

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

1. Introduction

Les **besoins** sont les conditions auxquelles un système ou projet doit satisfaire ([Grady 92]).

Les causes d'échec des projets informatiques sont souvent liées à une mauvaise gestion des besoins :

- 13 % : mauvaise information des utilisateurs
- 12 % : besoins incomplets
- 12 % : instabilité des besoins
- 7 % : faibles compétences techniques
- 6 % : mauvaise affectation du personnel
- 50 % : autres causes

2. Typologie des Besoins (modèle FURPS+)

Dans le **processus unifié RUP**, les besoins sont classés via le modèle **FURPS+** :

Acronyme	Catégorie	Détails
F	Functionality (fonctionnalité)	Fonctions, capacités, sécurité
U	Usability (utilisabilité)	Ergonomie, aide, documentation
R	Reliability (fiabilité)	Fréquence des pannes, reprise, prévisibilité
P	Performance	Temps de réponse, débit, précision, disponibilité, utilisation des ressources
S	Supportability (possibilité de prise en charge)	Maintenance, portabilité, internationalisation, testabilité
+	Autres facteurs	Contraintes légales, technologiques, etc.

3. Définition de l'Architecture Logicielle

Une **architecture logicielle** est l'ensemble des **décisions significatives** sur :

- L'organisation du système logiciel
- La sélection des **éléments structurels** et de leurs interfaces
- Le **comportement** des composants (via leurs collaborations)
- Leur **agencement en sous-systèmes**
- Le **style architectural** choisi ([BRJ99])

4. Analyse Architecturale

Elle consiste à :

- Identifier les **besoins non fonctionnels**
- Les traiter dans le **contexte des besoins fonctionnels**

Exemple : dans un point de vente

- Ajouter un calculateur de taxe impacte :
 - la performance
 - l'adaptabilité
 - l'interopérabilité
 - la scalabilité

5. Types et Vues Architecturales

- **Architecture applicative** : affectation des **fonctions** aux composants
- **Architecture système** : configuration du **matériel et OS**
- Dans RUP, ces architectures sont exprimées en termes de **vues**.

6. Décisions Architecturales & Scénarios

Les **besoins critiques** impactent l'architecture, et les solutions sont élaborées via :

- **Scénarios de qualité :**
<stimulus> <réponse> <mesurable>

Exemple : client à la caisse → système doit répondre rapidement.

7. Description des Facteurs Architecturaux

Un **facteur architectural** est un besoin (souvent **non-fonctionnel**) qui influence la structure du système.

Pour chaque facteur, on analyse :

1. **Nom du facteur :** un objectif technique (ex : "Reprise sur défaillance").
2. **Mesure et scénario de qualité :** un exemple mesurable du besoin (ex : reconnexion en 1 minute).
3. **Variabilité :**
 - **Souplesse actuelle :** que peut-on tolérer aujourd'hui ?
 - **Évolution future :** qu'est-ce qui changera à long terme ?
4. **Impact sur les parties prenantes :** qui est concerné et comment ?
5. **Priorité :** est-ce urgent, critique, ou secondaire ? (Faible / Moyenne / Élevée)
6. Évalue la **difficulté et le risque** pour répondre à ce facteur (Faible / Moyenne / Élevée).

Exemple d'application pour les grandes surfaces

Nom du facteur : Reprise sur défaillance des services distants

Mesure et scénario de qualité :

Quand un service distant est défaillant, le système doit rétablir la connectivité dans la **minute** où il redevient disponible.

Variabilité (souplesse actuelle et évolution future) :

- **Souplesse actuelle :**
Des services simplifiés côté client doivent rester **accessibles** (souhaitable) jusqu'à ce que la reconnexion soit possible.

- **Évolution future :**

Dans les **2 ans**, certains détaillants accepteront de payer pour une **duplication totale en local** des services distants (ex. : calcul des taxes).

Impact sur les parties prenantes :

Fort impact sur la **conception du système**, notamment pour assurer la reprise.
Les détaillants sont **très mécontents** lorsque les services distants sont inaccessibles, car cela **empêche les ventes**.

Priorité : Élevée (E)

Difficulté / Risques : Moyenne (M)

8. Facteurs et artefacts du processus unifié

- Dans les **cas d'utilisation**, il faut considérer :
 - les **besoins spéciaux**
 - les **variantes technologiques**
 - les **questions ouvertes**
- Ces éléments permettent de **regrouper les facteurs architecturaux**, explicites ou implicites.

Définition d'un artefact :

Tout **produit du travail** : code, diagrammes, documents, modèles, schémas de BDD, etc.

9. Exemple de cas d'utilisation : Traiter une vente

- **Acteur principal** : Caissier
- **Précondition** : le caissier est identifié et authentifié
- **Postconditions** :
 - Vente enregistrée
 - Taxes calculées
 - Comptabilité & inventaire mis à jour
 - Commissions enregistrées
 - Reçu généré
 - Paiement autorisé

10. Scénarios associés

- **Scénario principal :**
Le caissier enregistre une vente → le client paie → repart avec les marchandises et le reçu
- **Scénarios alternatifs :**
 - Le système tombe en panne → relance → récupération de l'état précédent
 - Code article invalide → rejet par le système
- **Scénario exceptionnel :**
Le client n'a pas d'argent → la vente est annulée
- **Besoins spéciaux :**
 - Réponse en moins de **30 secondes** dans **90 %** des cas
 - Récupération robuste en cas d'échec
- **Variantes technologiques :**
 - Code saisi via **lecteur de code-barres** ou **clavier**
- **Questions ouvertes :**
 - Quelles sont les variations des **taux de taxes** ?

11. L'art de résoudre les facteurs architecturaux

L'architecture logicielle, c'est un **art de faire des compromis** :

- Trouver des **solutions adaptées** aux besoins
- Gérer **interdépendances et priorités**
- Nécessite une bonne **connaissance des styles, patterns, technologies, produits, pièges et tendances**

12. Principes fondamentaux de la conception architecturale

♦ Faible couplage :

Les composants doivent être **indépendants** les uns des autres.

♦ Forte cohésion :

Chaque composant doit regrouper des **fonctions liées entre elles**.

♦ Protection contre les variations :

Prévoir des mécanismes (interfaces, indirections...) pour **éviter les modifications en cascade**.

13. Illustration de la cohésion

✗ Exemple de mauvaise cohésion :

Une classe **ClassPersonne** qui a des méthodes sans lien

Forte cohésion

ClassPersonne
soignerMalad() réparerVoiture() liverOrdonnanceMalad() monterMoteur() présenterCours() assurerTD() assurerTP() payerChambreEtud() affecterChambreEtud() programmerMatch() assignerArbitre()

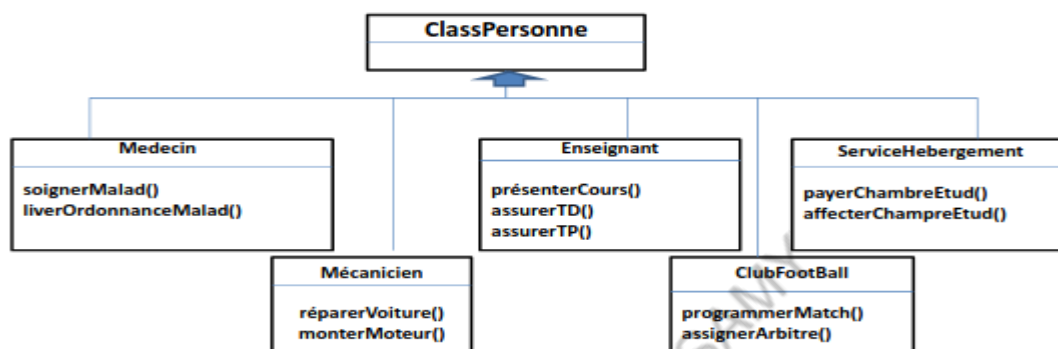
logique (soignerMalade, réparerVoiture, programmerMatch, etc.)

✓ Solution :

Diviser en plusieurs classes :

- Medecin : soignerMalade(), livrerOrdonnance()
- Mecanicien : réparerVoiture(), ...
- Enseignant, ClubFootball, ServiceHebergement, etc.

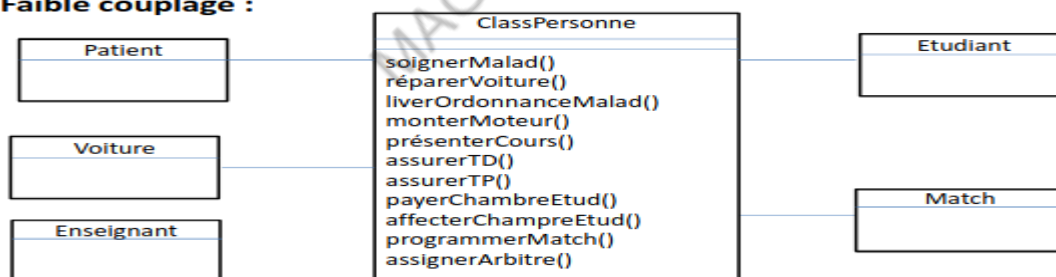
Avoir un forte cohésion :



Maintenant les classes sont cohésives, on favorise la réutilisation

14. Illustration du faible couplage

Faible couplage :



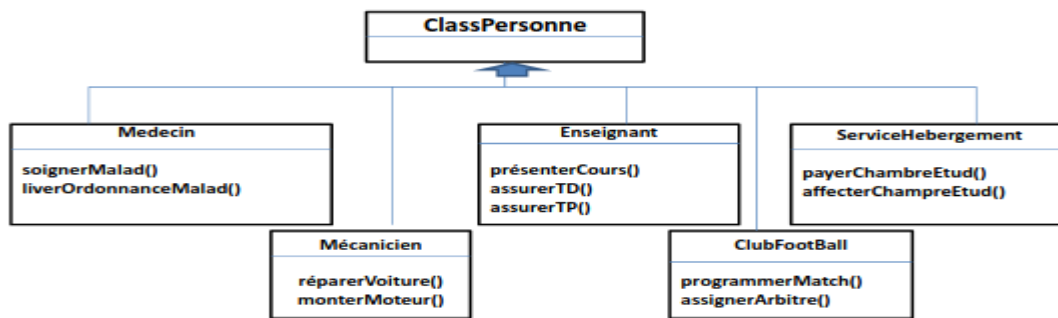
La classe ClassPersonne n'est pas faiblement couplée

Même principe :

Une classe très couplée est difficile à maintenir.

➡ On répartit les responsabilités dans des classes **indépendantes et spécialisées**.

Avoir un faible couplage:



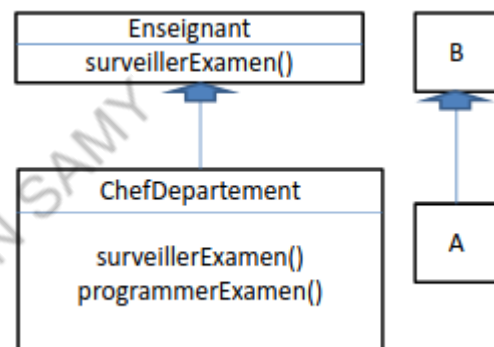
Maintenant les classes sont cohésives, on favorise la réutilisation

15. Principe de substitution (Liskov 1987)

Toute instance d'une **sous-classe** peut être utilisée à la place d'une instance de sa **super-classe** sans affecter le fonctionnement.

