IK3 : Cours n°3
Gestion de projets

semaine du 11 octobre 2010

1 But du cours

Il faut comprendre que quelque soit l'ampleur d'un projet, il existe des outils et des méthodes adaptés qui permettent de le mener à bien. C'est l'objet d'étude du génie logiciel qui a pour but la conception, la fabrication et la maintenance des systèmes complexes. Nous ne parlerons pas (ou peu) de l'aspect financier, de la présence d'un client et du monde de l'entreprise en général. Ce sont des aspects primordiaux qui seront traités en Génie Logiciel en L3 et surtout M1.

2 Vers un bon logiciel

Bibliographie : Enseignements de Yann Régis-Gianas : `http://www.pps.jussieu.fr/~yrg/teaching.php`

On veut atteindre plusieurs buts qui deviennent plus difficiles au fur et à mesure que les projets "grossissent". Le premier but est de respecter la spécification établie. Celle-ci peut prendre plusieurs formes, tel un texte descriptif, ou des descriptions mathématiques. En plus, il y a quatre principaux critères qui font que l'on aura un bon logiciel :

- Maintenabilité : Peut-on faire évoluer le logiciel ?
- Robustesse : Le logiciel est-il sujet à des dysfonctionnements ?
- Efficacité : Le logiciel fait-il bon usage de ses ressources ?
- Utilisabilité : Est-il facile à utiliser ?

3 Problèmes engendrés par les gros projets

Jusqu'ici (IF1/IS1) vous avez développé seul des applications allant jusqu'à 50 lignes maximum. Plusieurs données peuvent jouer sur un projet :

- Le nombre de lignes de code.
  On ne ne gère pas de la même façon 50 lignes de code ou 5 millions. Pour donner un ordre d'idée, le système d'exploitation de vos machines (FreeBSD) totalise presque 10 millions, Firefox c'est 25 millions, certains Windows plus de 50 millions.
  On a empiriquement constaté que le nombre de bugs croit de façon plus que linéaire par rapport au nombre de lignes de code.
- Le nombre de personnes.
  Le passage de une à plusieurs personnes nécessite au moins de la communication à plusieurs niveaux :
  - Niveau humain.
    Les personnes doivent dialoguer pour que le projet avance et cela nécessite souvent une personne qui dirige l'équipe.
  - Niveau programmation.
    Le code de tous les développeurs doit pouvoir être geré de manière efficace (cf. subversion)

  Plusieurs personnes implique également le découpage et l'attribution des différentes tâches. Les tâches sont rarement complètement indépendantes et vos collègues seront souvent amenés à lire votre code, il faut donc faire attention à votre écriture. Il est impératif d'avoir des conventions de codage.
- Le temps
  Un projet peut durer dans le temps pour deux raisons :
  - Il est énorme et demande beaucoup de temps (plusieurs années)
  - Il est doit être maintenu (correction de bugs, support de nouveaux standards, ...)

  Dans tous les cas, personne ne peut avoir en tête tous les recoins du programme. C'est une raison de pérenniser le code, en écrivant des commentaires et de la documentation.

- La présence d'un client (même si ce n'est pas forcément lié aux gros projets)
  Dans ce cas, un bon programme est souvent un programme pour lequel le client est satisfait. Un client peut changer d'avis et demander des modifications une fois le programme écrit. Il faut donc écrire du code qui soit modulaire et réutilisable pour faire face à ces imprévus.
- Séparation des tâches
  Pour réaliser un programme, la durée nécessaire est exprimée en jours-homme. Attention, un projet nécessitant 100 jours-homme ne sera pas faisable en une journée par 100 personnes car il faut tenir compte de quelles tâches sont parallélisables. De plus, il faudrait beaucoup d'administration pour délimiter qui doit faire quoi et les risques de conflit (sur le code source) sont immenses. Une étude doit être faite avant de commencer pour déterminer quel est le nombre optimal de personnes à faire travailler sur le projet.

