UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Jour 2

L’architecture logicielle

L'architecture logicielle représente l'organisation des composants d'un système logiciel ainsi que les interactions entre ces composants. Elle définit les fondations d'un logiciel, à partir desquelles toutes les décisions de conception sont prises. Une architecture bien pensée permet de répondre aux exigences fonctionnelles et non fonctionnelles d'une application.

Pourquoi a-t-on besoin d’une bonne architecture ?

1. **Faciliter la maintenance** :
   Une bonne architecture réduit la complexité du logiciel et facilite la maintenance à long terme. Les systèmes évoluent constamment, et il est essentiel que les nouvelles fonctionnalités puissent être ajoutées sans nécessiter de changements majeurs dans la structure du code existant. Cela permet d'éviter la création de "code spaghetti", c'est-à-dire un code où tout est interconnecté et où une modification dans une partie du système peut entraîner des effets indésirables ailleurs.
2. **Améliorer la lisibilité et la compréhension du code** :
   Une architecture claire et bien définie permet aux développeurs de comprendre rapidement le fonctionnement global du système. Le code devient plus lisible, les intentions sont plus explicites, et il est plus facile de naviguer dans les différentes parties du logiciel. Une bonne architecture impose souvent des règles sur la séparation des responsabilités (par exemple, avec des couches comme l'interface utilisateur, la logique métier, et la gestion des données), ce qui aide à isoler les préoccupations et à organiser le code de manière logique.

L’architecture logicielle

3. **Assurer l’évolutivité** :
   À mesure qu'un logiciel grandit en termes de complexité et de fonctionnalités, l'architecture doit permettre cette évolution sans compromettre les performances ou la stabilité. Une bonne architecture facilite l’ajout de nouvelles fonctionnalités ou modules sans devoir tout réécrire. En revanche, une mauvaise architecture rend l’évolution complexe, coûteuse, et peut même conduire à la nécessité de réécrire tout ou partie du système.

4. **Favoriser la réutilisabilité** :
   Une architecture modulaire et bien conçue permet de réutiliser certaines parties du code dans d'autres projets ou au sein d'autres modules du même projet. Par exemple, en séparant bien la logique métier du reste du système, il devient possible de réutiliser cette logique dans différentes applications (web, mobile, etc.) ou de la remplacer si nécessaire sans impacter l'intégralité du système.

5. **Réduire les risques et les coûts** :
   En ayant une architecture solide dès le début, nous réduisons les risques de voir le projet échouer ou devenir coûteux à maintenir. Une mauvaise architecture peut conduire à des délais importants pour corriger les bugs, intégrer de nouvelles fonctionnalités ou même maintenir la stabilité de l'application. Ces risques peuvent engendrer des surcoûts non prévus et diminuer la satisfaction des clients ou des utilisateurs finaux.

L’architecture logicielle

6. **Permettre une meilleure testabilité** :
   Un des principes fondamentaux du clean code est la testabilité. Une architecture bien définie permet de tester facilement chaque composant de manière isolée. L'isolation des responsabilités au sein du code facilite la création de tests unitaires ou d'intégration, garantissant ainsi que chaque brique du logiciel fonctionne correctement indépendamment du reste du système.

7. **Faciliter la collaboration entre les équipes** :
   Une bonne architecture impose des conventions et des règles claires sur la façon dont les composants interagissent entre eux. Cela aide différentes équipes à collaborer de manière efficace, car elles peuvent travailler sur différents modules ou parties du système sans risquer de créer des conflits ou des dépendances non maîtrisées.

4 notions importantes pour une application propre

1. Couplage faible

Le **couplage** fait référence à la manière dont les différents modules ou classes d'une application dépendent les uns des autres. Un couplage faible signifie que chaque module dépend le moins possible d'autres modules. Cela rend les modules indépendants et facilite leur maintenance ou modification sans affecter les autres parties de l'application.

