Projet de programmation

Année 2020 - 2021

M2107 – Projet de programmation

Bilan – Rendu final

IUT Vannes
8 Rue Montaigne – BP 561 – 56017 VANNES Cedex
02 97 62 64 64 – **www.iutvannes.fr**

Table des matières

I) Mise à jour des diagrammes

Concernant les diagrammes de séquence, on ne dénote aucun changement. L'interface graphique a été créée en adéquation avec ces diagrammes, il n'y a donc aucun changement notable au niveau des interactions utilisateur / logiciel.

Du point de vue des packages, deux packages supplémentaires sont présents : le package « test », content les différentes classes dédiées aux tests unitaires Junit (la campagne de tests sera évoquée plus tard dans ce rendu), et le package « control », dédié à la gestion des évènements de l'interface graphique du jeu.

Dans le package « model », il n'y a aucun changement au niveau du nombre de classes. En revanche, pour des **raisons algorithmiques** et dans l'objectif d'avoir des classes les plus fonctionnelles possible, un certain nombre de méthodes & d'attributs supplémentaires, **n'ayant été anticipés** lors de la phase d'analyse et de conception (à cause de la difficulté que représentait la projection des fonctionnalités sous forme de code) sont présents.

A) Game

La classe Game s'est vu ajoutée plusieurs attributs : turn, stackWorkers, stackBuidling & lesBat.

On retrouve les getter, setter et méthodes d'initialisation associés.

B) Card

La classe Card s'est vu ajoutée deux attributs : name et victPoints.

On retrouve les getter et setter associés.

C) Machine

La classe Machine s'est vu ajouté l'attribut workingStatus (les objets de cette classe ayant à la fois le comportement d'un Batiment puis d'un Ouvrier, il faut dissocier le statut « bâtiment fini » à celui de « en travail »), ainsi que l'attribut victPoints. On y retrouve les getter et / ou setter associés.

D) Ouvrier

La classe Ouvrier s'est vu rajouté la méthode setNullWorkOn(), qui est utilisée lorque le Batiment / la Machine sur le/laquel(le) l'objet Ouvrier travaille est fini.

E) Player

La classe Player s'est vu ajouté les attributs vPoints, counterSend, workers, machines, leJeu & joueSur. On y retrouve les getter et / ou setter associés.

E.1) HumanPlayer

La classe HumanPlayer s'est vu rajoutée une méthode loadScanner(), utilisée lors de la restauration d'un object Game, les objets Scanner ne pouvant implémenter l'interface Serializable.

F) IGame

L'interface IGame surcharge la méthode endGame(), en ajoutant une méthode renvoyant un HashMap permettant de faire correspondre un jouer à un nombre de points pour l'affichage du classement.

Toutes les classes **contiennent une méthode toString()**, servant à l'affichage du statut des différents objets instanciés dans la version console du jeu.

Dans le package « view », on notera que la classe GraphicalDisplay **n'est plus sous-type** de Display, pour la simple raison que GraphicalDisplay doit hériter de la super-classe JFrame, et le langage Java ne permet pas l'héritage multiple.

II) Choix techniques et algorithmiques

Les algorithmes les plus « complexes » à réaliser ont été ceux contenu dans certaines des méthodes correspondant aux différentes actions possible pour un joueur.

A) recruit() & openBuild()

```java
/**
 * This method allows a Player object to recruit a new worker displayed on the board.
 * @param ouvrier The Ouvrier object to add to the ArrayList of owned Ouvrier
 * @return true if the action has been correctly done, false otherwise.
 */
public boolean recruit(Ouvrier ouvr){

    boolean ret = false;

    if ( (ouvr != null) && (this.getCoins()-ouvr.getCost() >= 0)){

        this.workers.add(ouvr);
        ret = true;
        this.setCoins(this.getCoins()-ouvr.getCost());

        this.leJeu.getCards().remove(ouvr);
        this.leJeu.getStackWorkers().remove(ouvr);

    }

    else if (this.getCoins()-ouvr.getCost() < 0) System.err.println("recruit(ouvr) - Error : This card is too expensive.");

    else System.err.println("recruit(ouvr) - Error : Ths ouvr parameter is null.");

    return ret;
};
```

L'algorithme gère la présence de l'objet Ouvrier à recruter dans les différentes ArrayList du jeu. En effet, si un Ouvrier est recruté, il doit être ajouté à la liste des ouvriers du joueur, mais aussi retiré de la liste des Ouvriers disponibles et à l'ArrayList des cartes totale.

Le même principe est utilisé pour la méthode openBuild(), dont le code est affiché ci-dessous.

```java
/**
 * This method allows a Player object to open a new construction among the Batiment/Machine displayed on the board.
 * @param bat The building to put under construction.
 * @return true if the action has been correctly done, false otherwise.
 */
public boolean openBuild(Card bat){

    boolean ret = false;

    if ( (bat != null) &&  (bat.getClass() == Batiment.class || bat.getClass() == Machine.class) && (this.getCoins()-bat.getCost() >= 0)){

        this.playCards.add(bat);
        ret = true;
        this.setCoins(this.getCoins()-bat.getCost());

        this.leJeu.getCards().remove(bat);
        this.leJeu.getStackBuilding().remove(bat);

    }

    else if (bat == null) System.err.println("openBuild(bat) - Error : bat parameter is null.");

    else if (bat.getClass() == Ouvrier.class) System.err.println("openBuild(bat) - Error : A worker is not a building");

    else if (bat.getState()) System.err.println("openBuild(bat) - Error : The building is already finished.");

    else if (this.getCoins()-bat.getCost() < 0) System.err.println("openBuild(bat) - Error : This card is too expensive.");

    return ret;
};
```

B) sendToWork()

La première difficulté de cette méthode résidait en le fait qu'un Ouvrier ou une Machine peuvent être envoyé à la fois sur une Machine ou un Bâtiment. Machine n'étant pas sous-type de Batiment, il fallait donc utiliser un paramètre de type Card, même chose pour Machine et Ouvrier.