3.1 Grands principes du génie logiciel

Voici une liste :
- La rigueur.
  La plupart des bugs sont commis par les programmeurs. Il faut se questionner sur la validité de son action.
- La décomposition des problèmes en sous-problèmes indépendants.
  Traiter un seul problème à la fois est toujours plus simple pour la conception et l'implémentation.
- La modularité.
  Il s'agit de partitionner le logiciel en modules qui :
  - ont une cohérence interne (on ne regroupe pas ensemble des méthodes gérant la partie réseau et la partie affichage par exemple)
  - possèdent une interface ne divulgant sur le contenu du module que ce qui est strictement nécessaire aux modules clients.

  L'implémentation doit être indépendante de l'interface. Par exemple, au cours du semestre on sera peut être amenés à modifier la classe DeugSocket mais vous n'aurez rien à faire car l'interface et le comportement ne seront pas changés.
- L'abstraction.
  Il s'agit de trouver des concepts généraux à nos problèmes qui vont regrouper plusieurs cas particuliers Les classes abstraites de Java vont dans ce sens.
- La généricité. Un logiciel réutilisable a beaucoup plus de valeur qu'un dédié. Un logiciel est générique lorsqu'il est adaptable.
- La construction incrémentale.

3.2 Importance de l'écriture

Pour l'instant vous avez écrit du code. Ce n'est pourtant qu'une partie (pas si grande) de la programmation. Il y a également la lecture et la modification de code. L'écriture est **cruciale** car pour tout projet se maintenant dans le temps, chaque ligne sera lue et modifiée peut être plusieurs fois. De plus, il est très courant que l'écriture et la lecture soient faites par des personnes différentes. Qu'est-ce qui est plus facile à lire :

```
int f(int x, int y) {
    return x + x*y;
}

int caculPrixTTC(int prixHorsTaxe, int tauxTVA) {
    return prixHorsTaxe + tauxTVA*prixHorsTaxe;
}
```

Si on fait une erreur, dans quel cas saute-t-elle le mieux aux yeux :

```
int f(int x, int y) {
    return y + x*y;
}

int caculPrixTTC(int prixHorsTaxe, int tauxTVA) {
    return tauxTVA + tauxTVA*prixHorsTaxe;
}
```

Pour soi et pour les autres, il vaut mieux prendre du temps à l'écriture plutôt que d'en perdre par la suite. Certaines études ont montré que pour certains gros projets qui durent dans le temps, pour une ligne écrite, on en a 10 lues.

4 Modélisation

Bibliographie :
– UML2 Compact de Heide Balzert (disponible à la BU).
– `http://uml.free.fr`

Tout projet d'ampleur (et idéalement) même les plus modestes doivent être modélisés avant d'être implémentés. Cela permet d'avoir dès le début l'intégralité de l'architecture et de vérifier sa cohérence. En effet, un problème dans une méthode ou un algorithme nécessite une modification locale et n'impacte pas le reste du programme. En revanche, un problème d'architecture (par exemple quelle classe intéragit avec les autres) nécessite de transformer beaucoup de code.

Il existe plusieurs diagrammes standardisés proposés par divers consortiums. On choisit de présenter UML (Unified Modelling Language).

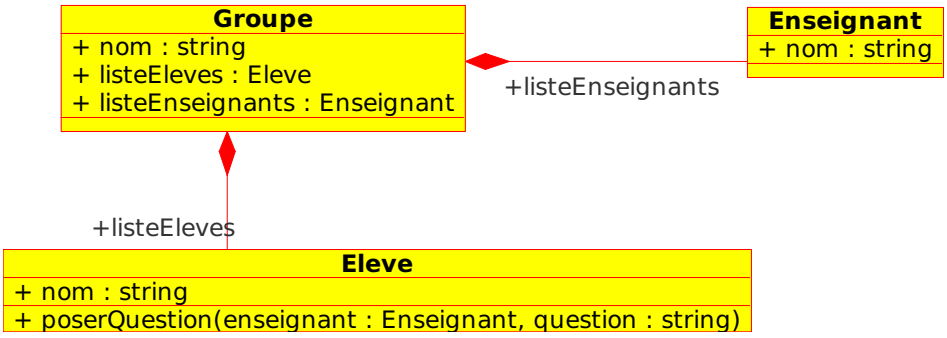
Des logiciels existent permetant de dessiner ces diagrames et de générer l'architecture globale du projet i.e. les classes avec les méthodes vides, ainsi que certaines spécifications en commentaire.

