AP4B : Programmation orientée objet

Rapport conception projet

Sommaire

1 Présentation du projet **3**

2 Organisation et technologie utilisés **4**

3 Architecture du projet **5**
3.1 Changements depuis le rapport de conception 5
3.2 Le modèle MVC 5
3.3 Le design pattern Observer 5

4 Cas d'utilisation et scénarii **5**
4.1 Diagramme des cas d'utilisation 5
4.2 Scénarii 6

5 Diagramme de classe **7**
5.1 Le modèle 7
5.1.1 Les types de bâtiments 8
5.2 La vue 9
5.3 Le controller 10
5.4 Le Design Pattern Observer 12

6 Diagramme d'activité **13**

7 Diagramme de séquence **14**

8 Annexe **15**

1 Présentation du projet

Le projet d'AP4B porte sur la réalisation d'un jeu de gestion en Java axé sur l'énergie. Le jeu de type "point and click" doit simuler une ville et sa consommation en énergie. Le joueur doit aménager la ville de manière à la faire prospérer en ajoutant des sources d'énergie qui peuvent être renouvelable ou non-renouvelable

Mécanique de jeu :
Le jeu évolue sur une map qui est divisée en un certain nombre de cases. Cette map possède des caractéristiques dynamiques qui sont le soleil, le vent et l'eau. Pour cela, chaque case de la map a des attributs modélisant la valeur du soleil, du vent et de l'eau. Nous avons définis des zones de cases qui sont plus propices à des fortes valeurs de soleil ou de vent. Ces caractéristiques dynamiques varient aléatoirement au cours du temps. La map possède aussi des ressources statiques. Celles-ci sont le charbon, le gaz et l'uranium. Chaque case possède ou non une ou plusieurs de ces ressources.

Il est possible de produire de l'énergie grâce à ces 6 différents types de ressources. Il existe un bâtiment différent pour produire de l'énergie à partir de chacune de ces ressources. L'exploitation des ressources fossiles pollue la map. Il y a donc un indicateur de taux de pollution qui correspond à la proportion d'énergie produite à partir de ressources fossiles par rapport à la quantité totale d'énergie produite.

Il est possible de construire des habitations (maison ou appartement) pouvant accueillir des habitants. Ces habitants payent des taxes qui donnent au joueur de l'argent pour acheter les bâtiments qu'il souhaite. Pour récolter l'argent des habitations il suffit de cliquer dessus lorsque cela est possible. Le nombre d'habitants voulant habiter dans la ville évolue en fonction du taux de pollution de la map et du prix de l'électricité. Le prix de l'électricité est calculé avec le ratio entre le nombre d'habitant (la population) et la quantité d'énergie produite.

La production d'énergie renouvelable varie donc en fonction du temps et peut chuter à certains moments. Cela a pour effet d'augmenter le prix de l'électricité et donc d'empêcher de nouveaux habitants de vouloir venir et donc de ralentir la récolte d'argent. Il faut donc équilibrer le nombre d'énergie renouvelables et fossiles. Mais attention ! Il faut faire attention de ne pas trop faire augmenter le taux de pollution sinon personne ne voudra habiter dans votre ville non plus. Le but est donc de trouver un équilibre pour continuer d'évoluer le plus rapidement possible.

2 Organisation et technologie utilisés

Pour s'organiser au sein de notre groupe, nous avons utilisé **github** pour partager nos versions de code. L'IDE que nous avons choisi est **IntelliJ**, celui-ci permet un gain de temps important par l'autogénération des fichier java à partir des diagrammes UML.

Pour la réalisation des diagrammes, nous avons utilisé **asatah** et **Visual Paradigm** en ligne pour le diagramme de cas d'utilisation.

Notre choix de librairie pour l'interface graphique s'est tourné vers **Java Swing** et **AWT** car c'est une librairie stable et relativement simple à utiliser.

3 Architecture du projet

3.1 Changements depuis le rapport de conception

Depuis le rendu du rapport de conception, l'architecture générale dont le modèle MVC n'a pas beaucoup évoluée. Notre implémentation du design pattern Observer a évoluée en raison de l'ajout d'une fonctionnalité qui est la récolte d'argent en cliquant sur les bâtiments. Au fur et à mesure que nous avons développé notre programme, nous nous sommes rendu compte d'améliorations et de modification nécessaires à notre conception. Nous les détaillons dans la suite de ce rapport.

3.2 Le modèle MVC

Le modèle MVC (Model View Controller) est un des pattern de conception des plus utilisés. Nous avons choisis d'utiliser ce modèle car il permet une grande modularité, réutilisabilité et maintenance du code. Il permet de séparer la partie graphique de la partie logique du programme tout en ajoutant une couche entre les deux permettant de faire le lien.

Nous avons donc un controller qui contient le modèle est la vue et qui permet de gérer les évènements extérieurs au programme. Dans notre cas, comme nous utilisions Java Swing pour la partie graphique, le contrôleur contiendra aussi les Listeners permettant de gérer les actions de l'utilisateur.

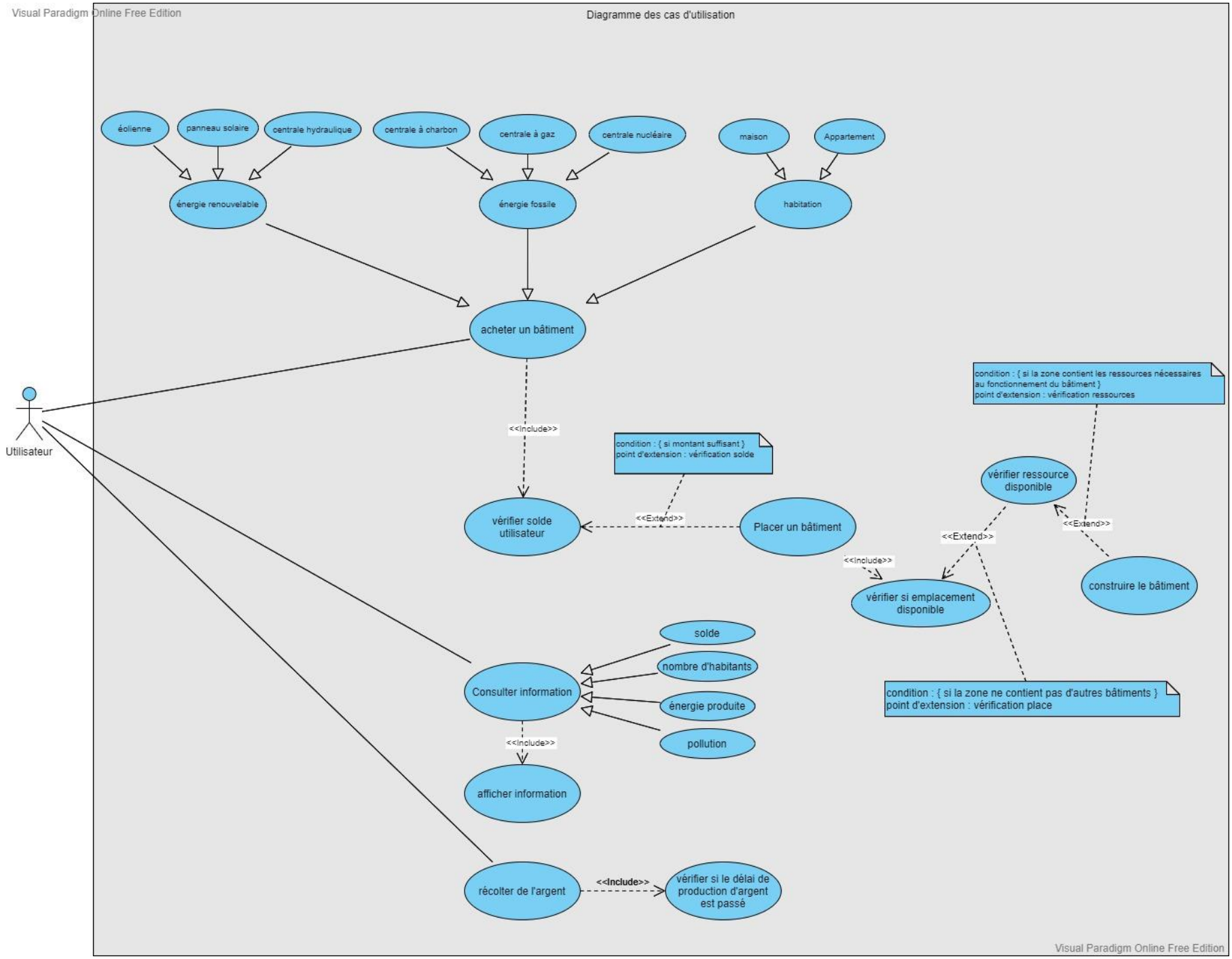
3.3 Le design pattern Observer

Nous avons choisis d'implémenter le design pattern Observer entre la vue et le model car celui-ci permet de gérer facilement les interactions et l'échange de données entre la vue et le modèle. Ce pattern permet de limiter le couplage entre la partie model et vue tout en respectant l'architecture MVC.