Une première vérification devait donc être faite : s'assurer du type des deux paramètres.

Une fois la machine/l'ouvrier envoyé sur le Batiment, il faut vérifier si la construction du Batiment/de la Machine peut être finie.

Il faut donc pour ce faire parcourir la liste des ouvriers appartenant au joueur, vérifier si ils travaillent sur le batiment/la machine concernée, et si oui, ajouter les ressources produites à un total.

Si les différents totaux de ressources produites sont suffisants, le Batiment/la Machine est alors finie. Il faut donc ajouter les points de victoire rapportés par la fin de la construction au total de points, et si le bâtiment est une Machine, définir le statut de la Machine comme libre et l'ajouter à la liste des ouvriers disponibles.

Enfin, on reparcourt la liste des Ouvriers/Machines travaillant sur ce chantier, et l'on met leur statut comme libre.

```java
/**
 * This method allows a Player object to send to work a worker.
 * It will link an Ouvrier Card to a Batiment/Machine card.
 * @param ouvr The Ouvrier object to link.
 * @param bat The building on which the Ouvrier object will work.
 * @return true if the action has been correctly done, false otherwise.
 */
public boolean sendToWork(Card ouvr, Card bat){

    boolean ret = false;

    //We check that the parameters are initialized, that the building is under construction, that the worker is not working on an another building, and that the cards are in the current player's li
    if( (bat.getClass() == Batiment.class || bat.getClass() == Machine.class) && (bat != null) && (ouvr != null) && (ouvr.getClass() == Ouvrier.class || ouvr.getClass() == Machine.class) && (!ouvr.

        ouvr.setWorkOn(bat);
        ret = true;
    }

    //We now check if the building linked is finished :

    boolean checkFin = false;

    int wood = 0;
    int stone = 0;
    int intell = 0;
    int tile = 0;

    //Ouvriers
    for (int i = 0; i < this.workers.size(); i++){

        Ouvrier LOuvr = this.workers.get(i);

        if (LOuvr.getWorkOn() == bat){

            wood += LOuvr.getWood();
            stone += LOuvr.getStone();
            intell += LOuvr.getIntell();
            tile += LOuvr.getTile();
        }
    }
```

```java
    //Machines
    for (int j = 0; j < this.machines.size(); j++){

        Machine mach = this.machines.get(j);

        if (mach.getWorkOn() == bat){

            wood += mach.getWood();
            stone += mach.getStone();
            intell += mach.getIntell();
            tile += mach.getTile();
        }
    }

    //We check if the sum of all the resources produced is enough finished the building
    if ( (wood >= bat.getWood()) && (stone >= bat.getStone()) && (intell >= bat.getIntell()) && (tile >= bat.getTile())){

        bat.setState(true);
        checkFin = true;
        ret = true;

        this.setVPoints(bat.getVictPoints());

        if (bat.getClass() == Machine.class) this.machines.add((Machine)bat);
    }

    //We now set free all the workers and machines
    if(checkFin){

        //Ouvriers
        for (int i = 0; i < this.workers.size(); i++){

            Ouvrier LOuvr = this.workers.get(i);

            if (LOuvr.getWorkOn() == bat){

                LOuvr.setState(false);
                LOuvr.setNullWorkOn();
            }
        }
```

```
        //Machines
        for (int j = 0; j < this.machines.size(); j++){

            Machine mach = this.machines.get(j);

            if (mach.getWorkOn() == bat){

                mach.setState(false);
                mach.setNullWorkOn();
            }
        }
    }

    if (ouvr.getState()) System.err.println("sendToWork(ouvr, bat) - Error : The worker is already working on an another building.");

    else if (bat.getState()) System.err.println("sendToWork(ouvr, bat) - Error : The building is already finished.");

    else if (bat.getClass() == Ouvrier.class) System.err.println("sendToWork(ouvr, bat) - Error : A worker is not a building, a worker can't work on a worker.");

    else if (bat == null || ouvr == null) System.err.println("sendToWork(ouvr, bat) - Error : One of the parameter is null.");

    else System.err.println("sendToWork(ouvr, bat) - Error : The worker and / or the building is / are not owned by the player.");

    return ret;
};

/**
```

C) takeCoin()

Pour cette méthode, il est important de gérer les actions achetées lors de l'échange, afin de ne pas bloquer un échange possible car le nombre d'actions « de base » est à 0.

