Architecture logicielle

Jean-Claude Royer

3 novembre 2010

Définition

Définition

Introduction à UML

Architecture logicielle

- Abstraire des informations structurelles, organisationnelles, fonctionnelles

Architecture logicielle

- Abstraire des informations structurelles, organisationnelles, fonctionnelles
- Utiliser plusieurs points de vue complémentaires

Architecture logicielle

- Abstraire des informations structurelles, organisationnelles, fonctionnelles
- Utiliser plusieurs points de vue complémentaires
- Organiser le système d'une façon "simple", "lisible"

Architecture logicielle

- Abstraire des informations structurelles, organisationnelles, fonctionnelles
- Utiliser plusieurs points de vue complémentaires
- Organiser le système d'une façon "simple", "lisible"
- Guide pour l'analyse et la conception

Architecture logicielle

- Abstraire des informations structurelles, organisationnelles, fonctionnelles
- Utiliser plusieurs points de vue complémentaires
- Organiser le système d'une façon "simple", "lisible"
- Guide pour l'analyse et la conception
- Améliorer la compréhension du système

Architecture logicielle

- Abstraire des informations structurelles, organisationnelles, fonctionnelles
- Utiliser plusieurs points de vue complémentaires
- Organiser le système d'une façon "simple", "lisible"
- Guide pour l'analyse et la conception
- Améliorer la compréhension du système
- Indispensable pour maintenir voire re-concevoir le système

Architecture logicielle

- Abstraire des informations structurelles, organisationnelles, fonctionnelles
- Utiliser plusieurs points de vue complémentaires
- Organiser le système d'une façon "simple", "lisible"
- Guide pour l'analyse et la conception
- Améliorer la compréhension du système
- Indispensable pour maintenir voire re-concevoir le système
- Un petit exemple pour illustrer

Architecture logicielle

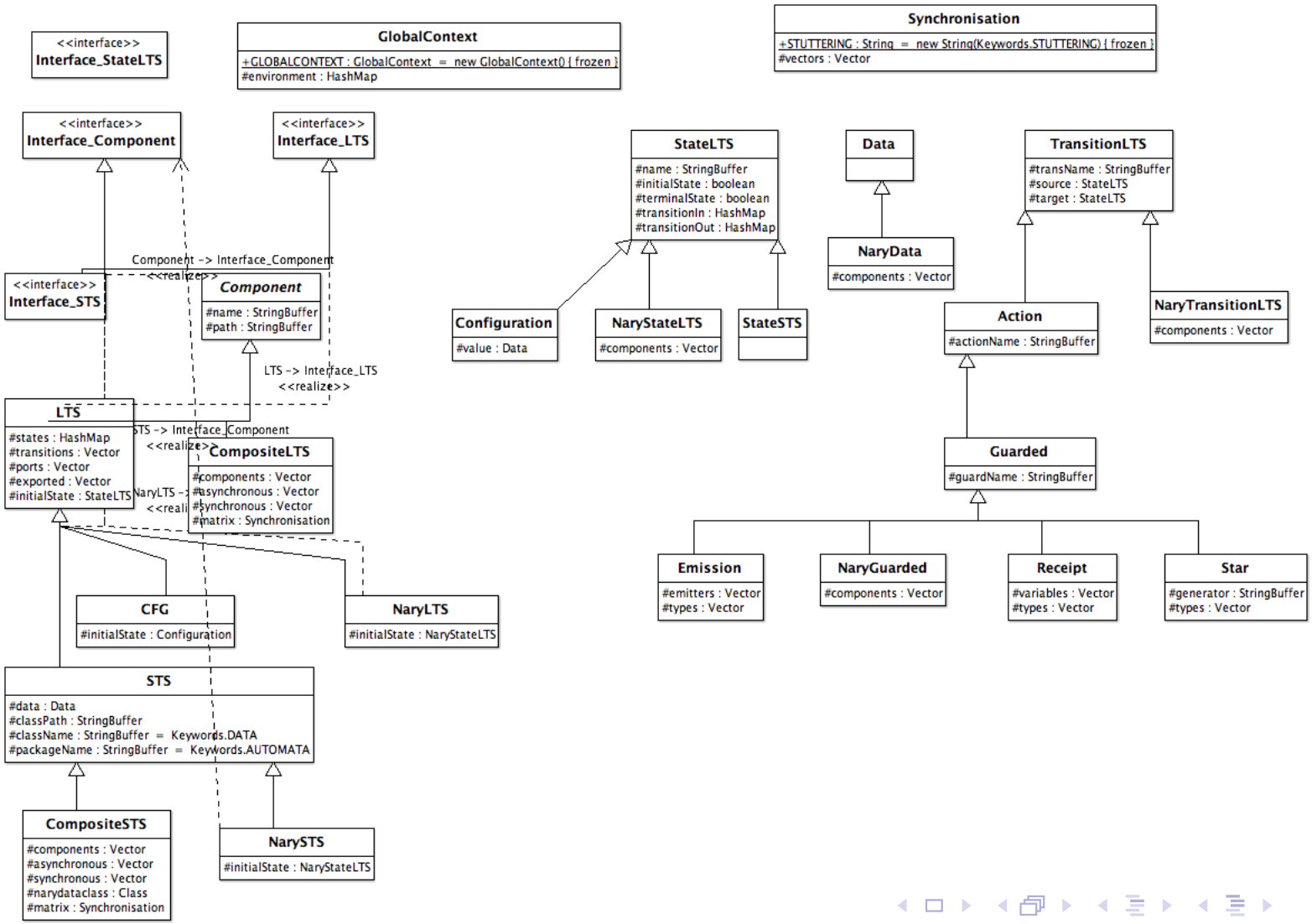
- Abstraire des informations structurelles, organisationnelles, fonctionnelles
- Utiliser plusieurs points de vue complémentaires
- Organiser le système d'une façon "simple", "lisible"
- Guide pour l'analyse et la conception
- Améliorer la compréhension du système
- Indispensable pour maintenir voire re-concevoir le système
- Un petit exemple pour illustrer
- Schéma électronique, dessin technique, plan de construction, ...

Architecture logicielle

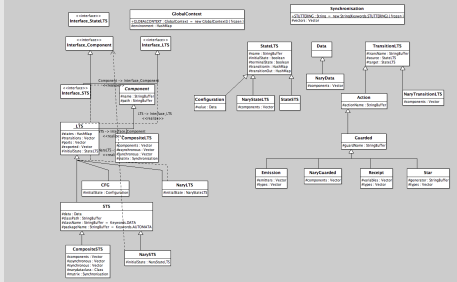
- Abstraire des informations structurelles, organisationnelles, fonctionnelles
- Utiliser plusieurs points de vue complémentaires
- Organiser le système d'une façon "simple", "lisible"
- Guide pour l'analyse et la conception
- Améliorer la compréhension du système
- Indispensable pour maintenir voire re-concevoir le système
- Un petit exemple pour illustrer
- Schéma électronique, dessin technique, plan de construction, ...

On prendra une version simple du jeu de Nim, nous avons trois catégories de joueur. Une partie avec deux joueurs et choix du nombre d'allumettes manuellement ou aléatoirement. Après une présentation du fonctionnement utilisateur comment faire pour expliquer la conception de ce système ?

Un diagramme de classe

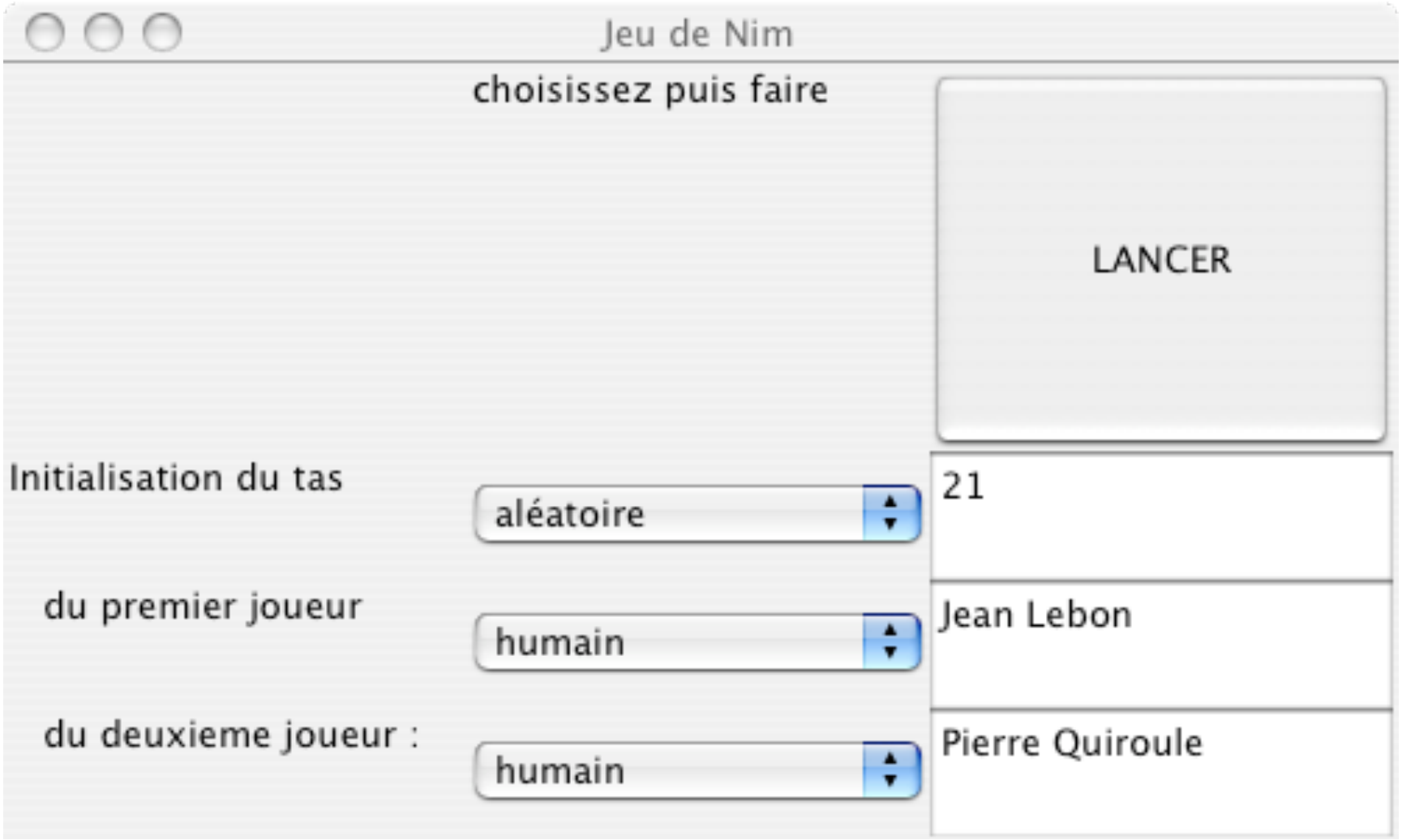


Un diagramme de classe

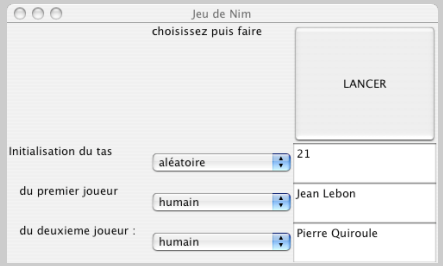


Voila une vue possible d'un système complexe, on va illustrer les principaux éléments sur l'exemple précédents d'applications.

