



Composition, Héritage & Tests

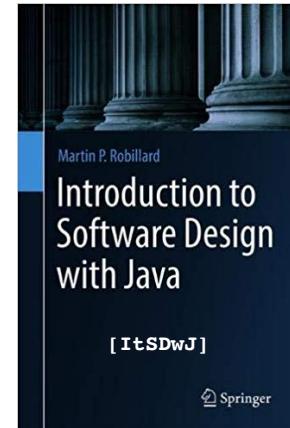
UQÀM | Département d'informatique

Crédit Images: Pixabay & Pexels

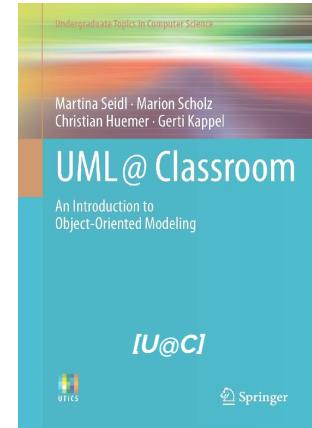
Sébastien Mosser
INF 5153 - Cours #3 - A19



Bibliographie de ce cours



Chapitres 5, 6 & 7



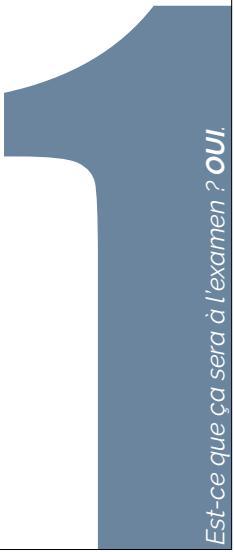
Chapitres 1 & 4

Intermède Publicitaire

- Ce cours utilise intensivement les principes mis en avant dans le livre de Martin Robillard (McGill) aux chapitres cités.
 - Ainsi que le cheminement pour passer de l'un à l'autre
- Les exemples dans le livre sont ceux d'un jeu de cartes, on utilise pour INF-5153 aussi un jeu mais de facture différente.
 - On va aussi prendre de la liberté sur certains exemples.
- Ne recopiez pas le code du livre ou du cours dans vos projets,
 1. Parce que ça serait du **plagiat** ...
 2. **Ce ne sont pas les même problématiques** qui sont rencontrée dans les spécifications de l'arbitre de Poker.

- 1 Dans les épisodes précédents ...
- 2 Relation de composition
- 3 Relation d'héritage
- 4 Tests & Conception
- 5 Q&R sur le TP#1
- 6 Étude de cas “Schotten Totten”

Dans les épisodes précédents ...



Principes SOLID & Loi de Demeter

- **Responsabilité Unique** (*Single Responsibility*)
 - Un objet fait une et une seule chose
- **Principe Ouvert/Fermé** (*Open/Closed Principle*)
 - Une évolution du projet minimise le nombre de modifications et exploite les capacités d'extensions.
- **Ségrégation des interfaces** (*Interface Segregation*)
 - On préfère des interfaces spécialisées à des fourre-tout
- **Loi de Demeter** (*Principle of least knowledge*)
 - "On ne parle pas avec les gens qu'on ne connaît pas".

Encapsulation

- Mise en oeuvre du principe de "*Information Hiding*"
 - Les données sont encapsulées,
 - les objets exposent des services (méthodes).
- **Alignement entre logique d'affaire et services exposés**
- **Attention aux fuites de données** (via **getters** et **setters**)
 - Mécanismes de copies à prendre en compte
- **Trop encapsuler** = ne plus pouvoir travailler
 - Mais **pas assez** = porte ouverte à toutes les fenêtres

Types & Interfaces

- Les **Types** :
 - décrivent de manière **abstraite** les **services rendus** par un objet
 - On peut **affecter à une variable typé T** tout élément dont le type concret est **T ou un de ses sous-types**.
 - P-ex., `List<Card> result = new ArrayList<>();`
 - Sont mis en oeuvre par "**Généralisation**" ou "**Réalisation**".
 - **Abstraction** : Iterable, Comparable, Clonable, ...
- Principe de "**Ségrégation des Interfaces**" (I de SOLID) :
 - Une classe doit dans la mesure du possible **implémenter plusieurs interfaces spécialisée** plutôt qu'une grosse.

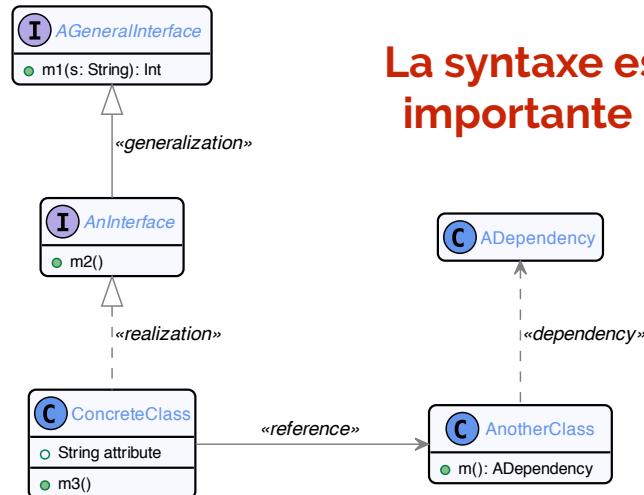
Etat des Objets

- L'espace d'état modélisé par des objets peut devenir très grand
- On s'intéresse uniquement aux états abstraits
 - P-ex. : la pile de carte est vide, mélangée, ...
- Attention: On ne conçoit pas jusqu'à un automate fini !
- Définition de relations entre objets :
 - P-ex., Équivalence, Égalité, ...
- Faire en sorte de minimiser l'espace d'état
 - Pas d'information en double à synchroniser
- Comment modéliser l'absence de valeur ?

Diagramme de Classe (Syntaxe UML)

Structure

La syntaxe est importante !



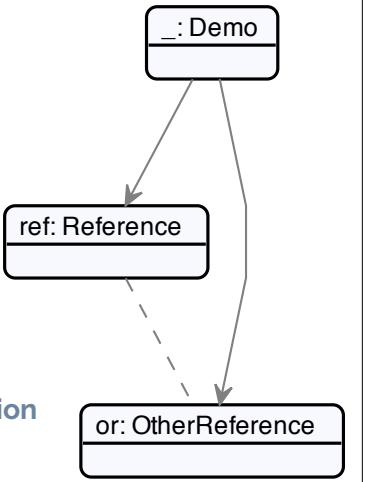
[ItSDwJ, p46]

Diagramme de Séquence (Syntaxe UML)

Structure

Diagramme d'objet (Syntaxe UML)

```
class Demo {
    private Reference ref;
    public void method() {
        OtherRef or = new OtherRef();
        ref.send(or);
    }
}
```