4 Cas d'utilisation et scénarii

4.1 Diagramme des cas d'utilisation

Visual Paradigm Online Free Edition



4.2 Scénarii

Nom : acheter bâtiment
Auteur : Utilisateur
Données d'entrée :
Le cas d'utilisation commence lorsque le client clique sur le bouton "Menu"

Scénario principal :
1. Le système propose le type de bâtiment à acheter dans les différentes sections
2. L'utilisateur choisit le type de bâtiment à acheter en cliquant sur un des onglets
3. L'utilisateur choisit un bâtiment en cliquant dessus
4. Le système vérifie si l'utilisateur est solvable en fonction du bâtiment choisit
5. Si le solde est suffisant, l'utilisateur choisit un emplacement pour le bâtiment
6. Le système vérifie si la place disponible de l'emplacement 7. Si l'emplacement est libre, le système vérifie la ressource disponible de l'emplacement en fonction du type de bâtiment choisi
8. Si la ressource est disponible, le bâtiment est placé sur l'emplacement

Cas particulier :
4a. Le solde est insuffisant, impossible d'acheter le bâtiment choisi, retour à l'étape 3
6a. L'emplacement n'est pas libre, il faut en choisir un autre, retour à l'étape 5
7a. La ressource en accord avec le bâtiment n'est pas disponible, impossible de posé ce bâtiment dans cette emplacement, retour à l'étape 5

5 Diagramme de classe

Nous avons divisé nos différentes classes dans différents package toujours en respectant l'architecture MVC. Il y a donc 3 packages principales : model, view et controller. Il y a aussi un package ressource qui contient toutes les ressourcse graphiques nécessaires commes les images, et un package pattern qui contient les classes et interfaces nécessaires au design pattern Observer.

5.1 Le modèle

Le modèle contient toutes les classes permettant d'implémenter la logique de jeux et les données du jeux. Ce package s'articule autour de 3 classes principales avec un niveau d'abstraction plus ou moins élevé.

Tout d'abord la classe Map est la classe principale du modèle. Celle-ci modélise la carte sur laquelle évolue notre jeu. Cette classe possède toutes les données globales du jeu comme par exemple l'énergie totale produite ou le nombre d'habitants etc...

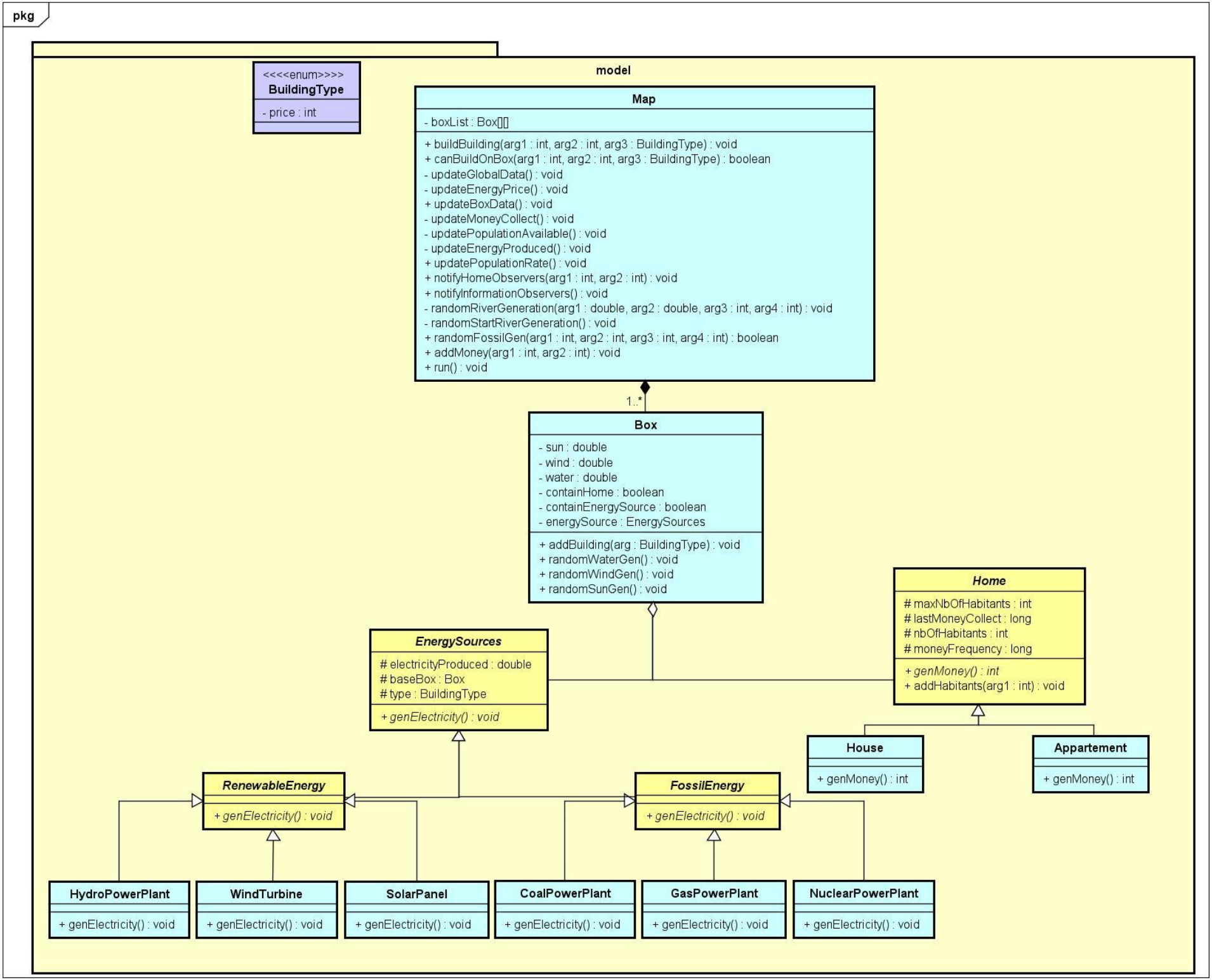
Ensuite la classe Map est composée d'un tableau à deux dimensions de type Box. La Map de notre jeu est divisée en cases. La classe Box modélise donc ces cases. Comme la valeur des caractéristiques dynamiques ou statiques de la Map ne sont pas égales partout, chaque objet de type Box possèdent donc des attributs contenant chacune des valeurs différentes de ces caractéristiques.

De plus, chaque Box est composée de 0 ou 1 bâtiment. C'est pourquoi la classe Box possède un attribut de type EnergySource et un de type Home. Cependant seulement un des deux peut être instancié.

Enfin la classe Building et ses classes dérivés vont permettre de modéliser les bâtiments du jeux contenant ainsi leurs caractéristiques et les méthodes nécessaires à leur fonctionnement. Il y a deux principales sous classes de la classe Building qui sont la classe Home, qui représente les bâtiments pouvant accueillir des habitants, et la classe EnergySource qui représente tous les bâtiments générant de l'énergie.

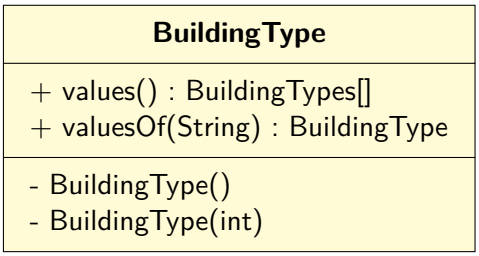
Nous avons créé ces deux sous classes car leur fonctionnement sont très différents. En effet les classes dérivant de Home possèdent des méthodes permettant de récolter de l'argent en fonction de leur nombre d'habitants tandis que les classes dérivés de EnergySource possèdent des méthodes permettant de générer de l'énergie. Nous avons dérivé la classe EnergySource en deux sous classe car l'une (la classe RenewableEnergy) produit de l'énergie en fonction des caractéristiques dynamiques de la Box dans laquelle elle est, alors que l'autre (FossilEnergy) non.

Vous trouverez le diagramme des classes complet en annexe de ce rapport.


pkg
model
<<<<enum>>>>
BuildingType
- price : int
Map
- boxList : Box[][]
+ buildBuilding(arg1 : int, arg2 : int, arg3 : BuildingType) : void
+ canBuildOnBox(arg1 : int, arg2 : int, arg3 : BuildingType) : boolean
- updateGlobalData() : void
- updateEnergyPrice() : void
+ updateBoxData() : void
- updateMoneyCollect() : void
- updatePopulationAvailable() : void
- updateEnergyProduced() : void
+ updatePopulationRate() : void
+ notifyHomeObservers(arg1 : int, arg2 : int) : void
+ notifyInformationObservers() : void
- randomRiverGeneration(arg1 : double, arg2 : double, arg3 : int, arg4 : int) : void
- randomStartRiverGeneration() : void
+ randomFossilGen(arg1 : int, arg2 : int, arg3 : int, arg4 : int) : boolean
+ addMoney(arg1 : int, arg2 : int) : void
+ run() : void
1..*
Box
- sun : double
- wind : double
- water : double
- containHome : boolean
- containEnergySource : boolean
- energySource : EnergySources
+ addBuilding(arg : BuildingType) : void
+ randomWaterGen() : void
+ randomWindGen() : void
+ randomSunGen() : void
Home
maxNbOfHabitants : int
lastMoneyCollect : long
nbOfHabitants : int
moneyFrequency : long
+ genMoney() : int
+ addHabitants(arg1 : int) : void
EnergySources
electricityProduced : double
baseBox : Box
type : BuildingType
+ genElectricity() : void
House
+ genMoney() : int
Appartement
+ genMoney() : int
RenewableEnergy
+ genElectricity() : void
FossilEnergy
+ genElectricity() : void
HydroPowerPlant
+ genElectricity() : void
WindTurbine
+ genElectricity() : void
SolarPanel
+ genElectricity() : void
CoalPowerPlant
+ genElectricity() : void
GasPowerPlant
+ genElectricity() : void
NuclearPowerPlant
+ genElectricity() : void

5.1.1 Les types de bâtiments

Afin de faciliter la généralisation des bâtiments, nous avons créer une classe BuildingType qui se compose ainsi :

