



# W3 - Tests

---

W-WEB-310

## my\_tcg

---

initiation aux tests unitaires



# my\_tcg

delivery method: tcg on Github  
language: javascript  
build tool: no need here



- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

## INTRODUCTION

Ce sujet sera l'occasion d'aborder une notion très importante dans le développement d'application moderne, les tests unitaires. Pour ce faire vous allez développer (la partie logique métier) un jeu de type « Collectible Card Game ». Les tests unitaires permettront de garantir la qualité du code.

## RESTRICTIONS

Toutes les technologies (JS) sont autorisées, toutefois, si certains outils/technologies sont indiquées au fil du sujet, vous devez les utiliser.

## CONTEXTE

---

Souhaitant vous concentrer sur la conception des mécaniques de votre jeu, vous avez délégué la partie visuelle à un tiers développeur. Par ailleurs, et comme vous avez pu le constater les technologies web évoluent très vite. Vous décidez donc que le projet sera codé en EcmaScript 6 (aussi appelé ES6). Une fois qu'il a fini son travail, le développeur en charge de la partie visuelle vous a livré un pack de scripts (jointe au sujet).



Vous devez avant de commencer ce projet, effectuer le Bootstrap qui lui est dédié.

## PROJET

---

### PREMIÈRE PARTIE : PRÉAMBULE

---

Le framework « Jasmine » promeut la méthode « Behavior Driven Development » (ou « BDD »), qui consiste à écrire les tests au fur et à mesure que l'on décrit le comportement souhaité de l'application (avant d'avoir écrit du code applicatif à proprement parler). Vous irez vous renseigner sur l'intérêt de cette méthode.

Pour ce projet, vous serez principalement évalué sur votre respect de cette méthode, qui pourra être constaté entre autres lors des différents follow-up du module.



Il est inutile de continuer si vous n'avez pas réussi à lancer vos tests unitaires avec Jasmine. (cf. Bootstrap)

## DEUXIÈME PARTIE : MES PREMIERS TESTS UNITAIRES

---

Conformément, à la philosophie de la méthode BDD, nous allons commencer par écrire des tests en suivant les spécifications suivantes :

- Une seule classe par fichier
- Le nom du fichier sera écrit en « camelCase » et doit avoir le même nom que la classe qu'il contient (ex : pour la classe EventManager, le fichier s'appelle eventManager.js)
- Les classes que vous allez créer dans cette partie du sujet concernent toutes la logique métier de votre application. Elles devront donc toutes être placées dans le dossier « **src/models/** »



Savez-vous où sont rangés les classes JS avec Babel ?

## CLASSE DECK

Permet de gérer les opérations courantes sur un paquet de carte.  
Doit implémenter les méthodes : `shuffle`, `insertAt`, `draw`, `getCardsCount`.  
Doit avoir la propriété : (array) `cards`.

### Détails des méthodes :

*constructor(object config) :*

- Accepte un objet en premier argument. Cet objet doit avoir une propriété `cards` contenant un tableau (de cartes) qui constitueront les cartes du paquet (le deck).

*bool shuffle() :*

- Doit mélanger les cartes du paquet, retourne `true` si le paquet a été mélangé. `false` sinon.

*bool insertAt(mixed card, [int position]) :*

- Doit permettre d'ajouter une carte (premier paramètre) dans le paquet à une position donnée en deuxième paramètre. La rajoute à la fin du paquet si la position n'est pas donnée.

*mixed draw() :*

- Doit permettre de piocher la première carte du paquet. Retourne la première carte du paquet et l'enlève du deck. Retourne `false` s'il n'y a pas de cartes dans le paquet.

*int getCardsCount() :*

- Retourne le nombre de cartes actuellement dans le deck.

Exécutez les tests. Si vous n'avez pas d'erreurs (erreur de syntaxe, mauvaise utilisation du framework, etc...), le rapport d'exécution (ce qui s'affiche) devrait vous dire les tests « FAIL ».



En BDD, il faut écrire les tests avant d'écrire les méthodes afférentes. Pour ce premier exercice (et seulement cet exercice), rien ne vous empêche de commencer par écrire les méthodes pour ensuite tester l'écriture de vos tests.

Finissez d'implémenter la classe Deck (dans le fichier `src/models/deck`). Une fois que vos tests « pass » (indique « SUCCESS »), et tant qu'il en sera ainsi, cela signifiera que vous n'avez pas à vous soucier de la classe Deck, elle fait ce qu'on attend d'elle (donc aussi les différentes parties de son code). Pour preuve, si vous retournez et exécutez le nouveau code dans un navigateur, en cliquant sur l'un des decks, vous devriez être en mesure de faire piocher une carte à l'un des joueurs.



