Institut National des Sciences Appliquées de Rouen

ITI3

Projet Intégratif Développement Orienté Objet

Développement orienté objet du projet ITI Aventure

Auteurs :

Mohamed Aziz Ayari
maya00@insa-rouen.fr

Joachim Curtelin
joachim.curtelin@insa-rouen.fr

Hamza Fadili
hamza.fadili@insa-rouen.fr

Louis Lenoble
louis.lenoble@insa-rouen.fr

Jack Su
jack.su@insa-rouen.fr

Enseignants :

Robin Condat
robin.condat@insa-rouen.fr

Nicolas Delestre
nicolas.delestre@insa-rouen.fr

Géraldine Del Mondo
geraldine.del_mondo@insa-rouen.fr

Nicolas Malandain
nicolas.malandain@insa-rouen.fr

Laurent Vercouter
laurent.vercouter@insa-rouen.fr

30 mai 2025

Table des matières

1 Introduction **2**

2 Persistance BD **3**
2.1 Description et fonctionnement 3
2.1.1 Passage du modèle entité-association en modèle relationnel 3
2.1.2 EnregistreurBD 5
2.1.3 LecteurBD 5

3 Interprétation fichier structuré **6**
3.1 Lecture du format du fichier structuré 6
3.2 Écriture de la grammaire 6
3.3 Lecture de l'AST 7
3.4 Relation entre variable et entité du jeu 8

4 Design Patterns **9**
4.1 Description et fonctionnement 9

A Annexes BD **11**
A.1 Annexe : Exemple de code d'enregistrement 11
A.2 Annexe : Exemple de code de lecture 12

B Annexes Interpréteur **13**
B.1 Annexe : Exemple de fichier de configuration 13

Chapitre 1

Introduction

Dans le cadre de notre troisième année à l'INSA Rouen Normandie, nous avons été amenés à participer aux Projets Intégratifs (PI). Ces projets ont pour objectif principal de nous permettre de mettre en pratique l'ensemble des compétences acquises tout au long de notre cursus.

Le projet sur lequel nous avons travaillé vise à assurer la bonne persistance du jeu *ITI Aventure*, et s'articule autour de trois axes principaux : Tout d'abord, nous avons mis en place une solution permettant de sauvegarder et de charger une partie depuis une base de données (BD). Ensuite, nous avons conçu un interpréteur capable de charger une partie à partir d'un fichier structuré. Enfin, nous avons modifié l'architecture du projet afin d'intégrer le design pattern *Factory*, dans le but de généraliser et de simplifier la création de différents types d'aventures.

Chapitre 2

Persistance BD

Dans cette partie, notre objectif est d'établir un lien entre la logique métier du jeu et une base de données, afin de pouvoir sauvegarder l'état d'une partie. Pour ce faire, nous devons mettre en place un système de mapping objet-relationnel, permettant de faire correspondre les objets du jeu aux données stockées dans la base.

Nous utilisons le langage Java, qui propose l'API JDBC (Java Database Connectivity). Cette API permet à une application Java de se connecter à une base de données, d'exécuter des requêtes SQL, et de traiter les résultats de manière structurée.

2.1 Description et fonctionnement

2.1.1 Passage du modèle entité-association en modèle relationnel

La première étape consiste à établir un modèle entité-association. Celui-ci nous a été founit au début du projet. Nous devons donc le transformer en modèle relationnel. Pour cela, nous nous sommes du cours de BD1 à notre disposition.

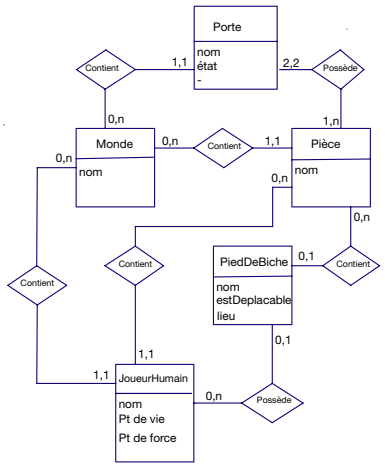

FIGURE 2.1 – Schéma entités-associations pour ITI Aventure

Explication des transformations :