Cliché des objets en cours d'exécution

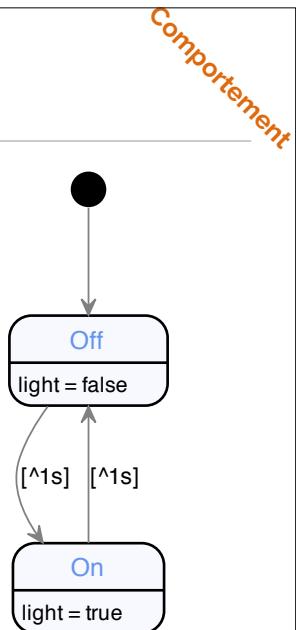
"conformité"

Diagramme d'état (Syntaxe UML)

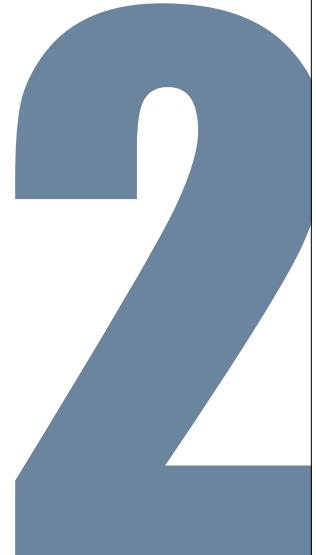
```
#include <avr/io.h>
#include <util/delay.h>

void init(void)
{
    DDRB |= 0b00100000;
    DDRD |= 0b11111110;
}

int main(void)
{
    init();
    while(1)
    {
        PORTB ^= 0b00100000;
        _delay_ms(1000);
    }
    return 0;
}
```



Relation de composition



Relation de Composition ?

- On crée des **systèmes complexe** par **décomposition**
 - Le classique "Diviser pour mieux régner"
- En orienté-objet, la **relation de composition est duale** :
 1. *Le fait qu'un tout soit composé de parties*
 2. *Le fait qu'un objet délègue à un autre objet un traitement*
- **Maintenir une bonne décomposition** est compliquée
 - Problème du "path of least resistance" lors d'une évolution.
 - *On fait apparaître des "Classes Dieu" très facilement.*



Publicité outrancière 😊



Scan me

<https://ace-design.github.io>

ACE := "Abstract **Composition** Engine"

Cas #1 : Un tout est constitué de parties

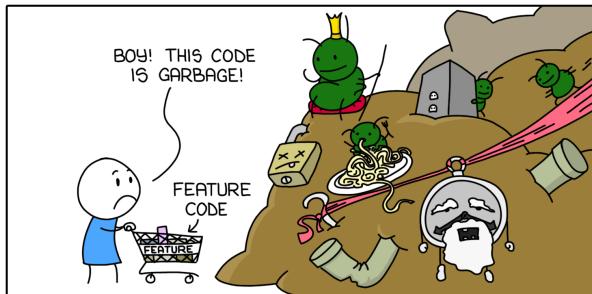
- **Structuellement**, un **élément** fait partie d'un **agrégat**
 - Les cartes constituent la pioche
 - La classe **CardPile** est un agrégat d'instances de **Card**
 - Une voiture contient 4 pneus
- Dans ce cas, **la composition est intrinsèque à la situation**
 - c.à.d. "les objets reflète la réalité"

[ItSDwJ, p120]

Cas #2 : Délégation à un autre objet

- La composition est "**conceptuelle**" plus que structurelle
 - Une partie de poker contient un croupier
- On utilise ce type de composition pour **casser la complexité d'un objet** qui aurait tendance à centraliser trop de choses
 - Permet de **maintenir le principe de responsabilité unique**.
 - Notion de "**fournisseur de service**" pour la classe
- Il est **facile/dangereux** de rajouter des choses dans une classe :
 - Pas besoin de réfléchir à l'**encapsulation**, on voit tout.
 - **La délégation permet de diminuer l'entropie** dans le code.

CODE ENTROPY



Voir la question #1 du premier projet.



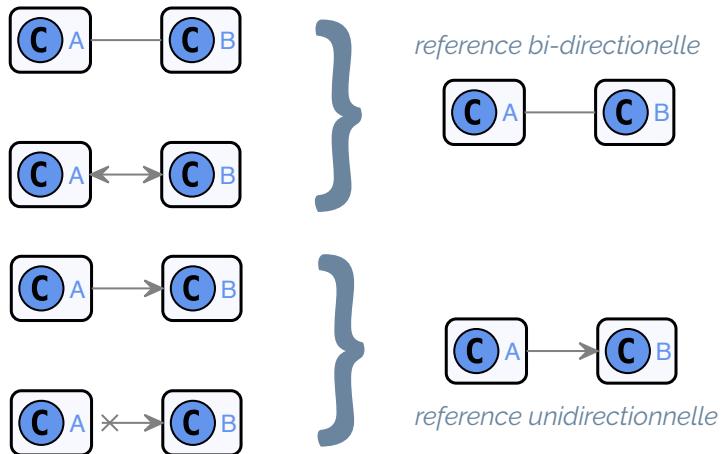
Que coûterait l'ajout d'une nouvelle fonctionnalité dans le code "en l'état" ?

Représentation de la composition

- On parle de **composition** dès qu'un objet détient une référence vers un autre objet.
- Le langage UML définit **trois (3) types de relations** pour ce cas :
 - L'**association** (flèche simple, *pas vraiment une composition*)
 - L'**agrégation** (diamant blanc ◇, souvent utilisée pour le cas de délégation)
 - La **composition** (diamant noir ◆, *composition structurelle forte*)
- Chaque type de relation a une **sémantique forte**
 - **Elles ne sont pas interchangeable** (*dans le détail*) !



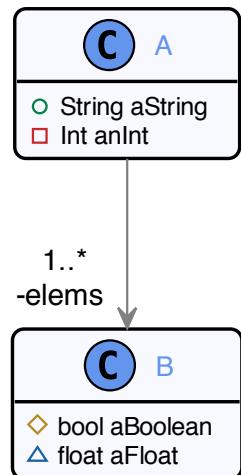
Bonnes pratiques UML



Rappel : Le sens de lecture est inversé !

```
class A {
    public String aString;
    private int anInt;
    private Set<B> elems;
}

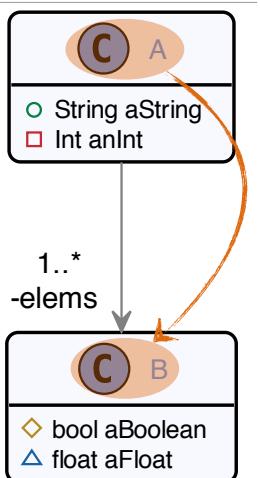
class B {
    protected boolean aBool
    float aFloat
}
```



Références ≠ Attributs

```
class A {
    public String aString;
    private int anInt;
    private Set<B> elems;
}

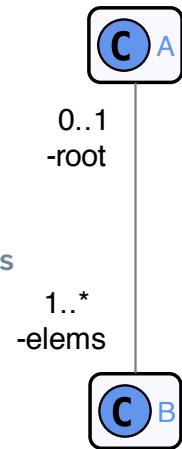
class B {
    protected boolean aBool
    float aFloat
}
```



Composition et Multiplicité

```
class A {
    private Set<B> elems;
}

class B {
    private A root;
}
```



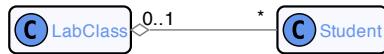
Soyez explicite dans vos modèles si des **points sont importants** pour le code :

- **root** : null ? Optionnel ?
- **elems** : ordonné ? Non-ordonné ? Taille min ?

