Jeu de Construction

-

Rapport Java - 1re Partie

Pierre-Marie Drévillon, Mathieu Le Steun

16 mars 2007

Table des matières

Introduction

Le programme présenté tente de simuler la construction et le fonctionnement d'une ville de manière globale. L'idée est de faire évoluer cette ville en fonction des demandes de logement : tous les mois la population de sans-logis est mesurée et va déclencher toutes les opérations d'aménagement du territoire. Le programme cherche donc chaque mois à loger un maximum de personnes en fonction d'un certain nombre de contraintes. Afin d'assurer une interactivité minimale entre le programme et le joueur, on propose à celui-çi de choisir une immigration mensuelle et d'en observer les conséquences au travers d'indicateurs chiffrés de l'état de la ville (budget, pollution, nombre d'habitations, ressources disponibles, etc.). Bien évidemment des seuils sont définis : si par exemple le joueur mène la ville à la faillite, ou qu'il dépasse le taux de pollution autorisé, il perd la partie !

L'intérêt du programme réside donc dans sa capacité à faire évoluer un ensemble de variables (qui dépendent toutes les unes des autres) de manière cohérente. Cet objectif est d'autant plus ambitieux que la simulation se veut complète et réaliste : pour cette raison, nous avons choisi de limiter le nombres de classes et de variables disponibles, afin de faciliter la mise en œuvre du jeu et de clarifier son fonctionnement algorithmique. Ainsi, par exemple, seuls deux types de ressources (bois et pierres) sont disponibles, pour éviter une surabondance de variables et de possibilités qui n'apporteraient rien d'intéressant en termes de programmation.

1 Idées générales

Avant de décrire chacune des classes du programme, tentons de décrire son fonctionnement général. Commençons par dire que le "cœur" du programme est situé dans la classe Ville : c'est elle qui contient le *main()* et qui va contenir les méthodes "de haut niveau" de la ville (logement des sans-abris, demande de construction des routes, affichage du rapport mensuel, etc.). A partir de cette classe sont donc gérées les ressources et les constructions de la ville, qui sont stockées sous forme de listes chaînées afin de faciliter leur parcours et leur gestion. Chaque type de ressource ou de construction dispose de sa propre liste, et afin d'assurer une factorisation efficace du code, plusieurs niveaux d'héritage sont utilisés (voir figure 1). Pour finir, précisons le rôle de la classe Global : elle sert principalement de "fichier de stockage" pour toutes les variables *globales* de la simulation et n'est donc pas instanciée (toutes les variables et méthodes qu'elle contient sont du type *static*).

2 Description des classes

2.1 La classe Global

Cette méthode (voir figure 2) contient toutes les variables *globales* de la simulation (budget, pollution, etc.), stockées sous forme de variables de classe (i. e. du type *static*). Elle contient également des méthodes telles que *titre()* qui sont utilisées de façon globale par plusieurs autres classes.

2.2 La classe Ville

La classe Ville (voir figure 3) a un double rôle : elle contient le constructeur de la ville et permet donc, en étant instanciée, de créer l'objet *ville* en lui-même, et elle constitue également le centre nerveux du programme avec les méthodes de plus haut niveau. C'est en effet dans cette classe que se trouvent les méthodes chargées de loger la population, d'aménager le territoire et de faire avancer le temps.

2.2.1 La méthode *loge()*

Cette méthode récursive permet de loger les nouveaux immigrants et les SDF. Elle cherche d'abord a compléter les immeubles déjà construits (mais pas les maisons). Si une demande de logement est supérieure a cette capacité, elle cherche a loger les personnes restantes a hauteur de la capacité d'accueil mensuelle de la ville en construisant de nouveaux habitats (car il n'y a plus de places libres). On construit des maisons ou des immeubles suivant le ratio actuel et le ratio souhaité. Ainsi, il peut y avoir des SDF et des places libres, ce qui peut paraître aberrent (mais c'est la