1. Cas 1 : Entités en relation : Les objets qui rentrent dans le cas 1.
   - Monde d'attribut nom en MONDE(nom)
   - Porte d'attributs nom et état en PORTE(nom,etat)
   - PiedDeBiche d'attributs nom, estDéplaceable en PIEDEBICHE(nom,estDéplaceable)
   - JoueurHumain d'attributs nom, pV, pF, en JOUEURHUMAIN(nom, pV, pF)
   - Pièce d'attribut nom en PIECE(nom)
2. Cas 2 : Entités faibles en relations. Il n y'a pas d'entités faibles dans notre table EA.
3. Cas 3 : Association binaire 1-1 par clés étrangères. Les entités concernées sont les suivantes :
   - PORTE reçoit la clé de Monde comme attribut qui est nomMonde.
   - JOUEURHUMAIN la clé de Monde et la clé de PIECE comme attributs qui sont nomMonde et nomPiece.
   - PIECE reçoit la clé de Monde comme attribut qui est nomMonde.
4. Cas 4 : Association binaire M-N. Les entités concernées sont les suivantes :
   - PIECEPOSSEDEPORTE reçoit les clés de PORTE et PIECE comme clés, qui sont nomPiece et nomPorte. Il aurait été possible de transformer cette relation par le cas 3 : association binaire 2-2, en associant dans la table PORTE deux attributs, pieceA et pieceB.
   - PIECECONTIENTPDB reçoit comme clés, les clés de PIECE et PIEDDEBICHE et qui sont nomPiece et nomPiedDeBiche.
   - JOUEURPOSSEDEPDB reçoit comme clés, les clés de JOUEURHUMAIN et PIEDDEBICHE, qui sont nomJoueur et nomPiedDeBiche.

Voici le résultat final :

- **MONDE**(<u>nom</u>)
- **PORTE**(<u>nom</u>, etat, nom_monde)
- **PIEDDEBICHE**(<u>nom</u>, estDeplaceable)
- **JOUEURHUMAIN**(<u>nom</u>, ptVie, ptForce, nom_monde, nom_piece)
- **PIECE**(<u>nom</u>, nom_monde)
- **JOUEURPOSSEDEPDB**(<u>nom_joueur</u>, <u>nom_pied_de_biche</u>)
- **PIECECONTIENTPDB**(<u>nom_piece</u>, <u>nom_pied_de_biche</u>)
- **PIECEPOSSEDEPORTE**(<u>nom_porte</u>, <u>nom_piece</u>)

Maintenant que nous avons définit notre modèle relationnel, nous pouvons commencer à travailler avec nos tables SQL. Pour cela, nous initialisation les tables vides grâce à un fichier Initialisation.sql. Ensuite, nous avons développé la classe LecteurBD qui implémente l'interface Lecteur de notre projet ITI Aventure. Cette dernière s'occupe de lire chaque table (en envoyant une instruction SQL via un PreparedStatement) de chaque relation et de créer ou assigner les attributs de chaque objet. Nous avons procédé de la même manière pour la classe EnregistreurBD. Cette classe s'occupe de vider les tables à chaque instanciation, puis de récupérer chaque objet du monde et de les enregistrer dans la table correspondante.

2.1.2 EnregistreurBD

La classe EnregistreurBD a été utilisée pour sauvegarder l'état du jeu. Dans notre contexte, elle a permis d'enregistrer le monde du jeu, les pièces, les portes qui relient deux pièces, ainsi que les objets et entités tels que le joueur humain ou le pied-de-biche. Cette classe joue le rôle de pont entre les objets Java et la base de données, assurant la persistance des données. Elle utilise JDBC pour interagir avec la base et permet de traduire l'état du jeu en requêtes SQL.

Les étapes de l'enregistrement sont les suivantes :

— établir une connexion à la base de données.
— préparer la requête à l'aide de PrepareStatement (l'insertion dans notre cas).
— utiliser comme des méthodes comme setString ou setInt pour attribuer aux paramètres de la requête les valeurs des attributs à enregistrer (par exemple, le nom d'une pièce ou les points de force).
— exécuter pour insérer ou mettre à jour les données dans la base (executeUpdate).
— fermer la requête.
— fermer la connexion à la base de données.

Ces étapes sont répétées pour chaque entité du jeu (sauf la fermeture de connexion avec la BD), tout en respectant l'ordre d'enregistrement imposé par les dépendances entre les tables. (voir Annexe A.1)

2.1.3 LecteurBD

La classe LecteurBD est utilisée pour lire les données depuis la base de données. Elle permet de charger l'état du jeu précédemment enregistré, en reconstruisant les différentes entités et leurs relations. Cette classe joue un rôle essentiel de mapping entre les données persistées dans la base et les objets Java.