Exemple :

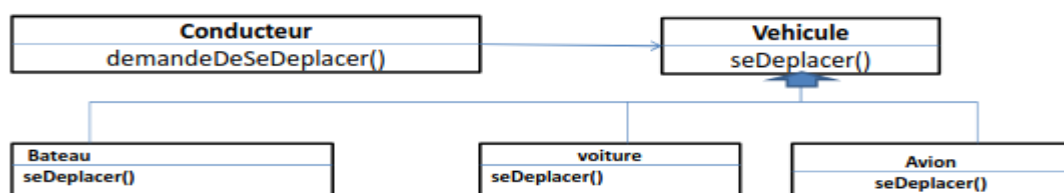
Enseignant e1 = new ChefDepartement(); // OK
e1.surveillerExamen(); // OK
e1.programmerExamen(); // NON, pas accessible via super-classe



16. Polymorphisme

Permet à plusieurs classes de **réagir différemment** à un même message.

Rappel Polymorphisme:



Exemple :

Vehicule v;
v.seDeplacer(); // Peut appeler une version différente selon le type réel : Avion, Bateau, Voiture...

17. Interface & Pattern Adaptateur

- Une **interface** définit les opérations publiques communes
- Le **pattern adaptateur** permet de **faire correspondre** une interface source à une interface cible

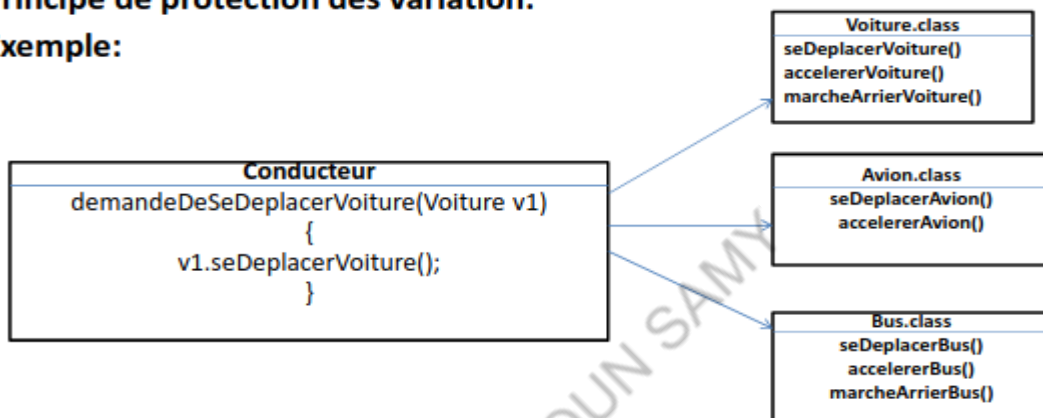
Exemple :

Adapter une méthode `rembourserEnEuro()` pour qu'elle fonctionne dans un contexte `rembourserEnDinar()`.

18. Principe de protection contre les variations

Principe de protection des variation:

Exemple:



❌ Mauvais exemple :

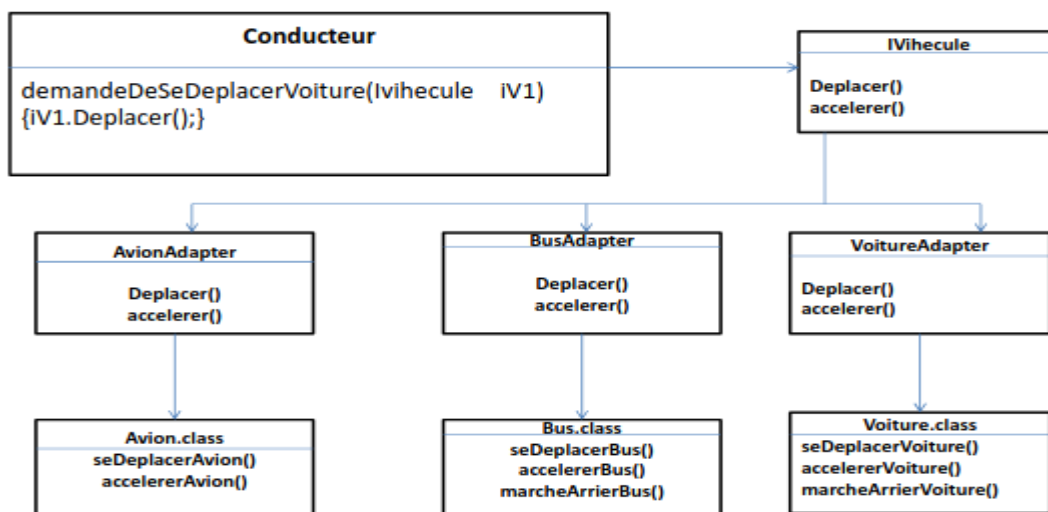
`Conducteur.demandeDeSeDeplacerVoiture(Voiture v1)`

➡ Ne fonctionne qu'avec la classe Voiture.

✅ Bon exemple (via adaptateurs) :

`Conducteur.demandeDeSeDeplacerVehicule(IVehicule iV1)`

➡ Fonctionne avec `VoitureAdapter`, `BusAdapter`, `AvionAdapter`, etc.



Remarque : Les flèches doivent partir des classes *Adaptateur* vers *IVehicule*, car ce sont les adaptateurs qui implémentent l'interface. Donc je pense qu'il y a une erreur dans cette image.

19. Séparation des préoccupations et localisation de l'impact

- **Principe de séparation :** Divise la logique applicative, la persistance et la sécurité en trois entités distinctes.
 - Un objet contenant uniquement la logique applicative.
 - Un sous-système de persistance.
 - Un sous-système de sécurité, sans se soucier de la persistance.

20. Techniques de séparation des préoccupations

Il existe trois techniques pour appliquer la séparation des préoccupations :

1. **Modularisation en composants séparés :** Composants indépendants interagissant via leurs services.
2. **Utilisation des décorateurs :** Ajouter dynamiquement des fonctionnalités à un système. Exemple : ajouter la sécurité avec un objet décorateur.
3. **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

- Exemple : Séparer les sous-systèmes (sécurité, persistance) et les organiser en architecture en couches.
- Les services de persistance offrent une façade pour interagir avec les autres services.

22. Utilisation des décorateurs

- **Approche décorateur :** Permet d'ajouter des fonctionnalités comme la sécurité sans modifier l'objet original.
- Exemple : Un conteneur dans les EJB (Enterprise JavaBeans) est un décorateur qui ajoute la sécurité aux objets.

23. Utilisation des postcompilateurs et des technologies orientées aspect

- Exemple : Dans les EJB, un postcompilateur ajoute la persistance à une classe **Vente** après compilation.
- **Postcompilateur** : Un outil exécuté après la compilation normale pour ajouter des fonctionnalités (ex. persistance) sans toucher à la logique applicative.

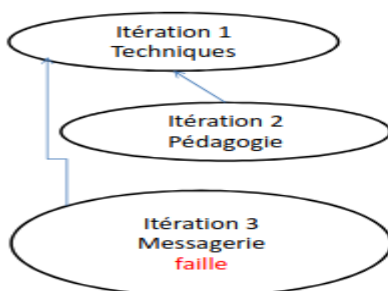
24. Processus Unifié et analyse architecturale

- Dans le **Processus Unifié**, les décisions architecturales sont prises progressivement, avec des itérations et des retours en arrière (feedback), plutôt que de tout décider avant l'implémentation.
- Ce processus permet d'adapter les solutions architecturales en fonction des retours et de la compréhension croissante du projet.

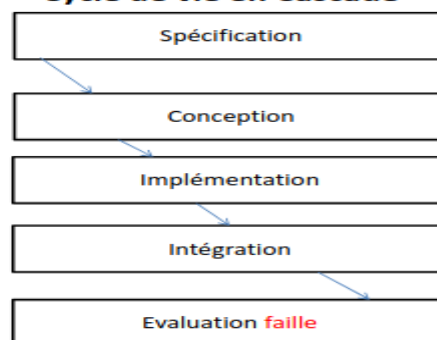
25. Cycle de vie en cascade (approche traditionnelle)

- Cette approche suit des étapes prédictives et rigides :
 1. Clarification des spécifications.
 2. Conception du système basé sur les spécifications.
 3. Implémentation selon la conception.
 4. Intégration des modules.
 5. Évaluation et tests de fonctionnement et de qualité.

Exemple: Processus UP



Cycle de vie en Cascade



26. Description de l'Architecture Logicielle

- **Objectif principal** : La description de l'architecture logicielle dans le cadre du Modèle de Conception du Processus Unifié (RUP) présente les idées principales et les décisions prises lors de la conception du système.
- **Utilité pédagogique** : C'est un document conçu pour aider les développeurs à comprendre les principes essentiels du système, tout en restant centré sur ce public.

27. Vue Architecturale

- **Définition** : Une vue architecturale est un point de vue particulier sur le système qui met en évidence les informations essentielles et ignore les détails non pertinents. Elle se concentre sur la structure, la modularité, les composants essentiels, et les principaux flux de contrôle du système.
- **Rôle** : Elle sert d'outil de communication, de formation ou de réflexion pour l'équipe de développement et se présente sous forme de texte et de diagrammes UML.

28. Types de Vues Architecturales selon le Processus Unifié

Le processus Unifié propose six vues principales de l'architecture (et permet d'ajouter d'autres vues, comme la vue de sécurité) :

1. Vue Logique :

- **Contenu** : Elle décrit l'organisation conceptuelle du logiciel à travers les couches, sous-systèmes, packages, frameworks, classes et interfaces essentielles.
- **Fonctionnalité** : Cette vue résume la fonctionnalité des principaux éléments logiciels, comme les sous-systèmes.
- **Représentation UML** : Diagrammes de packages, de classes et d'interactions, illustrant les scénarios de réalisation de cas d'utilisation et les aspects clés du système.

2. Vue Processus :

- **Contenu** : Elle décrit les processus et threads, la responsabilité, la collaboration et l'allocation des éléments logiques (couches, sous-systèmes, classes, etc.).
- **Fonctionnalité** : Elle permet de modéliser le comportement dynamique du système, en mettant en évidence la gestion de la concurrence (threads, processus), la synchronisation et la communication entre les composants. Elle est essentielle pour les systèmes distribués, temps réel ou multi-tâches.

- **Représentation UML** : Diagrammes de classes et d'interactions, en utilisant la notation UML pour les threads et processus.