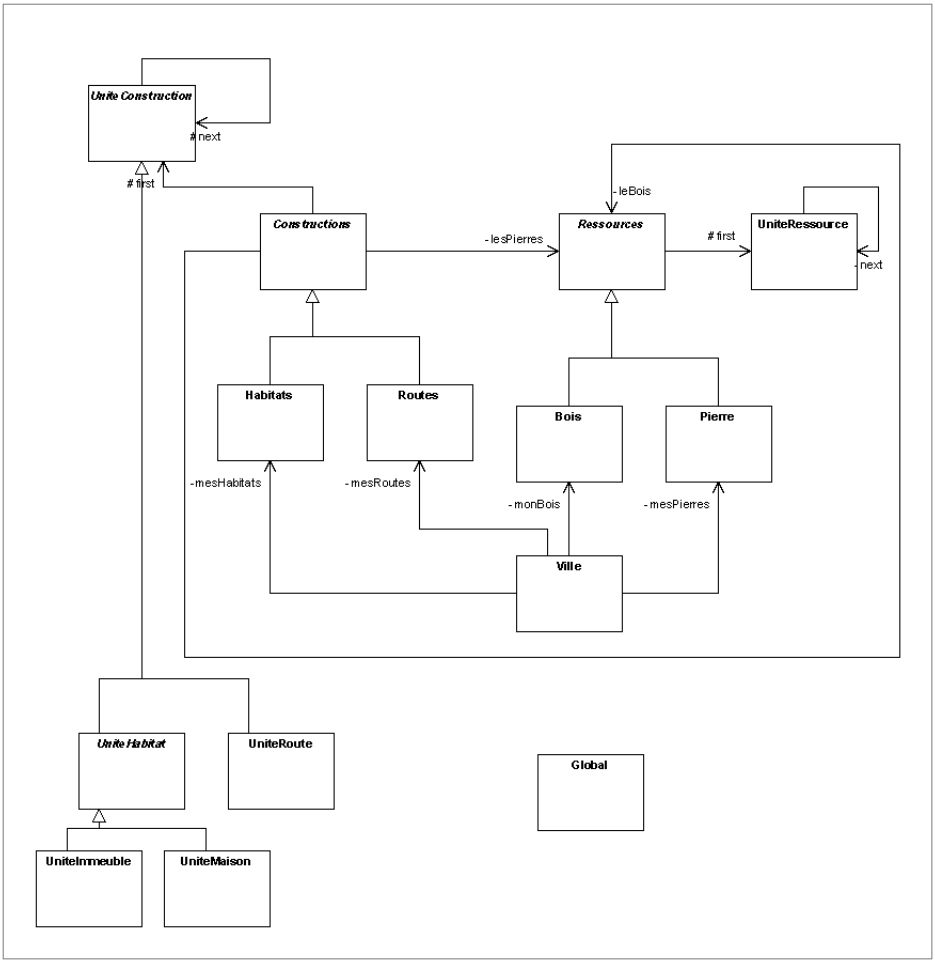

FIG. 1 – Diagramme de classes condensé

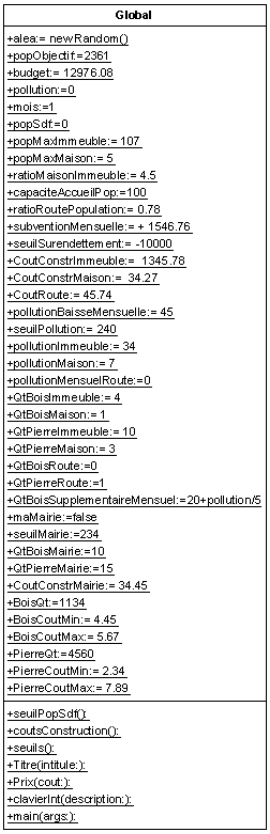

FIG. 2 – Diagramme de la classe Global

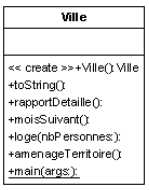

FIG. 3 – Diagramme de la classe Ville

réalité !) car si on dépasse le seuil d'accueil mensuel, on a des SDF, mais la conjoncture des habitations fera qu'il sera peut-être nécessaire de construire un immeuble qui ne sera pas rempli.

2.2.2 La méthode *amenageTerritoire()*

Cette méthode aménage le territoire en fonction de l'avancement de la ville (construit des routes et la mairie si la population atteint un certain seuil).

2.2.3 La méthode *moisSuivant()*

Cette méthode permet de faire avancer le temps d'un mois. Pour cela elle :
- fait baisser la pollution et rajoute les subventions mensuelles au budget.
- vérifie que les seuils de "GAME OVER" ne sont pas atteints.
- fait interagir le joueur en lui demandant le nombre d'immigrants pour le mois.
- loge les SDF (dans la mesure du possible) grâce la méthode *loge()*.
- aménage le territoire grâce a la méthode *amenageTerritoire()*.
- crée les ressources qui se régénèrent.
- affiche le rapport mensuel d'activité.

