Des tests *a posteriori* &
Mise en place architecture MVC

Exercice 1 : Tester le *fizz buzz* : Des tests *a posteriori* sur un code existant

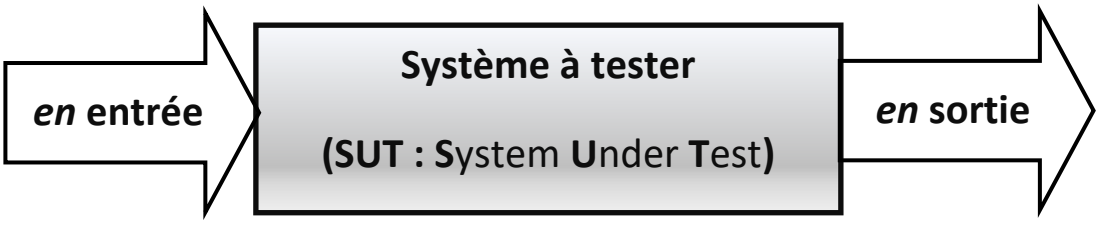
Ce matin, vous venez de rejoindre une nouvelle équipe développement qui travaille sur le projet **`jeuxmathematiques.`** Hier, elle a développé le programme suivant pour implémenter le jeu du *fizz buzz* qui est à l'origine un petit jeu pour apprendre aux enfants le principe de la division.

```
public class FizzBuzz {

     public String jouer(int nombre) {

          if (nombre % 15 == 0)
               return "fizz buzz";

          if (nombre % 3 == 0)
               return "fizz";

          if (nombre % 5 == 0)
               return "buzz";

          return Integer.toString(nombre);
     }
}
```

1. Commencez par prendre du connaissance du code de ce projet.
   Comprenez-vous son comportement ?

2. Le travail que l'équipe souhaite vous confier aujourd'hui est d'écrire des tests afin de vérifier et valider l'implémentation de la méthode **`jouer.`**

   Avant d'écrire le code de votre(vos) test(s), vous devez passer par une phase de réflexion pour bien identifier les 3 étapes du pattern AAA.
   Pour faciliter cette réflexion, nous vous demandons, dans cet exercice, d'illustrer le cas de test que vous souhaitez implémenter avec la notation graphique suivante, qui n'est autre que celle d'un système avec ses entrées (pertinemment choisies) et ses sorties (attendues). C'est ce que l'on appelle un test en boîte noire.



Pour notre exemple, le **SUT** n'est autre que la méthode **`jouer`** avec 1 seule entrée (`nombre`) et une sortie une chaine de caractère (`String`). L'entrée et la sortie sont de vraies valeurs qui permettent d'illustrer le comportement du système.

a. **Cas de test n° 1 : fizz pour un multiple de 3**

On souhaite que ce premier cas de test vérifie et valide le comportement du programme lorsqu'un **fizz** est attendu en sortie

⇨ Analyse : Recherchez un exemple qui permet d'illustrer le comportement que vous souhaitez tester. Illustrez cet exemple en utilisant la notation graphique d'un système donnée ci-dessous. Nous avons déjà complété la sortie, puisque la sortie attendue est **fizz** Il ne vous reste plus qu'à choisir une valeur en entrée pertinente qui est censée engendrer le comportement attendu.

⇨ Implémentation : Ecrivez le code Java d'une **méthode de test** qui implémente l'exemple précédent (en utilisant le code métier de la classe `FizzBuzz`) et permet d'exécuter automatiquement ce test à l'aide du framework JUnit.
La qualité du code de test (nommage notamment) doit être soignée.

Remarque : Rappelez-vous qu'un *cas de test* est un ***exemple*** du comportement du système, donc vous n'avez peut-être pas tous utilisé la même valeur pour écrire ce test.
Par contre toutes vos valeurs doivent vérifiées **la règle métier testée** par ce comportement c-a-d que votre valeur doit être un multiple de 3, mais pas de 15😉.