Les étapes du chargement sont les suivantes (voir Annexe A.2) :

— Se connecter à la base de données.
— Préparer la requête de recherche avec une instruction SELECT.
— Exécuter la requête et récupérer les résultats dans un ResultSet.
— Parcourir chaque ligne du ResultSet à l'aide de la méthode next().
— Récupérer les valeurs des colonnes avec des méthodes comme getString() ou getInt().
— Instancier les objets correspondants à partir des données récupérées.
— Fermer la requête.
— Fermer la connexion à la base de données.

Chapitre 3

Interprétation fichier structuré

3.1 Lecture du format du fichier structuré

Dans ce projet, une manière de charger des données du jeu est d'interpréter un fichier structuré avec un langage fonctionnel. C'est pourquoi dans cette partie, nous verrons comment nous avons fait pour décomposer ce type de fichier afin de l'implémenter dans notre jeu ITIAventure. Un fichier `exemple.cfg` nous est donné (voir Annexe B.1)

La structure du fichier se présente ainsi :

- Le fichier présenté est un fichier de configuration
- Chaque ligne de ce fichier de configuration est une affectation
- L'affectation fera appel à une fonction dont le retour sera stockée dans une variable
- Les paramètres de ces appels de fonction pourront être des identifiants, des constantes ou des appels de fonction
- Tout identifiant devra être affecté, et donc déclaré, avant d'être utilisé
- Les constantes pourront être :
  - `SUCCES, ECHEC, ENCOURS`
  - Des chaînes de caractères entourés de doubles guillemets
  - Des nombres entiers

Afin d'exploiter notre fichier de configuration, il est nécessaire d'avoir un outil qui permet d'interpréter ce langage. C'est pourquoi nous utilisons `JavaCC` (Java Compiler Compiler) qui permet de créer automatiquement un analyseur lexical et un analyseur syntaxique.

3.2 Écriture de la grammaire

À partir du fichier de configuration et des règles sur la structure du fichier, nous pouvons écrire la grammaire suivante (avec les unités lexicales) :

⟨CONFIGURATION⟩ : := AFFECTATION+
⟨AFFECTATION⟩ : := ID AFF APPEL_FONCTION
⟨APPEL_FONCTION⟩ : := IDF PARAG PARAMS PARAD
⟨ID⟩ : := [A-Z,a-z,_][A-Z,a-z,0-9,_]∗
⟨AFF⟩ : := "="
⟨VIR⟩ : := ","

⟨IDF⟩ : := "monde" | "piece" | "serrure" | "porte" | "pied_de_biche" | "cle" | "monstre" | "humain" | "cdf_vivant_dans_piece" | "cdf_vivant_possede" | "cdf_conjonction"
⟨PARAG⟩ : := "("
⟨PARAD⟩ : := ")"
⟨PARAMS⟩ : := PARAM (VIR PARAM)∗
⟨PARAM⟩ : := ID | CHAINE | NB_ENTIER | APPEL_FONCTION | CONST
⟨CONST⟩ : := "SUCCES" | "ECHEC" | "ENCOURS"
⟨CHAINE⟩ : := '"'LETTRE+'"'
⟨LETTRE⟩ : := caractère alphanumérique
⟨NB_ENTIER⟩ : := [0-9]+

Explication de la grammaire :

- ⟨**CONFIGURATION**⟩ : une configuration est composée d'une ou plusieurs affectations.
- ⟨**AFFECTATION**⟩ : une affectation associe un identifiant à un appel de fonction, séparés par un signe égal.
- ⟨**APPEL_FONCTION**⟩ : un appel de fonction commence par un identifiant de fonction, suivi d'une parenthèse ouvrante, d'une liste de paramètres, puis d'une parenthèse fermante.
- ⟨**ID**⟩ : un identifiant commence par une lettre ou un underscore, suivi de zéro ou une infinité de lettres, chiffres ou underscores.
- ⟨**AFF**⟩ : le symbole d'affectation est le signe égal `=`.
- ⟨**VIR**⟩ : les paramètres sont séparés par des virgules.
- ⟨**IDF**⟩ : les identifiants de fonction sont des mots-clés prédéfinis correspondant aux entités du jeu.
- ⟨**PARAG**⟩ et ⟨**PARAD**⟩ : parenthèses ouvrante et fermante pour entourer les paramètres d'une fonction.
- ⟨**PARAMS**⟩ : une liste de paramètres, séparés par des virgules.
- ⟨**PARAM**⟩ : un paramètre peut être un identifiant, une chaîne, un nombre entier, un appel de fonction (imbriqué dans une fonction), ou une constante.
- ⟨**CONST**⟩ : une constante est l'un des mots `SUCCES`, `ECHEC` ou `ENCOURS`.
- ⟨**CHAINE**⟩ : une chaîne de caractères est entourée de guillemets et contient une ou plusieurs lettres.
- ⟨**LETTRE**⟩ : une lettre est un caractère alphanumérique.
- ⟨**NB_ENTIER**⟩ : un nombre entier est composé d'un ou plusieurs chiffres.

