Université de Strasbourg

UFR de mathématique et d'informatique Université de Strasbourg

MASTER 2 — MENTION INFORMATIQUE

SCIENCES ET INGÉNIERIE DU LOGICIEL

ANALYSE ET ARCHITECTURE LOGICIELLE ORIENTÉE OBJETS

Projet PaperJS

BRUA Hugo

LAFORÊT Nicolas

PRINCELLE Maxime

Printemps 2022

Enseignant — **BECKER Stéphane**

UNIVERSITÉ DE STRASBOURG

ANNÉE UNIVERSITAIRE 2021 - 2022

Sommaire

1. Présentation du sujet - Énoncé

Vous êtes une équipe de développement intégré à Bidule Market, un gérant de la distribution qui souhaite à présent réaliser un système permettant de tracer les plans des magasins et de préparer leur implémentation. Le système doit être en ligne et permettre à la direction centrale d'établir les plans des différents magasins, puis aux équipes des magasins de choisir la façon d'implémenter les choses avant d'être finalement validé par la direction. Des intervenants extérieurs reçoivent les plans pour effectuer les travaux.

Un plan se compose :

- De l'enceinte extérieur du bâtiment, qui est inamovible.
- De cloison permettant de diviser l'espace.
- D'éléments fixes prédéterminés (bloc d'étagères, bloc caisse, …).
- D'éléments de tailles variables en fonction de la longueur que l'on attribue (le stand boulangerie).
- D'éléments suspendus au plafond genre des lampes, spots …
- D'extincteurs.
- Et de panneaux de marquage.

Pour l'édition on souhaite avoir toutes les facilités modernes ; pouvoir tracer le plan, supprimer des murs, poser des éléments fixes et une fonction undo/redo.

Vous avez la charge de l'architecture du système, à vous de jouer, d'établir un système, d'expliquer vos choix.

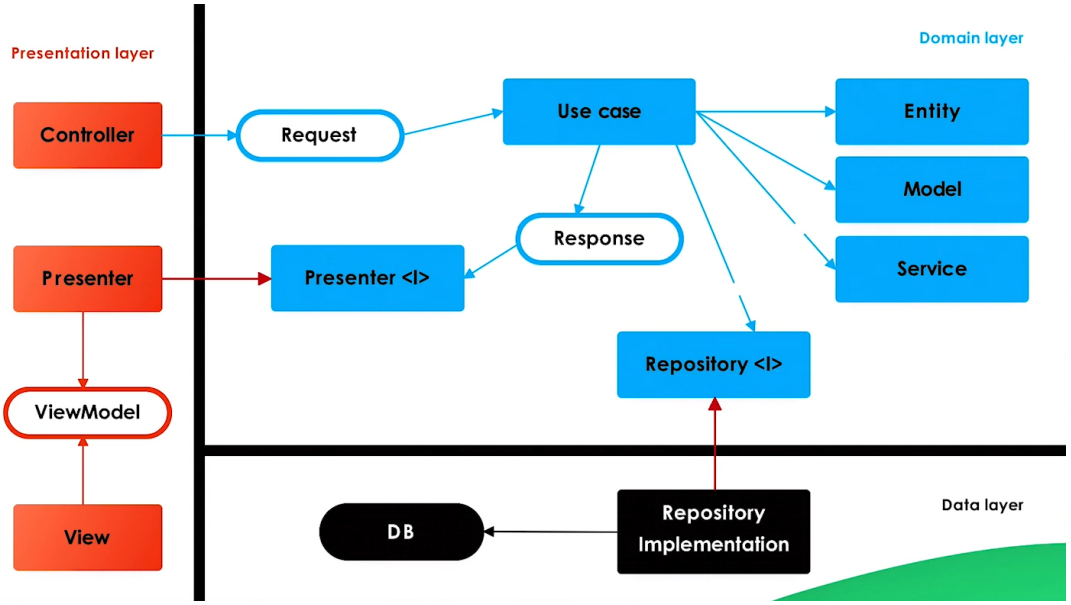
Avec l'énoncé, était fourni un dépôt de code initial constituant un projet **NodeJS** utilisant Typescript, avec un minimum de dépendance. La librairie **PaperJS** est utilisée afin d'afficher et d'interagir avec des formes géométriques depuis un élément HTML5 `<canvas />`. Le projet est servi grâce à **webpack**.

