Chapitre 1 : Introduction à l'Analyse Architecturale et à la Description de l'Architecture Logicielle

1. **Introduction** : Les échecs de projets informatiques sont souvent causés par une mauvaise gestion des besoins, comme des informations inexactes, des besoins incomplets, et une instabilité dans les besoins.

2. **Typologie des Besoins (modèle FURPS+)** : Dans le processus unifié RUP, les besoins sont classés en :

    - **F** (Fonctionnalité) : Fonctions, capacités, sécurité.
    - **U** (Utilisabilité) : Ergonomie, aide, documentation.
    - **R** (Fiabilité) : Pannes, reprise, prévisibilité.
    - **P** (Performance) : Temps de réponse, débit, etc.
    - **S** (Supportabilité) : Maintenance, portabilité, testabilité.
    - **+** (Autres facteurs) : Contraintes légales et technologiques.

3. **Définition de l'Architecture Logicielle** : L'architecture logicielle définit l'organisation du système, la sélection des composants, leur interaction et agencement en sous-systèmes.

4. **Analyse Architecturale** : Elle consiste à identifier et traiter les besoins non fonctionnels tout en les intégrant dans le contexte des besoins fonctionnels, par exemple, l'impact d'un calculateur de taxe sur la performance et la scalabilité du système.

5. **Types et Vues Architecturales** : On distingue l'architecture applicative (affectation des fonctions aux composants) et l'architecture système (configuration du matériel et OS). Ces vues sont exprimées dans RUP.

6. **Décisions Architecturales & Scénarios** : Les besoins critiques influencent l'architecture, avec des solutions élaborées à travers des scénarios de qualité (réponse mesurable aux stimuli externes).

7. **Description des facteurs architecturaux** :

    Nom du facteur
    ➡️ Un **objectif technique** ou une **propriété souhaitée** du système (ex. : disponibilité, reprise sur défaillance, sécurité, etc.).
    Mesure et scénario de qualité
    ➡️ Un **critère mesurable** qui permet de **vérifier le besoin** dans une situation concrète (ex. : reconnexion en moins de 1 minute).

Variabilité

- **Souplesse actuelle** :
  ➡️ Ce que le système **peut tolérer aujourd'hui** sans conséquences graves.

- **Évolution future** :
  ➡️ Ce qui risque de **changer ou devenir plus exigeant** dans le futur (ex. : obligations contractuelles, montée en charge, etc.).
  Impact sur les parties prenantes
  ➡️ Qui est concerné (utilisateurs, développeurs, partenaires) et **quelles sont les conséquences** en cas de mauvaise gestion.
  Priorité (Urgente / Critique / Secondaire)
  ➡️ À quel point il est **nécessaire de répondre rapidement** à ce facteur pour le projet.
  Difficulté (Faible / Moyenne / Élevée)
  ➡️ Le **niveau de complexité technique** pour satisfaire ce facteur.
  Risque (Faible / Moyen / Élevé)
  ➡️ Les **conséquences possibles** en cas d'échec (ex. : perte de revenus, insatisfaction client, incident de sécurité…).

  Par exemple, la reprise sur défaillance des services distants dans les grandes surfaces, où le système doit rétablir la connectivité dans la minute suivant la défaillance du service distant.

8. **Facteurs et Artéfacts du Processus Unifié** : Il est important de considérer les besoins spéciaux, les variantes technologiques et les questions ouvertes dans les cas d'utilisation pour identifier les facteurs architecturaux explicites ou implicites.

9. **Exemple de Cas d'Utilisation - Traiter une vente** : Le cas implique un caissier, avec des préconditions (identification et authentification) et des postconditions (vente enregistrée, taxes calculées, inventaire mis à jour).