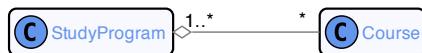
Attention, des infos du modèle de conception peuvent être "invisible" dans la structure du code.

Agrégation (\diamond) versus Composition (\blacklozenge)

- L'agrégation représente une **composition "faible"**
 - "La partie peut exister sans son tout"
 - Définit un **graphe orienté acyclique** d'appartenance
 - Cycles dans le modèle = problème de conception (presque)**



- Exemples :



- Un étudiant fait partie (ou non) d'un seul groupe de labo;
- Un cours peut faire partie de plusieurs programmes d'études.

Agrégation (\diamond) versus Composition (\blacklozenge)

- La composition représente une **composition "forte"**
 - Si on **détruit l'agrégat**, on **détruit aussi l'élément**
 - La partie ne peut être **contenue** que par **un seul** agrégat
 - Donc la multiplicité coté agrégat est toujours de 0 ou 1**



- Exemple :

- Si je détruit "INF-5153", est-ce que je détruis aussi les étudiants ?

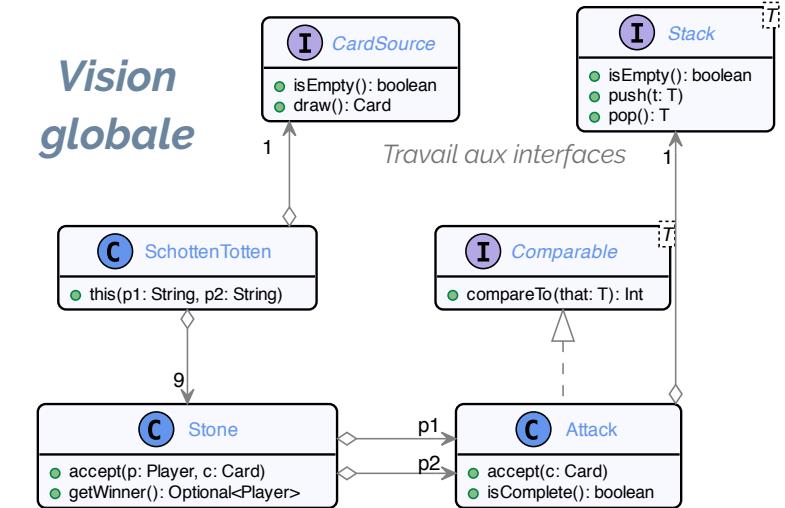
Agrégation (\diamond) versus Composition (\blacklozenge)



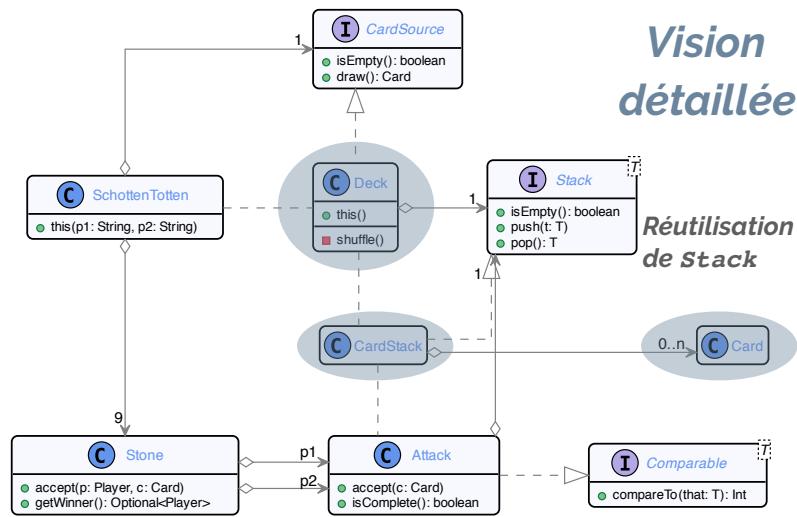
Dans les faits, entre \diamond et \blacklozenge , on peut souvent approximer ce qu'on veut dire en utilisant **toujours la même relation**.

Exemple de Composition : Pile et Attaques

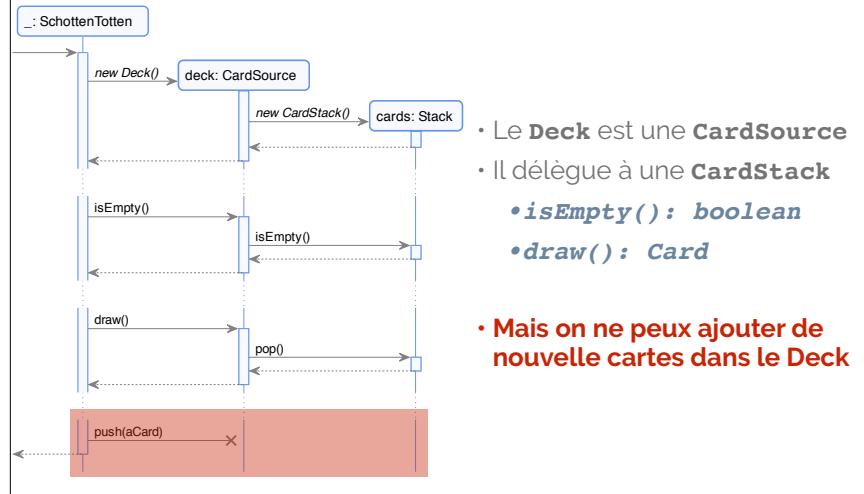
Vision globale



Exemple de Composition : Pile et Attaques



Exemple de Composition : Pile et Attaques



Composition ≠ Sous-Typage

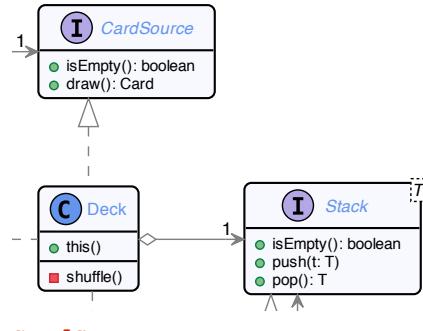
- Une relation de composition n'implique pas le sous-typage

Contrairement à la *généralisation* et la *réalisation*

- Dans l'exemple précédent
 - Un Deck **est une** CardSource
 - Un Deck **contient** une Stack

Mais une Stack n'est pas une CardSource

(ce qui est discutable)



Un problème de composition : la multi-pile

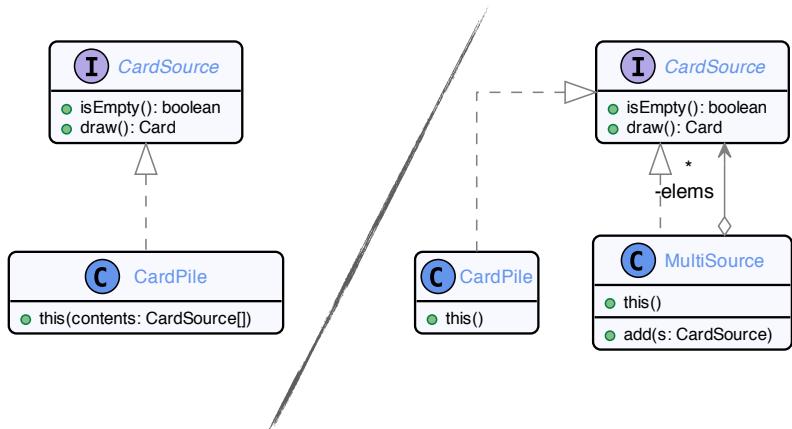
- Comment jouer avec deux jeux de cartes ?