2. Architecture

2.1. Présentation générale

Au début de la conception de l'architecture de ce projet, nous avons fait le choix de découpler tout traitement dit "métier" de la librairie PaperJS. En effet, la base de code du dépôt initial pouvait mener à une conception où les traitement sont fortement couplés avec PaperJS ce qui n'est pas souhaitable et pas maintenable.

Nous avons ainsi décidé de réaliser une architecture dite **hexagonale**, c'est un patron d'architecture *(architectural pattern)* visant à créer des systèmes à base de couches et composants faiblement couplées. Ces couches sont connectées grâce à des ports représentés par des interfaces, qui sont implémentés par des adaptateurs.



Les composants sont facilement interchangeables, ainsi, le noyau ou domaine de l'application ne connaît pas l'implémentation concrète des composants tels que la base de données (système de persistance) ou l'interface *(web/command line/etc.)*.

Chaque composant est connecté aux autres par des "ports" qui représentent un canal de communication. Ces ports définissent une interface de programmation applicative (API) abstraite. Cela permet de développer les différents composants en isolation, seul l'API définit les éléments qui transitent entre les composants.

Les adaptateurs sont les implémentations des ports, ils représentent le ciment entre les composants avec l'extérieur du domaine (code non maîtrisé des libraires, interface, etc.).

La puissance de notre architecture est que nous pouvons remplacer n'importe quel système sans impacter le reste de la base de code, il est ainsi facile de changer la librairie PaperJS pour une autre, ou d'utiliser une base de données SQL au lieu du *localstorage* du navigateur, etc.

2.2. Couche domaine

La couche domaine représente le noyau de l'application, on y trouve le code dit métier, ce code ne dépend d'aucune librairie.

2.2.1. Model

L'objectif de la modélisation est que notre code parle la langue du métier. Le modèle représente les objets métier qui seront manipulés par notre code.

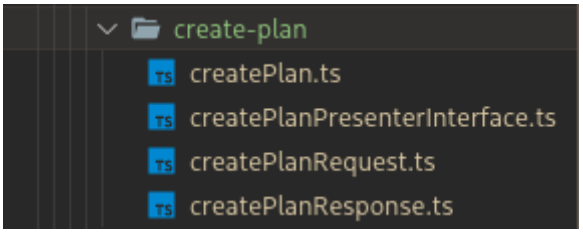
Dans notre cas précis, on y trouve :

- **Point** : représente des coordonnées dans un espace cartésien.
- **Shape** : représente la forme d'un *item* (rectangle, cercle, chemin, polygone).
- **Item** : est l'élément le plus basique, il représente tous les éléments qui peuvent constituer un plan architectural (fondation, cloison, escaliers, extincteur, table, etc.).
- **Layer** : contient une liste d'*items*, regroupe de manière logique les *items* (fondation, structure, objet au sol, objet suspendu).
- **Plan** : contient une liste des différents types de *layer*, il constitue le plan d'architecture d'un bâtiment.

2.2.2. Use cases

Afin de manipuler les éléments du modèle, le comportement et la modification de ces objets est traité à l'aide de composants appelés *use cases*. Un use case ne dispose pas de *state*, il récupère les données persistées à l'aide de repository qui lui sont passées dans son constructeur grâce à l'**inversion de dépendance**. En effet, le use case ne dépend pas directement des implémentations concrètes de ces repository, mais de leur interfaces.

