

L'objet

Définitions :

Il est défini par :

- son ID (obligatoire et inaccessible)
- Données (pas obligatoires)
- Traitements(un minimum obligatoire fourni par héritage de Object en JAVA)

Référence : c'est quand un obj connaît un autre (il possède une référence vers cet autre obj)

Les obj communiquent par échanges de messages pour effectuer des traitements complexes

La **POO**, c'est découper un programme en petites entités qui communiquent entre elles via leur références afin d'effectuer un traitement.

REGLES : en objet, il est interdit que des objets partagent les mêmes traitements et les mêmes données :

- **Limiter les interactions** entre les objets
- Limiter le nombre d'objet pour **optimiser la mémoire**

Responsabilité et encapsulation :

l'objet est **responsable des traitements qu'il propose** donc l'objet **dispose de tous les éléments pour les réaliser** (i.e. il ne redirige pas la demande vers un autre objet).

Traitements = suites d'instructions (-> éléments) + **paramètres d'entrés** (+ données en param ou que possède l'obj en son sein) + **résultats**

En poo **les traitements dépendent des données**(s'il en a) sinon cest de la prog fonctionnelle. ex :

- calculatrice simple = prog fonctionnelle
- calculatrice à mémoire = POO

Les objets doivent verrouiller leurs données nécessaires aux traitements qu'ils proposent d'où la nécessité d'encapsulation.

- **Responsabilité** = l'obj a tout pour réaliser ses traitements
- **Encapsulation** = Si l'obj utilise ses données pour réaliser ses traitements il doit les protéger !

Couplage et cohérence :

Qu'est-ce que le couplage ?

Pour réaliser un traitement, l'objet peut demander l'aide d'autres obj. => **Plus on a besoins "d'amis" plus on est couplé !**

ATTENTION : En POO, il faut limiter les couplages car si un objet couplé vient à être modifier ou à disparaître le traitement n'est plus possible

=> **objectif = couplage faible**

Qu'est-ce que la cohérence ?

Un objet ne peut pas faire tout et n'importe quoi, **i.e.** plein de traitements sans rapports entre eux au sein d'un même objet.

Etre incohérent , c'est proposer des traitements sans rapport **traitements aves des rapports**, ex : qui partagent les mm données.

Regles : Si un objet peut etre coupé en deux (donnés + traitements) + (donnés + traitements) , alors l'objet initiale n'était pas cohérent!

=> **objectif = forte cohérence**

Analyse d'objet : Sens critique -> est-ce que mon objet est bon?

CF SCHEMA 1

Cycle de vie d'un objet :

1. **Création (new)**: N'importe quel obj peut créer d'autres objets . **L'objet créateur est le seul à avoir la référence de l'objet créé!** Si d'autres obj ont besoin de l'objet, la référence doit être transmise.
2. **Reception de messages + réalises traitements**: traitement synchrone ou asynchrone
3. **Destruction** : En JAVA si un objet ne peut plus recevoir de message (plus personne n'a de référence vers lui), alors JAVA le met au Garbage Collector.

Sythèse 1 :

- **Objet** = #ID , (data), traitement
- **Référence** : On référence des objets (en JAVA on peut aussi référencer les types primitifs)
- **Responsabilité et cohérence**
- **Couplage** : Responsable mais pas seul(besoin d'autres obj pour effectuer des traitements)
- **Cohérence** : On ne peut pas couper en deux un objet (data,traitements)

Les Classes

Le JAVA est un **langage orienté objet**,le dev décrit des classes d'objet(jamais un seul objet)

Les classes ont 2 rôles :

- **Moule/fabrique/patron de conception** : un objet est construit à partir d'une seule classe = **INSTANCIATION**
- **Relation de typage** : est-ce que l'on voit que l'objet a été créé avec une classe.

voir schéma 2:

Qu'est-ce qu'une classe? => un moule (new) = relation statique/figée => un contrat (conformité) = dynamique(es-tu conforme ?)

La classe comme contrat/moule (conformité)

- **donnée** : `visibilité(public ...) Type(MaClasse) nom(toto);` => conforme : l'obj possède les données définies par la classe
- **traitement** : `visibilité TypeDeRetour nom(Type param1, Type param2, ...);` => conforme : l'objet est responsable des traitements définis par la classe.

Ducktyping : ex : typescripE.Evans 2003 = Blue Book Vaughn = Red Bookt, "si tu fais coin coin => alors t'es un canard", le ducktyping regarde que du côté objet

JAVA : Qui est ton créateur ? => JAVA regarde côté classe.

Conformité = = TYPAGE :

schéma 3:

Double controle du typage pour les langages compilés. On a une phase de compilation code-> code compilé -> run. le rôle du typage est de :

1. Préviens les err
2. Aide l'IDE

ex : si on veut des entier en attribut +: 2 solutions :

1. On ajoute une exception dans le constructeur de la classe
2. On Créer se propre classe PositivInt

Il ne faut pas faire de try/catch : ClearCode = lire le livre. Le mieux c'est que le programme plante.

