

## Le modèle PAC :

Afin de réaliser cette application, nous avons choisi d'utiliser l'implémentation combinant le modèle *PAC*, le *patern Proxy* ainsi que l'*Héritage*. Cela nous a permis de concevoir et de développer une application facilement maintenable et évolutive, notamment grâce au modèle *PAC* qui sépare les couches applicatives; et avec une dépendance minimal entre les composants grâce au patron *Proxy* ainsi qu'à l'utilisation du patron *Abstract Factory*.

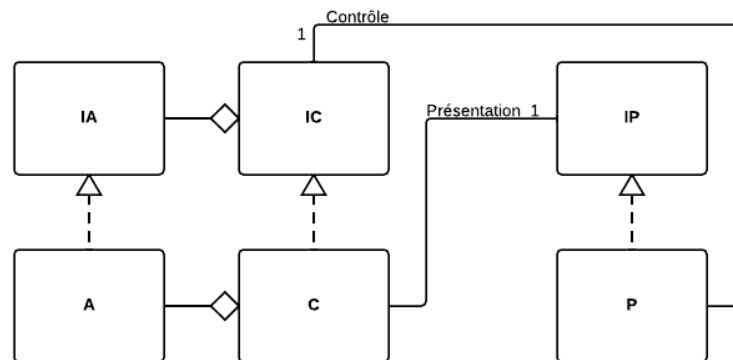


Schéma du modèle choisi

## Ce qui marche :

Les fonctionnalités qu'offre l'application sont les suivantes :

- Il est possible de jouer au solitaire, les règles sont bien respectées,
- Le gestion du Drag'n'Drop fonctionne parfaitement,
- L'application utilise différents retours sémantiques :
  - Changement du curseur de souris et du contour de la carte afin de prévenir l'utilisateur qu'il peut la sélectionner,
  - Changement de la couleur de fond ou du contour des tas de cartes afin de prévenir l'utilisateur si l'action à réaliser est possible ou non.

## Ce qui ne marche pas:

Par manque de temps, il est cependant pas possible de jouer au Solitaire en mode console.

## Améliorations et évolutions :

L'application réalisée pourrait être améliorée afin de :

- élargir la gestion du Drag'n'Drop afin de pouvoir jouer au Solitaire entre plusieurs JVM. Pour cela, il suffit d'apporter des modifications à la classe *solitaire.dnd.MyDragGestureListener.java* et *solitaire.dnd.MyDropTargetListener.java* afin de permettre cette gestion et notamment grâce à l'ajout des *flavors* adéquats.
- améliorer la gestion du retour sémantique pour la rendre plus réactive et optimisée et surtout quant à son intégration avec la gestion du Drag'n'Drop,
- enrichir le menu pour pouvoir relancer une partie sans pour autant relancer l'application.

## Choix d'implémentation :

### Le retour sémantique

La gestion du retour sémantique se fait grâce à l'utilisation de l'interface *solitaire.feedback.Feedbackable.java* qui offre les différentes méthodes nécessaires à tout composant voulant implémenter sa propre façon de retourner une information à l'utilisateur, autrement-dit chaque composant gère séparément son retour sémantique. Afin de faciliter l'intégration avec la gestion du Drag'n'Drop, la classe *solitaire.feedback.Feedback.java* permet de centraliser les appels aux différents composants utilisant cette fonctionnalité de retour sémantique.