3.3 Lecture de l'AST

Après l'analyse syntaxique du fichier de configuration, un Arbre Syntaxique Abstrait (AST) est généré automatiquement à partir de la grammaire définie dans `AnalyseurSyntaxique.jj`. Cet AST représente la structure du fichier, où chaque nœud correspond à une unité lexicale du langage.

Pour exploiter cet AST, nous avons implémenté le patron de conception `Visiteur` ainsi que l'interface `Visitable`.

Chaque classe représentant un nœud de l'AST implémente l'interface `Visitable`, qui définit une méthode `accepter`. Le visiteur contient alors une méthode `visiter` spécifique pour chaque type de nœud.

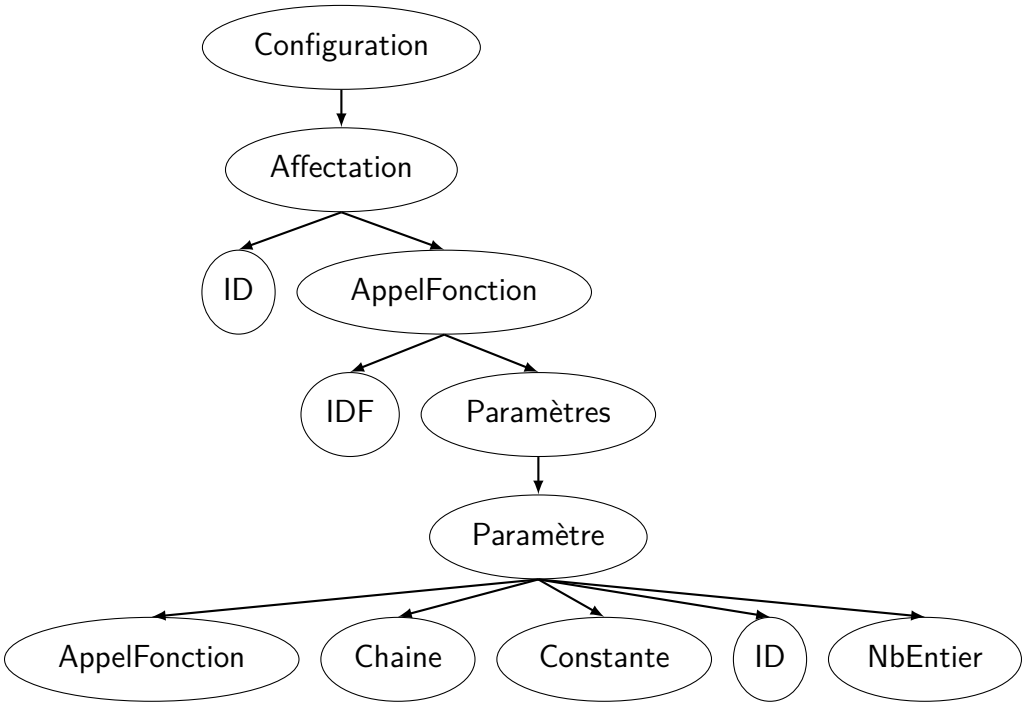

FIGURE 3.1 – Représentation de l'Arbre Syntaxique Abstrait

Ainsi, lors de la lecture de l'AST, il suffit de parcourir l'arbre en appelant la méthode `visiter` sur la racine, et le visiteur se charge de traiter chaque nœud selon sa nature.

Dans notre cas, notre classe qui implémente `Visiteur` sera un lecteur de jeu, nommé `LecteurAST.java`. Cette classe prendra comme paramètre dans son constructeur un fichier de configuration. Il ira visiter l'AST généré à partir de ce fichier, en parcourant chaque nœud. Ce lecteur joue alors un rôle d'analyseur sémantique et intervient dans la phase de traitement, en vérifiant et en instanciant les entités du jeu.

3.4 Relation entre variable et entité du jeu

Un problème rencontré dans l'implémentation du `LecteurAST.java` a été la gestion des variables déclarées et leurs entités dans le jeu.