⇨ Sur le code de production (classe **`FizzBuzz`**) de la première page, surlignez les lignes de code qui seront parcourues (couvertes) lorsque ce test sera exécutée : c'est ce que l'on appelle la ***couverture de code par les tests*** 😉

Pour que le code de production (classe **`FizzBuzz`**) soit **couvert à 100%** c-a-d que pour que toutes les lignes de **`FizzBuzz`** soit *couvertes/parcourues* lorsque tous les tests de la classe **`FizzBuzzTest`** seront exécutés, il est nécessaire d'écrire en tout **4 tests testant 4 comportements différents**
Vous venez d'écrire un test, il ne vous reste plus que 3 tests à écrire dans l'ordre que vous voulez 😉

b. **Cas de test n° 2 : …….. (indiquez le comportement à tester)**

⇨ Analyse : Recherchez un exemple qui permet d'illustrer le comportement à tester

⇨ Implémentation :

⇨ Surlignez pour visualiser la couverture de code et mieux identifier ce qu'il reste à tester

c. **Cas de test n° 3 : …….. (indiquez le comportement à tester)**

⇨ Analyse : Recherchez un exemple qui permet d'illustrer le comportement à tester

⇨ Implémentation :

⇨ Surlignez pour visualiser la couverture de code et mieux identifier ce qu'il reste à tester

d. Cas de test n° 4 : ……

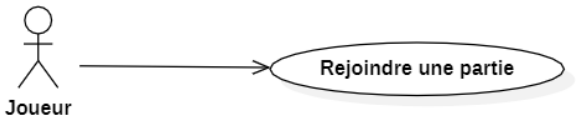
⇨ Analyse : Recherchez un exemple qui permet d'illustrer le comportement à tester

⇨ Implémentation :

⇨ Surlignez pour visualiser la couverture de code et mieux identifier ce qu'il reste à tester

Exercice 2 : Mise en place architecture MVC 😉

Le première partie de l'énoncé de cet exercice reprend les transparents présentés et expliqués à la fin du cours sur le diagramme de séquence
qui se focalise sur la fonctionnalité **`Rejoindre une partie`** *du jeu de cartes de la bataille*
Ce cours est disponible sur https://github.com/iblasquez/enseignement-but1-developpement



Remarque : *Un UC (Use Case) est en principe nommé au singulier.*
S'il y a plusieurs joueurs dans le jeu, il suffira de répéter le UC…
ou lors de l'implémentation de rappeler la fonctionnalité ce qui évitera la duplication de code 😉

❑ Phase d'Analyse autour de la fonctionnalité :

Comprendre le comportement de la fonctionnalité *(**Quoi** faire lorsque la fonctionnalité est appelée)*

⇨ **Un scénario nominal détaillé** pourrait être :

1. **Le système** demande le nom du joueur
2. **Le joueur** saisit son nom
3. **Le système** enregistre un nouveau joueur dans la partie avec le nom saisi
4. **Le système** affiche un message de bienvenue avec le nom du joueur

⇨ Outre l'écriture d'un scénario, il est fortement recommandé **durant la phase d'analyse de proposer des maquettes** pour montrer au client comment vous imaginez l'interface graphique qui répond à son problème et qui montre que vous avez bien compris (ou non) ses besoins (maquettes aussi bien pour le cas nominal que pour les cas)

⇨ Durant la phase d'analyse, c'est donc le bon moment pour s'interroger et identifier **les règles métiers particulières qui devront absolument être implémentées et testées afin que votre fonctionnalité respecte les besoins du client et s'exécute sans bug. Utilisez des exemples (données/valeurs) dans vos maquettes pour bien identifier ces règles et leurs comportements et faciliter la compréhension de tous les acteurs**😉

