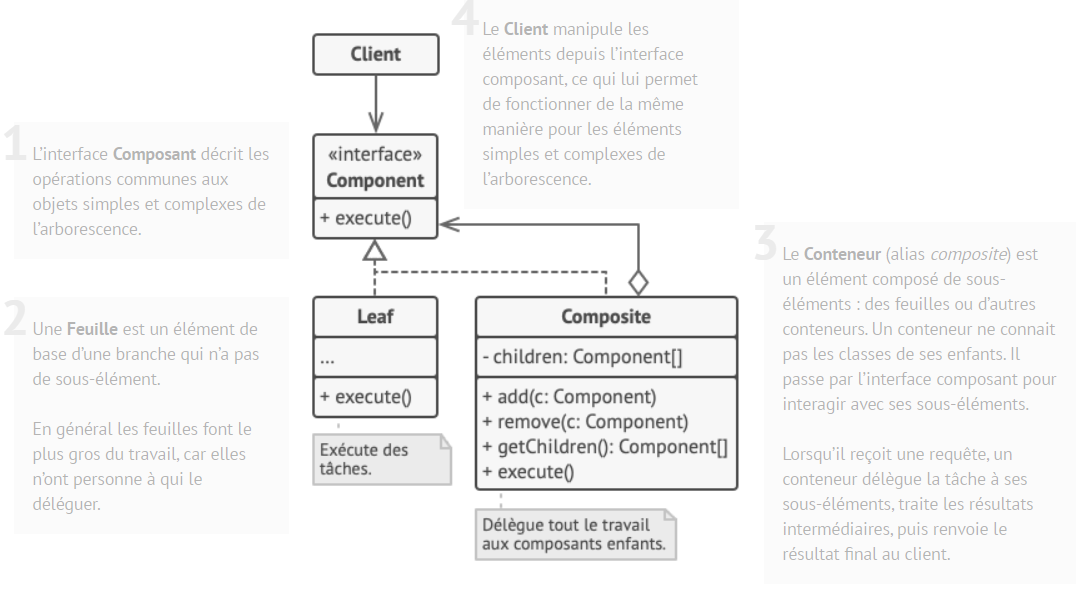
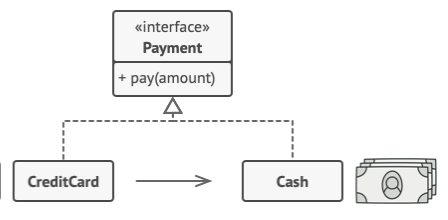
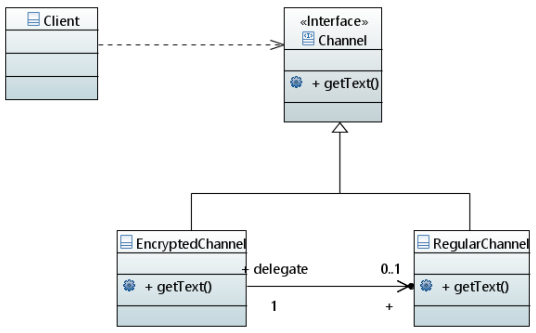
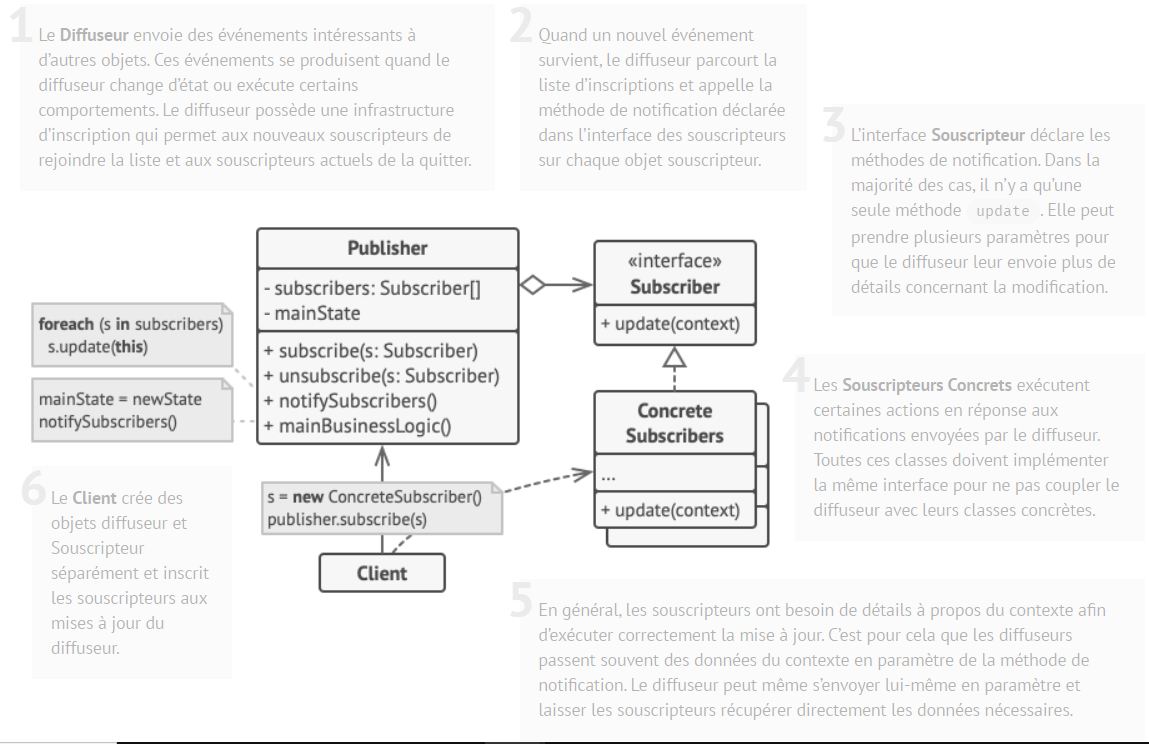
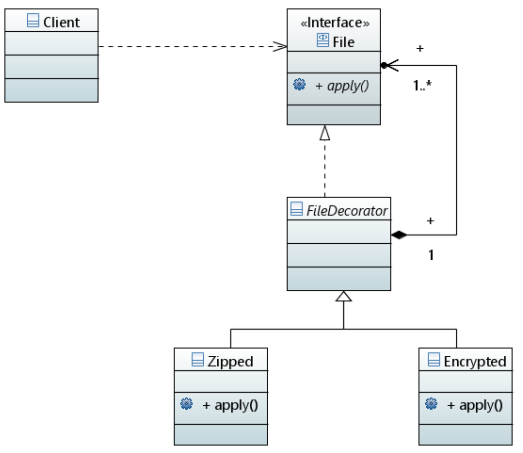
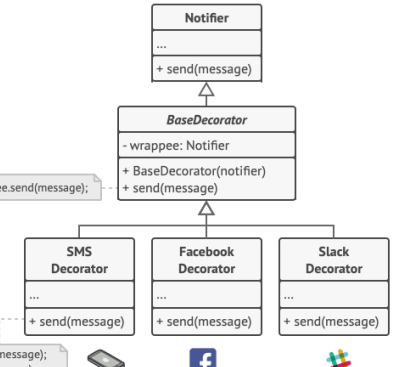
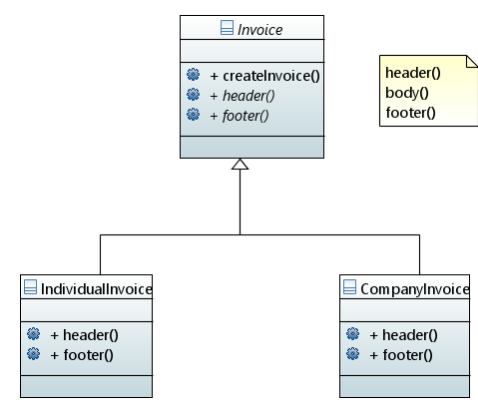
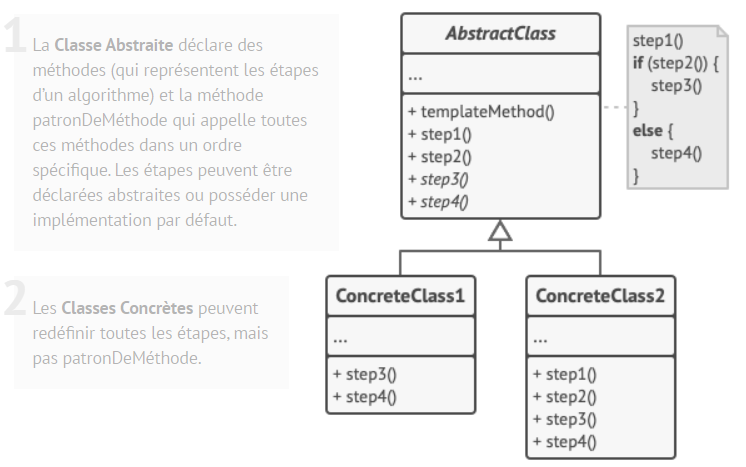
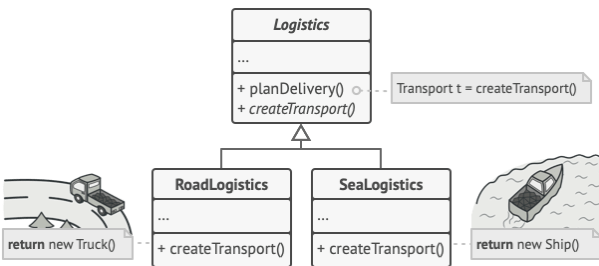
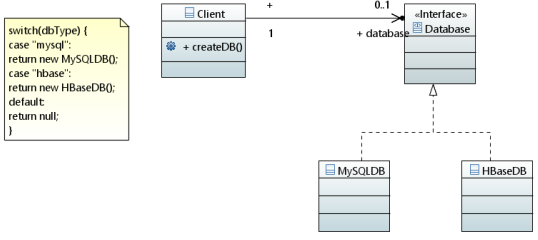
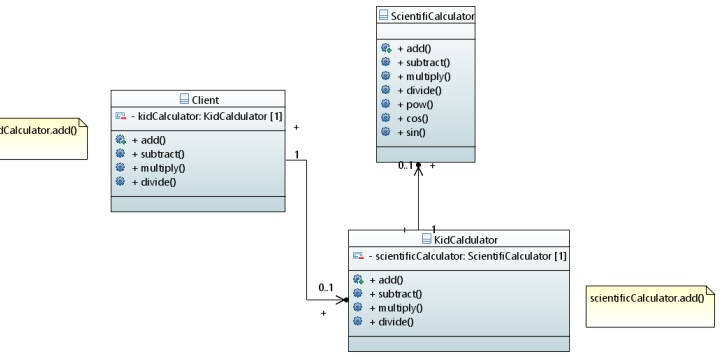
**Q**.Nommez une différence entre les concepts d'architecture et de conception. Pour chaque élément donné, indiquez s'il est pertinent pour l'architecture ou la conception **R**. Classe-Conception.Niveau-architecture.Base de données-architecture. Héritage-Conception. Architecture-Architecture. Différence: l'architecture réponds au "quoi?". Donc, ça réponds à quels paquetages, composantes et relations que nous voudrons avoir pour le système tandis que la conception consiste au "comment" nous allons concrètement implémenter l'architecture. **Q**.Dans vos propres mots, définissez le cloning ou la duplication de code. Quel problème cela pose-t-il lors de la maintenance? **R**. La duplication de code est en gros du code qui est copié-collé à plusieurs places. En effet, cela peut avoir des répercussions lors d'une maintenance. Effectivement, les développeurs devront corriger la même erreur à toutes les places du code dupliqué. Cela peut devenir une tâche fastidieuse et on pourrait oublier à certaines places de corriger un bug qui a été réglé auparavant. **Q**. Vous recevez des métriques pour la classe IncomeTaxCalculator et ses méthodes. Sur la base des métriques, identifiez et nommez un antipatron présent. Justifiez votre réponse. Comment refactoriser le code pour supprimer l'antipatron? **R**. L'antipatron manque d'abstraction est pertinent aux données de la classe. Une abstraction a plus d’une responsabilité. Refactoring🡪Extraire de la classe ou de la superclasse. **Q**.Nommez une différence entre les architectures en couches et multi-niveaux. Quel style est distribué et lequel est