En effet, lors de la lecture du fichier de configuration, chaque affectation crée une nouvelle entité du jeu (par exemple une pièce, une porte, une clé, etc.) et l'associe à un identifiant (variable). Pour assurer ce lien, nous avons utilisé une structure de type `HashMap<String, String>` appelée `tableDesNoms`, qui fait le lien entre chaque identifiant déclaré et le nom de l'entité correspondante.

Lors du parcours de l'AST, la méthode `visiter(Affectation aff)` extrait l'identifiant de l'affectation ainsi que le nom de l'entité (lorsqu'il s'agit d'une chaîne de caractères parmi les paramètres de l'appel de fonction). Si ce nom est trouvé, il est nettoyé de ses guillemets éventuels puis ajouté à la table de correspondance. Ainsi, à chaque nouvelle affectation, le couple identifiant-nom est enregistré dans la table. Cela permet, lors de l'utilisation ultérieure d'un identifiant comme paramètre dans un autre appel de fonction, de retrouver rapidement le nom associé à cet identifiant.

Chapitre 4

Design Patterns

Le but de cette dernière partie est de modifier l'architecture du projet ITI Aventure afin de généraliser et simplifier la création d'une aventure.

4.1 Description et fonctionnement

Pour être indépendant du type des classes, nous avons implémenté le design pattern "Abstract Factory" de la facon suivante : ITIAventureFactory est la factory depuis laquelle les autres héritent. Ensuite, Monde, Monstre, Pièce, Porte, Serrure, Clé sont non seulement des produits et mais aussi les produits desquels tous les autres produits héritent. La classe LecteurDescription utilise ainsi l'une ou l'autre Factory pour instancier les objets en fonction de la première ligne du fichier de description.

Pour mieux comprendre, voici le diagramme UML représentant l'utilisation de ce design pattern :

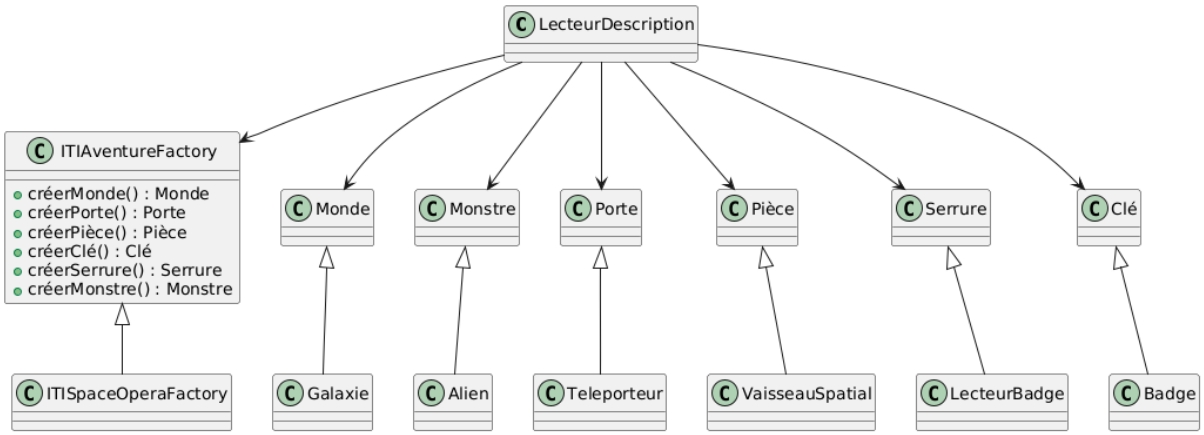

Figure 4.1 – Diagramme UML du design pattern factory pour ITI Aventure

On constate que pour créer une nouvelle aventure, il suffit simplement de développer une nouvelle classe (comme ITISpaceOperaFactory), chargée d'instancier les objets spécifiques, eux-mêmes hérités des éléments d'une aventure standard.

Conclusion générale

Ce projet nous a permis de mettre en pratique de nombreuses compétences acquises durant notre formation, en particulier autour du développement orienté objet, de la persistance des données et de la conception logicielle. Nous avons appris à manipuler une base de données relationnelle, à écrire un interpréteur avec JavaCC, à générer et parcourir un AST, et à structurer notre code avec des design patterns comme Factory ou Visiteur.

Grâce à ce travail, nous avons conçu une architecture claire, modulaire et facilement extensible, ce qui facilite l'ajout de nouveaux types d'aventures ou de nouvelles entités de jeu.

