VAN LIEDEKERKE Florian
BINGINOT Etienne
MAZUY Axelle

# Projet d'Application Informatique: Implémentation du jeu de société LogikVille

**Sommaire:**

# I) Présentation du projet

## A) Présentation du jeu

LogikVille est un jeu de logique visant à résoudre des énigmes. L'objectif est d'attribuer personnages et/ou animaux à la bonne maison.
Chaque énigme ne possède qu'une seule solution.
Chaque carte représente un niveau. Toutes les cartes sont numérotées de 1 à 84, dans un ordre croissant de difficulté.
Le nombre de maisons est identique au nombre de personnages et évolue entre 3 et 5. Chaque niveau peut comporter ou non des animaux.

L'objectif de ce projet était de réaliser une implémentation fidèle de ce jeu de plateau.

## B) Cahier des charges et contraintes imposées

Nous avons tout d'abord réalisé un cahier des charges regroupant la présentation que nous venons de faire du jeu LogikVille, ainsi que les contraintes imposées/requêtes client.
Ce cahier des charges comprend également des visuels graphiques des différents menus et interfaces du jeu.

Le cahier des charges étant réalisé en amont de la réalisation, beaucoup de choses ont pu évoluer au cours du projet. Notamment les concepts graphiques et les fonctionnalités possibles. En effet, les concepts graphiques se sont adaptés aux besoins pratiques mais aussi à un besoin esthétique. De plus, les fonctionnalités essentielles ont été priorisées pour permettre un jeu agréable à manipuler et efficace en termes de temps, ce qui laisse certaines fonctionnalités ajoutables en suspens.

Toutefois, chaque contrainte a été respectée et étudiée pour être réalisée de la meilleure manière possible, assurant à la fois modulabilité et ergonomie de l'application.

## C) Organisation des dossiers

Pour respecter l'architecture MVC, 2 types de dossiers se distinguent :
Les dossiers touchant à la vue (aspect graphique de l'application), et les dossiers touchant au modèle (le fond de l'application).
On peut donc tout d'abord présenter les dossiers model et view comportant l'essentiel du code. De plus, on peut retrouver le dossier main, contenant l'application principale ainsi que

les dossiers ressources_classic (contenant des ressources essentielles au fonctionnement de l'application) et le dossier utils comportant des outils utiles pour le code.

Dans le dossier model, on retrouve différents dossiers également gérant les parties essentielles en fond.
En effet par exemple on retrouve un package Save gérant la partie "sauvegardes" du jeu (les niveaux déjà implantés et ceux que l'on peut implanter soi-même).
Dans le package Entities, on retrouve la gestion des personnages et animaux du jeu, ainsi que les fichiers de thèmes pouvant être chargés pour les images et les rôles.
Dans le package Constraints enfin, on trouve les différentes contraintes imposées par les énigmes du jeu.
Les autres points importants se trouvent dans le dossier model : l'éditeur de cartes, la gestion d'un niveau, le modèle de gestion de l'ensemble de l'application, l'inventaire des entités, l'objet représentant la carte de jeu, la maison, et le solveur permettant de trouver la réponse aux énigmes.

Dans le dossier view se trouvent deux dossiers : gui comportant le code des menus graphiques et interfaces utilisateurs, et element gérant les composants graphiques affichables.
Dans gui, on retrouve les différents types de menus graphiques que l'on peut rencontrer allant de l'écran titre à l'écran de jeu, en passant par le menu d'édition, alors que dans element sont définies les composants graphiques comme les boutons spéciaux ou les composants représentant la carte ou les contraintes.

Enfin dans le dossier res sont stockées les ressources pouvant être modifiées par l'utilisateur : les fichiers de thèmes contenant le rôle et le lien vers l'image dans le dossier de l'entité.

Il est donc possible pour l'utilisateur de modifier le thème de son jeu par l'utilisation du menu d'option, qui en créant un thème téléchargera au bon endroit les images souhaitées et en créeraun simple fichier texte indiquant le type de l'entité (personnage ou animal) suivi de son nom et du chemin vers l'image.

# II) Organisation du travail

## A) Répartition des rôles

Afin d'être efficace sur notre travail, nous avons choisi de répartir les rôles selon différents axes afin de travailler parallèlement.
Ainsi, comme établi dans le cahier des charges, nous avons réparti de la manière suivante :

- Etienne BINGINOT était chargé de la vue et de l'aspect graphique de l'application, gérant les composants graphiques et l'interface utilisateur.
- Florian VAN LIEDEKERKE était chargé du modèle et notamment de la partie solveur, mais aussi des contraintes et de leur gestion. Il a également effectué une partie des tests sur le modèle.
- Axelle MAZUY était chargée du modèle, notamment de la partie sauvegarde et gestion des entités, mais aussi de la vue avec la gestion de l'interface utilisateur et des contrôleurs.

Nous avions donc chacun nos tâches attitrées tout en travaillant en binôme sur certaines tâches spécifiques.

Cette répartition nous a permis d'avancer relativement rapidement sur différents aspects majeurs de l'application tout en travaillant en groupe et en faisant des points réguliers.

## B) Organisation de la charge de travail

Pour aborder ce projet, nous avons tout d'abord évalué nos besoins matériels. Mais en ayant accès à la boîte du jeu, seul un espace d'échange et partage de code était nécessaire, c'est ainsi que le GitLab nous a été utile.
Ensuite, nous nous sommes réunis à trois pour réaliser un diaporama contenant l'ensemble des objets qui nous semblaient nécessaires après avoir joué au jeu et analysé ses composantes.
Ces réunions se sont déroulées de manière hebdomadaire pendant environ deux mois et nous ont permis d'établir de manière précise un ensemble de modules et TDA, en respectant les contraintes imposées et en étant fidèle au jeu initial.

Par la suite, une fois ce diaporama terminé, nous avons pu commencer le codage des classes. Cette partie s'est déroulée à partir de janvier jusqu'à la mi-avril en comprenant également les tests réalisés pour parfaire l'application.
Le codage s'est réalisé en 3 étapes : une première partie où seul le modèle a commencé à être implanté pour avoir accès aux objets de base et étudier les concepts graphiques en parallèle; une deuxième partie où la vue et le modèle étaient avancé en parallèle, sans être reliés; et une troisième partie où seule la vue restait, ce qui nous a permis d'effectuer par le biais de classes, des tests sur le modèle.

Pour s'assurer de la bonne direction de notre projet, nous avons tenu à prendre régulièrement rendez-vous avec notre professeur référent pour le projet. En effet, il était important de ne pas partir dans une direction non-souhaitée.
Nous avons grandement pris à cœur de nous comporter comme dans un milieu d'entreprise face à une demande de client.

Nous avons tenu à travailler de manière individuelle tout en échangeant à chaque étape sur l'avancement de nos parties respectives et en ne prenant des décisions importantes qu'en groupe.
C'est pourquoi nous pouvons dire que ce projet a été l'oeuvre d'un groupe et non de 3 personnes individuellement.

# II) Stratégies et approches adoptées

## A) Schéma général du code

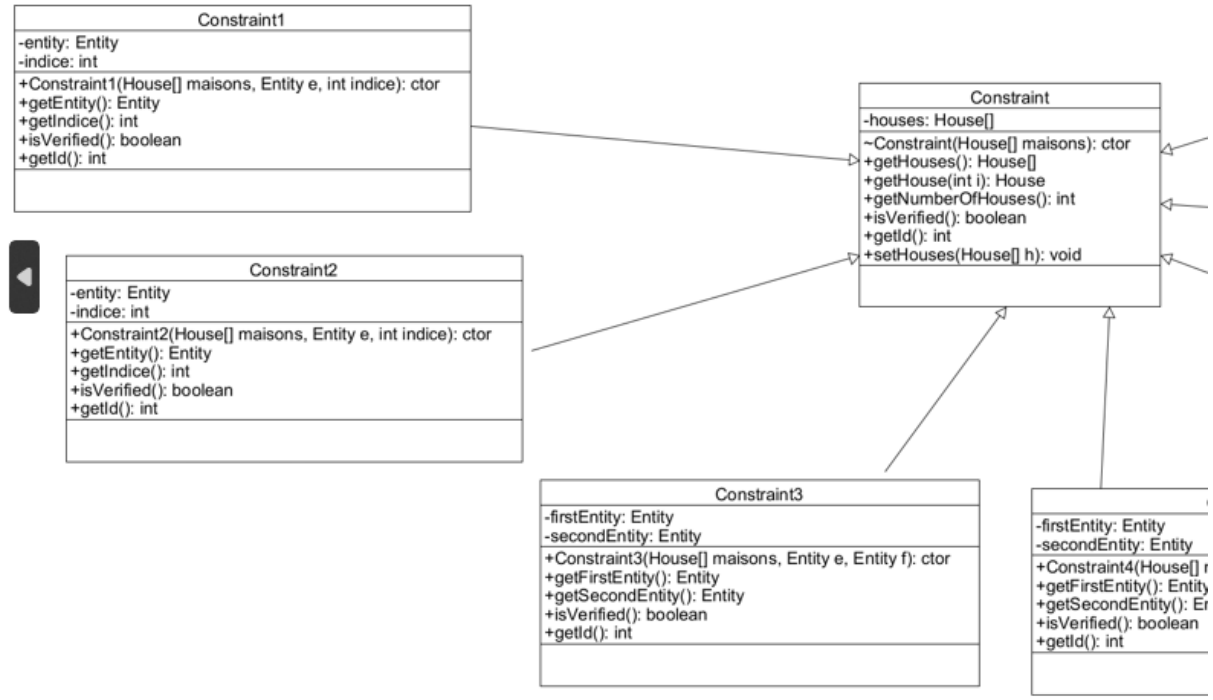
Nous allons tout d'abord vous présenter les relations d'implémentation et d'héritage au sein du modèle. Nous expliciterons également les relations d'utilisation importantes.

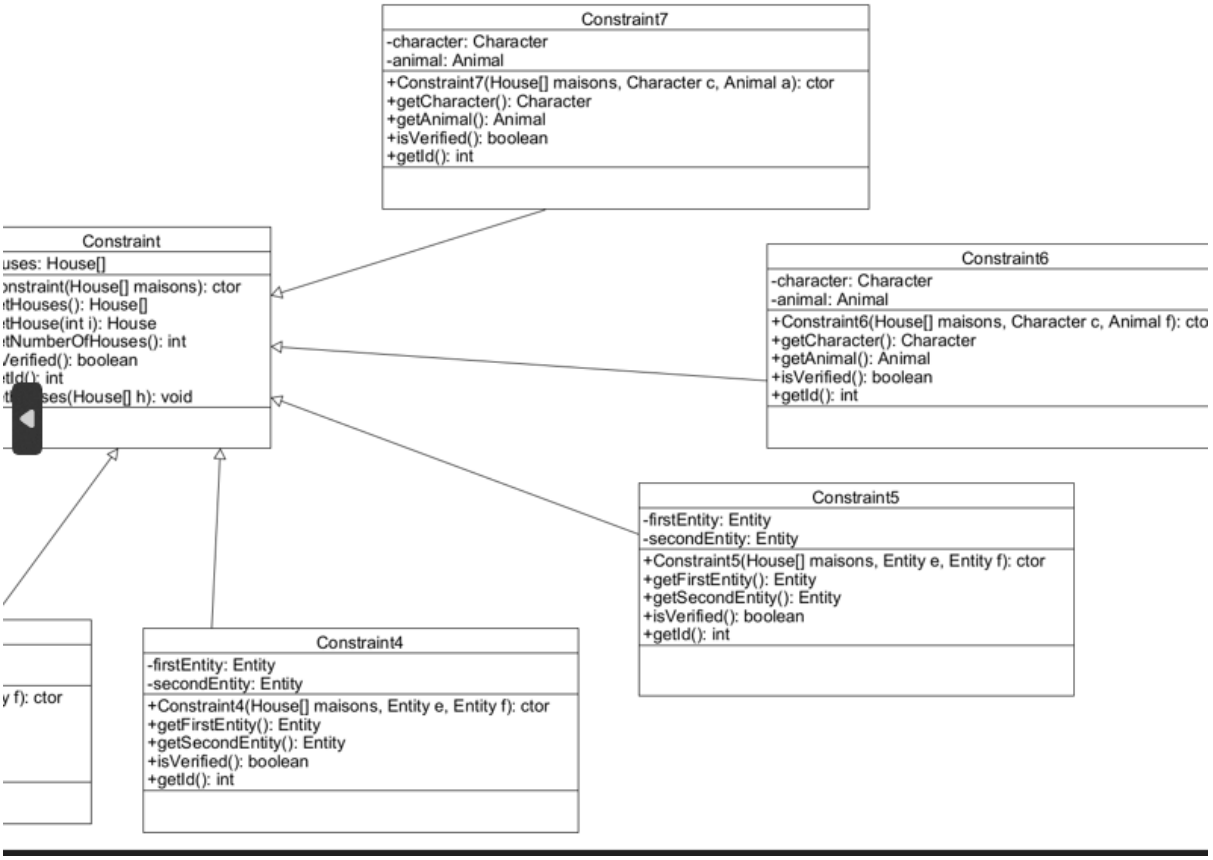
### Package "constraints"

Le package constraint est composé de 8 classes : Constraint qui est une classe abstraite et Constraint1 à Constraint7, qui représentent chacune des 7 contraintes du jeu (voir plus bas).