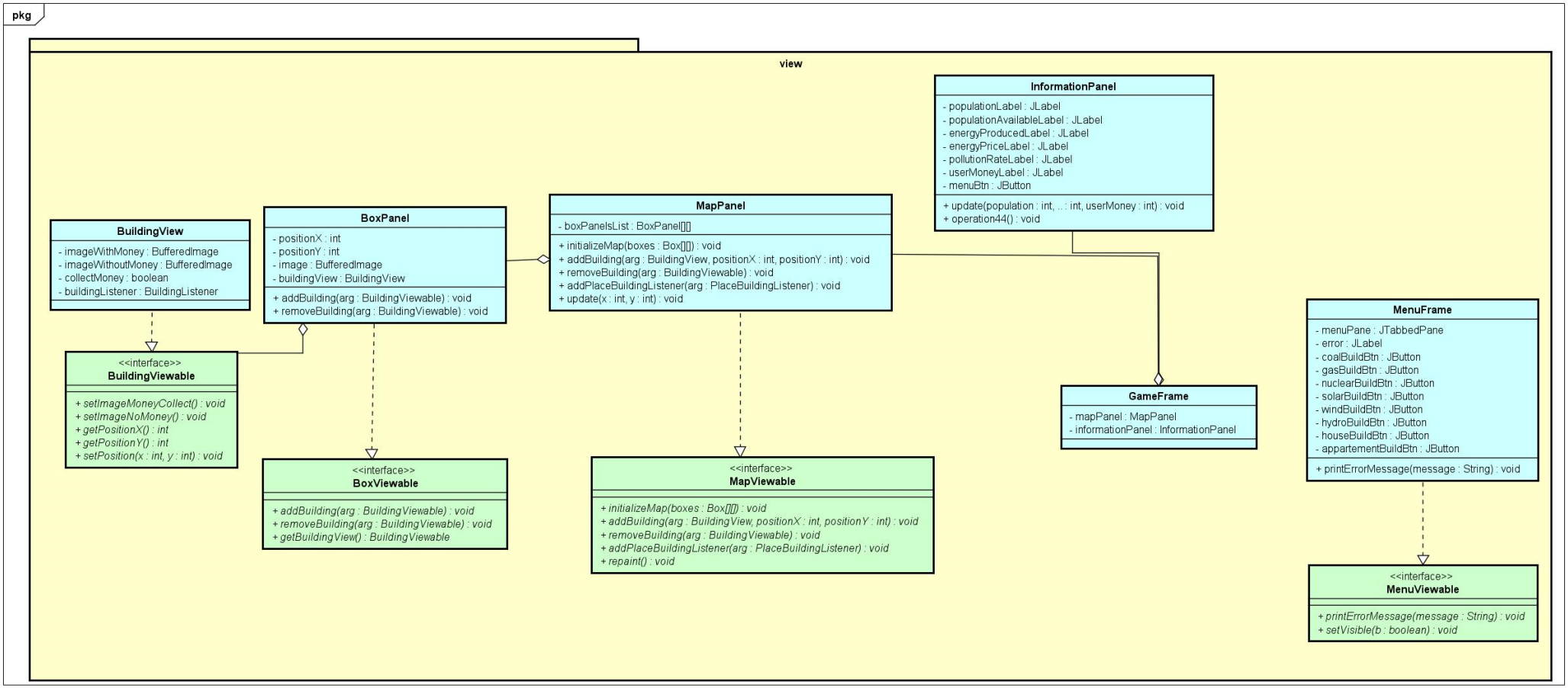


5.2 La vue

Nous avons choisir d'utiliser la bibliothèque graphique Swing pour implémenter la vue de notre programme. Cela a donc influencé nos choix de conception. Nous avons essayé au maximum de concevoir une vue modulaire et d'appliquer des bonnes pratiques de conception afin de favoriser la réutilisabilité et l'amélioration du programme d'un point de vue graphique.

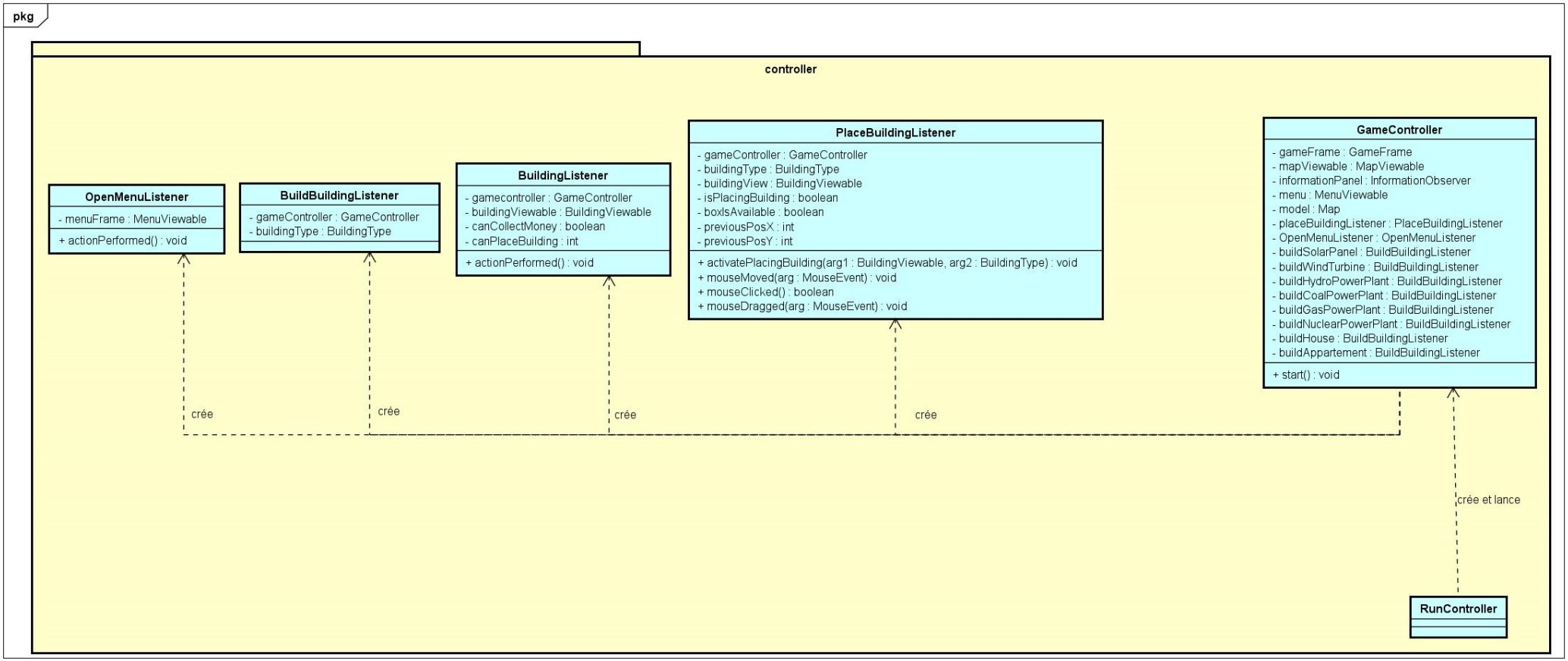
Pour cela nous avons essayé de limiter le couplage entre les différentes classes. C'est pourquoi la classe GameFrame n'instancie pas elle même les classes InformationPanel et MapPanel. Ces deux classes sont instanciées dans le controller et passées en paramètre dans le constructeur de GameFrame. C'est une injection de dépendance. De plus nous avons mis en pratique un principe qui est l'inversion de dépendance. Ce principe a aussi pour but de limiter le couplage et les dépendances des différentes classes entre elles, surtout lorsqu'on utilise l'agrégation ou la composition. Ce principe d'inversion de dépendance dit qu'il faut éviter d'avoir une dépendance directe entre les classes, et plutôt passer par une couche d'abstraction. Ici, cette couche d'abstraction sont les interfaces InformationsObserver, HomeObserver et toutes les interfaces présente dans le package view. Ainsi si, dans le futur, nous voudrons changer radicalement notre vue et utiliser par exemple JavaFX comme bibliothèque graphique, nous auront juste à implémenter ces interfaces et la vue sera fonctionnelle et il y aura très peu de modifications à apporter au controller. Nous essayons au maximum d'appliquer ces principes dans notre programme lors de notre conception.

Ensuite, les différentes classes de la vue suivent un peu près la même logique que dans le modèle c'est à dire qu'il a un MapPanel qui contient des BoxPanel qui permet d'afficher chaque case indépendamment. Ces cases peuvent contenir ou non la vue d'un bâtiment.