Nous allons donner quelques exemples de diagramme qui permettent de décrire les différentes facettes d'un logiciel tel les relation entre les composants du logiciel, avec le monde extérierur (humain ou autre logiciel), au fil du temps, ...

Le but ici n'est pas de comprendre la sémantique donnée pour chaque boite ou flèche par UML, mais plutôt qu'il existe des solutions pour chaque aspect du logiciel que l'on souhaite décrire.

4.1 Diagramme de classes

Un diagramme de classes représente les classes avec les attributs et les opérations ainsi que la généralisation ainsi que les associations entre classes.



La flèche avec losange pleine indique la composition.

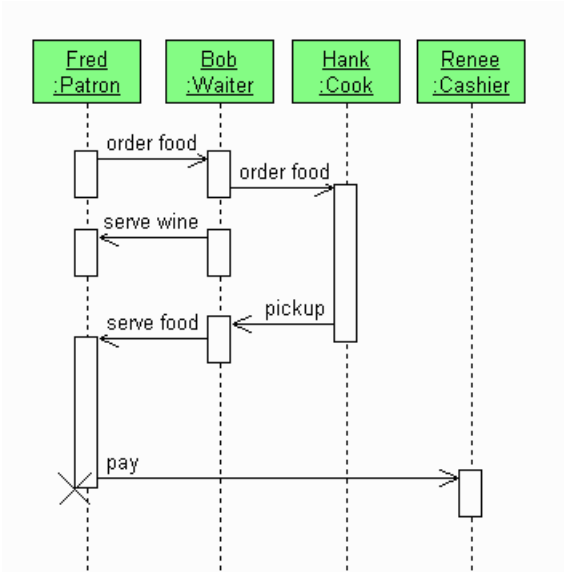
Ici, il n'est pas question de chose dynamique, ni de temps. C'est pour cela que l'on utilise façon complémentaire d'autres diagrammes.

4.2 Diagramme de cas d'utilisation

Ici on se place du point de vue des personnes interagissant avec le logiciel. Ceci est un exemple de Site Web :

4.3 Diagramme de séquence

Cette sorte de diagramme permet de voir dans le temps les intéractions entre les différents acteurs en montrant quelles méthodes sont utilisées.