Vue principale

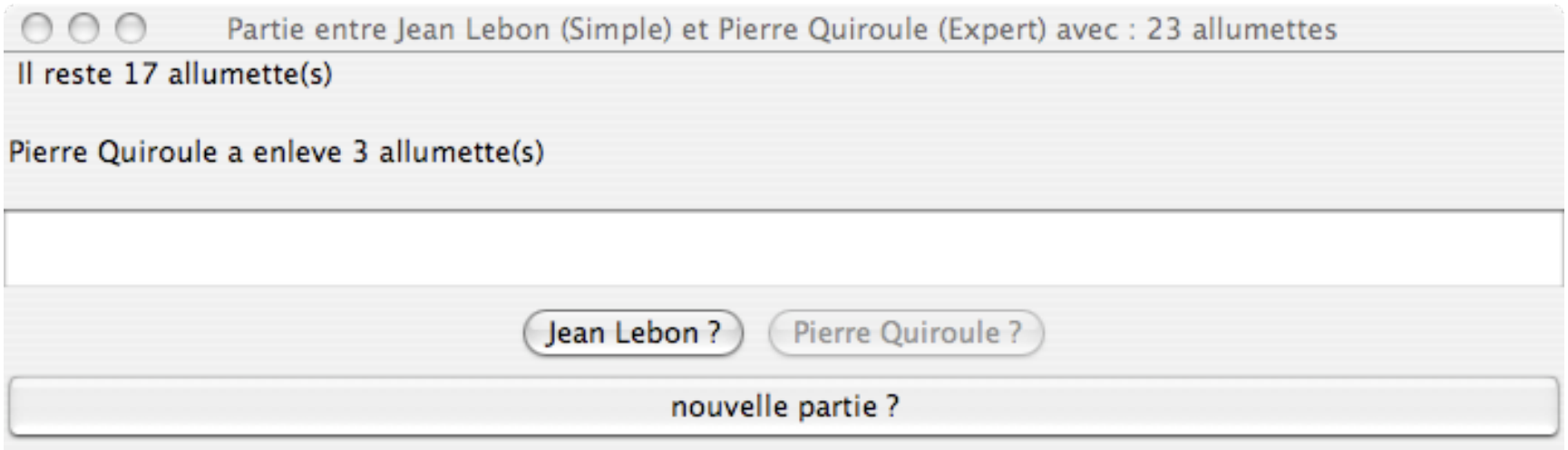


Vue principale

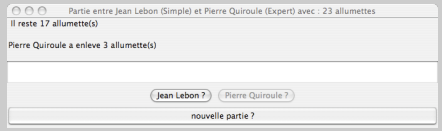


Le joueur doit faire les choix du type d'initialisation du tas ainsi que la catégorie des joueurs (Humain, Simple, Expert). Il peut également proposé des valeurs en remplacement de celles par défaut. La sortie se fait par le bouton de sortie de la fenêtre. Le bouton LANCER lance la partie qui se fait par l'intermédiaire d'une autre fenêtre.

Vue d'une partie



Vue d'une partie



Cette nouvelle fenêtre affiche les informations relatives au déroulement de la partie. Un bouton est associé à chacun des joueurs et ne s'utilisent qu'en alternance. Lorsque la partie est finie le bouton nouvelle partie permet de retourner à la fenêtre d'initialisation et ferme la fenêtre courante. On peut interrompre la partie en fermant la fenêtre.

Les notions et les outils

- Quelques règles de conception

Les notions et les outils

- Quelques règles de conception
- Notion d'interface

Les notions et les outils

- Quelques règles de conception
- Notion d'interface
- Petite introduction à UML

Les notions et les outils

- Quelques règles de conception
- Notion d'interface
- Petite introduction à UML
- Organisation statique d'un système

Les notions et les outils

- Quelques règles de conception
- Notion d'interface
- Petite introduction à UML
- Organisation statique d'un système
- Notion de communications et de dynamique

Les notions et les outils

- Quelques règles de conception
- Notion d'interface
- Petite introduction à UML
- Organisation statique d'un système
- Notion de communications et de dynamique
- Langage d'architecture

Recommandations pour la conception

- Une classe concrète doit définir des attributs privés et des accesseurs public, un constructeur par défaut

Recommandations pour la conception

- Une classe concrète doit définir des attributs privés et des accesseurs public, un constructeur par défaut
- La fonction principale `main` doit être placée dans une classe à part

Recommandations pour la conception

- Une classe concrète doit définir des attributs privés et des accesseurs public, un constructeur par défaut
- La fonction principale `main` doit être placée dans une classe à part
- Ne pas mélanger méthodes fonctionnelle et procédurale :

Recommandations pour la conception

- Une classe concrète doit définir des attributs privés et des accesseurs public, un constructeur par défaut
- La fonction principale `main` doit être placée dans une classe à part
- Ne pas mélanger méthodes fonctionnelle et procédurale :
  - Fonction : retourne une valeur pas de modifications des paramètres y compris l'objet receveur

Recommandations pour la conception

- Une classe concrète doit définir des attributs privés et des accesseurs public, un constructeur par défaut
- La fonction principale `main` doit être placée dans une classe à part
- Ne pas mélanger méthodes fonctionnelle et procédurale :
  - Fonction : retourne une valeur pas de modifications des paramètres y compris l'objet receveur
  - Procédure : pas de valeur de retour et modifie uniquement l'état de l'objet receveur

Recommandations pour la conception

- Une classe concrète doit définir des attributs privés et des accesseurs public, un constructeur par défaut
- La fonction principale `main` doit être placée dans une classe à part
- Ne pas mélanger méthodes fonctionnelle et procédurale :
  - Fonction : retourne une valeur pas de modifications des paramètres y compris l'objet receveur
  - Procédure : pas de valeur de retour et modifie uniquement l'état de l'objet receveur
- Respecter les conventions d'écriture, notamment de nommage pour les classes, variables et méthodes

Recommandations pour la conception

- Une classe concrète doit définir des attributs privés et des accesseurs public, un constructeur par défaut
- La fonction principale `main` doit être placée dans une classe à part
- Ne pas mélanger méthodes fonctionnelle et procédurale :
  - Fonction : retourne une valeur pas de modifications des paramètres y compris l'objet receveur
  - Procédure : pas de valeur de retour et modifie uniquement l'état de l'objet receveur
- Respecter les conventions d'écriture, notamment de nommage pour les classes, variables et méthodes
- ...

Recommandations pour la conception

- Une classe concrète doit définir des attributs privés et des accesseurs public, un constructeur par défaut
- La fonction principale `main` doit être placée dans une classe à part
- Ne pas mélanger méthodes fonctionnelle et procédurale :
  - Fonction : retourne une valeur pas de modifications des paramètres y compris l'objet receveur
  - Procédure : pas de valeur de retour et modifie uniquement l'état de l'objet receveur
- Respecter les conventions d'écriture, notamment de nommage pour les classes, variables et méthodes
- ...

Toujours commencé par savoir construire ses objets
puis ensuite savoir les afficher (String).

Finalement s'attaquer aux autres services en essayant de commencer par les plus simples d'abord.

Interface

- Uniquement les services publiques

Interface

- Uniquement les services publiques
- Profil d'une opération, *eg.*,
  `check : Joueur TextField Integer --> Integer`

Interface

- Uniquement les services publiques
- Profil d'une opération, *eg.*,
  `check : Joueur TextField Integer --> Integer`
- Nom + fonction ou procédure + types des arguments + type du résultat

Interface

- Uniquement les services publiques
- Profil d'une opération, *eg.*,
  `check : Joueur TextField Integer --> Integer`
- Nom + fonction ou procédure + types des arguments + type du résultat
- Signature d'un type abstrait de données

Interface

- Uniquement les services publiques
- Profil d'une opération, *eg.*,
  `check : Joueur TextField Integer --> Integer`
- Nom + fonction ou procédure + types des arguments + type du résultat
- Signature d'un type abstrait de données
- Documentation

Interface

- Uniquement les services publiques
- Profil d'une opération, *eg.*,
  `check : Joueur TextField Integer --> Integer`
- Nom + fonction ou procédure + types des arguments + type du résultat
- Signature d'un type abstrait de données
- Documentation
- Spécification (formelle ou informelle)

Attention le vocabulaire est varié : on parle de type ceci est justifié car une interface défini un type ; on parle aussi de type abstrait car l'implémentation n'est pas décrite, un terme équivalent est aussi celui de signature qui s'applique à la fois pour un profil et une interface. Fondamentalement, aux détails de présentation près, une signature c'est un nom d'opération, si c'est une fonction ou une procédure, les types des arguments et le type du résultat si besoin.

- `javap` permet d'avoir l'interface (et plus) d'une classe compilée
- API : Application Programming Interface
- Pré-post condition ou autres techniques formelles permettent de compléter ces descriptions un peu trop syntaxiques

Introduction à UML

- Langage de modélisation très utilisé en pratique

Introduction à UML

- Langage de modélisation très utilisé en pratique
- C'est une norme

Introduction à UML

- Langage de modélisation très utilisé en pratique
- C'est une norme
- Quelques éléments des diagrammes de classes et de séquences

Introduction à UML

- Langage de modélisation très utilisé en pratique
- C'est une norme
- Quelques éléments des diagrammes de classes et de séquences
- Pas très adapté aux architectures, un peu mieux avec la V 2.0

Introduction à UML

- Langage de modélisation très utilisé en pratique
- C'est une norme
- Quelques éléments des diagrammes de classes et de séquences
- Pas très adapté aux architectures, un peu mieux avec la V 2.0
- Le langage est très complexe nous voyons ici seulement une toute petite partie

Introduction à UML

- Langage de modélisation très utilisé en pratique
- C'est une norme
- Quelques éléments des diagrammes de classes et de séquences
- Pas très adapté aux architectures, un peu mieux avec la V 2.0
- Le langage est très complexe nous voyons ici seulement une toute petite partie
- Un développement plus complet est fait dans l'UV GL en GS1

- Langage de modélisation très utilisé en pratique
- C'est une norme
- Quelques éléments des diagrammes de classes et de séquences
- Pas très adapté aux architectures, un peu mieux avec la V 2.0
- Le langage est très complexe nous voyons ici seulement une toute petite partie
- Un développement plus complet est fait dans l'UV GL en GS1

Relativement à la notion d'architecture nous pouvons signaler :
En programmation modulaire, la notion de base est un module et on adopte le principe de *faible couplage et forte cohésion* ceci s'applique surtout aux classes et aux packages.
Il y a également des ADL ou langage de description d'architectures.

Organisation en packages

- Un package est un ensemble d'éléments (des classes ou d'autres packages)

Organisation en packages

- Un package est un ensemble d'éléments (des classes ou d'autres packages)
- Notion d'importation : le package `Solver` utilise/importe `Fourrier`

Organisation en packages

- Un package est un ensemble d'éléments (des classes ou d'autres packages)
- Notion d'importation : le package `Solver` utilise/importe `Fourrier`
- Notion de client - fournisseur : une entité (opération, module) a besoin d'une autre entité pour se réaliser

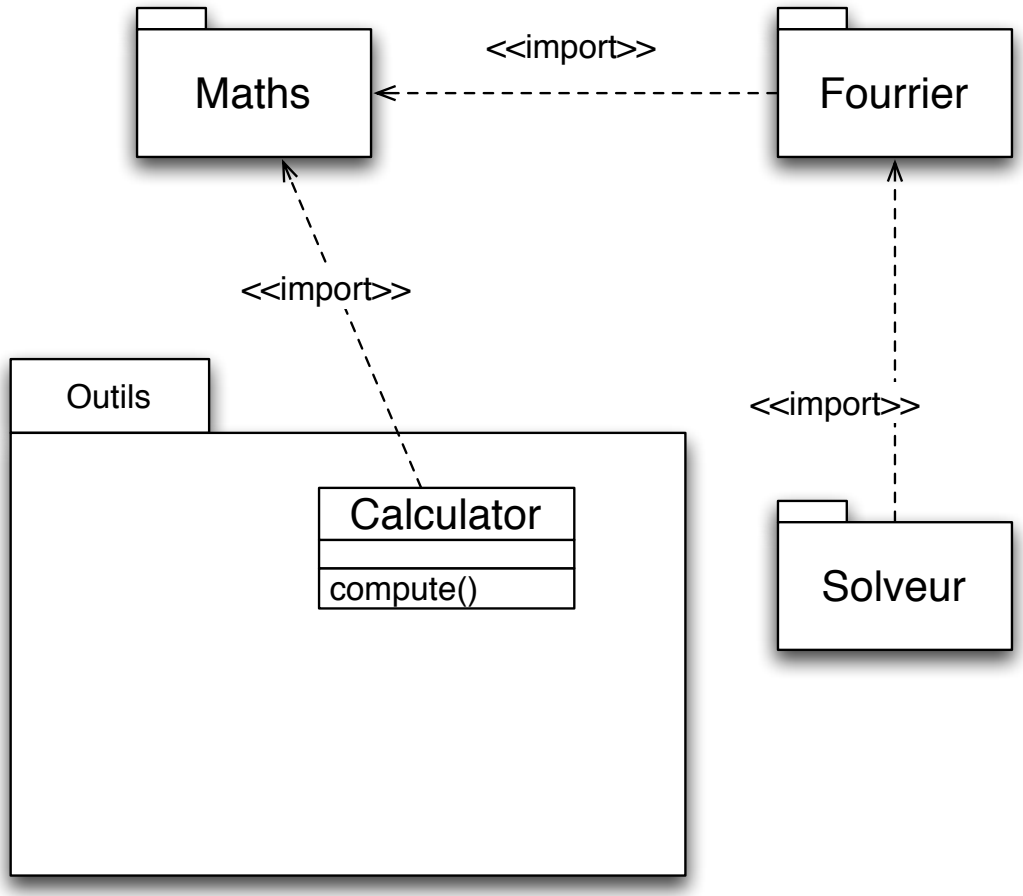
Organisation en packages

- Un package est un ensemble d'éléments (des classes ou d'autres packages)
- Notion d'importation : le package `Solver` utilise/importe `Fourrier`
- Notion de client - fournisseur : une entité (opération, module) a besoin d'une autre entité pour se réaliser
- La flèche en pointillée indique une dépendance en UML que l'on peut qualifier par un stéréotype si on veut en dire plus