modulaire? **R**. L'architecture multi-niveau est distribué. L'architecture en couches est modulaire.La différence entre les deux architectures est qu'avec les architectures en couche, on ne peut pas communiquer entre les couches qui sont au même niveau. Cependant, avec l'architecture multi-niveau, il est possible de communiquer entre chaque couche qui sont au même niveau. **Q**. Vous développez un système de données pour les cas de COVID-19 dans la ville de Montréal. Le système stockera des données patient sensibles, accessibles uniquement à un nombre limité d'utilisateurs. Le nombre de caisses stockées est limité par la population de la ville, nous savons donc de combien de ressources nous avons besoin. Quel style architectural utiliseriez-vous pour implémenter ce système et pourquoi?**R**. Selon l'énoncé, il est mentionné que nous voulons stocker les données des patients sensibles. En effet, c'est un principe de confidentialité. Donc, on peut penser à la sécurité du logiciel. Ensuite, on mentionne que les informations sont diponibles à un certain nombre de personnes. Ainsi, on peut penser à l'authenticité lorsqu'une personne tente d'accéder aux informations. Si elle ne fait pas partie des membres qui ont le droit d'accès le système refusera l'accès à cette dernière. Cela signifie que la sécurité est encore plus importante. Ainsi, on peut penser à une architecture du style CBA : Architecture des composantes. Aussi, au niveau des erreurs, il en faut le moins possible : reliability. **Q**.Dans vos propres mots, décrivez le fonctionnement de MapReduce. Quel style architectural suit-il?**R**.MapReduce constitue un modèle de programmation pour l'analyse distribuée des mégadonnées. Elle se sépare en 2 parties: une tâche map et une tâche reduce. La tâche map constitue un travail d'analyse sur les données comme un tri ou une addition tandis que reduce représente un travail de sommation. Donc, sa tâche sera d'agréger les résultats des tâches map individuelles. MapReduce suit l'architecture master-slave. Donc, le master s'occupe de