Visibilité et encapsulation :

1. Pour les date : en JAVA tout mettre en private ou en public final s'il doit y avoir que du read only dessus.
2. Pour les traitements : Les traitements qui modifie les données : si **c'est voulu** on met en ****public**, **sinon** on met en **private**.

L'héritage :

C'est une relation entre deux classes qui a un impact sur les objets instances des classes en relation.

ATTENTION ne **JAMAIS** écrire : "mon objet hérite de..." !!! **L'héritage est une relatioentre classes et non entre objet!**

voir schéma 4:

A **extends** B A est classe **filie** B est **super classe / mère** obj A **conforme** à classe B obj B **non conforme** à classe A

Principe de subsitution :

- Si j'ai besoin d'un B => on peut me donner un A car A est conforme à la classe B (tout les A save faire foo())

exemple de violation d'encapsulation :

```
public class Shape {
    private List<Point> pointList;
    public Shape(){
        pointList = new ArrayList<Point>();
    }
    public List<Point> getPointList(){
        return pointList; // Constructeur par copie
    }
}
```

ici on a donné une référence directe vers les données privé de l'objet instancié par shape. Du coup quelqu'un peut appeler les méthode de Liste sur pointList sans passer par l'objet qui contient pointList. => **pointlist devient donc mutable = violation d'encapsulation**

correction :

```
public class Shape {
    private List<Point> pointList;
    public Shape(){
        pointList = new ArrayList<Point>();
    }
    public List<Point> getPointList(){
        return new ArrayList<Point> (pointList); // Constructeur par copie
    }
}
```

Polymorphisme et surcharge :

Le **contrat des classe** (type) ne porte **jamais sur le code**. C'est uniquement sur les éléments de typage.

Polymorphisme : changer le code dans une sous classe.

Surcharge : changer le type / ajouter des contraintes de typages

schéma 5:

cast = dire au compilateur, tkt c'est un rectangle, au pire le prog pète au 2ème check de typage. on lui indique nous même le type entre parenthèse ex: (Rectangle)

Règle :

Covariance : On a le droit de sous typer des paramètres

Le principe de la contravariance = pas au programme, on inverse les paramètres d'héritage en entrée avec ceux en sortie.

Les classes génériques = on peut donner n'importe quel type et ce type sera le type utiliser dans la classe.

```
public T transform(T in);
<T extends Shape>
```

Quand faut-il utiliser l'héritage?

1. Si on est en train de dupliquer du code entre deux classes
2. Si on veut réutiliser du code sans le changer. mais **attention au abus** ex: `Shape extends List =>` Tout ce qui est `public` dans List le serait aussi dans Shape!
3. Le plus intéressant : quand on est en train de définir un contrat qui **devra** être complété par un autre dev.

Interface

Délégation, classes et interfaces :

Rappels :

- Cas 1 : Limiter la redondance de code.
- Cas 2 : Réutiliser, customiser, adapter du code existant.
- Cas 3 : Faire en sorte qu'un autre développeur complète notre travail.

Peut-on réaliser le cas 2 sans héritage ? On peut créer une instance de la classe A dans la classe B et appeler les fonctions de la classe A à travers l'instance créée. C'est la délégation.

avantage/inconveniet	Héritage	Délégation
+	- moins d'objet, moins d'appels de méthodes, "moins de dépendances directes"	On ne dépend pas du code de réalisation grâce à l'interface. On perd en performance mais on gagne en efficacité.
-		- plus d'objets, plus d'appel de méthodes, pas de substituabilité

Pour héritage : 1 objet et 2 flèches. Pour délégation : 2 objets et 4 flèches.

En terme de compilation de quel classe le code dépend ?

- héritage : b dépend du code et A dépend de A.
- Délégation : Bdel dépend du code et dépend de A et A dépend du code.

Peut-on appliquer le principe de substitution, si besoin d'un B peut-on avoir un bdel ? Tous les Bdel sont inclus dans la classe des B ? NON.

- Si on a un délégué alors on ne peut pas appliquer le principe de substitution.

Interface

Définition : c'est une définition d'un ensemble "cohérent" de méthodes (signature) : principe de base : pas de code. On dissocie de la **définition** des méthodes(**signatures**), de la façon dont elles sont **implémentées**(le

code).

```
public interface BItf{
    public void f();
}
```

J'ai besoin de f() peut importe la façon dont c'est codé. J'ai B qui réalise f() {...}

```
public class B implements BItf {
    public void f(){
        // code ...
    }
}

public class BHeritage extends B{
    public void f(){
        super f(); // j'utilise le code de B
    }
}

public class Bdel {
    private B b;
    public void f(){
        b.f();
    }
}
```

Peut t-on déléguer le new ? Je ne veux plus faire de new. Tout objet est une usine et peut donc faire new.