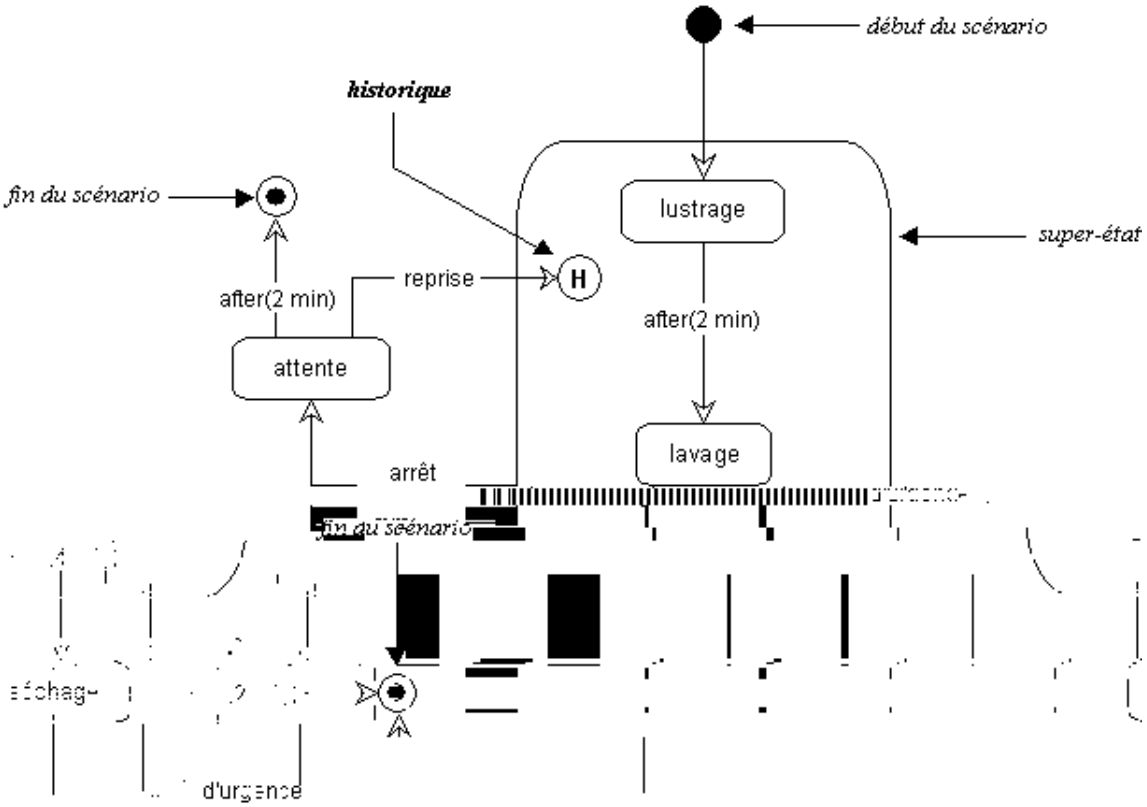


4.4 Diagramme d'états-transitions

Les diagrammes d'états-transitions permettent de décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composants ou avec des acteurs.

Un état se caractérise par sa durée et sa stabilité, il représente une conjonction instantanée des valeurs des attributs d'un objet.

Une transition représente le passage instantané d'un état vers un autre. Une transition est déclenchée par un événement. En d'autres termes : c'est l'arrivée d'un événement qui conditionne la transition.



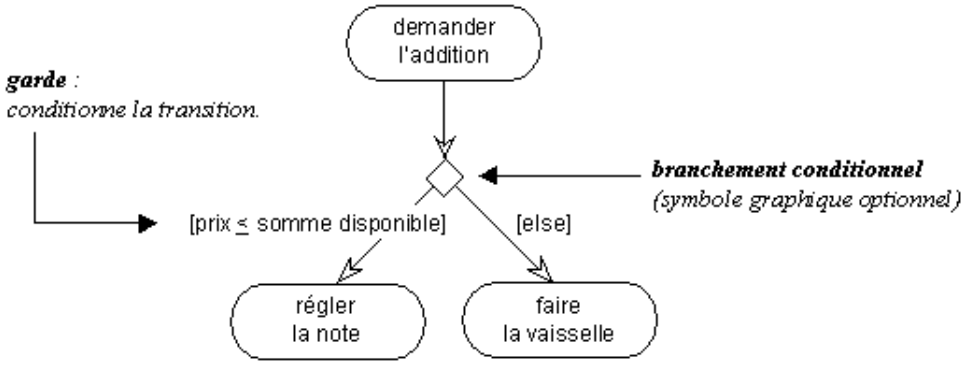
4.5 Diagramme d'activités

Une activité représente une exécution d'un mécanisme, un déroulement d'étapes séquentielles.

Le passage d'une activité vers une autre est matérialisé par une transition.

Les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre (elles sont automatiques).

