

L3 Info – INF5A1

Jour 7

le 18 octobre 2021

Design Patterns

(suite)

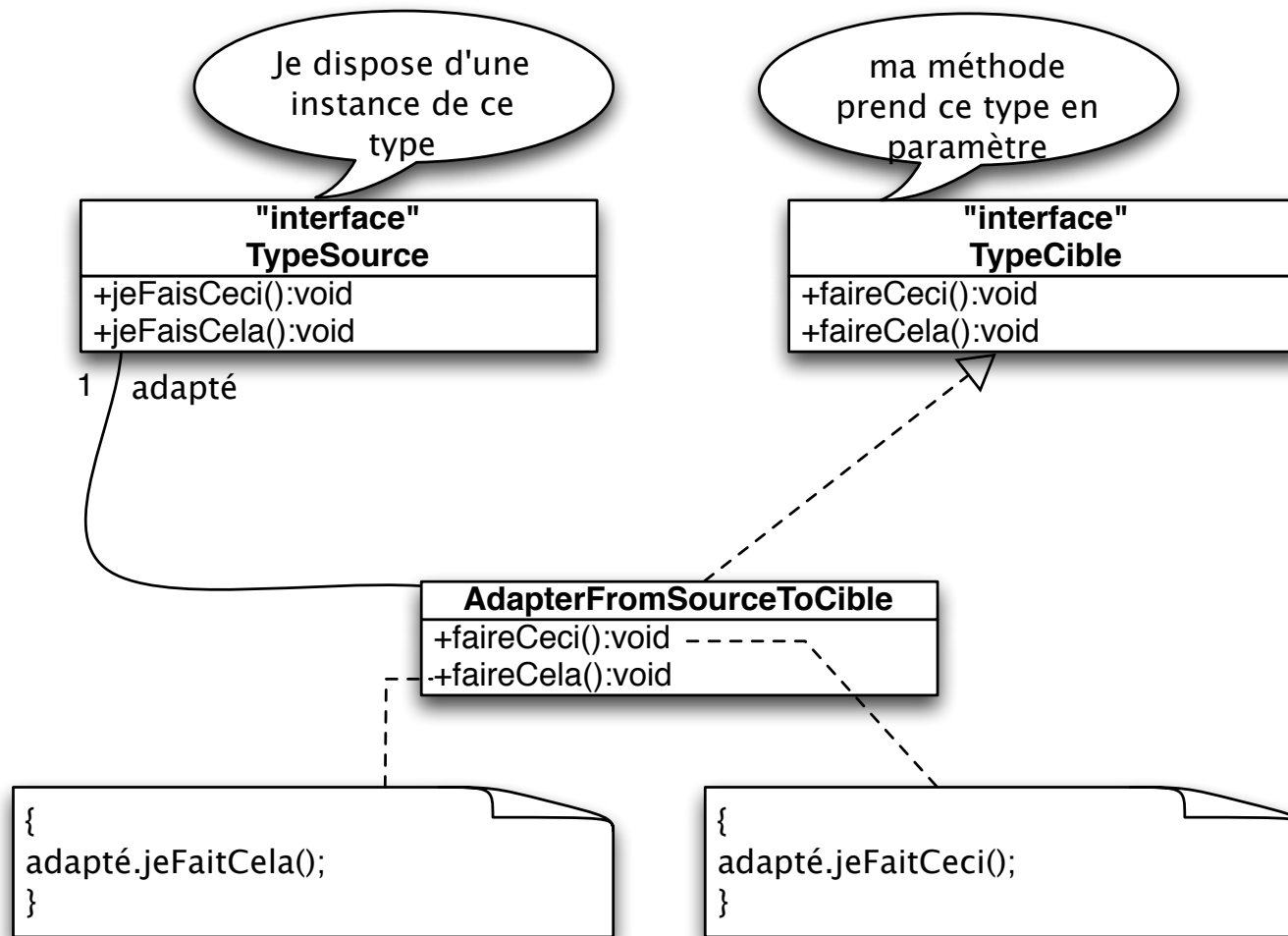
# Rappel : Types de Patterns

- De **construction** (Singleton, Abstract Factory), permettant de déléguer la construction d'instances à un acteur particulier.
- De **structuration** (Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy)
- De **comportement** (Chain of Responsibility, Command, Interpreter, Iterator, Memento, Observer, State, Strategy, Template Method)

# Adapter (d'instance)

- Problème : on souhaite soumettre une instance à un traitement qui prend en entrée un type (une interface) différent (et non compatible par polymorphisme).
- Solution : utiliser une instance d'une classe compatible avec le type cible, et qui va représenter notre objet. C'est un adapter (adaptateur en français).
- L'adapter va se faire passer pour notre objet et traduire les méthodes du type cible vers notre objet.