Organisation en packages

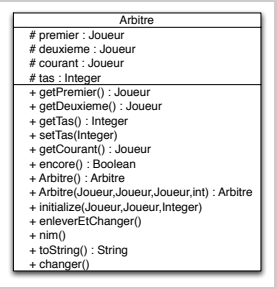
- Un package est un ensemble d'éléments (des classes ou d'autres packages)
- Notion d'importation : le package `Solver` utilise/importe `Fourrier`
- Notion de client - fournisseur : une entité (opération, module) a besoin d'une autre entité pour se réaliser
- La flèche en pointillée indique une dépendance en UML que l'on peut qualifier par un stéréotype si on veut en dire plus
- Donne une description statique pauvre mais utile par exemple pour ordonnancer les tâches de compilation des différents modules d'une application

Exemple de paquetage



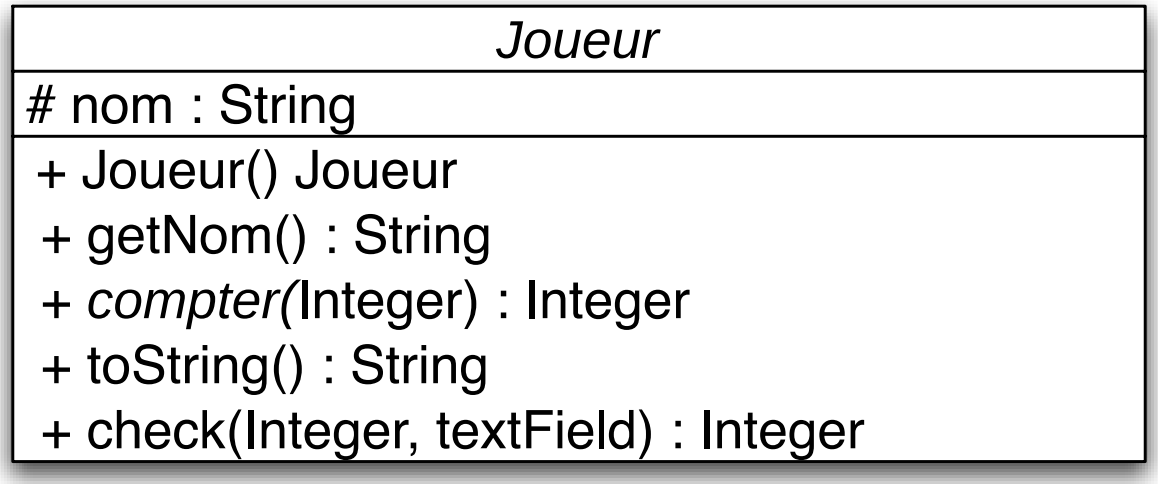
La classe Arbitre

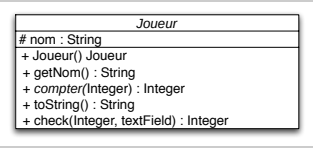
Arbitre
premier : Joueur # deuxieme : Joueur # courant : Joueur # tas : Integer
+ getPremier() : Joueur + getDeuxieme() : Joueur + getTas() : Integer + setTas(Integer) + getCourant() : Joueur + encore() : Boolean + Arbitre() : Arbitre + Arbitre(Joueur,Joueur,Joueur,int) : Arbitre + initialize(Joueur,Joueur,Integer) + enleverEtChanger() + nim() + toString() : String + changer()



Cette classe définit 4 attributs : les deux joueurs, le courant et le nombre d"allumettes. Il y a plusieurs services, certains pour l'accès aux attributs mais d'autres qui calculent des informations ou modifient l'état de l'objet courant. La description de l'interface est insuffisante pour montrer la sémantique d'une opération, elle donne la syntaxe uniquement. Cette description nous donne l'interface de la classe `Arbitre` mais aussi les parties publiques car ceux-ci ont une importance forte. Une *partie publique* est soit un champ publique ou protégé soit un champ privé pour lequel il existe un accesseur (lecture ou écriture) publique.

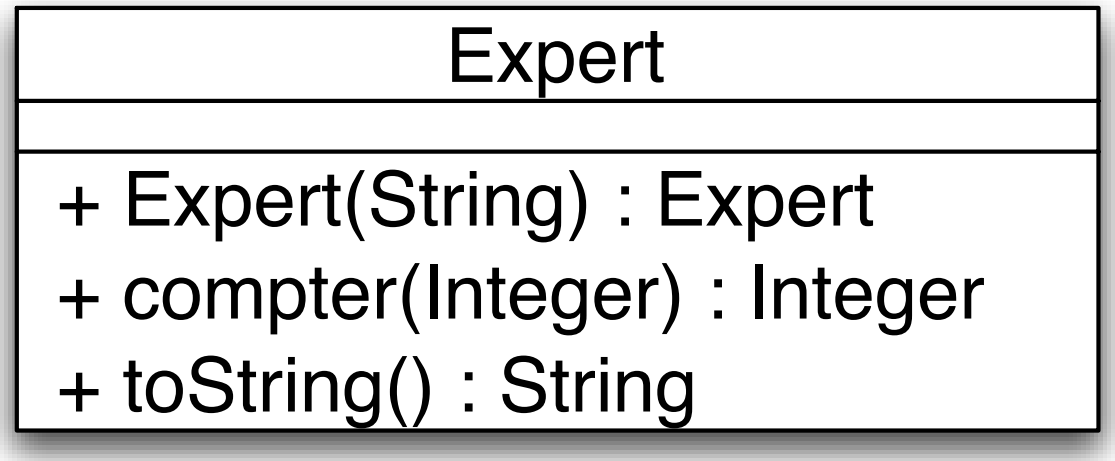
La classe Joueur

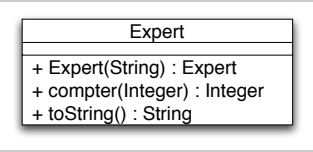




Classe abstraite en italique ou avec le stéréotype « abstract», notation similaire pour les méthodes.

La classe Expert

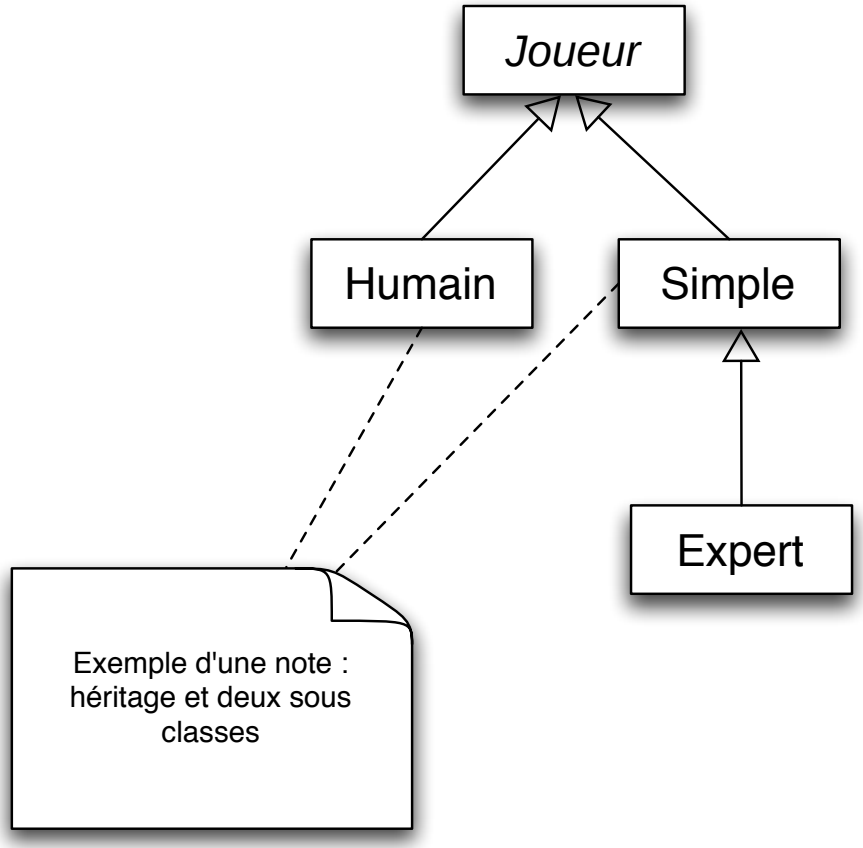


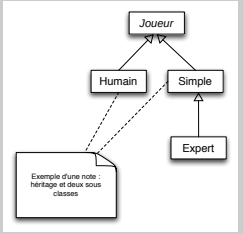


Sous-classe de `Joueur` (sera indiquée + loin) qui redéfinit certaines méthodes et un exemple de commentaire.
Les notations de portée sont :

- `#` : désigne un élément protégé en UML
- `+` : désigne un élément publique (le défaut)
- `-` : serait un élément privé donc peu de chance de le voir dans une interface avec seulement les services publiques