Transition conditionelle :

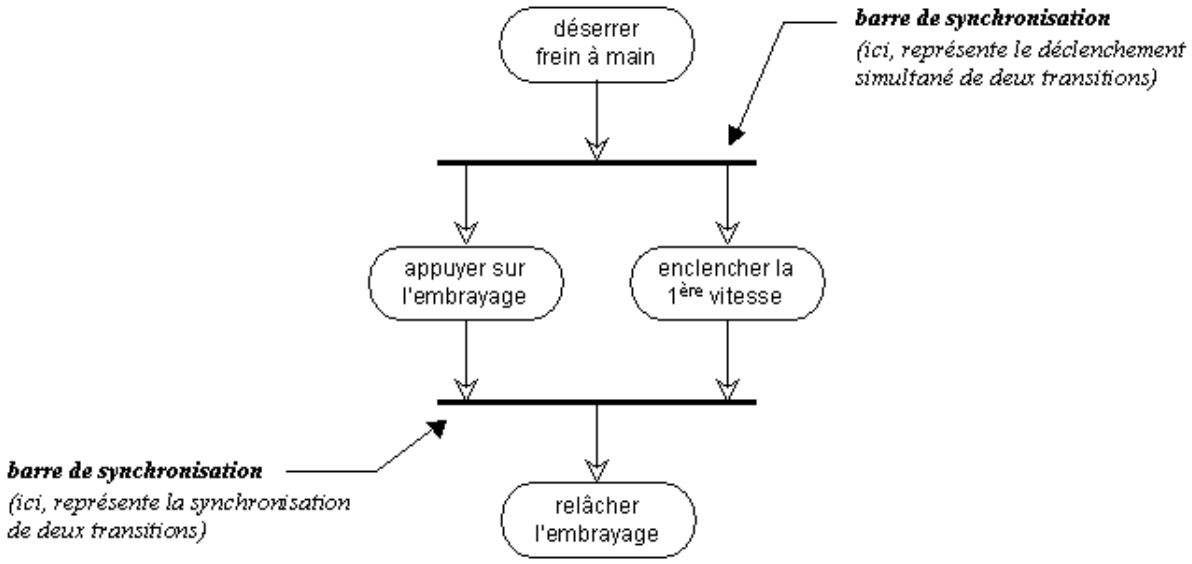


Synchronisation : Il est possible de synchroniser les transitions à l'aide des "barres de synchronisation" (comme dans les diagrammes d'états-transitions).

Une barre de synchronisation permet d'ouvrir et de fermer des branches parallèles au sein d'un flot d'exécution :

- Les transitions qui partent d'une barre de synchronisation ont lieu en même temps.
- On ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent.

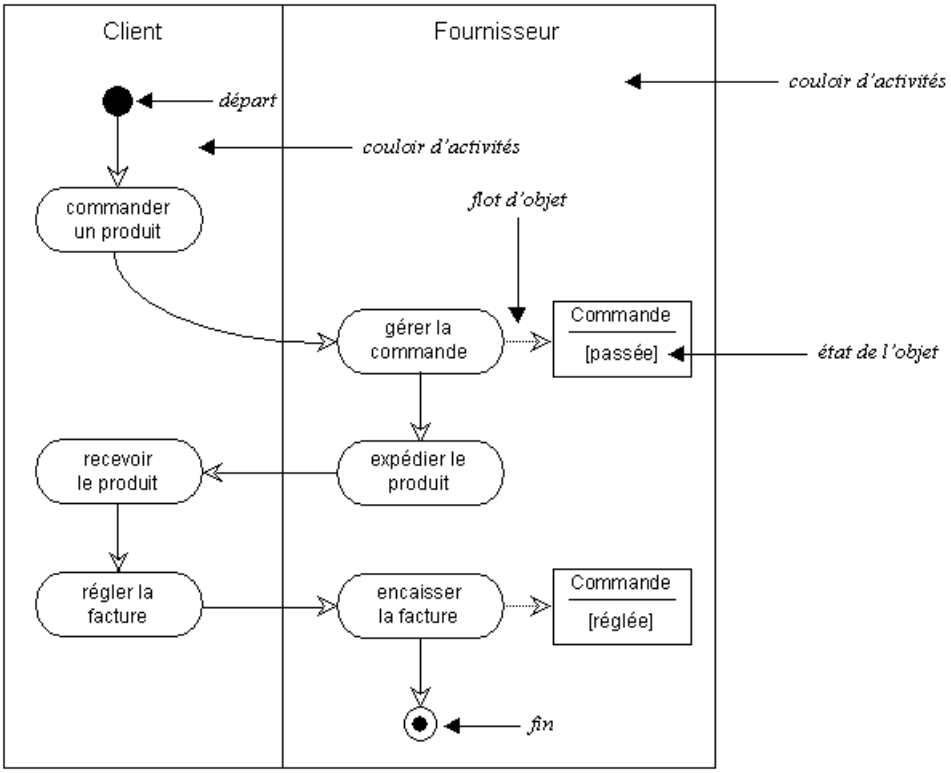
Exemple :



