

# Identification de l'étudiant

---

- Amine Fanid
- amine.fanid.1@ens.etsmtl.ca
- AU97120
- AmineFanid

## Squelette pour un API simple dans Node, Express et TypeScript

---

### Introduction

Ce squelette est proposé pour commencer les projets en LOG210. Il possède les qualités suivantes:

- il est simple pour les débutants en LOG210
  - il n'y a pas de framework pour le front-end ni pour la persistance, mais ça n'empêche pas d'ajouter ces dimensions.
  - il est seulement [REST niveau 1](#), mais ça n'empêche pas de modifier l'API pour qu'il soit [REST niveau 3](#).
- il est orienté objet (avec TypeScript)
- il contient des tests pour l'API (avec Jest et Supertest)
- il fait une séparation entre les couches présentation et domaine, selon la méthodologie de conception du cours LOG210 (Larman)
- il fournit une structure (en Bootstrap et Pug) permettant de gérer les connexions d'utilisateur et les vues selon le type d'utilisateur
- il fonctionne sur Windows 10 (et probablement d'autres systèmes d'exploitation avec Node)

### D'où vient l'idée de base pour ce squelette?

Le code d'origine a été expliqué dans ce [texte de blogue](#).

Dans le cadre du cours [LOG210 de l'ÉTS](#), nous utilisons la méthodologie documentée par [Craig Larman dans son livre \*Applying UML and Patterns\*](#). Ce livre documente beaucoup de principes avec des exemples en Java, qui n'est plus à la mode comme à l'époque où le livre a été écrit.

Pourtant, il est encore possible de suivre cette méthodologie avec des technologies modernes comme JavaScript, Node.js, surtout en utilisant TypeScript. Cependant, il n'est pas évident de trouver des exemples de ces technologies qui respectent les éléments clés de la méthodologie de Larman: la séparation des couches (présentation, domaine) avec les opérations système et les classes du domaine.

Ce squelette montre ces aspects importants, dans le contexte du *Jeu de dés*, qui est l'exemple utilisé dans le chapitre 1 du livre du cours. Nous avons modifié l'exemple pour le rendre un peu plus complexe (plusieurs opérations système). Les diagrammes (faits avec [PlantUML](#)) sont présentés plus bas dans la partie Artefacts.

L'éditeur [Visual Studio Code](#) est très utile, mais n'est pas nécessaire avec ce squelette.

## Voulez-vous utiliser ce squelette?

1. (Créer une fork et) Cloner
2. Installer [node](#)
3. Installer les dépendances node - `npm install`
4. Compiler - `npm run build`
5. Lancer serveur de développement - `npm start`
6. Accéder à la page template de l'application - <http://localhost:3000>