3. Vue Déploiement :

- **Contenu** : Elle montre le déploiement physique des processus et composants sur les nœuds de traitement et la configuration physique du réseau entre ces nœuds.
- **Fonctionnalité** : Elle permet de planifier et comprendre comment les éléments logiciels sont distribués physiquement sur le matériel (serveurs, clients, réseaux). Elle est cruciale pour la performance, la scalabilité, la sécurité et la gestion des ressources.
- **Représentation UML** : Diagrammes de déploiement qui montrent la configuration complète du système, y compris les nœuds et leur interaction.

4. Vue Données :

- **Contenu** : Elle décrit la structure des données persistantes du système. Cela inclut la correspondance entre les objets métier et la base de données (mapping objet-relationnel), les entités, relations, contraintes, ainsi que les mécanismes de persistance (procédures stockées, triggers, ORM...).
- **Fonctionnalité** : Elle permet d'assurer la cohérence, la persistance et l'intégrité des données manipulées par le système. Cette vue est cruciale pour la gestion des transactions, de la performance et de la sécurité des données.
- **Représentation UML** : Diagrammes de classes (pour le modèle de données orienté objet), Schémas relationnels (si base de données relationnelle), Mapping ORM si applicable.

5. Vue Cas d'utilisation :

- **Contenu** : Elle décrit les principales fonctionnalités attendues du système à travers des cas d'utilisation significatifs. Ces cas décrivent les interactions entre les acteurs et le système.
- **Fonctionnalité** : Elle sert de point de départ pour la modélisation de l'architecture, en identifiant les scénarios clés qui influenceront les décisions de conception.
- **Représentation UML** : Diagrammes de cas d'utilisation UML descriptions textuelles des scénarios principaux.

6. Vue d'Implémentation :

- **Contenu** : Elle présente l'organisation physique du code source, des composants logiciels, des modules, des fichiers, et des bibliothèques (DLL, JAR, etc.). Elle inclut aussi les livrables binaires et les dépendances.
- **Fonctionnalité** : Cette vue permet aux développeurs de comprendre la structure du projet, les dépendances, et la façon de construire (build) et déployer le système.

- **Représentation UML :**
Diagrammes de **composants** et de **packages** UML, illustrant la relation entre les différents artefacts logiciels.

29. Vues Architecturales

- **Création de vues architecturales :** Ces vues sont générées à différents moments du projet, telles que :
 - **Après la construction du système :** Utilisées pour résumer l'architecture et servir d'aide didactique pour les nouveaux développeurs.
 - **Après certaines itérations :** Pour aider les équipes de développement et intégrer de nouveaux membres.
 - **En cours de conception :** Permettent d'encourager la créativité et l'évolution continue de l'architecture.
- **Vue architecturale des systèmes non fonctionnels :** Exemple d'une application de vente en grand surface pour assurer la **fiabilité** des services distants (calcul de la taxe, gestion de l'inventaire, etc.).
 - **Solution :** Utilisation de la **Protection des variations** pour garantir la robustesse du système face aux défaillances des services et de la base de données.

30. Représentation Architecturale pour un Facteur Spécifique

- **Problème :** Fiabilité et reprise en cas de défaillance des services distants.
- **Solution :** Application de la protection des variations pour la localisation des services via un adaptateur dans une fabrication de services.
- **Facteurs architecturaux :** Dépendances aux spécifications et aux mémos techniques résumant les décisions prises.
- **Vue Logique :** Diagrammes de packages et de classes représentent la structure à grande échelle et la fonctionnalité des principaux composants du système.
- **Vue des processus :** Diagrammes représentant les processus et les threads du système, y compris la communication entre eux (exemple avec RMI en Java).
- **Vue des Cas d'Utilisation**
 - **Résumé des cas d'utilisation significatifs :** Identification des scénarios clés qui influencent l'architecture.
 - **Diagrammes d'interaction :** Illustrations des processus importants de l'architecture à travers des interactions spécifiques.

- **Vue de Déploiement**
 - **Diagrammes de déploiement UML** : Représentent l'allocation des composants aux nœuds du réseau et les processus déployés sur les serveurs.
 - **Commentaire sur le réseau** : Détails sur la structure du réseau et les interactions entre les différents composants du système.

Conclusion

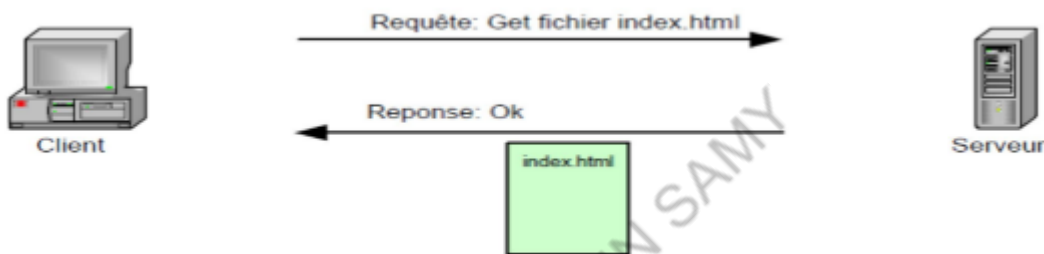
L'architecture logicielle décrite repose sur une structure robuste avec un modèle de données clair, une organisation soignée des processus, et une fiabilité garantie par la protection contre les défaillances. L'utilisation de vues UML permet de documenter efficacement l'ensemble de l'architecture, en incluant les cas d'utilisation, les diagrammes de classes, les processus, ainsi que les décisions architecturales prises tout au long du développement.

MAGUEMOUN SAMY

Chapitre 2:Le protocole HTTP

1. Introduction à HTTP

- **HTTP** (Hypertext Transfer Protocol) est un protocole de communication utilisé pour accéder aux fichiers et ressources sur **Internet**, principalement dans le cadre du **World Wide Web (WWW)**.
- Il repose sur le protocole **TCP** pour le transport des données.
- HTTP suit un **modèle client-serveur** basé sur un échange **requête/réponse** :
 - Le client (navigateur) envoie une **requête HTTP**
 - Le serveur traite la requête et retourne une **réponse HTTP**



2. Fonctionnement de HTTP

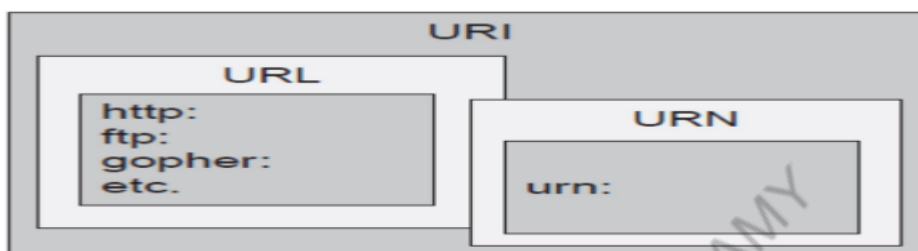
- **Étapes du fonctionnement :**
 1. Connexion TCP au serveur
 2. Envoi de la requête HTTP
 3. Réception de la réponse HTTP
 4. Déconnexion (sauf si la connexion est persistante)

3. Versions du protocole HTTP

- **HTTP 1.0** : une connexion par requête
- **HTTP 1.1** : amélioration majeure, notamment avec :
 - Cinq nouvelles **méthodes**
 - **Connexions persistantes** (réutilisation de la connexion TCP)
- **HTTP/2** : performances optimisées, multiplexage, compression des en-têtes, etc.

4. URI, URL, URN

- **URI (Uniform Resource Identifier)** : identifie de manière unique une ressource sur le Web.
- Trois types :
 - **URL** : localise une ressource (<http://site.com/page>)
 - **URN** : identifie une ressource de manière permanente (ex. ISBN pour les livres)
 - URI pouvant faire à la fois office d'URL et d'URN



5. Format d'une URL HTTP

- Format général :
<http://<host>:<port>/<path>?<query>#<fragment>>
 - [<host>](#) : nom de domaine ou adresse IP
 - [<port>](#) : optionnel (par défaut 80 en HTTP)
 - [<path>](#) : chemin d'accès à la ressource
 - [<query>](#) : paramètres (clé=valeur)
 - [<fragment>](#) : ancre dans la page (ex. [#section1](#))

6. Champs de l'URL HTTP

- **host** : adresse IP ou nom de domaine du serveur (ex. www.example.com)
- **port** : numéro du port utilisé (80 par défaut pour HTTP, donc souvent omis)
- **path** : chemin vers la ressource sur le serveur (ex. [/dossier/fichier.html](http://dossier/fichier.html))
- **query** : paramètres envoyés à un script (ex. ?id=5&cat=sport)
- **fragment** : renvoie à une section précise dans la page (ex. #section2)

7. Encodage d'URL :

Certains caractères spéciaux (comme **&**, **/**, **?**) doivent être **encodés** (ou "échappés") dans une URL.

On les remplace par leur code **ASCII en hexadécimal précédé de %**.

Par exemple :

- **&** devient **%26**
- espace () devient **%20**

Exemples d'URL encodées :

- <http://www.exemple.com/Texte%20Avec%20Espace/Exemple.html>

8. Méthodes HTTP – Résumé :

Méthode	HTTP 1.0	HTTP 1.1	Description courte
GET	Oui	Oui	Récupère un document.
POST	Oui	Oui	Envoie des données (formulaires, etc.).
HEAD	Oui	Oui	Récupère uniquement les en-têtes.
OPTIONS	Non	Oui	Liste les options disponibles pour une ressource.
PUT	Non	Oui	Envoie un document au serveur pour stockage.
DELETE	Non	Oui	Supprime une ressource.
TRACE	Non	Oui	Retourne la requête telle qu'elle a été reçue (debug).
CONNECT	Non	Oui	Établit un tunnel via un proxy (HTTPS).

9. Requête et Réponse HTTP :

♦ Requête HTTP (client → serveur) :

- Contient :
 1. **Ligne de requête** : méthode (GET, POST...), URL, version HTTP.
 2. **En-têtes** : infos supplémentaires (ex. User-Agent, Host...).
 3. **Entité (corps)** : optionnelle, utilisée surtout avec POST, séparée des en-têtes par un double saut de ligne (CRLF x2).

♦ Réponse HTTP (serveur → client) :

- Contient :
 1. **Ligne de statut** : version HTTP + code d'état (ex. 200 OK).
 2. **En-têtes** : infos sur la réponse (ex. Content-Type, Date...).
 3. **Corps** : le contenu retourné (HTML, image, JSON...). Peut être absent (ex. méthode HEAD).

10. En-têtes HTTP :

♦ En-têtes génériques (requête + réponse)

- **Content-Length** : taille des données en octets.
- **Content-Type** : type MIME des données (ex. text/html).
- **Connection** : précise si la connexion TCP reste ouverte (**Keep-Alive**) ou non.