distribuer les tâches et de balancer la charge entre les ouvriers. Les ouvriers peuvent accepter des tâches map ou des tâches reduce. **Q**. Dans vos propres mots, définissez les bases de données de documents. Nommez deux de leurs avantages. **R**. Les bases de données "Document" traitent les données comme des fichiers qui peuvent avoir n'importe quelle taille et complexité, mais qui suivent un encodage spécifique (JSON, XML, etc). On peut demander des documents grâce aux clés. Deux de leurs avantages sont: 1)Si notre application ont des données qui sont des documents et que la structure peut varier dans le temps, cette base de données s'adapte très facilement.2) Avec cette base de données, on peut demander des documents en fonction de leurs attributs. **Q**. Vous développez un système de stockage des dossiers des patients. La plupart des cliniques enregistrent les dossiers patient sous forme de feuilles de calcul. Chaque clinique peut avoir des colonnes spécifiques différentes, mais toutes ont les mêmes données générales. Quel type de base de données NoSQL utiliseriez-vous pour implémenter le système?**R**. Pour implémenter ce système, j'utiliserais une base de données Wide-column, puisque les clients enregistrent les dossiers sous formes de feuilles de calcul (je supposes comme Excel), Wide-column est le système qui se rapproche le plus de ces fonctionnements. Donc, pour le client qui communique avec l'interface, il ne sera pas complètement perdu puisqu'il assume l'existence d'un schéma et que Wide-column conserve ces concepts. En effet, il conservent les concepts de tableaux et de colonnes qui se retrouvent aussi dans les feuilles de calcul. De plus, la définition des colonnes sont très flexibles, ce qui est un point important pour ce système puisque chacune des cliniques peuvent avoir certaines colonnes spécifiques à elles. Ce système sera capable de supporter cela. **Q**. Nommez la fonctionnalité qui rend Spark plus rapide que Hadoop. Expliquez pourquoi. **R**. La fonctionnalité qui rend Spark plus rapide que Hadoop sont