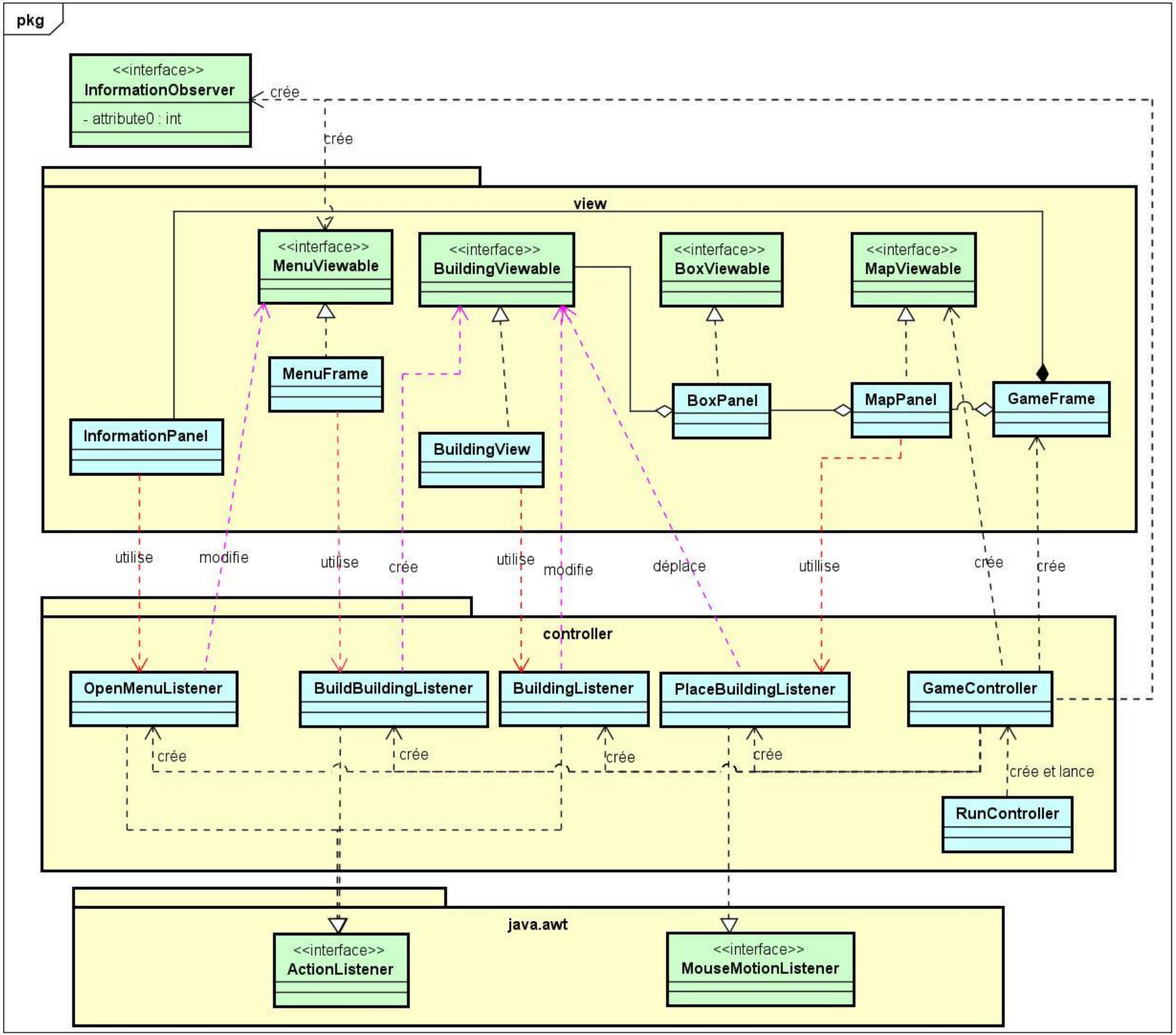

pkg
view
InformationPanel
- populationLabel : JLabel
- populationAvailableLabel : JLabel
- energyProducedLabel : JLabel
- energyPriceLabel : JLabel
- pollutionRateLabel : JLabel
- userMoneyLabel : JLabel
- menuBtn : JButton
+ update(population : int, .. : int, userMoney : int) : void
+ operation44() : void
BuildingView
- imageWithMoney : BufferedImage
- imageWithoutMoney : BufferedImage
- collectMoney : boolean
- buildingListener : BuildingListener
BoxPanel
- positionX : int
- positionY : int
- image : BufferedImage
- buildingView : BuildingView
+ addBuilding(arg : BuildingViewable) : void
+ removeBuilding(arg : BuildingViewable) : void
MapPanel
- boxPanelsList : BoxPanel[][]
+ initializeMap(boxes : Box[][]) : void
+ addBuilding(arg : BuildingView, positionX : int, positionY : int) : void
+ removeBuilding(arg : BuildingViewable) : void
+ addPlaceBuildingListener(arg : PlaceBuildingListener) : void
+ update(x : int, y : int) : void
<<interface>>
BuildingViewable
+ setImageMoneyCollect() : void
+ setImageNoMoney() : void
+ getPositionX() : int
+ getPositionY() : int
+ setPosition(x : int, y : int) : void
<<interface>>
BoxViewable
+ addBuilding(arg : BuildingViewable) : void
+ removeBuilding(arg : BuildingViewable) : void
+ getBuildingView() : BuildingViewable
<<interface>>
MapViewable
+ initializeMap(boxes : Box[][]) : void
+ addBuilding(arg : BuildingView, positionX : int, positionY : int) : void
+ removeBuilding(arg : BuildingViewable) : void
+ addPlaceBuildingListener(arg : PlaceBuildingListener) : void
+ repaint() : void
GameFrame
- mapPanel : MapPanel
- informationPanel : InformationPanel
MenuFrame
- menuPane : JTabbedPane
- error : JLabel
- coalBuildBtn : JButton
- gasBuildBtn : JButton
- nuclearBuildBtn : JButton
- solarBuildBtn : JButton
- windBuildBtn : JButton
- hydroBuildBtn : JButton
- houseBuildBtn : JButton
- appartementBuildBtn : JButton
+ printErrorMessage(message : String) : void
<<interface>>
MenuViewable
+ printErrorMessage(message : String) : void
+ setVisible(b : boolean) : void

5.3 Le controller

Le controller est celui qui fait le lien entre la partie graphique et la partie logique. C'est pour cela qu'il contient le modèle et la plupart des éléments de la vue. Cela lui permet en plus de mieux les manipuler. Pour pouvoir recevoir et traiter les évènements créés par l'utilisateur, le package Swing nous oblige à créer des classes qui implémentent des interfaces de listener telles que ActionListener ou MouseMotionListener. Il faut ensuite ajouter une instance de ces classes au composants graphiques que l'on souhaite. Une pratique courante en Java est d'implémenter cette interface de Listener "à la volée" grâce à une classe interne anonyme. Cependant il est préférable, plus propre et clair de créer notre propre classe qui implémente une de ces interfaces. C'est ce que nous avons fait avec les classes suivantes. BuildBuildingListener qui peut être ajoutée à un JButton, pour pouvoir acheter un bâtiment dans le menu par exemple. PlaceBuildingListener qui permet à l'utilisateur de placer un bâtiment en bougeant sa souris. BuildingListener qui permet de cliquer sur un bâtiment pour le placer ou pour récolter de l'argent. Nous pouvons voir que les classes du controller ne manipule aucune classe de la vue directement à part GameFrame. Les classes du controller instancient et manipulent uniquement des variables dont le type est soit une classe du model ou du controller, soit une interface de la vue ou du package pattern. Ainsi, s'y on en vient à modifier la vue, il n'y aura que très peu de modification à apporter aux classes du controller.


pkg
controller
OpenMenuListener
- menuFrame : MenuViewable
+ actionPerformed() : void
BuildBuildingListener
- gameController : GameController
- buildingType : BuildingType
BuildingListener
- gamecontroller : GameController
- buildingViewable : BuildingViewable
- canCollectMoney : boolean
- canPlaceBuilding : int
+ actionPerformed() : void
PlaceBuildingListener
- gameController : GameController
- buildingType : BuildingType
- buildingView : BuildingViewable
- isPlacingBuilding : boolean
- boxIsAvailable : boolean
- previousPosX : int
- previousPosY : int
+ activatePlacingBuilding(arg1 : BuildingViewable, arg2 : BuildingType) : void
+ mouseMoved(arg : MouseEvent) : void
+ mouseClicked() : boolean
+ mouseDragged(arg : MouseEvent) : void
GameController
- gameFrame : GameFrame
- mapViewable : MapViewable
- informationPanel : InformationObserver
- menu : MenuViewable
- model : Map
- placeBuildingListener : PlaceBuildingListener
- OpenMenuListener : OpenMenuListener
- buildSolarPanel : BuildBuildingListener
- buildWindTurbine : BuildBuildingListener
- buildHydroPowerPlant : BuildBuildingListener
- buildCoalPowerPlant : BuildBuildingListener
- buildGasPowerPlant : BuildBuildingListener
- buildNuclearPowerPlant : BuildBuildingListener
- buildHouse : BuildBuildingListener
- buildAppartement : BuildBuildingListener
+ start() : void
crée
crée
crée
crée
crée et lance
RunController

Le diagramme de classe plus détaillé ci-dessus met en évidence les relations entre le controller et la vue, et notamment les associations entre les classe Listeners et les Interfaces de la vue. On remarque qu'il n'y a pas d'association entre les classes du controller et les classes de la vue à part avec GameFrame. Cela permet une bonne modularité et amélioration du code.