Voir "n" jeux de cartes ?

- Deux conceptions opposées :

- Fabriquer **une pile qui contient toutes les cartes** des autres
 - Se pose le problème de la copie (voir cours précédent)
- Fabriquer **une pile qui est composée des autres piles**
 - Une sorte de "pile composite" (analogie avec des LEGO®)

Propositions : Copie versus Composition



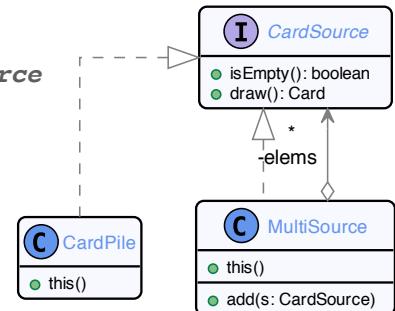
Analyse de l’approche par composition

Forces :

- **MultiSource** réalise **CardSource**
- Couplage faible
- On peut rajouter des sources dynamiquement
- Pas de copie en mémoire

Faiblesses :

- Plus difficile à mettre en oeuvre
- Pour ajouter une source, il faut savoir que c'est une **MultiSource**



Cette forme est un “patron de conception” (le “composite”), on l’étudiera en détail plus tard

Un autre problème : Ajout de comportement

- On veut une **CardSource** qui puisse :

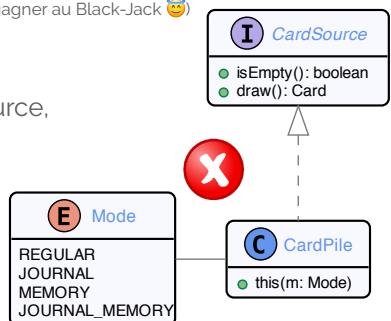
- **Faire son travail** normalement;
- **Journaliser les actions** faites dessus (*exigence légale pour le casino*)
- **Mémoriser les cartes** tirées (pour gagner au Black-Jack 😊)

Proposition

- On rajoute un “**mode**” dans la source,
- et on décide en fonction !

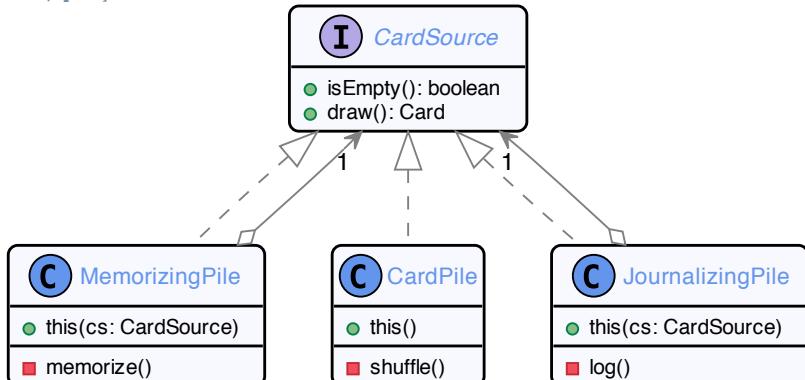
Switch Statement
Explosion Combinatoire !

[ItSDwJ, p130]



Proposition : Composer les comportements

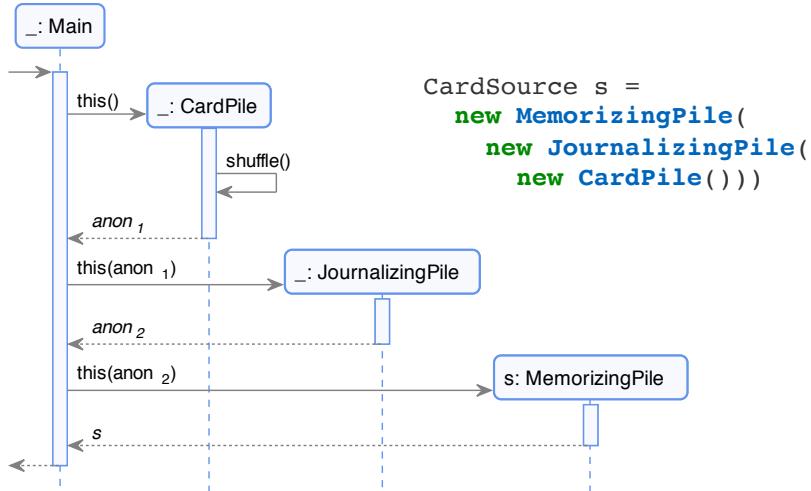
[ItSDwJ, p133]



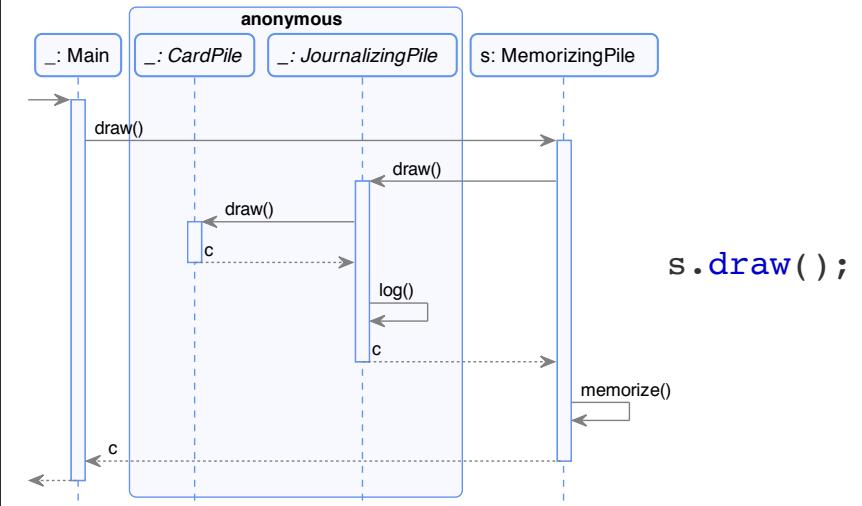
```

CardSource s =
  new MemorizingPile(new JournalizingPile(new CardPile()))
  
```

On repose sur la délégation de comportement



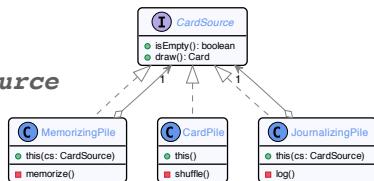
On repose sur la délégation de comportement



Analyse de l'approche par délégation

Forces :

- Réalisation homogène de **CardSource**
- Couplage faible
- On peut rajouter des comportements dynamiquement pendant l'exécution
- Responsabilité unique et ouvert-fermé en même temps



Faiblesses :

- Pleins de petits objets, dont l'ordre de construction importe.
- Problème de l'identité entre objets. Comment définir **equals** ?

