Chapitre 1 : Limites du paradigme objet pour la conception

« La perfection n'est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à enlever. » ***Antoine de Saint-Exupéry***

1. Motivations du cours

- Vers une conception des logiciels de qualité professionnelle.
- Recherche de « l'excellence technique » d'une conception. Cf. artisanat du logiciel (*software craftsmanship*).

1.1. Prérequis

- Paradigme : conception orientée objet.
- Formalisme : langage de modélisation UML.
- Langage de programmation orientée objet : Java.

2. Les six piliers du paradigme objet

1. Abstraction
2. Modularité
3. Encapsulation
4. Héritage
5. Polymorphisme
6. Composition

2.1. Abstraction

Procédé de construction d'un modèle simplifié d'un système complexe tout en conservant ses fonctions essentielles.

- Identifier le **comportement attendu** des objets d'intérêt du système.
- Focaliser sur la vue externe des objets en masquant sa complexité interne.

2.2. Modularité

Procédé de construction d'un système par assemblage de modules compacts.

- Le développement de systèmes complexes émerge de la combinaison de modules plus simples.
- La notion de module en langage objet se décline en Fonction, Classe, Paquet, Composant (ie, un ensemble de classes qui représentent une fonctionnalité, généralement stockées dans une bibliothèque) et Nœud (un composant physique ou virtuel, tel que serveur, client, ordinateur,

téléphone ou imprimante).

2.3. Encapsulation

Principe de séparation de l'interface contractuelle d'une abstraction avec son implémentation.

- Emballer l'information de manière à masquer la représentation interne laissant ainsi toute latitude pour modifier cette représentation sans impact sur le reste du code.
- Toutes les interactions avec l'objet sont faites par l'intermédiaire d'une **interface publique** (ie, la liste des services rendus par la classe).

2.4. Héritage

Procédé de réutilisation par lequel une nouvelle fonctionnalité est obtenue par extension de l'implémentation d'une classe existante.

- La classe de généralisation définit les services abstraits et capture explicitement les attributs et les méthodes communes.
- La classe de spécialisation réifie les services abstraits et étend l'implémentation avec des attributs et des méthodes supplémentaires.

2.5. Polymorphisme

Capacité des objets appartenant à une même hiérarchie à répondre aux appels de méthodes de même nom, chacun selon le comportement spécifique de sa classe.

- Les objets doivent présenter une interface compatible (ie, la même signature pour les méthodes).
- Le programme n'a pas à connaître la classe exacte de l'objet à l'avance, de sorte que le comportement est mis en œuvre lors de l'exécution.

2.6. Composition

Procédé de réutilisation par lequel une nouvelle fonctionnalité est obtenue en combinant les services de plusieurs objets.

- La composition encapsule plusieurs objets à l'intérieur d'un autre.
- La nouvelle fonctionnalité est obtenue en déléguant sa réalisation aux différents objets de la composition.

3. Les promesses du paradigme objet

Le paradigme de conception orientée objet a pour ambition :

1. **Développabilité** : confort avec lequel le logiciel peut être développé.
   - L'abstraction et la modularité donnent une vision décorrélée du logiciel en niveaux d'abstraction et en composants qui permet une approche cartésienne du développement : décomposition de problème en sous-problèmes indépendants plus simples. Le polymorphisme permet une programmation à un niveau abstrait sans se soucier des détails de l'implémentation concrète.

2. **Extensibilité** : faculté d'étendre simplement les fonctionnalités d'un logiciel sans compromettre son intégrité et sa fiabilité.
    - La modularité, l'abstraction et le polymorphisme permettent de décomposer une application en sous-parties plus autonomes et plus faciles à étendre.
3. **Maintenabilité** : facilité avec laquelle on peut corriger des erreurs ou des manques.
    - Grâce à l'encapsulation et la composition, les modifications d'une partie du code n'impactent pas les autres parties du code.
4. **Réutilisabilité** : aptitude d'un logiciel à être réutilisé en partie pour de nouvelles applications.
    - Le développement d'un nouveau système tire avantage de l'assemblage par héritage ou par composition de modules intègres.

