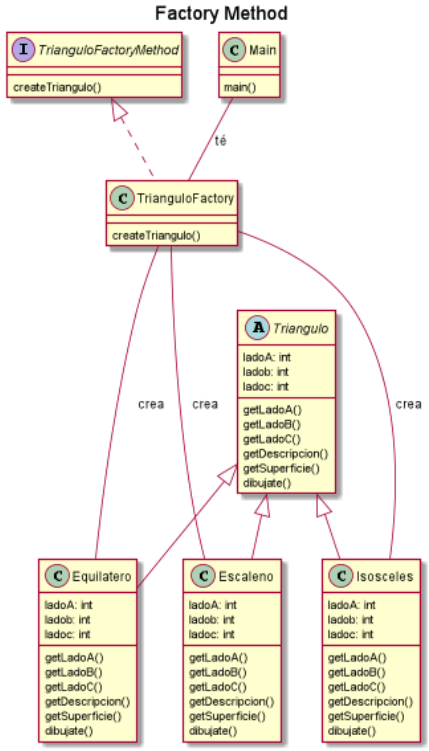
En aquest problema volem crear 3 objectes diferents que hereten d'una classe comuna mitjançant un únic mètode. De tal manera que, en funció dels paràmetres que passem al mètode ens creï un tipus d'objecte o un altre.

2. Patró a aplicar:

Patró Factory Method



3. Aplicació del patró:



4. Anàlisi del patró aplicat. (si s'escau)

Primerament, disposem de la classe Triangle, la qual és abstracta ja que sempre instanciarem un tipus de triangle concret i, per tant, mai disposarem d'un objecte d'aquest tipus.

```
public abstract class Triangulo {
    private int ladoA;
    private int ladoB;
    private int ladoC; // con sus get y set

    public Triangulo(int ladoA, int ladoB, int ladoC) {
        setLadoA(ladoA);
        setLadoB(ladoB);
        setLadoC(ladoC);
    }

    //Cada subclase debe redefinir estos tres métodos abstractos.
    public abstract String getDescripcion();

    public abstract double getSuperficie();

    public abstract void dibujate();

    public int getLadoA() {
        return ladoA;
    }

    public void setLadoA(int ladoA) {
```

Seguidament, disposem de una classe per cada tipus de triangle, les quals seran les que instanciarem. Aquí es mostra el cas equilàter, ja que els altres dos són anàlegs.

```
public class Equilatero extends Triangulo {

    public Equilatero(int anguloA, int anguloB, int anguloC) {
        super(anguloA, anguloB, anguloC);
    }

    public String getDescripcion() {
        return "Soy un Triangulo Equilatero";
    }

    public double getSuperficie() {
        // Aca iría el algoritmo para calcular superficie de un triangulo equilatero.
        return 0;
    }

    public void dibujate() {
        // Aca iría el algoritmo para dibujar un triangulo equilatero.
    }
}
```

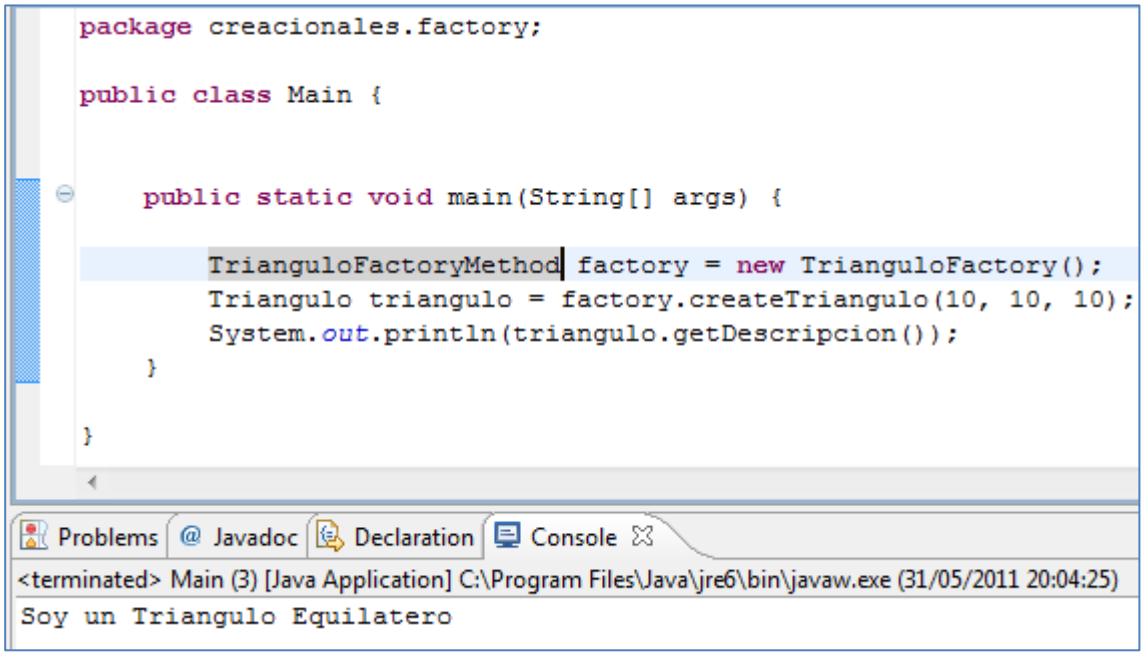
No obstant això, la classe encarregada de crear els tipus de triangles no ha de saber com es comporten aquests triangles internament, per la qual cosa creem aquesta Factory amb la seva corresponent interfície que s'encarregarà de fer els *new* dels triangles.

```
public interface TrianguloFactoryMethod {
    public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC);
}
```

```
public class TrianguloFactory implements TrianguloFactoryMethod {

    public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC) {

        if ((ladoA == ladoB) && (ladoA == ladoC)) {
            return new Equilatero(ladoA, ladoB, ladoC);
        }

        else if ((ladoA != ladoB) && (ladoA != ladoC) && (ladoB != ladoC)) {
            return new Escaleno(ladoA, ladoB, ladoC);
        }

        else {
            return new Isosceles(ladoA, ladoB, ladoC);
        }
    }

}
```

