

Introduction à la programmation en JAVA



Table des matières

- I. Introduction à Java et historique du langage
- II. Notre outil de développement : *Eclipse Mars*
- III. Le langage Java et sa syntaxe
- IV. La POO avec Java
- V. API Java
- VI. La gestion des exceptions
- VII. Les collections
- VIII. La sérialisation
- IX. Les Design Patterns

Les Design Patterns



I .Design Pattern - Définition

- Le terme "Design Pattern" peut se traduire en français par "modèle de conception" ou "patron de conception" Il s'agit en fait d'une solution générale à un problème récurrent en développement logiciel.
- C'est une façon générique d'organiser ses modules pour répondre à une problématique et ce peu importe le langage.
- On en trouve 23 regroupés en 3 types :
 - Création : pour faciliter la configuration de classes et objets.
 - Structure : pour organiser les classes d'une application.
 - Comportement : pour organiser les objets et faciliter la communication entre eux.

Aperçu du chapitre

I. Strategy

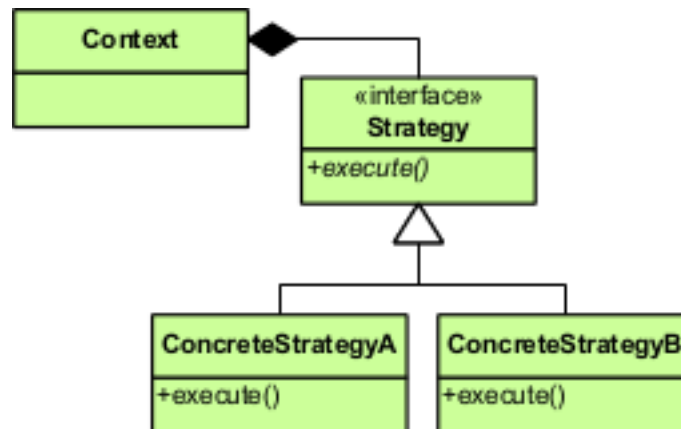
II. Composite

III. Observer

IV. Abstract Factory

I . Le Design Pattern Strategy

- La problématique de ce pattern est plutôt simple : comment faire pour réaliser différentes opérations avec un seul et même objet ?
Vous me direz, c'est simple on fait une classe avec toutes les opérations ! Mais non car procéder comme ceci violerait un principe SOLID : le Single Responsibility principle (une classe gère une chose).
- Pour éviter ceci, on structure nos classes de la manière suivante :



I . Le Design Pattern Strategy : Avantages: - Inconvénients

- Ce pattern offre les avantages:
 - Une meilleure lisibilité du code.
 - D'éviter de violer un principe SOLID.
 - Mais avant tout de définir plusieurs algorithmes interchangeables dynamiquement.
- Son seul inconvénient ? Il nécessite d'ajouter une classe.

I . Le Design Pattern Strategy : Exemple

- Vous naviguez sur un site d'e-commerce de votre choix. Un article vous intéresse, vous voulez l'acheter : vous l'ajoutez au panier. Vous ouvrez votre panier, et là, le site vous demande d'effectuer l'action abstraite de "payer".
- Elle ne vous paraît pas abstraite, car vous auriez des idées d'implémentations ? Il est possible de payer de différentes manières : directement par carte, par Paypal... La méthode "payer" peut être implémentée de différentes manières : c'est là l'intérêt du Pattern Strategy.