Grâce à la délégation on ne dépend pas du code.

```
public class Point{
    private int x,y;
    getter;
}
```

Je veux me séparer de la classe Point.

```
public interface PointFactoryIt{
    public Point createPoint(int x, int y);
}

public class PointFactory implements PointFactoryIt{
    public Point createPoint(int x, int y){
        return new Point();
    }
}
```

```

    }
}

PointFactoryIt pf = new PointFactory();
pf.createPoint();

```

J'ai un bout de code qui demande à faire des tris :

- mode classique : j'ai un objet qui fait le tri.

Si je veux utiliser des modes de tri différents : benchmark = tester les performances des algo de tris.

```

public class BenchmarkTri extends Tri{
    public int bench(){
        int times 0;
        for (/*start*/){
            tri();
            time=stop-start;
        }
        return time/N
    }
} // dans la classe tri il y a sort

public interface SortIt {
    public String[] sort(String[] in);
}

public class BenchMark{
    private SortAlgo SortAlgo;
    public int bench(){
        //code
    }
}

public void setSortAlgo(SortIt nouvelAlgo){
    this.SortAlgo = nouvelAlgo;
}

```

Le benchmark il va appeler le tri, un objet, donc il a besoin d'avoir dans sa base de donnée l'id du tri. Quand on fait de la délégation quand est-ce qu'il à l'id ?

```

public class PNJ{
    private SortAlgo sa;
    public BM(SortAlgo sa){
        this.sa=sa;
    }
    public void findPlayer(){
        //code
    }
}

```

Autres possibilité je fais un setter

```
public class BM{
    private SortAlgo sa;
    public void setSortAlgo(SortAlgo sa){
        this.sa=sa;
    }
}
```

PNJ = personnage non joueur, et on cherche un joueur avec findPlayer : quand on crée un PNJ on lui affecte un algo, par délégation on change l'algo, pour trouver le joueur.

```
public class PNJ extends PlayerFinder{
    public void findAndKill(){
        findPlayer();
    }
    public void findAndRun(){
        findPlayer();
    }
}
```

Je délègue je peux faire une interface du findPlayer.

```
public interface PlayerFinderItf{
    public void findPlayer();
}
```

Trois solutions:

- Je fais un délégué mais j'ai pas compris l'intérêt

```
public class PNJ{
    private PlayerFinderItf= new PlayerFinder();
}
```

Aucun intérêt, on a délégué par pnj.

- deuxième solution : je passe le délégué à la construction de l'objet : je lui donne le délégué à la l'instanciation

```
public class PNJ{
    public PNJ(PlayerFinder pf){
        this.pf=pf;
    }
}
```



```
}
}
```

Ça manque de fluidité et on ne peut plus changer le délégué.

- troisième solution : je lui donne peut importe quand je le veux.

```
public class PNJ{
    public void setPlayerFinder({});
}
```

Là on accepte de donner le délégué au moment où l'on veut. On peut avoir plusieurs délégué par pnj.

Les PNJ ont besoin de ces délégués donc il faut le leur donné à un moment donné.

Lire Design Patterns : Going of Four (GOF) : il explique les design pattern : Erik Gamma (eclipse). Lire les premières pages (ils donnent la golden rule), ils donnent des cas concret : create with no new.

Il explique que l'héritage est pas ouf et que le mieux c'est la délégation quand il faut réutiliser du code.

Pattern observer : que de la délégation.

```
public class PNJ{
    private PlayerFinderImpl pf ;
    pf.findPlayer();
}
```

PNj dépend de playerFinderImpl pour la compilation.

```
public class pNJ{
    private PFITF pf;
    setPf();
}
```

L'interface ne dépend de rien , donc il compile tout le temps, pour PNJ, il faut l'interface, l'implémentation a besoin de l'interface, PNJ ne dépend plus de l'implémentation.

package = ensemble de classes, modules.

Grâce à l'interface on inverse la dépendance. Avant je dépendait d'un code maintenant le code dépend de moi.

```
public class PNJ extends Savable{
    save();
}
```

Je réutilise du code avec l'héritage, le PNJ dépend de la base de donnée. On peut pas compiler le PNJ tant que la base de donnée n'est pas faite, on va donc inverser la dépendance.

```
public interface SavableItf{
    public void save();
}
public class PNJ{
    private SavableItf saver;
    setSavable();
}
public class SavableSQL implements SavableItf{
    save(){/*SQL*/};
}
```

Le point fort de la délégation c'est qu'il est capable de jouer avec les classes, les liens deviennent dynamiques, en runtime je peux switcher le code. Dans le cas où je dois réutiliser le code, le mieux c'est la délégation = interface.

Test et lint

Tests et Clean

Rappels : Cohérence et couplage :

Si on a besoin de faire un tri dans un catalogue : On commence à coder un algo de tri dans la classe de catalogue alors on manque de cohérence!