La méthode prend en paramètre le nombre d'actions à échanger. En fonction de celui-ci, le nombre d'actions diminue et le nombre de pièces augmente (deux boucles while diminuent les valeurs actions et actionsBought).

```
/**
 * This method allows a Player object to exchange some of his actions for coins.
 * @param nbActions The number of actions to exchange, that must be between 1 & 3.
 * @return True if everything went well, false otherwise.
 */

public boolean takeCoin(int nbActions){

    boolean ret = false;

    if ( (nbActions > 0) && (nbActions < 4) ) {

        if ( (nbActions == 1) && ( (this.actions+this.actionsBought) >= 1)) {

            if (this.actions >= 1) this.actions--;
            else this.actionsBought--;

            this.coins++;
            this.leJeu.setReserve(leJeu.getReserve()-1);
            ret = true;

        } else if ( (nbActions == 2) && ((this.actions+this.actionsBought)) >= 2 ){

            int counter = 2;

            while ( (this.actions > 0) && (counter > 0)){

                counter--;
                this.actions--;
            }

            while ( (this.actionsBought > 0) && (counter > 0)){

                counter--;
                this.actions--;
            }

            this.coins += 3;
            this.leJeu.setReserve(leJeu.getReserve()-3);
            ret = true;

        }
```

```
        else if ( (nbActions == 3) && (this.actions+this.actionsBought) >=3 ) {

            int counter = 3;

            while ( (this.actions > 0) && (counter > 0)){

                counter--;
                this.actions--;
            }

            while ( (this.actionsBought > 0) && (counter > 0)){

                counter--;
                this.actions--;
            }

            this.coins += 6;
            this.leJeu.setReserve(leJeu.getReserve()-6);
            ret = true;

        }
    } else System.err.println("takeCoin(nbActions) - Error : The number of actions to exchange must be between 1 & 3.");

    return ret;
}
```

D) HumanPlayer – play()

Afin que le joueur puisse sélectionner l'action à effectuer, la méthode play() de HumanPlayer utilise un Scanner pour permettre la saisie du numéro de l'action au clavier.

Les méthodes de Player correspondant aux actions renvoyant un boolean, une boucle demande à l'utilisateur de rentrer une action (et les n° de cartes / nombres d'actions à échanger, etc.) tant que l'action ne se déroule pas avec succès. La plus grande difficulté ici a été de **gérer le nombre d'actions à imputer au joueur**. En effet, la règle stipulant que le nombre d'actions augmente si deux ou plusieurs ouvriers sont envoyés sur le même chantier durant le même tour fait qu'un traitement particulier de la diminution du nombre d'action doit être effectuée ici.

```
/**
 * This method allows the player to make a move.
 */

public void play(){

    //We first display the possible actions and the corresponding number
    System.out.println("\n==============================");

    System.out.println("Actions : ");
    System.out.println("- Ouvrir un chantier \t\t\t--> 0");
    System.out.println("- Recruter un ouvrier \t\t\t--> 1");
    System.out.println("- Envoyer travailler un ouvrier \t--> 2");
    System.out.println("- Prendre un ou plusieurs écus \t\t--> 3");
    System.out.println("- Acheter une nouvelle action \t\t--> 4");
    System.out.println("- Sauvegarder la partie \t\t--> 5");

    boolean execCorrecte = false;
    int action = 0;

    //The Card (building or machine) where the worker will be send to work in case the corresponding action is played, the boolean tell if the building is already in the ArrayList.
    Card bat = null;
    boolean alreadyPlayed = false;


    while (!execCorrecte){

        System.out.println("\nJouer l'action n° : \t");

        //The chosen action number is get
        action = this.scanAct.nextInt();

        //While a correct action id has not been selected, a new action id is asked
        while ( (action < 0) || (action > 5)) {

            System.out.println("N° incorrect, veuillez entrer un nouveau n° entre 0 et 5 : ");
            action = this.scanAct.nextInt();
        }

        //The game's available cards
        ArrayList<Card> lesCartes = this.getLeJeu().getStackBuilding();
        ArrayList<Ouvrier> lesOuvr = this.getLeJeu().getStackWorkers();
```

```
//Opening of a building
if (action == 0) {

    System.out.println("Veuillez choisir la carte Bâtiment/Machine : ");

    int numBat = this.scanAct.nextInt();

    //We check if the chosen card is part of the 5 displayed cards on the board

    while ( (numBat < 0) || (numBat > 4) ) {

        System.out.println("Veuillez choisir une carte posée sur le plateau :");
        numBat = this.scanAct.nextInt();
    }

    Card build = lesCartes.get(numBat);

    execCorrecte = this.openBuild(build);
}

//Worker recruitement
else if (action == 1){

    System.out.println("Veuillez choisir l'ouvrier à recruter : ");

    int numOuvr = this.scanAct.nextInt();

    while ( (numOuvr < 0) || (numOuvr > 4) ) {

        System.out.println("Veuillez choisir une carte posée sur le plateau :");
        numOuvr = this.scanAct.nextInt();
    }


    Ouvrier ouvr = lesOuvr.get(numOuvr);

    execCorrecte = this.recruit(ouvr);

}
```

```
    //Send a worker to work
    else if (action == 2){

        System.out.println("Veuillez choisir l'ouvrier à envoyer travailler : ");

        int idOuvr = this.scanAct.nextInt();

        //While a correct worker has not been selected, a new id is asked
        while ( (idOuvr <0 ) || (idOuvr > this.getWorkers().size()) ){

            System.out.println("Veuillez choisir un ouvrier dans la liste des ouvriers détenus : ");
            idOuvr = this.scanAct.nextInt();
        }

        System.out.println("Veuillez choisir le chantier sur lequel l'ouvrier doit travailler : ");

        int idChantier = this.scanAct.nextInt();

        bat = this.getCards().get(idChantier);
        Ouvrier ouvr = this.getWorkers().get(idOuvr);

        execCorrecte = this.sendToWork(ouvr, bat);

        if (execCorrecte) {

            alreadyPlayed = this.getJoueSur().contains(bat);

            if (!alreadyPlayed) this.getJoueSur().add(bat);

            else this.setCounterSend(this.getCounterSend()+1);

        }
    }
```

```
if (execCorrecte) {

    //This part is made to include the cost of actions that increase with the number of workers send on the same building in a turn.
    if ( (action == 2) && (alreadyPlayed)){

        int costActions = this.getCounterSend();

        while (this.getActions() > 0 && costActions > 0){

            this.setActions(this.getActions()-1);
            costActions--;
        }

        while (this.getActionsBought() > 0 && costActions > 0){

            this.setActionsBought(this.getActionsBought()-1);
            costActions--;
        }
    }

    else{

        if (this.getActions() > 0) this.setActions(this.getActions()-1);

        else if (this.getActionsBought() > 0 ) this.setActionsBought(this.getActionsBought()-1);
    }
}
else {

    System.out.println("Entrez le chemin du fichier de sauvegarde, de la forme /Documents/sauvegarde/monFichier.txt : ");
    String savePath = this.scanAct.next();
    this.getLeJeu().getLesBat().save(savePath);
}
```