Pour aller plus loin, nous pourrions réfléchir à implémenter une interface Homme-Machine afin de rendre le jeu plus intuitif, ou encore à développer un outil rendant la création de fichiers de configuration plus facile. Nous pourrions alors nous aider de la bibliothèque graphique JavaSwing qui a été abordé en cours.

Nom :	Aziz	Joachim	Hamza	Louis	Jack
PARTIE BD					
Modèle EA vers modèle relationnel					
Classe EnregistreurBD					
Classe LecteurBD					
PARTIE COMPILATION					
Formalisation grammaire					
Classes de l'AST					
Classe LecteurAST					
PARTIE UML					
Classe ITIAventureFactory					
Classe ITISpaceOperaFactory					
Classe LecteurDescription					

FIGURE 4.2 – Matrice de répartition du projet

Annexe A

Annexes BD

A.1 Annexe : Exemple de code d'enregistrement

Voici un exemple de tiré de notre code qui illustre ce qui a été décrit précédemment. Cette méthode permet d'enregistrer un monde et les portes :

```
1  public void enregistrer(Monde monde, Collection<ConditionDeFin>
       conditionsDeFin) throws Throwable {
2      Connection connection = connexionBD(this.base);
3
4      PreparedStatement pstMonde = connection.prepareStatement("
           INSERT␣INTO␣MONDE␣VALUES␣(?)");
5      pstMonde.setString(1, monde.getNom());
6      pstMonde.executeUpdate();
7      pstMonde.close();
8
9      PreparedStatement pstPorte = null;
10     for (Porte porte : monde.getEntites().stream().filter(elt ->
           elt instanceof Porte).map(elt -> (Porte) elt).toList()) {
11         pstPorte = connection.prepareStatement("INSERT␣INTO␣PORTE
               ␣VALUES␣(?,␣?,␣?)");
12         pstPorte.setString(1, porte.getNom());
13         if (porte.getEtat() == Etat.OUVERT || porte.getEtat() ==
               Etat.FERME) {
14             pstPorte.setString(2, porte.getEtat().toString());
15         } else {
16             pstPorte.setString(2, Etat.OUVERT.toString());
17         }
18         pstPorte.setString(3, monde.getNom());
19         pstPorte.executeUpdate();
20         pstPorte.close();
21     }
22         connection.close();
23 }
```

A.2 Annexe : Exemple de code de lecture

Voici un exemple de notre code qui illustre ce qui a été décrit précédemment. Cette méthode (lecturePieces) permet de lire les données des pièces dans la base de données, puis de recréer les objets Piece correspondants.

```
1    private void lecturePieces() throws SQLException,
        ClassNotFoundException, ITIAventureException {
2        PreparedStatement pstPiece = connection.prepareStatement(
            "SELECT␣*␣FROM␣PIECE");
3        ResultSet tablePiece = pstPiece.executeQuery();
4        while (tablePiece.next()) {
5            String nomDePiece = tablePiece.getString("nomPiece");
6            System.out.println(nomDePiece);
7            Piece piece = new Piece(nomDePiece, monde);
8        }
9        pstPiece.close();
10   }
```

Annexe B

Annexes Interpréteur

B.1 Annexe : Exemple de fichier de configuration

```
m1 = monde("LeMondeImpitoyabledITI")
bureau_nicos = piece(m1, "BureauDesNicolas")
bureau_dir = piece(m1, "BureauDuDirecteur")
bureau_clement = piece(m1, "BureauDeClement")
bureau_laurent = piece(m1, "BureauDeLaurent")
couloir = piece(m1, "Couloir")
porte1 = porte(m1, "Porte1", bureau_nicos, couloir, serrure(m1))
porte2 = porte(m1, "Porte2", bureau_dir, couloir, serrure(m1))
porte3 = porte(m1, "Porte3", bureau_clement, couloir)
porte4 = porte(m1, "Porte4", bureau_laurent, couloir)
trape = porte(m1, "Trape", bureau_clement, bureau_dir)
passage_secret = porte(m1, "PassageSecret", bureau_nicos,
   bureau_laurent)
cle1 = cle(porte1, bureau_clement)
cle2 = cle(porte2, bureau_laurent)
etudiant = humain(m1, "Etudiant", 10, 12, couloir)
cdf1 = cdf_vivant_dans_piece(SUCCES, etudiant, bureau_nicos)
cdf2 = cdf_vivant_dans_piece(ECHEC, etudiant, bureau_dir)
```