Cette forme est un "patron de conception" (le "décorateur"), on l'étudiera en détail plus tard



Relation d'héritage



Ça fait **TROIS** cours de soit-disant “conception objet” et on a toujours pas parlé d’héritage ...

Effectivement. Et pourtant ...

- La relation de **composition** permet de **modulariser** un système
- La relation de **réalisation** permet d'introduire du **sous-typage**
- **On peut faire beaucoup de choses sans héritage**
 - Il existe même des langages objets pour lesquels cette construction n'existe pas (*p.-ex. Go dans une certaine mesure*)
- Par expérience, **vous utilisez très mal la relation de généralisation**
 - *“Si vous hésitez entre une généralisation et une composition, c'est souvent que c'est une composition”* – Privat 2019.

Le mécanisme d'héritage, qui permet facilement de factoriser le code des classes similaires, repose fondamentalement sur la relation de généralisation des concepts associés.

La contraposée est intéressante car elle permet de déterminer facilement l'usage abusif d'héritage : s'il n'y a pas de relation évidente de généralisation, c'est sans doute pas de l'héritage

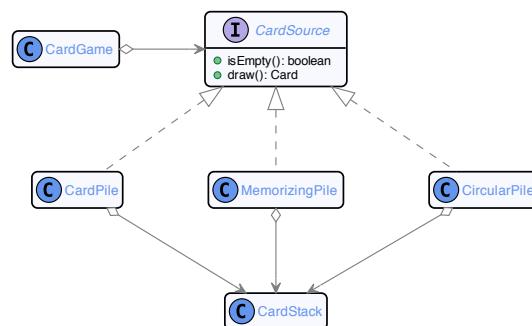
(rasoir de Privat)

Du polymorphisme à l'héritage

- Dans les faits, **utiliser uniquement des interfaces amène de la redondance** dans le code
 - Phénomène de “**déception de code**”

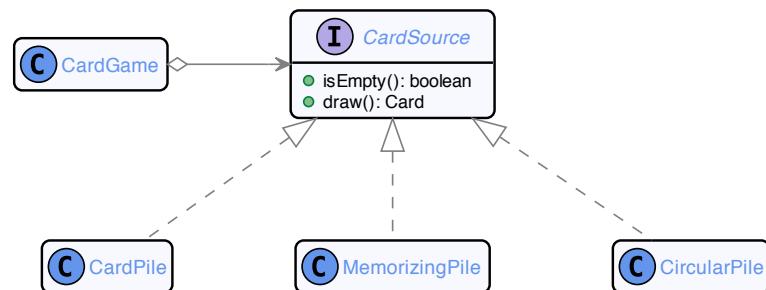
Principe “DRY” :

Don't Repeat Yourself



Du polymorphisme à l'héritage

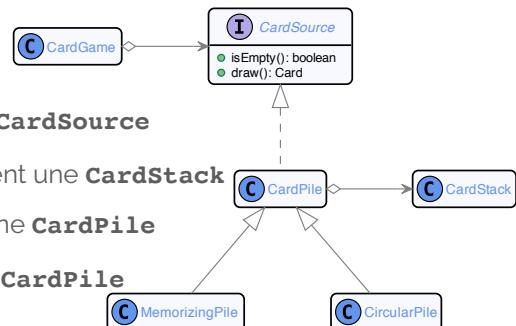
- En Java, le polymorphisme repose sur le sous-typage



On peut sous-typer avec de la réalisation (interfaces)

Du polymorphisme à l'héritage

- L'héritage **met en oeuvre la relation de généralisation**, et permet d'**étendre** une "base class" dans une "subclass"
 - En français, on parle de "classe mère" et de "classe fille"
 - La **généralisation** est une **relation de sous-typage + réutilisation**
 - Dans l'exemple :



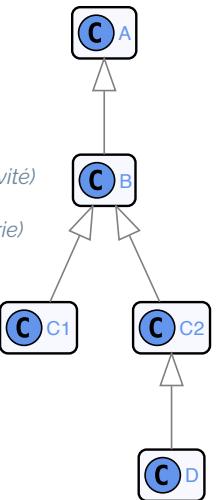
Le sous-typage comme une relation d'ordre

- La **généralisation** met en oeuvre une **relation d'ordre sur les classes**

- $\forall c \in C, c \leq c$ (réflexivité)
- $\forall (c_1, c_2, c_3) \in C^3, (c_1 \leq c_2 \wedge c_2 \leq c_3) \Rightarrow c_1 \leq c_3$ (transitivité)
- $\forall (c_1, c_2) \in C^2, (c_1 \leq c_2 \wedge c_2 \leq c_1) \Rightarrow c_1 = c_2$ (antisymétrie)

- Il ne peut exister de cycles** de généralisation

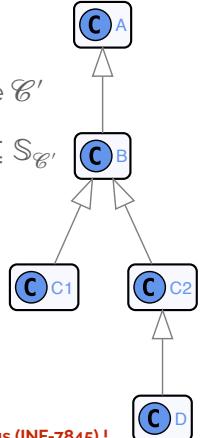
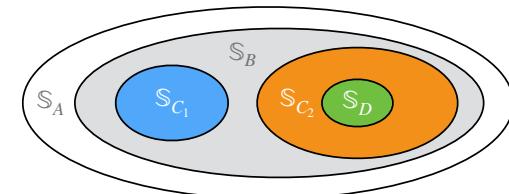
- Par définition d'une relation d'ordre, c'est cadeau



- L'ordre est **partiel**: $C_1 \gtrless C_2 ? \quad C_1 \gtrless D ?$

Sémantique ensembliste associée

- On note \mathcal{C} une classe, et $\mathbb{S}_{\mathcal{C}}$ l'ensemble des instances de \mathcal{C}
- Soit deux classes \mathcal{C} et \mathcal{C}' . Si \mathcal{C} spécialise \mathcal{C}' , alors :
 - On note $\mathcal{C} <: \mathcal{C}'$ le fait que \mathcal{C} est un sous-type de \mathcal{C}'
 - Toute instance de \mathcal{C} est une instance de \mathcal{C}' : $\mathbb{S}_{\mathcal{C}} \subseteq \mathbb{S}_{\mathcal{C}'}$



Le sous-typage c'est compliqué, venez à la maîtrise en info pour en voir plus (INF-7845) !