E) intializeStackBuilding & initializeStackWorkers

Afin de pouvoir afficher les cartes Batiments, Machines & Ouvriers disponibles, il était plus simple de créer deux ArrayList, de taille 5, contenant pour l'une des Batiments et des Machines, et pour l'autre des Ouvriers. En effet, cela permet ensuite au joueur de sélectionner un numéro de carte entre 0 et 4 (il était très compliqué de sélectionner une carte sinon, car il aurait fallu parcourir l'entièreté du paquet de cartes à chaque fois, tout en laissant passer un certain de nombre de cartes Ouvrier, Machine ou Batiment si l'on ne souhaitait pas choisir la première carte qui correspondait).

Ces paquets sont initialisé avant chaque appel à la méthode start() de Player.

```
/**
 * This method intialize the stack of building with the first 5 Card objects from the batch of Card.
 */

public void intializeStackBuilding(){

    this.stackBuilding.clear();
    int i = 0;
    boolean fin = false;

    while ( (i < this.cards.size()) && (!fin)){

        Card carte = this.cards.get(i);
        if (carte.getClass() != Ouvrier.class) {

            this.stackBuilding.add((carte));
        }

        if (this.stackBuilding.size() == 5) fin = true;

        i++;
    }
};
```

```java
/**
 * This method intialize the stack of workers with the first 5 Ouvrier objects from the batch of Card.
 */

public void intializeStackWorkers(){

    this.stackWorkers.clear();
    int i = 0;
    boolean fin = false;

    while ( (i < this.cards.size()) && (!fin)){

        Card carte = this.cards.get(i);
        if (carte.getClass() == Ouvrier.class) {

            this.stackWorkers.add((Ouvrier)carte);
        }

        if (this.stackWorkers.size() == 5) fin = true;

        i++;
    }
};
```

F) start()

Enfin, la méthode qui gère le jeu et le tour des joueurs : start().

Après chaque fin de tour, on vérifie le nombre de points du joueur. Si il est égal ou supérieur à 17, la partie se finie, les autres joueurs jouent encore une fois.

```java
public void start(){

    /*
    Reminder for the end of the game :

        - Only the points made by the building are counted
        - There is a bonus point for each 10 coins left
    */

    boolean fin = false;

    ConsoleDisplay displ = this.lesBat.getAffichageTxt();

    while (!fin){

        this.changeCurrent();
        this.current.setActions(3);
        this.current.setActionsBought(0);
        this.current.setCounterSend(0);
        this.current.getJoueSur().clear();
        this.intializeStackWorkers();
        this.intializeStackBuilding();

        displ.printGame();

        while ( (this.current.getActions() > 0) || (this.current.getActionsBought() > 0) ){

            this.current.play();
            this.intializeStackWorkers();
            this.intializeStackBuilding();
            displ.printGame();
        }

        this.turn++;
```

```
            if (this.current.getPoints() >= 17) {

                fin = true;

                //Fin du tour après un vainqueur
                Player vainqueur = this.current;

                this.changeCurrent();

                while (this.current != vainqueur){

                    this.current.setActions(3);
                    this.current.setActionsBought(0);
                    this.current.setCounterSend(0);
                    this.current.getJoueSur().clear();
                    this.intializeStackWorkers();
                    this.intializeStackBuilding();

                    displ.printGame();

                    while ( (this.current.getActions() > 0) || (this.current.getActionsBought() > 0) ){

                        this.current.play();
                        this.intializeStackWorkers();
                        this.intializeStackBuilding();
                        displ.printGame();
                    }

                    this.changeCurrent();

                    this.turn++;
                }

            }
        }

        displ.endOfGame();
    };
```