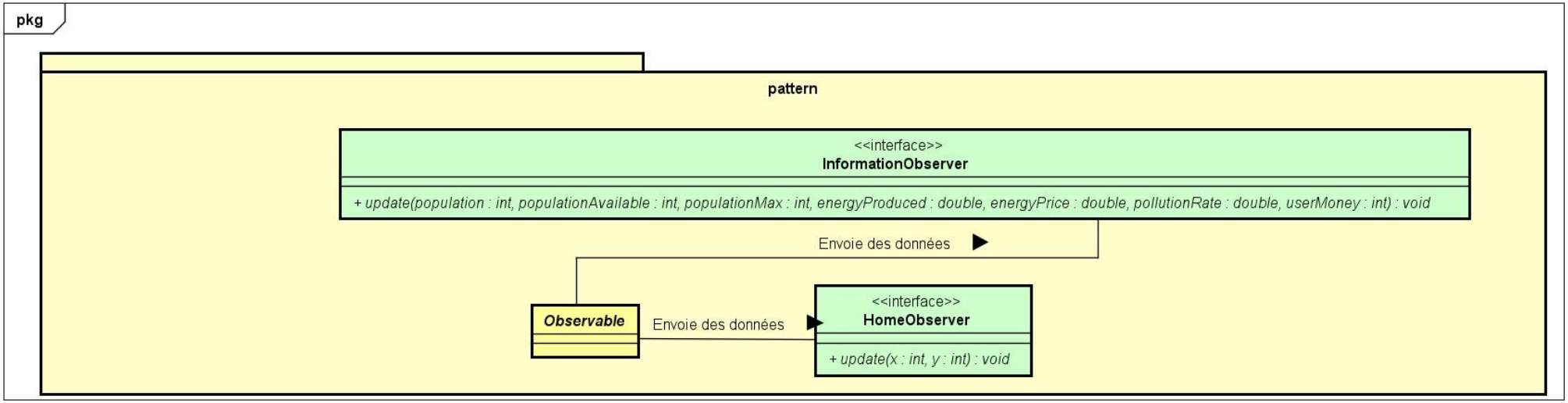

pkg
<<interface>>
InformationObserver
- attribute0 : int
crée
crée
view
<<interface>>
MenuViewable
<<interface>>
BuildingViewable
<<interface>>
BoxViewable
<<interface>>
MapViewable
MenuFrame
BoxPanel
MapPanel
GameFrame
InformationPanel
BuildingView
utilise
modifie
utilise
crée
utilise
modifie
déplace
utillise
crée
crée
controller
OpenMenuListener
BuildBuildingListener
BuildingListener
PlaceBuildingListener
GameController
crée
crée
crée
crée
crée et lance
RunController
java.awt
<<interface>>
ActionListener
<<interface>>
MouseMotionListener

5.4 Le Design Pattern Observer

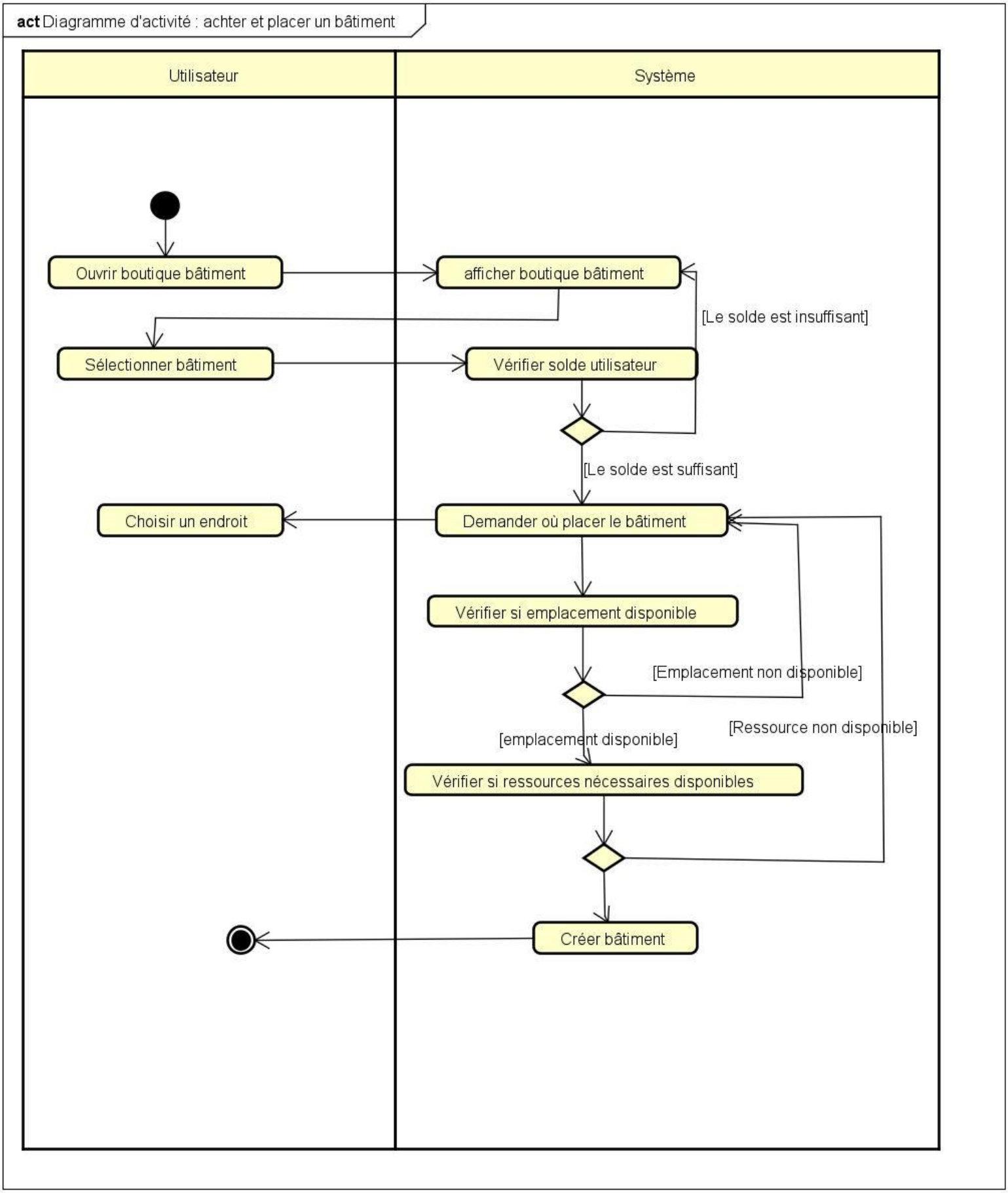
Nous avons choisis d'utiliser un design pattern Observer pour pouvoir notifier et transmettre les données nécessaires à la vue quand celles-ci ont changé dans le modèle. Le modèle, en l'occurrence la Map dans notre projet, hérite de la classe abstraite Observable.

Nous avons choisit d'utiliser une classe abstraite Observable et non une interface pour qu'elle puisse contenir un attribut qui est une liste qui contient tous les observers qui veulent être notifiés par le modèle. Cette classe possède aussi deux méthodes pour enregistrer et supprimer des observers. Cette classe possède aussi deux méthodes abstraites notifyInformationObservers() et notifyHomeObervers() qu'il est nécessaire d'override et qui permet d'appeler la méthode update() de tous les observers enregistrés dans la liste.

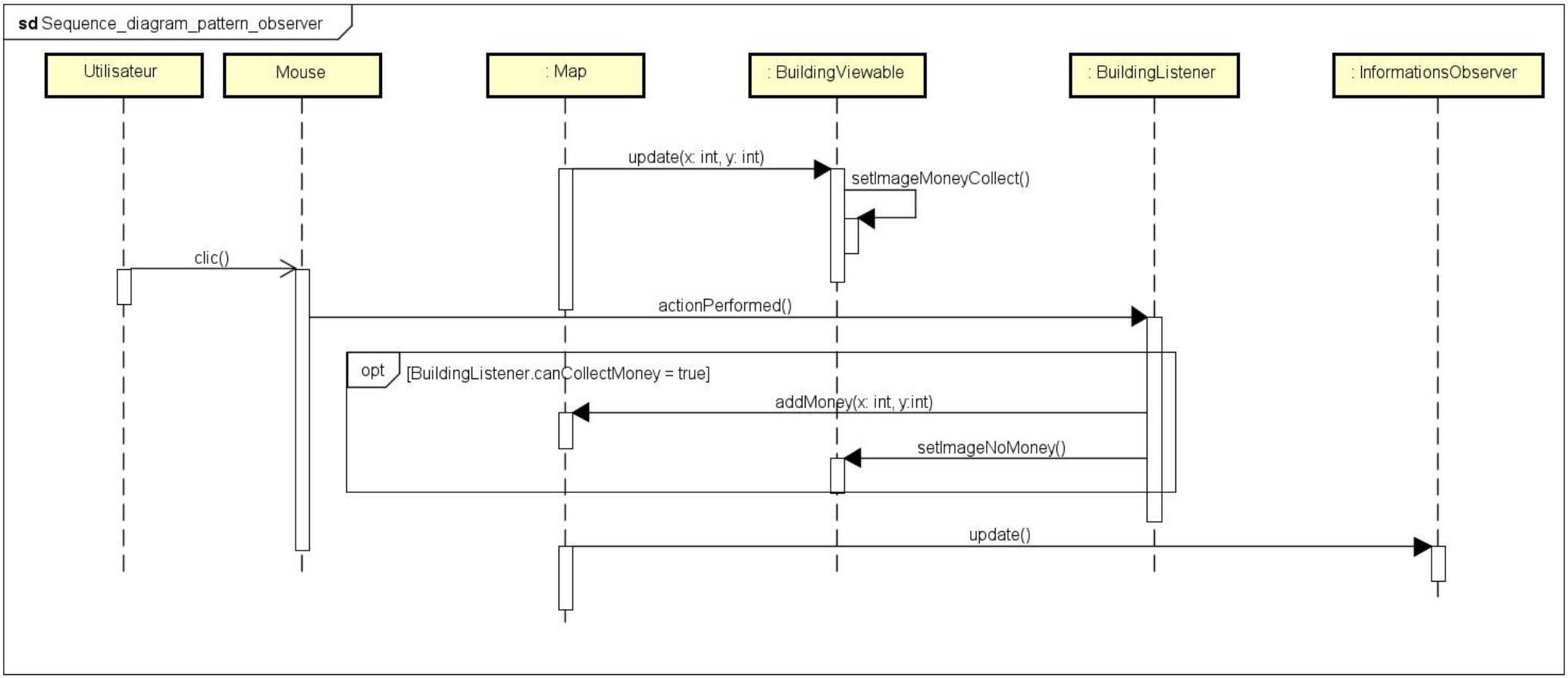
Nous avons choisis de concevoir deux interfaces différentes pour transmettre séparémment les données correspondant aux donnée générales de la Map et les données correspondant à l'état de la collecte d'argent des habitations. En effet ces deux catégories de données ne sont pas du tout mise à jour au même moment. Cela permet donc de limiter l'échange de données. De plus, ces deux catégories de données ne sont pas affichées par les même composants graphiques. Nous avons donc préféré de créer deux interfaces différentes.