Héritage et Typage (en Java)

- Pour chaque objet, **il existe deux types** :

- Son type run-time** \mathcal{R} , attribué lors de la création
- Son type statique** \mathcal{S} , le type de la variable associée

- A tout moment, le système de type de Java garanti $\mathcal{R} <: \mathcal{S}$

- Exemple** : `Set<String> o = new HashSet<>()`

- Le type run-time de $o (\mathcal{R}(o))$ est `HashSet<String>`
- Le type statique de $o (\mathcal{S}(o))$ est `Set<String>`, $\mathcal{R}(o) <: \mathcal{S}(o)$
- Le type run-time $\mathcal{R}(o)$ ne changera plus jamais**

Surcharge et Aiguillage Dynamique (en Java)

- Surcharge** : (*method overloading*)

- Méthodes de même nom mais de signatures différentes
 - `public int count(CardPile p)`
 - `public int count(CircularPile p)`
 - on ne peut pas uniquement surcharger le type de retour

- Aiguillage dynamique** : (*dynamic dispatch & method overriding*)

- Méthode d'une super-classe redéfinie dans une sous-classe
- Permet les appels de méthodes polymorphiques

Relation avec le typage (en Java)

```

public class Main {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        A ab = b;
        method(a);
        method(b);
        method(ab);
        System.out.println(a.toString());
        System.out.println(b.toString());
        System.out.println(ab.toString());
    }

    private static void method(A object) {
        System.out.println("I'm called on an A");
    }

    private static void method(B object) {
        System.out.println("I'm called on a B");
    }
}

```

```

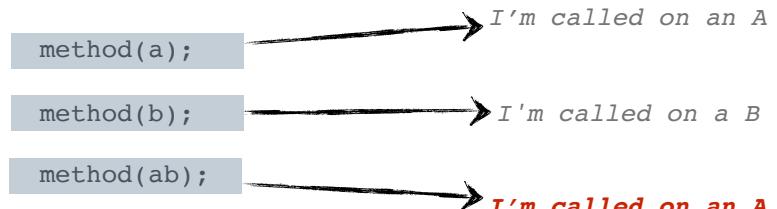
public class A {
    @Override
    public String toString() {
        return "I'm an A";
    }
}

public class B extends A {
    @Override
    public String toString() {
        return "I'm a B";
    }
}

```

Résultat d'exécution

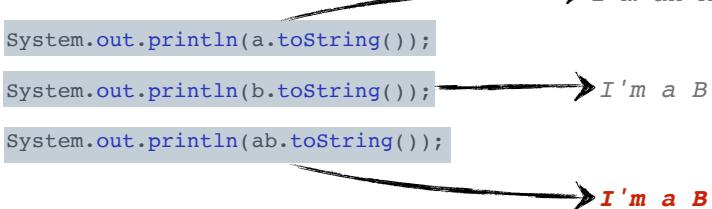
- $\mathcal{R}(a) = A, \mathcal{S}(a) = A$
- $\mathcal{R}(b) = B, \mathcal{S}(b) = B$
- $\mathcal{R}(ab) = B, \mathcal{S}(ab) = A$
- $a \neq b, b = ab$



Pour la surcharge, c'est le type statique qui est utilisé

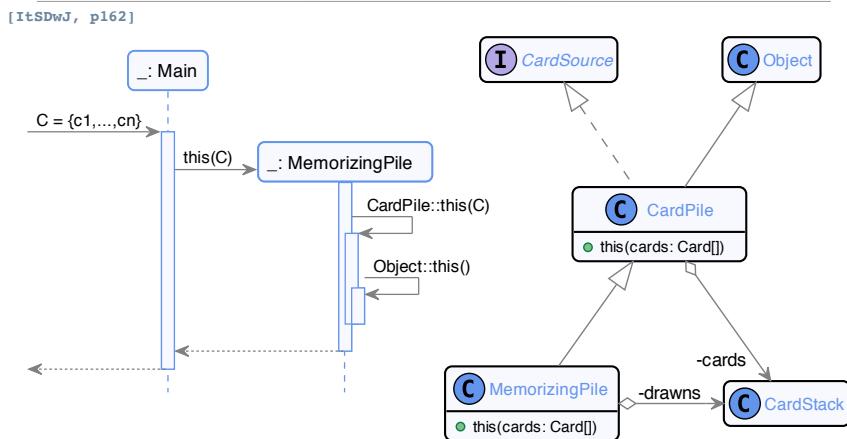
Résultat d'exécution

- $\mathcal{R}(a) = A, \mathcal{S}(a) = A$
- $\mathcal{R}(b) = B, \mathcal{S}(b) = B$
- $\mathcal{R}(ab) = B, \mathcal{S}(ab) = A$
- $a \neq b, b = ab$



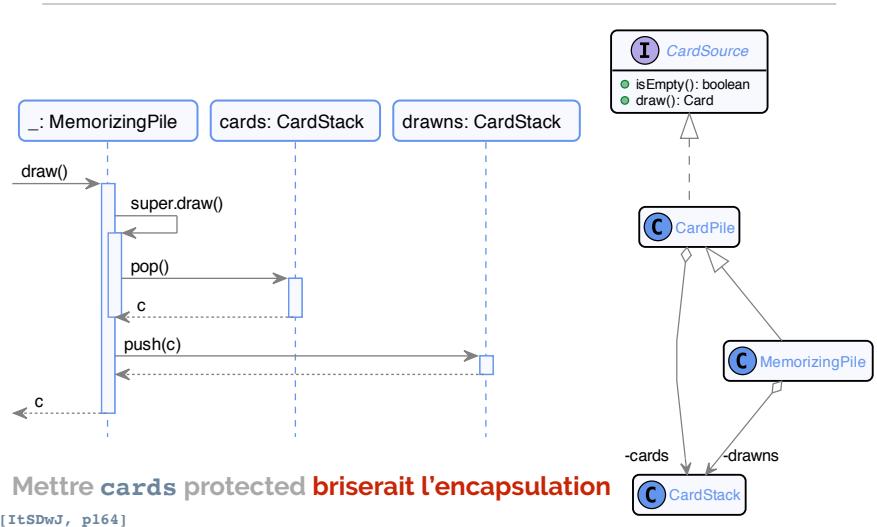
Pour l'aiguillage, c'est le type run-time qui est utilisé

Construction des objets : “bottom-up”