I . Le Design Pattern Strategy : Exemple

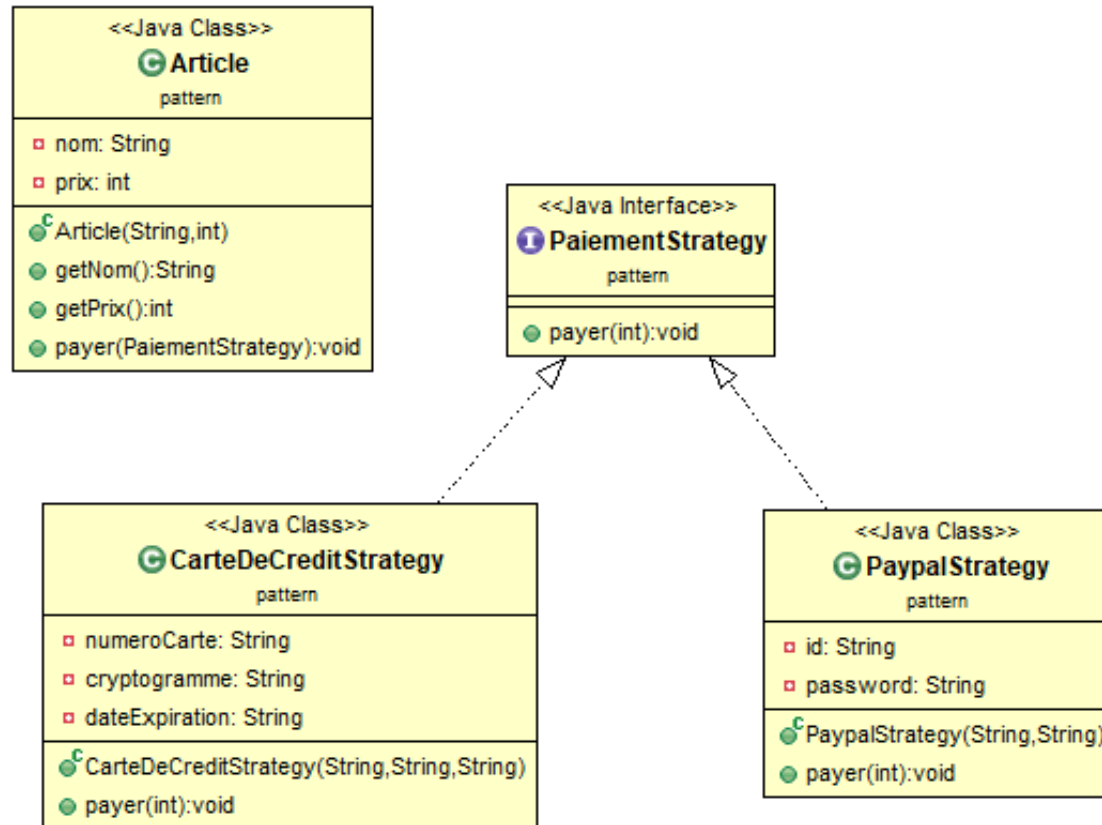


Diagramme de classes

I . Le Design Pattern Strategy : Exercices

1. Rajouter à l'exemple un nouveau moyen de paiement en espèces. Remplissez les méthodes d'actions par un affichage console.
2. On se place dans le cadre d'une application de facturation destinée à plusieurs pays. Proposez une solution flexible permettant de personnaliser l'affichage d'une facture selon le pays. Le texte et les prix doivent être automatiquement adaptés. La solution proposée doit faciliter l'ajout d'un nouveau pays sans changer l'interface du client. Les pays visés sont la Belgique, le Royaume-Uni et l'Allemagne. Modélisez puis programmez les différents algorithmes nécessaires à la mise en place de ce système.

Le Design Pattern Strategy : Exercices

Output - prjFacturation (run) x

run:
Par défaut (Belgique)

Le montant de la facture est :50.5 €

Allemagne

Der Rechnungsbetrag ist : € 50.5

Royaume-Uni

The amount of the invoice is : £ 50.5
BUILD SUCCESSFUL (total time: 0 seconds)

```
package prjfacturation;
public class Client {
    public static void main(String[] args) {
        Facture maFacture=new Facture(50.50);
        System.out.println("Par défaut (Belgique)");
        System.out.println("-----");
        37
        maFacture.affiche();
        System.out.println("\nAllemagne");
        System.out.println("-----");
        maFacture.setAffichage(new AffichageAllemagne());
        maFacture.affiche();
        System.out.println("\nRoyaume-Uni");
        System.out.println("-----");
        maFacture.setAffichage(new AffichageRoyaumeUni());
        maFacture.affiche();
    }
}
```

I . Le Design Pattern Strategy : Exercices

3. On veut définir une classe Valideur capable de valider différentes saisies (sous forme de String), par exemple la saisie de valeurs entières ou la saisie d'adresse mail. Exploitez les exceptions.

```
Output - prjValideur (run)
run:
La chaine de caractère : 6
Est-ce que 6 est un entier valide ? true
Est-ce que 6 est un mail valide ? false
-----
La chaine de caractère : toto@gmail.com
Est-ce que toto@gmail.com est un entier valide ? false
Est-ce que toto@gmail.com est un mail valide ? true
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
package prjvalideur;
public class Client {
    public static void main(String[] args) {
        String txt="6";
        System.out.println("La chaine de caractère : "+txt);
        Valideur valEntier=new Valideur(new FormatEntier(),txt);
        System.out.println("Est-ce que "+txt+ " est un entier valide ? "+valEntier.isValide());
        Valideur valMail=new Valideur(new FormatMail(),txt);
        System.out.println("Est-ce que "+txt+ " est un mail valide ? "+valMail.isValide());
        System.out.println("-----");
        txt="toto@gmail.com";
        System.out.println("La chaine de caractère : "+txt);
        Valideur valEntier2=new Valideur(new FormatEntier(),txt);
        System.out.println("Est-ce que "+txt+ " est un entier valide ? "+valEntier2.isValide());
        Valideur valMail2=new Valideur(new FormatMail(),txt);
        System.out.println("Est-ce que "+txt+ " est un mail valide ? "+valMail2.isValide());
    }
}
```