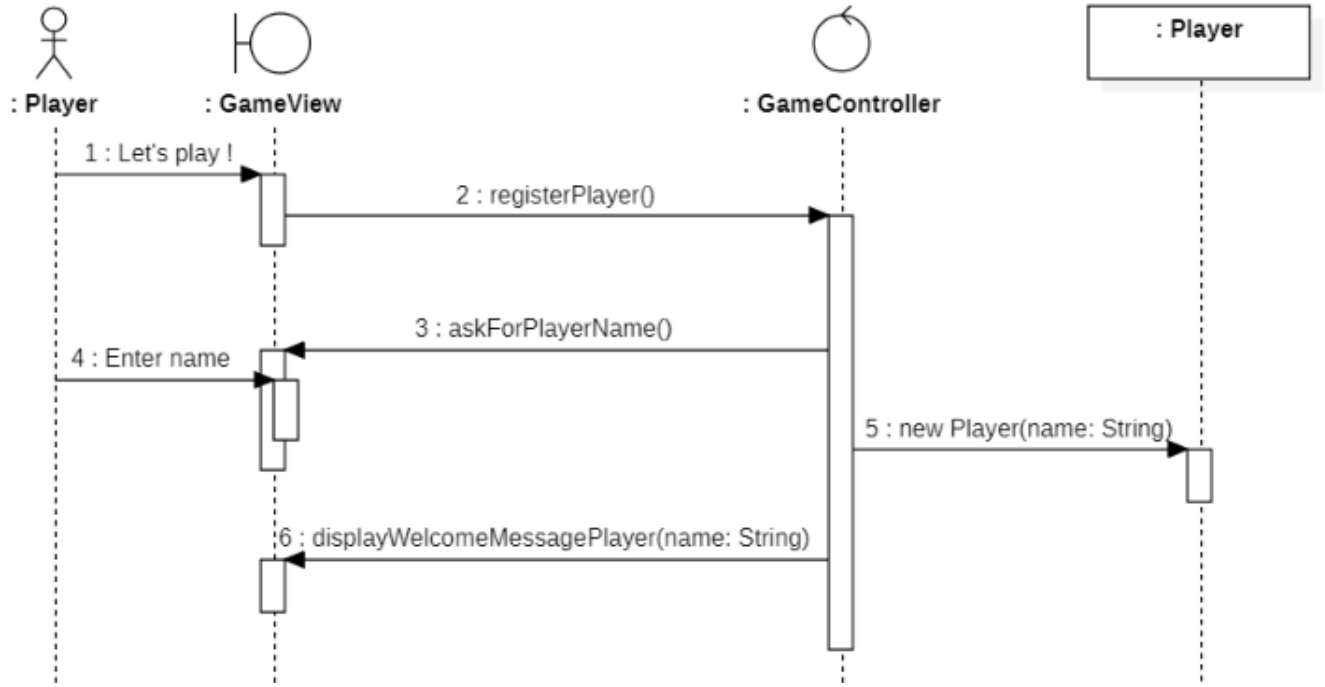
❑ Phase de Conception :

La mise en place d'une **interactivité système/utilisateur** a une incidence sur l'architecture logicielle d'une application.
Il est important lors de la mise en place d'une architecture de ne pas « réinventer la roue », mais de s'appuyer sur des patterns (modèles de programmation) robustes et simples qui ont fait leur preuve.
Le **pattern MVC** (**M**odèle **V**ue **C**ontrôleur) est le plus couramment utilisé pour mettre en place une IHM en proposant une architecture en 3 couches (3 paquetages) qui permet de séparer les responsabilités et faciliter l'extension et la maintenance de l'application
(le pattern MVC a été présenté en cours : un petit résumé, en annexe, sera à (re)lire après ce TD 😉)

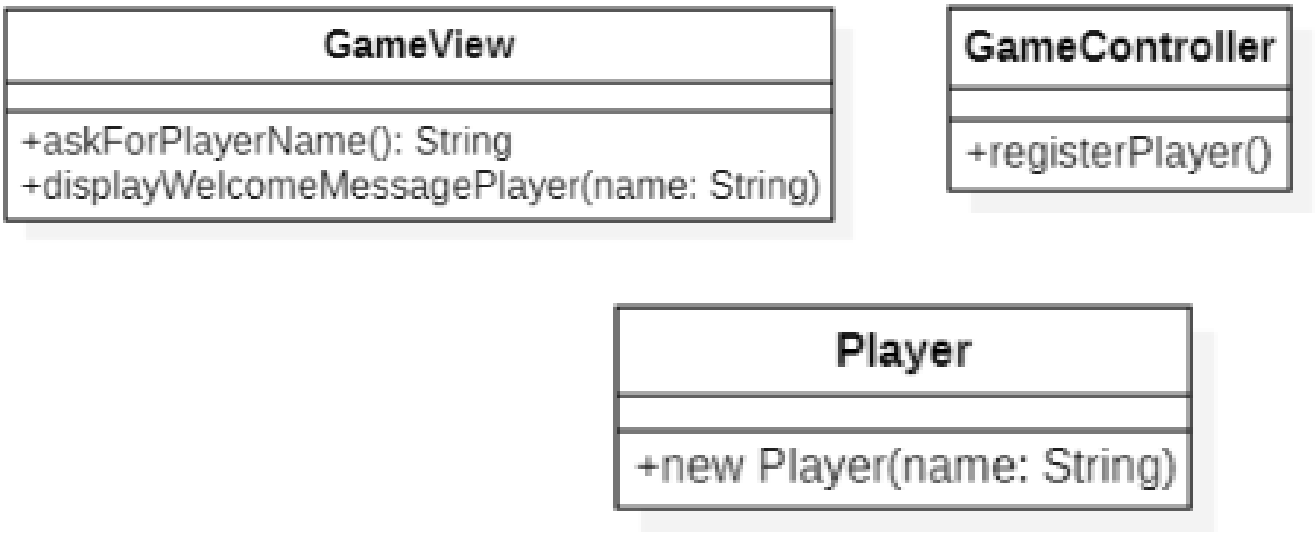
Votre équipe de développement, vous demande maintenant
de concevoir **cette application à l'aide du pattern MVC.**