2.3 Les Ressources

Il existe deux types de ressources (bois et pierres) qui sont gérées par des listes chaînées (voir figure 4). La classe *UniteRessource* permet de construire les "nœuds" des listes de type *Bois* et *Pierre* (qui héritent de la classe abstraite *Ressources*). Ainsi, chaque objet dans une liste représente une unité élémentaire de la ressource considérée, rangée selon son coût. Ces listes sont générées lors de l'initialisation du programme et sont ensuite maintenues à jour au fil des commandes et du replantage des arbres (il est par contre impossible de rajouter des pierres). Afin de mieux cerner ces mécanismes de gestion, nous allons décrire rapidement les méthodes les plus significatives.

2.3.1 La méthode *commande()*

Cette méthode récursive permet de commander une certaine quantité n d'une ressource en supprimant les n premiers éléments de la liste considérée (qui sont les moins chers car les listes sont triées). Il est à noter le budget global de la ville est modifié en conséquence. De plus, cette méthode génère une erreur si la ressource est épuisée avant que la commande ne puisse être satisfaite.

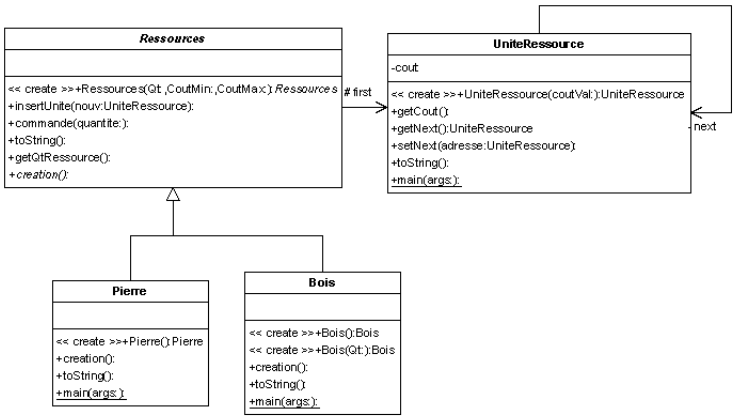

FIG. 4 – Diagramme des classes gérant les ressources

2.3.2 La méthode *insertUnite()*

Cette méthode permet l'insertion des ressources dans la liste en les triant par coût croissant. Elle peut-être testée grâce au test unitaire de la classe Bois.

2.3.3 La méthode *creation()*

Cette méthode permet de créer des nouvelles ressources en bois tous les mois, et ainsi de simuler un reboisement de l'environnement.

2.4 Les constructions

De même que les ressources, les constructions (sauf la mairie) sont gérées par des listes chaînées, chaque "nœud" représentant une unité de construction. La gestion de l'héritage est plus complexe qu'avec les ressources (voir figure 5) : on distingue en effet plusieurs types de constructions, chaque type ayant des caractéristiques et des méthodes propres.

2.4.1 Les habitats

Il y a deux types d'habitats : les immeubles et les maisons. Il sont construits depuis la méthode *loge()*, en fonction des demandes de logement, des ressources disponibles et du ratio souhaité entre le nombre de maisons et d'immeubles. Hormis une capacité d'accueil et un coût différents, les immeubles se distinguent des maisons par leur capacité à pouvoir être remplis mois après mois, en fonction de la demande. Ceci n'est pas possible avec les maisons : les citoyens du futur n'aiment pas voir arriver un inconnu sous le toit de leur habitation particulière... Chaque habitat (du