10. **Scénarios Associés** : Le scénario principal implique l'enregistrement d'une vente, tandis que les scénarios alternatifs gèrent les pannes ou erreurs comme un article invalide ou un paiement non autorisé. Des besoins spéciaux, comme la réponse en moins de 30 secondes dans 90 % des cas, sont également mentionnés.
11. **L'art de résoudre les facteurs architecturaux** : L'architecture logicielle nécessite de faire des compromis pour trouver des solutions adaptées aux besoins, tout en gérant les interdépendances et les priorités. Cela demande une bonne maîtrise des styles, patterns, technologies, produits, tendances, et pièges.

12. **Principes fondamentaux de la conception architecturale** :

- **Faible couplage** : Les composants doivent être indépendants.

- **Forte cohésion** : Les fonctions d'un composant doivent être liées entre elles.

- **Protection contre les variations** : Utilisation de mécanismes (interfaces, indirections) pour éviter des modifications en cascade.

13. **Illustration de la cohésion** : Exemple de mauvaise cohésion : une classe `ClassPersonne` qui regroupe des méthodes sans lien logique. Solution : diviser en plusieurs classes comme `Medecin`, `Mecanicien`, etc.

14. **Illustration du faible couplage** : Une classe très couplée est difficile à maintenir. Il est préférable de répartir les responsabilités dans des classes indépendantes et spécialisées.

15. **Principe de substitution (Liskov 1987)** : Toute instance d'une sous-classe peut être utilisée à la place de sa super-classe sans affecter le fonctionnement. Exemple : une classe `ChefDepartement` peut être utilisée là où une instance de `Enseignant` est attendue, mais pas l'inverse si des méthodes spécifiques à `Enseignant` ne sont pas présentes dans la super-classe.

16. **Polymorphisme** : Permet à plusieurs classes de réagir différemment à un même message. Exemple : un véhicule peut se déplacer de manière différente selon qu'il s'agisse d'un avion, d'un bateau ou d'une voiture.

17. **Interface & Pattern Adaptateur** : Une interface définit les opérations communes. Le pattern adaptateur permet de faire correspondre une interface source à une interface cible. Exemple : adapter une méthode `rembourserEnEuro()` pour fonctionner avec `rembourserEnDinar()`.

18. **Principe de protection contre les variations** : Exemple de mauvais et bon principe d'adaptateur, où l'utilisation d'interfaces permet de travailler avec différents types de véhicules (Voiture, Bus, Avion) sans dépendre d'une classe spécifique.

19. **Séparation des préoccupations et localisation de l'impact** : Il est important de diviser la logique applicative, la persistance et la sécurité en entités distinctes pour simplifier la gestion des changements.

20. **Techniques de séparation des préoccupations** :
    Modularisation en composants séparés : Composants indépendants interagissant via leurs services.
    Utilisation des décorateurs : Ajouter dynamiquement des fonctionnalités à un système, comme ajouter la sécurité via un décorateur.
    Utilisation des postcompilateurs et technologies orientées aspect : Ajouter des fonctionnalités à une classe via des outils post-compilation sans modifier directement la logique applicative.
21. **Modularisation en composants séparés** : Séparer les sous-systèmes comme la sécurité et la persistance, et organiser l'architecture en couches. Les services de persistance offrent une façade pour interagir avec d'autres services.

22. **Utilisation des décorateurs** : Permet d'ajouter des fonctionnalités (comme la sécurité) sans modifier l'objet original. Par exemple, un conteneur dans les EJB peut être un décorateur ajoutant des fonctionnalités de sécurité aux objets.

23. **Utilisation des postcompilateurs et des technologies orientées aspect** : Un postcompilateur ajoute des fonctionnalités (comme la persistance) après la compilation. Cela permet d'ajouter des comportements sans toucher à la logique de base du code source.

24. **Processus Unifié et analyse architecturale** : Les décisions architecturales sont prises progressivement, en itérations, et adaptées en fonction des retours. Cela permet d'ajuster l'architecture selon la compréhension croissante du projet.

25. **Cycle de vie en cascade** : Une approche rigide en étapes prédictives : spécifications, conception, implémentation, intégration, tests.

26. **Description de l'Architecture Logicielle** : Le document présente les idées et décisions prises lors de la conception du système, aidant les développeurs à comprendre les principes clés sans s'éparpiller.