Pourquoi c’est important :

- Réduit l'impact des modifications sur l'ensemble du système.
- Facilite la réutilisation des modules dans d'autres projets.
- Permet de tester des composants de manière isolée.

4 notions importantes pour une application propre

2. Cohésion

La **cohésion** fait référence à la mesure dans laquelle les responsabilités d'une classe ou d'un module sont étroitement liées. Une classe est dite fortement cohésive lorsqu'elle n'a qu'une seule responsabilité claire et que toutes ses méthodes sont directement liées à cette responsabilité.

Pourquoi c’est important :

- Facilite la lisibilité et la compréhension du code.
- Rend le code plus simple à maintenir.
- Réduit les risques d’introduire des erreurs lors des modifications.

4 notions importantes pour une application propre

3. Changements locaux

Les **changements locaux** signifient que les modifications dans une partie du code ne devraient affecter que des composants spécifiques, et non l'ensemble du système. Cela est souvent le résultat d'une bonne modularité, d'un faible couplage, et d'une forte cohésion.

Pourquoi c’est important :

- Réduit le risque d'introduire des bugs lors des modifications.
- Facilite l'implémentation de nouvelles fonctionnalités.
- Accélère le développement.

4 notions importantes pour une application propre

4. Nommage

Le **nommage** est l'art de donner des noms significatifs et explicites aux variables, méthodes, classes, et autres éléments du code. Un bon nommage rend le code plus lisible et compréhensible sans avoir à parcourir tout le code.

Connaître les principes de programmation objet SOLID

1. Single Responsibility Principle (SRP) — Principe de responsabilité unique

Le **Single Responsibility Principle** stipule qu'une classe ou un module ne doit avoir qu'une seule raison de changer, c'est-à-dire qu'elle doit avoir une seule responsabilité clairement définie.

Pourquoi c’est important :

- **Simplifie la maintenance** : Quand une classe ne fait qu'une seule chose, elle est plus facile à comprendre et à modifier.
- **Réduit les risques d'erreur** : En limitant la portée d'une classe, on minimise les effets de bord et les risques de bugs lorsque cette classe est modifiée.

Connaître les principes de programmation objet SOLID

2. Open-Closed Principle (OCP) — Principe ouvert/fermé

Le **Open-Closed Principle** stipule que les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes à l'extension mais fermées à la modification. Cela signifie que l'on doit pouvoir ajouter de nouvelles fonctionnalités sans avoir à modifier le code existant.

Pourquoi c’est important :

- **Réduit le risque d'introduire des bugs** : En ne modifiant pas le code existant, on évite les régressions potentielles.
- **Facilite l'ajout de nouvelles fonctionnalités** : Le système peut évoluer sans perturber les fonctionnalités existantes.

Connaître les principes de programmation objet SOLID

3. Liskov Substitution Principle (LSP) — Principe de substitution de Liskov

Le **Liskov Substitution Principle** stipule que les objets d'une sous-classe doivent pouvoir être remplacés par des objets de leur superclasse sans que cela n'affecte la fonctionnalité du programme. En d'autres termes, les sous-classes doivent respecter les contrats définis par leurs classes parentes.

Pourquoi c’est important :

- **Assure la cohérence** : Si une sous-classe ne peut pas être utilisée à la place de sa superclasse, le polymorphisme est cassé.
- **Améliore la fiabilité** : Les classes dérivées doivent respecter les attentes de la classe parent, sinon des comportements imprévus peuvent survenir.

Connaître les principes de programmation objet SOLID

4. Interface Segregation Principle (ISP) — Principe de ségrégation des interfaces

Le **Interface Segregation Principle** stipule qu'il est préférable de créer plusieurs interfaces spécifiques plutôt qu'une interface générale unique. Cela permet d'éviter que les classes implémentent des méthodes dont elles n'ont pas besoin.

Pourquoi c’est important :

- **Évite la surcharge des classes** : Les classes n'ont pas à implémenter des méthodes inutiles ou non pertinentes.
- **Simplifie le code** : Les interfaces plus petites et plus spécifiques sont plus faciles à comprendre et à utiliser.