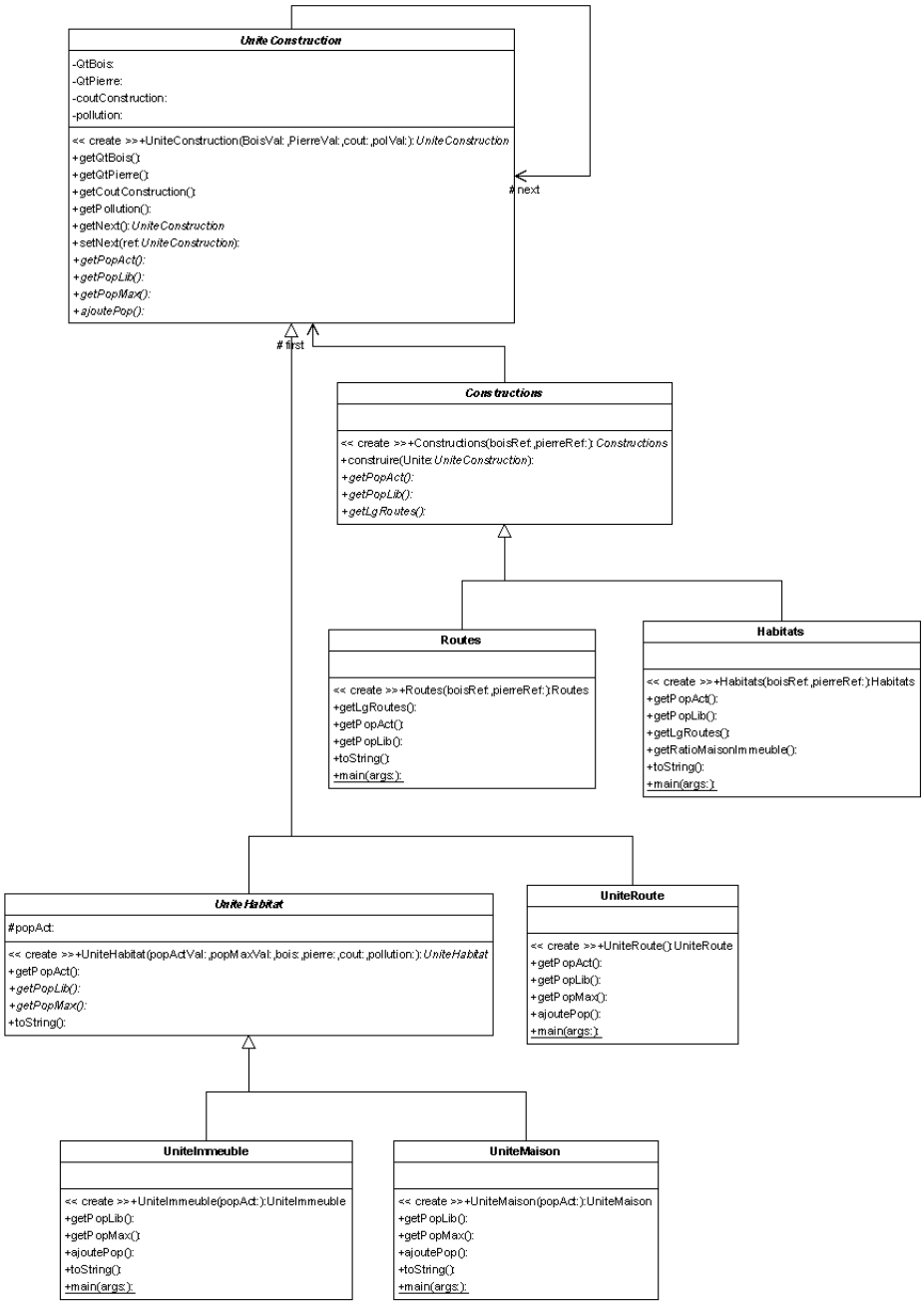

FIG. 5 – Diagrammes des classes gérant les constructions

type *uniteHabitat*) contient des informations chiffrées sur le nombre de personnes qui y vivent, le nombre de places libres (pour les immeubles), etc.

La méthode *construire()*

Cette méthode permet de "construire" une nouvelle unité :
- elle commande le bois et les pierres nécessaires avec le méthode *commande()*
- elle agit sur le budget global et sur la pollution
- elle place l'unité construite en première position de la liste concernée

Remarquons que cette méthode est aussi valable pour les routes.

2.4.2 Les routes

Chaque nœud dans la liste représente un tronçon de route de 1 km, construits depuis la méthode *amenageTerritoire* qui gère la longueur du réseau routier par rapport à la population.

2.4.3 Le cas particulier de la mairie

La mairie est le seul bâtiment qui ne soit pas géré par une liste chaînée : il aurait été inutile de créer une liste pour un seul élément. Il s'agit en fait d'une variable booléenne de la classe Global qui passe de *false* à *true* lorsque celle-ci est construite. L'ordre de construction est passé par la méthode *amenageTerritoire* lorsque la population dépasse un certain seuil. On pourrait également se passer de cette variable qui contient une information redondante. En effet, la construction de la mairie est déclenchée par le dépassement d'une seuil de population. Sachant que la population ne décroît pas, si la population est supérieur à ce seuil, la mairie est construite.