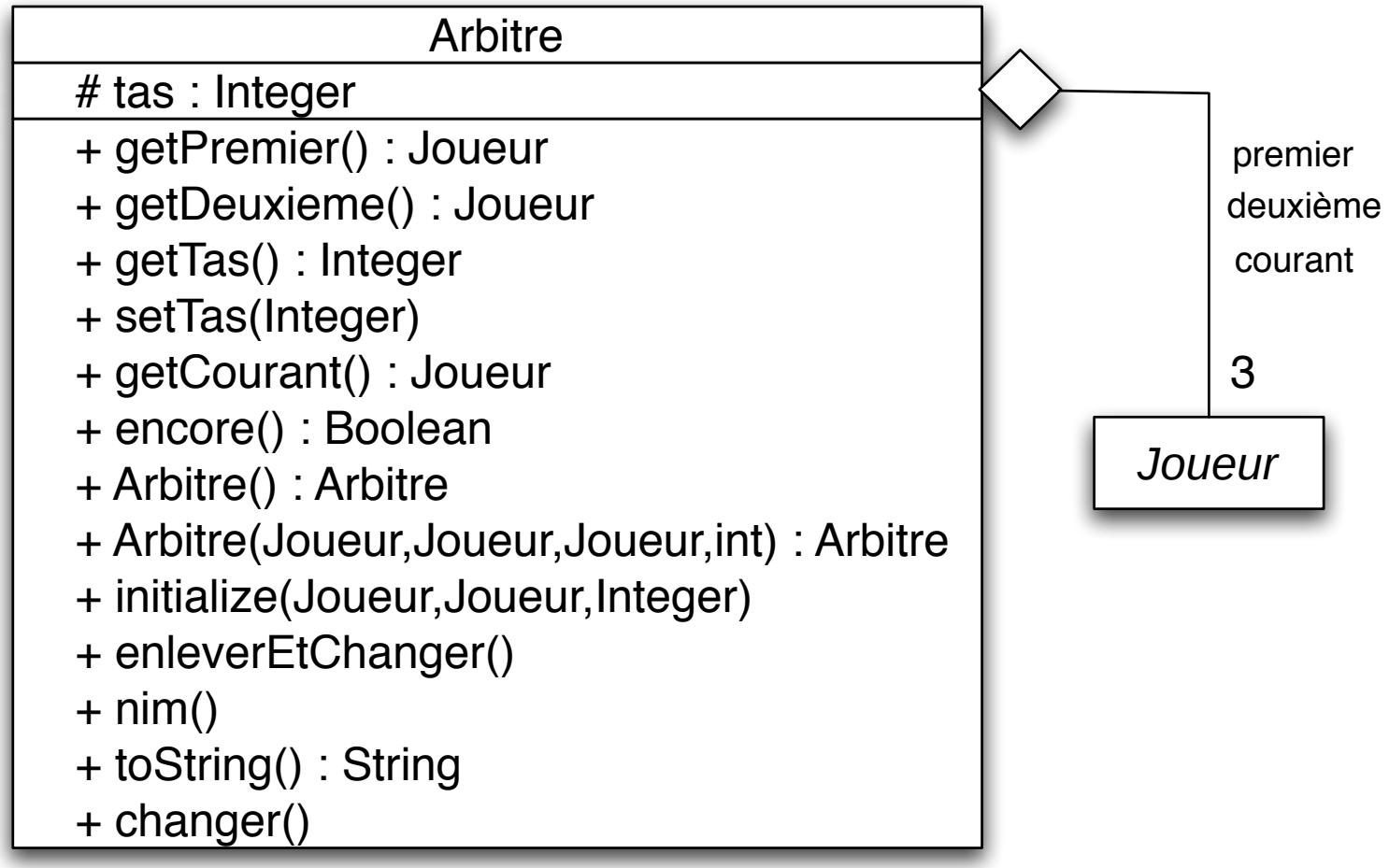
L'organisation des joueurs

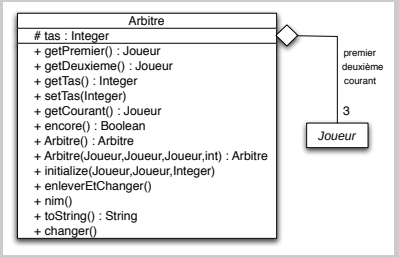




Les flèches de ce type représente l'héritage en UML. C'est un cas particulier de relation entre les classes.

La relation de composition

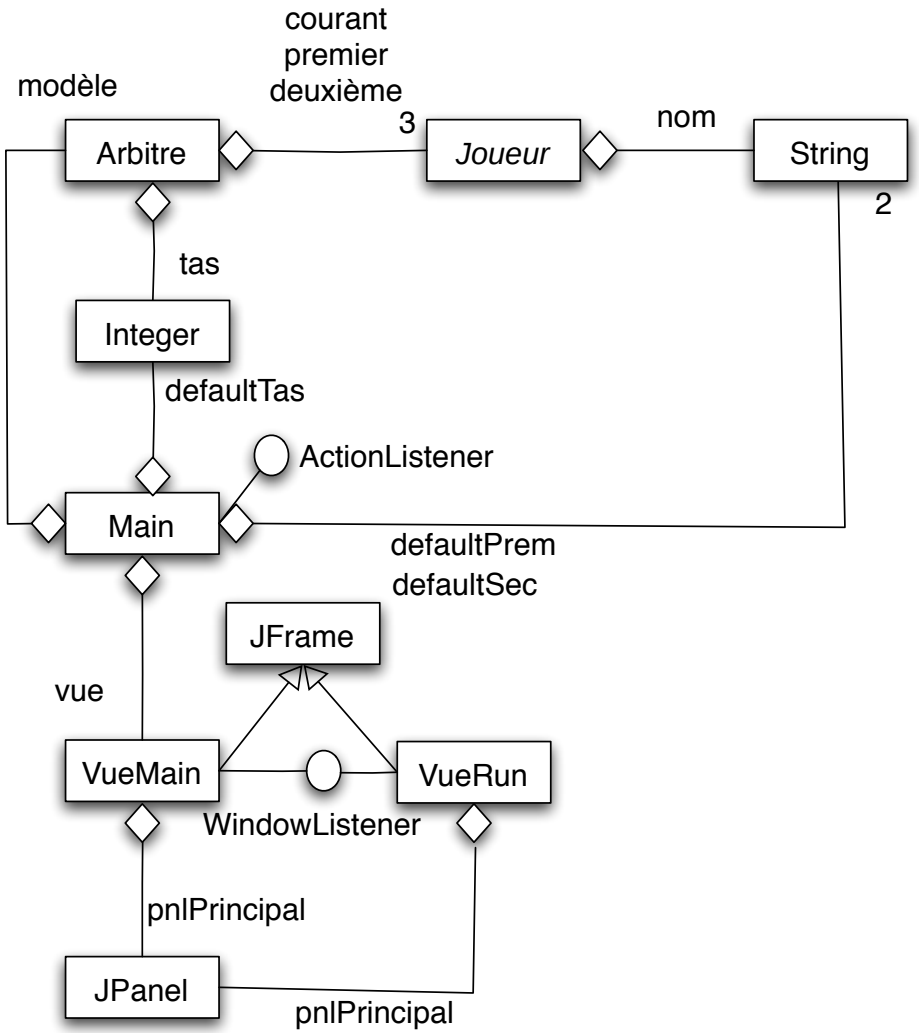


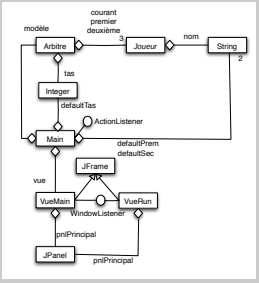


Ce nouveau diagramme dit qu'une instance de la classe arbitre utilise trois instances de la classe joueur. La relation de composition/agrégation désigne une notion de tout partie entre les objets. Le chiffre 3 représente la cardinalité de la relation.

Attention : en UML il y a le losange noir et le blanc, ici pas de distinction entre les deux.

La structure générale





Ce diagramme est plus compliqué. L'idée est de faire apparaître la structure de l'ensemble mais uniquement les parties et services publiques. Pour les services publiques on se réfère à la description de l'interface. Bien sûr des commentaires doivent accompagner et expliquer l'intérêt des liens et des parties. Cela donne une idée des dépendances structurelles entre les classes, mais ce n'est pas suffisant car n'explique pas le lien entre les vues par exemple.
La notation avec le rond désigne l'implémentation d'une interface (exemple avec `Main` et `ActionListener`).
Bien sûr on peut et on doit enrichir ce premier schéma avec les interfaces plus complètes des différentes classes.
Et si on jetait un oeil à l'application réelle sous Eclispe ?

Que se passe-t-il à l'exécution ?

- Avoir une description du fonctionnement dynamique

Que se passe-t-il à l'exécution ?

- Avoir une description du fonctionnement dynamique
- Les diagrammes de séquences apportent une réponse simple et limitée mais très appréciée

Que se passe-t-il à l'exécution ?

- Avoir une description du fonctionnement dynamique
- Les diagrammes de séquences apportent une réponse simple et limitée mais très appréciée
- Ils décrivent la chronologie des événements entre les objets

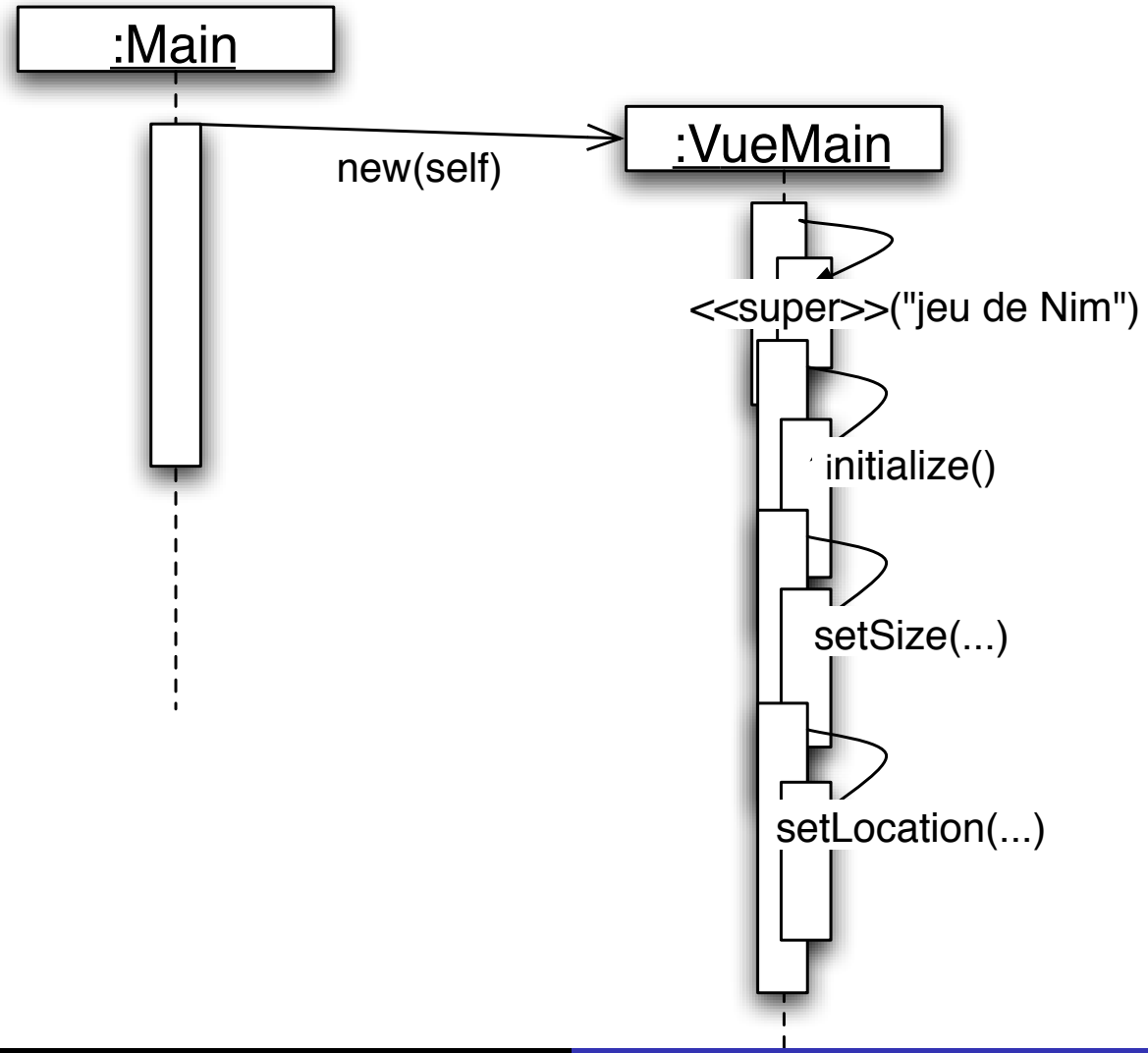
Que se passe-t-il à l'exécution ?

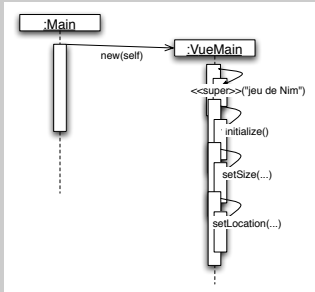
- Avoir une description du fonctionnement dynamique
- Les diagrammes de séquences apportent une réponse simple et limitée mais très appréciée
- Ils décrivent la chronologie des événements entre les objets
- Verticalement nous avons les lignes de vies des objets

Que se passe-t-il à l'exécution ?