5. Programa principal que mostra l'ús del patró utilitzat.

Finalment, en el programa principal es fa ús del patró de la manera següent:



6. Observacions addicionals

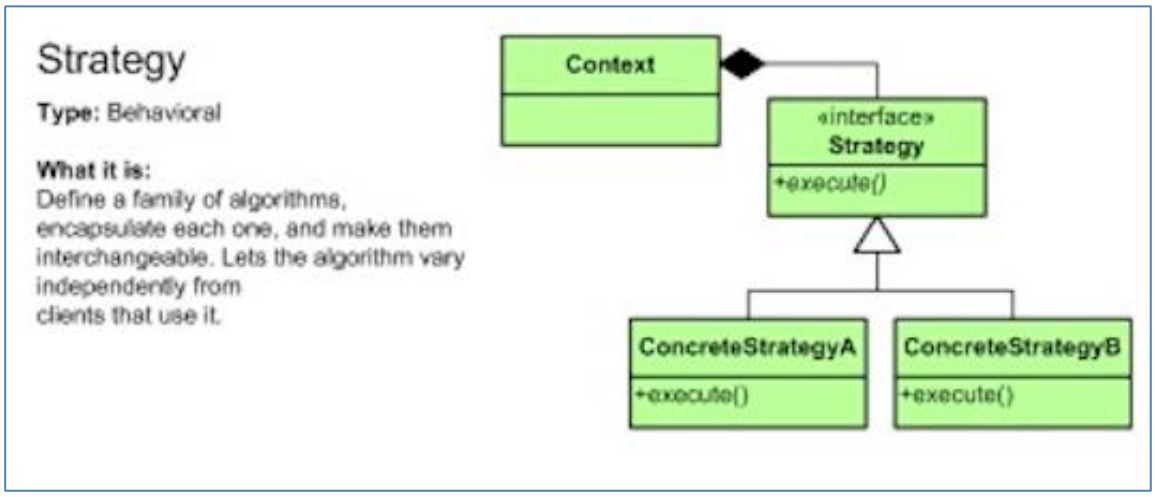
Finalment, cal destacar que en la implementació de la classe *TrianguloMethod* es vulnera el principi *Open-Closed*, tot i que no suposa un gran problema mentre ho tinguem controlat, ja que no hi ha nous tipus de triangles per afegir. També el podríem solucionar amb l'ús de reflexivitat.

6. Imaginem una biblioteca d'un institut que presta llibres als alumnes i professors. Imaginem que els alumnes poden associar-se a la biblioteca pagant una mensualitat. Amb la qual cosa un llibre pot ser prestat a Alumnes, Socis i Professors. Per decisió de l'administració, als socis se'ls prestarà el llibre més nou, després que està situat en bon estat i, finalment, aquell que estigués en estat regular. En canvi, si un alumne demana un llibre, passa tot el contrari. Finalment, als professors intentaran prestar-els-hi llibres bons, després els acabats de comprar i, finalment, els regulars. Suposa que tens la classe LibroFinder que disposa d'un mètode de cerca del llibre amb un soci concret. Per a dissenyar aquest problema, quin patró podries fer servir? Com l'aplicaries? Escriu el diagrama de classes que dissenyaries sota aquest patró, identificant les diferents classes del patró amb el teu problema concret. Dona l'exemple d'un programa principal per a veure com utilitzaria el teu disseny. Analitza si vulneres algun principi de disseny amb la teva solució i explica per què.

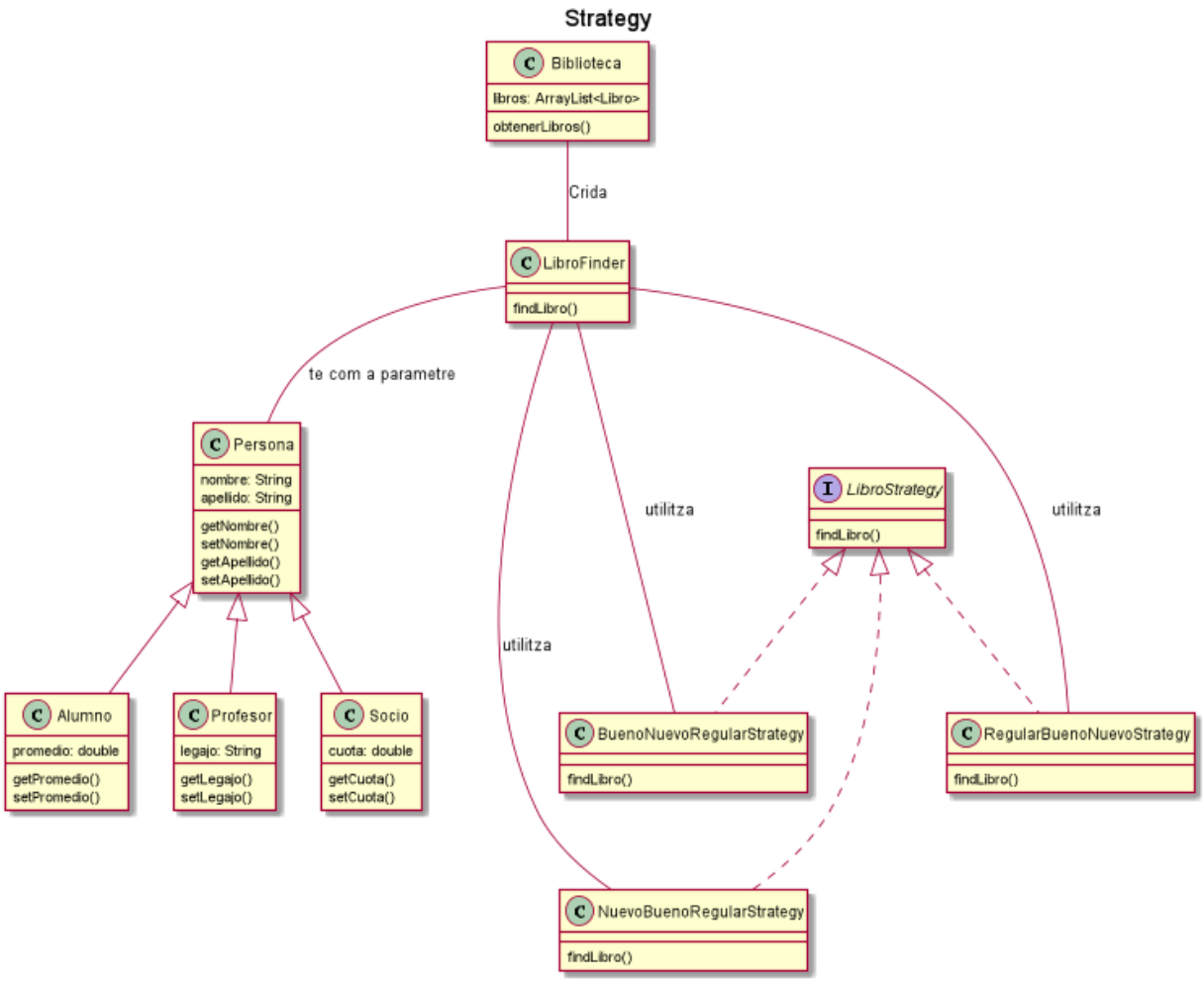
1. Problema identificat a solucionar:

En aquest problema hem de modificar l'ordenació del préstec d'uns llibres en funció de la persona que sol·liciti el préstec. És a dir, en funció d'un paràmetre hem de seguir una estratègia o una altra.

2. Patró a aplicar: Patró Strategy



3. Aplicació del patró:



4. Anàlisi del patró aplicat. (si s'escau)

Primerament, disposem de les classes que conformen els diversos tipus de clients de la biblioteca.

```
public class Persona {
    private String nombre;
    private String apellido;

    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
        return apellido;
    }
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

}
```

```
public class Alumno extends Persona{
    private double promedio;

    public double getPromedio() {
        return promedio;
    }

    public void setPromedio(double promedio) {
        this.promedio = promedio;
    }

}
```

```
public class Profesor extends Persona {
    private String legajo;

    public String getLegajo() {
        return legajo;
    }

    public void setLegajo(String legajo) {
        this.legajo = legajo;
    }

}
```

```
public class Socio extends Persona {
    private double cuota;

    public double getCuota() {
        return cuota;
    }

    public void setCuota(double cuota) {
        this.cuota = cuota;
    }

}
```

Seguidament, disposem de la classe *Biblioteca* la qual disposa de tot el catàleg de llibres.

```
public class Biblioteca {
    private static ArrayList<Libro> libros = new ArrayList<Libro>();

    public Biblioteca(){
        Libro libro = new Libro();
        libro.setEstado("Bueno");
        libro.setTitulo("Un titulo");

        // crear más libros
        libros.add(libro);
    }

    public static ArrayList<Libro> obtenerLibros(){
        return libros;
    }
```

Finalment, implementem les diverses estratègies de préstec dels llibres amb la seva corresponent interfície.

```
public interface LibroStrategy {

    public Libro findLibro(String titulo);

}
```

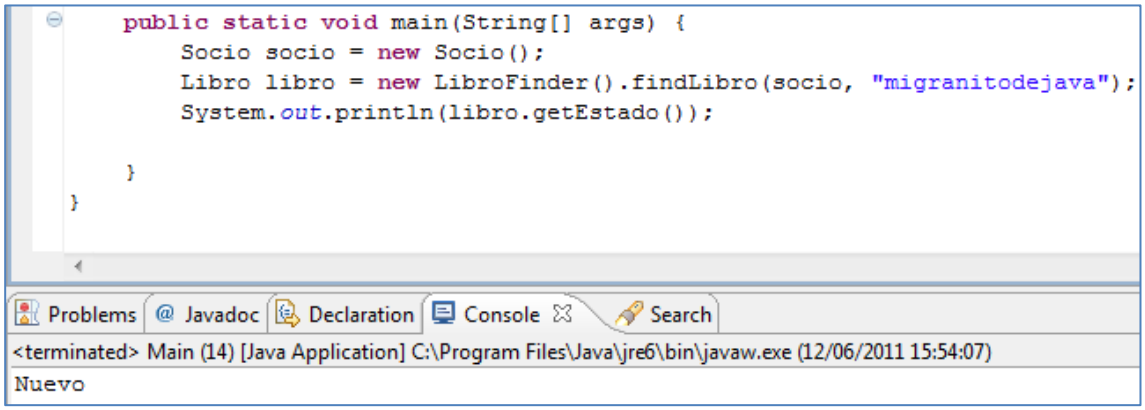
```java
public class NuevoBuenoRegularStrategy implements LibroStrategy {

    @Override
    public Libro findLibro(String titulo) {
        ArrayList<Libro> libros = Biblioteca.obtenerLibros();
        // Aquí iría un algoritmo que busque libros según el título.
        // Primero buscaría aquellos libros en estado nuevo,
        // luego los buenos y por último los regulares.
        // Aca soy vago y solo retorno un libro en estado nuevo.
        Libro libro = new Libro();
        libro.setEstado("Nuevo");
        return libro;
    }

}
```

```java
public class BuenoNuevoRegularStrategy implements LibroStrategy {

    @Override
    public Libro findLibro(String titulo) {
        ArrayList<Libro> libros = Biblioteca.obtenerLibros();
        // Aquí iría un algoritmo que busque libros según el títu
        // Primero buscaría aquellos libros en estado bueno,
        // luego los nuevo y por último los regulares.
        // Aca soy vago y solo retorno un libro en estado bueno.
        Libro libro = new Libro();
        libro.setEstado("Bueno");
        return libro;
    }

}
```

```java
public class RegularBuenoNuevoStrategy implements LibroStrategy {

    @Override
    public Libro findLibro(String titulo) {
        ArrayList<Libro> libros = Biblioteca.obtenerLibros();
        // Aquí iría un algoritmo que busque libros según el título.
        // Primero buscaría aquellos libros en estado regular,
        // luego los buenos y por último los nuevos.
        // Aca soy vago y solo retorno un libro en estado regular.
        Libro libro = new Libro();
        libro.setEstado("Regular");
        return libro;
    }

}
```