⇨ La première étape de votre phase de conception consistera donc à modéliser le scénario *nominal* de la phase d'analyse sous forme de diagramme de séquence.

Après quelques itérations, voici un exemple de diagramme de séquences que vous pourriez obtenir



⇨ La seconde étape de votre phase de conception consistera à compléter le diagramme de classes à partir du diagramme de séquences précédemment proposé.



❑ **Phase d'implémentation :** (**Comment** le faire)

Une fois la conception finalisée, il est temps de passé à l'implémentation

⇨ Commençons par mettre en place les classes du pattern MVC dans les paquetages suivants :

- un paquetage **`control`** (rappelez-vous que le projet `wadcardgame` doit être codé en anglais 😉) qui contiendra la classe **`GameController`**
- un paquetage **`gui`** (ou **`view`**) qui contiendra la classe **`GameView`**
- un paquetage **`model`** qui contiendra les classes métiers qui seront découvertes au fur et à mesure de l'avancée du projet. Pour **mettre en œuvre** ce premier scénario, il est seulement nécessaire d'implémenter la classe **`Player`** pour le moment 😉

❑ Implémentation du **Contrôleur** : **`GameController`** dans un paquetage **`control`**

```
public class GameController {

	GameView view;
	Player player;

	public GameController(GameView view) {
		this.view = view;
	}


	public void startGame() {
		view.displayWelcomeMessage();
		this.registerPlayer();
	}

	public void registerPlayer() {
		String name = view.askForPlayerName();
		this.player = new Player(name);
		view.displayWelcomeMessagePlayer(player.name());
	}


}
```

Par rapport au cours, nous avons ajouté la méthode **`startGame`** car nous ne voulons ***qu'un seul point d'entrée*** dans le contrôleur depuis la classe qui contient le **`main`** c.-à-d. la classe qui lance l'application

Rappelons que le **contrôleur** est celui qui va **orchestrer** l'application. Pour notre application de jeu, il joue en quelque sorte le rôle que pourrait jouer un ***maître du jeu*** (animateur du jeu) 😉

❑ Implémentation de la **Vue** : **GameView** dans un paquetage **gui**

```
import java.util.Scanner;
import warcardgame.control.GameController;

public class GameView {

	GameController controller;
	Scanner keyboard = new Scanner(System.in);

	public void setController(GameController gc) {
		this.controller = gc;
	}

	public void displayWelcomeMessage() {
		System.out.println("-----------------------------------------");
		System.out.println("--  Welcome in our wonderful cardgame!  --");
		System.out.println("-----------------------------------------");
	}

	public String askForPlayerName() {
		System.out.println("Enter Player Name: ");
		String name = keyboard.nextLine();
		return name;
	}

	public void displayWelcomeMessagePlayer(String name) {
		System.out.println("Welcome  ! Let's start !" + name);
	}




}
```

⇨ Il est également nécessaire d'implémenter, dans un paquetage **application**, une classe **CardMain** qui est responsable de lancer l'application (classe qui contient la méthode **main**, point d'entrée dans un programme Java) et qui permet :

- de **mettre en place une vue** (qui fait l'interface avec l'utilisateur)
- de **mettre en place contrôleur** (qui orchestre l'application)
- et d'instaurer une **communication bidirectionnelle entre ces deux composants** pour permettre l'interactivité de notre application 😉

```
public class CardMainMVC {

	public static void main(String[] args) {
		GameView gameView = new GameView();
		GameController gameController = new GameController(gameView);
		gameView.setController(gameController);
		gameController.startGame();
	}
}
```

❑ Implémentation de la **logique métier** (classe métier **`Player`**) dans un paquetage **`model`**

```
public class Player {

	private final String name;



	public Player(String name) {
		this.name = name;
	}

	public String name() {
		return this.name;
	}









}
```