27. **Vue Architecturale** : Une vue qui met en lumière les informations essentielles du système, comme sa structure, sa modularité, ses composants et les principaux flux de contrôle, à travers des textes et des diagrammes UML.

28. **Types de Vues Architecturales selon le Processus Unifié** :

- **Vue Logique** : Décrit l'organisation conceptuelle avec les couches, sous-systèmes, classes et interfaces.

- **Vue Processus** : Décrit les processus, threads, collaborations et allocation des éléments.

- **Vue Déploiement** : Montre le déploiement physique des composants sur les nœuds du réseau.
- **Vue des Cas d'Utilisation :**Décrit les principales fonctionnalités attendues du système via des scénarios d'interaction entre les acteurs et le système.
- **Vue d'Implémentation :**Décrit l'organisation physique du code source, des composants logiciels et des livrables (fichiers, modules, bibliothèques…).
- **Vue Données :**Décrit la structure des données persistantes (modèle objet-relationnel, schéma de base de données, mapping ORM).

29. **Vues Architecturales** : Création de vues architecturales à différents moments du projet, comme après la construction ou certaines itérations, pour aider à l'intégration de nouveaux membres et à l'évolution de l'architecture.

30. **Représentation Architecturale pour un Facteur Spécifique** : Exemple de fiabilité et reprise après une défaillance des services distants, utilisant la protection des variations avec des adaptateurs pour la localisation des services.
    Vue Logique : Diagrammes de packages et de classes représentant la structure à

grande échelle et la fonctionnalité des composants essentiels du système.
Vue des Cas d'Utilisation : Identification des scénarios clés influençant l'architecture, avec des diagrammes d'interaction pour illustrer les processus importants.
Vue de Déploiement : Diagrammes UML représentant l'allocation des composants aux nœuds du réseau et les interactions entre ces composants dans le système déployé.

Conclusion

L'architecture logicielle repose sur une structure solide, avec un modèle de données clair, des processus bien organisés et une protection contre les défaillances. Les vues UML permettent de documenter efficacement l'ensemble de l'architecture, des cas d'utilisation aux décisions techniques prises durant le développement.

Chapitre 2 : Le protocole HTTP

1. Introduction à HTTP

HTTP (Hypertext Transfer Protocol) est un protocole utilisé pour accéder aux fichiers et ressources sur le Web. Il fonctionne sur le protocole TCP pour le transport des données et suit un modèle client-serveur avec un échange requête/réponse : le client (navigateur) envoie une requête HTTP, et le serveur y répond.

2. Fonctionnement de HTTP

Le fonctionnement d'HTTP se déroule en trois étapes :

- Connexion TCP au serveur.
- Envoi de la requête HTTP.
- Réception de la réponse HTTP et déconnexion (sauf si la connexion est persistante).

3. Versions du protocole HTTP

- **HTTP 1.0** : Une connexion par requête.
- **HTTP 1.1** : Connexions persistantes et de nouvelles méthodes.
- **HTTP/2** : Optimisations des performances avec multiplexage et compression des en-têtes.

4. URI, URL, URN

- **URI** : Identifie une ressource sur le Web.
- **URL** : Localise une ressource.
- **URN** : Identifie une ressource de manière permanente.

5. Format d'une URL HTTP

Le format d'URL est : `http://<host>:<port>/<path>?<query>#<fragment>`

- **host** : nom de domaine ou adresse IP.
- **port** : numéro de port (par défaut 80 pour HTTP).

- **path** : chemin d'accès à la ressource.
- **query** : paramètres envoyés.
- **fragment** : ancre dans la page.

6. Champs de l'URL HTTP

Les composants d'une URL incluent :

- **host** : adresse du serveur.
- **port** : numéro de port.
- **path** : chemin vers la ressource.
- **query** : paramètres.
- **fragment** : référence à une section spécifique.

7. Encodage d'URL

Certains caractères spéciaux doivent être encodés en utilisant des codes ASCII hexadécimaux, par exemple, `&` devient `%26` et l'espace devient `%20`.