- Avoir une description du fonctionnement dynamique
- Les diagrammes de séquences apportent une réponse simple et limitée mais très appréciée
- Ils décrivent la chronologie des événements entre les objets
- Verticalement nous avons les lignes de vies des objets
- Horizontalement ce sont les envois de messages entre objets

Un cas d'exécution dynamique

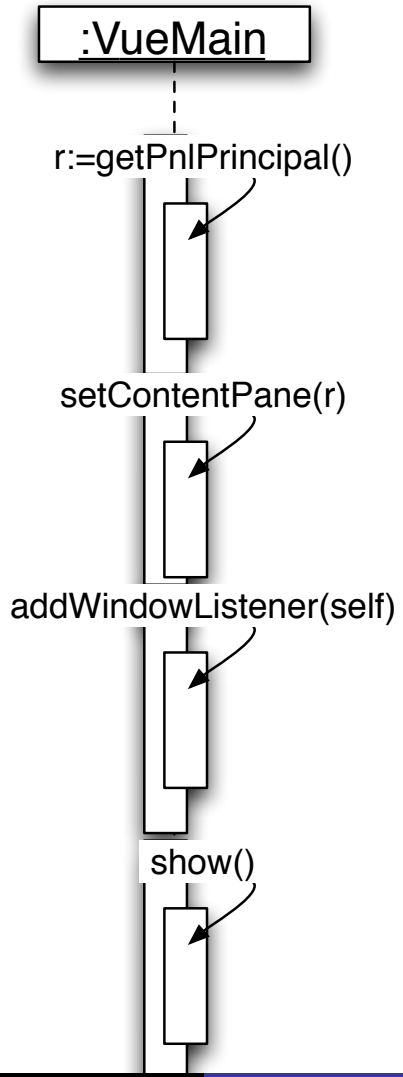


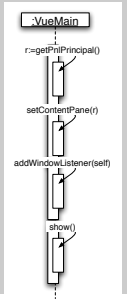


Au départ (lecture du haut vers le bas) il y a un envoi de création d'un objet de type VueMain par un objet de type Main. C'est une création car la pointe de la flèche arrive sur la boite désignant l'objet. Les deux instances sont anonymes mais on connaît leur classe. Ensuite le VueMain s'envoie plusieurs messages avec ou sans argument.

- `<<super>>("jeu de Nim")` : UML ne connaît pas `super`
- `initialize()`
- `setSize()`
- `setLocation()`

suite ...

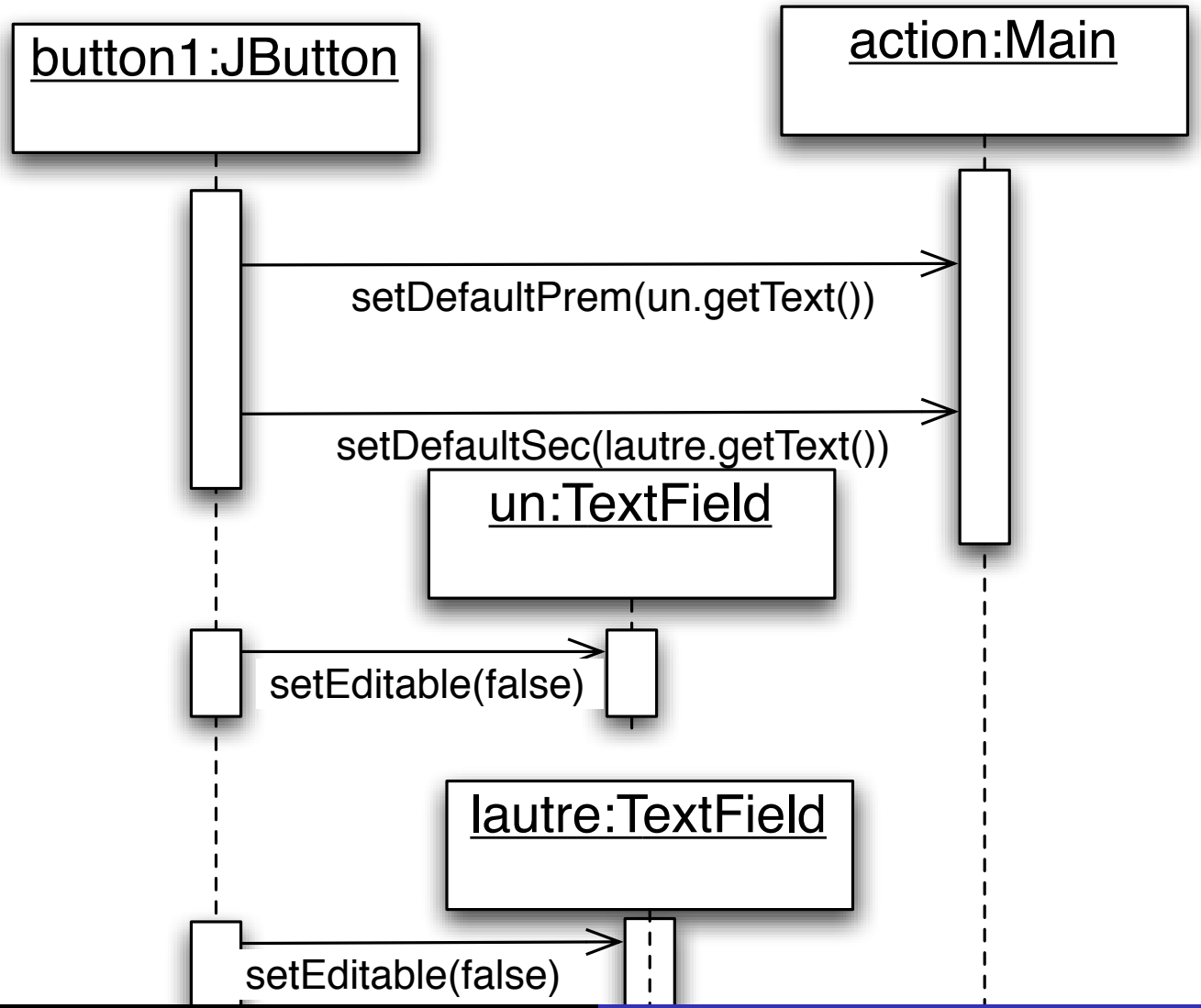

:VueMain
r:=getPnlPrincipal()
setContentPane(r)
addWindowListener(self)
show()



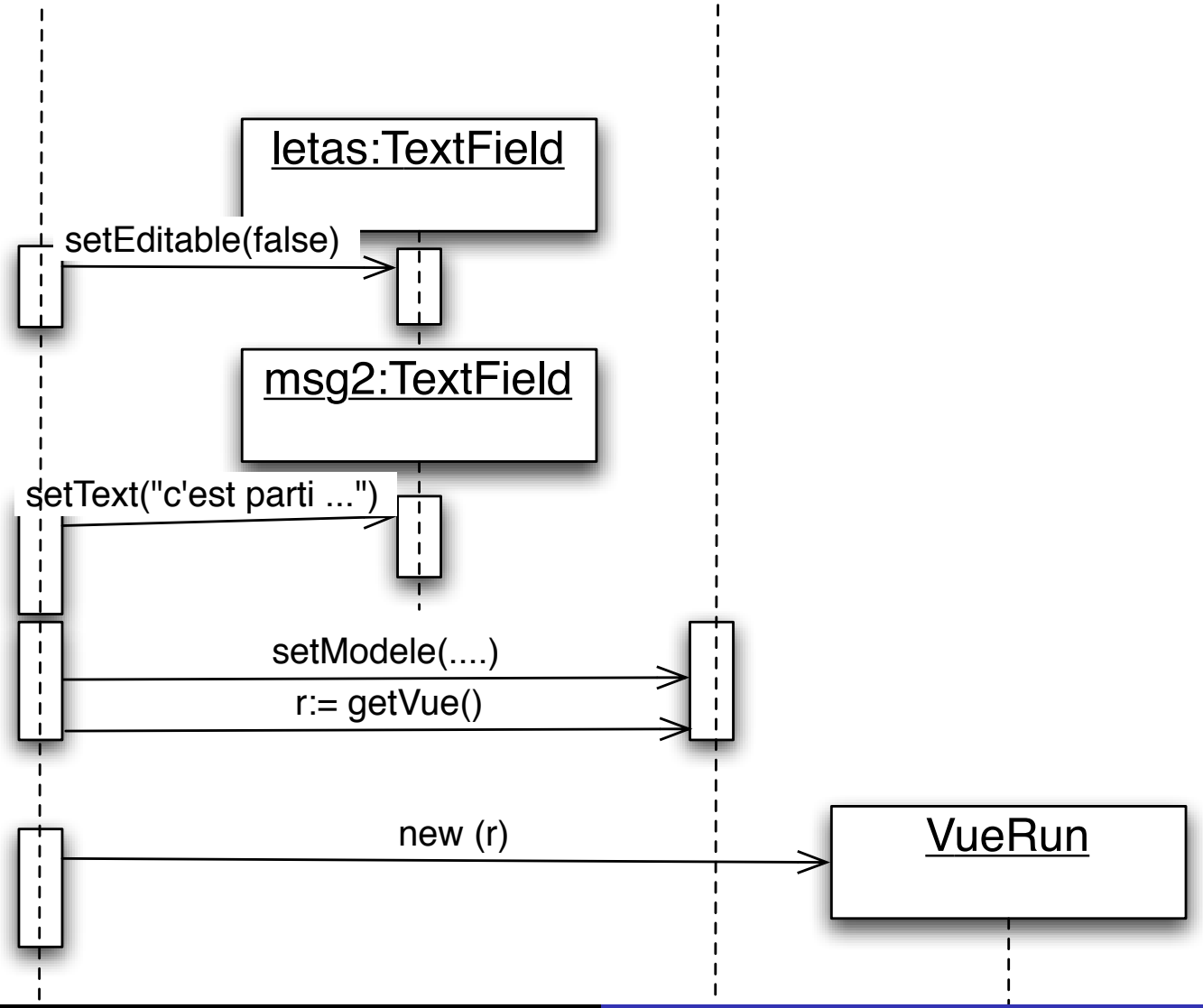
- `r=getPnlPrincipal()`
- `setContentPane(r)`
- `addWindowListener(self)`
- `this.show()`

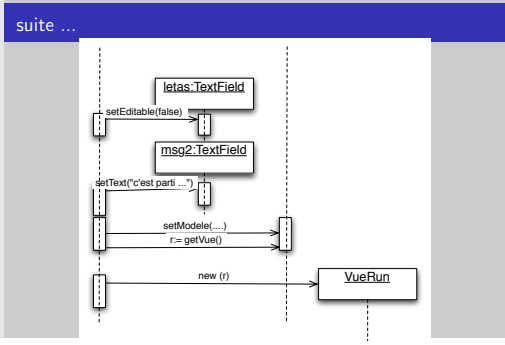
La plupart vous sont connus car proviennent de SWING ou AWT. Le premier avec r= permet de récupérer un résultat qui sera utilisé plus tard. **Attention** la flèche atteint l'objet receveur et donc le code `objet.message(para)` dans un contexte `C` correspond à une flèche de `C` vers `objet` d'étiquette `message(para)`.

Un autre exemple



suite ...


letas:TextField
setEditable(false)
msg2:TextField
setText("c'est parti ...")
setModele(....)
r:= getVue()
new (r)
VueRun



Noter que UML est très riche et très complexe.