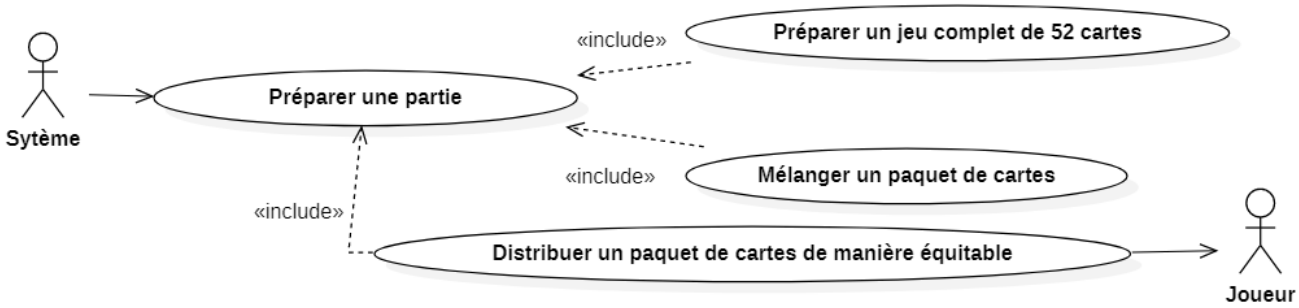
Si vous disposiez de tout le code précédent dans votre IDE préféré, vous pourriez lancer l'application et constater que le projet est bien fonctionnel et qu'il fournit, sur la console, l'affichage suivant :

```
-----------------------------------------
--  Welcome in our wonderful cardgame!  --
-----------------------------------------
Enter Player Name:
Isa
Welcome! Let's start Isa!
```

Développement de la fonctionnalité suivante : Préparer une partie

❑ Phase d'Analyse autour de la fonctionnalité :
Comprendre le comportement de la fonctionnalité :

⇨ La première ébauche d'analyse réalisée lors de l'élaboration du diagramme des cas d'utilisation a amené votre équipe à modéliser le cas d'utilisation **Préparer une partie** de la manière suivante :



⇨ **Le scénario nominal** du cas d'utilisation/fonctionnalité **Préparer une partie** a été ensuite retravaillé dans une autre séance de travail par votre équipe de développement et il en a résulté l'*artefact* (document) suivant :

1. **Le système** (re)constitue le jeu initial de 52 cartes (c.-à-d. un jeu complet)
2. **Le système** mélange le jeu initial de 52 cartes
3. **Le système** demande à chaque joueur de faire en sorte de disposer d'un emplacement vide destiné à accueillir un nouveau paquet de cartes
4. **Le système** distribue un paquet de cartes équitable à chaque joueur
   - **3.1 Le système** pioche une carte dans le jeu initial de 52 cartes
   - **3.2 Le système** attribue cette carte à un joueur
   - **3.3 Le système** répète les points 3.1 et 3.2 pour chaque joueur tour à tour et cela jusqu'à épuisement des cartes du jeu initial
5. **Le système** affiche un message qui indique que la partie va pouvoir commencer puisque toutes les cartes ont été distribuées.

⇨ **Mise à jour du glossaire métier** (et donc des termes métiers à utiliser dans le code projet 😉)
Remarque : Le **glossaire métier** est également couramment appelé aujourd'hui ***U***biquitous ***L***anguage (langage omniprésent). Il permet de s'assurer une compréhension commune et non ambiguë des termes métiers entre toutes les *parties prenantes* du projet (client, représentant du métier, développeurs…). Dans **un développement de qualité**, ces termes (appelé communément **terminologie métier**) doivent se retrouver dans toutes les phases du développement (artefacts produits lors de l'**analyse**, de la conception ainsi que dans le code, les tests, la documentation… 😉)

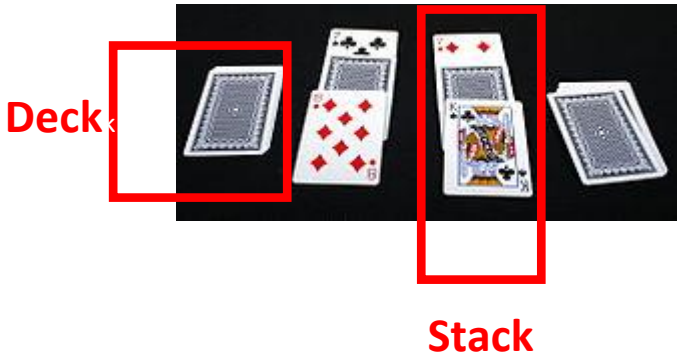
A la (re)lecture des règles du jeu et suite à une *quick **design** session* avec vos collègues (une rapide séance collaborative de discussion et de réflexion autour de la ***conception***), vous avez identifié trois paquets de cartes différents qui sont utilisés pour jouer à la bataille et qui ont des rôles (responsabilités) différentes.