► Regarder exemple de la fonctionnalité

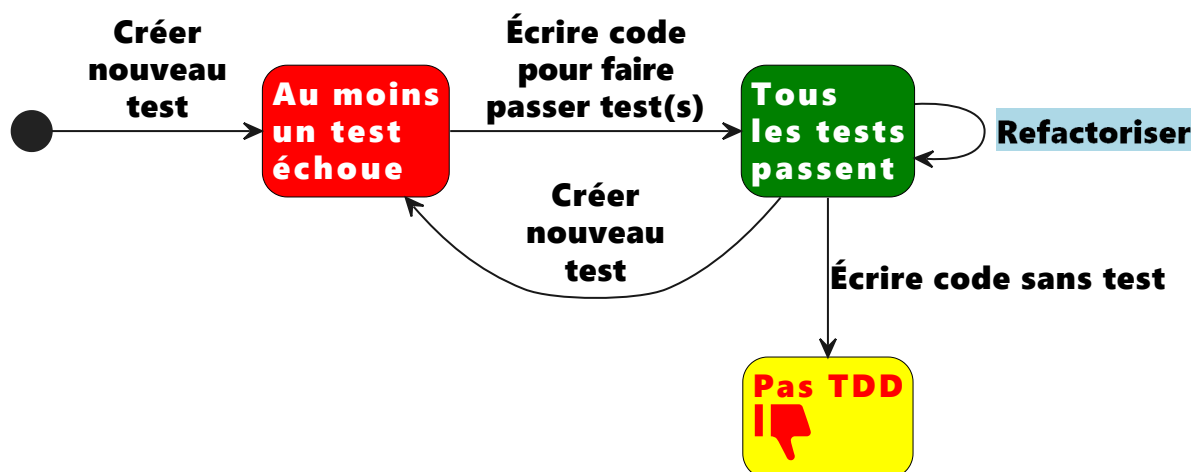


7. Lancer les tests (pas besoin de lancer le serveur d'abord) - `npm test`

## Développement piloté par les tests (TDD)

► Plus de détails

### États du TDD (Développement piloté par les tests, anglais: Test-driven development)



Le développement piloté par les tests (Test-Driven Development, TDD) est une façon de développer des logiciels en commençant par les tests. Il y a plusieurs avantages de cette façon de faire et ce squelette supporte la méthodologie.

Le TDD suit un cycle particulier, comme vous pouvez voir à l'image plus haut:

1. Écrire un nouveau test
2. Exécuter le test (qui échouera)
3. Écrire juste assez de code pour faire passer le test
4. Refactoriser le code (et les tests) au besoin, et recommencer

Il y a des tests pour tous les appels de l'API du serveur web, mais on devrait également faire des tests pour les autres classes (par exemple, des tests unitaires des classes du domaine).

## Support pour débogage

### ► Plus de détails

Ce squelette offre la possibilité de déboguer le code du serveur à l'aide de points d'arrêt placés à l'intérieur des fichiers TypeScript.

Voici comment il est possible de déboguer le projet à l'aide de différents environnements de développement.

### Débogage avec Visual Studio Code

VS Code offre la possibilité d'ajouter des configurations d'exécution à l'aide d'un fichier local. Ce fichier doit être nommé `launch.json` et être placé dans un dossier nommé `.vscode` à la racine du projet.

On peut utiliser ce fichier afin de créer des configurations d'exécution de débogage pour le projet. Un exemple de contenu pour ce fichier pourrait être :

```
{  
  "version": "0.2.0",
```

```
"configurations": [  
  {  
    "command": "npm start",  
    "name": "Debug",  
    "request": "launch",  
    "type": "node-terminal"  
  },  
  {  
    "command": "npm run start:watch",  
    "name": "Debug:Watch",  
    "request": "launch",  
    "type": "node-terminal"  
  }  
]  
}
```

Le lien suivant présente les subtilités de l'utilisation du fichier `launch.json` de VS Code dans le cadre d'un projet NodeJS : <https://code.visualstudio.com/docs/nodejs/nodejs-debugging>

Les configurations créées dans ce fichier peuvent ensuite être lancées à partir de l'onglet « Run » de la barre de régions à gauche de VS Code.

Sinon, il est possible d'attacher le débogueur de VS Code à une nouvelle exécution du projet sans avoir à créer un fichier `launch.json`. Il suffit de se rendre sur le fichier `package.json` et de cliquer sur le bouton « Debug » qui apparaît au-dessus de la section « Script ».

Une fois le débogueur attaché par l'une ou l'autre des méthodes présentées ci-dessus, l'exécution du code cessera lors de la rencontre d'un point d'arrêt sur un fichier TypeScript et il sera possible d'inspecter la valeur des variables visibles.

Pour plus d'informations au sujet de l'utilisation des breakpoints dans VS Code, voir [https://code.visualstudio.com/docs/editor/debugging#\\_breakpoints](https://code.visualstudio.com/docs/editor/debugging#_breakpoints)

## Débogage avec JetBrains WebStorm

Les mêmes instructions présentées dans cette section peuvent être utilisées pour déboguer le projet à partir d'un autre environnement JetBrains (comme IntelliJ) lorsque les plug-ins nécessaires sont installés.

Dans WebStorm, il est possible de créer une configuration d'exécution à l'aide de la liste déroulante à côté du bouton d'exécution « Run ». Pour ce projet, il est nécessaire de créer une configuration de type « npm » et de lui associer la commande « run » ainsi que le script « start » ou « start:watch ».

Le lien suivant présente les subtilités de cette configuration avec plus de détails : <https://www.jetbrains.com/help/webstorm/run-debug-configuration-npm.html>

Les configurations ainsi créées peuvent ensuite être lancées en mode débogage en cliquant sur le bouton « Debug » à droite du bouton « Start ».