8. Méthodes HTTP

Les principales méthodes HTTP incluent :

- **GET** : Récupère un document.
- **POST** : Envoie des données.
- **HEAD** : Récupère uniquement les en-têtes.
- **PUT** : Envoie un document pour stockage.
- **DELETE** : Supprime une ressource.

9. Requête et Réponse HTTP

- **Requête HTTP** : Contient la méthode, l'URL, la version HTTP, les en-têtes et éventuellement un corps (pour POST).

- **Réponse HTTP** : Contient la ligne de statut (code et message), les en-têtes, et le corps (contenu retourné).

10. En-têtes HTTP

Les en-têtes HTTP incluent des informations sur le contenu, la connexion, et la gestion des cookies. Ils sont classés en en-têtes génériques, de requête, de réponse, et hop-by-hop.

11. Codes de statut HTTP

Les codes de statut HTTP indiquent le résultat de la requête :

- **1XX** : Informations.
- **2XX** : Succès (ex. 200 OK).
- **3XX** : Redirection (ex. 301 Moved Permanently).
- **4XX** : Erreurs du client (ex. 404 Not Found).
- **5XX** : Erreurs du serveur (ex. 500 Internal Server Error).

12. Connexions persistantes

Avec HTTP 1.1, la connexion TCP peut être réutilisée pour plusieurs requêtes, ce qui améliore l'efficacité, en particulier pour les pages contenant plusieurs ressources (images, scripts, etc.). De plus, HTTP 1.1 introduit le pipelining, permettant d'envoyer plusieurs requêtes sans attendre les réponses.

Chapitre 3 : Présentation de la technologie Java EE

1. Introduction

Java EE (anciennement J2EE) est une extension de Java 2SE, permettant de développer des applications web et d'entreprise déployées sur un serveur d'applications. Il inclut des API, une architecture, des méthodes de packaging/déploiement, et des outils pour la gestion des applications.

2. Java 2SE vs Java EE

- **Java 2SE** : Fournit la JVM (pour l'exécution multiplateforme via bytecode), une bibliothèque standard (collections, I/O, etc.), et des outils de développement (javac, java, javadoc...).

- **Java EE** : Étend Java 2SE pour développer des applications multi-tiers orientées entreprise, avec des fonctionnalités spécifiques pour les applications web et d'entreprise.

3. Pourquoi choisir Java EE ?

- **Portabilité** : Les applications sont portables grâce à la JVM.

- **Gratuité et indépendance** : Java EE est gratuit et indépendant de la plateforme.

- **Sécurité intégrée** : Des mécanismes de sécurité sont fournis (HttpAuthenticationMechanism, SecurityContext...).

- **Librairies robustes** : Java EE offre de nombreuses librairies pour les bases de données, la gestion des mails, les transactions, les fichiers/images, le téléchargement, et la supervision système.

4. Composants clés de Java EE

- **Servlets** : Traitement des requêtes HTTP côté serveur.

- **JSP (JavaServer Pages)** : Génération dynamique de contenu HTML.

- **EJB (Enterprise JavaBeans)** : Gestion de la logique métier, des transactions, et de la sécurité.

Java EE permet également l'intégration d'autres éléments comme des applets, des services web, et des applications Java standards.

Chapitre 4 : Présentation de la technologie des Servlets

1. Qu'est-ce qu'une Servlet ?

Une **Servlet** est un composant Java côté serveur qui gère des requêtes (souvent HTTP) et génère des réponses dynamiques.
Elle est **chargée une seule fois**, ensuite une **seule instance** gère toutes les requêtes via **des threads**.

2. Rôle et Avantages

- Gérer des pages web dynamiques.
- Portables, performants, multithreadés.
- Utilisés principalement avec le protocole **HTTP** via la classe `HttpServlet`.

3. Cycle de vie d'une Servlet

1. `init()` – appelé une fois au chargement.
2. `service()` – gère les requêtes et redirige vers `doGet()` ou `doPost()`.
3. `destroy()` – appelé avant la suppression.
4. Le **Garbage Collector** nettoie la mémoire.