On peut imaginer décrire le protocole de notre projet où le serveur doit attendre les deux réponses des clients à la question de savoir si l'on rejoue.

Couloirs d'activités : Afin d'organiser un diagramme d'activités selon les différents responsables des actions représentées, il est possible de définir des "couloirs d'activités".

Il est même possible d'identifier les objets principaux, qui sont manipulés d'activités en activités et de visualiser leur changement d'état.



5 Modèle-Vue-Controleur

Source : Wikipedia

Le Modèle-Vue-Controleur (MVC) est une architecture et une méthode de conception qui organise l'interface homme-machine (IHM) d'une application logicielle. Ce paradigme divise l'IHM en un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (logique de contrôle, gestion des événements, synchronisation), chacun ayant un rôle précis dans l'interface.

Le modèle Le modèle représente le comportement de l'application : traitements des données, interactions avec la base de données, etc. Il décrit ou contient les données manipulées par l'application. Il assure la gestion de ces données et garantit leur intégrité. Dans le cas typique d'une base de données, c'est le modèle qui la contient. Le modèle offre des méthodes pour mettre à jour ces données (insertion, suppression, changement de valeur). Il offre aussi des méthodes pour récupérer ces données. Les résultats renvoyés par le modèle sont dénués de toute présentation.

La vue La vue correspond à l'interface avec laquelle l'utilisateur interagit. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toutes les actions de l'utilisateur (clic de souris, sélection d'une entrée, boutons, etc). Ces différents événements sont envoyés au contrôleur. La vue n'ef-

– la requête envoyée depuis la vue est analysée par le contrôleur (par exemple un clic de souris pour lancer un traitement de données),
– le contrôleur demande au modèle approprié d'effectuer les traitements et notifie la vue que la requête est traitée (via par exemple un handler ou callback),
– la vue notifiée fait une requête au modèle pour se mettre à jour (par exemple affiche le résultat du traitement via le modèle).

Si vous utilisez Swing (bibliothèque gérant l'affichage mais aussi la gestion des clics, ...) pour le projet, vous utiliserez implicitement le MVC. Il existe plein d'exemples de Swing sur le site de Sun, si vous êtes curieux et désirez faire de belles extensions à votre projet.

6 Design Patterns

Bibliographie :
– Wikipedia
– Les design patterns en Java (disponible à la BU)

Un desgin pattern (DP) ou en français patron de conception ou motif de conception est un concept de génie logiciel destiné à résoudre les problèmes récurrents suivant le paradigme objet.

Les patrons de conception décrivent des solutions standard pour répondre à des problèmes d'architecture et de conception des logiciels. Tout comme les algorithmes, il ne s'agit pas de fragments de code source, puisque les patrons de conception sont le plus souvent indépendants du langage de programmation. En revanche là où un algorithme s'attache à décrire d'une manière formelle comment résoudre un problème particulier, les patrons de conception décrivent des procédés de conception généraux et permettent en conséquence de mieux capitaliser l'expérience appliquée à la conception logicielle. Il a donc également une grande influence sur l'architecture logicielle d'un système.

