POO - Projet Rogue

Rémy Chaput - Amaury Maillé - Rémy Cazabet
Noura Faci - Frédéric Armetta

Mars 2021

1 Introduction

Rogue est un jeu d'énigmes dans lequel vous contrôlez Le Professeur qui tente de s'échapper d'un labyrinthe en 2D vue du dessus. Ce labyrinthe est composé de **salles** reliées entre elles par des **portes**. Pour s'enfuir Le Professeur doit parvenir à trouver la **salle finale**. Chaque salle est composée d'un certain nombre de **dalles**. Certaines de ces dalles sont à **usage unique**, Le Professeur ne peut les traverser qu'une seule fois, après quoi elles s'enflamment et deviennent inutilisables. Cependant, dans chaque salle Le Professeur peut utiliser un certain nombre de **capsules** contenant de l'eau capable de rendre ces dalles traversables une nouvelle fois en éteignant le feu ; une capsule permet d'éteindre une seule dalle enflammée. Il dispose d'une **quantité limitée** de capsules par salle, il lui faut donc utiliser ces capsules avec parcimonie et de façon réfléchie.

Les portes qui relient les salles peuvent être *verrouillées*. Dans ce cas-là, Le Professeur doit trouver des **clés** dans les salles, afin de déverrouiller les portes. Ces clés peuvent se trouver à même le sol, ou être contenues dans des **coffres**.

1.1 Illustration

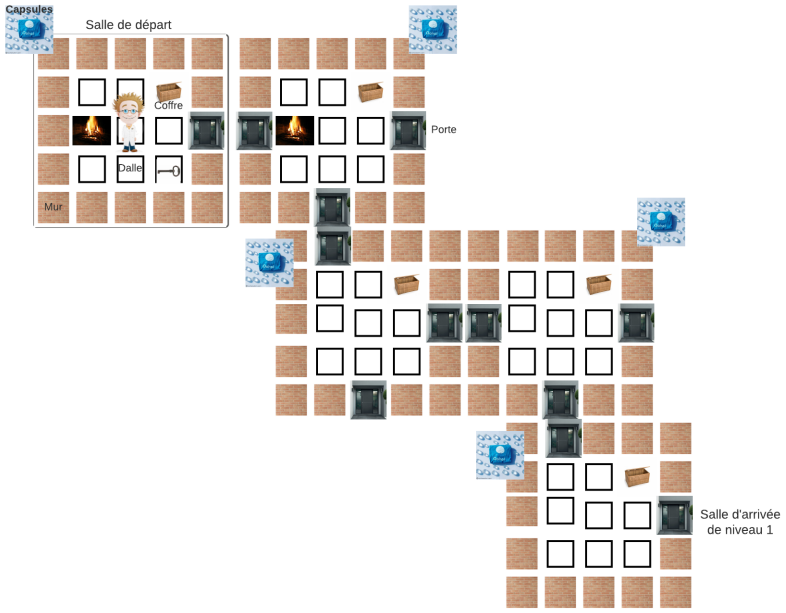


Figure 1: Exemple de grille de jeu (= 1 niveau).

La Figure 1 présente une vue globale du jeu. Les Figures 2 et 3 illustrent le comportement du jeu lors de déplacements et du saut. Le gris représente un mur, le orange une porte, le bleu une dalle normale, le vert une case à usage unique traversable, le rouge une dalle à usage unique non traversable, le jaune représente la position du professeur, le noir représente du vide. Sur la Figure 3 qui présente le comportement de l'extension "saut", la quatrième ligne présente deux choix possibles pour le déplacement du professeur : sauter de l'autre côté du vide, ou poursuivre dans sa direction.

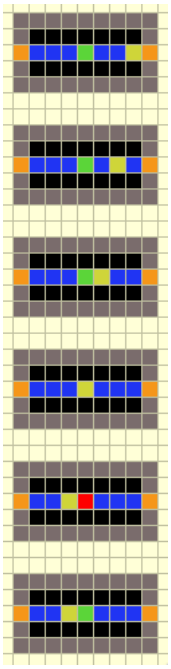


Figure 2: Illustration du déplacement et du changement d'état de case

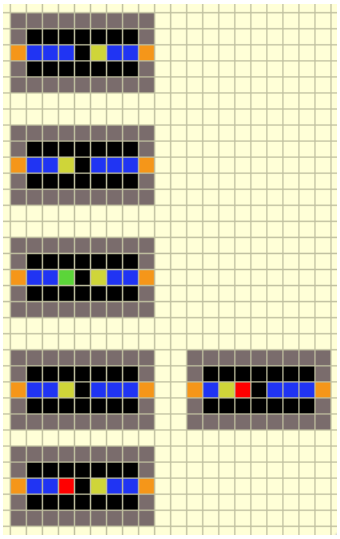


Figure 3: Illustration du saut au-dessus d'une case vide (extension)