Chaque contrainte hérite de Constraint et l'étend, une contrainte étant symbolisée par un tableau de maisons, elle utilise naturellement l'interface House.

Une maison étant composée d'un personnage et d'un animal, House et Constraint utilisent Entity, Animal et Character.


Constraint1
-entity: Entity
-indice: int
+Constraint1(House[] maisons, Entity e, int indice): ctor
+getEntity(): Entity
+getIndice(): int
+isVerified(): boolean
+getId(): int
Constraint
-houses: House[]
~Constraint(House[] maisons): ctor
+getHouses(): House[]
+getHouse(int i): House
+getNumberOfHouses(): int
+isVerified(): boolean
+getId(): int
+setHouses(House[] h): void
Constraint2
-entity: Entity
-indice: int
+Constraint2(House[] maisons, Entity e, int indice): ctor
+getEntity(): Entity
+getIndice(): int
+isVerified(): boolean
+getId(): int
Constraint3
-firstEntity: Entity
-secondEntity: Entity
+Constraint3(House[] maisons, Entity e, Entity f): ctor
+getFirstEntity(): Entity
+getSecondEntity(): Entity
+isVerified(): boolean
+getId(): int
-firstEntity: Entity
-secondEntity: Entity
+isVerified(): boolean
+getId(): int


Constraint7
-character: Character
-animal: Animal
+Constraint7(House[] maisons, Character c, Animal a): ctor
+getCharacter(): Character
+getAnimal(): Animal
+isVerified(): boolean
+getId(): int
Constraint
Constraint6
-character: Character
-animal: Animal
+getCharacter(): Character
+getAnimal(): Animal
+isVerified(): boolean
+getId(): int
Constraint5
-firstEntity: Entity
-secondEntity: Entity
+Constraint5(House[] maisons, Entity e, Entity f): ctor
+getFirstEntity(): Entity
+getSecondEntity(): Entity
+isVerified(): boolean
+getId(): int
Constraint4
-firstEntity: Entity
-secondEntity: Entity
+Constraint4(House[] maisons, Entity e, Entity f): ctor
+getFirstEntity(): Entity
+getSecondEntity(): Entity
+isVerified(): boolean
+getId(): int

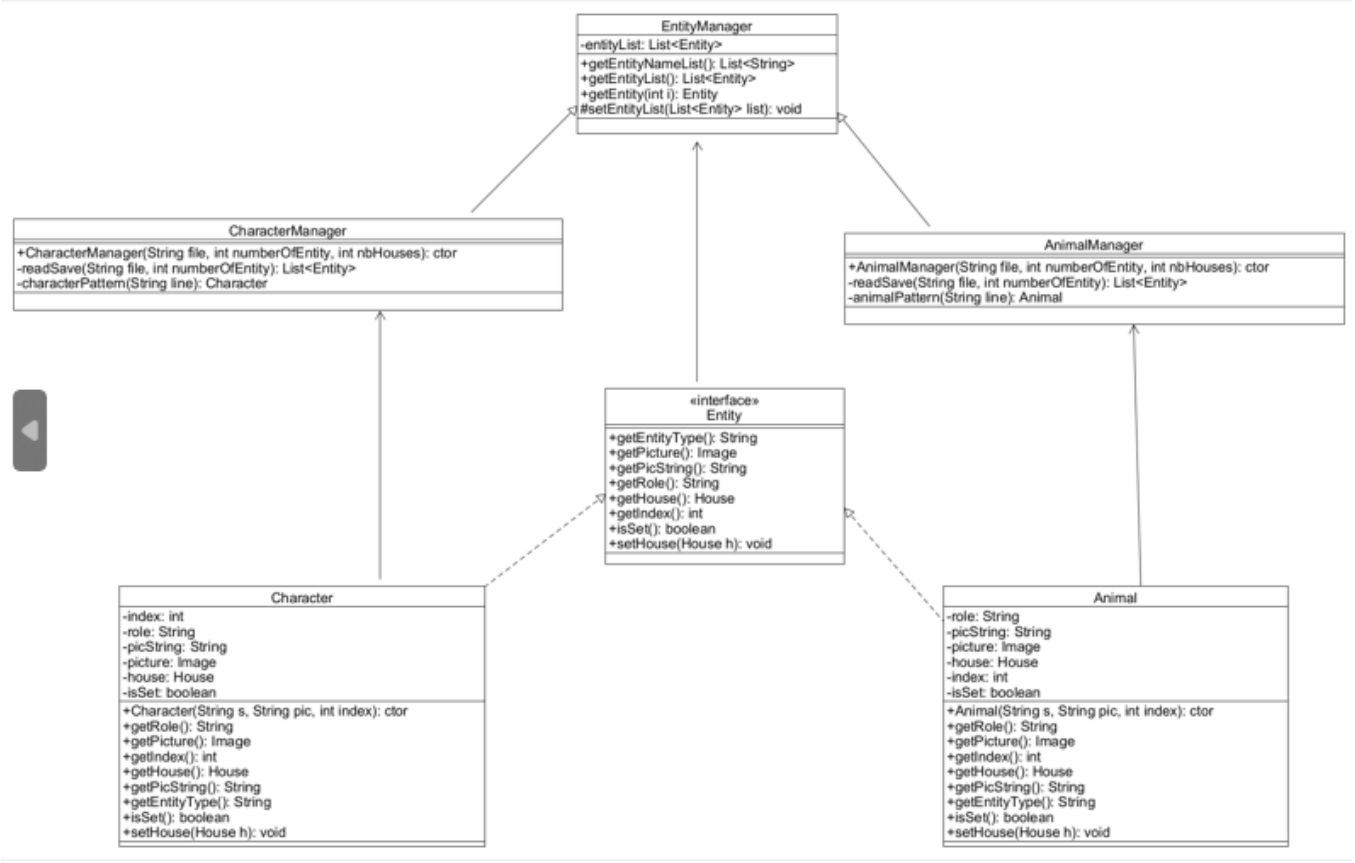
# Package “entites”

Le package entites est composé de 6 classes que l’on peut décomposer en 2 sous-groupes : les entités et les manager d’entités. Leurs fonctionnements sont détaillés plus bas.

Entity est une interface implémentée par Character et Animal.
De son côté EntityManager est une classe abstraite étendue par CharacterManager et AnimalManager.

Logiquement, chaque Manager utilise son type d’entité.

Chaque entité utilise House : si *m* est une maison et *e* une entité alors
si *m* est la maison de *e*, alors *e* habite dans *m*



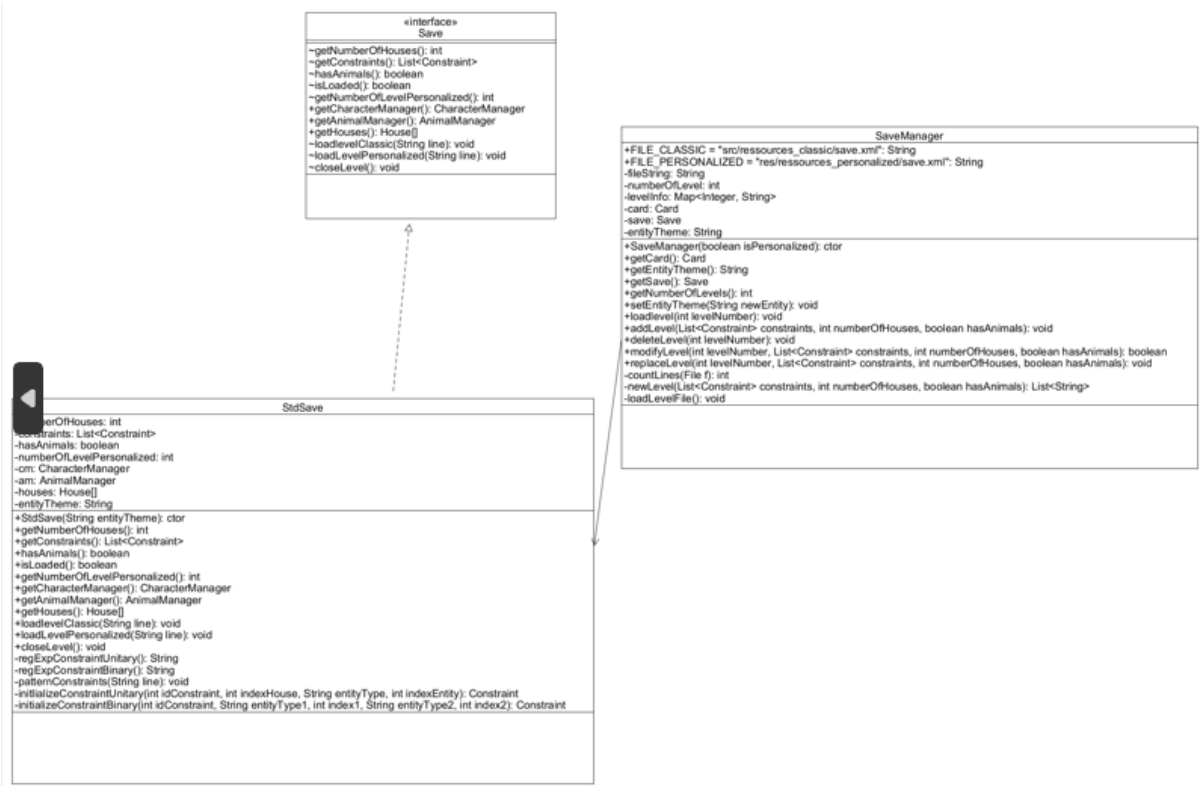
# Package “save”

Le package save est composé de 4 classes.

Save est une interface implémentée par StdSave, elle-même utilisée par SaveManager.
Nous retrouvons également ThemeSaver, une classe à part gérant les fichiers et images de thèmes (allant de la création du fichier de thème jusqu’à la copie d’un fichier d’un chemin source à un chemin destination).

Save utilise des contraintes, des AnimalManager et CharacterManager et des House.

SaveManager, elle, utilise une Card (expliquée plus bas).