Connaître les principes de programmation objet SOLID

5. Dependency Inversion Principle (DIP) — Principe d'inversion des dépendances

Le **Dependency Inversion Principle** stipule que les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions (interfaces). En d'autres termes, le code doit dépendre d'abstractions plutôt que de détails concrets.

Pourquoi c’est important :

- **Facilite la modification et la maintenance** : Le code de haut niveau ne change pas lorsque les modules de bas niveau changent.
- **Favorise l'inversion de contrôle** : Les dépendances sont injectées plutôt que créées directement au sein des modules de haut niveau.

Comprendre la notion de couplage

1. Héritage

L'**héritage** est un mécanisme en programmation orientée objet où une classe dérive d'une autre classe, héritant ainsi de ses attributs et méthodes. La classe dérivée est souvent appelée **sous-classe** ou **classe enfant**, et la classe dont elle hérite est appelée **superclasse** ou **classe parent**.

Avantages de l'héritage :

- Réutilisation du code : La sous-classe peut utiliser les fonctionnalités déjà définies dans la classe parent.
- Organisation hiérarchique : Il permet de structurer les classes de manière hiérarchique, facilitant ainsi la compréhension des relations entre classes.

Inconvénients de l'héritage :

- Couplage fort : La sous-classe dépend fortement de la superclasse, ce qui rend les modifications difficiles. Si nous modifions la superclasse, cela peut avoir des répercussions imprévues sur toutes les sous-classes.
- Difficulté à évoluer : Lorsque la hiérarchie devient trop complexe, l'héritage devient difficile à maintenir et à étendre.

Comprendre la notion de couplage

2. Composition

La **composition** est une technique où une classe inclut une instance d'une autre classe comme attribut. Cela signifie que la classe "composante" fait partie de la classe "contenante". Contrairement à l'héritage, la composition favorise un couplage plus faible et flexible, car la classe contenant ne dépend pas directement de la structure interne de la classe composante.

Avantages de la composition :

- Couplage faible : La classe dépend de l'interface ou de l'implémentation d'une autre classe, mais pas directement de ses détails internes.
- Flexibilité : La composition permet de changer facilement le comportement en remplaçant l'instance utilisée par une autre.
- Favorise la réutilisation : Les objets composés peuvent être réutilisés dans différents contextes sans modifier la structure des classes.

Comprendre la notion de couplage

3. Agrégation

L'**agrégation** est une forme plus faible de composition. Elle indique qu'une classe est liée à une autre, mais l'objet composant peut exister indépendamment de l'objet conteneur. En d'autres termes, une agrégation décrit une relation "a un" (has-a) où la classe contenante peut exister sans la classe composante, et inversement.

Avantages de l'agrégation :

- Couplage encore plus faible : L'objet agrégé peut exister indépendamment de l'objet qui l'agrège.
- Flexibilité : Comme la classe agrégée n'est pas fortement liée à la classe principale, elle peut être partagée entre plusieurs instances sans répercussions.

Comprendre la notion de couplage

Différence entre héritage, composition, et agrégation :

- **Héritage** : C'est une relation "est un" (is-a). La sous-classe hérite des attributs et comportements de la superclasse, mais cela crée un fort couplage entre les classes.
- **Composition** : C'est une relation "a un" (has-a) forte, où la classe contenante ne peut pas exister sans la classe composante. La composition est souvent préférée à l'héritage pour réduire le couplage.
- **Agrégation** : C'est une relation "a un" plus faible, où les objets agrégés peuvent exister indépendamment des objets contenant.

Les patrons de conceptions

1. Origine des patrons de conception

Les **patrons de conception** ont été popularisés par les membres du **"Gang of Four" (GoF)**, à savoir Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. Dans leur livre **"Design Patterns: Elements of Reusable Object-Oriented Software"**, publié en 1994, ils ont décrit 23 patrons de conception qui aident à résoudre des problèmes récurrents dans le développement logiciel orienté objet.