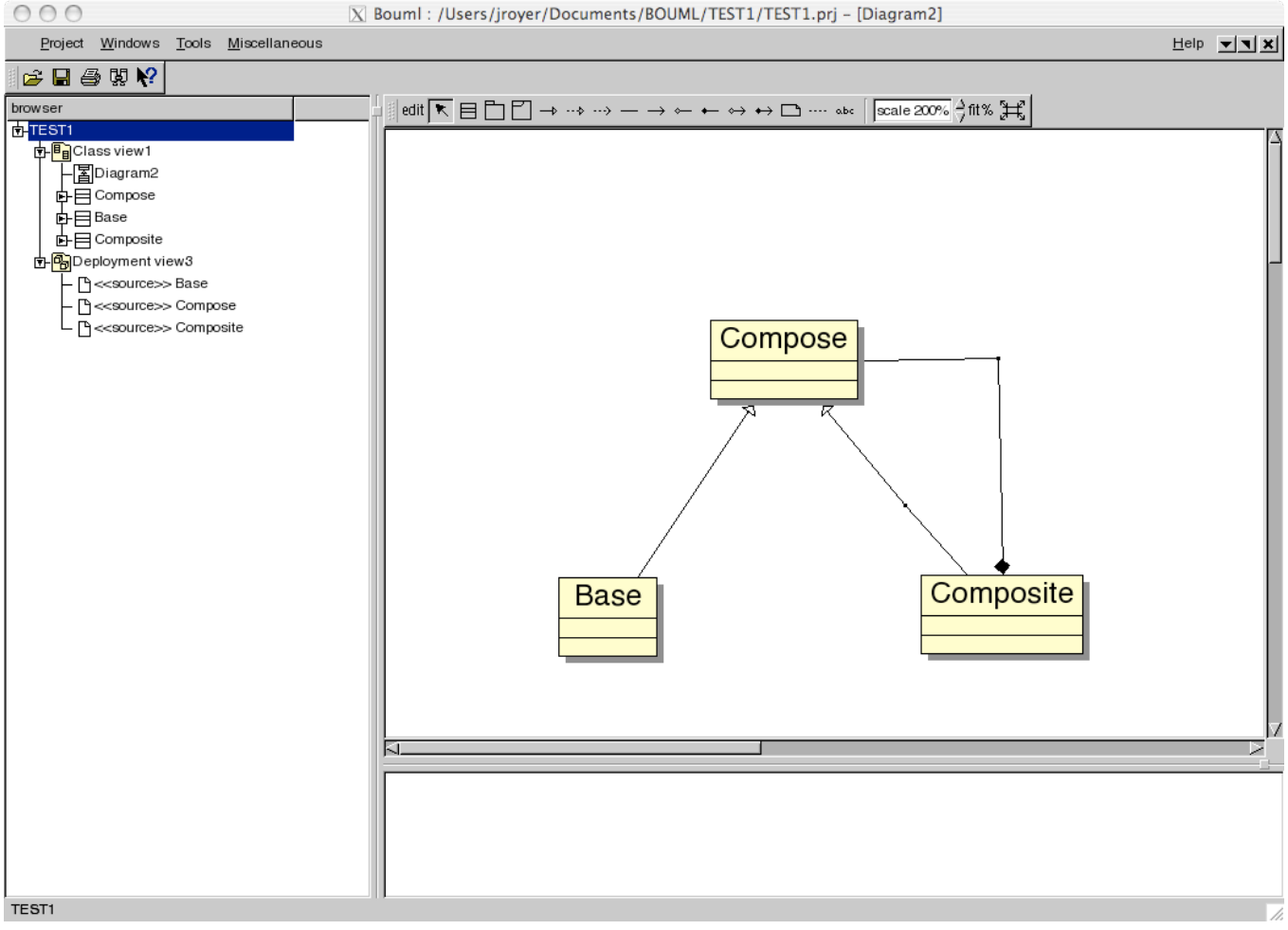
- Pour faire des présentations plus complexes, en programmation concurrente ou en IHM on utilise les diagrammes d'états.
- UML fournit un formalisme de ce type appelé Statechart.
- On peut les voir comme une forme condensé d'un ensemble fini ou non de diagrammes de séquences.

Outils UML

- Ils sont capables d'éditer les différents diagrammes
- Certains font de la génération de code
- Des contrôles sont possibles
- Parfois la rétro-conception est possible
- Il y en a de nombreux : gratuits, version d'essai illimitées, version commerciales
- Un exemple parmis d'autres : `bouml`
- Il y'a des pulgins Eclipse pour UML...
- Une préférence dans ce cours pour ArgoUML

Outils UML

- Ils sont capables d'éditer les différents diagrammes
- Certains font de la génération de code
- Des contrôles sont possibles
- Parfois la rétro-conception est possible
- Il y en a de nombreux : gratuits, version d'essai illimitées, version commerciales
- Un exemple parmis d'autres : `bouml`
- Il y'a des pulgins Eclipse pour UML...
- Une préférence dans ce cours pour ArgoUML



Patrons de conception

- Un concept utile pour améliorer une architecture et rationaliser la conception

Patrons de conception

- Un concept utile pour améliorer une architecture et rationaliser la conception
- Solution particulière mais éprouvée à un problème récurrent

Patrons de conception

- Un concept utile pour améliorer une architecture et rationaliser la conception
- Solution particulière mais éprouvée à un problème récurrent
- Exemple : organiser des hiérarchies d'objets

Patrons de conception

- Un concept utile pour améliorer une architecture et rationaliser la conception
- Solution particulière mais éprouvée à un problème récurrent
- Exemple : organiser des hiérarchies d'objets
- Classification : structurel, comportemental et fabrique

Patrons de conception

- Un concept utile pour améliorer une architecture et rationaliser la conception
- Solution particulière mais éprouvée à un problème récurrent
- Exemple : organiser des hiérarchies d'objets
- Classification : structurel, comportemental et fabrique
- Un autre élément orthogonal est la cible : classe ou instance

- Un concept utile pour améliorer une architecture et rationaliser la conception
- Solution particulière mais éprouvée à un problème récurrent
- Exemple : organiser des hiérarchies d'objets
- Classification : structurel, comportemental et fabrique
- Un autre élément orthogonal est la cible : classe ou instance

Les patrons servent surtout en conception à capturer de bonnes solutions à des problèmes récurrents. Il existe une autre notion voisine.
Framework : modèle d'architecture qui peut être utilisée pour organiser une application complète, par exemple J2EE pour les applications Internet en Java. Une différence importante est la granularité.

`http://www.research.umbc.edu/~tarr/dp/fall00/cs491.html` : un cours complet avec application à Java.

Patron : *singleton*

- Le problème : définir une classe ayant une seule instance accessible à des clients

Patron : *singleton*

- Le problème : définir une classe ayant une seule instance accessible à des clients
- Besoin de sous-classage de cette information

Patron : *singleton*

- Le problème : définir une classe ayant une seule instance accessible à des clients
- Besoin de sous-classage de cette information
- Un cas typique est la classe `Main` principale d'un programme

Patron : *singleton*

- Le problème : définir une classe ayant une seule instance accessible à des clients
- Besoin de sous-classage de cette information
- Un cas typique est la classe `Main` principale d'un programme
- Patron de type fabrique pour un objet

Patron : *singleton*

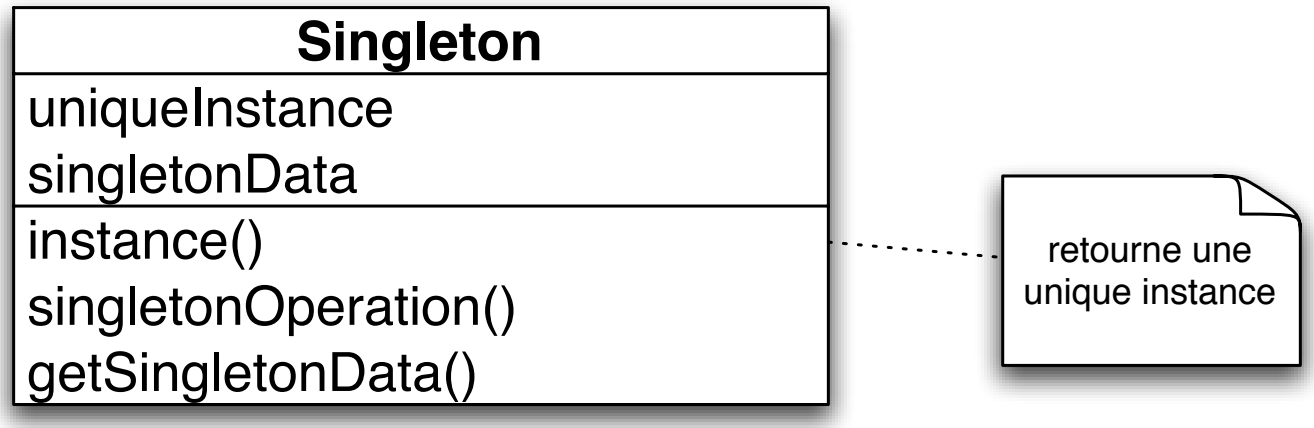
- Le problème : définir une classe ayant une seule instance accessible à des clients
- Besoin de sous-classage de cette information
- Un cas typique est la classe `Main` principale d'un programme
- Patron de type fabrique pour un objet
- Exemples : exemple la liste vide, ...

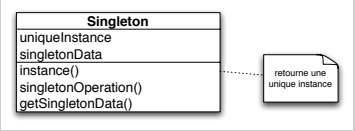
Patron : *singleton*

- Le problème : définir une classe ayant une seule instance accessible à des clients
- Besoin de sous-classage de cette information
- Un cas typique est la classe `Main` principale d'un programme
- Patron de type fabrique pour un objet
- Exemples : exemple la liste vide, ...
- La solution passe par une création d'instance

Le besoin existe dans beaucoup de cas.
Exemple d'une unique connexion réseau non partageable, d'un compteur qui est global et partagé par plusieurs processus, maintenir et connaître la liste des objets créés d'une certaine classe, la liste vide, certaines valeurs singulières d'un système peuvent devenir des singletons, etc.
Peut être adapté pour créer plusieurs éléments de ce type au lieu d'un seul.

Schéma UML : le singleton





Une solution un peu rigide est de recourir au qualificatif static qui remplit en partie ce rôle.
Toutefois la création d'une instance est plus flexible et plus évolutive grâce a l'héritage.
Ici cas simple héritage et concurrence peuvent notablement compliquer les choses ...
`http://java.sun.com/developer/technicalArticles/Programming/singletons/` : attention très avancé !

MonSingleton Minimal

```
public class MonSingleton {
 // il est unique
 private static  MonSingleton _objet = new MonSingleton();

 // le constructeur
 private MonSingleton() {
    //faut qd meme travailler un peu ...
 }

 // l'accesseur
 public  MonSingleton instance() { return _objet;}

} // end
```

MonSingleton Minimal

```
public class MonSingleton {
 // il est unique
 private static  MonSingleton _objet = new MonSingleton();

 // le constructeur
 private MonSingleton() {
    //faut qd meme travailler un peu ...
 }

 // l'accesseur
 public  MonSingleton instance() { return _objet;}

} // end
```

Exercice : dans l'exemple des listes d'entiers par patron composite il serait bien d'ajouter le singleton pour la liste vide ...
Exercice : imaginons que certaines fois j'ai besoin de cet objet mais d'autre fois non alors comment faire pour l'instancier à la demande lors du premier accès ?

Patron : composite

- Le problème : organiser des informations en liste ou en structures plus complexes

Patron : composite

- Le problème : organiser des informations en liste ou en structures plus complexes
- Structurel pour des objets

Patron : composite

- Le problème : organiser des informations en liste ou en structures plus complexes
- Structurel pour des objets
- Permet d'uniformiser les manipulations des objets de base et des composés en utilisant la classe racine : `Compound`

Patron : composite

- Le problème : organiser des informations en liste ou en structures plus complexes
- Structurel pour des objets
- Permet d'uniformiser les manipulations des objets de base et des composés en utilisant la classe racine : `Compound`
- Exemple : arbre binaire, liste d'arité et de profondeur variable, objets graphiques, ...

Patron : composite

- Le problème : organiser des informations en liste ou en structures plus complexes
- Structurel pour des objets
- Permet d'uniformiser les manipulations des objets de base et des composés en utilisant la classe racine : `Compound`
- Exemple : arbre binaire, liste d'arité et de profondeur variable, objets graphiques, ...
- La solution passe par une organisation subtile liant héritage et composition

Patron : composite

- Le problème : organiser des informations en liste ou en structures plus complexes
- Structurel pour des objets
- Permet d'uniformiser les manipulations des objets de base et des composés en utilisant la classe racine : Compound
- Exemple : arbre binaire, liste d'arité et de profondeur variable, objets graphiques, ...
- La solution passe par une organisation subtile liant héritage et composition

Attention l'implémentation dans un langage fortement typé peut-être délicate.
Bien sûr ce schéma possède de nombreuses variations suivants les opérations et données définies dans les classes.
On peut également jouer sur le nombre de classe de base, sur la cardinalité du lien de composition, la racine concrète ou abstraite etc.