Pour garder une trace de votre réflexion commune et ***aligner toutes les parties prenantes du projet sur une compréhension commune non ambiguë de ces concepts***, votre équipe a complété **le glossaire métie**r de la manière suivante :

➔ **Le jeu de cartes initial (standard deck)** est le paquet de cartes initial constitués de 52 cartes. Il est créé une fois en début de partie et n'est utilisé qu'une seule fois en début de partie pour distribuer les cartes de manière équitable et aléatoire entre les joueurs

➔ **Le paquet de cartes d'un joueur (deck)** est le paquet de cartes utilisé par un joueur tout au long de la partie. Il change d'état à chaque tour, puisqu'à chaque tour au moins une carte est retirée (plus s'il y a une bataille). Si le joueur gagne le tour, des cartes sont également ajoutées à ce paquet. Le nombre de cartes présentes dans ce paquet permet de déterminer si le joueur est éliminé (si le paquet est vide) ou si le joueur a gagné (si le paquet contient les 52 cartes du jeu de cartes initial).

➔ **La main (ou tas) d'un joueur (stack) : est un nouveau paquet propre à chaque tour.** Au début de chaque tour, la main doit être vide, puis une ou plusieurs cartes sont posées (suivant si une bataille doit avoir lieu). Les cartes sont ensuite retirées de la main pour être redistribuées au joueur gagnant le tour.



➔ **Mélanger un jeu de cartes (shuffle)** est une action réalisée avec un jeu de cartes où les cartes sont mélangées de manière aléatoire afin de garantir que leur ordre ne soit pas connu avant de distribuer ou de tirer les cartes.

➔ **Distribuer les cartes (deal)** implique de donner les cartes aux joueurs de manière équitable et aléatoire.
⇨ Lorsque vous dites "distribuer un paquet de cartes **équitable** à chaque joueur", cela signifie que chaque joueur reçoit un ensemble de cartes distribuées de manière impartiale et sans partialité. Cela garantit que chaque joueur a les mêmes chances de succès pendant la partie.

➔ **Piocher (draw)** fait référence à l'action de prendre une carte du jeu.

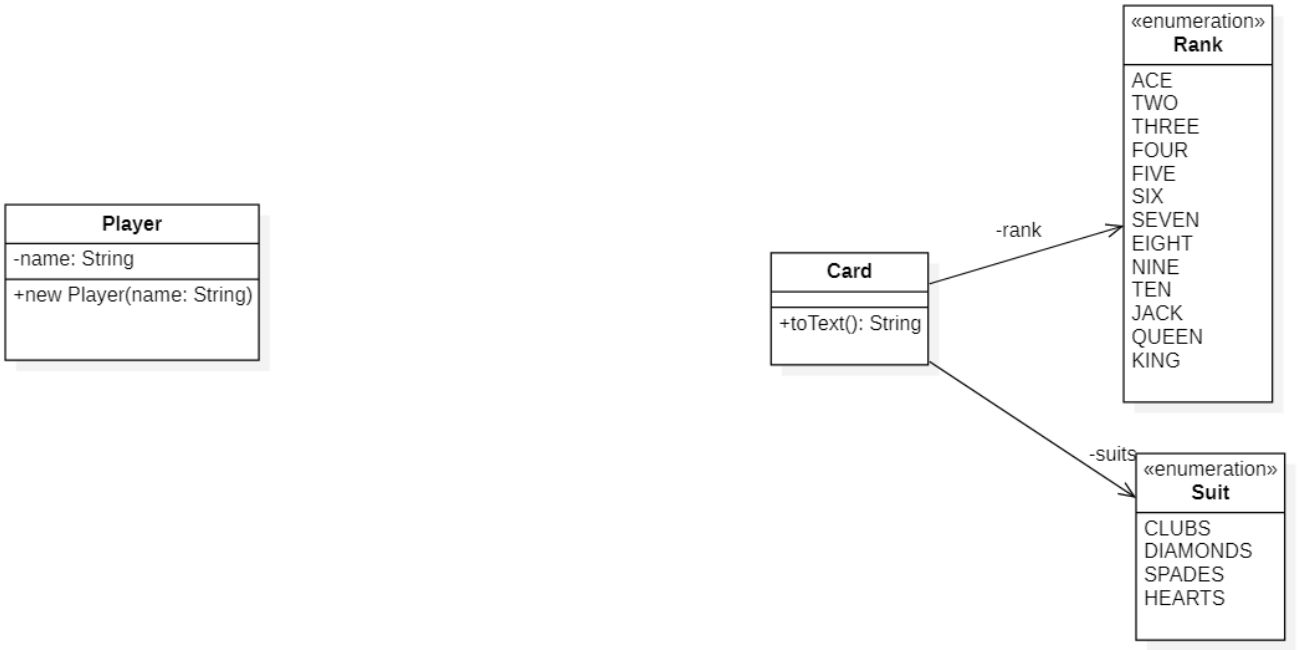
A vous de jouer !