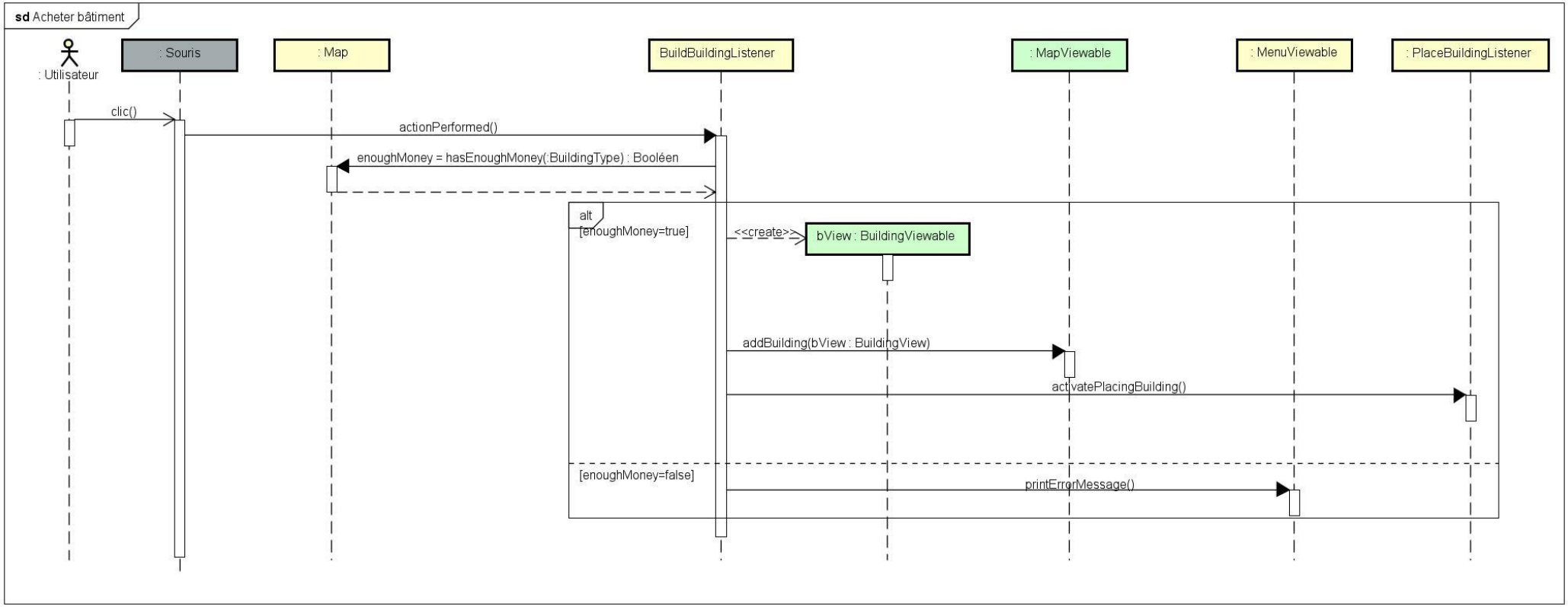

pkg
pattern
<<interface>>
InformationObserver
+ update(population : int, populationAvailable : int, populationMax : int, energyProduced : double, energyPrice : double, pollutionRate : double, userMoney : int) : void
Envoie des données
Observable
Envoie des données
<<interface>>
HomeObserver
+ update(x : int, y : int) : void