5. Programa principal que mostra l'ús del patró utilitzat.

Finalment, en el programa principal es fa ús del patró de la manera següent:



On el mètode *findLibro* de la classe *LibroFinder* funciona de la manera següent:

```
public class LibroFinder {

    public Libro findLibro(Persona persona, String titulo){
        LibroStrategy strategy =null;
        if (persona instanceof Socio) {
            strategy= new NuevoBuenoRegularStrategy();
        } else if (persona instanceof Profesor) {
            strategy= new BuenoNuevoRegularStrategy();
        } else {
            strategy= new RegularBuenoNuevoStrategy();
        }
        return strategy.findLibro(titulo);
    }

}
```

6. Observacions addicionals

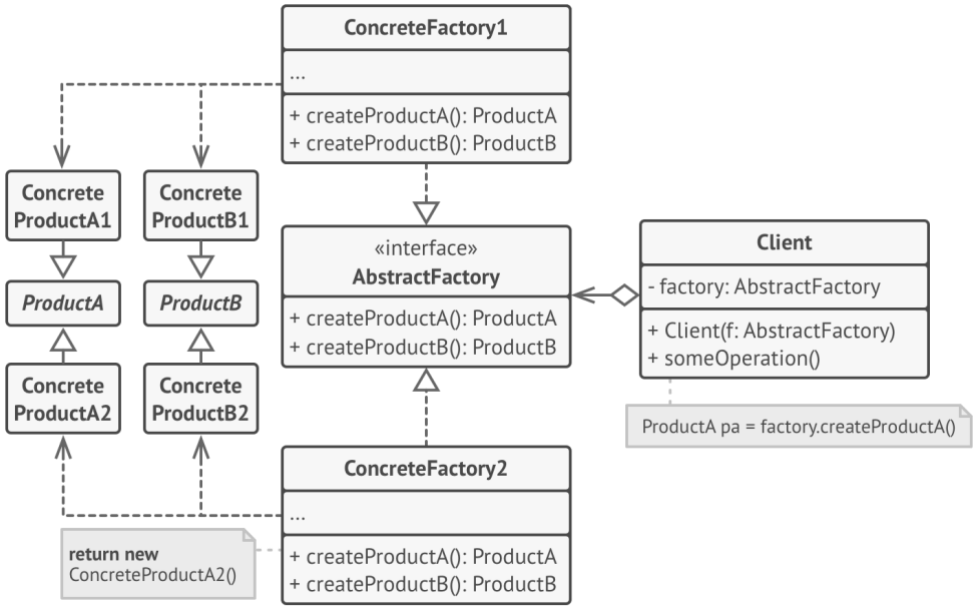
Aquí es vulnera el principi Open-Closed a la classe LibroFinder. que està actuant de Factory Method.

7. Es tenen finestres a definir per Microsoft, iOS i Linux, botons i checkbuttons que són diferents en Windows, iOS i Linux. Es vol donar el servei de construir aquests elements de forma flexible i extensible. Per a dissenyar aquest problema, quin patró podries fer servir? Com l'aplicaries? Escriu el diagrama de classes que dissenyaries sota aquest patró, identificant les diferents classes del patró amb el teu problema concret. Dona l'exemple d'un programa principal per a veure com utilitzaria el teu disseny. Analitza si vulneres algun principi de disseny amb la teva solució, on el vulneres i explica per què.

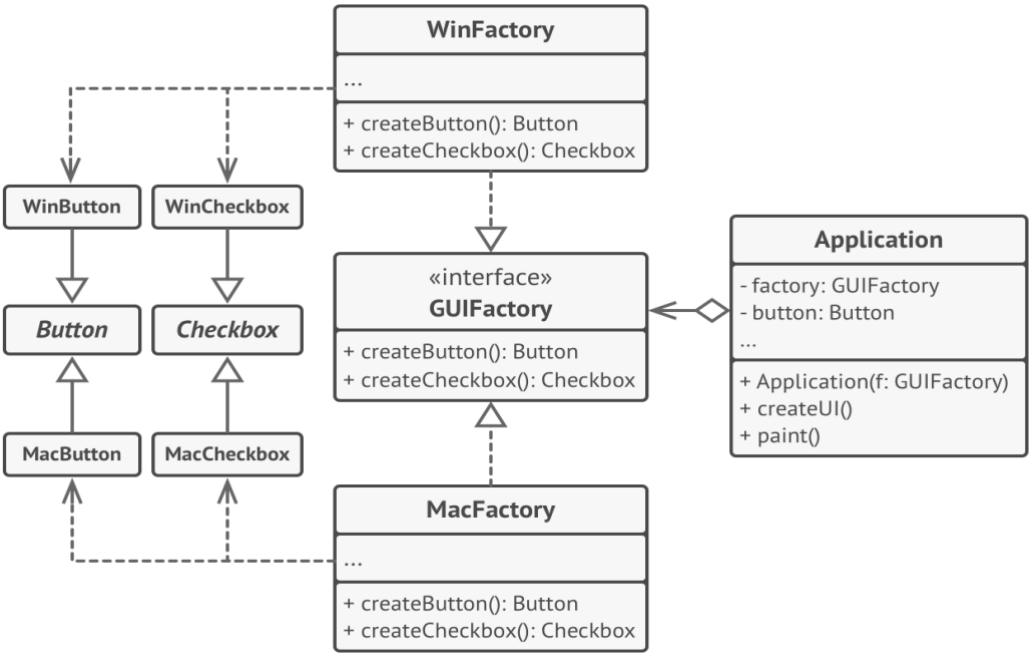
1. Problema identificat a solucionar:

Volem crear diversos paquets d'objectes que tenen característiques diferents els uns dels altres. És a dir, en funció del sistema operatiu que escollim, els botons que creem seran d'una manera o una altra, per exemple.