Aperçu du chapitre

I. Strategy

II. Composite

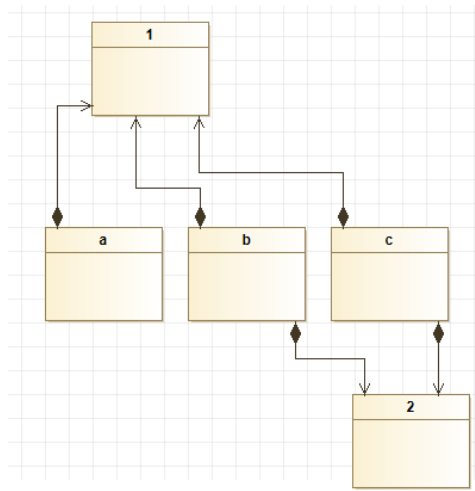
III. Observer

IV. Abstract Factory

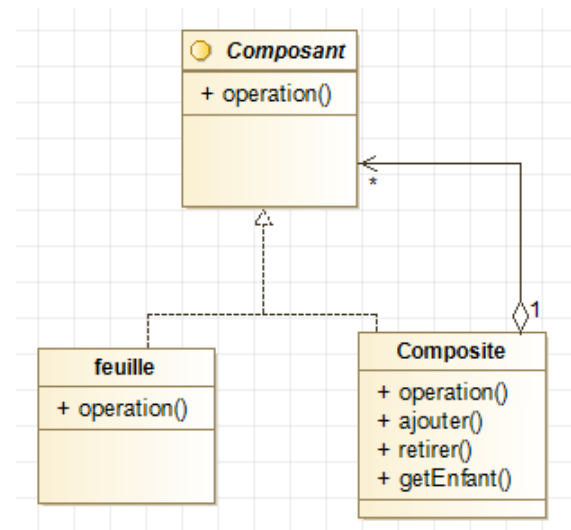
II . Le Design Pattern Composite - Exemple

- Il est facile de gérer des objets en Java. Il est toutefois beaucoup plus dur de gérer un ensemble d'objets en tant qu'un seul.
- Comme nous pouvons voir sur cette image, en java lorsque l'on veut créer des objets composés, il faut créer une classe par objet. De plus, pour supprimer un élément d'un objet, il faut directement aller dans le code de sa classe. Comment pouvons nous faciliter la gestion de cet ensemble d'objets ?

Problème:

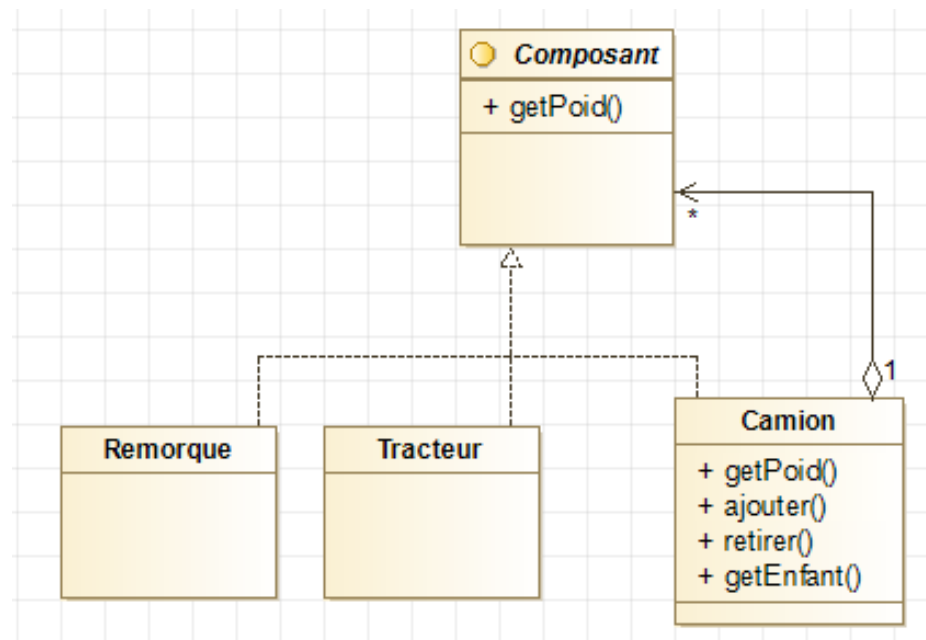


Solution:



II . Le Design Pattern Composite - Exemple

- Prenons l'exemple d'un camion semi-remorque. Ce dernier est composé d'un tracteur et d'une remorque, qui ont un poids séparé mais également un poids camion entier. Un tracteur routier doit également pouvoir rouler sans remorque. Nous disposons donc du diagramme suivant:



II . Le Design Pattern Composite - Exemple

```
public interface Composant {  
    public int getPoids();  
}
```

```
public class Remorque implements Composant {  
    private int poids;  
  
    public Remorque(int poids) {  
        this.poids = poids;  
    }  
  
    @Override  
    public int getPoids() {  
        return this.poids;  
    }  
}
```

```
public class Tracteur implements Composant {  
    private int poids;  
  
    public Tracteur(int poids) {  
        this.poids = poids;  
    }  
  
    @Override  
    public int getPoids() {  
        return this.poids;  
    }  
}
```


II . Le Design Pattern Composite - Exemple

```
public class CamionComposite implements Composant {  
    private Collection children;  
  
    public CamionComposite() {  
        children = new ArrayList();  
    }  
  
    public void add(Composant composant){  
        children.add(composant);  
    }  
  
    public void remove(Composant composant){  
        children.remove(composant);  
    }  
  
    public Iterator getChildren() {  
        return children.iterator();  
    }  
  
    @Override  
    public int getPoids() {  
        int result = 0;  
        for (Iterator i = children.iterator(); i.hasNext(); ) {  
            Object objet = i.next();  
  
            Composant composant = (Composant)objet;  
  
            result += composant.getPoids();  
        }  
        return result;  
    }  
}
```

II . Le Design Pattern Composite - Exemple

```
public class Main
{
    public static void main(String[] args)
    {
        Remorque maRemorque = new Remorque(11);
        System.out.println("Le poids de ma remorque est:");
        System.out.println(maRemorque.getPoids());
        System.out.println("tonnes");

        Tracteur monTracteur = new Tracteur(8);
        System.out.println("Le poids de mon tracteur est:");
        System.out.println(monTracteur.getPoids());
        System.out.println("tonnes");

        CamionComposite semiRemorque = new CamionComposite();
        semiRemorque.add(maRemorque);
        semiRemorque.add(monTracteur);
        System.out.println("Le poids de mon semi-remorque est:");
        System.out.println(semiRemorque.getPoids());
        System.out.println("tonnes");
    }
}
```

II . Le Design Pattern Composite - Exercices

1. Rajouter une remorque groupe électrogène au camion.
2. Supposons que vous devez construire un logiciel pour un revendeur connu. Ce dernier possède plusieurs magasins répartis dans tout le pays. L'application doit être capable de calculer le bénéfice par ville et par province.
3. Un système de fichiers contient des fichiers et des répertoires. Ces répertoires contiennent à leur tour des fichiers et des sous-répertoires. On veut pouvoir lister le système de fichiers, les répertoires et les fichiers (en donnant leur nom).