Il est inutile de continuer si vous n'avez pas réussi à faire passer tous vos tests.

Nous allons donc passer à la suite, **en commençant d'abord par écrire les tests correspondants (aux spécifications) des différentes classes...**



Vous devez écrire les tests AVANT d'implémenter les méthodes afférentes !!

**Notes :**

- Pour la suite vous pouvez, au choix, écrire tous les tests (de toutes les classes), puis implémenter (toutes) les classes, ou implémenter les classes au fur et à mesure (une fois que les tests correspondant ont été écrits). Quoi qu'il en soit, relier « l'intelligence de jeu » (vos classes) et l'affichage (le code qui vous ai fourni) comme pour le Deck, n'est prévu que dans la 3e partie du sujet.
- Certaines classes sont déjà créées, il s'agit alors de les modifier / compléter selon les spécifications.

Comme dans toute conception en POO, les objets auront un rôle précis et délimité. Nous allons donc faire des classes pour chaque « aspect » du jeu.



## CLASSE HAND

---

Permet de gérer la main d'un joueur.

Doit implémenter les méthodes suivantes : `addCard`, `removeCard`, `getAllCards`, `getCardsCount`.

Doit avoir les propriétés suivantes :

- (int) `limit`.
- (array) `cards`.

### **Détails des méthodes :**

*constructor(object config)*

- Accepte un objet en premier argument. Cet objet doit avoir une propriété `cards` contenant un tableau (de carte) qui constitueront les cartes de la main.
- On pourra préciser une limite maximum de carte qu'il est possible d'avoir en main. Pour ce faire, il faudra préciser une propriété « `limit` » dans l'objet passé en paramètre au constructeur.
- Si cette limite n'est pas précisée, la valeur par défaut de cette limite sera 7.

*bool addCard(object card)*

- Ajoute une carte (qui est passé en paramètre) à la fin de la liste des cartes déjà dans la main. Si tout s'est bien passé, la fonction retourne `true`, sinon `false`.
- Aucune carte ne doit être ajoutée si la limite est déjà atteinte, la méthode doit retourner `false`.

*bool removeCard(int position)*

- Retire de la main la carte positionnée à l'index passé en paramètre. Retourne la carte s'il n'y a pas eu de souci, `false` sinon.

*array getAllCards()*

- Retourne un tableau contenant toutes les cartes de la main.

*int getCardsCount()*

- Retourne le nombre de carte actuellement dans la main.

## CLASS BOARD

---

- Permet de gérer un board (l'ensemble des cartes mises en jeu).
- Doit étendre la classe Hand.



## CLASS CEMETARY

---

- Permet de gérer un cimetière.
- Doit étendre la classe Deck.

## CLASS PAWN

---

Classe dont héritera tout ce qui pourra combattre.

Doit étendre la classe EventManager.

Doit implémenter les méthodes suivantes : `getLife`, `getStrength`, `getDef`, `attack`, `receiveAttack`.

### Détails des méthodes :

*constructor(int life, int strength, int def)*

- Accepte 3 entiers en paramètre, le premier étant le niveau de vie initial, le second la force, le dernier la défense.

*getters : int getLife(), int getStrength(), int getDef()*

- Permet de lire les valeurs des différents attributs de l'objet.

*bool attack(Pawn target)*

- Accepte en paramètre un objet (ci-après désigné par « target ») qui sera lui aussi une instance de Pawn (ou d'une classe dérivée).
- Attaquer consiste à invoquer la méthode `receiveAttack` de l'objet `target` avec en seul paramètre une instance de l'attaquant (l'objet attaquant lui-même).

*bool receiveAttack(Pawn opponent, bool strikeBack = false)*

- Cette méthode doit décrémenter la vie de l'objet courant de la valeur de la force de l'attaquant (passé en premier paramètre), puis effectuer une contre-attaque (appeler la méthode `receiveAttack` de l'attaquant avec `true` en second paramètre).
- A l'exécution de cette méthode si le 2nd paramètre est présent et qu'il vaut « true » (il s'agit d'une contre-attaque), on décrémentera la vie de la valeur de la défense de l'(contre-)attaquant, non de la valeur de la force, et on ne fera pas de contre-contre-attaque.

## CLASS PLAYER

---

Permet la gestion d'un joueur.