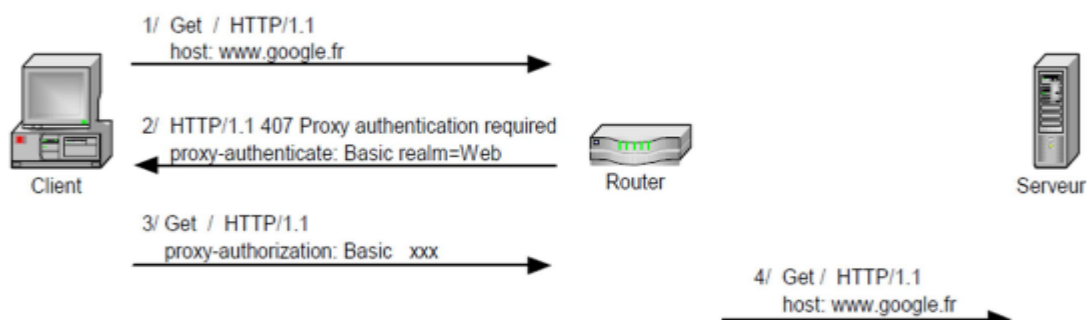
♦ En-têtes de requête (client → serveur)

- **Host** (obligatoire en HTTP/1.1) : nom du serveur demandé.
- **Accept**, **Accept-Encoding**, **Accept-Language** : formats, compressions et langues acceptés.
- **User-Agent** : navigateur utilisé.
- **Referer** : page d'origine.
- **Cookie** : envoie les cookies au serveur.

- **If-Modified-Since, If-None-Match** : demande conditionnelle selon la date/modification.
- ♦ **En-têtes de réponse (serveur → client)**
 - **Allow** : méthodes autorisées (GET, POST...).
 - **Content-Encoding, Content-Language, Content-Type** : compression, langue, type MIME.
 - **Date, Expires, Last-Modified** : dates utiles pour le cache.
 - **Location** : redirection.
 - **Set-Cookie** : création de cookies.
 - **Server, Etag, Pragma** : infos sur le serveur ou gestion de cache.
- ♦ **En-têtes Hop-by-hop (traités à chaque nœud)**
 - **Connection, Proxy-Authenticate, Proxy-Authorization, Transfer-Encoding**.

📌 Exemple d'usage avec un proxy :

1. Le **client** souhaite accéder à une ressource sur Internet, mais doit passer par un **proxy**.
2. Il établit une **connexion TCP** avec le proxy et lui envoie une **requête HTTP**.
3. Le proxy, nécessitant une authentification, répond avec un en-tête **Proxy-Authenticate**.
4. Le client renvoie sa requête, cette fois avec les **données d'identification** dans l'en-tête **Proxy-Authorization**.
5. Une fois le client authentifié, le proxy ouvre une **connexion TCP avec le serveur cible**.
6. Il **transmet la requête** du client au serveur **sans l'en-tête Proxy-Authorization**, car cet en-tête ne concerne que la communication client <-> proxy.



11. codes de statut HTTP :

indiquent le résultat de la requête envoyée par le client au serveur. Ils sont classés en 5 catégories principales :

1. 1XX - Information :

- *100 Continue* : Utilisé quand une requête avec un corps est en cours.
- *101 Switching Protocol* : Réponse pour changer de protocole.

2. 2XX - Succès :

- *200 OK* : Requête traitée avec succès.
- *201 Created* : Un document a été créé via une requête PUT.
- *202 Accepted* : Requête acceptée mais traitement non terminé.
- *204 No Content* : Aucune information à renvoyer.
- *206 Partial Content* : Une partie du document est renvoyée.

3. 3XX - Redirection :

- *301 Moved Permanently* : Le document a changé d'adresse de façon permanente.
- *302 Found* : Le document a changé d'adresse temporairement.
- *304 Not Modified* : Le document demandé n'a pas été modifié depuis la dernière requête.

4. 4XX - Erreurs du client :

- *400 Bad Request* : Requête incorrecte.
- *401 Unauthorized* : Authentification nécessaire pour accéder au document.
- *403 Forbidden* : Accès interdit au document.
- *404 Not Found* : Document introuvable.
- *405 Method Not Allowed* : Méthode de la requête non autorisée.

5. 5XX - Erreurs du serveur :

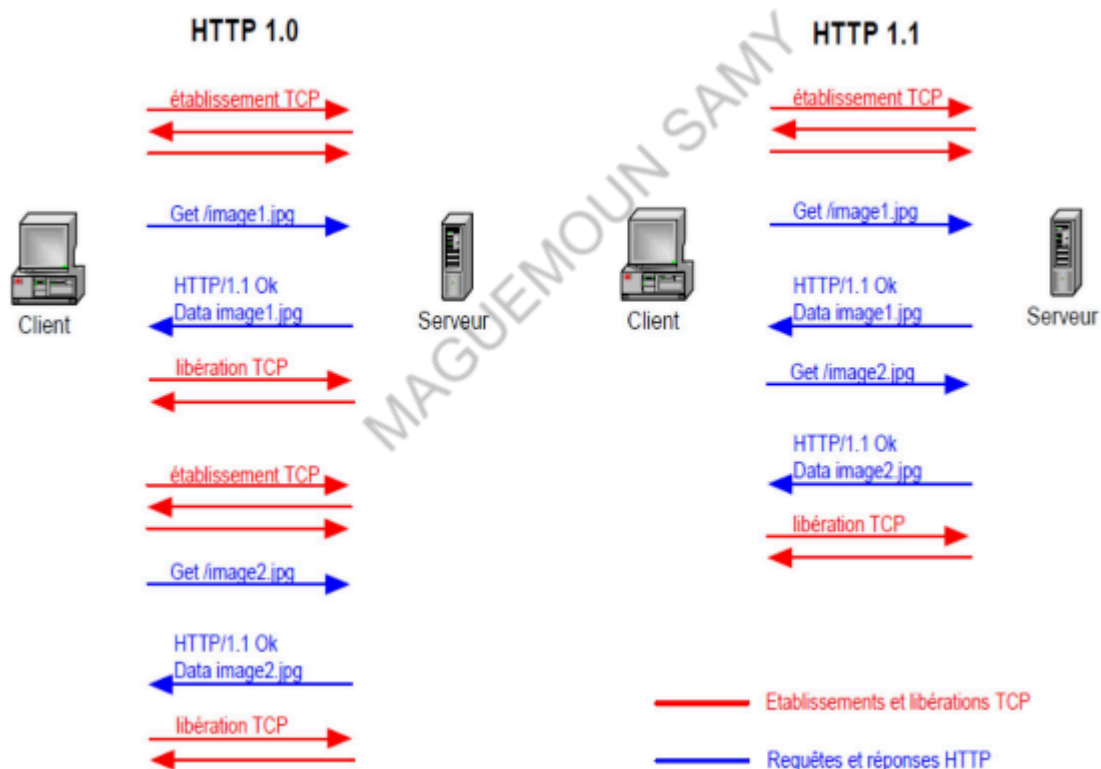
- *500 Internal Server Error* : Erreur serveur interne.
- *501 Not Implemented* : La méthode utilisée n'est pas implémentée.
- *502 Bad Gateway* : Erreur serveur lors d'une requête proxy.

12. connexions persistantes :

Avec **HTTP 1.0**, une nouvelle connexion TCP doit être établie pour chaque URL demandée.

Avec **HTTP 1.1**, il est possible de réutiliser une connexion TCP existante pour demander plusieurs URLs. Cela permet d'améliorer l'efficacité, surtout pour des pages contenant de nombreux éléments (comme des liens vers des fichiers JavaScript ou des images).

De plus, **HTTP 1.1** introduit le **pipelining**, qui permet d'envoyer plusieurs requêtes HTTP sur la même connexion TCP sans attendre les réponses, ce qui réduit le temps d'attente



Chapitre 3: Présentation de la technologie Java EE

1. Introduction

- Java EE (anciennement J2EE) est une **extension** de J2SE.
- Elle permet de développer des **applications web et d'entreprise** déployées sur un **serveur d'applications**.
- Java EE inclut : des **API**, une **architecture**, une **méthode de packaging/déploiement**, et des outils de **gestion des applications**.

2. Java 2SE vs Java EE

- **J2SE** fournit :
 - La **JVM** (exécution multiplateforme via bytecode).
 - Une **bibliothèque standard** (collections, I/O, etc.).
 - Des **outils de développement** (javac, java, javadoc...).
- **Java EE** étend Java 2SE pour le **développement multi-tiers** orienté entreprise.

3. Pourquoi choisir Java EE ?

- **Portabilité, gratuité, indépendance.**
- Sécurité intégrée (HttpAuthenticationMechanism, SecurityContext...).
- Nombreuses **librairies robustes** pour :
 - Bases de données, mails, transactions,
 - Gestion de fichiers/images,
 - Téléchargements,
 - Supervision système.

4. Composants clés de Java EE

1. **Servlets** : traitement des requêtes HTTP côté serveur.
2. **JSP (JavaServer Pages)** : génération dynamique de contenu HTML.

3. **EJB (Enterprise JavaBeans)** : logique métier, transactions, sécurité.

Java EE permet aussi d'intégrer d'autres éléments comme des **applets**, **services web**, et des **applications Java standards**.

MAGUEMOUN SAMY

Chapitre 4 : Présentation de la technologie des servlets

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

Une **Servlet** est un **composant Java côté serveur** qui traite les requêtes des clients (souvent HTTP) et génère des réponses dynamiques.

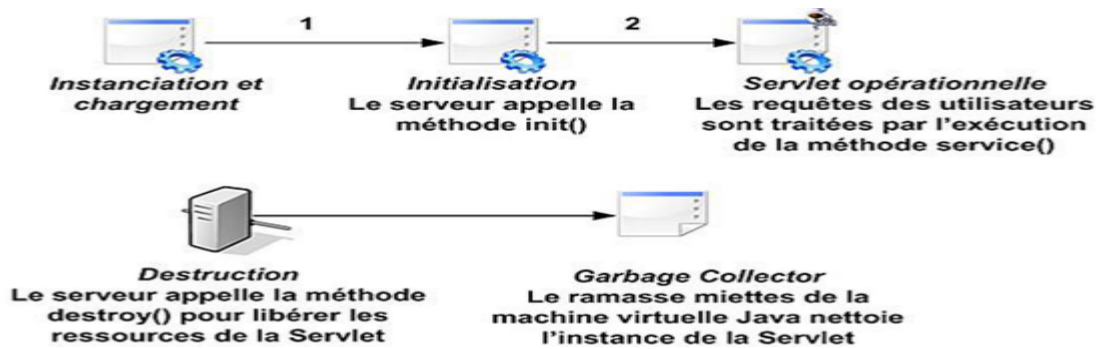
Elle est **chargée une seule fois en mémoire** au premier appel, puis **une seule instance** est utilisée pour gérer toutes les requêtes, via des **threads**.

2. Rôle des Servlets :

- Composants Java côté **serveur** permettant de générer des **pages web dynamiques**.
- Exemple : affichage d'articles selon une plage de prix.
- Avantages :
 - **Portables et évolutives**
 - **Performantes** (chargées une seule fois)
 - **Multithreadées** (une seule instance gérant plusieurs requêtes simultanées)
- ♦ **Utilisation pratique :**
 - Bien qu'utilisables avec plusieurs protocoles (HTTP, FTP...), elles sont **essentiellement utilisées avec des serveurs web HTTP**.
 - L'API fournit la classe **HttpServlet** pour gérer les méthodes HTTP comme **GET**, **POST** et **HEAD**.