2. Patró a aplicar: Abstract Factory



3. Aplicació del patró:



4. Anàlisi del patró aplicat. (si s'escau)

Primerament, observem la *Absract Factory* que estendran les respectives fàbriques concretes.

Aquí podem observar quins objectes hauran de crear aquestes fàbriques.

```
// The abstract factory interface declares a set of methods that
// return different abstract products. These products are called
// a family and are related by a high-level theme or concept.
// Products of one family are usually able to collaborate among
// themselves. A family of products may have several variants,
// but the products of one variant are incompatible with the
// products of another variant.
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox
```

Seguidament, disposem de les *Concrete Factory* encarregades de crear els objectes del tipus concret que correspon a cada fàbrica.

```
// Concrete factories produce a family of products that belong
// to a single variant. The factory guarantees that the
// resulting products are compatible. Signatures of the concrete
// factory's methods return an abstract product, while inside
// the method a concrete product is instantiated.
class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

// Each concrete factory has a corresponding product variant.
class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()
```

Finalment, cal destacar que tots els tipus d'objectes que es creen a cada fàbrica disposen d'una interfície comuna.

```
// Each distinct product of a product family should have a base
// interface. All variants of the product must implement this
// interface.
interface Button is
    method paint()

// Concrete products are created by corresponding concrete
// factories.
class WinButton implements Button is
    method paint() is
        // Render a button in Windows style.

class MacButton implements Button is
    method paint() is
        // Render a button in macOS style.
```

```
// Here's the base interface of another product. All products
// can interact with each other, but proper interaction is
// possible only between products of the same concrete variant.
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in Windows style.

class MacCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in macOS style.
```

5. Programa principal que mostra l'ús del patró utilitzat.

Finalment, en el programa principal es fa ús del patró de la manera següent:

```
// The client code works with factories and products only
// through abstract types: GUIFactory, Button and Checkbox. This
// lets you pass any factory or product subclass to the client
// code without breaking it.
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()

// The application picks the factory type depending on the
// current configuration or environment settings and creates it
// at runtime (usually at the initialization stage).
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Error! Unknown operating system.")

        Application app = new Application(factory)
```

En aquest cas, al *main* creem la instància de la aplicació passant-li per paràmetre el tipus de dispositiu en el que es troba i és la pròpia classe *Application* l'encarregada de cridar a la fàbrica corresponent.

8. Es vol modelar una aplicació que dissenyi un estat amb el seu propi govern. El govern és únic per a tota l'aplicació. Per a dissenyar aquest problema, quin patró podries fer servir? Com

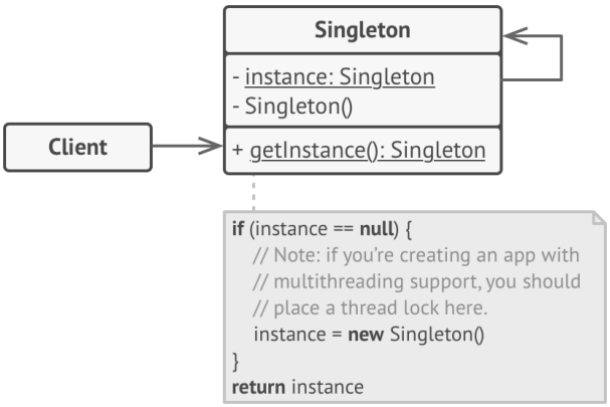
l'aplicaries? Escriu el diagrama de classes que dissenyaries sota aquest patró, identificant les diferents classes del patró amb el teu problema concret. Dona l'exemple d'un programa principal per a veure com utilitzaria el teu disseny. Analitza si vulneres algun principi de disseny amb la teva solució, on el vulneres i explica per què.

1. Problema identificat a solucionar:

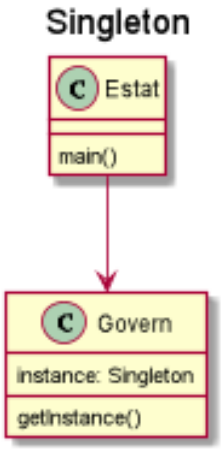
Volem disposar d'un únic govern. Així doncs, ens hem d'assegurar que en qualsevol moment únicament hi ha una única instància de la classe Govern a cada Estat.

2. Patró a aplicar:

Patró Singleton



3. Aplicació del patró:



4. Anàlisi del patró aplicat. (si s'escau)

Creem el mètode *getInstance* a la classe Govern per assegurar-nos que només disposarem d'una única instancia d'aquesta classe.

```
public class Govern {
      private volatile static Govern uniqueInstance;
      public static Govern getInstance() {
          if (uniqueInstance == null) {
```

```
                synchronized (Govern.class) {
                    if (uniqueInstance == null) {
                        uniqueInstance = new Govern();
                    }
                }
            }
            return uniqueInstance;
        }
}
```