Doit étendre la classe Pawn.

Doit implémenter les méthodes suivantes : shuffle, draw, playCard, discard, attack.

Doit avoir les propriétés suivantes : (Deck) deck, (Board) board, (Hand) hand, (Cemetery) cemetery.

### Détails des méthodes :

*constructor(object config)*

- Accepte un objet de configuration en paramètre. Cet objet, pour initialiser le deck du joueur, doit avoir une propriété deck qui sera soit un tableau, soit une instance de la classe Deck.

*bool shuffle(string type = "deck")*

- Accepte en paramètre une chaîne de caractère servant à désigner quel paquet mélanger.
- Les valeurs possibles sont « deck » et « cemetery ».
- Si un paquet a été mélangé, la méthode retourne true, dans tous les autres cas, elle retourne false

*bool draw()*

- Doit piocher la première carte du deck du joueur et l'ajouter à la main du joueur. Retourne la carte piochée si toutes les opérations se sont bien passées, false sinon.

*bool playCard(int position)*

- Retire la carte, à la position indiquée en paramètre, de la main du joueur et l'ajoute à son board. Retourne true en cas de succès, false sinon.

*bool discard(int position)*

- Retire la carte, à la position indiquée en paramètre, de la main du joueur et l'ajoute à son cimetière. Retourne true Si toutes les opérations se sont bien passées, false sinon.

*bool attack(int position, Pawn target)*

- Doit déclencher une attaque avec la carte du board désignée par le premier paramètre, et cibler l'adversaire passée en second paramètre.

## CLASS GAME

Permet l'encapsulation du jeu dans une seule instance.

Doit étendre la class EventManager.

Doit implémenter les méthodes suivantes : proxy, getTurn, changeTurn.

### Détails des methodes :

*constructor(object config)*

- Accepte en paramètre un objet avec 2 propriétés « up » et « down » chacune d'elle devra représenter un des adversaires.

*string getTurn()*

- Retourne une chaine qui indique le tour en cours (« up » ou « down »)

*string changeTurn()*

- Change le tour courant et retourne le nouveau tour (« up » ou « down »)

*bool proxy(string side, string action, mixed payload)*

- Exécute la méthode indiquée par le paramètre « action » pour le joueur désigné par le paramètre « side ». Le « payload » sera passé en paramètre à la méthode appelée.
- Si tout s'est bien passé la méthode proxy retourne le résultat de la méthode invoquée, false sinon.

## TROISIÈME PARTIE : LET'S PLAY

Vous avez désormais toutes les cartes en main pour implémenter un jeu de type « Collectible Card Game ».

Un peu d'explication sur le fonctionnement de la partie graphique de l'application pourra s'avérer utile. De la même façon que nous avons découpé la logique de jeu de façon à encapsuler les unes dans les autres les différentes parties de l'application, le code qui gère l'interface graphique est, de la même façon, découpé selon les différents « composant » et encapsulé les un dans les autres.

Les écouteurs d'événements DOM (clic par exemple) sont ajoutés au niveau du composant, puis, s'il y a lieu les « événement » sont remontés au composant parent. Les événements peuvent être remontés jusqu'à la racine des composants (l'arène). C'est ce dernier (et lui uniquement) qui devra communiquer avec le modèle, puis éventuellement (en fonction des retours du modèle) lancer à son tour un « événement » qui sera écouté par les composants enfants.



Pour commencer à développer votre jeu, fixez-vous des objectifs modestes dans un premier temps, par exemple, afficher les cartes de la main du joueur courant dans votre console, etc... Ensuite, abordez les mécaniques de jeu de la plus simple à la plus complexes : ajouter des cartes au deck, piocher 7 cartes en début de jeu, afficher les PV des joueurs, permettre à une carte d'attaquer un joueur, etc...



Il est inutile de continuer si vous n'avez pas réussi à afficher les informations de jeu courante (hand, board, cemetery, etc...) dans l'arène.

La partie graphique étant bien séparée de la logique de jeu, les « points de contact » étant identifiés et limités (seul l'arène doit communiquer avec le modèle), la logique de jeu peut être aisément portée côté serveur pour permettre de jouer à 2. Les tests unitaires nous permettront de nous assurer qu'une fois le code porté côté serveur, la logique n'aura pas changé.

Pas de restriction quand aux règles du jeu, vous pouvez implémenter les règles qui vous plaisent, attention toutefois à conserver les précédentes (ce sont elles qui comptent le plus dans la notation).