Est-ce cohérent d'avoir un objet qui va contrôler le tri et un autre le catalogue ? Oui => Donc on divise la class : on doit avoir une classe catalogue et une classe tri.

Doit-on faire hériter la classe catalogue de la classe tri ? C'est trop coupler dans ce cas là. => Donc on pense à la délégation : mais le couplage possède alors une référence vers tri : c'est mieux mais c'est encore trop fort.

Peut-on facilement changer de déléguer ? Non => finalement le catalogue doit dépendre de l'interface et l'implémenter dans le catalogue.

Ainsi, on donne une valeur, une qualité de conception.

Les tests :

Testeur : Amazonterq Crowd Testing : tu leur donne ton code et ils le vérifient. On cherche que ça marche pas : on cherche les erreurs. On doit vérifier que ça ne marche pas. Il faut avoir une définition de "ça marche" et montrer l'inverse ! **Oracle**.

Quand a-t-on terminé de tester ? Dans l'absolu jamais, mais on va essayer de définir le périmètre des tests et on va le partitionner en classes d'équivalence. 4 partitions infini : Avec 4 tests je considère que j'ai fais ma

partition.

```
int add(int a, int b);
```

Je considère que tester 2 nombres positifs, 2 nombres négatifs, 1 + et 1 - et vice versa c'est la même chose pour chacun. Autres partition : 3 : valeurs trop grande en + et trop grandes en - et les bonnes valeurs.

```
f(){
  // 11
  // 12
  // 13
}
```

On peut faire les tests par lignes, on a les tests qui passent par chaque lignes. Donner la couverture : donner la partition. On doit vérifier que tu es passé par toutes les lignes.

Pour tester :

- Avoir l'Oracle
- Définir votre partition
- 1 Test / partitions

```
Tester String[] sort(String[])
boolean oracleTri(String[] sortest){
  for (int i =0; i< sortest.size()-1;i++){
    if(sortest[i]>sortest[i+1]){
      return false;
    }
  }
  return true;
}
```

Sort : tri dans l'ordre alphabétique. L'oracle : coder l'oracle = coder la fonction ?

Il faut trouver des points au hasard : Je construis un tableau au hasard et je le teste. L'oracle teste si ton code fonctionne. Il aurait partitionné en 3 tests list et 3 tests pour les chaînes de caractères.

Clean code

Du bon code c'est quoi ? Le moins il y a de WTF, le mieux c'est, c'est un peu subjectif.

Identifier les règles d'un code de qualité, quelque soit le langage.

Clean code : un bon code est lisible. rabbitmq-java-client : client java de rabbitmq.

Nommage des variables

Le code est une histoire, chaque nom fait parti d'une histoire. Je lis la variable, ça doit être prononçable. cherchable.

Nom de développeurs, termes de métiers, du contexte.

Faire attention aux magic value : expliquer les valeurs pourquoi on choisit celle là. Il faut nommer la constante et l'expliquer.

Les fonctions

Pas plus de 3 paramètres , sinon créé un objet options qui contient tout le nécessaire

Les commentaires

Un commentaire n'améliore pas le code. S'il est lisible alors ne pas le commenté, il n'en a pas besoin.

Bon commentaires :

- licences sont nécessaires
- avertissement
- todo
- javadoc public api : uniquement sur ce qui est utiliser depuis l'extérieur.

Supprimer les blocs qui expliquent le code.

Lint

Encode les règles, et explique ce qui n'est pas bien dans le code.

Bad smell = il regarde si il y a des choses qui peuvent indiquer des défauts.

Katta potter <http://codingdojo.org/kata/Potter/>

Domain Driven Design

E.Evans 2003 = Blue Book Vaughn = Red Book

Principe fondamentale : => le code fait autorité (la vérité se trouve dans le code)

3 endroits où peut se trouver la vérité :

1. Le cahier des charges (besoin exprimer par le owner (celui qui cmd l'appli)) il donne les spécifications présentes /futures du logiciel (ici le codes s'appuie sur les données)

=> on veut un package qu icontient le code et on sait éxectement ce que fait le code

=> On concentre dans le **couche DOMAINE** tout ce qui fait le métier

=> le **dommaine = l'essentiel de l'appli** donc dedons il n'y a pas d'IMH, de réseau, de save etc...

=> Si on commence par l'UML on reste trop abstrait

=> Si on commence par la BDD, on travaille uniquement sur quelque chose de "static"

exemple : application de gestion de compétition

concept :

- Joueur -> il ne change pas durant la compétition (nom, prénom, club ...) => il est => **IMMUTABLE**
- match -> il ya debut, une fin, un score, un vainqueur, des participants)
- compétition -> ouverte à l'inscription, 1 grand gagnant, on peut jouer des match, il ya plusieurs joueurs.