Sinon, il est possible d'attacher le débogueur de WebStorm à une nouvelle exécution du projet sans avoir à créer une nouvelle configuration d'exécution. Il suffit de se rendre sur le fichier `package.json` et de cliquer

sur le bouton en forme de triangle vert à côté des scripts « start » ou « start:watch » et de sélectionner l'option « Debug ».

Une fois le débogueur attaché par l'une ou l'autre des méthodes présentées ci-dessus, l'exécution du code cessera lors de la rencontre d'un point d'arrêt sur un fichier TypeScript et il sera possible d'inspecter la valeur des variables visibles.

Pour plus d'informations au sujet de l'utilisation des breakpoints dans WebStorm, voir

<https://www.jetbrains.com/help/webstorm/using-breakpoints.html>

## Couplage souhaitable entre la couche Présentation et la couche Domaine

Dans un design favorisant la maintenabilité, on évite que la couche Présentation ait la responsabilité de gérer les événements système (opérations système). Larman présente dans son livre un exemple avec un JFrame (en Java Swing) à la figure F16.24. On l'adapte ici au contexte d'un service Web dans le framework Express (Node.js):

Séparation des couches  
(selon la Figure A17.24/F16.24 de Larman)



Dans la figure ci-dessus, l'objet **:JeuDeDes** (qui est un objet en dehors de la couche présentation) reçoit l'opération système **démarrerJeu(nom)** selon le principe GRASP Contrôleur. Ce squelette respecte cette séparation.

Voici la même figure, mais sous forme de diagramme de séquence avec l'acteur. On y voit tous les détails sordides de l'implémentation avec Node Express. **Notez que cette figure est présentée pour faciliter la compréhension seulement. On ne produit pas ce genre de diagramme dans la méthodologie.**



## Artefacts d'analyse et de conception

### Cas d'utilisation

#### Jouer aux dés

1. Le Joueur demande à démarrer le jeu en s'identifiant.
2. Le Joueur demande à lancer les dés.
3. Le Système affiche le nom du joueur et le résultat de la partie, ainsi que le nombre de parties et le nombre de fois que le Joueur a gagné. Pour un lancer, si le total est égal à sept, le Joueur a gagné. Dans tous les autres cas, il a perdu.

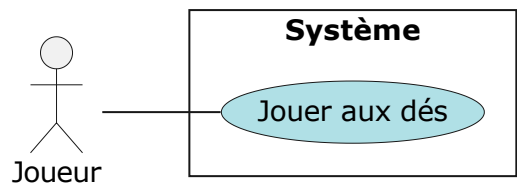
*Le Joueur répète l'étape 3 jusqu'à ce qu'il ait fini.*

4. Le Joueur demande à terminer le jeu.
5. Le Système affiche un tableau de bord avec les noms des joueurs et le ratio des parties gagnées (nombre de fois gagné / nombre de lancers).

#### Redémarrer

1. Le Joueur demande à redémarrer l'application.
2. Le Système termine tous les jeux en cours et redémarre l'application.

Diagramme de cas d'utilisation



Modèle du domaine

Modèle du domaine (adapté du Jeu de dés du Ch. 1 de Larman)

