
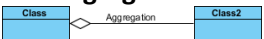



Nom	Définition	Utilité
Classe abstraite	<p>Classe particulière :</p> <ul style="list-style-type: none"> - Dont implémentation pas complète - Pas instanciable directement <p>Règle : Au moins une méthode abstraite (avec signature mais pas d'implémentation) rend sa classe abstraite.</p>	<p>Réutilisation du code. Les classes dérivées d'une classe abstraite, possèdent un ensemble d'éléments en commun : il s'agit de l'abstraction d'un concept, le design. Ces classes peuvent :</p> <ul style="list-style-type: none"> - Le spécialiser en définissant l'implémentation des méthodes abstraites de la classe parente (overriding) - Continuer à être abstraite
Héritage	<p>Mécanisme qui permet à des classes d'hériter les attributs et les méthodes d'une (héritage simple) ou plusieurs classes parentes (héritage multiple). Les objets de la classe dérivée sont aussi des objets de la classe parente.</p>	<p>L'héritage permet :</p> <ul style="list-style-type: none"> • Code facilement maintenable ; • Polymorphisme ; • Overriding.
Héritage multiple	<p>Il s'agit d'un mécanisme rare (présent dans une minorité de langages comme par exemple C++ ou Python) dans lequel une classe peut hériter de plusieurs classes.</p>	<p>Permet de combiner plusieurs comportements. Par exemple, un avion est un objet motorisé mais également un objet volant.</p>
Interface	<p>L'interface d'un objet est un « contrat de service », un ensemble de comportements génériques qu'un objet doit suivre.</p>	<p>Permet de baisser le couplage. Permet de définir les habilités périphériques d'une classe.</p>
Polymorphisme	<p>Il s'agit d'un concept qui permet d'envoyer le même message et recevoir des réponses différentes en fonction du type de l'objet.</p>	<p>Permet une programmation plus générique car, le message peut être envoyé à un objet sans se soucier de comment celui va réagir.</p>
Overriding	<p>Mécanisme applicable dans une classe dérivée.</p>	<p>Permet à une classe dérivée <u>de spécialiser des comportements de la classe parente</u> ou d'une super-classe à travers la surcharge des méthodes.</p>
Overloading	<p>Mécanisme procédurale (des fonctions).</p>	<p>Permet d'avoir plusieurs fonctions de même nom mais signature différente.</p>

Self/this	Super
Référence vers l' <u>objet courant</u> .	Pointe au parent de la classe où la méthode réside.
Lien <u>dynamique</u> (car l'objet pourrait être une classe fille par exemple).	Lien <u>statique</u> (car il suffit de regarder la classe dont on hérite et on a trouvé).
<u>Pas connu</u> au moment de la compilation.	<u>Connu</u> à la compilation car lien statique.
Méthod lookup commence dans la classe de l'objet <u>auquel</u> le <u>message</u> est <u>envoyé</u> .	Méthod lookup commence dans la <u>superclasse</u> de la classe où la méthode qui appelle « super » est définie.
Attention → Ne référence <u>pas toujours</u> la classe la <u>plus basse</u> dans la hiérarchie.	Attention → N'est <u>pas</u> le <u>parent</u> de l'objet courant.
Sert au polymorphisme. Permet de s'assurer qu'on appellera la <u>méthode courante</u> si elle existe (sinon on cherche chez le parent).	Sert au polymorphisme. Permet de <u>réécrire</u> ou <u>enrichir</u> des méthodes du père.

Requirements Collection	On demande au client ce dont il a besoin.
Analysis	« WHAT » - Les analystes modélisent et précisent les différents besoins.
Design	« HOW » - Les architectes conçoivent l'architecture de l'application, modélisent et précisent une solution.
Implémentation	Les codeurs implémentent la solution.
Testing	Vérification de la solution en cherchant le nombre maximum d'erreurs.
Maintenance	Corrective, de compatibilité, perfective.

Principaux problèmes de la méthode « Waterfall »	
Couches	Division en couches rend l'approche très lente.
Communication	les nombreuses couches communiquent par documents, pas souvent mis à jour.
Coût de correction	Le coût de correction est exponentiel car il faut remonter la cascade.
Codeurs limités	Le codeur ne connaît pas la finalité.