On commence par quoi? => on peut par joueur

```
public class Joueur{  
    public final String nom, prenom;  
}
```

Mais Ici on peut mettre "1234" comme nom... => 2 solutions :

1. soit on pose des conditions d'exceptions dans le constructeur
2. soit on considère que String est un type trop "lâche" et que **c'est pas assez métier** donc on **créer sa propre classe** ex : ChaîneAlfabétique.

on aura donc :

```
public class Joueur{  
    public final ChaîneAlfabétique nom, prenom;  
  
    public Joueur(ChaîneAlfabétique nom, ChaîneAlfabétique prenom){  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}  
  
public class ChaîneAlfabétique{  
    public final String valeur;  
  
    public ChaîneAlfabétique(String val){  
        if(regexHere){ //[a-zA-Z]* plus inclure -, \, _ etc  
            this.valeur = val;  
        }else{  
            runtime exception;  
        }  
    }  
}  
} // peut etre aussi limiter le nb de char pour opti la BDD
```

Les Value Object :

définition : ils sont définies par la valeur de leurs propriétés! **Ils sont donc IMMUTABLES**

ATTENTION => On doit **redéfinir .equals()**

FAUX :

```

public class Joueur{

    public boolean equals(Object other){
        if!(other instanceof Joueur){
            //ne fonctionne pas dans le cas de l'héritage
            //si B extends A alors b.equals(A) return false meme si b est un A
            //et qu'il contient des données égale à un A.
            return false;
        }else{
            Joueur otherJoueur = (Joueur) other;
            return nom.equals(otherJoueur.nom) &&
            prenom.equals(otherJoueur.prenom);
        }
    }
}

```

On a deux classes avec la méthode equals : une classe A et une classe B qui hérite de A. a.equals(b) on appelle la méthode de la classe A, si je fais b.equals(a) j'appelle la méthode de b mais là on commence par tester le type donc a dégage car c'est une classe qui hérite.

On doit faire attention quand on crée les valueobject, utiliser une factory : si il est d ?

Les Entity

Le cas des matchs : le score peut évoluer au cours du match -> il évolue => il est **MUTABLE**

```

public class Match{
    public final Joueur j1, j2;
    private EtatDuMatch etatDuMatch;
    private Score nbPointJ1, nbPointJ2;

    public void demarer(){ }

    public void majScore(int numJoueur, Score pointJoueur){
        if(numJoueur == n...) => flag pramameter
    }
}

```

ATTENTION : Si un paramètre est une condition de **if** => alors c'est un flag parameter. => on doit faire une deuxième méthode !

Retour aux entity : On ne veut pas d'entité avec le même ID => l'ID sera donc donné par le championnat !

Les Aggregate

=> Dès que l'on a des entité, il faut qu'il y ai un objet qui rassemble **les entités qui ont le même sens**

=> cette objet c'est l'**AGGREGATE**. C'est lui qui **possède, protège** et donne l'**ID** des entités.

```
public class Competition{
    set<Joueur> setJoueur;
    set<Match> setMatch;

    public void inscrire(Joueur j){}

    public void fermerInscription(){ new Match() //on creer les match}
}
```

Pour changer les score si on fait `c.getMatch(1).majScore()`, on casse l'encapsulation => en effet `getMatch()` donne une référence d'un match appartenant a l'aggregate , ce qui fait que l'aggregate n'est plus responsable des ses propres match!!!!

=> on doit faire une méthode `c.majScore(MajScore option)` qui prend un objet qui sert d'option de paramètre (il contient tout les infos dont compétition c a besoin pour mettre à jour le score d'un match en particulier).

=> Quand est-ce qu'on save? Avant quand la BDD servait de vérité, on sauvegardait à chaque modification. En DDD, c'est le code qui a la main => on peut le faire de manière sync ou async quand on veut et comme on veut en fonction du cahier des charges.

En DDD => Nouveau contexte = Nouvelle app (pas de réutilisation des classes).

Concept du DDD = Tactical Pattern

- **Value Object** : immutable, **equals by value**, typage plus fort que celui de java (ex: entier positif). => pas de problème d'encapsulation
- **Entity** : possède un état(state) donc peut évoluer = mutable , **equals by ID**
- **Aggregate** : C'est un **composite**, il contient des composants(les Entity). => il sert à la gestion des Entity(créer l'eu id, etc...), => il permet les save.

Aggregate, Entity, VO + Services

Les questions à se poser :

1. Qu'est-ce qui ne bouge pas? => Value Object
2. Qu'est-ce qui bouge? => Entity
3. Est-ce qu'il y a des ensembles? =>Aggregate

exemple: de l'échiquier

A	B	C	D	E	F	G	H
1							
2							

A	B	C	D	E	F	G	H
3							
4							
5							
6							
7							
8							

On va avoir des aggregate et la technique move qui va permettre de bouger la pièce.