Aperçu du chapitre

I. Strategy

II. Composite

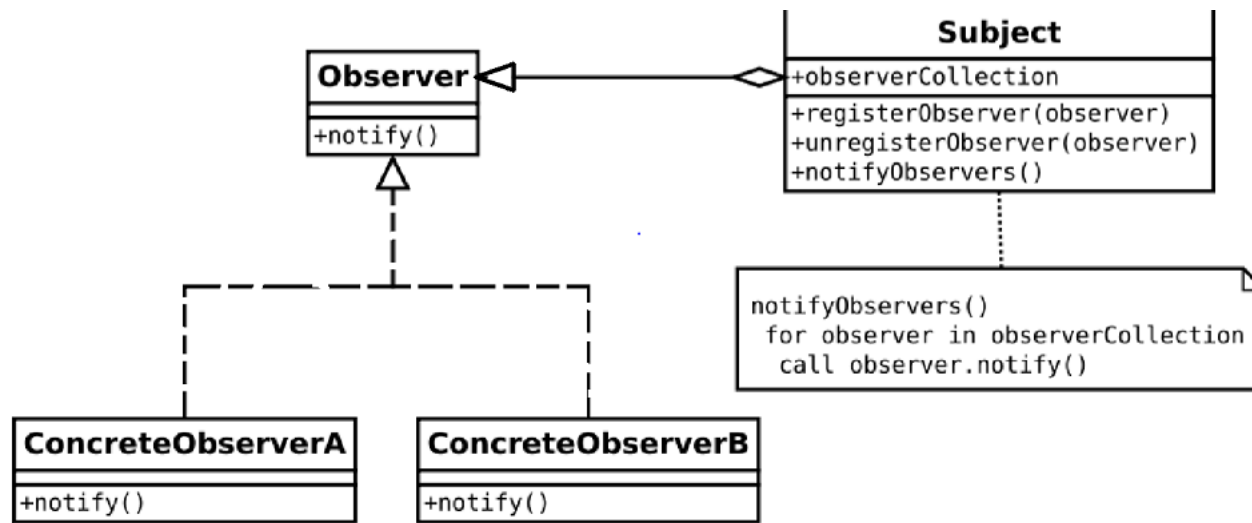
III. Observer

IV. Abstract Factory

III . Le Design Pattern Observer

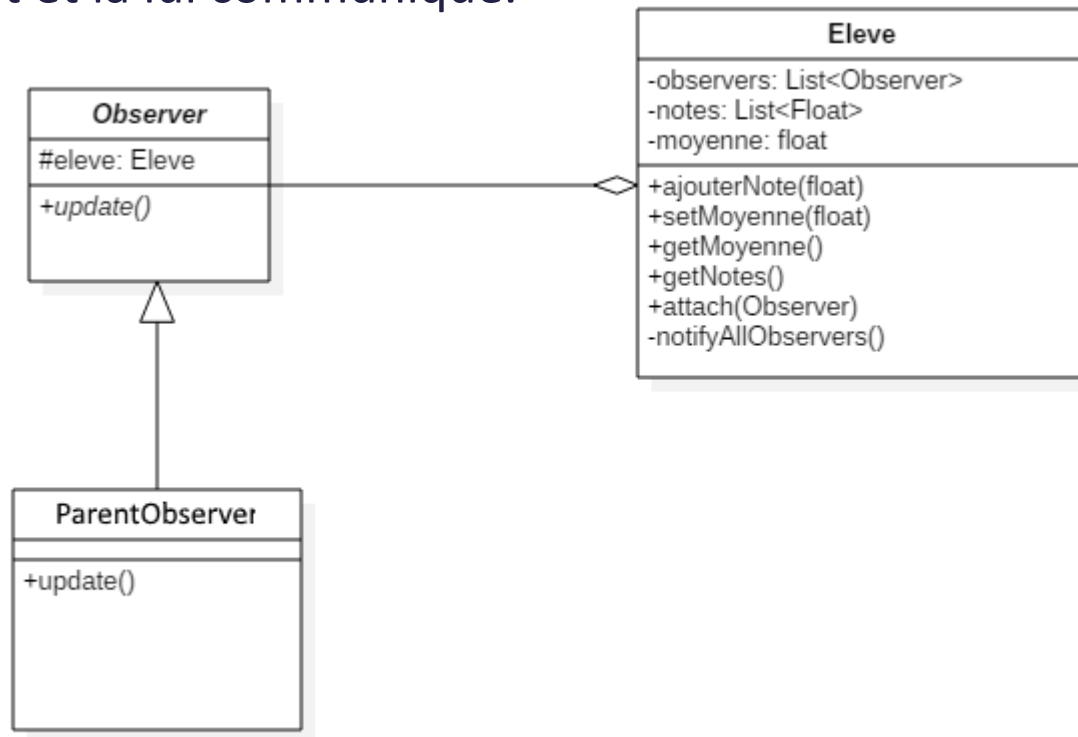
- On utilise le pattern Observer quand on doit gérer des événements.
- Dans une classe qui doit déclencher des événements, on ajoute:
 - En attribut : une liste d'Observateurs
 - Une méthode permettant d'ajouter un Observateur dans la liste
 - Une méthode permettant d'envoyer un signal à tous ses observateurs.
- "Observateur" est une classe abstraite avec une méthode signal, dont héritent des observateurs "concrets" qui implémentent cette méthode.
- Quand l'état de la classe change elle doit envoyer un signal à tout ses observateurs qui doivent effectuer l'action nécessaire en fonction du nouvel état de la classe.