1.2 Votre travail

Votre travail consiste à coder Rogue en vous basant sur le code fourni qui est disponible ici, et qui respecte le diagramme fourni en Figure 4. Ce que nous avons présenté dans l'introduction est une version très simple du jeu, et vous devrez **à minima** coder tout ce qui est nécessaire pour que cela fonctionne. Ensuite, vous réaliserez une ou deux extensions (voir Section 2.2). La Section 2 vous présente la progression suggérée.

Le travail sera réalisé **en binôme** ; vous devrez travailler entre les séances encadrées. Il vous est fortement conseillé d'utiliser un outil de versionnage de code, tel que Git, afin de faciliter la collaboration au sein de votre binôme. Il peut être intéressant pour vous de publier le code sur GitHub (cela peut faire un exemple de votre travail, à citer dans votre CV).

Pour réaliser ce projet, vous travaillerez avec le *framework* Swing dans le langage Java, qui vous permet de réaliser des interfaces graphiques. Si vous avez déjà eu l'occasion de travailler avec un autre *framework* et que vous vous estimez capable de partir de cet autre *framework* sans vous pénaliser en terme de temps, vous êtes libre de le faire.

Vous êtes responsables de l'analyse objet : **le code doit être le plus objet possible. Privilégiez donc une programmation objet plutôt qu'un algorithme central, long et complexe.** Veillez à implémenter les fonctionnalités de manière incrémentale, afin d'avoir une démonstration opérationnelle lors de la soutenance.

Vous rendrez votre travail, accompagné d'un rapport, dans une unique archive au format **ZIP**. Les informations détaillées d'évaluation sont disponibles en Section 3.