# Diagramme d'adapter



# Exemple : adapter un objet à une JTable

- JTable, composant swing dessinant un tableau de données, s'appuie sur l'interface modèle TableModel. Cette dernière dispose notamment des méthodes :
  - [getValueAt](#)(rowIndex, : int, columnIndex : int) : Object
  - [getRowCount](#)() : int
  - [getColumnCount](#)() : int
- On dispose d'un objet du type DiscCatalog qui possède les méthodes :
  - getDiscCount() :int
  - getDisc(i : int) : DiscEt un Disc dispose de getAuthor(), getTitle(), getYear()
- Comment représenter un Catalog sous forme de JTable ?

# Avantages

- Si l'on dupliquait les données de notre Catalog dans une classe spécifique qui implémente TableModel, on aurait un problème lors de la mise à jour du Catalog : il faudrait aussi mettre à jour les données dupliquées
- En utilisant ce pattern, les données sont stockées uniquement dans l'objet initial, et traduites à la volée vers la JTable, donc la JTable est toujours à jour.
- On gagne en temps d'exécution, en place mémoire, et surtout en facilité de conception.

# Attention : penser à adapter aussi les événements...

- Quand notre Catalog est modifié, comment la JTable sait-elle qu'elle doit se redessiner ?
- JTable écoute TableModel, qui dispose de méthodes événementielles telles que `fireTableDataChanged()`.
- Il faut donc adapter aussi les événements : notre adapter doit écouter l'objet adapté, et lorsque ce dernier le prévient qu'il a changé, il doit prévenir sa JTable en lançant `fireTableDataChanged()`.

# Variante : Adapter de classe

- Il existe un second pattern nommé Adapter de classe. Son intérêt est très limité.
- Il consiste à créer une classe qui implémente le type cible (comme dans le pattern Adapter d'objet), mais qui, au lieu d'être lié par association à l'objet adapté, hérite de sa classe (ou implémente son interface).
- Il ne peut donc pas adapter un objet déjà créé via le type initial ou l'un de ses sous-types. Par exemple, pour avoir un Catalog visualisable dans une JTable, il faudrait le créer directement via la classe CatalogAdapter.