Un *use case* est composé de quatres éléments :



- **Use case** : le traitement métier représentant ce cas d'utilisation. Cette *use case* dispose d'une méthode *execute* qui prend en argument un objet *Request* et une implémentation concrète de l'interface *PresenterInterface* et ne retourne rien (*void*), le retour se fait à travers l'interface *PresenterInterface*. Les *use cases* communique avec les systèmes de persistance des données (***repository***) à travers une ou plusieurs interfaces, dont les implémentation concrètes sont fournies au use case dans son constructeur en suivant le

principe d'**inversion de dépendance**.

```
export class CreatePlan {
    constructor(
        private repository: PlanRepositoryInterface,
        private historyrepository: HistoryRepositoryInterface
    ) {}

    async execute(
        request: CreatePlanRequest,
        presenter: CreatePlanPresenterInterface
    ): Promise<void> {
        const response = new CreatePlanResponse();

        // Validate plan name
        if (request.name.trim().length === 0) {
            response.isNameInvalid = true;
            presenter.presentCreatePlan(response);
            return;
        }

        const id = request.id || (await this.repository.nextId());
        const plan = new Plan(id, request.name);
        await this.repository.create(plan);
        response.plan = plan;

        await this.historyrepository.clear();

        presenter.presentCreatePlan(response);
    }
}
```

➢ **Request** : les données nécessaires à ce traitement, fournies par l'appelant.

```
export class CreatePlanRequest {
    constructor(public name: string, public id?: string) {}
}
```

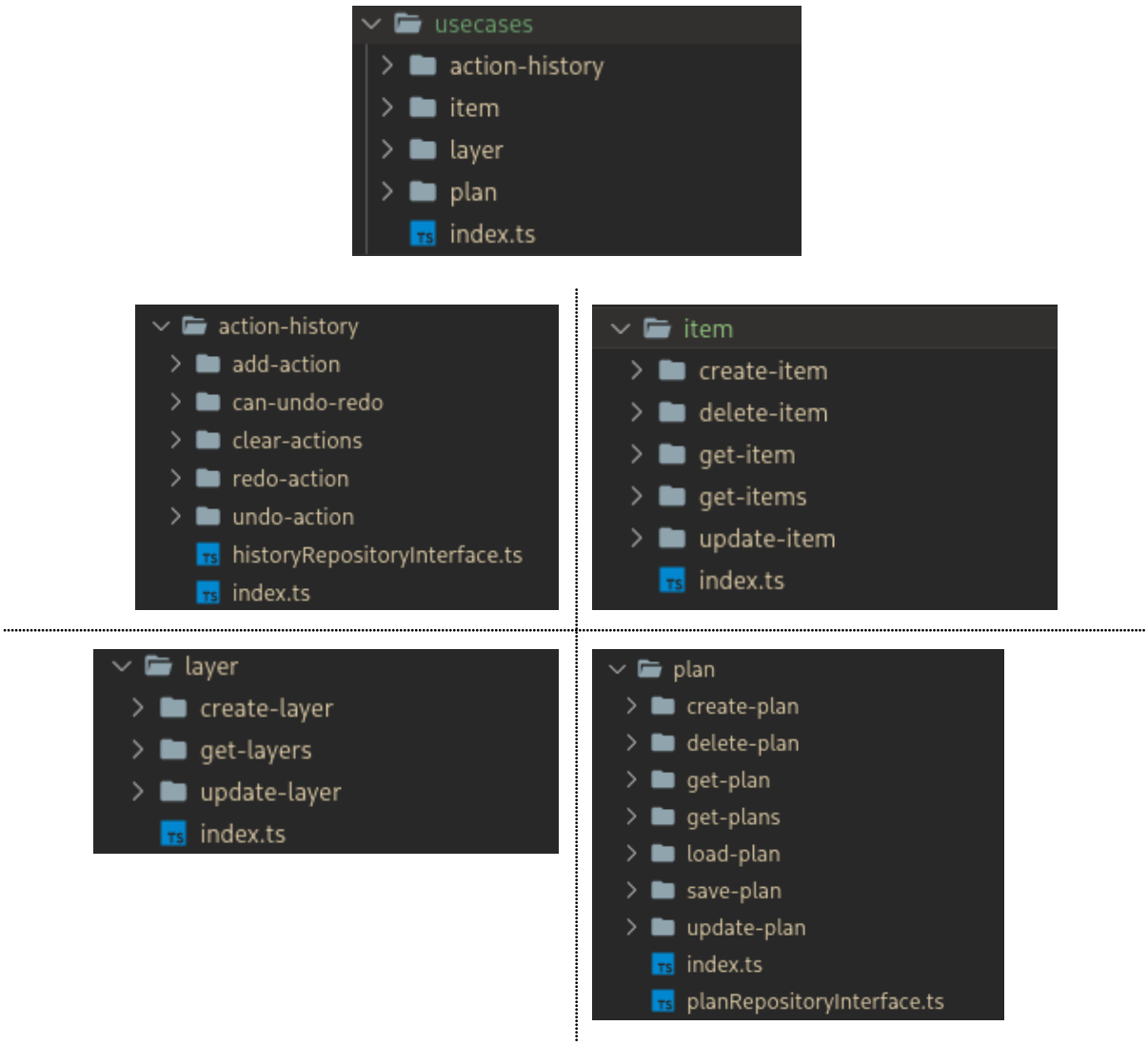
➢ **Response** : les données retournées par le traitement du *use case*, les retours d'erreurs se font grâce à des booléen dans la réponse.

```
export class CreatePlanResponse {
    plan?: Plan;
    isNameInvalid: boolean = false;
}
```