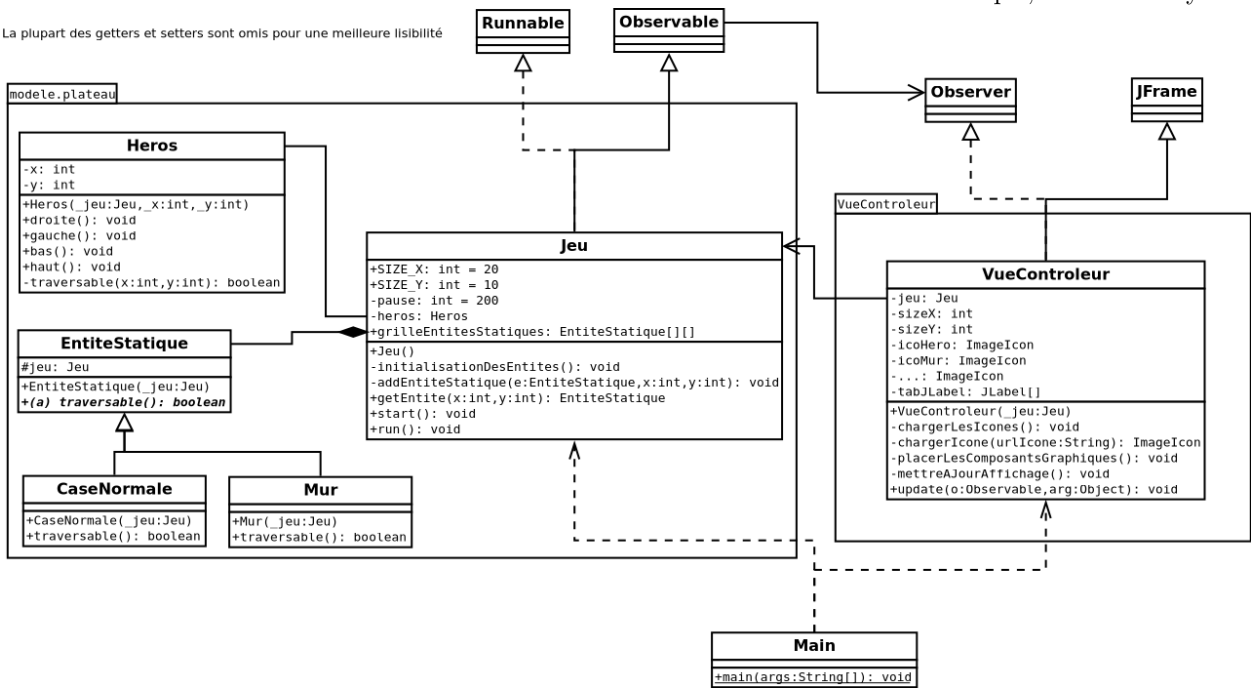


Figure 4: Diagramme de classes du code de base (fourni)

2 Progression

Ce projet se déroule en deux parties distinctes : une première partie commune à tous les étudiants, et une seconde partie composée d'extensions que vous choisirez parmi une liste de suggestions ou que vous proposerez. Vous devez réaliser toutes les tâches de la partie commune avant de choisir les extensions que vous souhaitez ajouter.

2.1 Partie commune

Cette partie est obligatoire, et vous devez la finir avant de vous attaquer aux extensions. Nous vous invitons à réaliser les différentes étapes suivantes dans l'ordre dans lequel nous vous les présentons. Certaines précisions toutefois :

- Le jeu est composé de plusieurs salles, reliées entre elles par des portes. Pour vous simplifier la vie, nous vous invitons à faire en sorte que l'ensemble des salles tienne sur l'écran sans qu'il y ait besoin de *scroller* (décaler l'affichage à mesure que le personnage se déplace).

- Vous allez être amenés à créer des pièces (des salles). Nous ne sommes pas là pour évaluer vos compétences en *level design*, aussi il n'est pas nécessaire que vos pièces soient prêtes à l'emploi dans un jeu d'énigmes, mais il est nécessaire qu'elles permettent d'illustrer toutes les fonctionnalités que vous avez pu mettre en place.

- Les salles sont composées de cases ; le déplacement dans Rogue est considéré *discret*, c'est-à-dire qu'on se déplace d'une case à chaque fois (on ne peut pas être entre deux cases).

- Nous vous recommandons de mettre rapidement en place une touche qui permette de réinitialiser soit la salle courante, soit tout le jeu, afin de ne pas avoir à quitter l'application puis à la relancer. Cela vous permettra de tester plus facilement. Il n'y a pas de bon ou de mauvais moment pour mettre

cette réinitialisation en place, et vous serez sans doute amenés à la modifier à mesure que votre Modèle évolue.

Sans plus attendre, voici la progression que nous vous proposons :

1. Commencez par vous familiariser avec la base de code qui vous est fournie, ainsi que le **diagramme UML** associé (voir Figure 4). Comprenez comment est-ce que les différentes parties du code interagissent entre elles. Vous reconnaîtrez le *design pattern* MVC (un rappel sur le *pattern* MVC est disponible en Annexe A ; nous vous invitons également à vous référer à l'Annexe B pour vous (re)familiariser avec le *design pattern* Observer).

2. Codez l'inventaire du joueur. Dans un premier temps celui-ci ne comportera que des clés. Avez-vous besoin d'une classe particulière pour cela ? Ajoutez-vous une classe `Inventaire` instanciée dans la classe `Heros` ? Ou ajoutez-vous un compteur de clés dans le code de la classe `Heros` ?

3. Ajoutez les classes nécessaires pour représenter les portes (qui peuvent être verrouillées ou non), ainsi que les coffres, les clés et les capsules d'eau. Avez-vous besoin d'une classe `Porte` et d'une classe `PorteVerrouillee` ? Ou pouvez-vous vous contenter d'une simple classe `Porte` ?

   - En termes techniques, les clés, capsules et coffres sont des ramassables (*pickups*). Un coffre est un *pickup* particulier dans le sens qu'il peut contenir d'autres *pickups* (pourquoi pas d'autres coffres tant qu'on y est !). Cette structure particulière pour le coffre correspond au *design pattern* Composite, sur lequel vous pouvez vous documenter ici. Vous êtes libres de traiter cette remarque de la façon qui vous semble la plus appropriée.