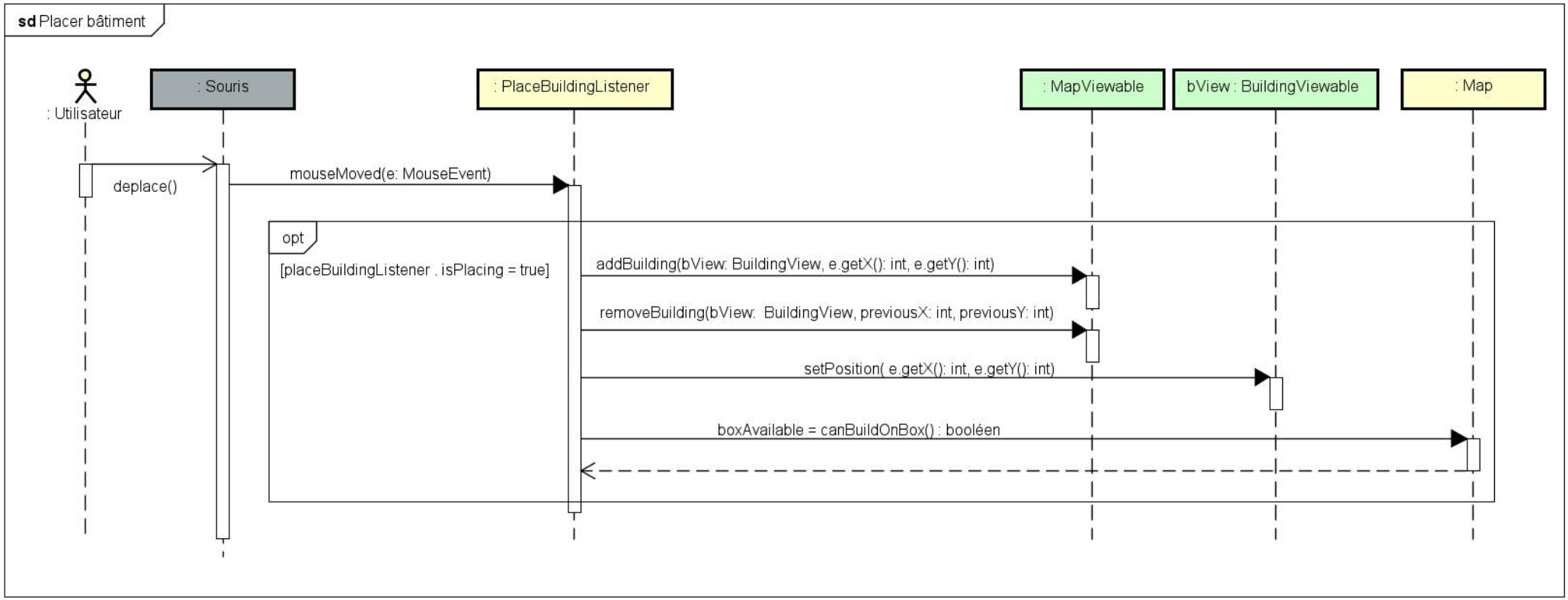
6 Diagramme d'activité

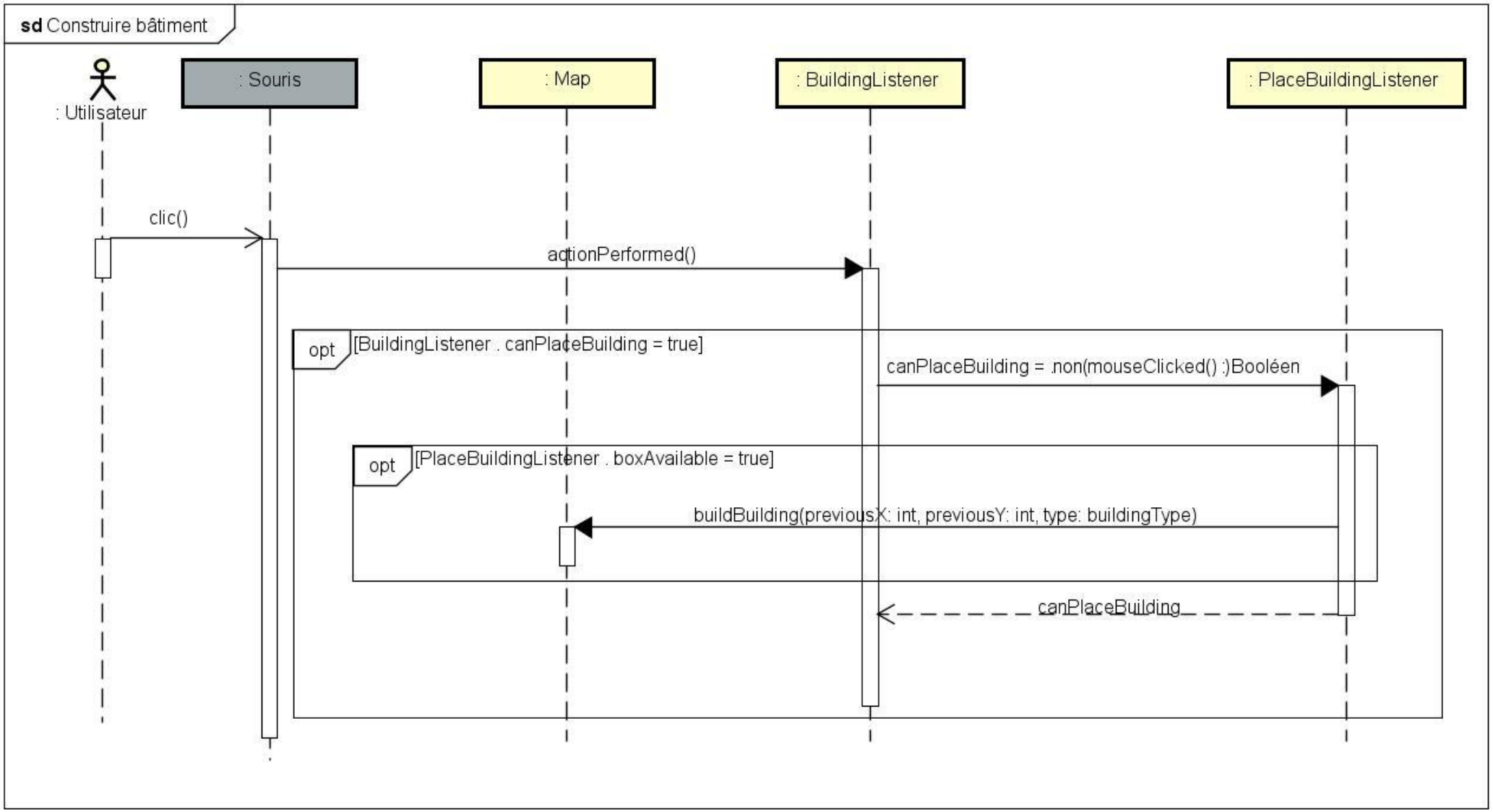

act Diagramme d'activité : achter et placer un bâtiment
Utilisateur
Système
Ouvrir boutique bâtiment
afficher boutique bâtiment
[Le solde est insuffisant]
Sélectionner bâtiment
Vérifier solde utilisateur
[Le solde est suffisant]
Choisir un endroit
Demander où placer le bâtiment
Vérifier si emplacement disponible
[Emplacement non disponible]
[Ressource non disponible]
[emplacement disponible]
Vérifier si ressources nécessaires disponibles
Créer bâtiment

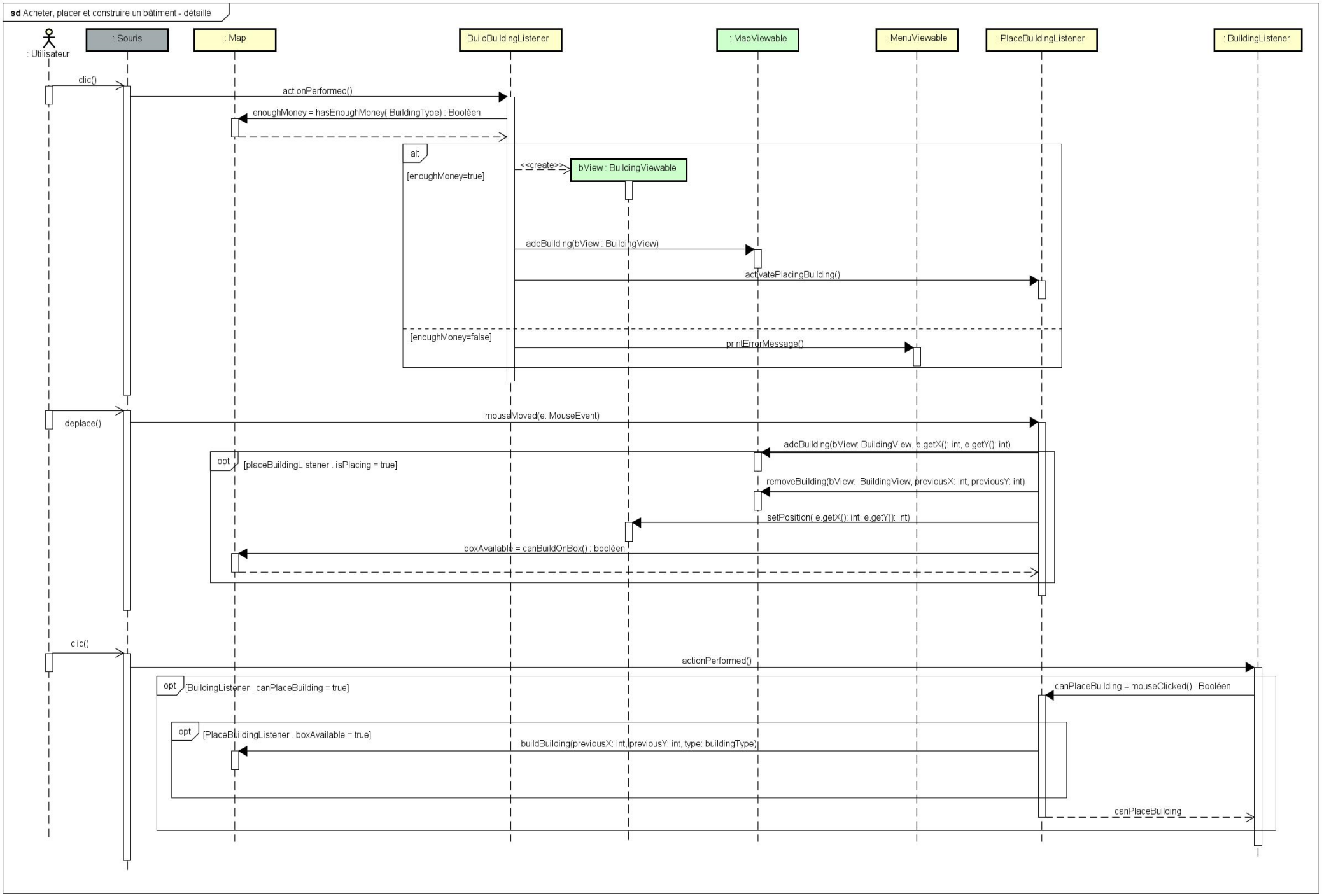
7 Diagrammes de séquence


sd Sequence_diagram_pattern_observer
Utilisateur
Mouse
: Map
: BuildingViewable
: BuildingListener
: InformationsObserver
update(x: int, y: int)
setImageMoneyCollect()
clic()
actionPerformed()
opt
[BuildingListener.canCollectMoney = true]
addMoney(x: int, y:int)
setImageNoMoney()
update()

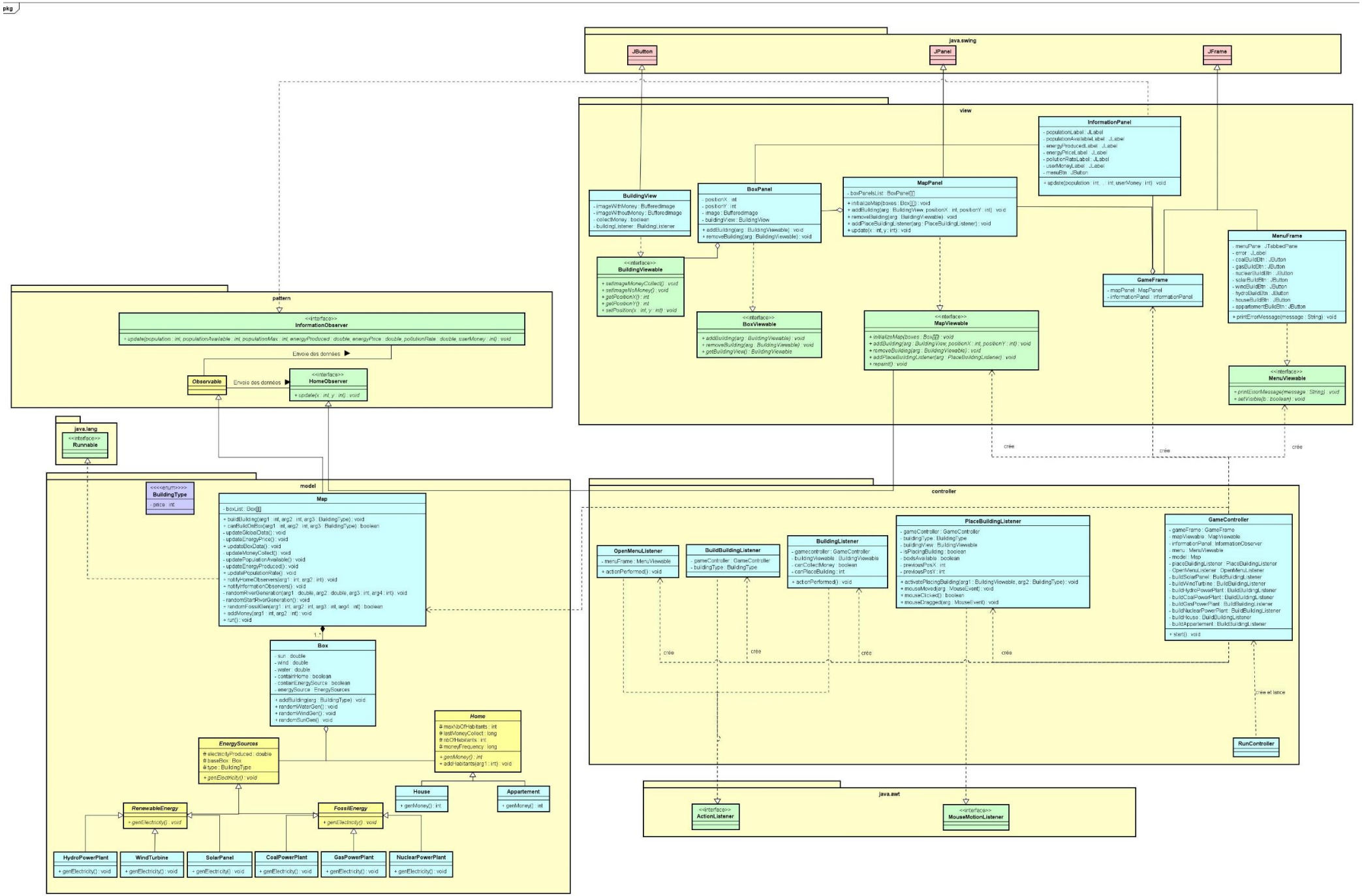

sd Acheter bâtiment
: Utilisateur
: Souris
: Map
BuildBuildingListener
: MapViewable
: MenuViewable
: PlaceBuildingListener
clic()
actionPerformed()
enoughMoney = hasEnoughMoney(:BuildingType) : Booléen
alt
[enoughMoney=true]
<<create>>
bView : BuildingViewable
addBuilding(bView : BuildingView)
activatePlacingBuilding()
[enoughMoney=false]
printErrorMessage()