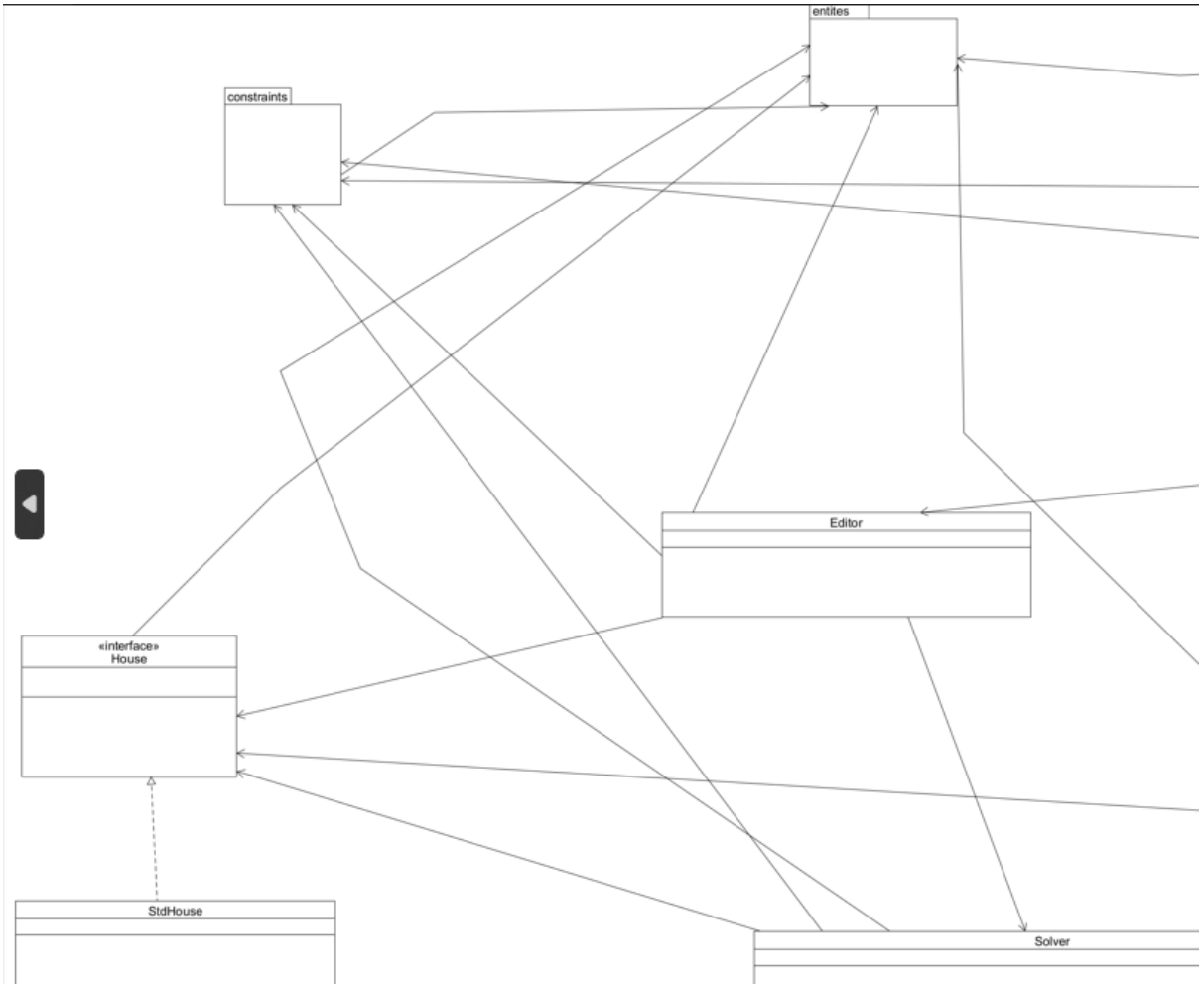


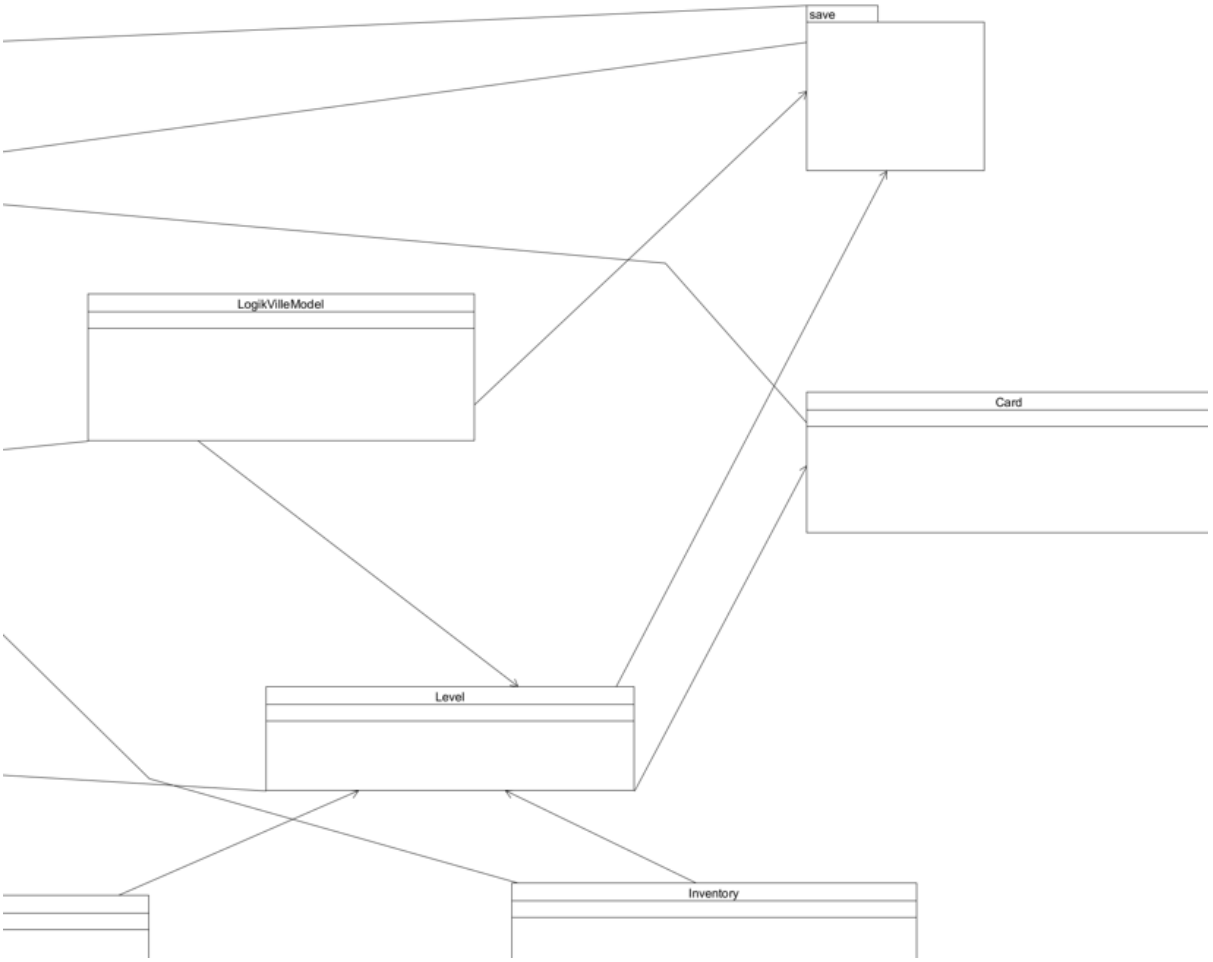

# Package “model”

Finalement nous arrivons au package model où nous pouvons trouver une classe : LogikVilleModel qui sera le modèle manipulé par la vue.
Nous avons aussi Editor, qui sera le modèle manipulé par la vue au niveau de l’éditeur de cartes.

Nous y trouvons l'interface House, implémentée par StdHouse. Pour le reste ce ne sont que des relations d'utilisation :

- Solver utilise House, Constraint (le package dans son ensemble) et le package entites.
- Inventory utilise Animal et Character
- Card utilise des Constraint
- Level utilise un SaveManager, House, Card, AnimalManager et CharacterManager, Entity
- Editor utilise Solver, Constraint, House, Entity, SaveManager, AnimalManager et CharacterManager
- LogikVilleModel utilise 2 SaveManager, Level, Editor






## Package “element”

Dans ce package, nous retrouvons 9 classes toutes en lien avec Swing. Ces classes sont en effet des sortes de JComponent que nous avons redéfinis, pour en améliorer le rendu graphique et rajouter des fonctionnalités.

De plus, nous avons également un Layout Manager, WrapLayout, servant à régler un bug du FlowLayout : celui-ci ne recalcule pas sa taille s’il se trouve dans une BoxLayout. Pour cela, il en hérite en redéfinissant les méthodes nécessaires pour régler le bug.

ClassicButton hérite de JButton. Il s’agit d’un bouton modifié afin d’apporter une interface graphique plus cohérente avec l’application.

EntityLabel est une sorte de JLabel représentant graphiquement une entité.

ConstraintLabel est une sorte de JComponent. Celui-ci permet de représenter graphiquement une contrainte : il est composé d'un EntityLabel à gauche, d'un EntityLabel à droite, et d'un JLabel au milieu contenant une image représentant graphiquement la contrainte.

GraphicCard est une sorte de JPanel permettant de représenter graphiquement une carte. Celui-ci rédéfinit sa méthode PaintComponent afin de réaliser un rendu de carte à l'écran.

GraphicHouse est une sorte de JPanel permettant de représenter graphiquement une maison. Celui-ci est configuré pour contenir deux composants, un en haut et un en bas. Il se dessine suivant une image contenue dans les fichiers du jeu.

GraphicSolution est une sorte de JComponent permettant de représenter graphiquement une solution. Celui-ci permet d'afficher plusieurs composants sous forme de grille (utilisation d'un GridLayout) mais avec un affichage amélioré.

LevelButton est une sorte de JButton permettant de créer un bouton se décomposant en deux parties : une image et un texte écrit en dessous. L'image est directement donnée à la création du bouton.

RoundPanel est une sorte de JPanel servant à rendre son fond visible avec la couleur souhaitée et avec des bords arrondis, afin d'ajouter un effet esthétique au JPanel.

## Package "gui"

Ce package permet de définir 10 classes dérivant toutes de JPanel : elles permettent de définir les différents menus graphiques de notre jeu.
De plus, des sous classes sont elle-même créées afin de factoriser le code, comme AbstractMenu et AbstractLevelMenu.
Toutes ces classes utilisent directement les composants définis dans le package element.

AbstractMenu est une sorte de JPanel. Cette classe abstraite permet de définir le squelette de base de tous les Menu. Elle fournit notamment de nombreuses méthodes outils nécessaires pour tous les héritiers.

TitleMenu dérive AbstractMenu afin de créer le menu d'accueil du jeu. Celui-ci utilise 5 ClassicButton afin de se réaliser. De plus, le fond d'écran est en réalité un JLabel tout comme le titre du jeu.

AbstractLevelMenu dérive AbstractMenu afin de créer une classe abstraite servant de squelette aux deux menus de sélection de niveaux : LevelMenu et PersonalizedLevelMenu. Celui-ci utilise deux rangées de LevelButton afin de pouvoir choisir les niveaux, ainsi que des ClassicButton pour naviguer dans les menus. De plus, un JTextField est disponible pour rechercher un niveau si nécessaire.

LevelMenu dérive AbstractLevelMenu afin de créer la classe du menu de sélection des niveaux classiques. Celui-ci utilise juste les fonctionnalités de AbstractLevelMenu afin de le configurer efficacement pour réaliser le menu souhaité.

PersonalizedLevelMenu dérive AbstractLevelMenu afin de créer la classe du menu de sélection des niveaux personnalisés. Il réalise la même chose que LevelMenu.

PlayMenu dérive AbstractMenu afin de créer le menu du plateau de jeu. Celui-ci utilise des GraphicHouse pour l'affichage des maisons sur le plateau de jeu. Il utilise également une GraphicCard remplie avec des ConstraintLabel pour réaliser la carte de jeu. Les entités sont représentées par des EntityLabel, stockées dans des JPanel en haut du menu. Enfin, des ClassicButton sont disposés pour rendre le menu interactif.

VictoryMenu dérive AbstractMenu afin de créer le menu de victoire. Celui-ci s'affiche quand le joueur a réussi un niveau. Il utilise des JLabel pour afficher le logo du jeu et un texte. De plus, un ClassicButton permet de passer à la suite.

EditorMenu dérive AbstractMenu afin de créer le menu d'édition de niveau. Celui-ci se décompose en 3parties : une GraphicCard à gauche que l'on complète au fur et à mesure avec des JLabel et des ConstraintLabel, un premier JPanel à droite servant à réaliser les premières configurations du niveau (nom du niveau, image du niveau, nombre d'entités et présence d'animaux ou non) puis enfin un second JPanel qui permet de créer des contraintes (via des JComboBox et des ClassicButton) et d'afficher les solutions de la carte actuelle en utilisant des GraphicSolution. Il est possible de passer du premier panel au second via un ClassicButton suivant. Enfin, de nombreux ClassicButton sont disposés pour rendre le menu interactif.