4. Intégrez la notion d'*orientation* : lorsque vous appuyez sur une touche pour déplacer le personnage dans une direction, l'orientation du personnage devient cette direction. L'orientation du personnage sera importante pour le lancé de capsules. Avez-vous besoin d'une classe pour cela ? Travaillez-vous uniquement dans le Modèle ?

5. Codez les dalles à usage unique. Encore une fois, avez-vous besoin d'une nouvelle classe, ou pouvez-vous adapter une classe existante ? Pourquoi ?

6. Passez à la création de plusieurs salles, reliées entre elles par des portes. Pour vous simplifier la vie, commencez par faire en sorte que les salles soient gérées sur une seule et même grille. Une des extensions consistera à avoir une grille distincte par salle, ce qui se prête mieux à la génération procédurale. Implémentez également le passage d'une salle à l'autre grâce aux portes.

7. Modifiez la génération de salles de façon à placer des *pickups* au sol (clés, coffres, peut-être des capsules additionnelles). Essayez de les placer de façon aléatoire.

8. Mettez en place les interactions avec les *pickups* ainsi qu'avec les dalles à usage unique. Les clés sont ajoutées à l'inventaire du joueur lorsqu'elles sont ramassées. Comment souhaitez-vous gérer les coffres ? Est-ce que leur contenu apparaît au sol autour d'eux lorsque le joueur les touches ? Est-ce que leur contenu est placé dans l'inventaire du joueur directement ? Voulez-vous peut-être faire apparaître une petite fenêtre supplémentaire ? Vous êtes libres de faire comme bon vous semble.

9. Ajoutez le concept de cases vides : le joueur ne peut pas se déplacer sur une case vide, et ne peut rien en faire du tout. Il s'agit tout bonnement d'un trou dans le sol.

10. Ajoutez l'utilisation des capsules : lorsque le joueur appuie sur une touche de votre choix, il utilise une capsule sur la dalle en face de lui (concept d'orientation). Attention, il serait absurde de laisser le joueur utiliser une capsule sur une dalle encore utilisable !

11. Ajoutez la recharge des capsules d'eau lorsque le joueur entre dans une salle. Attention, les charges de capsules ne sont pas conservées entre les salles !

Ici s'arrête la partie commune. Nous vous présentons à présent les extensions que vous pouvez réaliser.

2.2 Extensions

Les extensions marquées (Simple) sont considérées, comme leur nom l'indique, simples, et vous rapporteront moins de points. Certaines extensions vous demanderont de repenser votre architecture, d'autres vous poseront des difficultés algorithmiques. Choisissez comme bon vous semble, et il n'est pas nécessaire de vous limiter à ce que nous proposons : vous pouvez inventer vos propres extensions (à valider avec votre encadrant au préalable).

- (Simple) Plusieurs niveaux. Cette extension est plus algorithmique qu'orientée objet, aussi est-elle considérée "simple". Faites en sorte d'avoir plusieurs niveaux, éventuellement avec une difficulté croissante. Encore une fois, la qualité des salles que vous allez générer n'entre pas en compte.
- Ajouter le saut. Lorsque le joueur se trouve devant une case vide, mais qu'une dalle non enflammée est accessible immédiatement de l'autre côté de la case vide, il peut sauter par-dessus le trou. Attention, le saut permet de franchir une unique case vide, pas plusieurs, et n'est possible que lorsque le joueur est orienté en direction du trou.
- (Simple) Plusieurs types de coffres et de clés. Vous pouvez imaginer que la porte de la dernière salle requiert une clé en or, qui se trouve dans le coffre en argent. Le coffre en argent s'ouvre avec la clé en argent que l'on trouve dans le coffre en bronze, qui est lui-même déverouillé par la clé en bronze qui se trouve quelque part dans les salles. Avez-vous besoin de nombreuses classes pour résoudre ce problème ?
- *Scrolling.* Faites en sorte de n'afficher que la salle dans laquelle le joueur se trouve, et changez l'affichage lorsqu'il sort de celle-ci.
- Ajoutez différents types de dalles : dalles à pics par exemple, qui infligent des dégâts au joueur s'il marche dessus lorsque les pics sont sortis. Posez-vous ici la question du temps : est-ce que le temps va être une fonction des actions du joueur, ou bien va-t-il évoluer de façon indépendante ? Autrement dit, est-ce que les pics sortent / se rétractent lorsque le joueur se déplace, ou bien le font-ils de façon indépendante. Faut-il modifier le modèle ou la vue pour gérer cela ? Pourquoi ?
- Ajoutez des pièges. Ici vous pouvez laisser libre court à votre imagination, vous pouvez imaginer des dalles piégées, ou imaginer que des fléchettes traversent la salle en continu (ajoutez alors des points de vie au joueur, et donc des cœurs dans les salles pour le régénérer), vous pouvez imaginer des effets de statut (empoisonnement par exemple).
- Ajoutez une pièce dans laquelle le joueur peut dépenser de l'argent récupéré dans les salles pour par exemple obtenir plus de capsules afin de traverser les salles. Que modifiez-vous ? Modèle ? Vue ? Contrôleur ? Pourquoi ?
- Créez des salles à contraintes : impossible de sauter tant que le joueur n'a pas récupéré un certain objet dans la salle, ou impossible d'utiliser des capsules tant que le joueur n'a pas récupéré un certain objet. Pourquoi pas limiter le nombre de sauts disponibles dans une salle ? Comment modifiez-vous le Modèle pour prendre cela en compte ?