➢ **PresenterInterface** : cet élément n'est pas commun dans les architectures hexagonales. Nous avons fait le choix de ne pas retourner l'objet *Response* directement à l'appelant, mais plutôt de passer la réponse à cet objet *PresenterInterface* qui est aussi fourni par l'appelant. C'est une interface qui expose une unique méthode ayant pour but de traiter l'objet *Response* une fois le traitement terminé.

```
export interface CreatePlanPresenterInterface {
    presentCreatePlan(response: CreatePlanResponse): void;
}
```

Nous avons fait le choix de regrouper les *use cases* dans des répertoires reprenant les notions principales de l'application :



2.3. Couche infrastructure

La couche infrastructure, aussi appelée *data layer*, représente le stockage des données de manière plus ou moins persistante. Cette couche est composée des implémentations concrètes des ***Repository Interface*** utilisés par les *use cases* du domaine.

Notre choix d'architecture nous permet de changer très facilement de système de persistance sans impacter le reste de notre base de code. On pourrait choisir d'utiliser un système de base de données SQL, ou de la sauvegarde sur fichier, ou sur une API distante, etc.

2.3.1. LocalStorage - Plan

Nous avons décidé d'utiliser le *localstorage* du navigateur afin de stocker de manière "permanente" le plan du bâtiment réalisé par l'utilisateur.

C'est dans cette classe concrète de l'interface du repository responsable de la sauvegarde des données du plan, que nous avons la liberté de l'implémentation des politique de cache et d'écriture sur un support permanent.

Nous avons opté pour l'utilisation d'une variable locale qui stocke l'état courant du plan qui est en train d'être édité par l'utilisateur, agissant comme *cache*. Les données du plan sont enregistrées dans le *localstorage* uniquement lorsque l'utilisateur demande la sauvegarde de son plan via le *use case* **SavePlan**. Cela permet de ne pas écrire et lire dans le *localstorage* à chaque modification par l'utilisateur.