4. Périls de l'approche objet

Mais, le paradigme et le langage seuls ne suffisent pas à assurer ces promesses. Par exemple, l'encapsulation est très souvent mise à mal par l'utilisation d'attributs non privés ou d'accesseurs / mutateurs qui révèlent la représentation interne et empêchent sa modification.
Sans règles et principes supplémentaires, une conception finit inévitablement en « usine à gaz » (*spaghetti code* en anglais) et le logiciel finit par pourrir (*software rot* en anglais). Un code mal conçu présente de graves défauts tels que :

- **Rigidité** : le logiciel est difficile à faire évoluer. L'extension impacte de nombreuses parties de l'application générant un coût d'extension élevé.
- **Fragilité** : le logiciel est difficile à maîtriser. Le changement provoque des erreurs éparses non prévisibles. Les modifications sont de plus en plus risquées.
- **Immobilité** : le logiciel est difficile à réutiliser. À chaque nouveau développement, il faut repartir de zéro. Le coût de développement ne profite pas de l'expérience accumulée.

Bien sûr, ces problèmes sont d'autant plus sensibles que l'application est conséquente. Mais, ils sont déjà perceptibles pour des applications de quelques milliers de lignes de code.

5. Critères de qualité d'une conception

La qualité d'un logiciel se réfère à deux critères absolus : **cohésion** et **couplage**. Ils portent sur la notion de module : classe, méthode, paquet, composant ou nœud. Ces deux critères sont difficiles à mesurer automatiquement. Toutefois, il existe des métriques qui permettent d'apprécier ces critères. Mais, ces métriques ne sont que des indicateurs et pas des mesures de qualité absolues.

5.1. Critère 1. Cohésion

Il caractérise l'étendue de la responsabilité d'un module, autrement dit, le degré d'indissociabilité des éléments d'un module. Il faut maximiser le degré de cohésion dans une conception.
Les questions associées sont :

- Quel est le but du module ?
- Fait-il une ou plusieurs choses ?

Le nombre d'attributs est un bon indicateur du degré de cohésion.

5.1.1. Exemple d'une métrique : Tight Class Cohesion (TCC)

Le principe consiste à compter le nombre relatif de paires de méthodes qui accèdent directement aux mêmes attributs de la classe. Cela permet d'identifier les groupes de méthodes indépendantes.

- Mesure : TCC = NDC / NP
  - NDC : le nombre de paires de méthodes directement liées.
  - NP : le nombre total de paires de méthodes.
- On considère qu'une cohésion est trop faible si TCC < 1/3.

5.2. Critère 2. Couplage

Il caractérise la force d'interaction entre les modules, autrement dit le degré de dépendance de chaque module aux autres modules. Il faut minimiser le couplage dans une conception.
Les questions associées sont :

- Comment les modules collaborent-ils ensemble ?
- Qu'ont-ils besoin de connaître les uns des autres ?

La liste des importations est un bon indicateur de la force de couplage, e.g, nombre de `#include` en C++ et C# ou `import` en Java et Python.

5.3. Exemple d'une métrique : Access To Foreign Data (ATFD)

Le principe consiste à compter la proportion de méthodes d'une classe qui accèdent directement ou via des accesseurs/mutateurs à des attributs d'autres classes.

- On considère qu'un couplage est trop fort si ATFD > 5 %.

6. Objectifs du cours

Ce cours poursuit deux objectifs principaux :

1. Développer un esprit critique sur la conception logicielle.
   - Savoir évaluer la qualité d'une conception et se rendre compte qu'elle a tendance à pourrir.
   - Savoir produire une conception qui soit :
     - extensible,
     - maintenable,
     - réutilisable.
   - Savoir réutiliser une conception :
     - connaître et savoir apprécier le patrimoine en matière de modélisation,
     - adapter une solution existante.
   - Savoir organiser son code en paquets pour favoriser :
     - le développement,
     - la maintenance,
     - la réutilisation.
2. Intégrer le fait que le code est malléable et doit sans cesse être remodelé pour garantir la meilleure qualité.