Aggregate	Entity	ValueObject
- Game	-case (Piece p, Location loc =ID)	- Location
-Position		- Joueur
		- Color
		- LegalPosition
		- Move

```

public class Location{
    private char column;
    private int line;
    public Location(char col, int lin){
        if(col !∈ ["A-H"]){
            throw new IllegalParameters();
        }
        else{
            this.column=col;
        }
        if(lin >0 && lin <= 8){
            this.column=col;
        }
        else{
            throw new IllegalParameters();
        }
    }
}

public class Game{
    public Game(){}
    public void move(Location Pion, Location step){}
}

```


Piece :

- Sa couleur Ne bouge pas
- Sa position Bouge
- Son importance Bouge et ne bouge pas
- Vivant ou morte Bouge
- Son mouvement Bouge et ne bouge pas : ça dépend la vision des choses.

Position va bouger c'est sûr, est-ce que une pièce est définie pas sa position ?

```
public class Piece{
    private Location loc;

    public void move(Location newLoc){
        loc = new Loc;
    }
}
```

=> Application à la Java Bean!!! A éviter =>Une méthode va dans la classe qui peut OBTENIR TOUTES LES INFOS! Ici par exemple dans la classe Game => move(Location from, Location to)

Applying UML : Pages intéressantes sur les patterns GASP : méthode va dans la classe qui possède ou peut obtenir toutes les informations pour exécuter la méthode.

Le mouvement : si je suis ça est-ce que j'ai le droit de faire ça. ON doit mettre le move abstract, et le développer suivant le type de la pièce. Piece avec Knight, Queen, Bishop, etc...

On divise la position en 2, on split en LegalMove et move.

```
public abstract class Piece{
    public abstract boolean isLegalMove(Location from, Location to);
}

public class King extends Piece{
    public boolean isLegalMove(Location from, Location to){
        if (Math.abs(from.getline()-to.getline())>1){
            return false;
        }
        return true;
    }
}

// hidden entity en dessous dans la map:

public class Game{
    Map<Location,piece> positionPiece;
}

//Il vaut mieux créer une classe case
```

```

public class Case{
    Piece p; //change, représente l'état de l'entity
    Location loc; // l'emplacement de la case ne change pas
    // et deux case ne peuvent pas avoir la même loc => ID unique
}

//La classe game devient donc:

public class Game{
    List<Case> caseList;
    public void move(Location from, Location to){
        Case case= getCaseFromLocation(from);
        Piece pfrom= case.getPiece();
        boolean islegal=p.isLegalMove(from,to);
        Case[] intermediaryCase= => fonction stateless;
        //fct statless = getIntermedaryLoc(Loc from, Loc to);
    }
}

public class Case{
    Piece p;
    Location l;

}

public List<Location> getIntermediaryLocation(Location from, Location to){

}

public List<Location> getAccessibleCase(Piece p,Location at){

}

```

Réification : donner le caractère d'une chose à.

Je récupère la case de from, prend la pièce, regarde si le coup que je veux faire est légal et ensuite on regarde si on prend des pièces sur le chemin.

Obtenir les cases intermédiaires : pas besoin d'état. Des fonctions qui n'ont pas du tout besoin d'état, ils ont juste besoin des trucs en paramètre en entrée et ces paramètres sont des ValueObject.

getIntermediaryLocation et getAccessibleCase sont stateless. Intérêt des fonctions stateless : décomposition du code, elles n'ont pas besoin du changement du game, ce sont des services, réintroduction des lambda = fonction qui n'a besoin de rien, qui se suffit en elle-même, dans le DDD ce sont des services. On les mets dans des nouvelles classes, on fait une classe par service.

TACTIQUES EN PRATIQUE :

1. Penser au Value object 2. Voir l'aggregate avec les méthode haut niveau qui prennent en parametre des Value object PAS d'entity, Mais on peut utiliser des entity à l'intérieur 3. On trouve les entity dont on a besoin

Stateless : qui n'a pas d'état dans les paramètres, on appelle ça un **service** en poo (ex: fonction lambda). Il est conseillé de faire : **1 service = 1 classe**

- Value Object (Services)
- Aggregate méthode métier paramètres VO
- Structuration aggregate => entity

Factory

```
public class Game{
    Game(){
        // On génère la liste de toutes les cases,
        // y mettre les pièces etc etc = créations de 200/250 objets.
        // Quand le code de construction est grand alors on le déplace
        // dans une factory
        // c'est une classe qui va construire les new.
    }
}
```

Quand le new est compliqué et qu'il nécessite trop de lignes, on sort dans une classe dite factory, classe qui va contenir que le new, cette classe va être le lieu où on va soigner le code de construction. On fait ça sur les aggregates.

Autre exploitation des Factory est pour limiter le nombre d'objets VO égaux dans la VM.

Architecture Hexagonale (La couche Infra)

La couche domain, on va coder:

- le typage qui découle des Value Object
- les règles métier = entity et aggregate.