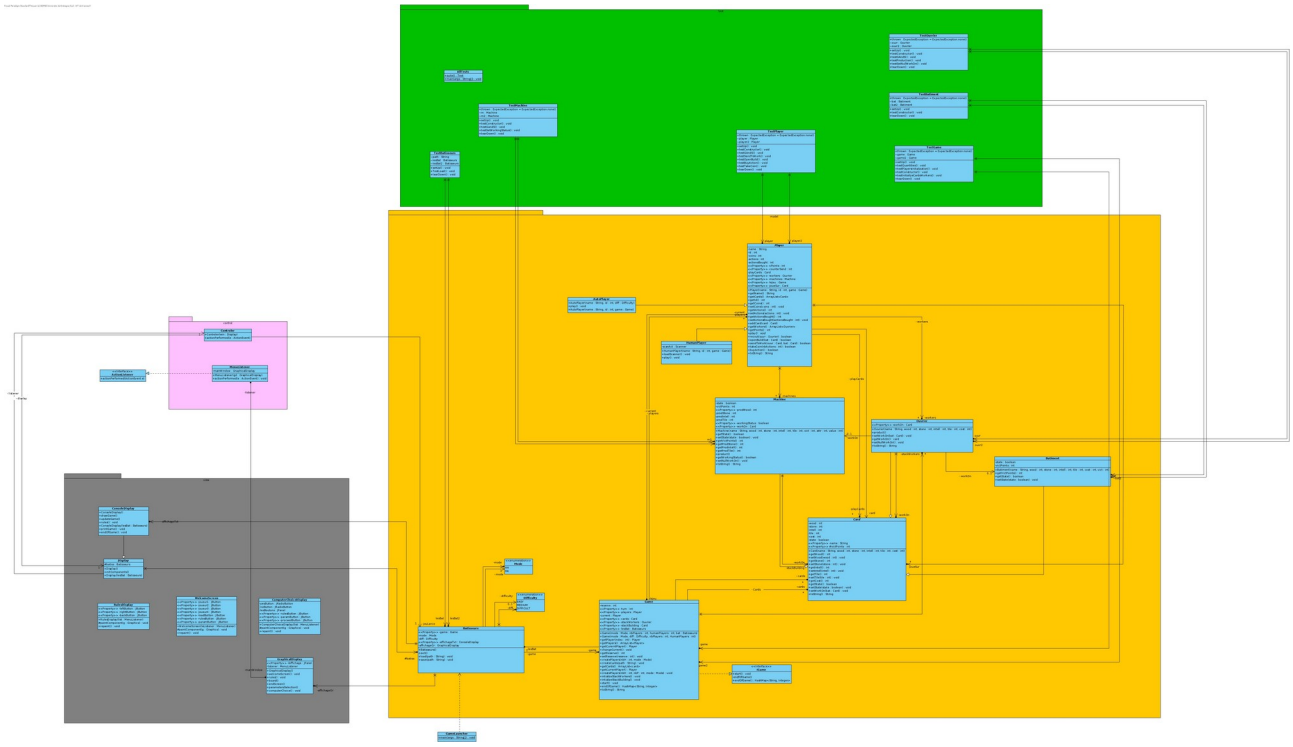
Commentaire sur le code: L'utilisation de if et else if a été préférée à l'utilisation de switch / case.

L'utilisation de ces derniers pourrait permettre de gagner en visibilité, mais les différentes conditions seraient plus implicite. Le gain de visibilité se ferait donc au détriment de la bonne compréhension du code.

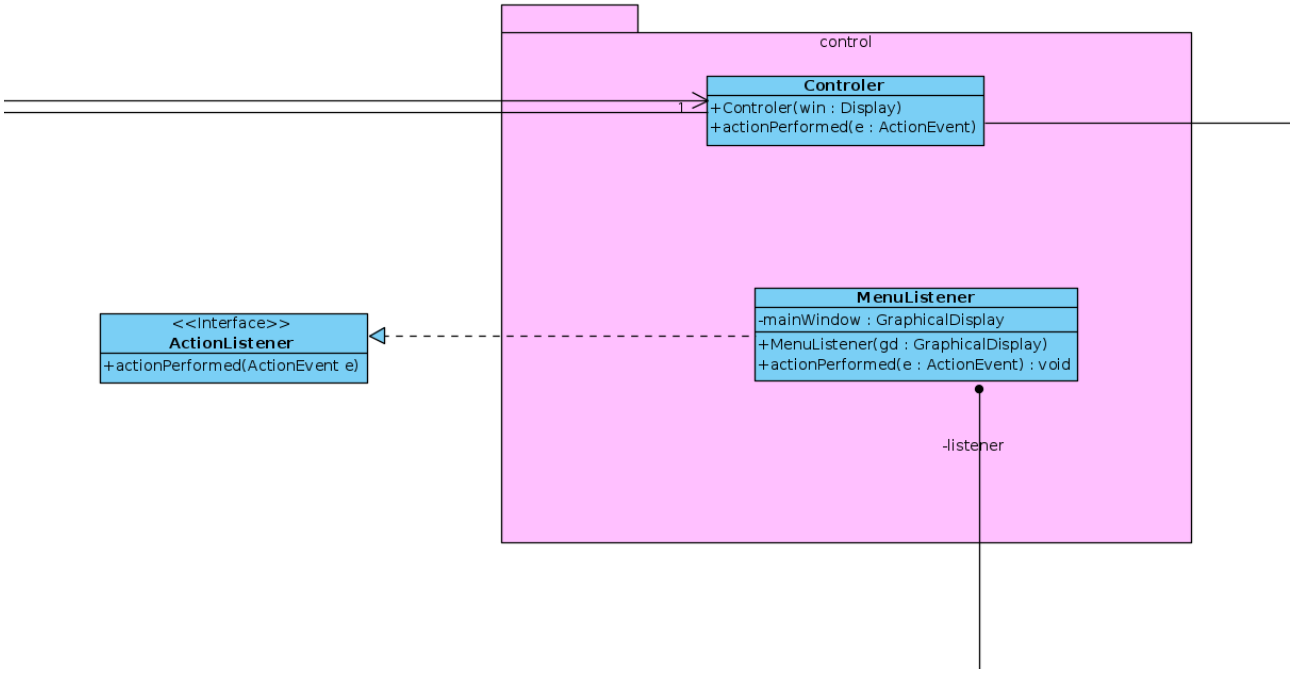
III) Diagramme de classe d’implémentation