3 Évaluation

3.1 Critères d'évaluation

Vous serez notés selon les critères suivants :

- Qualité de l'analyse objet et du code associé.
- Respect du *design pattern* Modèle Vue Contrôleur Strict (voir Annexes A et B).

- Modularité de l'approche (facilité à modifier votre code pour ajouter des fonctionnalités).
- Respect des fonctionnalités obligatoires à implémenter.
- Extensions proposées. Vous devez réaliser une à deux extensions, sachant que des extensions plus complexes rapporteront plus de points. Vous serez évalué sur la qualité de la mise en place de l'extension (modélisation UML et propreté du code).

L'évaluation sera individuelle, veillez donc à vous répartir le travail équitablement au sein de votre binôme. Le plagiat de code est, évidemment, interdit.

3.2 Présentation / Démo

Vous devrez présenter, en binôme (présence des deux membres obligatoire), votre projet lors d'une soutenance durant la dernière séance (sauf changement ultérieur, il s'agira du Mercredi 14/04, l'après-midi : sur un créneau d'une dizaine de minutes entre 14h et 17h15) :

- Quelques minutes de présentation : montrer le jeu en fonctionnement, les fonctionnalités implémentées, vos choix de conception.
- Choisissez une partie de votre analyse pour détailler l'explication et justifier (1-2 minutes).
- Quelques minutes de questions **individuelles** : attention, les deux membres du binôme doivent avoir compris **l'ensemble** du code et pouvoir répondre à toutes les questions, y compris sur les parties qu'ils n'ont pas codées.

Si votre bande passante le permet, vous montrerez durant la présentation le rendu final du jeu et votre code ; pas de diaporama.

3.3 Rendu

Vous devrez rendre, sur Tomuss, une archive *au format* ***ZIP*** contenant :

- Les sources de votre projet, c'est-à-dire tout ce qui est nécessaire pour le compiler puis l'exécuter (fichiers code source `.java`, ressources comme les fichiers audio, images, `.jar` pour les bibliothèques externes, etc.). Attention à rendre ***uniquement*** les sources : les fichiers objets comme les `.class` (compilés) seront pénalisés, ainsi que les fichiers de configuration de l'IDE ou le dossier `.git` (si vous utilisez Git).
- Un fichier `.jar` exécutable de votre projet.
- Un fichier `README` expliquant (brièvement) comment lancer votre projet. Insistez en particulier sur le dossier depuis lequel lancer le `.jar`. Le `README` peut contenir d'autres informations, telle qu'une description du projet, en particulier si vous avez l'intention de rendre votre dépôt public sur GitHub.
- Un rapport au format **PDF** qui présentera le travail que vous avez accompli, ainsi que **le diagramme de classes** représentant vos choix de conceptions. Le rapport (8 pages maximum) doit contenir :
  - La liste des fonctionnalités et extensions, ainsi que la proportion de temps passée sur chacune d'elles et la répartition au sein du binôme.
  - Une documentation UML, au minimum un **diagramme de classes**.
  - La justification de votre analyse et vos choix de conception.
  - Des copies d'écran d'une partie en cours, en particulier des fonctionnalités que vous souhaitez mettre en valeur.