2.5 Tests unitaires

Pour chaque classe, nous avons cherché à utiliser toutes les méthodes de la classe et, le cas échéant, à déclencher les exceptions. Pour les classes abstraites (donc qui ne peuvent être instanciées), le test se fait au travers des classes filles.

3 Base de données

3.1 Position du problème

Actuellement, notre programme ne nous permet pas de générer des villes de tailles différentes (i. e. village, ville moyenne, métropole, etc.) : nous ne disposons en effet que d'un seul jeu de constantes qui est stocké dans la classe Global. Afin de gérer plusieurs jeux et pouvoir ainsi choisir le type de ville à construire, nos envisageons l'intégration d'une base de données à notre programme. Afin de donner une idée au lecteur du contenu de cette base, voici regroupés par "genres" les différentes *variables globales* que nous utilisons actuellement.

3.2 Base de données relationnelles

Les bases de données remplaceront avantageusement les listes dynamiques. Chaque liste actuelle sera remplacée par 2 tables : une table "modèle" et une table des objets créés réellement, comme sur cet exemple :

Modèles de constructions						
key	Description	Qt Bois	Qt Pierre	Cout construction	Pollution	PopMax
0	Immeuble	4	10	1345.78	34	107
1	Maison	1	3	34.27	7	5
2	Mairie	10	15	34.45	7	0
3	Route	0	1	45.74	0	0

Les constructions de la ville		
key	key_etrangere	PopAct
0	0	92
1	0	78
2	3	0

3.3 Stockage de données hétérogènes

Les données suivantes se regroupent d'elles-même dans une table, car elles sont utilisées à l'initialisation du programme. Cependant, elles sont utilisées de façons différentes, ce qui limite l'intérêt de la table :

- le champ Valeur min contient des *int* et des *double*.
- les ressources Bois et Pierre sont générées en attribuant un prix à chaque unité, de façon linéaire. L'initialisation du budget est une simple variable double.

Cependant, éparpiller ces valeurs n'est pas non plus une solution satisfaisante.

Initialisation			
Entités	Qt	Valeur min	Valeur max
Bois	1134	4.45	5.67
Pierre	4560	2.34	7.89
Budget	1	12976.08	*NULL*
Pollution	1	0	*NULL*
Mois	1	1	*NULL*
popSdf	1	0	*NULL*

Le problème est similaire pour les seuils où les actions à déclencher sont très différentes, mais également pour les évènements mensuels. Petite précision concernant le champ *type* de la table Seuils :

- type minimal : si variable $>$ valeur alors action
- type maximal : si variable $<$ valeur alors action

Seuils			
Variables	type	valeur	action
Population	minimal	234	construire mairie
Pollution	minimal	240	Game Over : trop de pollution
Budget	maximal	-10000	Game Over : surrendettement
capaciteAccueilPop	minimal	100	surplut en popSdf
popSdf	minimal	0 si pop$<$500 45 sinon	Game Over : trop de SDF

Évènements mensuels	
Variables	Opération
budget	+1546.76
pollution	-45
bois	+20

3.4 Petites tables qui regroupent les dernières données

Il doit y avoir au moins ratioValeur fois plus de Variable1 que de Variable2.

Ratios		
Variable1	Variable2	Valeur
nbMaison	nbImmeuble	4.5
route	population	0.78

Unités	
Variable	String
budget	€
pollution	$mg.m^{-3}$

Conclusion

Pour conclure, nous pouvons simplement exposer nos objectifs pour la 2^{e}partie :

- réaliser les bases de données relationnelles afin de rendre plus simple la gestion des valeurs (Ville-Village) et d'arriver peut-être a un équilibre de la ville...
- trouver une solution pour stocker les données "non relationnelles" de façon intelligente
- créer une carte 2D de la ville, bien que cela nous semble difficile. Pour reprendre l'exemple des tables "Constructions", nous pensons rajouter 2 champs "dimX" et "dimY" dans la table modèle qui donne la longueur et largeur d'une unité de construction. Puis 2 champs dans la table "objets créer" qui informe de la position de l'élément (en prenant pour référence un coin ou le centre de l'objet). Nous ne savons pas encore comment fixer ces 2 dernières valeurs lors de la création de l'objet, sachant que (*à priori*) toute rotation est interdite.