Le diagramme ci-dessous est un diagramme d’implémentation obtenu via l’outil de rétro-conception de code Java du logiciel *Visual Paradigm*.

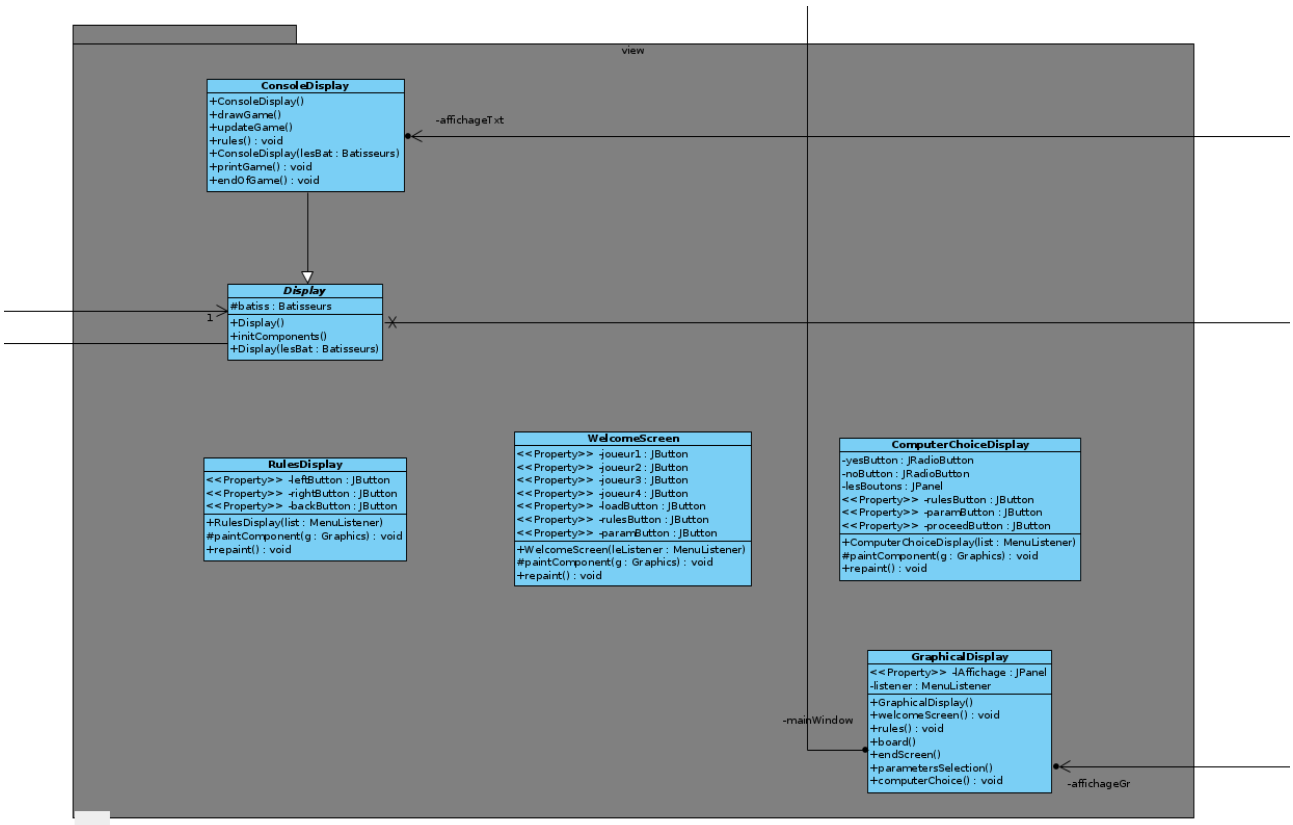
Ce diagramme permet de visualiser l’évolution de la cartographie du code du projet par rapport aux diagrammes de classe d’analyse et de conception du rendu précédent. La différence est notable, pour plusieurs raisons.