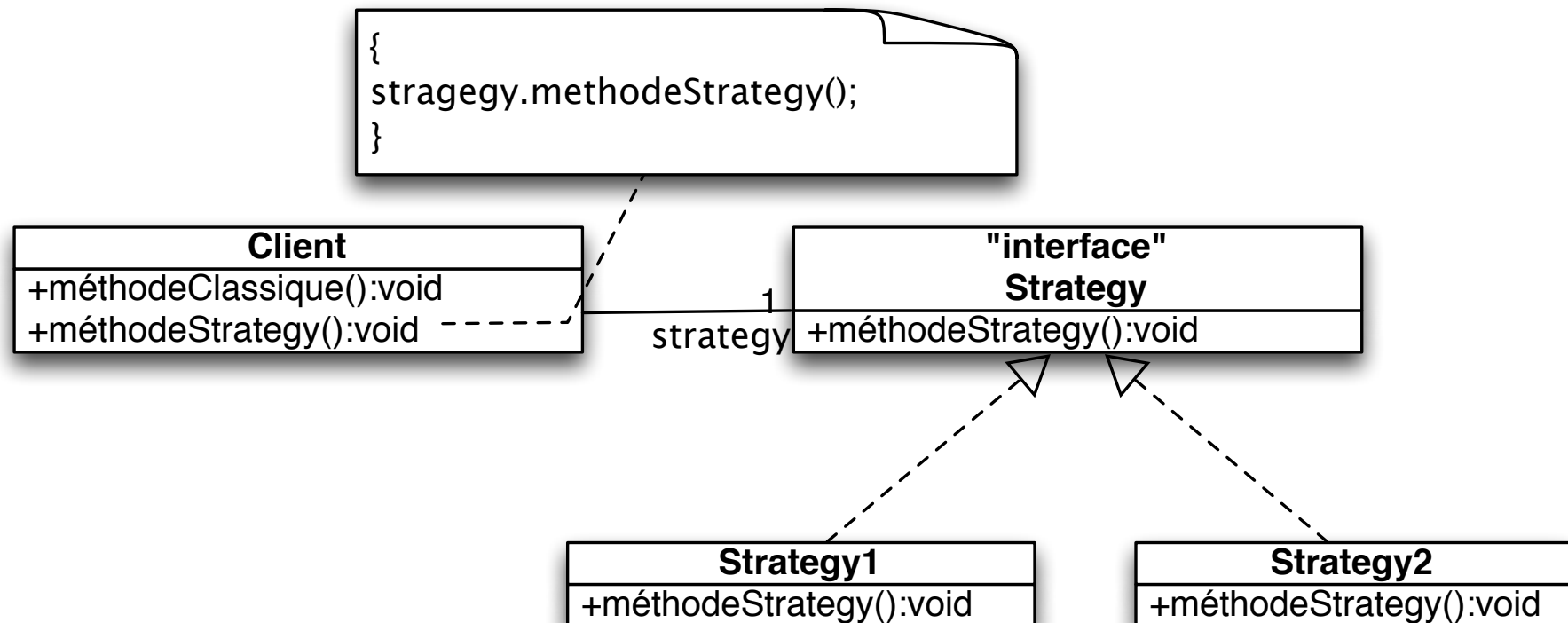
# Strategy

- Problème : on veut externaliser le contenu de certaines méthodes pour pouvoir modifier les comportements soit dynamiquement (sans modifier la classe, à l'exécution) soit en utilisant de nouvelles implémentations à l'avenir
- On peut ainsi par exemple proposer de nouveaux algorithmes sans recompiler le code client du pattern, proposer des variantes de comportement de certains objets à l'infini, etc.

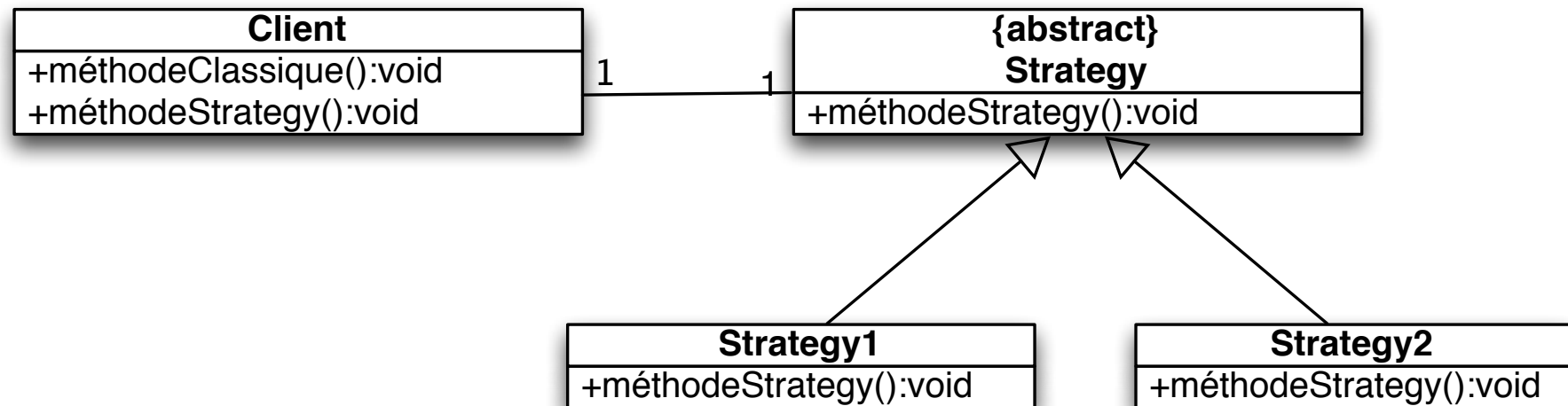
# Principe de Strategy

- Créer une interface déclarant les méthodes que l'on souhaite externaliser. Même si ce n'est pas formellement obligatoire, on leur donne en général le même nom que les méthodes à externaliser.
- Mettre un attribut du type de l'interface dans le client. Prévoir que le constructeur ou un setter permette d'attribuer ou de changer de la stratégie
- Pour toutes les méthodes à externaliser, utiliser la délégation vers l'instance de Strategy.

# Diagramme de Strategy



# Version Classe Abstraite pour implémenter un callback vers le client



# Exemple 1 : Strategy et les Layouts

## Awt/Swing

- Nous savons que tout Container AWT ou Swing utilise un LayoutManager pour définir la disposition de son contenu.
- Ce dernier n'est en fait rien d'autre qu'une interface de Strategy, et les différents Layouts sont des implémentations de cette interface.
- Lorsque le container se voit ajouter des composants, il délègue leur placement à son instance de LayoutManager (sa stratégie...)
- On peut créer de nouveaux LayoutManagers par implémentation ou dérivation, et les utiliser avec du code Swing standard. Ceci ne serait pas possible si ce pattern n'avait pas été utilisé nativement.