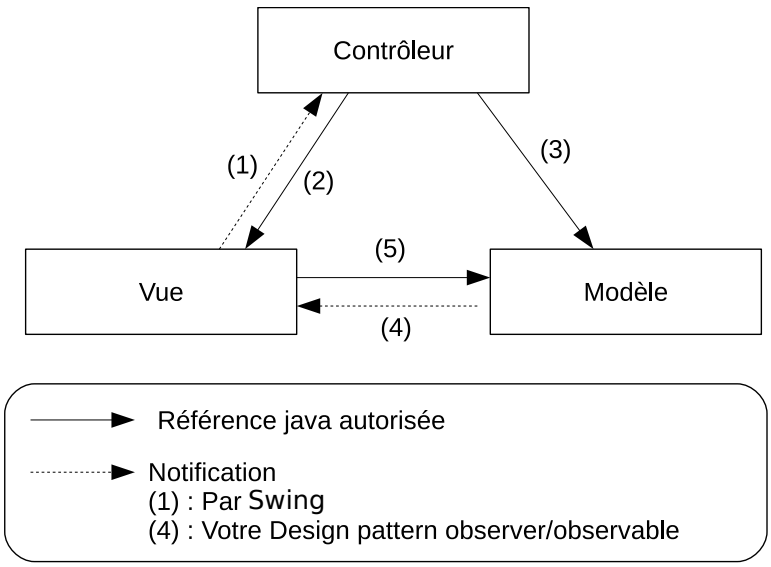
- Éventuellement, vous pouvez ajouter dans l'archive votre diagramme de classes, au format `PNG`, séparément du rapport (par exemple si le diagramme est trop grand pour être lisible correctement dans le rapport).

Merci de rendre une archive "propre" : la racine doit contenir un unique dossier (par exemple nommé "Projet_POO_pXXXXXXX_pYYYYYYY"), et ce dossier contiendra les éléments demandés (code source, rapport, exécutable, Readme). En d'autres termes, l'extraction de l'archive ne doit ***pas*** fournir les fichiers en vrac.

Annexes

A Rappel *Pattern* MVC (Modèle-Vue-Contrôleur)

Le *pattern* MVC désigne une certaine façon d'organiser l'architecture du code d'une application. C'est un *pattern* qui se prête bien à la conception d'applications dotées d'une interface graphique, et tout particulièrement à la conception d'applications de bureau. La Figure 5 présente l'architecture du code dans ce *design pattern*.[1]



MVC Strict :
(1) Récupération de l'événement Swing par le controleur
(2) Répercutions locale directe sur la vue sans exploitation du modèle
(3) Déclenchement d'un traitement pour le modèle
(4) Notification du modèle pour déclencher une mise à jour graphique
(5) Consultation de la vue pour réaliser la mise à jour Graphique

Application Calculette :
(1) récupération clic sur bouton de calculette
(2) construction de l'expression dans la vue (1,2...9, (,))
(3) déclenchement calcul (=)
(4) Calcul terminé, notification de la vue
(5) La vue consulte le résultat et l'affiche

Figure 5: Schéma du MVC Strict

Dans le *pattern* MVC, les classes sont organisées en trois grandes partie : le modèle, la vue et le contrôleur. La vue contient les classes qui se chargent de la gestion de l'affichage, de toute la partie "graphique" de l'application si l'on veut. Le modèle quant à lui se charge de ce qu'on nomme les "traitements métier", un terme technique pour désigner la logique interne de l'application. Par exemple, dans une application de traitement de texte, le modèle contiendrait l'ensemble des objets "Page", et "Texte", et la vue se chargerait d'interroger le modèle pour récupérer ces informations et les afficher à l'écran. Le contrôleur quant à lui fait office d'intermédiaire : c'est par exemple lui qui va se charger de récupérer les entrées clavier et les clics souris. Ces entrées (clavier et souris) sont ensuite envoyées au modèle afin qu'il puisse les prendre en compte. Prenons comme exemple le raccourci "Ctrl+B" qui permet d'activer le "Gras" sur un texte. Les actions suivantes seront réalisées dans un contexte MVC :

1. Récupération des pressions de touches dans le contrôleur[2].

[1] Nous présentons ici MVC strict. Dans MVC non strict certaines interactions supplémentaires sont permises entre les différents composants.