2.3.2. Cache - Undo/Redo

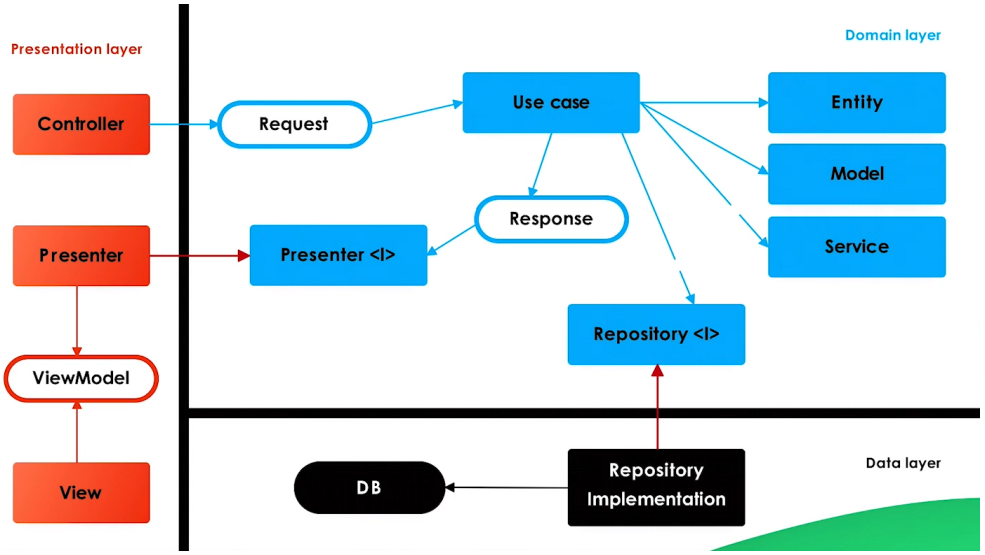
Pour ce qui est du repository responsable des actions undo / redo. Nous avons implémenté cette interface à l'aide de deux tableaux utilisés comme des **piles**. En effet, il n'était pas pertinent de sauvegarder les états de modifications intermédiaires dans le *localstorage*, les historiques d'actions ne sont quasiment jamais sauvegardés entre deux sessions de modifications sur tous les logiciels d'éditions.

Lorsqu'un utilisateur appelle des *use cases* qui modifient l'état du plan, ces *use cases* on la charge de sauvegarder les états intermédiaires dans ce repository afin de pouvoir annuler ou rétablir les actions précédentes.

Il est possible de faire en sorte d'avoir des piles de taille illimitées, mais cela pourrait poser des problèmes de performance après un grand nombre d'actions sur le plan. Nous avons donc mis en place une taille limite dans l'historique des actions qui peuvent être annulées et rétablies. Lorsque cette taille limite est atteinte, les actions les plus anciennes sont remplacées par les actions à ajouter à la pile afin de garder un historique d'action cohérent et continue.

2.4. Couche présentation

La couche présentation est responsable de l'affichage des données retournées par le domaine et de la création de requêtes aux différents *use cases* du domaine lors d'action réalisés par l'utilisateur.



2.4.1. Adapters

Lorsqu'on regarde le contenu du dossier ***frontend*** de notre projet, on peut y trouver un dossier ***interface-adapter***, il ne contient aucune dépendance autre que le domaine (pas même PaperJS).

C'est dans ce dossier que l'on implémente :

- ➢ les **contrôleurs** : responsable de la création des objets *Request* qui seront envoyés aux *use cases*.
- ➢ les **presenters** : responsable du traitement des objets *Response* fourni par les *use cases* après une requête. Dans notre implémentation, ce sont les presenters qui mettent à jour les objets *ViewModels* qui sont nécessaires aux *vues*.
- ➢ les **view models** : objets comportants les données nécessaires au remplissage des *vues* de l'application. Les view models sont des DTO *data transfer object*, ils ne contiennent aucune logique, aucune méthode, juste des champs.

2.4.2. Interface UI

Représenté par le répertoire ***vanilla-js*** présent dans le dossier ***frontend*** de notre projet, c'est ici que l'on retrouve les vues et composants qui forment l'interface utilisateur. Les *vues* utilisent les *view models* issues de la section précédente pour afficher l'état courant des données à l'utilisateur.

On retrouve la majeure partie du code présent dans le dépôt fourni avec le sujet du projet. En effet, c'est dans cette partie que nous allons dépendre de librairies graphiques telles que PaperJS.