OptionMenu dérive AbstractMenu afin de créer le menu de choix des options. Celui-ci se compose de deux JComboBox pour choisir la résolution de l'écran et le thème du jeu, ainsi que d'un JCheckBox pour choisir le plein écran ou non. De plus, 3ClassicButton sont disposés pour naviguer entre les menus et appliquer le choix des options.

CreatorThemeMenu dérive AbstractMenu afin de créer le menu de création de thème. Celui-ci se compose de nombreux éléments : pour chaque entité représentée par un JLabel (via son image), il est possible de lui affecter un nom via un JTextField, ainsi que de choisir une nouvelle image pour lui via un ClassicButton ouvrant un JFileChooser. De plus, un JComboBox est disponible pour charger un thème, ainsi qu'un JTextField pour choisir le nom du thème. De nombreux ClassicButton permettent de rendre interactif le menu et de naviguer entre les menus.

# B) Présentation des parties essentielles du code et des choix de programmation

# Du côté du modèle :

## Entités

LogikVille est avant tout un jeu de plateau caractérisé par ses personnages et animaux.
Pour rappel, dans le jeu de base, on peut jouer avec 3 à 5 personnages et de 3 à 5 animaux, ou aucun, cependant le nombre de personnages est toujours égal au nombre d'animaux.

Cette sous-partie a pour but de vous présenter d'un côté les entités et la manière dont on les a implantées, et de l'autre la manière dont on a géré les entités disponibles pour une carte : les Manager.

### 1) Gestion des entités

Afin de factoriser le code et de le rendre plus générique, nous avons créé une interface commune aux personnages et animaux : Entity.
Cette dernière possède une multitude de méthodes permettant notamment de récupérer : le nom de l'entité, son type (animal ou personnage) et son image.

Enfin une entité peut habiter dans une maison, c'est pourquoi nous avons défini des méthodes permettant de récupérer leur maison ou alors de leur en donner une.

Finalement, nous avons créé 2 classes : Animal et Character, implémentant cette interface.

### 2) Manager

Pour faciliter la gestion d'accès des entités disponibles depuis le solveur, la vue et l'éditeur nous avons créé une classe abstraite : EntityManager.

Un manager est représenté par une liste d'entités disponibles qui sera remplie via les sous-classes CharacterManager et AnimalManager.
Elle ne possède qu'une seule commande qui permet de modifier sa liste d'entités.

Afin de faciliter le travail de la vue au niveau de l'éditeur, nous avons choisi d'ajouter une méthode getEntityNameList() qui renvoie le nom de toutes les entités disponibles pour un niveau.

Nous en avions besoin car, au moment de la création d'un niveau personnalisé dans l'éditeur, en fonction du nombre de maisons choisies, s'il y a des animaux ou non et aussi en fonction des contraintes utilisées, alors les entités disponibles pour une contrainte seraient différentes.
En effet, au niveau des JComboBox pour choisir les entités, nous avions besoin d'avoir accès à leurs noms

Nous allons maintenant vous présenter la structure générale d'AnimalManager qui est très similaire à celle du CharacterManager.

Le constructeur prend 3 paramètres : un fichier, un nombre d'entités et un nombre de maisons.

Le fichier regroupe toutes les entités créées et les décrit à l'aide d'une syntaxe bien précise. On y trouve des informations sur le type d'une entité, son index dans le fichier, son nom et son chemin d'accès vers son image.

Les 2 entiers, eux, symbolisent le nombre d'entités à créer pour le niveau et, en particulier, s'il faut ou non instancier des animaux.

Le constructeur appelle une unique méthode : setEntityList (celle d'EntityManager)
Cette méthode prend elle-même en paramètre le résultat d'une autre méthode : readSave.
readSave prend 2 paramètres : un fichier et un entier.

Nous allons maintenant vous décrire le fonctionnement de cette méthode.

readSave : comme son nom l'indique, elle lit le fichier donné en paramètre et renvoie une liste constituée des n premiers animaux présents dans le fichier. (n étant l'entier donné en paramètre).

Cette méthode lit une à une les lignes du fichier et :

- appelle une méthode : animalPattern et analyse la ligne lue : via une expression régulière, elle récupère les infos lues sur la ligne et instancie le bon animal correspondant en le renvoyant.

- une fois l'animal récupéré on l'ajoute dans une liste nouvellement créée.

On répète ceci n fois. A la fin de la boucle, on aura les n premiers animaux instanciés et ajoutés dans une liste.

readSave renvoie donc cette liste à setEntityList d'EntityManager.

L'AnimalManager représente donc les n premiers animaux du niveau et CharacterManager les n premiers personnages du niveau.

## Contraintes

LogikVille est un jeu représenté par ses niveaux. Chaque niveau est lui-même caractérisé par ses contraintes qui permettent d'obtenir une unique solution. C'est précisément de ces contraintes que nous allons maintenant vous parler.

Pour rappel, il existe 7 types de contraintes :

1- Une entité habite dans une maison précise
2- Une entité n'habite pas dans une maison précise
3-Deux entités sont voisines
4-Deux entités ne sont pas voisines
5-Une entité se situe à gauche d'une autre entité
6- Un personnage habite dans la même maison qu'un animal
7- Un personnage n'habite pas dans la même maison qu'un animal

Chaque contrainte se ressemble et elles ne diffèrent que par le nombre d'entités et leur critère de validation. D'un point de vue code, ça implique automatiquement une classe abstraite : Constraint.

## Constraint

Une contrainte est avant tout caractérisée par un tableau de maisons. En effet, on ne peut tester la validation d'une contrainte si et seulement si la contrainte dispose d'un tableau de maisons sur lequel vérifier ses conditions. Une contrainte est donc représentée à tout moment depuis son instanciation par un tableau de maisons.

La classe mère, abstraite, des contraintes gère donc tout ceci. Elle permet de récupérer le tableau de maisons d'une contrainte ou de remplacer le sien par un autre. Nous nous sommes rendus compte de l'importance de cette dernière méthode au moment de la création du solveur. En effet (voir plus bas), le solveur calcule une multitude de cas possibles et donc agit sur différents tableaux de maisons. Il était donc important, pour une contrainte donnée, de pouvoir la tester sur différents cas potentiels.

De plus, elle définit deux méthodes abstraites qui seront propres à chacune des contraintes mais dont leur résultat variera en fonction de la contrainte.
La méthode isVerified(), qui vérifie si la contrainte est vérifiée, est en temps constant pour n'importe quel type de contrainte.

Si on se réfère à la liste des différentes contraintes, on peut aisément les séparer en deux types : les contraintes unitaires et les contraintes binaires.

## Contraintes unitaires

Les contraintes unitaires sont au nombre de 2 et ont la caractéristique suivante : elles ne possèdent qu'une entité et un entier, symbolisant l'indice de la maison sur laquelle la contrainte agit.

Elles possèdent 2 méthodes communes : une requête permettant de récupérer l'entité concernée par la contrainte et une seconde permettant de récupérer l'indice de la maison concernée.

Pour des raisons pratiques et pour les différencier plus aisément, nous avons créé une méthode getId() qui renvoie le numéro de la contrainte, tel que présenté dans la liste plus haut.

La première contrainte unitaire a donc comme Id 1 et la seconde a comme Id 2.
Dorénavant, nous appellerons "contrainte de type n", la n-ième contrainte telle que décrite dans la liste plus haut.

Pour les contraintes de type 1 et 2, il ne reste qu'une seule méthode, celle qui permet de déterminer si la contrainte est respectée pour son tableau de maisons.

La contrainte de type 1 renvoie vrai si et seulement si l'indice de la maison dans laquelle son entité se trouve est le même que l'indice fourni en paramètre de son constructeur.

De son côté, la contrainte de type 2 renvoie vrai si et seulement l'indice, son attribut, est différent de l'indice de la maison de son entité.

## Contraintes binaires

De leur côté, les contraintes binaires concernent seulement deux entités.
Chaque contrainte binaire possède 2 requêtes permettant de récupérer leur première entité et leur seconde.

En ce qui concerne les contraintes de type 6 et 7, nous sommes assurés que la première entité est un personnage et la seconde un animal (ce qui implique que l'on ne peut trouver des contraintes de type 6 et 7, seulement si le niveau comporte des animaux).

De même que les contraintes unitaires, elles possèdent une méthode getId() qui renvoie le même résultat qu'expliqué plus haut.

Il ne reste donc que la méthode isVerified() implantée comme suit :

Pour les contraintes de type 3, renvoie vrai si et seulement si l'indice de la maison de la première entité vaut celui de la seconde plus un (ou alors moins un).

Pour les contraintes de type 4, renvoie vrai si et seulement si l'indice de la maison de la première entité diffère de plus d'un que celui de la seconde.

Pour les contraintes de type 5, renvoie vrai si et seulement si l'indice de la maison de la première entité est strictement inférieur à celui de la seconde.

Pour les contraintes de type 6, renvoie vrai si et seulement si le personnage et l'animal ont le même indice de maison.

Enfin, pour les contraintes de type 7, renvoie vrai si et seulement si le personnage et l'animal ont un indice de maison différent.

# Solveur

Dans cette section, nous allons vous présenter la classe Solver et plus particulièrement, sa commande findSolutionsFor() qui permet de trouver toutes les solutions possibles pour une liste de contraintes et un nombre d'animaux et personnages disponibles.

Nous commencerons par vous expliciter nos algorithmes et fonctions annexes, puis nous finirons par décrire et donner un exemple d'utilisation de ce solveur.

## I- Algorithmes

```
Fonction cloneList (L : Liste de maisons) : Liste de maisons;
Var : h : Maison;  copy : Liste de maisons;
Début
        copy <- list();
        Pour chaque maison h de L faire :
                ajouter(copy, clone(h));
        Retourner copy;
FinFonction
```

```
Fonction allEntitiesCombinations (H : liste de maisons; L : Liste d'éléments) : Liste de Listes d'éléments;
Var : entity : élément; permut : Liste de Listes d'éléments; c, temp : Liste d'éléments; index : entier; result : Liste de Listes d'éléments;
Début
        Si L est vide alors :
                result <- list(list());
                ajouter(result, L);
                Retourner result;
        entity <- Retirer(L, 0);
        result <- list(list());
        permut <- allEntitiesCombinations(H, L);
        Pour chaque liste c de permut faire :
                Pour index de 0 à taille(c) faire
                        temp <- list(c);
                        ajouter(temp[index], entity);
                        ajouter(result, temp);
        Retourner result;
FinFonction
```

```
Fonction allCombinations (H : Liste de Maisons; cList : Liste de Listes de personnages; aList
: Liste de Listes d'animaux) :  Liste de Listes de Maisons;
Var : list : Liste de personnages ; list2 : Liste d'animaux ; tmp : Liste de Maisons; k, j, i :
entier; h : Maison; result : Liste de Listes de Maisons;
Début
        result <- list(list());
        Si aList est vide alors :
                Pour chaque liste list de cList faire :
                        tmp <- cloneList(H);
                        Pour j de 0 à taille de tmp faire :
                                h <- tmp[j];
                                Si pas de personnage dans h alors :
                                        Personnage de h < - list[k];
                                        k <- k + 1;
                        ajouter(result, tmp);
        Sinon :
                Pour chaque liste list de cList faire :
                        Pour chaque liste list2 de aList faire :
                                tmp <- cloneList(H);
                                k <- 0;
                                i <- 0;
                                Pour j de 0 à taille de tmp faire :
                                        h <- tmp[j];
                                        Si pas de personnage dans h alors :
                                                Personnage de h <- list[i];
                                                i <- i + 1;
                                        Si pas d'animal dans h alors :
                                                Animal de h <- list[k];
                                                k <- k + 1;
                                ajouter(result, tmp);
        Retourner result;
FinFonction

Procédure allSolutions(L : Liste de contraintes; L1 : Liste de contraintes de type 1; cm :
Gestionnaire de personnages ; am : Gestionnaire d'animaux) : Var :solutions : Liste de listes
de Maisons;
Var : characters : Liste de personnages; animals : Liste d'animaux; solutionPossible : Liste
de Maisons; i, index : Entier; c: Personnage; a : Animal; c1 : Contrainte de type 1; e: Entité;
h : Maison; temp : Liste de listes de personnages; temp 2 : Liste de listes d'animaux;
result : Liste de listes de maisons;
Début
        characters < -list();
        Pour chaque Personnage c de cm faire :
                ajouter(characters, c);

        animals< -list();
        Pour chaque Animal a de am faire :
                ajouter(animals, a);
```

```
        solutionPossible <- list();
        Pour i de 0 à taille de cm faire :
                ajouter(solutionPossible, Maison(i, taille de characters));
        Pour chaque Contrainte de type 1 c1 de L1 faire :
                e <- entité de c1;
                index <- indice de c1;
                h <- solutionPossible[index];
                Si e est un animal alors :
                        animal de h <- e;
                        retirer(animals, e);
                Sinon :
                        personnage de h <- e;
                        retirer(characters, e);
        temp <- allEntitiesCombinations(solutionPossible, characters);
        temp2 <- allEntitiesCombinations(solutionPossible, animals);
        result <- allCombinations(solutionPossible, temp, temp2);
        solutions <- result;
FinProc

Fonction getConstraint1 (L : Liste de contraintes) :  Liste de contraintes de type 1;
Var : c : Contrainte;  result : Liste de contraintes de type 1;
Début
        result <- list();
        Pour chaque contrainte c de L faire :
                Si c est de type 1 alors :
                        ajouter(L, c);
        Retourner result;
FinFonction

Algorithme findSolutionsFor :
{Entrée : L : Liste de contraintes; cm : gestionnaire de Personnages; am : gestionnaire
d'Animaux}
{Sortie : solutionFound : booléen; solutions : Liste de listes de Maisons}
Var : L1 : liste de Contraintes de type 1; c1: Contrainte de type 1; H : Tableau de Maisons;
tmp : Liste de listes de Maisons; i, j : Entier; s : Liste de Maisons; c : Contrainte; e : Entité;
Début
        solutionFound <- Faux;
        solutions <- list(list());

        L1 <- getConstraint1(L);
        Pour chaque Contrainte de type 1 c1 de L faire :
                retirer(L, c1);
        allSolutions(L, L1, cm, am);
        H <- tableau(taille de liste d'entité de cm);
        tmp <- list(list(solutions));
        Pour i de 0 à taille de tmp faire :
                s <- tmp[i];
```

```
            Pour j de 0 à taille de tmp faire :
                H[j] <- s[j];
                Maison de l'animal dans s[j] <- H[j];
                Maison du personnage dans s[j] <- H[j];
            Pour chaque Contrainte c de L faire :
                Maisons de c < - H;
                Si c n'est pas vérifiée alors :
                    Retirer(solutions, s);
        solutionFound <- (taille de solutions > 0);
        Pour chaque entité e de cm faire  :
            retirer e de sa maison;

        Pour chaque entité e de am faire  :
            retirer e de sa maison;
FinAlgo
```