`Exercice` : en principe cette cardinalité, dans le cas d'une racine abstraite, ne devrait pas être 0 ! pourquoi ?

Schéma UML : le composite

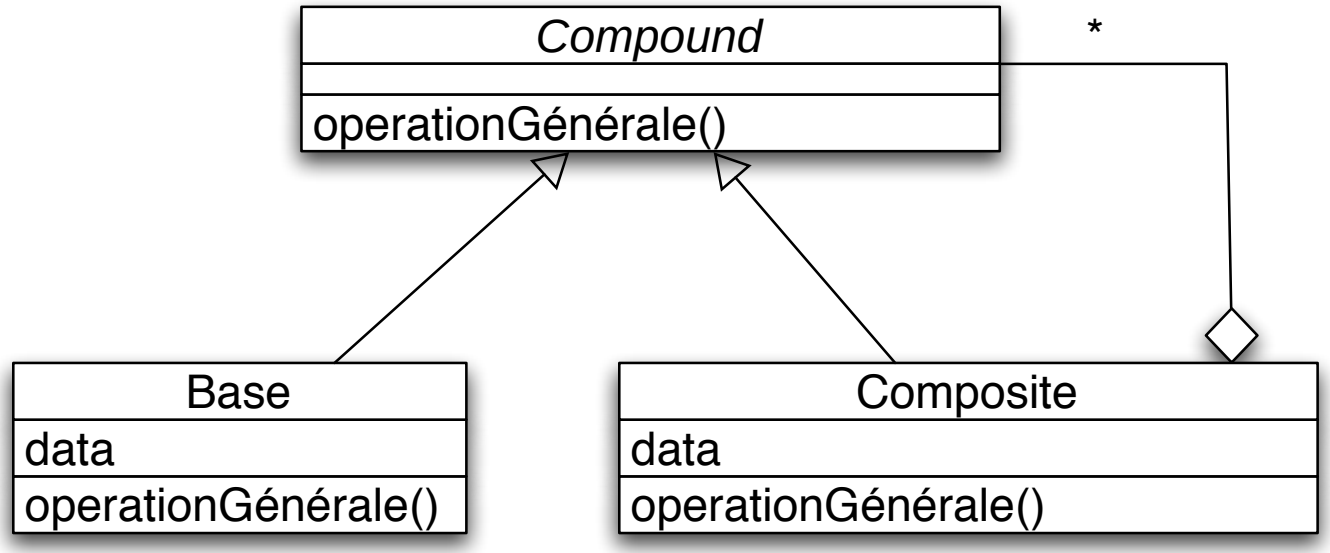
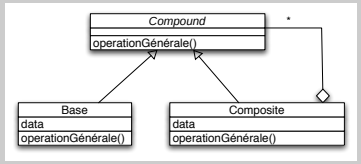


Schéma UML : le composite



Exemple d'application aux listes d'entiers donné dans les documents du cours.

On peut enrichir les services de `Composite` avec des opérations d'ajouts et d'accès aux éléments.

La définition des opérations du composite peut-être purement récursive ou itérative et dans ce cas il faut connaître le nombre d'éléments au niveau courant.

En général pas besoin de tester le type du composant, ça doit rester exceptionnel et contrôlé.

`http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html` : assez basique

Code Java du composite

```
// pensez aux attributs et constructeurs

public abstract class Compound {
    // une methode
    public     abstract Type method();
} // end

public class Base extends Compound {
   // redefinie
   public  Type method() { ... };
} // end

public class Composite extends Compound {
   // redefinie
   public  Type method() { ... };
} // end

// les signatures peuvent être redéfinis
```

Code Java du composite

```
// pensez aux attributs et constructeurs

public abstract class Compound {
    // une methode
    public     abstract Type method();
} // end

public class Base extends Compound {
   // redefinie
   public  Type method() { ... };
} // end

public class Composite extends Compound {
   // redefinie
   public  Type method() { ... };
} // end

// les signatures peuvent être redéfinis
```

`Compound` : généralement des abstraites redéfinies dans les sous-classes.
`Base` : redéfinition assez triviale
`Composite` : redéfinition assez simple récursive ou itérative

Règles

- Avoir une structure aussi précise que possible mais extensible : dans une liste "plate" les composants sont uniquement des feuilles

Règles

- Avoir une structure aussi précise que possible mais extensible : dans une liste "plate" les composants sont uniquement des feuilles
- Organiser logiquement les services : une méthode commune doit avoir un profil dans la super-classe

Règles

- Avoir une structure aussi précise que possible mais extensible : dans une liste "plate" les composants sont uniquement des feuilles
- Organiser logiquement les services : une méthode commune doit avoir un profil dans la super-classe
- Eviter les tests de type, ou passer par une méthode adéquate

Règles

- Avoir une structure aussi précise que possible mais extensible : dans une liste "plate" les composants sont uniquement des feuilles
- Organiser logiquement les services : une méthode commune doit avoir un profil dans la super-classe
- Eviter les tests de type, ou passer par une méthode adéquate
- Exception ou pas exception ? : par exemple la suppression d'un élément peut-être placée dans `Compound` ou dans `Composite`

Règles

- Avoir une structure aussi précise que possible mais extensible : dans une liste "plate" les composants sont uniquement des feuilles
- Organiser logiquement les services : une méthode commune doit avoir un profil dans la super-classe
- Eviter les tests de type, ou passer par une méthode adéquate
- Exception ou pas exception ? : par exemple la suppression d'un élément peut-être placée dans `Compound` ou dans `Composite`
- Une politique uniforme récursive ou itérative de définition des méthodes doit être privilégiée

Règles

- Avoir une structure aussi précise que possible mais extensible : dans une liste "plate" les composants sont uniquement des feuilles
- Organiser logiquement les services : une méthode commune doit avoir un profil dans la super-classe
- Eviter les tests de type, ou passer par une méthode adéquate
- Exception ou pas exception ? : par exemple la suppression d'un élément peut-être placée dans `Compound` ou dans `Composite`
- Une politique uniforme récursive ou itérative de définition des méthodes doit être privilégiée

Patron : *état*

- Problème assez classique de codage d'un aiguillage avec action

Patron : *état*

- Problème assez classique de codage d'un aiguillage avec action
- Utilisation du `switch` : un peu cablée

Patron : *état*

- Problème assez classique de codage d'un aiguillage avec action
- Utilisation du `switch` : un peu cablée
- Matrice de transition : bonne solution si le système de transition varie

Patron : *état*

- Problème assez classique de codage d'un aiguillage avec action
- Utilisation du `switch` : un peu cablée
- Matrice de transition : bonne solution si le système de transition varie
- Patron état : compromis intéressant car facilement extensible et modulaire

Patron : *état*

- Problème assez classique de codage d'un aiguillage avec action
- Utilisation du `switch` : un peu cablée
- Matrice de transition : bonne solution si le système de transition varie
- Patron état : compromis intéressant car facilement extensible et modulaire
- Analyse, lexicale, syntaxique et compilation : GS1 !

- Problème assez classique de codage d'un aiguillage avec action
- Utilisation du `switch` : un peu cablée
- Matrice de transition : bonne solution si le système de transition varie
- Patron état : compromis intéressant car facilement extensible et modulaire
- Analyse, lexicale, syntaxique et compilation : GS1 !

Problème très général du codage d'un ensemble d'aiguillage avec exécution d'actions. Celles-ci pouvant être de simple transformations de l'entrée ou des actions complexes.
Beaucoup de solutions existent mais elles ne sont pas forcément comparables en termes de maintenabilité, de modularité et de facilité de mise en œuvre.

`www.informit.com/articles/printerfriendly.asp?p=28677`

Patron : *état*

- Le problème : coder des états finis et des changements d'états en gardant un système maintenable

Patron : *état*

- Le problème : coder des états finis et des changements d'états en gardant un système maintenable
- Type comportemental pour un objet

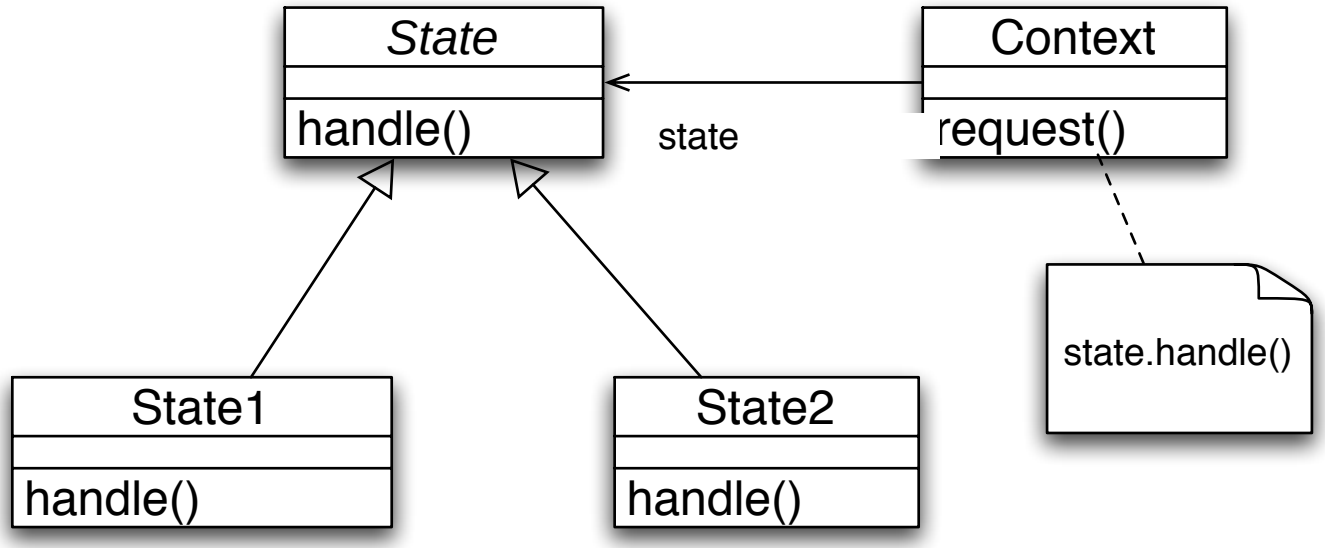
Patron : *état*

- Le problème : coder des états finis et des changements d'états en gardant un système maintenable
- Type comportemental pour un objet
- Exemple un ascenseur qui à un comportement différent suivant qu'il monte, descend ou est à l'arrêt, d'une présentation graphique ou d'une vue, etc

Patron : *état*

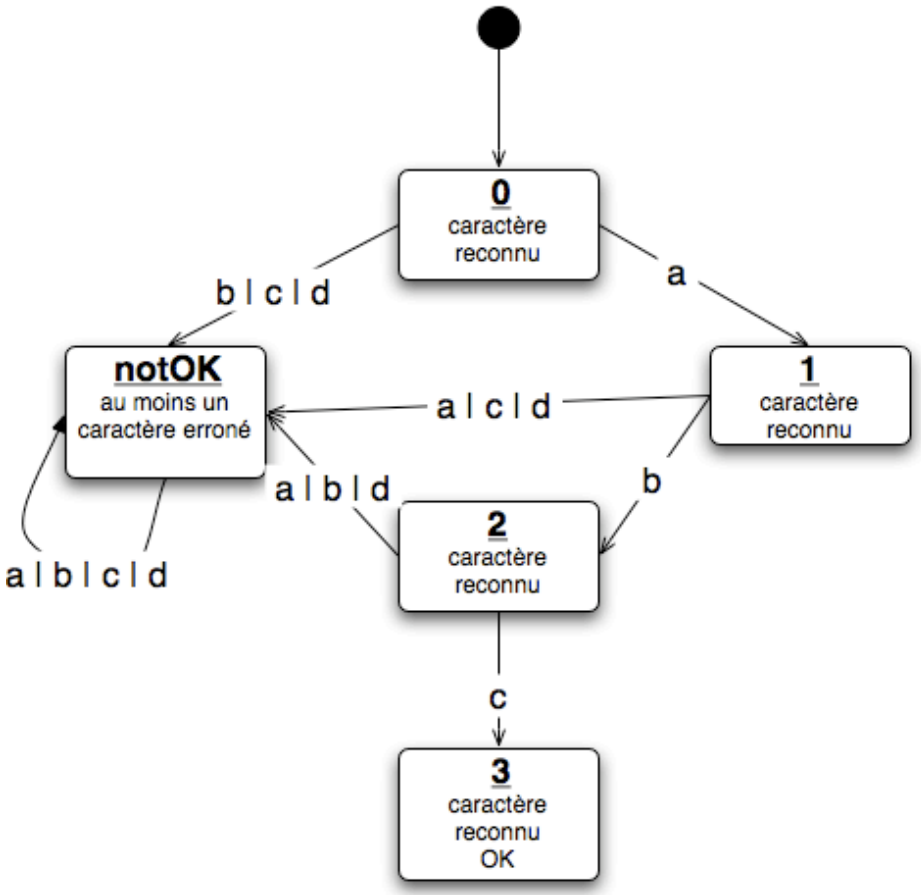
- Le problème : coder des états finis et des changements d'états en gardant un système maintenable
- Type comportemental pour un objet
- Exemple un ascenseur qui à un comportement différent suivant qu'il monte, descend ou est à l'arrêt, d'une présentation graphique ou d'une vue, etc
- La solution : passe par l'isolation des états dans une classe et la définition d'un "handler" pour associer les actions