[2] Pour les plus curieux d'entre vous, le contrôleur est capable de récupérer les informations du clavier en enregistrant une fonction de rappel dans le noyau du système d'exploitation pour que celui-ci lui remonte les informations d'entrées clavier

2. Le contrôleur appelle la méthode "onCompositeInput" (par exemple) dans la bonne classe du *package* "modèle" qui se charge de traiter les combinaisons de touches et lui passe en paramètre les touches pressées. N'oubliez pas que le modèle peut contenir plusieurs classes !

3. Le modèle traite la combinaison de touches, par exemple en regardant les différentes combinaisons qu'il connaît et se met à jour pour prendre en compte que le texte est désormais en gras.

4. Le modèle envoie un signal à la vue pour lui indiquer qu'il faut effectuer une mise à jour. En effet, l'utilisateur a (dés)activé le mode de saisie "Gras", il faut donc que le bouton change de façon à passer en mode "appuyé" (ou "non appuyé"). Comme il s'agit d'une mise à jour de l'interface graphique, le modèle délègue cette opération à la vue.

Cette explication diffère légèrement du schéma fourni en Figure 5. Il n'y a pas d'interaction du Contrôleur vers la Vue, mais on pourrait imaginer qu'il y en ait. Ce qui est important dans MVC est de respecter les interactions : le Contrôleur peut interagir directement avec la Vue, mais la Vue n'interagit pas *directement* avec le Contrôleur. Le Modèle n'interagit ni avec le Contrôleur, ni avec la Vue, mais le Contrôleur et la Vue ont tous les deux le droit d'interagir avec le modèle. En d'autres termes, la Vue et le Contrôleur peuvent avoir des références vers le Modèle, mais le Modèle ne doit *jamais* disposer d'une référence sur la Vue ou le Contrôleur. De la même façon, la Vue ne dispose jamais d'une référence vers le Contrôleur, mais le Contrôleur peut disposer d'autant de références vers la Vue qu'il le souhaite.

La notification du Modèle vers la Vue passe en général par le *pattern* Observer. Un rappel sur le *pattern* Observer, que vous devrez utiliser, est disponible en annexe B.

B Rappel *Pattern* Observer

Le *design pattern* Observer, également appelé Observer / Observable, est un *pattern* souvent utilisé dans la conception d'applications interactives. L'idée est qu'un objet "Observer" va "surveiller" des entités "Observable", et ces entités observées peuvent signaler à leurs surveillants qu'ils ont été mis à jour.

Dans le contexte de Java, une classe qui hérite d'`Observable` dispose d'une méthode `void notifyObservers(Object arg)`, et une classe qui implémente l'interface `Observer` dispose d'une méthode `void update(Observable o, Object arg)`. Lorsqu'un `Observable` appelle sa méthode `notifyObservers`, avec un paramètre `arg`, pour chaque `Observer` qui observe cet `Observable`, la méthode `update` est appelée avec en paramètres l'`Observable` et `arg`.

Si on reprend l'exemple de l'application de traitement de texte de l'Annexe A, on peut développer le déroulé de l'action 4. Une implémentation possible serait la suivante : le Modèle expose un Observable `TextFormat`, surveillé dans la Vue par un `TextFormatPanel`. Cet Observable contient le fameux champ `textFormat` évoqué précédemment. Lorsqu'un changement de mise en forme du texte est effectué, le `TextFormat` appelle sa méthode `notifyObservers` qui va appeler la méthode `update` du `TextFormatPanel` en lui passant en paramètre une liste des différents formats de texte et leur activation (italique, gras, souligné, barré...). La méthode `update` évalue le contenu de cette liste pour basculer les différents boutons associés dans le bon état (appuyé, non appuyé). Ainsi, il n'y a pas de lien direct entre le Modèle et la Vue.

Selon vous, pourquoi est-il intéressant de ne pas placer un lien direct entre le Modèle et la Vue ?

Remarque : les classes `Observer` et `Observable` ont été dépréciées (*Deprecated* depuis la version 9) en Java, car elles n'étaient plus considérées comme suffisamment flexibles. D'autres classes ont été proposées en remplacement, plus compliquées à utiliser mais offrant plus de fonctionnalités en échange. Toutefois, pour ce que nous faisons ici, les classes `Observer` et `Observable` suffiront largement : vous pouvez donc ignorer le *warning* que le compilateur vous fera.