5. Programa principal que mostra l'ús del patró utilitzat.

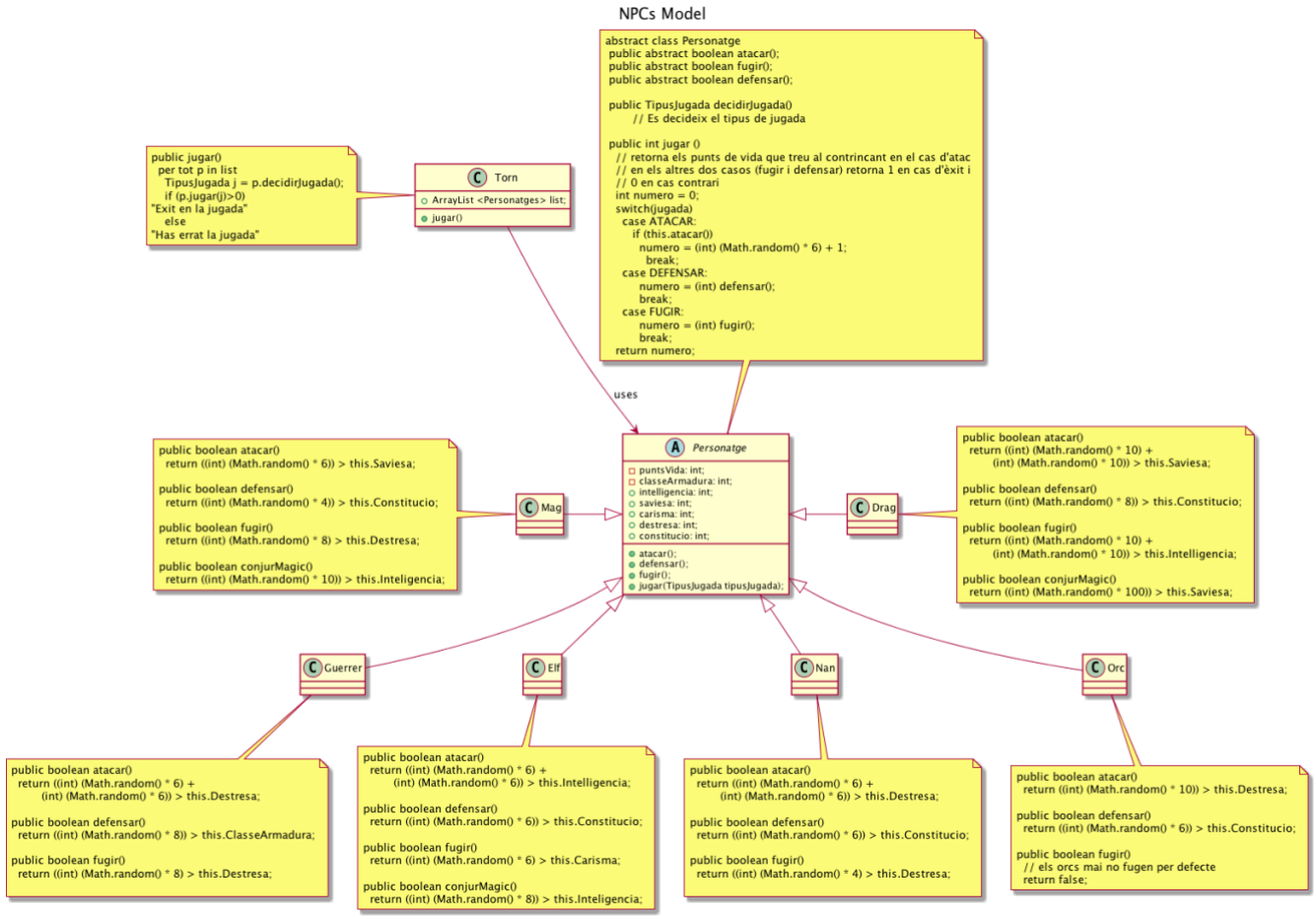
Finalment, en el programa principal es fa ús del patró de la manera següent:

```
public class Estat {

        public static void main(String args[]) {
            Controlador controlador = Controlador.getInstance();
        }
}
```

9. (Examen final 2017-18) En un joc de rol es volen dissenyar els diferents personatges que hi surten: guerrers, mags, elfs, nans, orcs i dragons. Tots els personatges tenen Punts de Vida i Classe d'Armadura i es caracteritzen per la seva Intel·ligència, Saviesa, Destresa, Carisma i Constitució. Els diferents personatges només poden atacar, defensar-se i fugir. Per això necessiten tirar un o varis daus - segons el tipus de personatge, el dau tindrà diferent número de cares - i superar amb la/es tirada/es alguna de les característiques que tenen. Per exemple, si un guerrer vol atacar, s'haurà de llençar un dau de 6 cares dues vegades i la suma final de las tirades haurà de ser més gran que el valor de la seva Destresa. Si en canvi, un drac vol fugir ha de tirar un dau de 10 cares dos cops i la suma de les tirades haurà de ser superior al valor de la seva Intel·ligència. Només quan s'aconsegueix superar el valor, el personatge aconsegueix realitzar l'acció. Si està atacant amb èxit, podrà treure Punts de Vida al seu atacant (tants com un dau de 6). Si no s'aconsegueix superar el valor, el seu atac fallarà.

Al llarg del joc, però, els personatges poden trobar objectes, pujar de nivell o trobar conjurs que fan canviar els tipus d'atac, defensa o fugida, canviant el tipus de dau, tenint bonificacions especials en les característiques del personatge en certes tirades, o inclús canviant la manera de calcular l'acció. Per exemple, un Mag de nivell 2 no pot usar conjurs màgics per atacar però quan sigui de nivell 3, el seu atac es decidirà fent la tirada d'atac normal i la del conjur màgic.

Davant d'aquest problema, un dissenyador de software, preveient aquest funcionament del joc, ha fet el disseny següent:



a) Quins principis S.O.L.I.D. vulnera aquest codi? Raona la resposta.

S: No el vulnera, tot i que hi hagin diferents mètodes, la classe té una única responsabilitat. Es pot raonar que el vulnera si es diu que la classe està donant la responsabilitat de decidirJugada i la responsabilitat de Jugar.

O: Es vulnera clarament. Quan es vulgui fer un altre tipus d'acció, s'ha de modificar la classe Personatge, tot i que es diu en l'enunciat que NOMES pot atacar, defensar i fugir i el que realment canviarà serà la manera de fer l'acció o bé que se'n podran fer de combinades. Si es considera que no pot haver més que atacar, defensar i fugir, aquest principi no el vulnera.