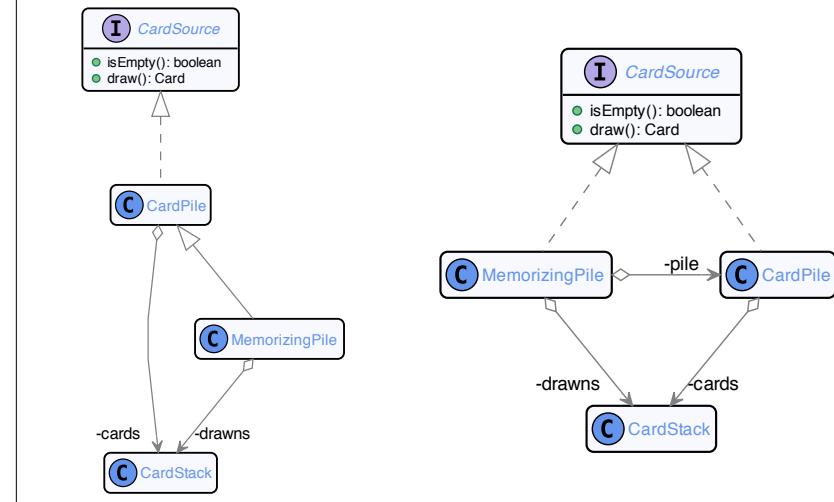


Pour faire référence au constructeur de la super-classe, on appelle *super*

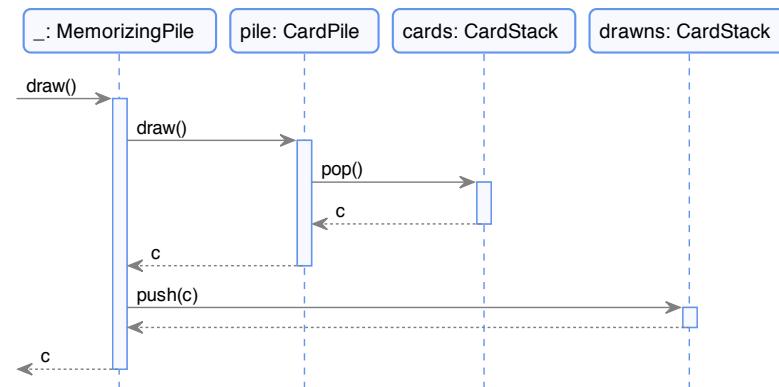
Héritage des méthodes



Héritage versus Composition



Héritage versus Composition



Ici il y a deux objets, une instance de `MemorizingPile` et une instance de `CardPile`

Héritage versus Composition

- On peut très souvent **utiliser l'une ou l'autre** des approches
 - Mais les systèmes obtenus n'ont pas les même propriétés*
- L'approche par héritage**
 - Permet de fixer des choses au moment de la compilation*
 - Souffre de problème d'explosion combinatoire*
- L'approche par composition**
 - Est moins efficace lors de l'exécution*
 - Permet de modifier dynamiquement le comportement*

Principe de Substitution de Liskov



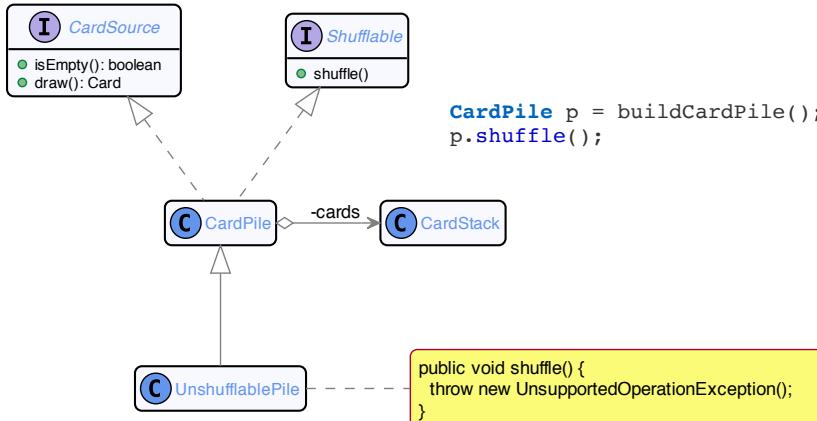
Turing Award 2008

- La **généralisation** est un mécanisme **strictement additif**
 - Et l'héritage met en oeuvre de la généralisation
 - Et **rien n'oblige le développeur à respecter l'additivité** de la relation dans son code
- Exemple :
 - Une **CardPile** réalise **CardSource** et **Shufflable**
 - Une **UnshufflablePile** est une **CardPile**, sans mélange
 - Pour respecter DRY, elle hérite de **CardPile**

Principe de Substitution de Liskov



Turing Award 2008



Sans garantie sur le type run-time de p, on risque l'accident

Principe de Substitution de Liskov



Turing Award 2008

- Principe de Liskov :
 - Une sous-classe ne doit pas restreindre ce que les clients de la super-classe de cette instance feront avec.
- Plus concrètement :
 - Pas de **pré-conditions plus strictes**, ou **post-condition plus larges**
 - Ne pas prendre de **type plus spécifique** en paramètre
 - Ne pas rendre la **méthode moins accessible**
 - Ne pas lever plus d'**exceptions**
 - Ne pas avoir un **type de retour moins spécifique**

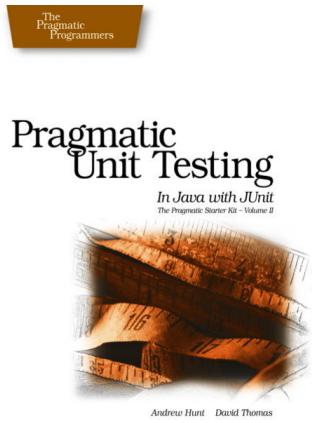
Pour aller plus loin : covariance et contravariance, INF-7845

Tests & Conception



Bibliographie Tests Unitaires

- Excellent livre :
- très utile
 - se lit très vite



Différents types de tests

- **"Tester, c'est douter"**
- Certes, mais du code qui compile pas ça a jamais tué personne, alors que du code qui compile ... (F. Bordeleau, 2018)
- Plusieurs type de tests (non exhaustif), en fonction des besoins
 - Tests unitaire
 - Tests d'acceptation
 - Tests par propriétés
 - Tests par mutation
- Loi empirique : **"Un programme bien conçu se teste bien"**
- Les tests sont de la validation, pas de la vérification (preuve)

Tests Unitaires

- Un **test unitaire** considère une "Unit under Test" (**UUT**)
 - Il exécute l'UUT sur des **données d'entrées connues**
 - Et **vérifie** la valeur de retour avec un **oracle**
- La **vérification** de la valeur est appelée une **assertion**
- **Il n'est pas possible de tester tous les cas possibles**
 - **Intuition** : si pour toute entrée je connais l'oracle, je n'ai pas besoin du programme en question
 - **Rappel** : L'espace d'état peut-être très grand
- **On cherche donc à tester "utile"**

Tests d'acceptation

- Permet de **modéliser des scénarios** d'acceptation
 - Qu'on va classiquement accrocher aux cas d'utilisation
- On utilise un système d'axiomes **Context / Action / Assertion** :
 - **Given** a situation,
 - **When** something happens,
 - **Then** a result is expected
- Le canevas de **référence** est **Cucumber**
 - Initialement en Ruby, maintenant dans beaucoup de langages

Est-ce que ça sera à l'examen ? **NON.**

Tests par propriétés

- Utilisé pour **modéliser** des tests de **propriétés algébriques**
 - P-ex. qu'une relation d'ordre en est bien une
- Repose sur **deux principes** :
 - On écrit les propriétés comme des méthodes
 - On écrit des générateurs d'objets
- A l'exécution du test, le système va :
 - **Générer** un grand nombre d'objets
 - **Appliquer** la propriété dessus, et **vérifier** le résultat.
- **C'est "empirique", a défaut de preuve** (ref: QuickCheck).