Schéma UML : le state



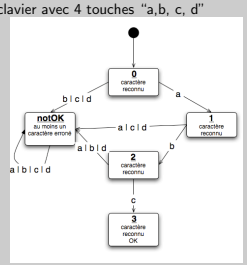
Un exemple de "digit code"

Réaliser un contrôleur qui vérifie que la séquence "a,b ,c" a été entré sur un clavier avec 4 touches "a,b, c, d"



Un exemple de “digit code”

Réaliser un contrôleur qui vérifie que la séquence “a,b ,c” a été entré sur un clavier avec 4 touches “a,b, c, d”



C’est l’occasion de parler de machines à états et transitions !
Il existe de nombreux formalismes apparentés : automates, réseaux de Petri, etc
Les états (les ronds) représentent des situations importantes du système (ici il s’agit du nombre de caractère reconnu) ou des états dit stables.
Ensuite une flèche (ou transition) décrit les changements d’états autorisés. Une transition porte une étiquette qui désigne un événement et une action (ici la frappe d’une touche).

Code Java

```
public abstract class State{
    // avec nommage explicite
    protected String _label;
    public String getLabel() { return _label;}

    // public abstract State handle(char k); ou mieux ici
    public  State handle(char k) {return new State4(); }
}
public class State0 extends State{

    public State0 () {_label = "0"; }

    public  State handle(char k) {
        if (k=='A') {
          return new State1();
      }
      else {
          return super.handle(k);
      }
    }
} // fin
```

Code Java

```
public abstract class State{
    // avec nommage explicite
    protected String _label;
    public String getLabel() { return _label;}

    // public abstract State handle(char k); ou mieux ici
    public  State handle(char k) {return new State4(); }
}
public class State0 extends State{

    public State0 () {_label = "0"; }

    public  State handle(char k) {
        if (k=='A') {
          return new State1();
      }
      else {
          return super.handle(k);
      }
    }
} // fin
```

Le code Java du programme correspondant est donné dans l'annexe.
Context= DigitCode.java, State=State.java
Les états sont des signetons ici ...

Règles

- Faire le diagramme avec le contexte, la classe abstraite état et les sous-classes + les services à définir

Règles

- Faire le diagramme avec le contexte, la classe abstraite état et les sous-classes + les services à définir
- Utiliser l'héritage pour organiser et simplifier le comportement d'aiguillage

Règles

- Faire le diagramme avec le contexte, la classe abstraite état et les sous-classes + les services à définir
- Utiliser l'héritage pour organiser et simplifier le comportement d'aiguillage
- Penser au cas d'erreur qui peut être le comportement par défaut dans la super-classe

Règles

- Faire le diagramme avec le contexte, la classe abstraite état et les sous-classes + les services à définir
- Utiliser l'héritage pour organiser et simplifier le comportement d'aiguillage
- Penser au cas d'erreur qui peut être le comportement par défaut dans la super-classe
- Préférer le nommage implicite (par la classe) plus simple et plus évolutif

Règles

- Faire le diagramme avec le contexte, la classe abstraite état et les sous-classes + les services à définir
- Utiliser l'héritage pour organiser et simplifier le comportement d'aiguillage
- Penser au cas d'erreur qui peut être le comportement par défaut dans la super-classe
- Préférer le nommage implicite (par la classe) plus simple et plus évolutif
- L'état peut-être un attribut d'un objet ou passé en paramètre des méthodes

La solution de nommage implicite est la suivante : au lieu d'avoir un champ avec ses accesseurs il suffit d'écrire dans la racine `State` :

```
public String getName () {return getClass().getName(); }
```

Nous utilisons ici le fait que nous avons accès à la classe de l'objet, et que celle-ci est vue comme un objet et donc a des attributs.

La généralisation de ce genre de sport s'appelle la *Métaprogrammation* ...

Patron : observateur

- Le problème : si deux entités sont dépendantes l'une de l'autre et qu'il faut maintenir dynamiquement ce lien

Patron : observateur

- Le problème : si deux entités sont dépendantes l'une de l'autre et qu'il faut maintenir dynamiquement ce lien
- Type comportemental pour un objet

Patron : observateur

- Le problème : si deux entités sont dépendantes l'une de l'autre et qu'il faut maintenir dynamiquement ce lien
- Type comportemental pour un objet
- Exemple : des données et leur présentation graphique comme dans le cas des IHM en général

Patron : observateur

- Le problème : si deux entités sont dépendantes l'une de l'autre et qu'il faut maintenir dynamiquement ce lien
- Type comportemental pour un objet
- Exemple : des données et leur présentation graphique comme dans le cas des IHM en général
- La solution passe par un double lien observateur - sujet observé qu'il faut maintenir dynamiquement. Ce lien sert de véhicule pour les notifications et les mises à jour.

Très utile si j'ai une donnée à présenter sous plusieurs vues qui doivent toutes être cohérentes avec ma donnée.
Les `update` sont propagés aux observateurs qui vont alors remettre à jour l'état observé (`getState`).
`Subject` dispose d'une interface pour gérer (ajout/suppression) les observateurs.
Le MVC, vu ou à voir en IHM, peut s'implémenter à l'aide de ce patron.
Ce patron existe de façon native dans `java.util` - `Observer/Observable`
Sert à la conception des GUI : sujet concret : événement source et objet concret écouteur d'évènement.

Schéma UML : l'observateur

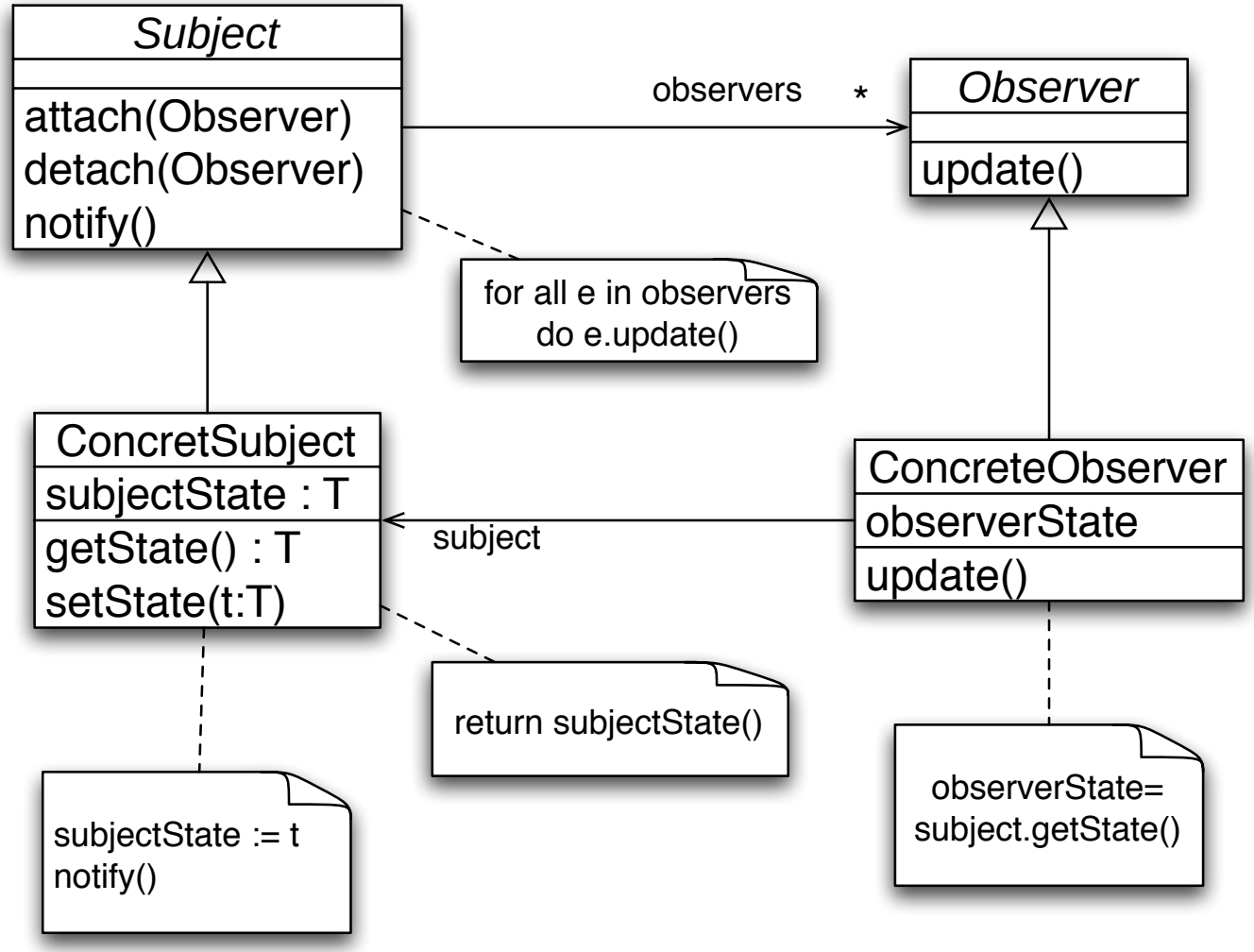
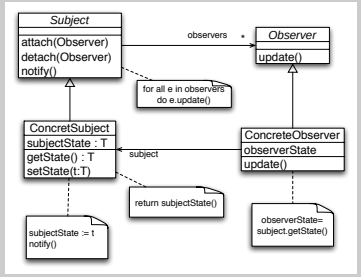


Schéma UML : l'observateur



On pourrait également parler du visiteur qui permet une meilleure séparation entre traitement et données.

Exemple d'API

- Pomme d'Api ou dieu égyptien ?

Exemple d'API

- Pomme d'Api ou dieu égyptien ?
- Application Programming Interface

Exemple d'API

- Pomme d'Api ou dieu égyptien ?
- Application Programming Interface
- `http://java.sun.com/j2se/1.5.0/docs/api/`

Exemple d'API

- Pomme d'Api ou dieu égyptien ?
- Application Programming Interface
- `http://java.sun.com/j2se/1.5.0/docs/api/`
- Utilitaire `javap`

- Pomme d'Api ou dieu égyptien ?
- Application Programming Interface
- `http://java.sun.com/j2se/1.5.0/docs/api/`
- Utilitaire `javap`

Nous en verrons d'autres exemples avec la plupart des applications et outils pour Java

javap java.lang.String

```
Compiled from "String.java" public final class java.lang.String
extends java.lang.Object
implements java.io.Serializable,java.lang.Comparable,java.lang.C
public static final java.util.Comparator CASE_INSENSITIVE_ORDER;

public int hashCode();
public int length();
static {};
public java.lang.String();
public byte[] getBytes();
public char[] toCharArray();
public char charAt(int);
public int indexOf(int);
public int lastIndexOf(int);
public int indexOf(int,int);
public int lastIndexOf(int,int);
```