Dans notre application il doit avoir un **package domain** : quand on ouvre un projet en DDD on a un répertoire domain où on va trouver toutes les **classes qui vont préciser le métier**. Pas d'IHM pas de graphique, pas de ligne de commande, pas de sauvegarde, que du métier. Le package **domain ne dépend de rien, aucune dépendance de classes**. Tout doit être défini à l'intérieur, **aucune dépendance = on peut tout le temps compiler les classes qui y sont**, ça doit compiler tout le temps. Le package domain ne doit pas dépendre du framework. C'est mieux si ça compile avec des versions de java pas trop récentes.

Repository

Comment faire pour **sauvegarder** ? Sauver des concepts de métier qui ne dépendent pas d'une base de donnée. Avant / ou sans approche DDD : Schéma de notre donnée (SQL etc ...) on utilise un framework pour générer des classes qui font le pont vers notre VM. On crée notre classes à nous et on fait hériter. Game.sql -> Game.java GameHibernate -> Game extends GameHibernate, on a alors la méthode save dans Game, et

qui est coder dans GameHibernate = ce code avec l'appel sql est soit dans une classe générée soit on le met dans Game. Le code de sauvegarde où le mettre ? Quel schéma ? Quelle signature de sauvegarde ?

On reprend game et on veut **sauvegarder une partie en cours c'est le repository**. Il faut **mettre le code de save dans une classe GameSaver et le chargement avec**. Comme il faut sauver, il faut penser aux **méthodes qui permettent de sauver et loader les aggregate**. Il faut faire un **findGameById = un load**, on peut faire un findGame par le nombre de coup par lequel la partie a été gagnée. **Update = met à jour une sauvegarde**.

Sans repository : Si on prend une approche sans réfléchir on peut manquer de méthode dont on a besoin.

Repository = concept DDD = définit les signatures = ce n'est pas encore le code en java c'est une interface, dans quelle couche ? C'est une interface ou classe abstraite, il va définir la signatures des méthodes, il est **dans domain** car on a pas encore le code de sauvegarde :

```
public interface GameRepository{
    public void save(Game g);
    public Game load(int gameId);
    public Game findBy();
    public void Update();
}
```

(l'implémentation sera dans la couche infrastructure, ex: GameRepositoryJson qui implémentera les méthodes de GameRepository et utilisera le format Json pour save et load).

Signature

La on reste dans la conception, on cherche d'abord la signature.

```
public class Game {
    GameRepository rep;
    public void move(Location from, Location to){
        rep.update(this);
    }
}
```

Ici on reprend le contrôle en tant que développeur on choisit quand s'effectue la sauvegarde avec rep.update(this)

Schéma ?

Maintenant quel schéma ? Quel format ? SQL ? CSV ? JSON ? etc ...

table Game :

- id

table Case :

- Game
- Location
- Piece

table Location :

- Column
- Line
- id

table Piece:

- id

Ici si on change le code alors il faut changer la base de donnée, la vérité est dans le code. La définition du schéma dans la couche infra.

Où mettre le code ?

On le met dans infra.

```
// Dans la couche domain
package domain;

public interface GameRepository{
    public void save(Game g);
    public Game load(int gameId);
}

// Dans la couche infra
package infra;

public class Game SQLRepository implements Game Repository{
    public void save(Game game);
}
```

GameRepository compile sans GameSQLRepository mais GameSQLRepository ne compile pas sans GameRepository.

L'exécution du programme

Qui / où / comment appelle le repository ? Notre décision (dans les aggregate), des les services à l'extérieur.

- Dans l'aggregate : On doit définir la loi de démeter on le fait via l'interface. On peut initialiser le game en lui donnant le repository ou on peut faire un setter de repository et on lui donne le GameRepository

```

public class Game{
    GameRepository rep
    public void setRepository(GameRepository rep ){
        this.rep=rep;
    }
    public void move(){
        // ...
        rep.update(this);
    }
}

// main

GameRepository rep = new GameSQLRepository();
Game g = new Game();
g.setRepository(rep);

```

On construit un repository. Le main est technique on le met en dehors, dans aucune des couches.

Synthèse DDD (Domain + Infrastructure)

exemple de l'agenda: ajouter un rdv etc...

Aggregate	Entity	Value Object	Factory	Repository	Service
Agenda	RDV	Date			
		sujet			
		lieu			

Value Object, Entity, Aggregate

```

public class Date{
    private int numMois, int numJour;

    public Date(int mois, int jour){
        this.numMois = mois;
        this.numJour = jour;
    }

    public int hashCode(){}
    public boolean equals(){}

}

```

ATTENTION : L'aggregate(ici Agenda) ne doit jamais donner accès à ses entity! cf schémas 6

```

public class Agenda{
    private int Id;
    private Set<Rdv> setRdv;
    private int genIdRdv = 0; //nb d'Id de Rdv créé

    public int ajouterUnRdv(Date debu, Date fin, Sujet sujet, Lieu lieu){
        // faire une création d'id pour l'entity Rdv
        Rdv rdv = new Rdv(int idRdvgénéré, Date début, Date fin, Lieu lieu);
        return idRdvgénéré;
    }
    public void modifierSujetRdv(Sujet, int IdRdv){}
}

public class Rdv{
    private int Id;
    private Date debut, fin;

    public void changerDebut(Date nouveauDebut){};
    public boolean equals(){ utiliser l id };
    public int hashCode(){utiliser l id};
}