4. Méthodes HTTP

- `GET` : pour récupérer des données via l'URL (limitée et peu sécurisée).
- `POST` : pour envoyer des données plus volumineuses et sensibles.
- `HEAD` : pour obtenir uniquement les métadonnées (sans corps de réponse).

5. Fonctionnement avec le serveur HTTP

- Le serveur crée les objets `HttpServletRequest` et `HttpServletResponse`.
- Appelle `service()` → redirige vers la méthode appropriée (`doGet()`, etc.).

6. Méthodes importantes de HttpServlet

- `init()` : initialisation.

- `service()` : routage des requêtes.
- `destroy()` : libération des ressources.
- `doGet()` / `doPost()` : traitement réel des requêtes.

7. Formulaires HTML & Servlets

- Formulaire envoie des données via `GET` ou `POST` à une Servlet.
- Chaque champ a un nom et une valeur.
- Les données sont récupérées via `getParameter()`.

8. Lecture des paramètres

- `getParameter(String)` : lire une valeur.
- `getParameterValues(String)` : plusieurs valeurs (ex : checkbox).
- `getParameterNames()` : récupérer tous les noms de champs.

9. La gestion des cookies avec les servlets

- Fichiers texte stockés chez le client pour retenir des infos.
- Manipulation facile avec la classe `Cookie`.
- Envoi : `response.addCookie(cookie)`.
- Récupération : `request.getCookies()`.

Exemple :

```
Cookie c = new Cookie("id", "1234");
c.setMaxAge(86400); // 1 jour
response.addCookie(c);
```

Limites : désactivation possible, navigateurs anciens non compatibles.

Méthodes principales de la classe `Cookie` :

`Cookie(String name, String value)` : Crée un cookie avec un nom et une valeur.

`getDomain()` / `setDomain(String)` : Récupère ou définit le domaine concerné par le cookie.

`getMaxAge()` / `setMaxAge(int)` : Récupère ou définit la durée de validité du cookie en secondes.

`getPath()` / `setPath(String)` : Récupère ou définit le chemin sur lequel le cookie est valide.

`getSecure()` / `setSecure(boolean)` : Indique si le cookie est transmis uniquement en HTTPS.

`getValue()` / `setValue(String)` : Récupère ou modifie la valeur du cookie.

`getName()` : Récupère le nom du cookie.

`getVersion()` / `setVersion(int)` : Récupère ou définit la version du cookie.

10. La gestion des sessions avec les servlets

1. **HTTP : protocole non connecté (stateless)**
    - Chaque requête est indépendante, sans mémoire entre elles. Le serveur ne sait pas si plusieurs requêtes viennent du même utilisateur.
    - Pour gérer l'état de l'application et identifier un utilisateur à travers ses requêtes, il est nécessaire de suivre l'identité de l'utilisateur, d'associer des données personnelles et de maintenir l'état de l'application.
2. **Méthodes traditionnelles de suivi de session**
    - **Ajout d'un identifiant dans l'URL** : Exemple : `<a href="http://serveur/servlet/exemple?id=674684641">`.
        - Limites : Taille de l'URL (255 caractères) et risque de fuite d'informations.
    - **Champs de formulaire cachés** : Exemple : `<input type="hidden" name="id" value="674684641">`.
        - Transmis via POST, ce qui est plus sécurisé. Limite : nécessite un formulaire avec bouton "submit".
    - **Utilisation des cookies** : Petits fichiers stockés sur le client avec des paires clé/valeur, envoyés automatiquement avec chaque requête HTTP.

        - Exemple de création d'un cookie : `Cookie c = new Cookie("id", "674684641"); c.setMaxAge(24 * 60 * 60); response.addCookie(c);`.

        - Inconvénients : Certains utilisateurs désactivent les cookies, certains anciens navigateurs ne les supportent pas.

3. **Objet HttpSession**

    - Permet de mémoriser des données utilisateur côté serveur, fonctionne comme une table de hachage.