❑ Phase de Conception :

⇨ A partir du scénario proposé précédemment, proposez **un diagramme de séquence**, modélisé à l'aide des stéréotypes de Jacobson, respectera les règles du pattern MVC et fera apparaître de nouvelles classes métier qui devront respecter la **terminologie métier définie dans le glossaire.**

⇨ Une fois, le diagramme de séquence terminée, compléter le **diagramme de classes ci-dessous uniquement avec les classes métiers** (c-a-d uniquement les classes du paquetage **model**) que le diagramme de séquence vous a permis d'enrichir ou de découvrir. En effet :

- Le TP *Pas de jeu sans carte !* vous a permis de commencer à vous focaliser sur l'élément de base de cette application : la carte à jouer, puisque sans carte, pas de jeu à développer ! et d'implémenter à cette occasion la classe **Card** et les énumérations **Suit** et **Rank**
- et la classe **`Player`** a été découverte lors du développement de la fonctionnalité précédente et son implémentation vous a été donnée en début d'énoncé 😉



Remarques :

- Les méthodes **`equals`**, **`hashCode`** et **`toString`** ne sont pas volontairement représentés sur ce diagramme de classes => c'est du sucre syntaxique Java (technique), pas du métier !
- Les *getteurs* auraient pu apparaître, mais nous avons volontairement choisi de ne pas les mettre pour ne pas alourdir la lisibilité du diagramme. Pour respecter le principe d'encapsulation, cela va de soi que les *getteurs* doivent être implémentés dans les classes métiers 😉

❑ Phase d'Implémentation :

Une fois la conception finalisée, il est temps de passé à l'implémentation

❑ Complétez directement (sur cette feuille) l'implémentation du **Contrôleur** : **`GameController`** donné ci-dessous pour que les objets de cette classe puissent mettre en œuvre le comportement de la fonctionnalité décrite dans le diagramme de séquence précédent.

Remarques :

⇨ Considérez que l'appel à la fonctionnalité *Préparer une partie* sera mis en œuvre dans le contrôleur par un appel à la méthode **`setUpTheGame`**

⇨ Pour simplifier le développement du MVP, votre équipe vous demande de gérer uniquement 2 joueurs pour commencer :

- le premier rejoindra la partie conformément au cas d'utilisation précédent
- le second ne sera autre que le système qui sera directement instancié dans le contrôleur rejoindra la partie conformément au cas d'utilisation précédent

Ces modifications ont été déjà réalisées dans le code donné ci-dessous, c'est pourquoi vous devez repartir de ce code pour poursuivre votre implémentation 😉

⇨ D'après le diagramme de séquences, quelle(s) nouvelle(s) méthode(s) apparaissent dans la classe **`GameController`** et de quelle manière ?

```
public class GameController {

	GameView view;
	Player player;
	Player system;

	public GameController(GameView view) {
		this.view = view;
	}


	public void startGame() {
		view.displayWelcomeMessage();
		this.registerPlayer();
		this.system = new Player("system");


	}

	public void registerPlayer() {
		String name = view.askForPlayerName();
		this.player = new Player(name);
		view.displayWelcomeMessagePlayer(player.name());
	}
```

```
} // fin GameController
```

❑ Complétez l'implémentation du **Vue** : **`GameView`** donnée en début d'énoncé pour que les objets de cette classe puissent mettre en œuvre le comportement de la fonctionnalité décrite dans le diagramme de séquence précédent.