sd Placer bâtiment
: Utilisateur
: Souris
: PlaceBuildingListener
: MapViewable
bView : BuildingViewable
: Map
deplace()
mouseMoved(e: MouseEvent)
opt
[placeBuildingListener . isPlacing = true]
addBuilding(bView: BuildingView, e.getX(): int, e.getY(): int)
removeBuilding(bView: BuildingView, previousX: int, previousY: int)
setPosition(e.getX(): int, e.getY(): int)
boxAvailable = canBuildOnBox() : booléen


sd Construire bâtiment
: Utilisateur
: Souris
: Map
: BuildingListener
: PlaceBuildingListener
clic()
actionPerformed()
opt
[BuildingListener . canPlaceBuilding = true]
canPlaceBuilding = .non(mouseClicked() :)Booléen
opt
[PlaceBuildingListener . boxAvailable = true]
buildBuilding(previousX: int, previousY: int, type: buildingType)
canPlaceBuilding


sd Acheter, placer et construire un bâtiment - détaillé
: Utilisateur
: Souris
: Map
BuildBuildingListener
: MapViewable
: MenuViewable
: PlaceBuildingListener
: BuildingListener
clic()
actionPerformed()
enoughMoney = hasEnoughMoney(:BuildingType) : Booléen
alt
[enoughMoney=true]
<<create>>
bView : BuildingViewable
addBuilding(bView : BuildingView)
activatePlacingBuilding()
[enoughMoney=false]
printErrorMessage()
deplace()
mouseMoved(e: MouseEvent)
addBuilding(bView: BuildingView, e.getX(): int, e.getY(): int)
opt
[placeBuildingListener . isPlacing = true]
removeBuilding(bView: BuildingView, previousX: int, previousY: int)
setPosition(e.getX(): int, e.getY(): int)
boxAvailable = canBuildOnBox() : booléen
clic()
actionPerformed()
opt
[BuildingListener . canPlaceBuilding = true]
canPlaceBuilding = mouseClicked() : Booléen
opt
[PlaceBuildingListener . boxAvailable = true]
buildBuilding(previousX: int, previousY: int, type: buildingType)
canPlaceBuilding

8 Annexe


pkg
java.swing
JButton
JPanel
JFrame
view
InformationPanel
- populationLabel : JLabel
- populationAvailableLabel : JLabel
- energyProducedLabel : JLabel
- energyPriceLabel : JLabel
- pollutionRateLabel : JLabel
- userMoneyLabel : JLabel
- menuBtn : JButton
+ update(population : int, : , : int, userMoney : int) : void
BuildingView
- imageWithMoney : BufferedImage
- imageWithoutMoney : BufferedImage
- collectMoney : boolean
- buildingListener : BuildingListener
BoxPanel
- positionX : int
- positionY : int
- image : BufferedImage
- buildingView : BuildingView
+ addBuilding(arg : BuildingViewable) : void
+ removeBuilding(arg : BuildingViewable) : void
MapPanel
- boxPanelsList : BoxPanel[][]
+ initializeMap(boxes : Box[][]) : void
+ addBuilding(arg : BuildingView, positionX : int, positionY : int) : void
+ removeBuilding(arg : BuildingViewable) : void
+ addPlaceBuildingListener(arg : PlaceBuildingListener) : void
+ update(x : int, y : int) : void
<<interface>>
BuildingViewable
+ setImageMoneyCollect() : void
+ setImageNoMoney() : void
+ getPositionX() : int
+ getPositionY() : int
+ setPosition(x : int, y : int) : void
<<interface>>
BoxViewable
+ addBuilding(arg : BuildingViewable) : void
+ removeBuilding(arg : BuildingViewable) : void
+ getBuildingView() : BuildingViewable
<<interface>>
MapViewable
+ initializeMap(boxes : Box[][]) : void
+ addBuilding(arg : BuildingView, positionX : int, positionY : int) : void
+ removeBuilding(arg : BuildingViewable) : void
+ addPlaceBuildingListener(arg : PlaceBuildingListener) : void
+ repaint() : void
GameFrame
- mapPanel : MapPanel
- informationPanel : informationPanel
MenuFrame
- menuPane : JTabbedPane
- error : JLabel
- coalBuildBtn : JButton
- gasBuildBtn : JButton
- nuclearBuildBtn : JButton
- solarBuildBtn : JButton
- windBuildBtn : JButton
- hydroBuildBtn : JButton
- houseBuildBtn : JButton
- appartementBuildBtn : JButton
+ printErrorMessage(message : String) : void
<<interface>>
MenuViewable
+ printErrorMessage(message : String) : void
+ setVisible(b : boolean) : void
crée
crée
crée
pattern
<<interface>>
InformationObserver
+ update(population : int, populationAvailable : int, populationMax : int, energyProduced : double, energyPrice : double, pollutionRate : double, userMoney : int) : void
Envoie des données
Observable
Envoie des données
<<interface>>
HomeObserver
+ update(x : int, y : int) : void
java.lang
<<interface>>
Runnable
model
<<<<enum>>>>
BuildingType
- price : int
Map
- boxList : Box[][]
+ buildBuilding(arg1 : int, arg2 : int, arg3 : BuildingType) : void
- canBuildOnBox(arg1 : int, arg2 : int, arg3 : BuildingType) : boolean
- updateGlobalData() : void
- updateEnergyPrice() : void
+ updateBoxData() : void
- updateMoneyCollect() : void
- updatePopulationAvailable() : void
- updateEnergyProduced() : void
+ updatePopulationRate() : void
+ notifyHomeObservers(arg1 : int, arg2 : int) : void
+ notifyInformationObservers() : void
- randomRiverGeneration(arg1 : double, arg2 : double, arg3 : int, arg4 : int) : void
- randomStartRiverGeneration() : void
+ randomFossilGen(arg1 : int, arg2 : int, arg3 : int, arg4 : int) : boolean
+ addMoney(arg1 : int, arg2 : int) : void
+ run() : void
1
*
Box
- sun : double
- wind : double
- water : double
- containHome : boolean
- containEnergySource : boolean
- energySource : EnergySources
+ addBuilding(arg : BuildingType) : void
+ randomWaterGen() : void
+ randomWindGen() : void
+ randomSunGen() : void
EnergySources
electricityProduced : double
baseBox : Box
type : BuildingType
+ genElectricity() : void
Home
maxNbOfHabitants : int
lastMoneyCollect : long
nbOfHabitants : int
moneyFrequency : long
+ genMoney() : int
+ addHabitants(arg1 : int) : void
House
+ genMoney() : int
Appartement
+ genMoney() : int
RenewableEnergy
+ genElectricity() : void
FossilEnergy
+ genElectricity() : void
HydroPowerPlant
+ genElectricity() : void
WindTurbine
+ genElectricity() : void
SolarPanel
+ genElectricity() : void
CoalPowerPlant
+ genElectricity() : void
GasPowerPlant
+ genElectricity() : void
NuclearPowerPlant
+ genElectricity() : void
controller
OpenMenuListener
- menuFrame : MenuViewable
+ actionPerformed() : void
BuildBuildingListener
- gameController : GameController
- buildingType : BuildingType
BuildingListener
- gamecontroller : GameController
- buildingViewable : BuildingViewable
- canCollectMoney : boolean
- canPlaceBuilding : int
+ actionPerformed() : void
PlaceBuildingListener
- gameController : GameController
- buildingType : BuildingType
- buildingView : BuildingViewable
- isPlacingBuilding : boolean
- boxIsAvailable : boolean
- previousPosX : int
- previousPosY : int
+ activatePlacingBuilding(arg1 : BuildingViewable, arg2 : BuildingType) : void
+ mouseMoved(arg : MouseEvent) : void
+ mouseClicked() : boolean
+ mouseDragged(arg : MouseEvent) : void
GameController
- gameFrame : GameFrame
- mapViewable : MapViewable
- informationPanel : InformationObserver
- menu : MenuViewable
- model : Map
- placeBuildingListener : PlaceBuildingListener
- OpenMenuListener : OpenMenuListener
- buildSolarPanel : BuildBuildingListener
- buildWindTurbine : BuildBuildingListener
- buildHydroPowerPlant : BuildBuildingListener
- buildCoalPowerPlant : BuildBuildingListener
- buildGasPowerPlant : BuildBuildingListener
- buildNuclearPowerPlant : BuildBuildingListener
- buildHouse : BuildBuildingListener
- buildAppartement : BuildBuildingListener
+ start() : void
crée
crée
crée
crée
crée et lance
RunController
java.awt
<<interface>>
ActionListener
<<interface>>
MouseMotionListener