*Est-ce que ça sera à l'examen ? **NON**.*

Tests par mutation

- Les tests valident le code. **Qui valide les tests ?**
- Principe de **méta-programmation**
 - Le système va faire muter le code pour **introduire des bogues**
 - P-ex. inverser une condition : `if (i<10) ↪ if (i>=10)`
 - Repose sur des mécanismes d'intercession de code
- On lance ensuite les tests sur le code mutant
 - **Si le mutant survit**, c'est que **les tests sont moisis**.
- Canevas de référence : PiTest (rien à voir avec python)

*Est-ce que ça sera à l'examen ? **NON**.*

Propriétés des Tests Unitaires (TU)

- **Rapide**
 - Les TU sont lancés à chaque incrément du développeur
- **Indépendants**
 - On peut les lancer en isolation les uns des autres
- **Répétable**
 - Le même paramétrage doit produire le même résultat
- **Précis**
 - "Responsabilité unique" appliquée aux tests
- **Lisible**
 - Parce que ça reste du code (et que ça sert de spécifications)

Exemple de Spécification à tester

$$\sqrt : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \\ x \mapsto y, \quad y^2 = x$$

- **Exemples** de test :
 - $\sqrt{4} = 2$
 - $\sqrt{100} = 10$
- **Mais comment tester "utilement" la spécification ?**
 - Problème de **sélection des tests**

Cible : Les critères BICEP

- Pour être utile, les tests doivent utiliser leur **BICEPs** :
 - **Boundary**
 - **Inverse Relationship**
 - **Crosscheck**
 - **Error**
 - **Performances**
- Comme d'habitude, les solutions dogmatique sont à adapter à la situation rencontrée dans chaque projet en particulier.

Cible BICEP

- **Boundary** : test aux frontières
 - Que vaut $\sqrt{0}$? que vaut $\sqrt{\infty}$? (peut-on le tester ?)
- **Inverse relationship** : test sans oracle, sur la définition
 - $\forall x \in \mathbb{R}^+, \sqrt{x} \times \sqrt{x} = x$
- **Crosscheck** : vérification avec un oracle certifié
 - P-ex. `java.util.Math.sqrt(x) = $\sqrt{x} \pm 0,0001$`

Les modèles de conception aident au choix des tests, en rendant explicite les informations nécessaires à prendre ce genre de décisions

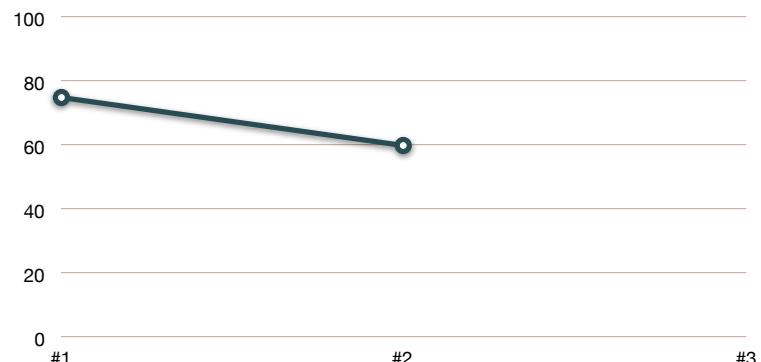
Cible BICEP

- **Error** : Identifier et tester les cas d'erreurs
 - Que vaut $\sqrt{-1}$? Une exception ? null ?
- **Performance** : Identifier les temps de réponses attendus
 - P-ex. $time(\sqrt{3825}) < 200ms$
- **Attention à la reproductibilité des tests de performances**
 - Souvent on les implémente dans un projet à part
 - Par exemple **un banc de test avec JMH**

Projet #1 Questions / Réponses



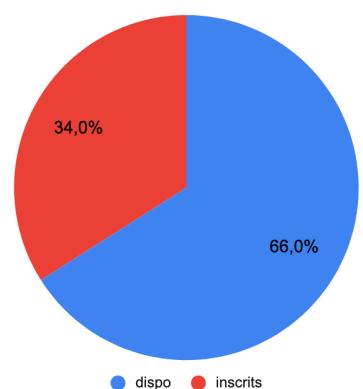
Taux de participation au Labos



Les labos servent à confronter votre conception à un évaluateur

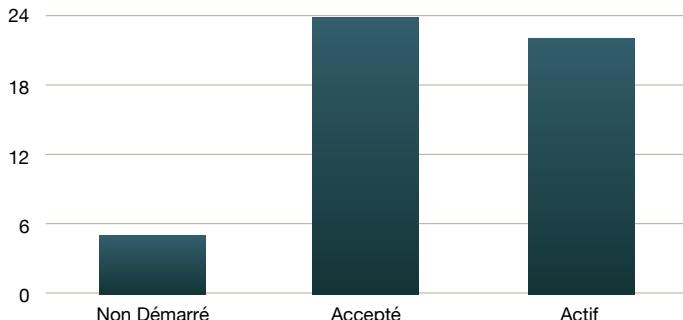
Préparation du Projet #2 - Island

Complétion des équipes



19.09.2019, 08:15AM

État du Projet #1 sur Classroom



Rappel

Travail

git push

git add

git commit

19.09.2019, 08:15AM

Rappel sur la grille d'évaluation

Élement	Critère d'évaluation	Note (/100)
Questions	(#1) Évolution du code légataire	/5
	(#2) Analyse des défauts du code légataire	/10
	(#3) Justification des choix de conception	/15
	(#4) Évolution du code objet	/5
Modèles	Justesse & Pertinence de la conception	/15
	Cohérence inter-modèles	/5
	Respect des principes de conception	/15
Code	Qualité du code Java et du dépôt Git	/10
	Cohérence du code avec les modèles	/10
	Qualité des tests	/10

35%

35%

30%

Problèmes observés

- Obsession primitive
- Modèles **UML non conforme** à la syntaxe
- Trop de **focus sur le code**
 - Et pourtant aucun **focus sur les tests**
- **Travail non poussé**
 - *Est-ce que ça compile ailleurs que sur ma machine ?*
 - *Est-ce que l'évaluateur y aura accès ?*
 - *Et si mardi prochain ça marche pas, on fait comment ?*

Je ne répondrais à aucune question sur le slack de 11h30 (fin du lab) à 23h50 le jour de la remise.

Basculer le “labo” dans une salle régulière ?



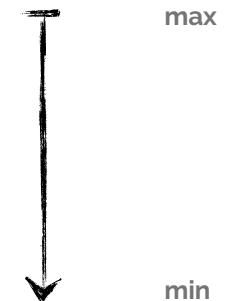
Sondage slack, clôture du vote vendredi soir

Étude de Cas (avec Schotten Totten)



Toute ressemblance avec un Jeu de Poker est fortuite ... 😊

Force des attaques



Conception des Attaques

• Décomposition Structurelle :

- Comment relier les joueurs à la Game, aux Pierres, aux attaques ?
- Comment tenir la limite de 3 cartes max par attaque ?

• Décomposition fonctionnelle :

- Comment trouver les combinaisons ?
 - Structurellement ? Dynamiquement ?
- Comment comparer deux attaques ?
- Comment réclamer une pierre pour une attaque incomplète ?