❑ Enrichissez l'implémentation du paquetage **`model`** en complétant les classes métiers existantes (vous pouvez directement compléter la classe Player donné en début d'énoncé).
Pour les plus rapides, commencez à implémenter les nouvelles classes **apparues dans le diagramme de classes métiers que vous avez modélisé précédemment**. Rappelez-vous que les classes **`Card`** et les énumérations **`Suit`** et **`Rank`** ont déjà été implémentées dans un TP précédent.
Pas besoin de les ré-écrire 😉

❑ Il est à noter que la classe qui permet de lancer l'application **CardMainMVC** (qui contient le **`main`**) reste inchangée suite à l'implémentation de cette nouvelle fonctionnalité 😉

Annexe : Quelques mots sur le pattern MVC

La mise en place d'une interactivité a une incidence sur l'architecture logicielle d'une application.
Il est bien sûr évident qu'il faut éviter d'écrire toutes les classes au même « niveau » et qu'il faut organiser son application en « packages » (couches) afin de faciliter le développement (extensions et changements plus faciles si les tâches sont *bien* réparties) et la maintenance de l'application…

Choix d'une architecture logicielle :
Il est important lors de la mise en place d'une architecture de ne pas « réinventer la roue », mais de s'appuyer sur des patterns (modèles de programmation) robustes et simples qui ont fait leur preuve.
Le **pattern MVC** (**M**odèle **V**ue **C**ontrôleur) est le plus populaire et le plus couramment utilisé dans le développement d'applications proposant une interface graphique.

Le *pattern MVC* permet de répartir les responsabilités du système en trois parties
⇨ **le Modèle** correspond **aux classes métier**. Il contient les **données** et implémente, au travers de **méthodes**, les ***règles métiers*** (logique métier) qui déterminent le comportement du système selon les besoins du client (bien souvent dans un paquetage **`model`** ou **`metier`**)
⇨ **le Vue** propose une (re)présentation visuelle du système l'utilisateur : elle permet **d'afficher les données** du modèle pour l'utilisateur et **propose des interfaces de saisie**. La Vue est ainsi **responsable de gérer les interactions avec l'utilisateur** (bien souvent dans un paquetage `gui` ou **`IHM`**)
⇨ **le Contrôleur** est en fait le cœur de l'application.
Le contrôleur agit sur demande de l'utilisateur dans la Vue et effectue alors les actions nécessaires sur le Modèle.
Tout l'intérêt du Contrôleur est que la Vue n'a pas le droit *d'attaquer d*irectement le Modèle. Ainsi, pour que la Vue ne *voit* pas le Modèle, **le contrôleur sert de pont entre les deux**. Le contrôleur veille à ce que les demandes de l'utilisateur soient correctement exécutées, en modifiant les objets du modèle et en mettant à jour la vue.
On pourrait considérer le Contrôleur comme **le chef d'orchestre** des **interactions entre l'utilisateur et le système** dont la responsabilité est de gérer le **flux de l'application :** ce qui explique pourquoi un contrôleur est habituellement représenté au centre du diagramme de séquence.
(bien souvent dans un paquetage **`control`** ou **`application`**)

Bon à savoir …
Il est à noter qu'au fil des années des variantes ont été proposées autour du MVC à quelques différences près comme par exemple le **MVP** ou le **MVVM** , que vous verrez surement dans la suite de vos études 😉 :
⇨ Le pattern **M**odel **V**iew **V**iew **M**odel (**MVVM**) :
Le pattern MVVM a été créé par deux architectes chez Microsoft. À l'instar du MVC, cette architecture vise à séparer le code concernant l'interface du code logique de l'application. La différence se trouve au niveau du **ViewModel.** Contrairement au Controller du MVC, le ViewModel sert de lien bidirectionnel entre l'interface. C'est ce qu'on appelle ***le data binding***.
Dans le patron MVVM il y a une communication bidirectionnelle entre la vue et le modèle, les actions de l'utilisateur entraînent des modifications des données du modèle
⇨ Le pattern **M**odel **V**iew **P**resenter (**MVP**) :
Dans le pattern MVP, le contrôleur est remplacé par une présentation. La présentation est créée par la vue et lui est associée par une interface. Les actions utilisateur déclenchent des événements sur la vue, et ces événements sont propagés à la présentation en utilisant l'interface.
(Extraits https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur et https://welovedevs.com/fr/articles/mvc/)