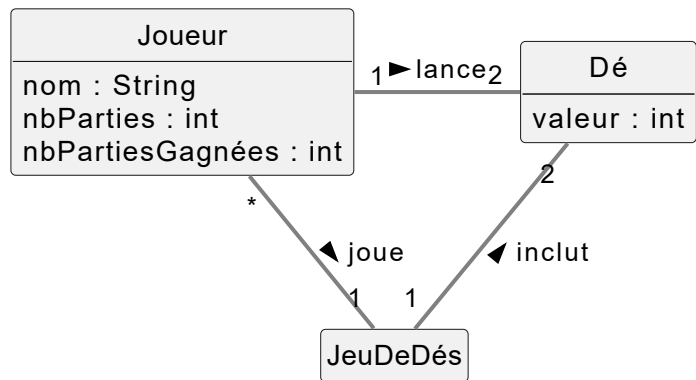
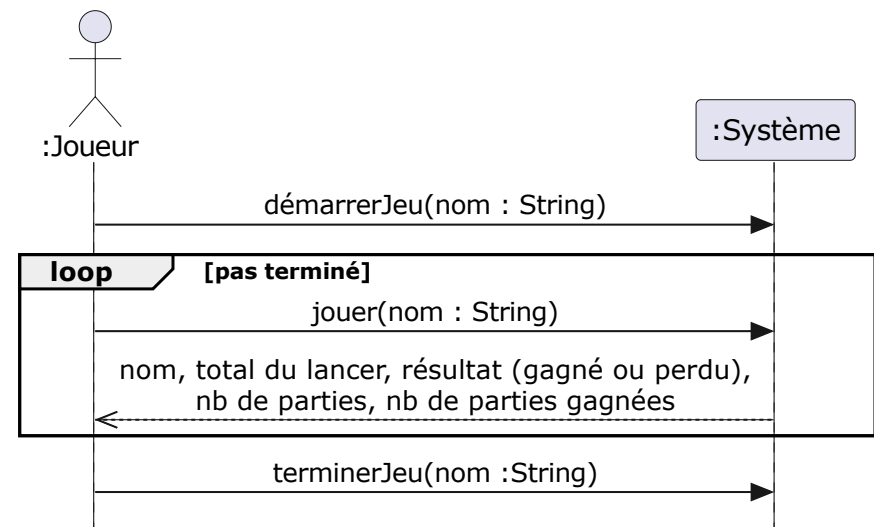


Diagramme de séquence système (DSS)

DSS pour un scénario adapté de *Jouer aux dés* (Ch. 1 de Larman)



Contrats d'opération et Réalisations de cas d'utilisation (RDCU)

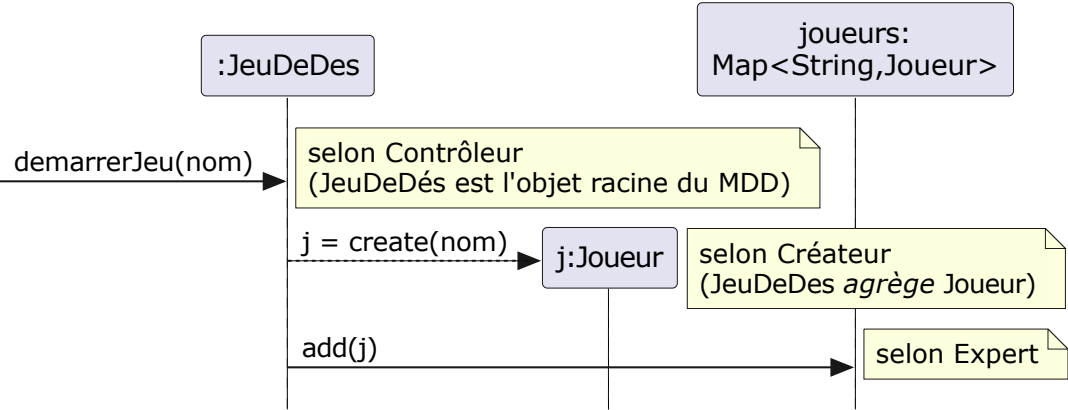
Opération: **démarrerJeu(nom:String)**

Postconditions

- Une instance j de Joueur a été créée
- j.nom est devenu nom
- j a été associé à JeuDeDes