La puissance de notre architecture se remarque à nouveau dans cette partie, en effet, il suffirait d'ajouter un dossier au même niveau que vanilla-js et d'implémenter uniquement les vues pour changer de librairie graphique ou pour ajouter une autre interface utilisateur telle qu'un outil de ligne de commande, ou même utiliser un framework tel que ReactJS ou VueJS, etc.

3. Réalisation

Nous avons développé un grand nombre de fonctionnalités dans la couche du domaine permettant la gestion de plusieurs plans. Pour chaque plan, la gestion de *layers* (création, mise à jour, suppression), ces éléments n'ont pas tous été utilisés au niveau de l'interface graphique.

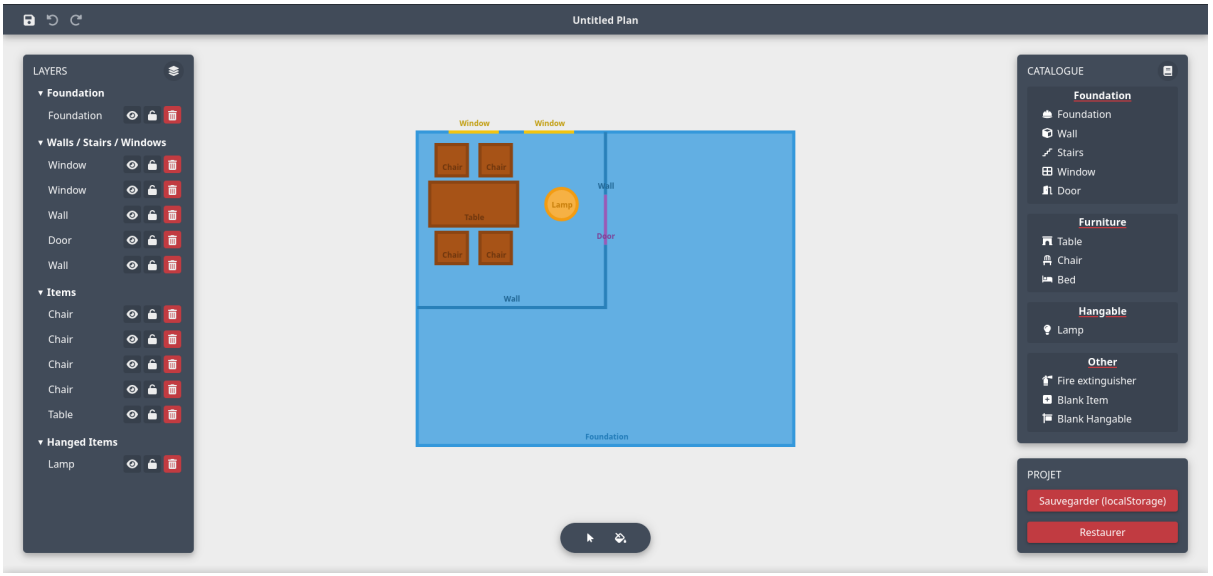
Nous avons mis en place des raccourcis clavier :

- **Sauvegarde le plan** : Ctrl + y
- **Undo** : Ctrl + z
- **Redo** : Ctrl + y

Il est aussi possible de renommer le plan depuis la barre située en haut de l'interface.

Le projet pouvait très facilement gagner en ampleur, il est toujours possible d'ajouter plus de fonctionnalités, mais à ce rythme, il est difficile de voir la fin, nous avons tendance à vouloir en ajouter toujours plus.

Nous avons donc misé sur la mise en place d'une architecture propre, pérenne, maintenable, extensible, modulaire et efficace, plus que sur l'ajout de fonctionnalités métier. Car l'ajout de fonctionnalités est "trivial" grâce à nos choix d'architecture. Après tout, *l'Unité d'Enseignement* est appelée "Analyse et architecture logicielle orientée objets".



Captures d'écran de l'interface du projet