3. Cycle de Vie d'une Servlet :

1. **Instanciation & Chargement** : Lors du premier appel, le serveur crée une instance de la servlet.
2. **Initialisation** : La méthode **init()** est appelée une seule fois pour préparer la servlet.
3. **Exécution** : Chaque requête est traitée par la méthode **service()**, qui délègue à **doGet()** ou **doPost()** selon le type de requête.
4. **Destruction** : Quand la servlet n'est plus utilisée, le serveur appelle **destroy()** pour libérer les ressources.
5. **Nettoyage** : Le **Garbage Collector** de la JVM supprime l'instance de la mémoire.



4. Méthodes HTTP

♦ Méthode GET

- Utilisée pour **recupérer une ressource** via une URL (ex. : www.google.com).
- Les paramètres peuvent être transmis via l'**URL** (query string) ou **cookies**.
- **Limites** : taille de l'URL restreinte (~255 caractères) et **peu sécurisé** (visible dans l'URL).
- En réponse, le serveur inclut des **en-têtes** (longueur, date...).

♦ Méthode POST

- Permet d'envoyer des données **plus volumineuses et sensibles**.
- Pas de limite de taille pour le corps de la requête.
- Adaptée pour les **opérations sensibles ou non répétables** (ex. : soumission de formulaires).

♦ Méthode HEAD

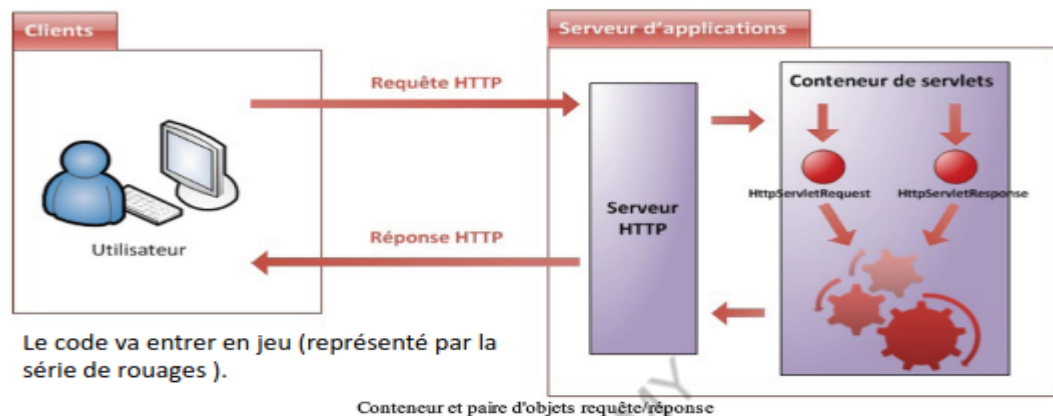
- Fonctionne comme GET, mais **sans renvoyer le corps** de la réponse.
- Sert à obtenir uniquement les **métadonnées** de la ressource (ex. : pour vérifier si une page a changé).

5. Fonctionnement du serveur HTTP avec une Servlet

Que fait-il le serveur HTTP lorsqu'une requête lui parvient?

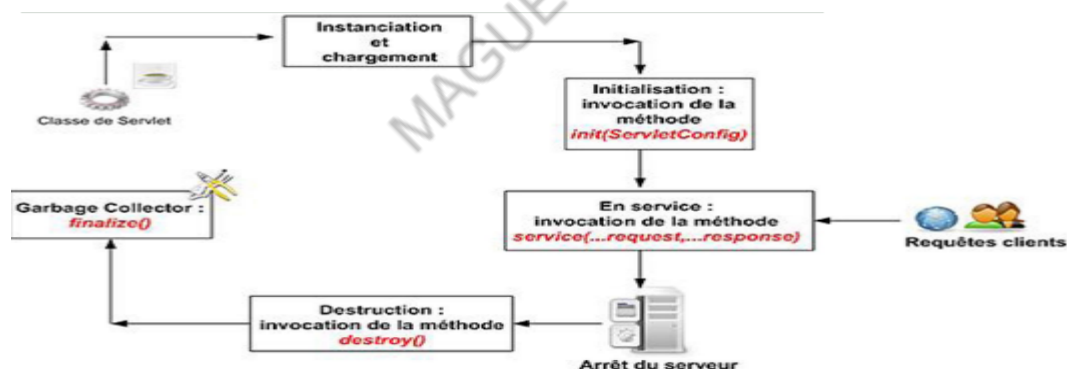
1. Le **serveur HTTP** reçoit une requête.
2. Il la transmet au **conteneur de servlets**.
3. Deux objets sont créés :

- **HttpServletRequest** : contient la requête (URL, en-têtes, paramètres...).
 - **HttpServletResponse** : construit la réponse (en-têtes, contenu...).
4. Le conteneur appelle la méthode **service()** de la servlet, qui redirige vers la bonne méthode :
- **doGet()**, **doPost()**, **doHead()**, etc., selon le type de requête.



Remarque : Une servlet est une classe Java qui hérite de **HttpServlet** pour traiter les requêtes HTTP et renvoyer des réponses personnalisées côté serveur.

6. Fonctionnement d'une servlet (la classe HttpServlet)



Les méthodes clés :

⚙️ **init()**

- Exécutée **une seule fois** lors du chargement de la servlet en mémoire.
- Sert à **initialiser des ressources** : connexion à une base de données, ouverture de fichiers, etc.
- Ne gère **aucune requête**.

service()

- Gère les requêtes du client.
- **Ne pas redéfinir** cette méthode : elle existe déjà dans **HttpServlet**.
- Elle redirige automatiquement la requête vers la bonne méthode **doGet()**, **doPost()**, etc.
- Le développeur doit uniquement redéfinir les méthodes **doXXX()** nécessaires.

destroy()

- Exécutée **une seule fois** avant la suppression de la servlet (ex. arrêt du serveur).
- Permet de **libérer les ressources**.
- Ne gère **aucune requête**.

Invocation d'une Servlet

- Se fait via un **navigateur Web** et le **protocole HTTP**.
- La méthode **doGet()** est appelée quand :
 - L'URL de la servlet est saisie dans la barre d'adresse.
 - Un **lien hypertexte** pointe vers l'URL de la servlet.
- Chaque méthode (**doGet()**, **doPost()**) reçoit deux objets :
 - **HttpServletRequest** : contient la requête du client (paramètres, en-têtes...).
 - **HttpServletResponse** : permet de construire la réponse à renvoyer au client.

7. Formulaire et les Servlets :

Pour permettre à une servlet de récupérer les données d'un formulaire, il faut :

- Définir l'attribut **action** de la balise **<form>** avec l'URL de la servlet qui recevra les informations.
- Spécifier la méthode HTTP (**GET** ou **POST**) via l'attribut **method** de la balise **<form>**.
 - Par défaut, la méthode est **GET**, mais elle peut être changée en **POST** selon le besoin.

Exemple de formulaire HTML :

```
<html>

<body>

  <form action="HelloForm" method="GET">

    First Name: <input type="text" name="first_name"> <br />

    Last Name: <input type="text" name="last_name" />

    <input type="submit" value="Submit" />

  </form>

</body>

</html>
```

Codage des Informations

- Chaque élément du formulaire doit avoir un **nom unique**.
- La paire **nom/valeur** est codée sous la forme suivante : **Nom_de_l_element=valeur**.
- L'ensemble des paires nom/valeur est séparé par des **&** (et commerciaux).

Exemple :

champ1=valeur1&champ2=valeur2&champ3=valeur3

Méthodes d'Envoi de Formulaire : GET vs POST

Méthode GET :

- Permet d'envoyer les données du formulaire via l'URL.
- L'URL du script se voit **complétée** par des paires nom/valeur, séparées par un **?**.

Exemple :

http://nom_du_serveur/test/script.cgi?champ1=valeur1&champ2=valeur2

- La **limite** de la longueur de l'URL est de **255 caractères**.
- **Problème de sécurité** : Les informations sensibles (comme les mots de passe) peuvent être exposées dans l'URL.

Méthode POST :

- **Alternative plus sûre** à GET.
- Envoie les données après les en-têtes HTTP, dans un champ appelé **corps de la requête**.
- **Pas de limite de taille** pour l'envoi des données.

8. Lire les Paramètres avec une Servlet

L'objet `HttpServletRequest` fournit plusieurs méthodes pour accéder aux données envoyées par les formulaires, y compris la méthode la plus courante : `getParameter()`.

Méthode `getParameter()` :

- Permet de **récupérer** la valeur d'un champ de formulaire en passant son nom en argument.

Syntaxe :

```
public String getParameter(String key);
```

- Si le champ est **vide**, une chaîne vide est renvoyée.
- Si le champ **n'existe pas**, la valeur **null** est renvoyée.

Exemple :

```
<input type="text" name="NomDuChamp">
```

Traitement en Java :

```
String Champ = req.getParameter("NomDuChamp");
```

Méthode `getParameterValues()` :

- Utilisée lorsque un champ peut avoir **plusieurs valeurs** (par exemple, dans des listes à choix multiples ou des cases à cocher).

Syntaxe :

```
public String[] getParameterValues(String key);
```

Méthode `getParameterNames()` :

- Permet d'obtenir **l'ensemble des noms** des champs du formulaire.
- Retourne un objet `Enumeration` contenant la liste des noms de champs, qui peut être traitée pour récupérer les valeurs des champs avec `getParameter()`.

Syntaxe :

Enumeration `getParameterNames()`;

9. La gestion des cookies avec les servlets

- **Les cookies** sont de petits fichiers texte stockés sur le **disque dur du client**. Ils permettent de **mémoriser temporairement des informations** (comme les préférences utilisateur) et de **les réutiliser plus tard**.
- Les **en-têtes HTTP** permettent aux **requêtes et réponses** d'échanger ces données.
- Le **cookie est envoyé via l'en-tête HTTP "Set-Cookie"**, sous la forme :
`Set-Cookie: NOM=VALEUR; domain=...; expires=...`

L'API Servlet Java et l'objet Cookie

- Java fournit la classe `javax.servlet.http.Cookie` pour **gérer facilement les cookies** dans les servlets.
- Cette classe permet de **créer, modifier, envoyer et lire** les cookies **sans manipuler directement les en-têtes HTTP**.


L'objet Cookie en Java (API Servlet)

- La classe `javax.servlet.http.Cookie` permet de **créer et manipuler des cookies** facilement.
- On peut modifier la valeur d'un cookie via la méthode `setValue()`.

♦ Restrictions :

- Le **nom** du cookie ne peut **pas contenir de caractères spéciaux**.
- La **valeur** peut inclure tous les caractères **sauf** les espaces et : `[] () = , " / ? : ;`

Envoi d'un Cookie au Client

- Pour envoyer un cookie au navigateur, on utilise la méthode :
`response.addCookie(Cookie cookie)`
-  **Important** : le cookie doit être créé **avant** tout envoi de données (texte ou HTML) vers le client, car il est inclus dans les **en-têtes HTTP**.