```

Service

Idéalement c'est une méthode métier (donc elle fait partie du domain) **sans état**, elle ne stock rien et on ne peut pas la ranger dans les Agg, Entity ou VO.

ex:

```

public Date getListCrenauLibre(Set<Agenda>, Date debut, Date fin, int
duree)

```

Ici on a le droit de donner des Aggregate car ils protègent leurs entity via l'encapsulation

on voit bien que cette méthode peut être rangée nulle part car elle a besoin de plusieurs agenda.

=> public class AgendaService{ rien ici pas de propriété => pas d'état => **stateless**

```

public Date getListCrenauLibre(Set<Agenda>, Date debut, Date fin, int
duree){}

```

```

}

```

Repository

Sert à sauvegarder et à charger. Il est lié au concept de factory. Et le Repository est plutôt fait par les Aggregate.

```
package domain;

public interface AgendaRepository{
    public void save(Agenda agenda);
    public Agenda findAgendaById(int id);
}
```

```
package infra;

public class AgendaFile implements AgendaRepository{
    public void save(Agenda agenda){
        List<String>rdvDTO = agenda.getRdvDTO();
        List<Rdv> listRdv = agenda.getRdv(); // seulement si pas ref direct!
        use cpy
    }
    public Agenda findAgendaById(int id);
}

public class Agenda{
    public RDV ou String... getRdvDTO()
}
```

schéma 7 dézoom Ui, Appli, domain, infra...

CQRS (la couche application)

voir schéma 8

couche application : C'est toute la logique (Quand est-ce qu'on save? update? etc...). La couche application contient le code qui permet la manipulation efficace des concepts métiers.

ex: à chaque fois que je fais un move() -> je save() => moveAutoSave() (dans la couche appli) = servie de la couche application.

Attention : Si le code traite d'autorité/autorisation => il va dans la couche domain!

Si le code facilite(efficacité : tps de traitement et/ou rapidité de réponse(**Async**)) => C'est dans le **couche application**.

Command Query Responsibility Segregation(séparation des responsabilité entre les requêtes et les commandes)

Le CQRS est une proposition pour traiter l'Asynchronisme en utilisant le pattern command.

=> Donc ici, on ne veut pas d'objet qui soit responsable à la fois des requêtes et des commandes

=> on a donc au minimum deux objets qui sont responsables et séparés, c'est à dire qu'il ne communique pas entre eux

Query = read only **Command** = write

cf voir schéma 9

Command

On part ici d'un code "faux" => synchrone et on va les transformé en code asynchrone.

```
package application;

public class ServiceApplication{
    GameRepository gr;

    public void moveAutoSave(Location from, Location to, int gameId){
        game g = gr.getGameById(gameId);//<- Query, read only sur la bdd
        g.move(from,to);
        gr.save(g);//<- Command, write sur la bdd
    }
}
```

Voir schéma 10 : le patter command(GOF)

Penser à Mcdo => on fait une command, on nous donne un num

on peut considéré la commande comme environ une constante / un Value object

```
public class SaveGameCommand{
    private gameToSave;

    public SaveGameCommand(Game g){ // g via DTO ou cpy!!
        this.gameToSave = g;
    }

    public void execute(GameRepository gr){
        gr.save(gameToSave);
    }
}

public class ServiceAplication{ [A]
    GameRepository gr;
    CommandQueue queue;

    public void move(Location from, Location to, int gameId){
        Game g = gr.getGamebyId(gameId);
        g.move(from,to);
        SaveGameCommand cmd = new SaveGameCommand(g);
        queue.add(cmd);
    }
}
```

Queue / Dispatcheur

```
public class CommandQueue{
    private Set<SaveGameCommand> commandSet;

    public synchronized void add(SaveGameCommand gc){
        commandSet.add(gc);
    }

    public synchronized SaveGameCommand pop(){
        return commandSet.getLast();
    }
}

public class Worker extends Thread{
    private CommandQueue queue;

    public void run(){
        while(true){
            queue.pop().exec()
        }
    }
}
```

voir schéma 11

Les Query

ex: `getGameById` => on va faire une seule classe qui gère les requêtes et qui va servir à peu près de mémoire cache.

```
public class GetGameServiceApp{
    private Game currentGame;

    public Game getGameById(int idGame){
        if(idGame != currentGame.id){
            currentGame = gr.getGameById(idGame); // ou est l'accès au gr?
        }
        return currentGame;
    }

    public class ServiceApplication{
        // La tout le monde avait trop faim...
    }
}
```