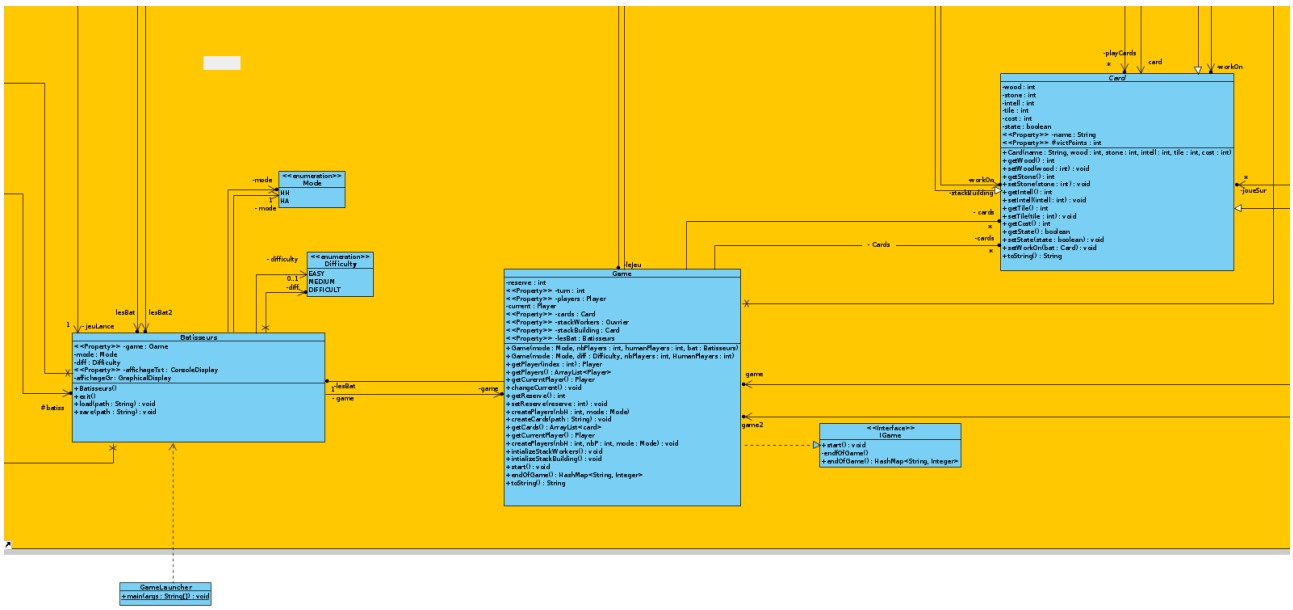
Vue d'ensemble du diagramme de classe



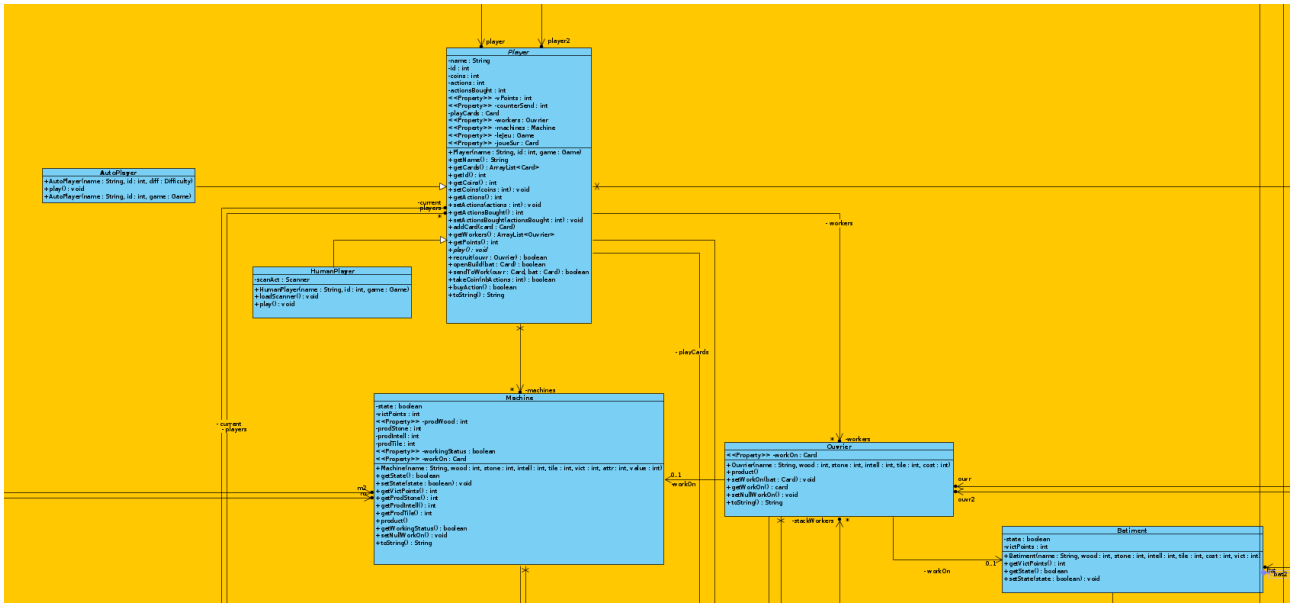
Package control



Package view



Package model - Partie basse



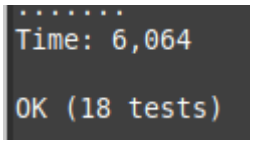
Package model - Partie haute

Package test

IV) Campagne de tests effectuée

Avant d'effectuer les différents scénarios de tests inscrit dans le cazhier des charges, il a fallu effectuer un certain nombre de tests unitaires.

Ci-dessous le rapport de test (on notera que les erreurs sont dues à une mauvaise intiliasation des cartes, à cause du fait que les tests sont compilés & lancés avec Ant dans le super-répertoire contenant src, class, etc. et non compilés depuis ws, car dans ce dernier cas, tous s'exécutent correctement)



Ci-dessus le résultat de l'exécution en ayant compilé depuis ws.