Exemple :

```
Cookie monCookie = new Cookie("nom", "valeur");
```

```
response.addCookie(monCookie);
```

Récupération des Cookies envoyés par le Client

- Pour récupérer les cookies d'une requête, on utilise :
`Cookie[] cookies = request.getCookies();`
- On peut **parcourir le tableau** pour chercher un cookie précis avec `getName()`.
- Pour récupérer sa valeur, on utilise :
`String valeur = cookie.getValue();`



Méthodes principales de la classe `Cookie` :

Méthode	Description
<code>Cookie(String name, String value)</code>	Crée un cookie avec un nom et une valeur.
<code>getDomain() / setDomain(String)</code>	Récupère ou définit le domaine concerné par le cookie.
<code>getMaxAge() / setMaxAge(int)</code>	Récupère ou définit la durée de validité du cookie en secondes.
<code>getPath() / setPath(String)</code>	Récupère ou définit le chemin sur lequel le cookie est valide.

<code>getSecure()</code> / <code>setSecure(boolean)</code>	Indique si le cookie est transmis uniquement en HTTPS (ligne sécurisée).
<code>getValue()</code> / <code>setValue(String)</code>	Récupère ou modifie la valeur du cookie.
<code>getName()</code>	Récupère le nom du cookie.
<code>getVersion()</code> / <code>setVersion(int)</code>	Récupère ou définit la version du cookie.

10. La gestion des sessions avec les servlets

HTTP : un protocole non connecté (stateless)

Le protocole HTTP ne conserve **aucune mémoire** entre les requêtes.

Chaque requête est **indépendante** : le serveur ne sait pas si plusieurs requêtes viennent du même utilisateur.

Pour maintenir la cohérence dans une application web, il faut :

- **Identifier les requêtes** d'un même utilisateur
- **Associer un profil** ou des données personnelles à cet utilisateur
- **Suivre l'état** de l'application (ex. : nombre de produits vendus)

Méthodes traditionnelles de suivi de session

Plusieurs techniques permettent de **maintenir l'identité de l'utilisateur** entre les requêtes HTTP :

- Ajout d'un **identifiant dans l'URL**
- Utilisation d'un **champ de formulaire caché**
- Utilisation des **cookies**
- Demander une **authentification à chaque requête**

Réécriture d'URL

- Exemple : ``

- **Limites :**

- Taille maximale de l'URL (255 caractères)
- Risque de **fuite d'informations** (visible et partageable)

Champs de formulaire cachés

- Utilisation : `<input type="hidden" name="id" value="674684641">`
- Transmis via **POST** (plus sécurisé)
- **Limite** : nécessite toujours un **formulaire avec bouton submit**

Utilisation des cookies

- Petits fichiers texte stockés sur le **client**
- Contiennent des **paires clé/valeur**
- Envoyés **automatiquement** avec chaque requête HTTP vers le même domaine
- Gérés via la classe **Cookie** du JSDK

Exemple :

```
Cookie c = new Cookie("id", "674684641"); //creation du cookie
```

```
c.setMaxAge(24 * 60 * 60); // définition de la limite de validité : 1 jour
```

```
response.addCookie(c); //envoi du cookie dans la réponse HTTP
```

Inconvénients :

- Certains utilisateurs **désactivent les cookies**
- Anciens navigateurs peuvent ne pas les **supporter**

Objet HttpSession

- Permet de **mémoriser des données utilisateur côté serveur**
- Fonctionne comme une **table de hachage**
- Associe un **id de session** aux données utilisateur

Obtention de la session :

```
HttpSession session = request.getSession();
```

Processus de gestion de session

1. Récupérer l'ID de session :

- Dans l'URL (GET)
- Dans les en-têtes (POST)
- Dans les cookies

2. Vérifier si la session existe :

- Si oui : récupérer les données
- Si non :
 - Générer un nouvel ID
 - L'envoyer via un cookie ou dans l'URL
 - Créer une nouvelle session sur le serveur

Méthodes principales de l'objet HttpSession

Action	Méthode	Description
Obtenir une session	<code>getSession(boolean create)</code>	Crée une session si elle n'existe pas
Lire une valeur	<code>getAttribute("clé")</code>	Récupère une info
Écrire une valeur	<code>setAttribute("clé", "valeur")</code>	Stocke une info
Supprimer la session	<code>invalidate()</code>	Termine la session

Chapitre 5 : les Patrons de Conception

1. Motivations :

Les **besoins pour une bonne conception et un bon code** incluent :

- **Extensibilité** : possibilité d'ajouter des fonctionnalités futures.
- **Flexibilité** : capacité à changer facilement 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, y compris la clarté et la structure.
- **Meilleure spécification, construction, documentation** : clarifier les exigences et 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 particulier. Il décrit :

- **Les objets communicants** et leurs **relations et rôles** dans la résolution du problème.
- Il peut être indépendant d'une application spécifique ou destiné à un domaine particulier (comme la concurrence, la programmation distribuée, ou les systèmes temps réel).
- Le terme "**Design Pattern**" est couramment utilisé pour désigner un patron de conception.

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

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

4. Comment décrire un patron de conception ?

- **Nom** : Vocabulaire spécifique pour désigner le patron.
- **Problème** : Description du problème à résoudre et du contexte dans lequel il survient.

- **Solution** : Description des éléments et de leurs relations pour résoudre le problème.
 - Cela inclut une description générique et une illustration par exemple concret.
- **Conséquences** : Effets de la mise en œuvre du patron, incluant la **complexité en temps/mémoire**, ainsi que son impact sur la **flexibilité**, la **portabilité**, etc.
- **Structure** : La composition et l'organisation du patron.

Patrons de Conception GoF (Gang of Four)

[E. Gamma, R. Helm, R. Johnson, J. Vlissides]

		Catégorie		
Portée	Classe	Création	Structure	Comportement
		Factory Method	Adapter	Interpreter
Objet				Template Method
		Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Les **23 patrons** sont classés en **3 catégories** :

1. **Création** (instanciation d'objets) :
 - Exemples : *Singleton* (une seule instance), *Factory Method* (délègue l'instanciation).
2. **Structure** (organisation des classes/objets) :
 - Exemples : *Adapter* (compatibilité entre interfaces), *Decorator* (ajout de fonctionnalités).
3. **Comportement** (interactions entre objets) :
 - Exemples : *Observer* (notification d'événements), *Strategy* (algorithmes interchangeables).

Portée :

- **Classe** : Utilise l'héritage (ex: *Template Method*).
- **Objet** : Utilise la composition (ex: *Composite*).

Objectif : Réutiliser des solutions éprouvées aux problèmes courants de conception logicielle.

5. patron de conception Singleton

Problème :

L'objectif est de s'assurer qu'une classe possède une **seule instance** et qu'il existe un moyen unique d'y accéder. Par exemple, cela pourrait être utilisé pour un **window manager** ou un **point d'accès à une base de données**.

Solution :

1. Première solution :

- Déclare un champ **public static INSTANCE**.
- Utilise un **constructeur privé** pour empêcher l'instanciation de l'objet à l'extérieur de la classe.
- **Pas d'héritage** possible.

2. Exemple :

```
class A {  
    final public static A INSTANCE = new A();  
    private A() { ... }  
}
```

Deuxième solution (avec méthode getInstance) :

- Déclare un champ **private static INSTANCE**.
- Utilise un **constructeur privé/protected** pour empêcher l'instanciation externe (l'héritage peut être autorisé si le constructeur est **protected**).
- Crée une méthode **public getInstance()** pour fournir l'accès à l'instance.

Exemple :

```
class A {  
    final private static A INSTANCE = new A();  
    private A() { ... }  
    static public A getInstance() { return INSTANCE; }  
}
```

6. Pattern Abstract Factory

But :

Le **patron de conception Abstract Factory** permet de créer **des familles d'objets liés** sans connaître leurs **classes concrètes**. Il favorise l'**encapsulation de la création** d'objets.

Exemple illustratif :

Dans un **système de vente de véhicules**, il existe :

- Des **véhicules à essence**
- Des **véhicules électriques**

La création de ces véhicules est confiée à un **objet catalogue**.

Structure des classes :

Pour chaque type de produit (par exemple, **Scooter**) :

- Une **classe abstraite** (ex: **Scooter**)
- Deux **sous-classes concrètes** :
 - **ScooterEssence** (version essence)
 - **ScooterElectricité** (version électrique)

✓ Conclusion :

Abstract Factory permet de regrouper les objets par **familles cohérentes**, comme essence et électricité, et de les créer **sans couplage direct** avec les classes spécifiques.

Solution Standards:

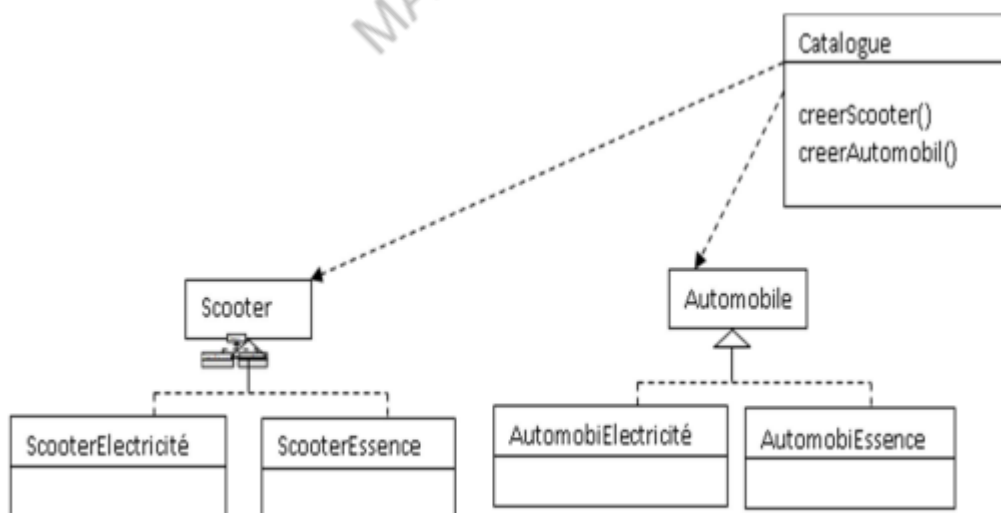


Figure : Solution Orientée Objet standard

Problème de l'Approche Classique

- La classe **Catalogue** doit connaître **tous les types concrets** (ex: **ScooterEssence**, **AutomobileElectrique**).
- **Modification obligatoire** si on ajoute un nouveau type (ex: **Hybride**).
- **Inflexible** et peu réutilisable.

Solution : Abstract Factory

1. **Interface FabriqueVehicule :**
 - Définit des méthodes génériques (**créerScooter()**, **créerAutomobile()**).
2. **Fabriques par famille :**
 - Ex: **FabriqueElectrique** crée des véhicules électriques, **FabriqueEssence** crée des véhicules essence.
3. **Catalogue devient indépendant :**
 - Il utilise l'**interface** et ne connaît pas les implémentations.
 - Reçoit une fabrique concrète (électrique/essence) **en paramètre**.