# II- Trame du solveur

## 1- Préambule

Le solveur est représenté, à tout moment, par sa liste de solutions trouvées pour la dernière carte donnée. Une solution, ici, étant une liste de maisons et une maison, un objet que nous pouvons rapprocher d'un couple (personnage, animal).

L'unique commande du solveur prend trois arguments :

- une liste de contraintes
- un gestionnaire de personnages
- un gestionnaire d'animaux

Ces trois informations nous permettent de décrire un niveau de jeu et sont suffisantes au solveur pour calculer ses solutions.

Une fois sa commande appelée, le solveur remet à zéro ses attributs, en effet, au début du calcul, plus aucune solution n'est trouvée.

## 2- Séparation des contraintes

La première étape importante du solveur est de séparer ses contraintes de type 1 des autres. Les contraintes de type 1 étant des contraintes imposant à une entité d'habiter dans une maison bien précise, par exemple : Pompier dans la maison la plus à gauche.
Ce procédé nous permettra de réduire drastiquement le nombre de tours de boucle à la fin.

*Prenons, par exemple, une carte composée de 5 personnages et 0 animal. Le nombre de solutions potentielles est de 5! = 120. Or grâce à ces contraintes de type 1, on élimine 4 positions possibles pour l'entité en question. Il reste donc que 4! = 24, solutions potentielles. Nous divisons donc par 5 le nombre de tours de boucle.*

Cette méthode est appelée getConstraint1 et prend en paramètre la liste de toutes les contraintes du niveau.
Puis, pour chaque contrainte de cette liste, si elle est de type 1 alors on l'ajoute dans une nouvelle liste nouvellement créée.
Une fois la boucle finie, on a traité toutes les contraintes et on renvoie la nouvelle liste.
Au retour de cette fonction, on n'oublie évidemment pas de supprimer les contraintes de type 1 de la liste de base.
Nous avons donc 2 listes : la première qui contient toutes les contraintes, sauf celles de type 1 et la deuxième qui contient toutes les contraintes de type 1.

## 3- Calcul de toutes les solutions

Une fois que toutes les contraintes sont bien séparées, nous basculons sur une méthode outil qui va s'occuper de calculer toutes les solutions potentielles en fonction de

- la liste de contraintes de base
- la liste de contraintes de type 1
- le gestionnaire de personnages
- le gestionnaire d'animaux

### *a) Création des listes de personnages, d'animaux et de maisons*

Nous commençons par créer une liste de personnages et, pour chaque personnage dans le gestionnaire, nous l'ajoutons dans cette liste. Et nous faisons de même pour les animaux.
Ainsi, nous disposons d'une liste de personnages et une liste d'animaux (éventuellement vide) disponibles. Ces 2 listes nous seront utiles pour calculer toutes les permutations.

Enfin, nous créons de la même manière une liste de maisons.
Et nous tâchons de la remplir en fonction des informations que l'on possède déjà, c'est-à-dire celles données par les éventuelles contraintes de type 1.

Pour chaque contrainte de la liste de contraintes de type 1, nous récupérons l'entité et l'indice de la maison concernée. Puis, nous ajoutons dans notre liste de maisons, à cet indice, l'entité récupérée. Et nous supprimons de la liste de personnages ou d'animaux disponibles cette entité. Encore une fois, ceci est fait en raison d'optimisations pour le nombre de tours de boucle à la fin.

*Par exemple, reprenons notre exemple de contrainte : pompier dans la maison la plus à gauche. Dans ce cas, l'indice récupéré est 0 et l'entité est un personnage, le pompier.*
*Dans la liste de maisons, nous ajoutons donc en indice 0, le pompier. Et nous retirons le pompier de la liste des personnages disponibles. Il ne nous en reste que 4.*

### *b) Calcul de toutes les permutations des entités*

Maintenant que nos maisons sont remplies correctement. Nous pouvons calculer toutes les permutations des personnages, puis toutes les permutations des animaux.

Pour faire ceci, nous avons une méthode outil récursive et générique qui prend en paramètre une liste de maisons et une liste d'entités. Elle renvoie une liste de listes d'entités, représentant toutes les permutations possibles de ces entités.

allEntitiesCombinations fonctionne ainsi :
Si la liste en paramètre est vide alors on crée une liste de listes résultat vide à laquelle on ajoute notre liste vide et on renvoie la liste de listes.
Sinon, on récupère et supprime la première entité de cette liste et on appelle récursivement sur notre liste à laquelle on a retiré un élément.
Quand notre liste de listes a été créée, on parcourt chacune de ses listes et pour chacune d'entre elles on ajoute l'entité précédemment supprimée à la bonne place.
Enfin, on renvoie la liste de listes.

Nous appelons cette fonction deux fois, une fois pour les personnages et une fois pour les animaux.

### c) *Calcul de toutes les combinaisons*

Nous disposons maintenant de deux listes de listes et nous souhaitons calculer toutes les combinaisons entre ces deux listes, et remplir au fur et à mesure des listes de maisons, qui symboliseront nos solutions potentielles.

Pour cela, nous disposons d'une méthode allCombinations prenant en paramètre :
-une liste de maisons
-une liste de listes de personnages
-une liste de listes d'animaux (éventuellement vide)
Elle renvoie une liste de listes de maisons, c'est donc la liste de toutes les solutions potentielles.

*Avant de décrire cette fonction, il est important de souligner qu'elle agit sur le principe suivant : chaque solution (liste de maisons) à sa création se base sur la liste de maisons fournie en paramètre. En effet, nous sommes assurés que les entités déjà présentes dans cette liste y sont à leur bonne place.*
*Cependant, nous ne pouvions agir sur la même référence de liste pour ne pas perdre ces informations et un clonage en surface n'était pas suffisant dans notre cas. Nous avons dû modifier notre classe House, afin qu'elle implémente cloneable et nous avons redéfini sa méthode clone(), afin d'avoir un clonage en profondeur.*

Passons à l'explication de cette fonction; 2 cas sont possibles :

- le niveau se joue sans animaux. Auquel cas, on parcourt chaque liste de notre liste de listes de personnages et, pour chaque liste on la clone. Puis pour chaque maison de cette liste, on y ajoute un personnage disponible pas encore ajouté, si de la place est disponible dans cette maison

*Par exemple, toujours dans notre exemple du pompier dans la maison la plus à gauche, au premier tour de boucle, personne ne serait ajouté, car la maison la plus à gauche comporte déjà un personnage*

- le niveau comporte des animaux. Auquel cas, on parcourt chaque liste de notre liste de listes de personnages. Et pour chaque liste, on parcourt chaque liste de notre liste de listes d'animaux. Enfin pour chaque liste, même fonctionnement qu'au-dessus, on clone puis on définit les animaux et personnages disponibles, si la place est libre.

### 4- Vérification des contraintes

Nous disposons maintenant d'une liste de listes de maisons et nous allons y traiter une à une chacune des solutions et vérifier si chaque contrainte de départ est respectée.
(Nous nous rendons bien compte, à quel point les optimisations sur les contraintes de type 1 sont nécessaires).

Pour chaque solution, nous remplissons un tableau de maisons avec les mêmes informations que dans la liste. Enfin nous parcourons chacune des contraintes et nous vérifions que chacune d'elle soit vérifiée pour ce tableau de maisons. Si ce n'est pas le cas, nous supprimons la solution étudiée des solutions possibles et nous n'oublions pas de break la boucle.

A la fin de toutes ces boucles, l'attribut solutions contient toutes les solutions du niveau.

*Notez bien qu'en pratique, les 84 niveaux de base ne comportent qu'une unique solution et ce sera également le cas pour chacun des niveaux personnalisés créés ultérieurement.*

## Save

Nous allons maintenant vous présenter le package save et son fonctionnement.

Au chargement d'un niveau, nous donnons un entier, représentant le numéro du niveau ainsi qu'un SaveManager et c'est spécialement de lui dont nous allons vous parler.

Un SaveManager est une classe qui va s'occuper d'aller chercher les informations d'un niveau dans un fichier xml préalablement rempli. Il permet aussi de sauvegarder un niveau personnalisé via l'éditeur (voir plus bas).

Il a donc accès à 2 fichiers distincts : le premier immuable est le fichier xml contenant les niveaux classiques, le second amené à changer régulièrement sera le fichier xml contenant les niveaux personnalisés.

A l'instanciation du SaveManager nous appelons une méthode outil, loadLevelFile().
Cette méthode, à l'aide d'un parser, va analyser le fichier xml et, en fonction des balises qu'il va trouver, va ranger les informations du niveau dans une HashMap contenant les informations du niveaux, et dans une autre HashMap pour les niveaux personnalisés, contenant les informations permettant d'identifier un niveau depuis le menu de choix.

Chaque information est transformée en une String bien précise, suivant une expression régulière bien définie. Petit à petit ces String sont concaténées dans une String résultat. A la fin de la lecture d'un niveau, on ajoute le résultat dans notre HashMap contenant les informations.

Quand on demande au SaveManager de charger un niveau précis, il ira donc chercher les informations dans l'un ou l'autre fichier, en fonction de si c'est un SaveManager de niveau classique ou non.

Dans tous les cas, un StdSave est créé et nous lui demandons de charger un niveau, en transformant la chaîne passée en paramètre en des informations bien précises (nombre de maisons du niveau, s'il y a des animaux, quelles contraintes à appliquer).

StdSave va donc analyser la chaîne à l'aide d'une expression régulière et, via des groupes de capture, instancier les CharacterManager, AnimalManager et contraintes correspondants.