## Exemple 2 : rappel avec SAX

- Un parseur SAX (Simple Api for XML) est une instance qui est capable d'interpréter un flux XML et de créer les événements associés (balise ouvrante, balise fermante, etc.)
- Ces événements sont transmis à une instance appelée ContentHandler, qui est finalement une stratégie d'action.
- Pour définir un traitement XML particulier, il suffit de créer une implémentation spécifique de ContentHandler (une stratégie particulière...)

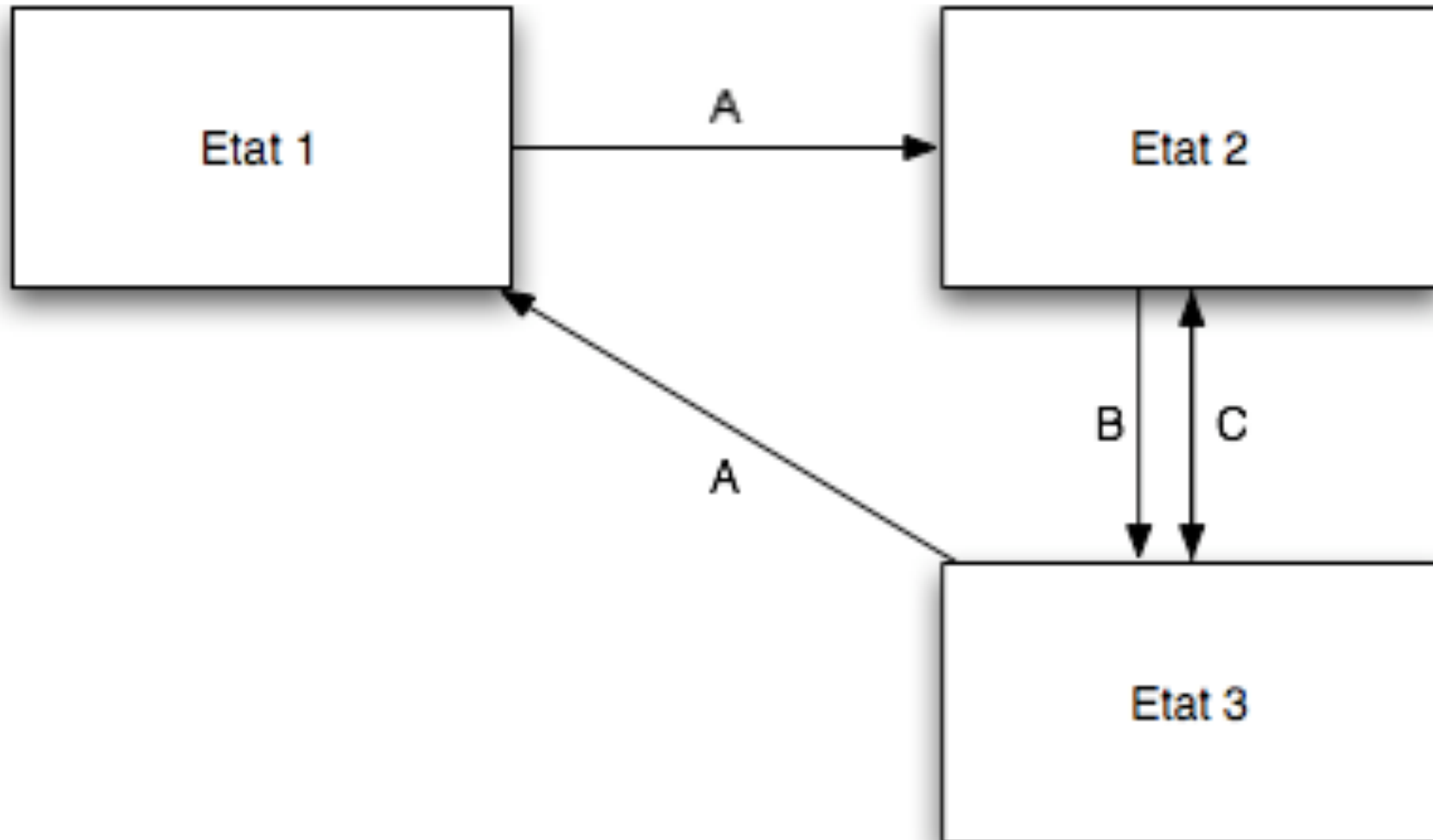
# Cas particulier de Strategy :