Bénéfices

- **✓ Couplage faible :** Plus de dépendance aux classes concrètes.
- **✓ Extensible :** Ajout de nouvelles familles (ex: **Hybride**) sans toucher au code existant.
- **✓ Réutilisable :** **Catalogue** fonctionne avec n'importe quelle fabrique.

→ L'Abstract Factory sépare la création d'objets de leur utilisation, pour un code plus flexible

Figure:

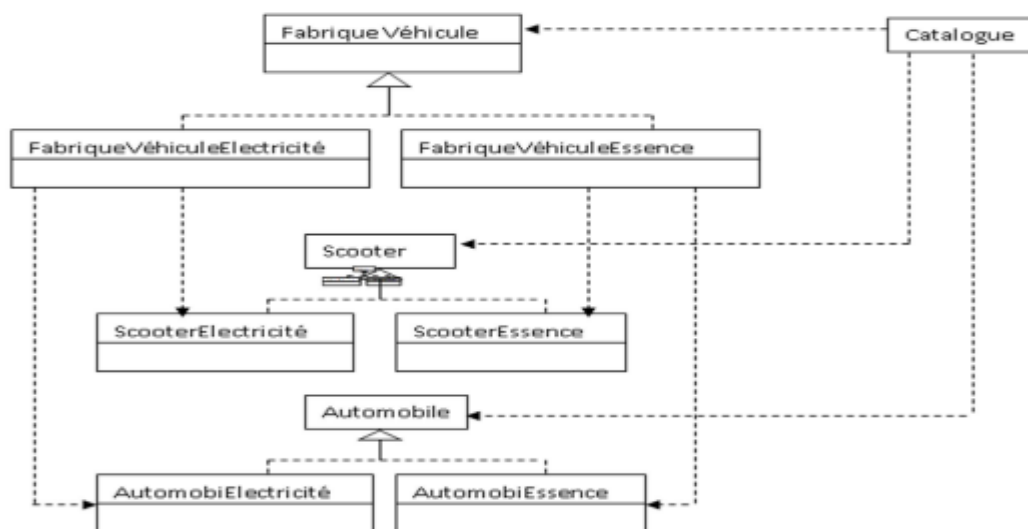


Figure : patron de conception Abstract Factory (appliquer à la famille des véhicules)

Patron Abstract Factory appliqué aux véhicules

1. Structure clé :

- **FabriqueVéhicule** (interface abstraite) : Définit les méthodes `créerScooter()` et `créerAutomobile()`.
- **Fabriques concrètes** :
 - **FabriqueVéhiculeElectricité** → Crée **ScooterElectricité** et **AutomobileElectricité**.
 - **FabriqueVéhiculeEssence** → Crée **ScooterEssence** et **AutomobileEssence**.

2. Fonctionnement :

- Le **Catalogue** utilise l'interface **FabriqueVéhicule** et ignore les implémentations concrètes.
- Il reçoit une fabrique (électrique ou essence) **en paramètre** pour créer les véhicules.

3. Avantages :

- ☒ **Couplage faible** : Le **Catalogue** ne dépend plus des classes concrètes.
- ☒ **Extensible** : Ajout facile d'une nouvelle famille (ex: **Hybride**) sans modifier le code existant.

→ **Abstract Factory** isole la création d'objets pour un code plus flexible et maintenable.



Structure:



Structure Générale du Pattern Abstract Factory

1. Composants Clés :

- **FabriqueAbstraite** : Interface définissant des méthodes pour créer des objets abstraits (A, B).



- **FabriqueConcrète1/2** : Implémentations créant des familles d'objets cohérents (ex: A1+B1 ou A2+B2).
 - **Client** : Utilise la **FabriqueAbstraite** sans connaître les classes concrètes.
2. **Fonctionnement** :
- Le **Client** appelle **FabriqueAbstraite** pour créer A ou B, et reçoit automatiquement la bonne version (A1/A2, B1/B2).
 - Exemple : Si **FabriqueConcrète1** est injectée, le **Client** obtient A1 et B1.
3. **Avantages** :
-  **Isolation** : Le **Client** ne dépend pas des implémentations.
 -  **Cohérence** : Garantit que tous les objets créés appartiennent à la même famille (ex: tous "Style Moderne" ou tous "Style Classique").

→ **Abstract Factory organise la création d'objets liés en familles, sans couplage.**






Exemple réel :

- Thème GUI : **FabriqueModerne** crée **BoutonModerne** + **MenuModerne**, tandis que **FabriqueClassique** crée **BoutonClassique** + **MenuClassique**.

Domaine d'utilisation du pattern Abstract Factory :

-  Lorsqu'un système **doit être indépendant de la manière** dont les objets sont créés ou regroupés.
-  Quand un système doit pouvoir être **paramétré par plusieurs familles de produits** (et ces familles peuvent évoluer avec le temps).

Conséquences de l'utilisation du pattern :

-  **Séparation claire** entre les classes concrètes et les classes clientes :
 -  Les **noms des classes concrètes** n'apparaissent pas dans le code client.
 -  **Échange facile** de familles de produits.
 -  **Cohérence renforcée** entre les produits d'une même famille.
-  Le **processus de création** des objets est **centralisé** dans une classe (la fabrique).

Exemple d'application :

- **java.awt.Toolkit** dans Java utilise ce pattern pour fournir des objets liés à l'interface graphique, indépendamment du système sous-jacent.

7. Pattern Façade

Description :

- Le pattern Façade vise à simplifier l'utilisation d'un ensemble de classes en fournissant une interface unifiée.
- Il encapsule des interfaces complexes (de bas niveau) dans une seule interface de plus haut niveau.
- Il peut nécessiter la définition de méthodes qui combinent les appels aux composants internes.

Exemple : Gestion des ventes de véhicules

Le système est composé de plusieurs composants indépendants :

- Catalogue
- GestionDocument
- RepriseVéhicule

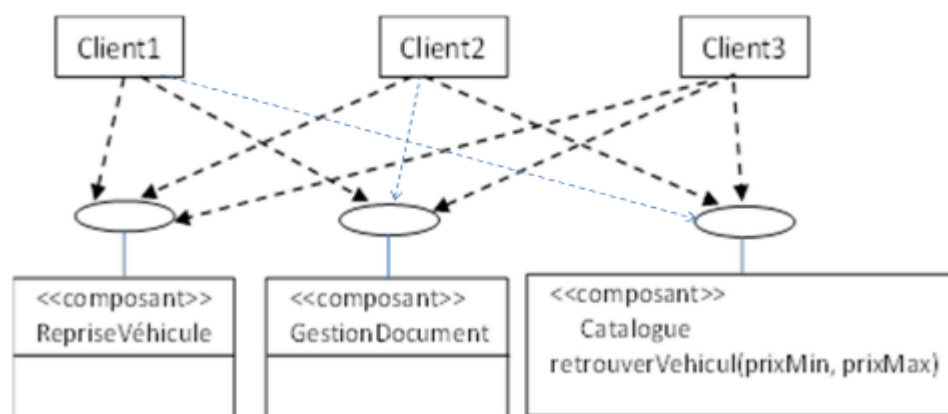
Problèmes d'une solution standard (sans Façade) :

1. Certaines fonctionnalités exposées par les composants ne sont pas utiles aux clients du service web (ex. affichage du catalogue).
2. L'architecture interne est pensée pour la modularité et l'évolution, ce qui engendre une complexité inutile pour les clients externes.

Solution avec Façade :

- Fournir une interface unique et simplifiée, masquant les détails des différents composants.
- Permettre aux clients d'interagir avec le système sans connaître ni gérer sa complexité interne.

Solution Orientée Objet Standards:



Explication de l'image :

Problème :

3 clients → 3 composants complexes (chaque client doit tout comprendre).

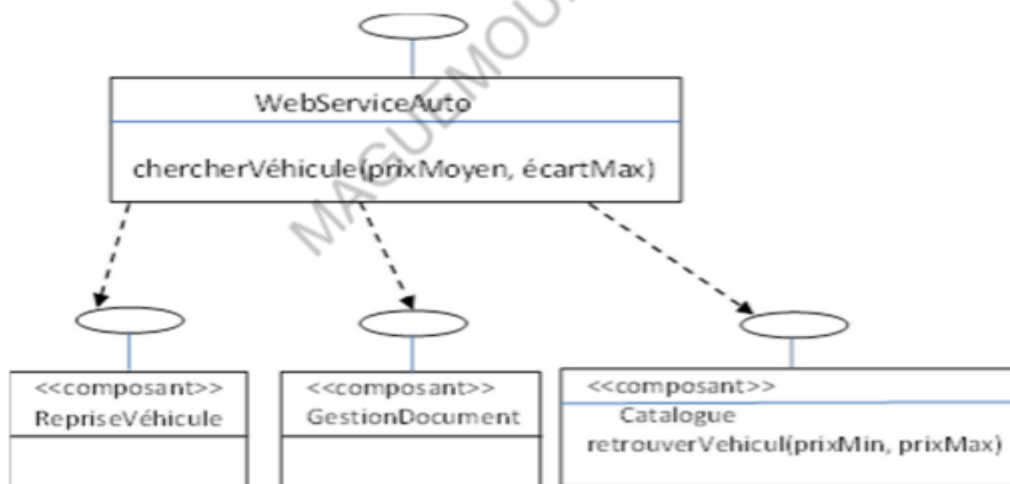
Solution Façade :

- **1 seule interface** qui cache les 3 composants.
- Exemple : `facade.acheter(prixMin, prixMax)` remplace :
java
- `catalogue.chercher();`
- `documents.générer();`
- `reprise.traiter();`

Bénéfices :

- 🧠 **Simplicité** : Le client appelle 1 méthode au lieu de 3.
- 🛡️ **Protection** : Les composants internes peuvent changer sans casser le client.

Le pattern Façade simplifie l'accès à un système complexe en introduisant une interface unifiée de haut niveau. Par exemple, la classe `WebServiceAuto` sert de façade en regroupant plusieurs appels internes (comme `retrouverVehicul`) derrière une méthode plus simple (`chercherVehicule`).



Explication de l'image :

Problème :

Un service web (`WebServiceAuto`) doit exposer une méthode simple `chercherVehicule(prixMoyen, ecartMax)`, mais s'appuie sur **3 composants complexes** :

1. **Catalogue** → `retrouverVehicule(prixMin, prixMax)`
2. **GestionDocument** → Gère contrats/factures
3. **RepriseVehicule** → Traite les échanges

Sans Façade :

- Le client devait manipuler ces 3 composants **manuellement** → complexité inutile.

Avec Façade :

- La méthode `chercherVehicule()` **encapsule tout** :
 - Convertit `prixMoyen ± écartMax` en `prixMin/prixMax` pour le `Catalogue`.
 - Appelle `GestionDocument` et `RepriseVehicule` si besoin.
- **Le client ne voit qu'une seule méthode simple !**

Avantages :

- 🔍 **Simplicité** : Une interface claire pour le client.
- 🛠️ **Flexibilité** : Modifiez les composants internes sans impacter le client.