Quand StdSave a fini son travail, SaveManager instancie une Card avec les informations récoltées par StdSave.

SaveManager permet également d'ajouter un niveau.
En effet, l'intérêt premier de SaveManager est de gérer les fichiers xml dans leur globalité, c'est donc lui qui accède aux fichiers et les modifie quand cela est nécessaire.
Ainsi, la méthode addLevel permet, en donnant les bonnes informations de niveau, d'ajouter un nouveau niveau au fichier des niveaux personnalisés. Il n'est pas possible de modifier les niveaux classiques, comme dit plus haut, ce fichier est immuable.
De plus, le fichier des niveaux personnalisés comprend certaines particularités qui se remarquent dans la fonction d'ajout: il est possible de donner un nom et une image au nouveau niveau.

SaveManager autorise aussi la suppression et la modification, ou encore le remplacement de niveaux, mais ces 2 dernières fonctionnalités ne sont pas encore supportées par la vue.
La commande deleteLevel lève une exception en cas de problème sur le fichier.

StdSave, comme dit plus haut implémentant Save, permet la gestion d'un niveau donné par SaveManager. Cette classe possède ainsi de nombreuses requêtes permettant de consulter les données du niveau, une fois celui-ci chargé au moyen de l'une des deux commandes loadLevelClassic ou loadLevelPersonalized appelée par SaveManager. Etant donné que StdSave ne gère qu'un niveau, il est possible par mesure de précaution de "fermer le niveau" au moyen de la commande closeLevel qui remet tous les attributs à 0.

Enfin, concernant la classe ThemeSaver, celle-ci n'est pas directement liée au modèle de base, mais agit plutôt comme un modèle pour la gestion des fichiers de thèmes prévue par

la vue. Comme dit précédemment, elle permet de créer un fichier de thème ou de le supprimer. Mais elle est aussi utile pour charger un fichier de thème et récupérer les informations importantes pour la vue (comme une liste de noms pour personnages et animaux ou encore une liste des chemins vers leur image représentative). Elle permet tout aussi bien de supprimer un dossier de thème. Enfin, elle permet de créer des répertoires inexistants à partir d'un chemin donné (l'action de createDirectories) et de copier d'un chemin source vers un chemin destination un fichier (copyFile).

## Level

Level est une classe permettant de gérer l'ensemble d'un niveau : du déplacement d'entité dans une maison à la vérification de la solution, en passant par le recommencement.
A son instanciation, Level prend un SaveManager et un numéro de niveau, lui permettant de charger à partir de ce SaveManager un niveau spécifique, Level ne sait donc pas s'il s'agit d'un niveau classique ou personnalisé, il se contente de gérer les informations du plateau de jeu.
Avec le chargement d'un niveau, on charge aussi sa solution : un appel au solveur sur les contraintes du niveau et les AnimalManager/CharacterManager chargés permet d'enregistrer la solution.

Ainsi, on pourra vérifier si le joueur a gagné en comparant le contenu des maisons de la solution à celui en jeu, c'est le principe de la requête isLevelWon.
Level met également à disposition de nombreuses requêtes permettant aux classes d'autres paquets n'ayant pas accès à la Save ou au SaveManager de consulter les informations du niveau.

Comme expliqué précédemment, Level gère surtout ce qu'il se passe en jeu, on a donc à notre dispositions les commandes setEntity et removeEntity qui prennent une entité et une maison en paramètres et qui attribuent ou retirent l'entité de la maison.
Dans cette même optique, la commande exchangeEntities permet à 2 entités et 2 maisons de s'interchanger pour faciliter le jeu de l'utilisateur.

Enfin, on retrouve la commande reset qui permet de relancer le plateau de jeu comme au chargement du niveau d'origine. Elle remet à neuf le contenu des maisons et l'attribution des maisons aux entités.

## Inventory

Inventory est une classe agissant quelque peu comme un historique : instanciée avec 2 listes correspondant aux listes d'origines des entités, elle permet la gestion des personnages et animaux restants sur le plateau de jeu.Par ce biais, il est possible de savoir exactement quelles sont les entités encore disponibles et celles qui ont été utilisées.
Pour cela, 4 requêtes sont disponibles : getCharactersLeft(), getAnimalsLeft(), getCharactersUsed(), getAnimalsUsed().
Ainsi, 2 commandes suffisent à la gestion de ces listes : addEntity et removeEntity.
En fonction de l'entité (s'il s'agit d'un Character ou d'un Animal) l'entité sera ajouté ou retirée de l'un ou l'autre des listes.

Inventory est ainsi utile dans Level lorsque l'on set et remove des entités de maisons.

## Card

Card est une classe assez simple regroupant les informations semblables à celles que l'on peut trouver sur une carte du jeu de plateau. On peut donc retrouver l'ensemble des contraintes, le nombre de personnages, et si le niveau comporte des animaux.

Une carte s'instancie en donnant ces trois informations et il est alors possible de les consulter à tout moment.

## House

Dans le jeu de plateau, nous devons ranger les entités dans des maisons. Ainsi, il était nécessaire de modéliser ce concept, l'interface House est née de cela.

Une House contient les informations des entités qui s'y trouvent : un personnage et un animal au plus. Il est également possible de savoir si une maison est vide, ou encore de consulter l'indice représentant la maison sur le plateau de jeu. Enfin, nous disposons de deux commandes : setCharacter et setAnimal, prenant respectivement un Character et un Animal permettent d'ajouter l'entité passée en paramètre à la maison. Effectuer un setCharacter(null) ou un setAnimal(null) revient à enlever l'entité de la maison.

Ces méthodes sont concrétisées par la classe StdHouse, qui ajoute en plus la commande clone, permettant de cloner une maison en profondeur.
Les maisons sont des objets essentiels de ce projet et sont donc utilisées par un bon nombre de classes expliquées ci-dessus.

## Editor

Dans notre version du jeu, il est également possible de créer ses propres niveaux.
Il était donc nécessaire de modéliser l'objet répondant à cette demande. Grâce à une chaîne de caractères décrivant le thème du fichier d'entité choisi, on instancie un éditeur qui créera deux listes contenant des contraintes, et un solveur.
L'éditeur sera complété au fur et à mesure de l'avancement de l'utilisateur, c'est pourquoi il était important de créer des beans pour l'image choisie pour représenter le niveau (getImage()/setImage()), le nom (getName()/setName()), le thème de l'entité (getEntityTheme()/setEntityTheme()), le nombre de maisons (getNumberOfHouses()/setNumberOfHouses()) et si le niveau doit comporter des animaux (getHasAnimals()/setHasAnimals()). En plus de cela, l'éditeur gère 2 listes de contraintes comme évoqué précédemment : les contraintes choisies, et les contraintes annulées.
En effet, si l'utilisateur choisi d'annuler des contraintes, elles ne sont pas perdues, elles sont stockées dans la liste constraintsUndone (pouvant être récupérées par une requête get du même nom), et lorsque l'utilisateur choisi de les remettre, elles sont de nouveau ajoutées dans constraintsChosen (pouvant être récupérées par une requête get du même nom).

Il est également possible de récupérer un tableau des maisons créé à partir du nombre de maisons donné.

Nous possédons donc un large choix de commandes : undo() permettant d'annuler la dernière action, redo() la réexécutant, reset() recommençant l'édition du début, addConstraintChosen(Constraint c) ajoutant une nouvelle contrainte à la liste, et removeConstraintChosen(Constraint c) en retirant une de la liste.
Enfin, les requêtes isValidable() et la commande createNewLevel() indiquent respectivement si le niveau ne possède bien qu'une seule solution et si toutes les données essentielles ont été entrées, avant de pouvoir créer un niveau grâce à createNewLevel().

Evidemment, le nombre de solutions trouvées par le solveur est calculé à chaque ajout ou retrait de contraintes, mais il est géré d'une certaine façon pour que cela ne soit pas coûteux en temps. La méthode calculateSolutionsSolver() s'occupe de cela.

Une fois le niveau créé, il sera ajouté par SaveManager dans le fichier .xml des niveaux personnalisés.

L'ensemble de ces choix de programmation (de la gestion par historique de contraintes, jusqu'à l'utilisation de nombreux beans) est appliqué pour permettre à l'utilisateur une plus grande maniabilité et le plus d'options possibles. Il peut faire des erreurs et revenir dessus sans trop de difficultés, et au fur et à mesure. Cela est notamment utile pour la vue qui demande une gestion en écriture des variables en deux temps.

## LogikVilleModel

Pour gérer et instancier l'ensemble de ces classes, il fallait un modèle global. C'est le rôle de LogikVilleModel.

Cette classe instancie dans le constructeur 2 SaveManager correspondant à un SaveManager pour les niveaux classiques, et l'autre pour les niveaux personnalisés. De plus, il initialise par défaut le thème des entités avec le thème classique et instancie un niveau éditable.

Etant donné que LogikVilleModel sert de modèle principal, il comporte un bon nombre de requêtes servant à faire le lien entre les entrailles du modèle et la vue.
Tout d'abord, il permet de récupérer le niveau courant (getCurrentLevel()), le thème d'entités actuellement appliqué (getEntityTheme()), et le menu d'édition de carte (getEditableLevel()). Il fait ainsi le lien avec de nombreuses classes, notamment en permettant d'accéder aux méthodes de Level et d'Editor.

Il est ensuite possible d'obtenir le nombre de niveaux disponibles dans chaque SaveManager instancié (getClassicLevelNumber() et getPersonnalizedLevelNumber()) mais également le lien vers l'image représentant le niveau personnalisé courant (getImagePathPersonalized()) et son nom (getNamePersonalized()). Il existe aussi une requête getLevelNumberFromName prenant une string correspondant au nom du niveau, et permettant de récupérer le numéro du niveau à partir du nom. Ces options étant uniquement

applicables sur les niveaux définis par le joueur, il est normal de ne trouver ces requêtes que pour le SaveManager correspondant aux niveaux personnalisés. Ces requêtes permettent à la vue d'afficher les données personnalisées du joueur.

Du côté des commandes, nous disposons tout d'abord d'une méthode permettant de recharger le SaveManager avec un nouveau thème d'entités, ce qui est un choix de simplicité pour la transmission des informations entre la vue et le modèle du côté des sauvegardes.
Ainsi, il est également possible de charger un thème d'entité selon un paramètre donné, pour changer dans les autres classes le thème appliqué.

De plus, comme c'est son rôle, LogikVilleModel propose 2 commandes pour ouvrir un niveau spécifique : openLevelClassic et openLevelPersonalized. Il ne demande en paramètre que le numéro du niveau qui sera communiqué par la vue grâce au clic de l'utilisateur. Toutes deux exécutent sur un thread l'instanciation du niveau, permettant une économie en temps tout en ne faisant pas figer l'application.
Il est également possible de supprimer un niveau personnalisé grâce à la commande deleteLevelPersonalized qui prend en paramètre un numéro de niveau et qui fait appel à SaveManager pour le supprimer.

Enfin, nous pouvons retrouver les commandes createEditableLevel(), appelée par le constructeur pour créer une instance de l'éditeur selon le thème d'entités appliqué, et closeLevel permettant de remettre à nul le dernier niveau instancié.

# Du côté de la vue :

## LogikVille

Se trouvant dans le package main, LogikVille constitue la classe principale de l'application. En effet, c'est elle qui constitue l'artère centrale.
Comportant un certain nombre de constantes correspondant aux différents états du jeu (autrement dit les différents menus) et une constante FPS, cette classe gère à la fois les paramètres de l'écran et l'affichage global.
On retrouve ainsi 8 constantes d'états pour les différents menus (TITLE_STATE, LEVEL_STATE, USERLEVEL_STATE, PLAY_STATE, EDITOR_STATE, VICTORY_STATE, OPTION_STATE, THEME_STATE).

Dès son constructeur, on retrouve l'une des constantes d'état du jeu indiquant le menu titre avant de créer le modèle (qui correspondra au LogikVilleModel expliqué précédemment), la vue, de placer les composants et de créer les contrôleurs.

Cette classe offre 5 requêtes permettant aux autres menus de récupérer les informations de base, comme : getScreenPixelX() et getScreenPixelY() qui permettent d'indiquer la taille de l'écran de l'utilisateur, ce qui sera utile notamment pour redimensionner les composants.