## Le pattern State

- Faire en sorte qu'un objet dispose d'un attribut « état », et faire en sorte que son comportement qui varie selon son état actuel.
- Proposer un setter sur l'état pour pouvoir changer dynamiquement d'état.
- C'est un cas particulier de Strategy dans lequel le client du pattern crée lui-même, dès sa création, ses différentes stratégies correspondant à chacun de ses états, ainsi qu'une méthode permettant de changer d'état

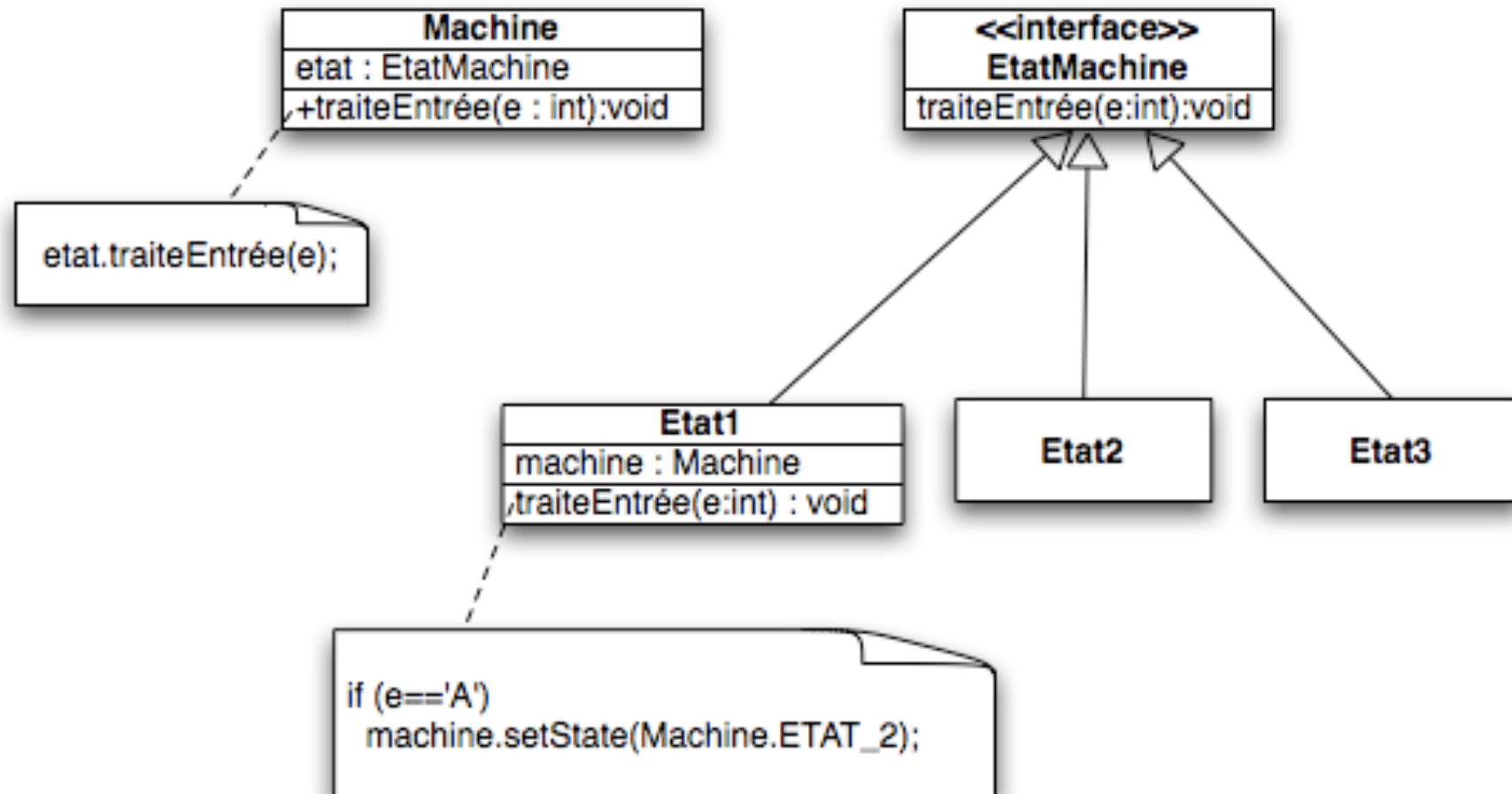
# Exemple d'application de State :

## Automate à états finis





# Diagramme de classes associé



# Suggestions de réalisations en TP pour le devoir de CC

- Adapter : en reprenant ce qui aura été vu en TP aujourd'hui, possibilité de tenir à jour une liste des personnages présents, avec certaines de leurs caractéristiques (nom, niveau d'énergie, armes)
- Strategy : pour mettre en place différentes stratégies possibles d'un joueur artificiel