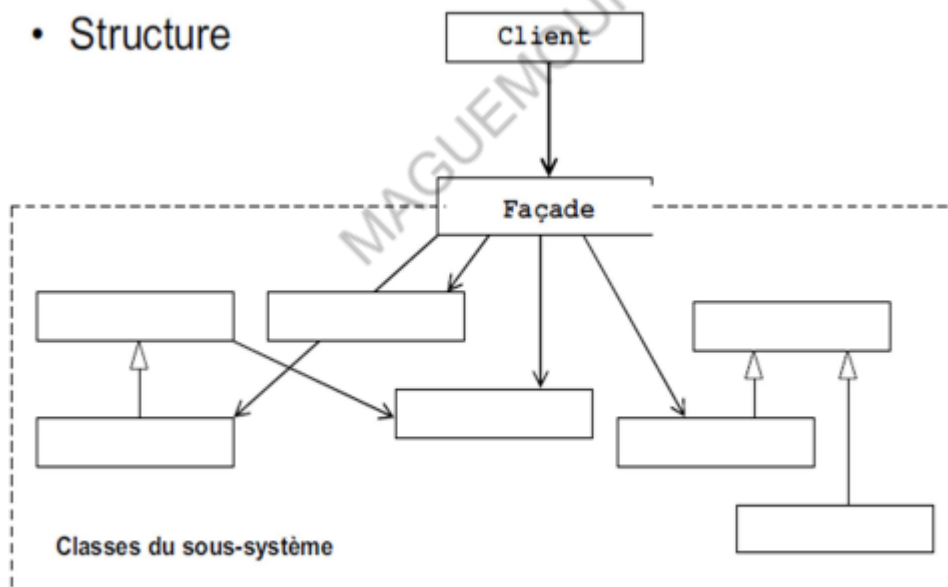
Exemple :

java

// Client : juste 1 appel !

```
List<Vehicule> résultats = WebServiceAuto.chercherVehicule(20000, 5000);
```

• Structure



Structure Façade

Éléments :

- **Client** : Utilise uniquement la **Façade** (simple).
- **Façade** : Cache la complexité du **sous-système** (boîte noire avec classes A, B, C...).

Fonctionnement :

Le client appelle `Façade.méthode()` → La façade orchestre les appels aux classes internes (A, B, C...) **à sa place**.

Avantage :

- Le client ne voit **qu'une interface simple**, pas le bazar interne.

Mini-Guide Façade Pattern

Participants

1. **Façade** (ex: `WebServiceAuto`) :
 - **Interface simplifiée** pour le client.
 - **Orchestre** les appels aux composants complexes (`Catalogue`, `GestionDocument`, etc.).
2. **Classes du sous-système** :
 - Fonctionnent **indépendamment** (ex: `Catalogue.retrouverVehicule()`).
 - Ignorent la façade → **réutilisables ailleurs**.

Collaborations

- **Client** → Appelle **seulement la façade**.
- **Façade** :
 1. Reçoit la requête.
 2. **Adapte** les paramètres (ex: transforme `prixMoyen` → `prixMin/prixMax`).
 3. **Appelle les composants nécessaires** et agrège les résultats.

Domaines d'utilisation

1. **Simplifier un système complexe** (ex: API REST qui cache une base de données + services métiers).
2. **Découpler des sous-systèmes** (chaque équipe travaille sur son module, la façade unifie tout).
3. **Protéger l'interne** (éviter que les clients accèdent à des fonctionnalités sensibles).

Exemple concret :

java

// Client :

```
voiture = WebServiceAuto.acheter(20000, 5000);
```

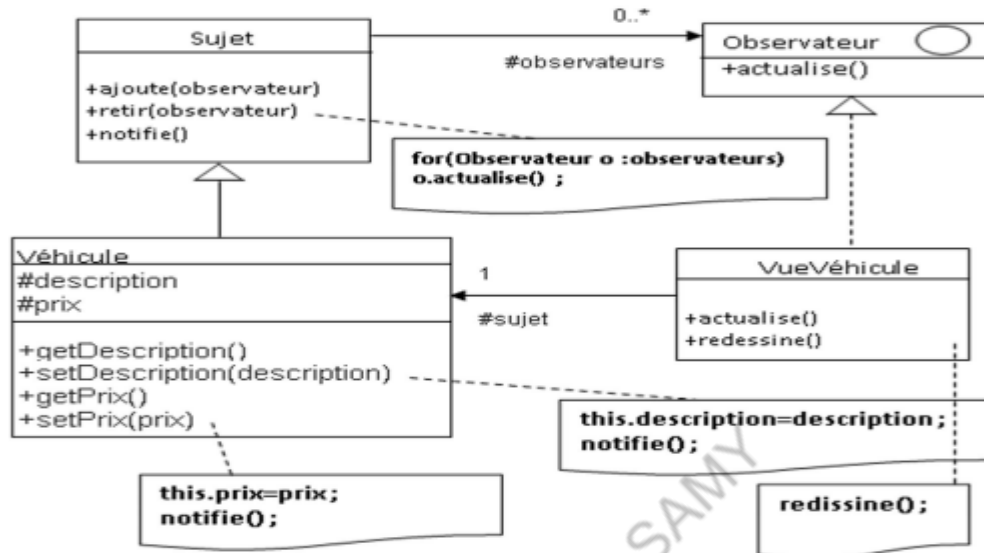
// La façade :

1. Recherche dans `Catalogue`.
2. Génère le contrat avec `GestionDocument`.

3. Propose une reprise si besoin.

8. Pattern Observer

Le **pattern Observer** établit une dépendance entre un **sujet** (ex. : un véhicule) et plusieurs **observateurs** (ex. : affichages). À chaque modification du sujet, tous les observateurs sont automatiquement notifiés pour **mettre à jour leur état**, permettant ainsi une **synchronisation en temps réel**.



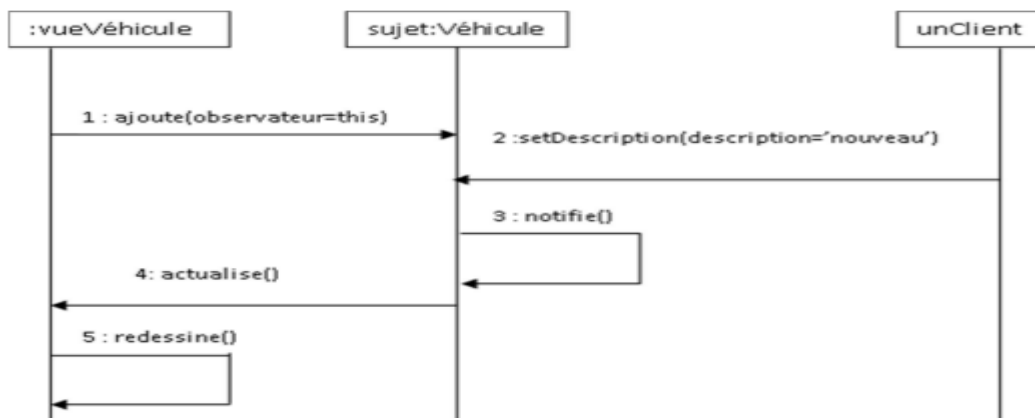
Le Pattern Observer appliqué à l'affichage de véhicules

La classe **Sujet** permet à des objets (observateurs) de s'inscrire pour recevoir des notifications.

Véhicule hérite de **Sujet** et contient **description** et **prix**.

VueVéhicule, qui implémente l'interface **Observateur**, s'inscrit via `ajoute()`.

Lorsqu'un attribut du véhicule change, la méthode `notifie()` est appelée et déclenche la méthode `actualise()` chez chaque vue, qui redessine alors l'affichage via `redessine()`.

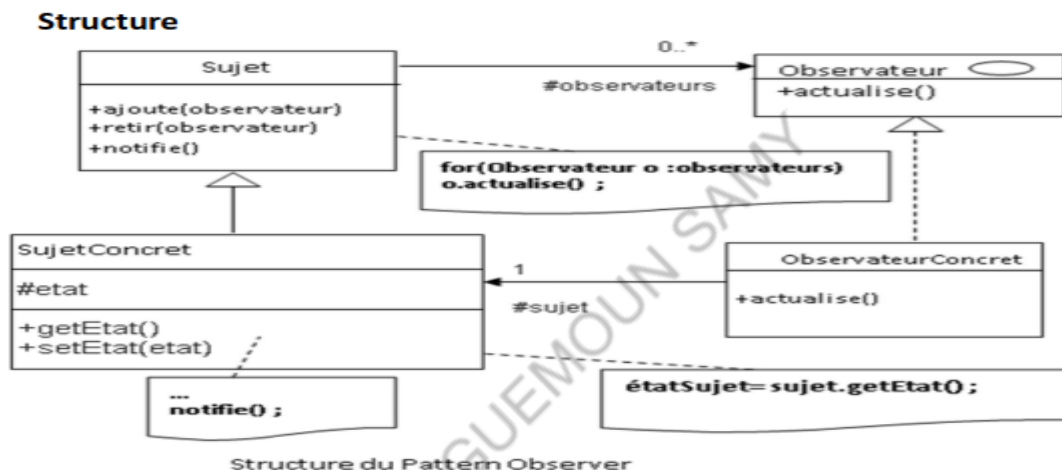


Ce diagramme illustre le fonctionnement du **pattern Observer** appliqué à un système de véhicules.

1. **VueVéhicule** s'abonne aux mises à jour d'un **Véhicule** en appelant `ajoute(this)`.
2. Lorsque le véhicule modifie sa description (`setDescription('nouveau')`), il déclenche `notifie()`.
3. La méthode `notifie()` avertit tous les observateurs en appelant leur méthode `actualise()`.
4. En réponse, **VueVéhicule** met à jour son affichage via `redessine()`.

Bénéfice : Découplage total entre le sujet (**Véhicule**) et ses observateurs (**VueVéhicule**), permettant des mises à jour automatiques sans dépendance directe.

Exemple : Si le prix d'un véhicule change, toutes les vues connectées se rafraîchissent instantanément. 🔄



Pattern Observer :

- **Sujet** : classe abstraite qui permet d'ajouter/retirer des observateurs.
- **Observateur** : interface avec la méthode `actualise()` pour recevoir des notifications.
- **Sujet concret** : classe qui envoie une notification aux observateurs lorsqu'un changement d'état survient.
- **Observateur concret** : contient une référence vers le sujet et implémente `actualise()` pour se mettre à jour.
- **Collaboration** : lorsqu'un sujet change, il appelle `notifie()` pour informer tous ses observateurs qui, à leur tour, interrogent le sujet pour actualiser leur état.
- **Domaines d'application** :
 - Quand un changement d'un objet affecte d'autres objets de manière dynamique.

- Quand un objet veut notifier d'autres sans être couplé à eux.
- Exemples : gestion d'événements (`java.beans`), synchronisation de threads (`java.util.concurrent`), callbacks.

MAGUEMOUN SAMY