L'idée derrière les patrons de conception est de documenter des solutions éprouvées pour faciliter la réutilisation et la communication entre développeurs. En appliquant ces patrons, nous réduisons le risque de réinventer des solutions inefficaces ou maladroites.

Les patrons de conceptions

2. Les catégories de patrons de conception

Les patrons de conception se divisent en trois grandes catégories :

A. Patrons de création

Ces patrons se concentrent sur la manière dont les objets sont créés, en nous aidant à instancier des objets de manière contrôlée et flexible. Ils masquent la logique complexe derrière la création d'objets et permettent de découpler le code de l'instanciation directe.

B. Patrons structurels

Les patrons structurels concernent l'organisation des classes et des objets pour former des structures plus grandes et plus flexibles. Ils facilitent la composition des objets pour résoudre des problèmes tout en minimisant les dépendances entre eux.

C. Patrons comportementaux

Ces patrons se concentrent sur la communication entre les objets et la manière dont ils interagissent. Ils permettent de définir clairement la manière dont les objets collaborent, en simplifiant le flux de communication et les responsabilités.

Les patrons de conceptions

3. Liste des patrons de conception et descriptions

A. Patrons de création

1. **Singleton** :
   Garantit qu'une classe n'a qu'une seule instance, et fournit un point d'accès global à cette instance.
2. **Factory Method** :
   Définit une interface pour créer un objet, mais laisse les sous-classes décider quelle classe instancier. Cela permet de différer l'instanciation à des sous-classes.
3. **Abstract Factory** :
   Fournit une interface pour créer des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes.
4. **Builder** :
   Sépare la construction d'un objet complexe de sa représentation afin que le même processus de construction puisse créer différentes représentations.
5. **Prototype** :
   Crée de nouveaux objets en copiant (clonant) des objets existants plutôt qu'en créant de nouveaux objets à partir de zéro.

Les patrons de conceptions

B. Patrons structurels

1. **Adapter** :
   Permet à des interfaces incompatibles de fonctionner ensemble. Il fait office d'intermédiaire entre deux objets n'ayant pas la même interface.
2. **Bridge** :
   Sépare l'abstraction de son implémentation afin qu'elles puissent évoluer indépendamment.
3. **Composite** :
   Permet de composer des objets en structures arborescentes pour représenter des hiérarchies partielles ou totales. Il traite les objets individuels et leurs compositions de manière uniforme.
4. **Decorator** :
   Permet d'ajouter dynamiquement des responsabilités à un objet, sans modifier son code. Il offre une alternative plus flexible à l'héritage pour l'extension des fonctionnalités.

Les patrons de conceptions

5. **Facade** :
   Fournit une interface simplifiée à un ensemble de classes ou à un système complexe. Il masque la complexité et offre une interface plus accessible.
6. **Flyweight** :
   Réduit la consommation de mémoire en partageant autant que possible les données entre les objets similaires.
7. **Proxy** :
   Fournit un substitut ou un placeholder pour un autre objet afin de contrôler l'accès à cet objet.

Les patrons de conceptions

C. Patrons comportementaux

1. **Chain of Responsibility** :
   Évite de coupler l'expéditeur d'une requête à son destinataire en laissant plusieurs objets avoir la possibilité de traiter la requête. Les objets sont chaînés et la requête passe le long de la chaîne jusqu'à ce qu'un objet la prenne en charge.
2. **Command** :
   Encapsule une requête en tant qu'objet, ce qui permet de paramétrer les clients avec des requêtes, de mettre en file d'attente ou d'annuler des opérations.
3. **Iterator** :
   Fournit un moyen d'accéder séquentiellement aux éléments d'un objet composite sans exposer sa représentation sous-jacente.
4. **Mediator** :
   Définit un objet qui encapsule la manière dont un ensemble d'objets interagit, réduisant ainsi les dépendances directes entre eux.