III . Le Design Pattern Observer



III . Le Design Pattern Observer - Exemple

- Nous avons une classe élève à laquelle on peut rajouter des notes. A chaque nouvelle note le parent est notifié et il recalcule la moyenne de son enfant et la lui communique.



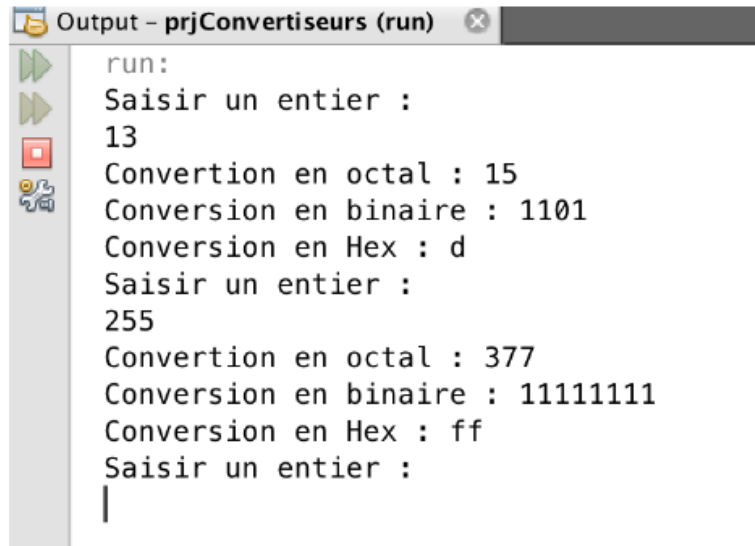
III . Le Design Pattern Observer - Exemple

```
public class Enfant {  
  
    private List<Observer> observers;  
    private List<Float> notes;  
    private float moyenne;  
  
    public Enfant() {  
        observers = new ArrayList<Observer>();  
        notes = new ArrayList<Float>();  
    }  
  
    public void ajouterNote(float note) {  
        notes.add(note);  
        notifyAllObservers();  
    }  
  
    public void setMoyenne(float moyenne) {  
        this.moyenne = moyenne;  
    }  
  
    public float getMoyenne() {  
        return moyenne;  
    }  
  
    public List<Float> getNotes() {  
        return notes;  
    }  
  
    public void attach(Observer observer){  
        observers.add(observer);  
    }  
  
    private void notifyAllObservers() {  
        for (Observer observer : observers) {  
            observer.update();  
        }  
    }  
}
```

```
public abstract class Observer {  
    protected Enfant enfant;  
    public abstract void update();  
}  
  
public class ParentObserver extends Observer{  
  
    private Enfant enfant;  
  
    public notesObserver(Enfant enfant){  
        this.enfant = enfant;  
        this.enfant.attach(this);  
    }  
  
    @Override  
    public void update() {  
  
        float moyenne = 0;  
  
        for(float note : enfant.getNotes()) {  
            moyenne += note;  
        }  
  
        moyenne /= enfant.getNotes().size();  
  
        enfant.setMoyenne(moyenne);  
    }  
}
```


III . Le Design Pattern Observer - Exercices

1. Un message sur la console demande de saisir un entier (de manière répétitive dans une boucle infinie). A chaque saisie, un objet observé contenant la valeur est mis à jour. Trois observateurs de cet objet sont notifiés, et affichent sur la console les conversions, hexadécimale, octale et binaire du nombre saisi.



```
run:
Saisir un entier :
13
Conversion en octal : 15
Conversion en binaire : 1101
Conversion en Hex : d
Saisir un entier :
255
Conversion en octal : 377
Conversion en binaire : 11111111
Conversion en Hex : ff
Saisir un entier :
|
```

III . Le Design Pattern Observer - Exercices

2. Une station météo maintient une liste des données météorologiques (températures et les taux d'humidité). Ces données seront exploitées par 2 affichages différents :

- Affichage de l'humidité et de la température maximale
- Affichage de la moyenne de température et de l'humidité

Chaque ajout d'une nouvelle données provoquera la mise à jour automatique des affichages.