RDCU

RDCU pour démarrerJeu



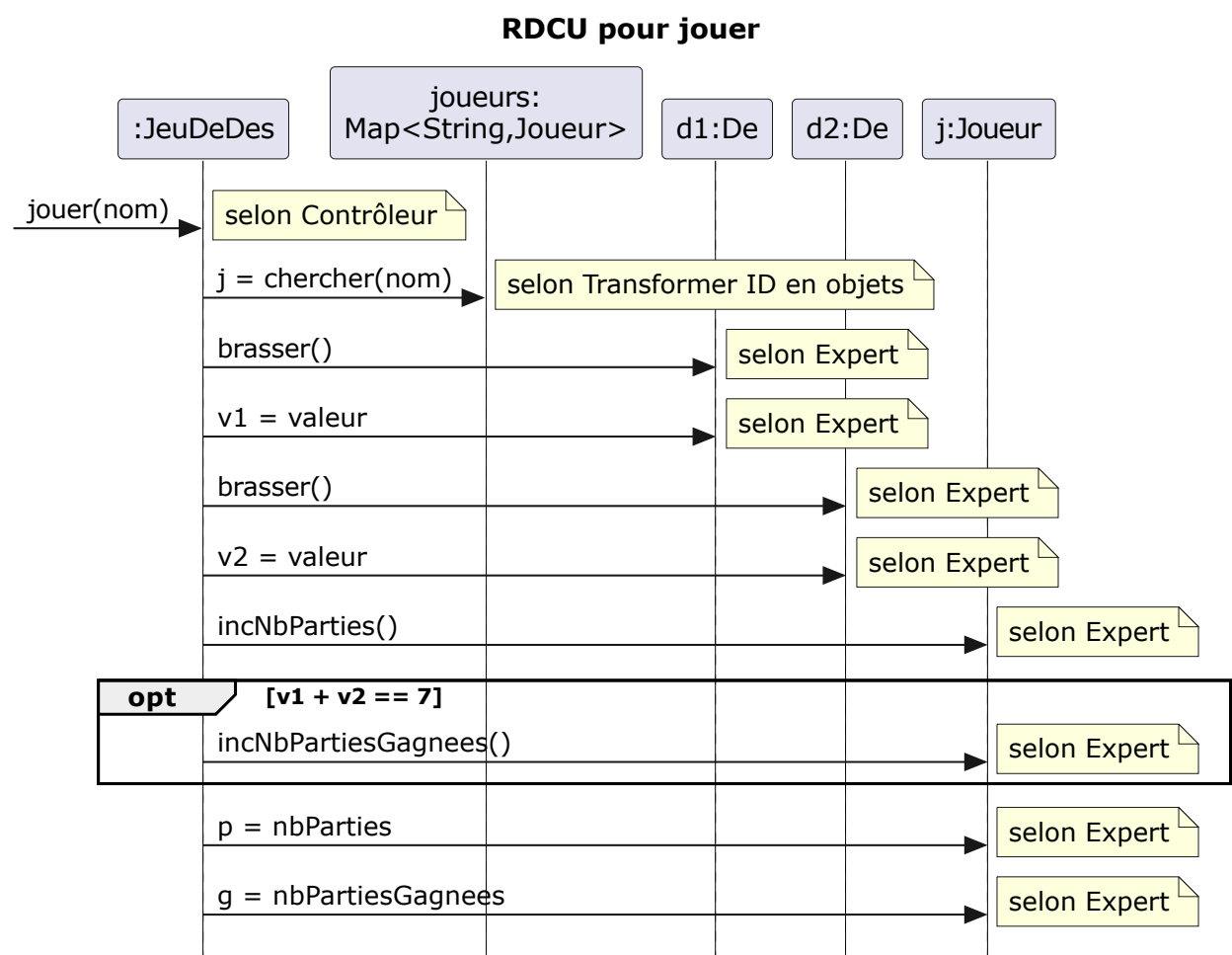
Opération: `jouer(nom:String)`

Postconditions

- `d1.valeur` est devenue un nombre entier aléatoire entre 1 et 6
- `d2.valeur` est devenue un nombre entier aléatoire entre 1 et 6
- `j.nbLancers` a été incrémenté sur une base de correspondance avec `nom`
- `j.nbLancersGagnés` a été incrémenté si la totale de `d1.valeur` et `d2.valeur` est égale à 7

RDCU

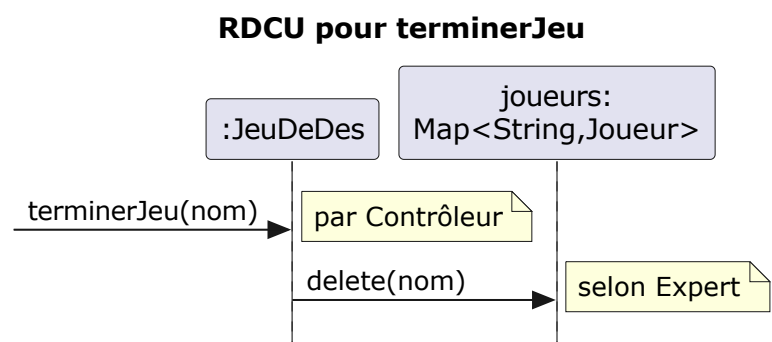




Opération: **terminerJeu(nom:String)**

Postconditions

- L'instance j de Joueur a été supprimée sur une base de correspondance avec nom



[README.md](#)

Diagramme de classes logicielles