    - Associe un ID de session aux données utilisateur.
      Exemple pour obtenir la session : `HttpSession session = request.getSession();`.

4. **Processus de gestion de session**

    - Récupérer l'ID de session via l'URL (GET), les en-têtes (POST) ou les cookies.

    - Vérifier si la session existe :

        - Si elle existe, récupérer les données.

        - Si elle n'existe pas, générer un nouvel ID et l'envoyer via un cookie ou dans l'URL.

    - Créer une nouvelle session sur le serveur si nécessaire.

5. **Méthodes principales de l'objet HttpSession**

    - **getSession(boolean create)** : Crée une session si elle n'existe pas.

    - **getAttribute("clé")** : Récupère une donnée de session.

    - **setAttribute("clé", "valeur")** : Stocke une donnée dans la session.

    - **invalidate()** : Termine la session.

Chapitre 5 : les Patrons de Conception

1. Motivations des Patrons de Conception :

Les besoins pour une bonne conception et un bon code sont :

- **Extensibilité** : possibilité d'ajouter facilement des fonctionnalités futures.
- **Flexibilité** : capacité à modifier sans perturber l'ensemble du système.
- **Maintenabilité** : facilité de modification et de correction des erreurs.
- **Réutilisabilité** : possibilité de réutiliser des composants dans différents projets.
- **Qualités internes** : amélioration de la qualité du code (clarté, structure).
- **Meilleure spécification et documentation** : clarification des exigences et de la construction du système.

2. Patron de Conception :

Un patron de conception est une solution générale et réutilisable pour un problème récurrent dans un contexte spécifique. Il décrit :

- Les objets communicants, leurs relations et leurs rôles.
- Peut être indépendant d'une application ou spécifique à un domaine (concurrence, systèmes temps réel, etc.).

3. Qu'est-ce qu'un patron de conception ?

Un patron de conception est une solution générale et réutilisable à un problème récurrent, qui formalise des bonnes pratiques.

4. Comment décrire un patron de conception ?

- **Nom** : Vocabulaire spécifique du patron.
- **Problème** : Description du problème et du contexte où il survient.
- **Solution** : Description des éléments et relations pour résoudre le problème.
- **Conséquences** : Effets de l'implémentation du patron (complexité, impact sur la flexibilité, portabilité, etc.).
- **Structure** : Organisation du patron.

5. Patron de Conception Singleton :

- **Problème** : Assurer une seule instance d'une classe et un moyen unique d'y accéder (par exemple pour un gestionnaire de fenêtres ou une base de données).

- **Solution** :

    - Première solution : utiliser un champ `public static INSTANCE` et un constructeur privé pour empêcher l'instanciation.

    - Deuxième solution : utiliser une méthode `getInstance()` pour fournir l'accès à l'instance.

6. Patron Abstract Factory :

- **But** : Créer des familles d'objets sans connaître leurs classes concrètes. Favorise l'encapsulation de la création d'objets.

- **Exemple** : Création de véhicules à essence ou électriques via un catalogue qui ne dépend pas des classes concrètes (ScooterEssence, ScooterElectricité).

- **Avantages** : Couplage faible, extensibilité, réutilisabilité.

7. Patron Façade :

- **Description** : Simplifie l'utilisation d'un système complexe en fournissant une interface unifiée, cachant les détails des composants internes.

- **Exemple** : Dans un système de vente de véhicules, une façade fournit une méthode unique pour rechercher un véhicule sans que le client connaisse les composants internes.

- **Avantages** : Simplicité, flexibilité, protection des composants internes.

8. Patron Observer :

- **Description** : Établit une dépendance entre un sujet et plusieurs observateurs. Lorsqu'un changement survient dans le sujet, tous les observateurs sont notifiés.

- **Exemple** : Un véhicule (sujet) notifie plusieurs vues (observateurs) lorsque sa description ou son prix change.

- **Avantages** : Découplage entre le sujet et ses observateurs, mises à jour automatiques sans dépendance directe.