Aperçu du chapitre

I. Strategy

II. Composite

III. Observer

IV. Abstract Factory

IV . Le Design Pattern Abstract Factory - Mise en situation

- Consigne: Nous avons une classe ComputerFactory avec une méthode static qui renvoie une instantiation du type de Computer passé en argument. Il existe deux types de Computer: PC et Server.

```
public abstract class Computer {  
  
    public abstract String getRAM();  
    public abstract String getHDD();  
    public abstract String getCPU();  
  
    @Override  
    public String toString(){  
        return "RAM= "+this.getRAM()+"", HDD="+this.getHDD()+"", CPU="+this.getCPU();  
    }  
}
```

- Remarquez que la classe Computer est abstraite car un ordinateur seul ne peut pas exister, c'est obligatoirement soit un PC soit un serveur.

IV . Le Design Pattern Abstract Factory - Mise en situation

```
public class PC extends Computer {  
  
    private String ram;  
    private String hdd;  
    private String cpu;  
  
    public PC(String ram, String hdd, String cpu){  
        this.ram=ram;  
        this.hdd=hdd;  
        this.cpu=cpu;  
    }  
  
    @Override  
    public String getRAM() {  
        return this.ram;  
    }  
  
    @Override  
    public String getHDD() {  
        return this.hdd;  
    }  
  
    @Override  
    public String getCPU() {  
        return this.cpu;  
    }  
  
    public void playSound() {  
        //.....  
    }  
  
    // ... autres méthodes de spécialisation pc  
}
```

```
public class Server extends Computer {  
  
    private String ram;  
    private String hdd;  
    private String cpu;  
  
    public Server(String ram, String hdd, String cpu){  
        this.ram=ram;  
        this.hdd=hdd;  
        this.cpu=cpu;  
    }  
  
    @Override  
    public String getRAM() {  
        return this.ram;  
    }  
  
    @Override  
    public String getHDD() {  
        return this.hdd;  
    }  
  
    @Override  
    public String getCPU() {  
        return this.cpu;  
    }  
  
    public void handleHttpCalls() {  
        //.....  
    }  
  
    // ... autres méthodes de spécialisation server  
}
```

IV . Le Design Pattern Abstract Factory - Mise en situation

```
public class ComputerFactory {  
    public static Computer createComputer(ComputerType computerType) {  
        switch(computerType) {  
            case PC: return new PC();  
            case SERVER: return new Server();  
            default: throw new RuntimeException("Ce type de computer n'existe pas");  
        }  
    }  
}
```

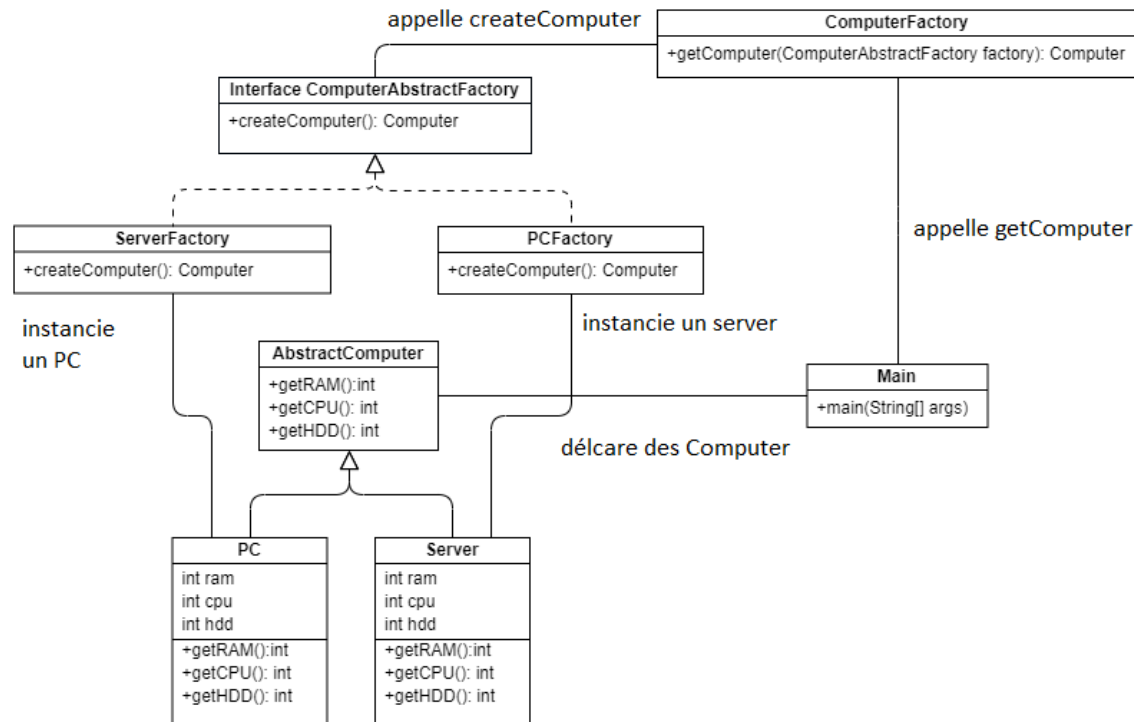
- Utiliser un switch ne nous donne pas assez de contrôle sur la création des Computer, imaginons qu'une série d'action différentes doivent être entreprise selon le type, on se retrouverait vite avec un switch volumineux.
- Gardons en tête le principe SOLID, un job => une classe. Donc il nous faudrait séparer le ComputerFactory en un PCFactory et un ServerFactory.