Association 	Indique que deux objets sont en relation : se connaissent et peuvent s'envoyer des messages. Exemple hors cours : <u>Un docteur et un patient.</u>
Agrégation 	Indique qu'un objet fait partie d'un autre. L'une partie existe indépendamment de l'autre. Exemple du cours : <u>Une flotte constituée de plusieurs bateaux.</u> Les bateaux font partie d'une flotte. Exemple hors cours : <u>Une classe possède des étudiants.</u>
Composition 	Sorte d' agrégation plus forte . Si on détruit le père, il faut tout détruire. Exemple du cours : <u>Un livre est composé de pages.</u> Exemple hors cours : <u>Une maison est composée par chambres.</u>

Modélisation statique	Montre la <u>structure</u> du système.
Diagrammes statiques	Class Diagram, Object Diagram, Deployment Diagram, Component Diagram
Modélisation dynamique	Montre l' <u>évolution</u> du système avec l'envoi des messages.
Diagrammes dynamiques	Sequence Diagram, Collaboration Diagram, State Diagram, Activity Diagram

Vues	Utilisées pour décrire le système du point de vue des différentes <u>parties intéressées</u> , telles que les utilisateurs finaux, les développeurs, etc. Description
1) Use case view	<u>QUI ? QUOI ?</u> Montre les fonctionnalités du systèmes vues par les acteurs externes.
2) Logical view	<u>COMMENT ?</u> ...les fonctionnalités sont conçues ? Montre la définition du système vu de l'intérieur
3) Component view	<u>QUELLES COMPOSANTES</u> sont utilisés dans la solution finale ? Montre l'organisation des composantes du codes et des dépendances entre modules.
4) Concurrency view	<u>COMMENT CES COMPOSANTES</u> interagissent entre eux ? Vue temporelle et technique. Montre la notion de taches concurrentes et de synchronisation.
5) Deployment view	Montre l' <u>architecture physique</u> de chaque élément du système.

Sequence Diagram	Collaboration Diagram
Représentent les <u>séquences de communication</u> entre les objets.	
Met l'accent sur le classement des messages par ordre chronologique.	Met en évidence la notion de structure, d'espace.

Différence entre message et méthode	Les méthodes décrivent le comportement à la réception d'un message par un autre objet
--	---

Exigence fonctionnelles	Exigence qui définit une fonction du système à développer. Ce que le système doit faire.	
Exigence non fonctionnelles	Exigence qui caractérise une propriété (qualité) du système. N'ajoute pas des services.	
	Exigences du Produit	Robustesse
		Performance
		Facilité d'utilisation
	Exigences Organisationnelles	Implémentation
		Livraison
	Exigences Externes	Obligations imposées par la législation
		Exigences éthiques

Cohésion	Mesure de combien les éléments d'un module/classe sont fortement corrélées. Doit être haut , afin de définir un module compact.
Couplage	Degré de dépendance entre plusieurs modules. Doit être bas , afin qu'une classe connaisse le moins possible des autres, sinon effectuer des changements sera difficile.
Loi de Déméter (Principe de connaissance minimale)	On peut envoyer des messages à : <ul style="list-style-type: none"> - Un argument qui est passé à mes méthodes ; - Un objet crée dans les méthodes ; - self, this ;
	Exemple : <code>book.pages().last().text()</code>
	Solution : <code>book.textOfLastPage()</code> Avantages : Toute méthode qui « dit » au lieu de « demander » est découplée de son environnement. Le code en résulte facilement maintenable.

Test Unitaire	Il s'agit d'une procédure qui permet de vérifier le correct fonctionnement d'une partie spécifique du programme.	
	Qualités :	Avoir un comportement déterministe et répétable, sinon les résultats ne voudront rien dire
		Être moins sensible aux changements du code, sinon la rédaction des tests devient complexe et peu déterministe.
		Ne pas nécessiter d'intervention humaine, de telle sorte que le test puisse tourner de façon autonome pendant longtemps.

Précondition	Doit être vraie avant le traitement d'un message (avant de rentrer dans une partie du code). Contrainte pour le monde extérieur, soulagement pour l'objet. Sert à être certain des conditions nécessaires pour le bon fonctionnement d'une méthode.
Postcondition	Doit être vraie après le traitement d'un message. Contrainte pour l'objet, bénéfice pour l'appelant. Permet d'être certain que l'output est celui attendu.
Invariant	Condition qui doit être vraie à tout moment du programme. Exemple : « La taille du stack est toujours ≥ 0 ». Si un jour sera faux, alors on a commis une erreur. Permet de tester le code avant de l'implémenter.

Extreme Programming	Technique de programmation fortement itérative où d'abord on écrit les tests, puis le code. Elle est efficace car permet de clarifier les idées avant l'implémentation.
----------------------------	--

Design Pattern	Chaque Design Pattern décrit un problème récurrent (pas d'implémentation) et offre le cœur d'une solution (pas ready-made).	
	Composite Pattern	Le but est de <u>modéliser un group d'objets</u> qui sont traités comme une seule instance d'un seul type d'objet. Permet d'avoir une <u>structure d'arbre</u> telle que chaque nœud s'occupe d'une tâche.
	Observer Pattern	Le but est de modéliser un système où les objets liés sont <u>mis au courant</u> si l'un entre eux change. L'utilité est d' <u>automatiquement</u> les <u>mettre à jour</u> .
	Avantages	Efficacité : Les solutions ont été amplement démontrées et sont très efficaces Communication : Le design pattern permet d'avoir un code plus lisible, maintenable.