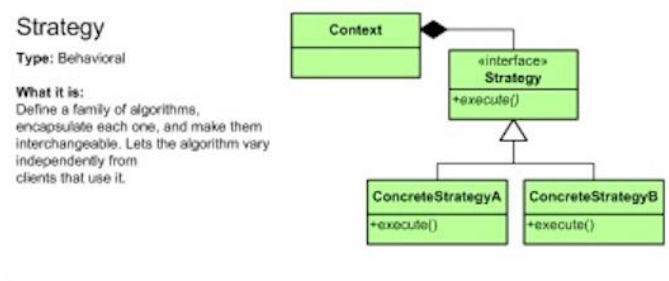
L: No el vulnera. Totes les subclasses fan un comportament esperat per una classe Personatge, tot i que es pot argumentar que els Orcs no fugen mai, però es el seu comportament. No es que no ho sàpiguen fer.

I: No aplica

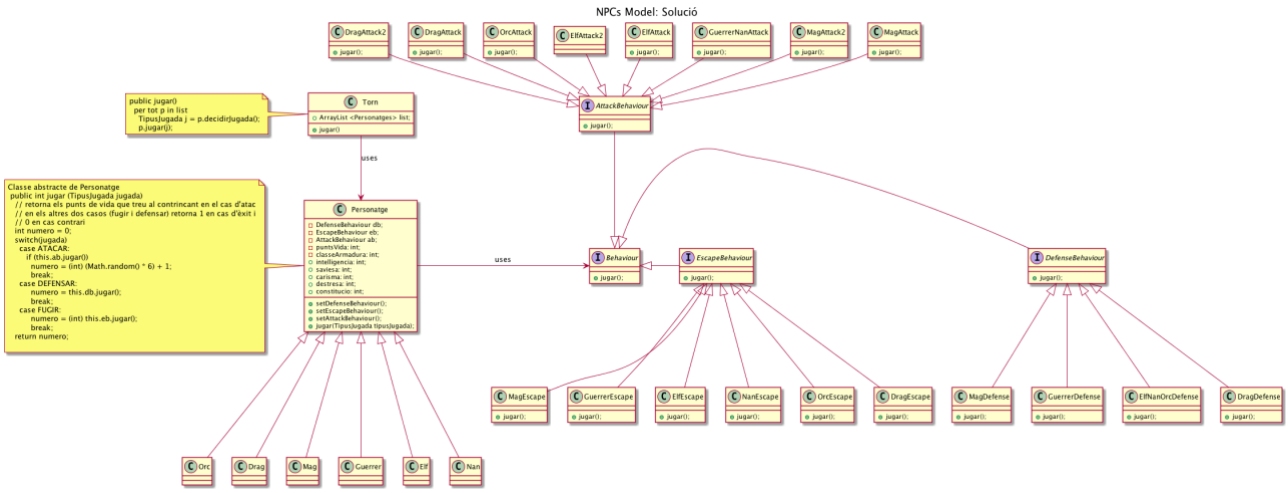
D: Es pot veure dependència de les accions de cada personatge en implementacions no concretes i estàtiques. Bàsicament és aquesta vulneració la que porta a aplicar el patró Strategy

b) Com redissenyaries aquest disseny? Quin/s patró/ns de disseny faries servir? Per a contestar aquest apartat omple els apartats següents.

b.1. Nom del patró i tipus de patró: **Strategy. Patró de comportament.**



b.2. Aplicació del patró:



b.3. Anàlisi del patró aplicat en relació als principis S.O.L.I.D.

Igualment es vulnera el Open-Closed Principle que no es pot evitar. Es podria acabar treient de Personatge el switch fent un array de 3 Behaviours en el Personatge i que decidirJugada ens doni l'index del tipus de Behaviour que es vulgui fer en aquell moment. Tot i que es treu la vulneració de la classe Personatge, igualment tindrem el problema d'Open-Closed en el decidirJugada.

En el mètode jugar, de la classe Personatge, es separa la decisió de l'èxit de l'atac i el dany que fa en atacar. Aquí es podria veure una vulneració del S ja que s'ataca i es calcula el dany al contrincant. El mètode jugar detallat a l'apartat 3.4 s'hauria de partir en dues parts per a poder calcular el dany només en cas d'èxit.

b.4. Programa jugar() que mostra l'ús del patró utilitzat:

```
public int jugar (TipusJugada jugada)
  // retorna els punts de vida que treu al contrincant en el cas d'atac
  // en els altres dos casos (fugir i defensar) retorna 1 en cas d'èxit i
  // 0 en cas contrari
  int numero = 0;
  switch(jugada)
    case ATACAR:
      if (this.ab.jugar())
            numero = (int) (Math.random() * 6) + 1;
            break;
    case DEFENSAR:
        numero = this.db.jugar();
            break;
    case FUGIR:
        numero = (int) this.eb.jugar();
            break;
  return numero;
```

b.5. Observacions addicionals:

El tema del Open-Closed no està solucionat. A part a la part de crear els diferents Personatges es pot aplicar una AbstractFactory que instancïi la forma d'atacar, de defensar i de fugir de cada personatge.

c) En una segona versió del joc es volen fer atac mitjançant tropes de personatges. Una tropa es defineix com el conjunt de diferents flancs, al marge que pot tenir personatges que no estiguin a cap flanc. En un flanc sempre hi ha com a mínim un personatge. Quin patró utilitzaries per a definir les tropes? Raona la teva resposta en 10 línies com a màxim.

Aquí aplicaríem el patró Composite per a poder fer aquest composició de conjunts formats per més conjunts o elements aïllats, on els Components són la tropa/flanc que estaries formats per flancs i les Leafs serien els personatges.