IV . Le Design Pattern Abstract Factory - Mise en situation

- Il y a un autre principe non respecté : c'est le Dependency Inversion Principe. Il consiste à dire que **les objets de forte valeur métier ne doivent pas dépendre des objets de faible valeur métier**. Sur notre exemple la création d'un Computer (forte valeur) dépend malheureusement de l'énumération ComputerType (faible valeur).

IV . Le Design Pattern Abstract Factory - Exemple

- Le pattern Abstract Factory, dérivé du pattern Factory, apporte la solution à cette programmation discordante.



IV . Le Design Pattern Abstract Factory - Exemple

- Création de la classe abstraite **Computer** mère de Server et PC

```
public abstract class Computer {  
  
    public abstract String getRAM();  
    public abstract String getHDD();  
    public abstract String getCPU();  
  
    @Override  
    public String toString(){  
        return "RAM= "+this.getRAM()+" , HDD="+this.getHDD()+" , CPU="+this.getCPU();  
    }  
}
```

IV . Le Design Pattern Abstract Factory - Exemple

```
public interface ComputerAbstractFactory {  
  
    public Computer createComputer();  
  
}
```

```
public class ComputerFactory {  
  
    public static Computer getComputer(ComputerAbstractFactory factory){  
        return factory.createComputer();  
    }  
}
```

```
public class ServerFactory implements ComputerAbstractFactory {  
  
    private String ram;  
    private String hdd;  
    private String cpu;  
  
    public ServerFactory(String ram, String hdd, String cpu){  
        this.ram=ram;  
        this.hdd=hdd;  
        this.cpu=cpu;  
    }  
  
    @Override  
    public Computer createComputer() {  
        return new Server(ram,hdd,cpu);  
    }  
}
```

```
public class PCFactory implements ComputerAbstractFactory {  
  
    private String ram;  
    private String hdd;  
    private String cpu;  
  
    public PCFactory(String ram, String hdd, String cpu){  
        this.ram=ram;  
        this.hdd=hdd;  
        this.cpu=cpu;  
    }  
  
    @Override  
    public Computer createComputer() {  
        return new PC(ram,hdd,cpu);  
    }  
}
```

IV . Le Design Pattern Abstract Factory - Exemple

```
class Main {  
  
    public static void main(String[] args) {  
        Computer pc = ComputerFactory.getComputer(new PCFactory("2 GB", "500 GB", "2.4 GHz"));  
        Computer server = ComputerFactory.getComputer(new ServerFactory("16 GB", "1 TB", "2.9  
        GHz"));  
        System.out.println("AbstractFactory PC Config: "+pc);  
        System.out.println("AbstractFactory Server Config: "+server);  
    }  
}
```

IV . Le Design Pattern Abstract Factory - Exercice

- Dans un contexte d'une invasion alien vous devez créer 2 usines aux total, une pour les humains et une pour les aliens. L'unité de chaque camps a des points communs et une spécialisation qui la différencie.
 - Unité humaine:
 - Attaque
 - Défense
 - Une liste d'équipement vide de base
 - Bonus de moral
 - Unité alien :
 - Attaque
 - Défense
 - Une liste d'équipement vide de base
 - Téléportation

IV . Le Design Pattern Abstract Factory - Exercice

- Maintenant les camps ont 2 types d'unité: soldat et blindé. Un soldat a la capacité unique de se cacher et un blindé d'accélérer. Il y aura au total 4 usines dans votre programme.