On peut également obtenir grâce à la requête getMenu qui prend un entier correspondant à un des états du jeu décrit par les constantes, un AbstractMenu (expliqué plus bas).

Du côté des commandes, on trouve tout d'abord la méthode d'affichage display qui fait elle même appel à la méthode refresh que l'on expliquera peu après.
Il est également possible de changer l'état de jeu par la méthode setGamestate, qui est un bean avec la requête setGameState(). Cette commande permet notamment d'afficher le menu correspondant à l'entier donné grâce à une classe énumérée.
On dispose également d'une commande setFps, setFramzSize et setFullscreen qui permettent de changer des options d'affichage (notamment grâce au menu des options).

La commande resetFrame permet de recréer le menu et de changer d'état.

La commande reloadPersonalizedLevelMenu permet quant à elle de réinstancier le menu des niveaux personnalisés en repartant de 0.

Mais la plupart des méthodes constituant l'artère centrale de la classe sont des méthodes outils. Par exemple, createModel() et createView(). Comme expliqué plus haut, createModel() s'occupe d'instancier un LogikVilleModel() seulement.
createView() quant à elle ne fait que définir la taille de l'application à son démarrage et appeler la méthode createMenu().

createMenu() est la méthode qui s'occupe notamment d'instancier tous les attributs comme il le faut, notamment tous les menus de type AbstractMenu. Une fois ceux-ci créés, ils sont ajoutés dans une Map associé à leur numéro donné dans les constantes expliquées plus haut. De cette façon, il est plus simple de naviguer entre les menus et les états de jeu.

Enfin, on retrouve les méthodes placeComponents(), se contentant de placer le menu de numéro gamestate dans la map sur la frame; createController() ne faisant que clôre l'application si l'on clique sur la croix et refresh n'appelant que repaint sur la frame.

Pour finir, la méthode main exécute sur EDT l'application grâce à la classe LogikVille.

## AbstractMenu

Afin de réaliser les nombreux menus de notre application, il nous fallait une classe englobante de tous les menus. Cette classe, héritant de JPanel, permet de définir le squelette d'un menu.
Celle-ci récupère la référence vers la classe mère : LogikVille. Cela permet par la suite aux contrôleurs des sous-classes d'accéder aux modèles mais également aux autres sous-menus.
Cette classe définit également des attributs statiques publiques, servant de paramètres globaux à toutes les autres classes.
Mais l'intérêt premier de cette classe réside dans toutes les méthodes outils qu'elle définit :

- getResizeX(int) : permet de redimensionner une valeur proportionnellement à la taille de l'écran en largeur
- getResizeY(int) : permet de redimensionner une valeur proportionnellement à la taille de l'écran en hauteur
- getFontSize(float) : permet de calculer la taille de police d'une écriture proportionnellement à la taille de l'écran en hauteur et en largeur
- addGameStateListener(JButton, int) : permet de rajouter un actionListener sur le JButton permettant de se rendre dans le menu d'état int. (voir LogikVille)
  De plus, cette méthode possède une surcharge lui ajoutant un paramètre Runnable : ce runnable est exécuté juste après le changement d'état, permettant une plus grande modularité
- buildConstraintLabel(Constraint, int, int) : Permet de construire un ConstraintLabel en fonction du nombre d'entités que l'on lui donne. De plus, le dernier paramètre permet de définir la taille du composant.

## VictoryMenu

VictoryMenu est une petite classe qui se contente d'afficher un message en cas de victoire sur un niveau.

Son constructeur est donc similaire à ceux des autres sous classe d'AbstractMenu.
Elle ne comporte que 3 méthodes outils constituant la base du modèle MVC :

- createView() s'occupant d'afficher les images et réglant leur taille ainsi que le texte dans des composants spécifiques, en instanciant également un bouton continuer pour retourner au menu des niveaux;
- placeComponents() utilisant le principe des Box verticales pour placer les éléments de haut en bas et pouvoir appliquer des glues, ainsi qu'un RoundPanel utilisant un WrapLayout pour redéfinir la taille du panel
- et createController() ne faisant qu'appeler la méthode addGameStateListener sur le bouton continuer pour retourner au menu des niveaux.

## TitleMenu

Tout comme VictoryMenu, TitleMenu ne possède qu'un constructeur et 3 méthodes outils.
De manière similaire, createView() s'occupe d'instancier tous les boutons se trouvant sur le menu (jouer, éditeur de cartes, l'aide, les options, et le bouton quitter), ainsi que les images de fond et le logo en redéfinissant leur taille.

placeComponents() utilise quant à lui 2 tableaux : un tableau de boutons et un tableau de GBC, ce qui permet de synthétiser le code de placement dans une simple boucle sur le nombre de contraintes du GBC et de les ajouter un à un sur la frame.
Les poids et le placement ont été étudiés au préalable grâce à un schéma pour permettre le rendu le plus agréable possible.

createController() ajoute sur les boutons quitter et aide les systèmes respectifs pour quitter l'application et ouvrir l'aide sur le navigateur, tout en permettant aux autres boutons de rediriger vers leur menu respectif.

# EditorMenu

EditorMenu est obtenu en cliquant sur le bouton éditeur de carte sur l'écran titre. C'est un menu complet composé de nombreuses informations.
Tout d'abord, on peut retrouver 4 constantes : le nombre de contraintes, le nombre de contraintes unaires, le nombre de contraintes binaires, le nombre maximum de solutions affichables et la taille des solutions.

EditorMenu possède également un grand nombre d'attributs pour chaque composant qu'il doit utiliser. En effet, l'édition de niveau requiert un grand nombre d'informations, on retrouve par exemple les boutons de navigations ou de validation, les composants d'affichage comme la carte, les JComboBox permettant de faire défiler les choix possibles pour l'utilisateur, ou encore des attributs plus à usage internes à la classe.

Tout comme les autres classes héritant d'AbstractMenu, son constructeur est identique et elle possède principalement 3 méthodes outils.
createView() instancie l'ensemble des attributs, boutons, JLabel, images et dimensions de ces dernières. Pour synthétiser le code, il n'est pas rare de retrouver des tableaux et des boucles pour rassembler les composants de même type.

placeComponents() utilise le principe de GridLayout pour répartir équitablement les différentes parties de l'éditeur, mais aussi un GridBagLayout pour placer plus facilement les composants. Cette méthode peut paraître longue et difficile à comprendre, mais elle fait seulement intervenir l'ensemble des composants graphiques nécessaires en les affichant sur différents panels et en utilisant les spécificités de chaque LayoutManager pour les utiliser avec le plus de précision et d'efficacité possible.

createController() quant à lui appelle les différentes méthodes du modèle (récupéré auprès d'AbstractLevel) pour instancier les bonnes choses selon les composants qui reçoivent une notification. Néanmoins, pour utiliser les méthodes d'ajout de contraintes, il faut disposer d'une instance de contraintes. EditorMenu devait donc gérer tout d'abord la transformation des informations envoyées par le client en contrainte : c'est l'utilité de la méthode createConstraint. En fonction de l'indice, elle instanciait une contrainte du bon type et la renvoyait.
Mais graphiquement nous devions également afficher une liste des solutions. Pour plus de limpidité dans le code, c'est la méthode createSolutionsList() qui s'en charge, en récupérant depuis le modèle la liste des solutions et en affichant chaque information en composant graphique.

setNamesMenu s'occupe de préparer la liste des rôles/noms des entités pour les JComboBox, ce seront les listes de choix pour ajouter une contrainte.
Dans cette même idée, calculateComboBox calcule selon la contrainte les noms d'entités à ne pas afficher : par exemple s'il s'agit d'une contrainte de type 1 ou 2, unaire donc, la

deuxième JComboBox contiendra les numéros des maisons. S'il s'agit d'une contrainte de type 6 ou 7, la première JComboBox contiendra les personnages et la seconde les animaux.

Cette différenciation est possible grâce à un type énuméré imbriqué : ConstraintEnum, répertoriant les différents types de contraintes ainsi que leur nom affichable dans leur liste.
Cela permet un code plus factorisé et plus clair, tout en permettant d'associer à chaque contrainte une méthode indiquant si cette contrainte peut être affichée en fonction de si elle utilise des animaux ou non. Par exemple, si on ne souhaite pas avoir des animaux sur le niveau, les contraintes 6 et 7 ne seront pas affichées dans la liste.

## CreatorThemeMenu

Ce menu est accessible par le menu d'options grâce au bouton pour créer son propre thème.
Il permet donc la gestion des thèmes préexistants mais aussi en voie d'existence par l'utilisateur.
Cette classe possède un bon nombre de tableaux de JLabel, de JTextField et de boutons pour permettre à l'utilisateur de modifier tous les personnages et animaux disponibles. De plus, elle possède un certain nombre de listes permettant la gestion interne.

Le constructeur de CreatorThemeMenu se contente d'appeler celui d'AbstractMenu, mais il appelle aussi createModel(). Son modèle est un ThemeSaver, défini précédemment comme s'occupant de la gestion des fichiers de thèmes.
Il est aussi possible de récupérer ce model par le biais de la requête getModel().

Comme dit précédemment, cette classe possède un bon nombre de listes. Il s'agit en fait des listes d'images et de noms pour les animaux et les personnages. Ces listes sont affectées par le biais des commandes set(Character/Animal)DefaultNames et set(Character/Animal)DefaultSprites. On peut également récupérer de la même façon une liste des chemins des images (set(Character/Animal)ImagePath).

Enfin les deux dernières commandes sont initMenu() et setInfoMenu. initMenu() agit de la même manière que les constructeurs précédents : un appel à createView(), placeComponents() et createController() mais aussi à revalidate sur l'instance en cours. L'objectif est d'actualiser l'affichage de l'ensemble des éléments.
setInfoMenu prend en paramètre un thème d'entités et appelle les commandes définies précédemment pour gérer les listes en donnant en paramètre les listes renvoyées par le modèle.

createView() a donc le rôle d'initialiser tous les composants graphiques et les listes qui serviront pour l'affichage. On redéfinit également la police et la taille des images pour rendre l'affichage harmonieux.

Nous avons également conçu une méthode getThemesNames() qui a pour objectif d'identifier dans le dossier des thèmes quels sont les thèmes déjà existants. Par défaut, on retire de cette liste le thème classic qui n'est pas modifiable. En effet, l'objectif de cette liste est de pouvoir charger les thèmes et les modifier à souhait.

placeComponents() utilise le principe des GridBagLayout pour afficher d'un côté les personnages de l'autre les animaux avec leur image, leur nom, et le bouton pour choisir une autre image dans notre bibliothèque. On utilise également le système des GridBagLayout pour les cadres de chargement de thèmes et de nomination de nouveaux thèmes. Il est en effet plus facile de placer les éléments graphiques tel qu'on le souhaitait avec un GridBagLayout. Ainsi, il ne restait qu'à ajouter les boutons de validation, retour et réinitialisation sur un BorderLayout.

Dans createController(), on retrouve les écouteurs sur les tableaux de boutons pour choisir l'image, ou encore pour confirmer le chargement ou pour recommencer l'édition. Néanmoins, c'est le bouton de confirmation qui effectue le plus : c'est lui qui fera changer d'état le modèle en fonction des données récupérées de part et d'autre, allant jusqu'à copier les images dans les sous répertoires créés dans le dossier de thèmes. C'est ici que les listes créées sont utilisées, pour appeler la méthode du modèle permettant de copier l'image choisie dans les dossiers.

# OptionMenu

OptionMenu permet de changer la résolution et le fait de passer en plein écran, mais c'est aussi le menu par lequel on doit passer pour charger un thème en jeu ou aller sur le menu d'édition des thèmes.
Son constructeur est similaire aux autres classes étendant AbstractMenu.
Cette classe ne possède qu'une commande : resetThemeList() qui permet de mettre à jour la liste des thèmes chargeables après avoir créé un nouveau thème.

Les méthodes outils comportent createView(), se contentant toujours d'initialiser les composants de la vue et de régler leur affichage, et placeComponents() utilisant le placement de haut en bas des box verticales.
createController() s'occupe seulement du bouton valider pour appliquer les préférences choisies par l'utilisateur, et changer l'état des menus en fonction de si l'on clique sur le bouton retour ou des thèmes, le dernier étant exécuté sur un thread pour gagner en temps d'exécution.