Les patrons de conceptions

5. **Memento** :
   Capture et externalise l'état interne d'un objet sans violer l'encapsulation, permettant ainsi de restaurer cet état ultérieurement.
6. **Observer** :
   Définit une relation de dépendance entre des objets tels que lorsqu'un objet change d'état, tous ses dépendants en sont informés et mis à jour automatiquement.
7. **State** :
   Permet à un objet de changer son comportement lorsque son état interne change. L'objet semblera changer de classe.
8. **Strategy** :
   Définit une famille d'algorithmes, encapsule chacun d'eux, et les rend interchangeables. Cela permet de choisir dynamiquement un algorithme à utiliser.

Les patrons de conceptions

9. **Template Method** :
   Définit la structure d'un algorithme, mais laisse certaines étapes à des sous-classes. Cela permet de réutiliser du code tout en laissant la flexibilité de l'implémentation des étapes spécifiques.
10. **Visitor** :
    Permet de définir de nouvelles opérations sur des objets sans modifier leurs classes.

Les Values Objects

Les Value Objects (Objets de Valeur)

Les **Value Objects** sont un concept fondamental en conception orientée objet, particulièrement dans le cadre des architectures DDD (Domain-Driven Design). Contrairement aux entités, qui sont définies par leur identité unique, les **Value Objects** sont définis par leurs attributs. Ils représentent des objets simples, sans identité propre, qui encapsulent des valeurs et du comportement.

1. Principes des Value Objects

Les **Value Objects** suivent plusieurs principes clés :

- **Absence d'identité** : Contrairement à une entité, un objet de valeur n'a pas d'identité propre. Ce qui signifie que deux objets de valeur contenant les mêmes données sont considérés comme égaux.
- **Encapsulation des valeurs** : Un objet de valeur encapsule un ou plusieurs attributs, et son comportement est centré sur la manipulation de ces valeurs.
- **Invariance** : Un **Value Object** ne change pas d'état. Au lieu de modifier les valeurs internes, on crée de nouveaux objets de valeur pour refléter des modifications, ce qui favorise la sécurité et la prévisibilité du code.
- **Egalité basée sur les valeurs** : Deux **Value Objects** sont égaux si et seulement s'ils contiennent les mêmes valeurs, indépendamment de leur position en mémoire.

Les Values Objects

2. Immutabilité des Value Objects

Un principe fondamental des **Value Objects** est leur **immutabilité**. Cela signifie qu'une fois créé, un **Value Object** ne peut pas être modifié. Cela permet de garantir l'intégrité des objets de valeur et de faciliter le raisonnement sur le code.

Pourquoi l'immutabilité est importante :

- **Sécurité des données** : Puisque les objets de valeur ne peuvent pas être modifiés, il n'y a pas de risque de voir un objet changer d'état de manière inattendue.
- **Simplicité** : Les objets immuables éliminent une classe entière de bugs liés aux modifications d'état indésirées.
- **Facilité dans les environnements concurrents** : Dans les contextes multithreadés ou parallèles, les objets immuables sont intrinsèquement thread-safe, car leur état ne change jamais après création.

Les Values Objects

3. Effectuer des opérations en parallèle avec les Value Objects

Les **Value Objects**, en tant qu'objets immuables, sont particulièrement adaptés aux **opérations en parallèle**. Dans des environnements multithread ou lorsqu'on effectue des calculs parallèles, le fait qu'un **Value Object** ne puisse pas être modifié garantit qu'il peut être partagé en toute sécurité entre plusieurs threads.

Avantages des Value Objects dans le traitement parallèle :

- **Thread-safety** : Comme les **Value Objects** ne peuvent pas être modifiés, il n'y a pas de risques de conditions de course ou d'accès concurrentiel inapproprié.
- **Réduction des bugs liés au partage d'état** : L'immutabilité élimine les erreurs causées par des modifications concurrentes sur un même objet partagé.