Ci-dessous le bilan du rapport de test, avec les erreurs provoquées par l'initialisation des cartes :

Testsuite: test.AllTests

Tests run: 18, Failures: 0, Errors: 10, Skipped: 0, Time elapsed: 0,051 sec

Fonctionnalité testée	Étapes à suivre	Résultat du test
Choix du nombre de joueurs	Version texte : • Lancer une partie • Entrer le nombre de joueur total Version graphique : • Lancer une partie • Appuyer sur le bouton correspondant	Fonctionnel
Jouer avec l'ordinateur	• Sélectionner le nombre de joueurs • Sélectionner l'option « Jouer avec l'ordinateur »	Fonctionnel
Mise en place du plateau	• Lancer une partie	Fonctionnel
Enregistrement d'une partie	Version texte :	Fonctionnel

	• Lancer une partie • Renseigner l'action « Enregistrer » Version graphique : • Lancer une partie • Appuyer sur le bouton avec la disquette	
Restauration d'une sauvegarde	Version texte : • Renseigner le chemin vers la sauvegarde Version graphique : • Appuyer sur le bouton « Reprendre » • Ouvrir la sauvegarde	Fonctionnel
Ouvrir un chantier	Version texte : • Lancer une partie • Renseigner le numéro de l'action correspondante Version graphique : • Lancer une partie • Appuyer sur le bouton correspondant à l'action	Fonctionnel
Recruter un ouvrier	Version texte : • Lancer une partie • Renseigner le numéro de l'action correspondante Version graphique : • Lancer une partie • Appuyer sur le bouton correspondant à l'action	Fonctionnel
Envoyer travailler un ouvrier	Version texte : • Lancer une partie • Renseigner le numéro de l'action correspondante Version graphique :	Fonctionnel

	• Lancer une partie • Appuyer sur le bouton correspondant à l'action	
Pouvoir faire travailler plusieurs ouvriers sur un même chantier	Version texte : • Lancer une partie • Renseigner le numéro de l'action correspondante deux fois de suite, en affectant les deux ouvriers deux fois sur le même chantier Version graphique : • Lancer une partie • Appuyer sur le bouton correspondant à l'action deux fois de suite, en affectant les deux ouvriers deux fois sur le même chantier	Fonctionnel
Échanger des écus	Version texte : • Lancer une partie • Renseigner le numéro de l'action correspondante Version graphique : • Lancer une partie • Appuyer sur le bouton correspondant à l'action	Fonctionnel
Acheter une nouvelle action	Version texte : • Lancer une partie • Renseigner le numéro de l'action correspondante Version graphique : • Lancer une partie • Appuyer sur le bouton correspondant à	Fonctionnel

	l'action	
Quitter le logiciel	Version texte : • A n'importe quel stade dans le jeu, appuyer sur la croix ou faire Ctrl + C Version Graphique : • A n'importe quel stade dans le jeu, appuyer sur la croix	Fonctionnel

V) État d'avancement du développement

En comparant avec les fonctionnalités annoncées dans le cahier des charges, voici l'état d'avancement du développement :

La version console du jeu implémente toute les fonctionnalités de base attendue, et implémente la fonctionnalité (déclarée comme supplémentaire) de l'affichage des règles du jeu.

La version graphique du jeu n'implémente que les fonctionnalités suivantes :

- Possibilité de choisir le nombre de joueur
- Possibilité de choisir de jouer avec ou sans l'ordinateur
- Possibilité d'afficher les règles du jeu

Il reste donc à effectuer, pour la version graphique, l'implémentation du système de sauvegarde, ainsi que la mise en place du plateau et la gestion des actions des joueurs.

VI) Synthèse des difficultés rencontrées et des solutions apportées

Comme évoqué dans le II), plusieurs difficultés algorithmiques ont été rencontrées. Les solutions apportées à ces difficultés sont présentés dans cette même partie.

L’élaboration du processus de sauvegarde / chargement utilise l’interface Serializable. L’utilisation d’un objet Scanner pose problème, car la classe Scanner est incompatible avec Serializable. Il a donc fallu passer l’attribut scan de HumanPlayer en *transient*, et créer une méthode loadScanner() pour pouvoir recharger les flux d’entrée des différents objets HumanPlayer.

Concernant l’interface graphique, la difficulté majeure a été le « rafraîchissement » de l’objet Jframe, après un clic sur un des boutons. En effet, cet objet Jframe a pour seul attribut un objet Jpanel, dont le contenu est modifié selon le bouton sélectionné, en reliant cet attribut à un objet WelcomeScreen pour l’écran d’accueil, RulesDisplay() pour l’affichage des règles, etc.

VII) Bilan personnel du travail réalisé

À la fin de cette période de programmation intensive, voici le bilan que l’on peut dresser : le jeu possède une interface console fonctionnelle, implémentant toutes les règles du jeu « Les Batisseurs », et une interface graphique, comportant uniquement les différents menus.

Les différents scénarios de tests donnent le résultat attendu.

Une meilleure gestion du temps, au début de la phase de programmation, aurait peut être permis de finir quelques aspects supplémentaires de l’interface graphique. La gestion des difficultés rencontrées avec l’utilisation de Java Swing et des bibliothèques associées a pris plus de temps que prévu, il était donc difficile de se tenir au planning initial.