Nous disposons ensuite des méthodes themeList() retournant l'ensemble des thèmes disponibles dans le dossier du même nom, resolutionBuilder() qui renvoie un tableau des dimensions disponibles pour que l'utilisateur fasse son choix, et une classe imbriquée Resolution décrivant la syntaxe de ces dimensions.

# PlayMenu

PlayMenu est la classe s'occupant du plateau de jeu, faisant le lien avec les classes du modèle sur ce point.
A cet effet, la classe comporte un bon nombre d'attributs indiquant le nombre d'entités en jeu, si le niveau comporte des animaux, ou encore les listes de contraintes et des images des entités.

Mais elle contient aussi tous les composants graphiques affichables, comme la carte, la liste des contraintes (sous format ConstraintLabel), les boutons et les entités.

Le constructeur de PlayMenu prend 2 paramètres : une instance de LogikVille et une frame.
La commande initMenu se charge d'appeler les méthodes outils telles que createView(), placeComponents() et createController() en faisant d'abord un appel à removeAll() pour s'assurer d'enlever tous les composants.

La classe contient 5 autres commandes servant davantage à initialiser les listes et le nombre d'entité ainsi que le booléen pour savoir si le niveau comporte des animaux.

Du côtés des méthodes outils, on retrouve createView() dont le rôle reste inchangé, et placeComponents() utilisant des gridBagLayout pour placer les personnages et animaux sur la fenêtre. Mais aussi, une box verticale pour ajouter de haut en bas les contraintes sur la carte, et une box verticale pour le placement des maisons.
createController() doit quant à lui gérer le drag and drop des entités dans les maisons. Pour cela, on utilise une classe imbriquée MouseHandler qui va donner les informations elles-mêmes au modèle. C'est donc elle qui réalise vraiment le code du drag and drop.

# AbstractLevelMenu

Cette classe sert notamment pour les menus de navigation entre les niveaux qu'ils soient classiques ou personnalisés. Grâce à cette classe abstraite, il est ainsi possible de factoriser efficacement le code des menus de niveaux classiques et personnalisés.

Le constructeur de AbstractLevelMenu permet la customisation de celui-ci : c'est lui qui permet d'indiquer si on veut les niveaux personnalisés ou classiques, et donc de changer le texte des composants graphiques en conséquence.

Le constructeur réalise 3 appels de méthode outils :

- createView() : qui va créer les composants de la vue, notamment les boutons de navigation, le champ de recherche de niveau, etc.
- placeComponents() : qui va placer les composants de la vue. Pour cela, on utilise un BorderLayout, afin de placer de façon globale tous les composants que l'on veut. De plus, les boutons de sélection de niveau sont eux placés sur une GridLayout au centre du borderLayout.
- createController() : cette méthode se charge tout d'abord d'ajouter des contrôleurs sur les boutons de navigation. Ceux-ci sont configurés de telle sorte à ce que si on arrive au bout de la liste des niveaux, on revient sur la première page (pareil en sens inverse pour aller sur la dernière page).
  De plus, le contrôleur de la recherche de niveau permet d'entrer un entier, le numéro de ce niveau, afin de directement se rendre sur la page où celui-ci se trouve.
  Enfin, une méthode outil protégée permet de créer un propertyChangeListener réalisant le chargement d'un niveau. Celle-ci permet de factoriser encore une fois le code de LevelMenu et PersonnalizedLevelMenu

# LevelMenu

LevelMenu est la classe permettant d'afficher graphiquement les niveaux classiques du jeu. Celle-ci hérite directement de AbstractLevelMenu afin d'en récupérer les fonctionnalités. En effet, en utilisant son constructeur et en lui passant les bonnes données, il est possible de totalement configurer AbstractLevelMenu pour que celui-ci affiche bien les niveaux classiques.
De plus, LevelMenu redéfinit CreateController afin de rajouter des comportements :

- Afin de naviguer efficacement, le bouton "Niveaux Personnalisés" doit bien mener vers le menu "Niveaux Personnalisés". Le code étant inversé dans le menu PersonalizedLevelMenu, il est nécessaire de le laisser ici.
- Nous ajoutons aussi des contrôleurs sur chaque LevelButton, afin de permettre d'ouvrir un niveau classique.
- Finalement, en utilisant la méthode outil de AbstractLevelMenu renvoyer un PropertyChangeListener de chargement, on écoute le modèle afin de détecter un chargement de niveau classique.

Enfin, LevelMenu implémente deux méthodes abstraites de AbstractLevelMenu : createLevelButtons et setLevelButtons. En effet, le code d'instanciation des LevelButtons doit etre différent entre le LevelMenu et le PersonalizedLevelMenu : Dans LevelMenu, les textes des boutons sont toujours "Niveau" + le numéro du niveau du bouton ainsi que son image. Nous verrons plus bas comment marche PersonalizedLevelMenu.

# PersonalizedLevelMenu

PersonalizedLevelMenu est la classe permettant d'afficher graphiquement les niveaux personnalisés du jeu. Celle-ci hérite également de AbstractLevelMenu en la configurant afin d'avoir l'affichage des niveaux personnalisés.

Celle-ci va également ajouter des contrôleurs comme l'a fait LevelMenu afin d'avoir les contrôleurs nécessaires au fonctionnement du menu. Mais aussi, la gestion d'un menu surgissant afin de pouvoir supprimer un niveau, et les messages associés à cette action.

Cependant, le code des méthodes createLevelButtons et setLevelButtons est légèrement plus complexe : en effet, l'utilisateur en créant un niveau personnalisé peut donner une image et un texte prédéfini. Il faut donc pouvoir récupérer ces deux éléments la, s'ils existent, afin de remplacer le texte par défaut et les images par défaut.

# ClassicButton

Cette classe est un outil utile de la vue. En effet, elle étend un JButton et permet d'en modifier l'aspect initial. Le but étant de rendre l'application la plus attrayante possible, nous

avons modifié les couleurs en fonction de si le bouton est survolé, cliqué, par défaut, ou non cliquable.
Nous avons donc 4 constructeurs dont 3 publiques, chacun prenant des paramètres différents et initialisant donc des couleurs différentes selon la nécessité.

Il est possible de récupérer la couleur du bouton initiale grâce à la requête getColor(). De plus, nous disposons de la commande paintComponent() que nous avons redéfinie pour recalculer la forme du bouton.

Enfin, nous avons plusieurs méthodes outils permettant : l'initialisation du bouton (initButton()) appelant également createController() qui va appliquer les couleurs pour les boutons cliqués ou décliqués; getCenterXForText et getCenterYForText quant à eux permettent d'aligner le texte sur la partie centrale sur l'axe vertical et horizontal.

# EntityLabel

EntityLabel est un type de Label utilisé pour les entités. Elle ne comporte qu'un constructeur et une requête. Le constructeur prend en paramètre une image et un indice correspondant à celui de l'entité. Ainsi, l'entité graphique est représentée par l'affichage de son image, à laquelle on a associé son indice (que l'on récupère avec getIndex()).

# GraphicCard

GraphicCard est un type de JPanel servant pour l'affichage de la carte en jeu.
On lui définit des couleurs particulières qui seront attribuées par paintComponent.
GraphicCard possède un constructeur prenant un texte décrivant la carte, une largeur et une hauteur. Les dimensions définies par cette hauteur et cette largeur sont accessibles par getPreferredSize(), mais aussi par getMaximumSize() qui sont redéfinies pour avoir la même taille toute deux (autrement dit la taille préférée est aussi la taille maximale).
paintComponent() permet également de donner aux angles de la carte un aspect plus arrondi. L'ensemble est fait de manière à ce que la carte ait un aspect esthétique et s'imprègne dans l'ensemble de la vue.
Enfin, tout comme pour les ClassicButton, nous retrouvons les méthodes getCenterXForText() et getCenterYForText() servant à aligner sur les axes verticaux et horizontaux le texte.

# GraphicHouse

Tout comme GraphicCard, GraphicHouse est un type de JPanel. Celui-ci a pour objectif de représenter une maison d'un point de vue graphique.
Le constructeur de GraphicHouse prend 3 paramètres : la taille en largeur, en hauteur et l'indice de la maison.
Nous disposons de plusieurs requêtes, comme isValidAddition qui à partir d'un booléen regarde si la partie haute ou la partie basse de la maison sont déjà occupées et donc s'il est possible d'ajouter une entité à cette partie.

De la même manière, il existe deux requêtes isComponentNorthAlreadyInUse() et isComponentSouthAlreadyInUse() indiquant si la partie haute ou la partie basse de la maison sont occupées ou non, ce sont des beans, nous disposons donc de leur méthode set associées (setComponentSouthAlreadyInUse() et setComponentNorthAlreadyInUse()) ainsi la propriété est dynamique dans le temps. Mais il est aussi possible de récupérer ce composant (au moyen des requêtes getComponentNorth() et getComponentSouth()) ou encore de les affecter à la maison (addComponentNorth et addComponentSouth). Il est possible également comme avec EntityLabel de récupérer l'indice de la maison au moyen de getIndex(). Enfin, la méthode paintComponent permet d'afficher l'image de la maison.

## ConstraintLabel

Cette classe permet de représenter graphiquement une contrainte. Elle dérive un JComponent pour créer un composant graphique.

Ce composant graphique se décompose en 3 parties : une EntityLabel à gauche, une EntityLabel à droite (les deux sont optionnelles, rajoutables via des fonctions addEntityLeft et addEntityRight) et enfin une image représentant la contrainte graphiquement au milieu.

Pour créer l'image représentant la contrainte, nous avons séparé le code en deux parties : soit il s'agit d'une contrainte binaire, et donc il suffit de charger l'image correspondante, soit il s'agit d'une contrainte unitaire. Dans ce cas là, nous avons une fonction outil réalisant l'image : createConstraintUnitary().

Cette fonction outil va, à partir du nombre de personnages, réaliser l'image de la contrainte. Cette contrainte étant plus complexe à représenter, nous avons choisi de la subdiviser en plusieurs images. Ainsi, nous avons un JPanel doté d'un GridLayout avec en nombre de colonnes le nombre d'entités et une ligne, pour disposer nos images dedans. Ensuite, nous rajoutons des ronds par défauts dans notre JPanel et si on arrive sur le numéro de l'entité sélectionnée, on rajoute la bonne image en fonction du type de la contrainte (1 ou 2).

## GraphicSolution

GraphicSolution est une sorte de JComponent permettant de représenter graphiquement une solution. Concrétement, il s'agit d'un composant doté un GridLayout configurable avec des bordures stylisées. On peut donc afficher ce que l'on veut avec. Par exemple, dans notre cas, nous mettons des EntityLabel dedans pour représenter notre solution graphiquement, mais nous aurions pu mettre des JLabel avec les noms des personnages.

## RoundPanel

RoundPanel est un type de JPanel. Possédant un constructeur prenant 2 paramètres (une couleur et un LayoutManager), c'est lui qui permet d'affecter une couleur et d'arrondir les angles à un layout au moyen de sa méthode paintComponent.

## LevelButton

Ce composant est une sorte de ClassicButton : il représente lui aussi un bouton avec deux couleurs (la couleur non sélectionnée vaut null), mais en plus du classicButton celui-ci peut contenir une image.

Ainsi, à l'instanciation de ce composant, nous pouvons lui passer une Icon qui représente l'image de ce bouton. De plus, il est également possible de lui rajouter un texte.
De base, la couleur de ce bouton est une couleur transparente. Ainsi, quand il n'est pas sélectionné seulement l'image et le texte s'affiche mais quand la souris passe dessus, un fond orange se rajoute.

De plus, des méthodes outils ont été rajoutées afin de pouvoir calculer où dessiner l'image pour que celle-ci soit centrée.

## WrapLayout

Le WrapLayout est un LayoutManager permettant de corriger un bug du FlowLayout dans certains cas : si un FlowLayout est lui-même disposé dans au Layout utilisant la composante "preferredSize" verticale d'un composant, alors seulement la taille de la première ligne du FlowLayout sera utilisé. Ainsi, certaines lignes peuvent être coupées voire absentes. Ce nouveau Layout règle ce problème en calculant la taille préférée verticale à chaque ajout de composant (calcul du "Wrap" en anglais).
Code trouvé sur internet.