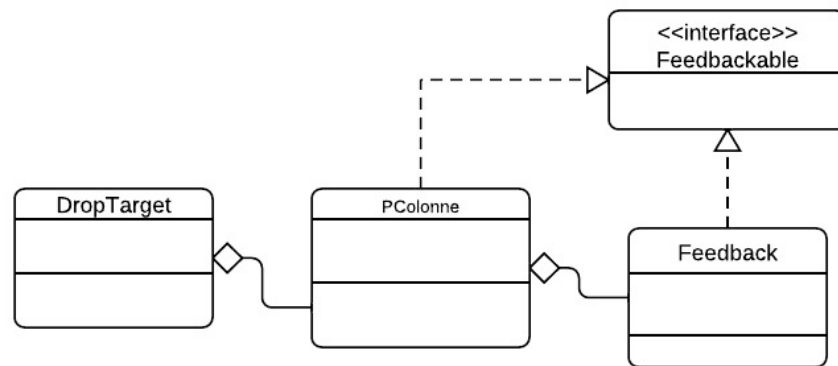


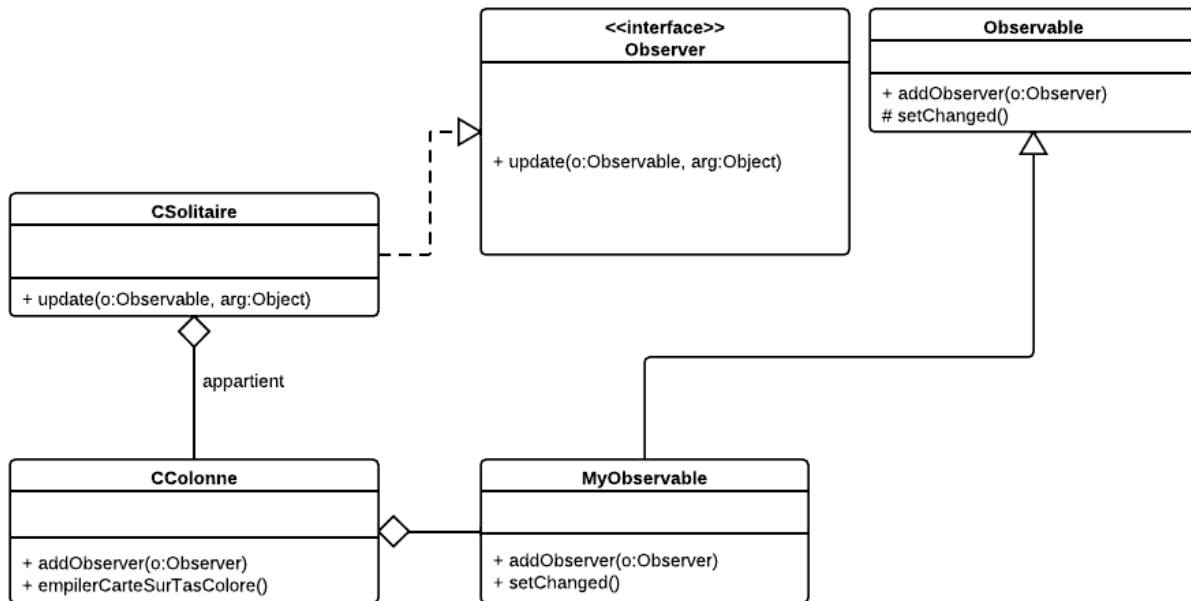
Diagramme UML simplifié de la gestion du Feedback

### Utilisation du pattern Observer

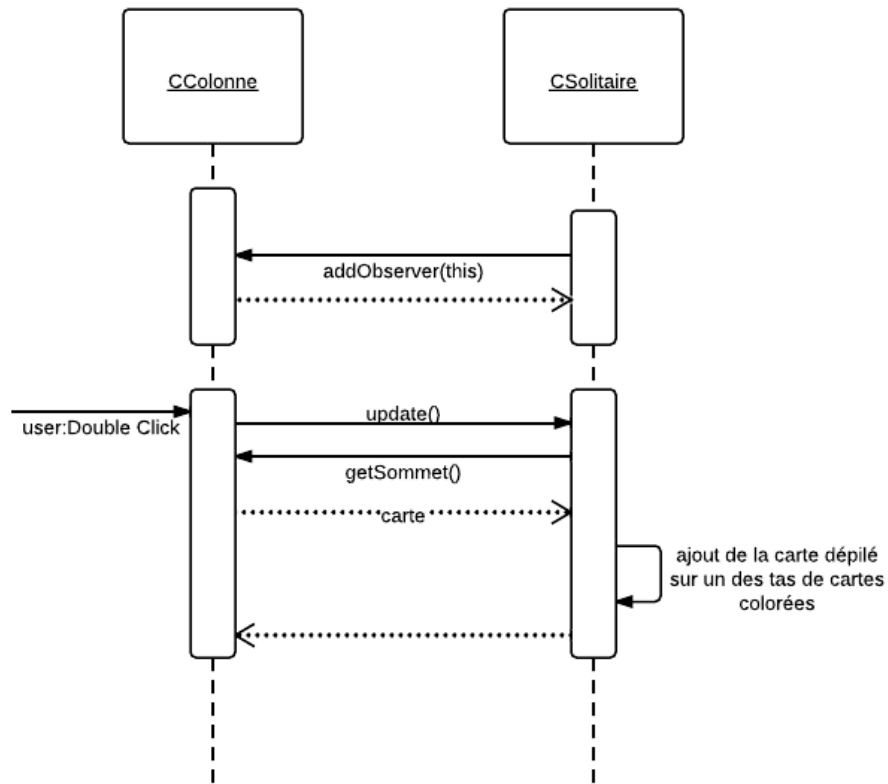
Nous avons été obligé, à certains endroits, d'utiliser un pattern *observer* afin de réaliser certaines actions. Par exemple lors d'un double click sur une colonne, afin d'empiler la dernière carte sur un des tas de cartes colorées (si la carte est empilable, bien sûr).

La classe *CSolitaire* joue le rôle d'*Observer*, et les classes *CColonne*, d'*Observable*. Le solitaire va s'ajouter en tant qu'observateur sur toutes les colonnes en appelant la méthode *addObserver* de celles-ci.

Voici le diagramme de classe :



Et le diagramme de séquence :



Un des problèmes rencontrés est que le contrôleur doit hériter de la classe *Observable* ainsi que de l'abstraction. L'héritage multiple n'étant pas possible en Java, nous avons dû utiliser une délégation. La classe *MyObservable* joue ce rôle. Il n'était pas possible d'utiliser une instance de *Observable* directement, car sa méthode *setChanged*, qui est indispensable, a une visibilité *protected*. Nous avons donc dû redéfinir cette méthode dans la classe *MyObservable* afin d'augmenter sa visibilité.