10. Imagineu-vos que el dia de Reis, al despertar-vos al mati, en lloc dels regals tradicionals sota l'arbre, trobeu unes caixes amb les següents característiques. D'una banda, tenim unes caixes que disposen d'una etiqueta on hi indica el nom d'un producte concret i que indicarà l'únic tipus de producte que trobarem quan obrim la caixa (en quantitats diverses). D'altra banda, hi ha caixes etiquetades amb franges d'edat, les quals contenen un conjunt d'objectes diversos però tots pensats per aquella edat concreta. Per exemple, a una caixa amb la franja d'edat 3-5 hi trobarem el conte de la Caputxeta Vermella, un trencaclosques de poques peces i una pilota petita. Quin patró o patrons creus que estan descrits en aquesta escena?

Les caixes representen dos patrons de disseny: ***Factory Method*** i ***Abstract Factory***.

D'una banda, les caixes amb franges d'edat fan referència al patró *Abstract Factory* perquè cada edat fa referència a una família de productes concrets. Així doncs, seguint l'esquema de teoria, cada caixa amb una franja d'edat concreta representa una *ConcretFactory*, la qual s'encarrega de produir (crear) el conjunt de productes referents a l'edat en qüestió, mentre que els productes que es creen a cada fàbrica d'edat representen els *ConcreteProduct*. Finalment, cal tenir en compte que totes aquestes capses per franja d'edat hereten d'una interfície genèrica sense franja d'edat concreta anomenada *AbstractFactory*.

D'altra banda, la caixa amb una sola etiqueta representa el patró de disseny *Factory Method*, ja que ens permet crear només un tipus d'objecte concret. Així doncs, seguint l'esquema de teoria,

cada fàbrica representa un *ConcreteCreator* que s'encarrega de la producció d'un tipus de producte concret, els quals representen els *ConcreteProduct*. A més a més, totes les *ConcreteCreator* implementen una interfície comuna anomenada *Creator*. No obstant això, si encara volguéssim filar més prim, podríem separar aquesta interfície comuna en diverses interfícies en funció de si els productes que crearem seran jocs, joguines, contes, etc, de tal manera que cada *ConcreteCreator* implementaria la interfície que fes referencia al tipus de producte que vol crear. Per exemple, la *ConcreteCreator* del llibre dels Tres Porquets, implementaria la interfície *Creator* referent als contes.

11. La Constitució dels Estats Units especifica els mitjans amb què és elegit un president, limita el mandat i defineix l'ordre de successió. Com a resultat, pot haver-hi com a màxim un president actiu en un moment donat. Independentment de la identitat personal del president actiu, el títol, "El president dels Estats Units", és un reconeixement global que identifica a la única persona que ostenta aquest títol en un moment concret. Quin patró creus que podem extreure d'aquesta situació?

En aquesta situació hi podem observar la implementació del patró ***Singleton***. El càrrec de President dels Estats Units és únic, per la qual cosa nomes hi pot haver una persona amb aquesta tasca. Així doncs, el Govern s'ha d'assegurar que no hi ha dues persones alhora que portin a terme aquesta funció. En termes de programació orientada a objectes, el títol de President seria la classe de la qual volem tenir una única instància i la persona que ostenta el càrrec seria l'única instància d'aquesta classe. Finalment, cal recordar que, en *Java*, una bona manera d'implementar aquest patró és fent ús del *Enum*.

12. En els combats de estan permesos tots els estils de batalla. Així doncs, els participants poden lluitar cos a cos, des de la distància, amb els punys, fent ús de puntades de peu i també aplicant claus d'immobilització. Tots aquests estils de combat són vàlids i els poden dur a la victòria durant un combat. No obstant això, cada lluitador destaca en un aspecte concret i mostra més mancances en els altres. Així doncs, cada lluitador decideix aplicar l'estil que més el beneficia. Explica quin patró de disseny faries servir en aquest context i com l'aplicaries.

En aquest cas podem aplicar el patró ***Strategy*****.** Tal i com especifica l'enunciat, tots els estils de lluita són vàlids i persegueixen el mateix objectiu: guanyar el combat, per la qual cosa podem

interpretar els diferents estils de combat com possibles estratègies alternatives. Així doncs, seguint l'esquema de teoria, en aquest cas el combat *Context* estaria representat pel combat, en el que cada lluitador decideix quin estil de lluita portarà a terme, per la qual cosa cadascun dels lluitadors representaria un *Client*. Finalment, és clar que els diversos estils de lluita representarien les diverses *ConcreteStartegy*, les quals implementarien una interfície comuna *Strategy* que, en aquest cas, equivaldria a la lluita en sí i disposaria, per exemple, d'un mètode *lluitar()*.

13. En una subhasta, cada potencial comprador disposa d'una paleta numerada que utilitza per indicar la quantitat que oferta per un producte concret. D'altra banda, el subhastador inicia la subhasta i detecta quan s'eleva una paleta per acceptar l'oferta en cas que aquesta sigui superior al preu actual del producte. L'acceptació de l'oferta fa que el subhastador modifiqui el preu del producte i aleshores torna a començar el cicle. Quin patró creus que s'escau més en aquesta situació?

En aquest cas el patró més escaient és el patró ***Observer***, ja que el subhastador ha d'estar pendent de les accions que realitzen els potencials compradors per tal de, en cas que algun d'aquest realitzi una licitació, actualitzar a l'alça el preu del producte. Així doncs, seguint l'esquema de teoria, el subhastador seria l'*Observer*, el qual estarà pendent de si algun potencial comprador (que en aquest cas fan la funció de *Observable* o *Subject*) realitza alguna acció que provoqui que hagi de modificar el preu del objecte subhastat. En addició, cal comentar que en argot informàtic direm que l'*Observer* s'enregistra a l'*Observable* per indicar que està pendent de les seves accions. Finalment, és important remarcar que en aquest cas concret disposem d'un únic objecte *Observer* i diversos objectes *Observables*, però en funció del *Context* aquest números poden variar.