6.1 Singleton

Le singleton est un DP dont l'objet est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsque l'on a besoin d'exactement un objet pour coordonner des opérations dans un système.

On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà. Dans beaucoup de langages de type objet, il faudra veiller à ce que le constructeur de la classe soit privé, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée.

Attention, cette solution n'est pas valide dans le cas de programmes multi-threadés !

Une solution, en Java :

```
public class Singleton {
     private static Singleton instance = null;

     /**
      * La présence d'un constructeur privé supprime
      * le constructeur public par défaut.
      */
     private Singleton() {}

     /**
      * Retourne l'instance du singleton.
      */
     public final static Singleton getInstance() {
         if (instance == null)
             instance = new Singleton();
         return instance;
     }
 }
```

Une autre solution :

```
 public class Singleton {
     /**
      * Création de l'instance au niveau de la variable.
      */
```

```
    private static final Singleton instance = new Singleton();

    /**
     * La présence d'un constructeur privé supprime
     * le constructeur public par défaut.
     */
    private Singleton() {}

    /**
     * Dans ce cas présent, le mot-clé synchronized n'est pas utile.
     * L'unique instanciation du singleton se fait avant
     * l'appel de la méthode getInstance(). Donc aucun risque d'accès concurrents.
     * Retourne l'instance du singleton.
     */
    public final static Singleton getInstance() {
        return instance;
    }
}
```

6.2 Fabrique

La fabrique (factory) est un patron de conception créationnel utilisé en programmation orientée objet. Comme les autres modèles créationnels, la fabrique a pour rôle l'instanciation d'objets divers dont le type n'est pas prédéfini : les objets sont créés dynamiquement en fonction des paramètres passés à la fabrique.

Comme en général, les fabriques sont uniques dans un programme, on utilise souvent le patron de conception singleton pour les implémenter.

Dans l'exemple suivant en Java une classe « fabrique » des objets dérivés de la classe Animal en fonction du nom de l'animal passé en paramètre. Il est également possible d'utiliser une interface comme type de retour de la fonction.

```
public class FabriqueAnimal {

  Animal getAnimal(String typeAnimal) throws ExceptionCreation {
    if (typeAnimal.equals("chat")) {
      return new Chat();
    } else if (typeAnimal.equals("chien")) {
      return new Chien();
    }
    throw new ExceptionCreation("Impossible de créer un " + typeAnimal);
  }

}
```

On peut imaginer l'utilisation d'une fabrique dans le cas où un client aimerait récupérer un objet mais ne sait pas lequel. Par exemple, le client peut demander un chien à poils longs qui soit un bon gardien mais ne sait pas à quelle chien (à quelle classe) cela correspond. Il peut donc fournir ces informations et récupère une sous classe d'un animal qui correspond à ce qu'il voulait.

6.3 Façade

Une façade a pour but de cacher une conception et une interface complexe difficile à comprendre (cette complexité étant apparue "naturellement" avec l'évolution du sous-système en question). La façade permet de simplifier cette complexité en fournissant une interface simple du sous-système. Habituellement, la façade est réalisée en réduisant les fonctionnalités de ce dernier mais en fournissant toutes les fonctions nécessaires à la plupart des utilisateurs.

On peut imaginer une librairie faisant des opérations de bas niveau. On peut alors créer une classe qui va s'abstraire de certains détails pour faciliter l'utilisation ou alors les tests. Une façade avec uniquement des méthodes statiques est appelée un utilitaire dans le langage UML.

6.4 Et plus

Il existe 23 patrons standards que l'on peut trouver dans le livre original introduisant les design patterns (en 95). On peut les classer en trois grands catégories :

- Création : ils définissent comment faire l'instanciation et la configuration des classes et des objets. (comme singleton ou fabrique)
- Structure : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation). (comme façade ou encore MVC)
- Comportement : ls définissent comment organiser les objets pour que ceux-ci collaborent

Toutes les informations sont trouvables sur Wikipedia.