les RDDs. Ici, on parle du Resilient Distributed Dataset. En effet, Spark fonctionne en mémoire. En d'autres termes, Spark est efficace lorsqu'on parle de traitement rapide des données ainsi qu'un processus itératif. D'ailleurs, en mémoire, Spark réalise des tâches en mémoire 100 fois plus rapidement que Hadoop et 10 fois plus rapidement lorsque c'est sur disque. Effectivement, la cause est qu'on utilise moins d'écritures et de lectures par cycle. **Q**. Dans vos propres mots, définissez des bases de données de wide-column. Nommez deux de leurs avantages. **R**. Les bases de données wide-column sont des bases de données de type NoSQL. Au niveau conceptuel, les wide-columns se rapprochent des bases de données relationnelles. Cependant, la notion de colonnes est assez flexibles. Effectivement, il est possible de regroupper des attributs dans des familles de colonnes. En quelques sortes, on pourrait considérer que les wide-columns sont des bases de données de type key-value en 2 dimensions. Il faut savoir qu'ils possèdent des colonnes et des rangées. Au niveau des avantages, on pourrait penser au fait qu'ils peuvent supporter de grandes quantités de données. Aussi, la flexibilité des familles colonnes améliore la maintenabilité des données et la gestion des données variantes. **Q**. Vous développez un système de prévision météorologique. Le système reçoit les données météorologiques toutes les 30 minutes, il reformate les données et les stocke dans une base de données. Après une journée, le système utilise les données du jour pour mettre à jour un modèle de prédiction entraîné et stocke la prédiction mise à jour dans une base de données. Quel type d'architecture de traitement (Lambda ou Kappa) utiliseriez-vous et pourquoi? **R**. L'architecture de traitement que je choisirais serait Lambda. En effet, on mentionne que les données sont reçues à chaque 30 minutes et elles sont traitées. Alors, l'architecture Lambda permet de traiter ces informations de façon efficace. Il faut savoir que ce n'est pas du vrai temps réel même si on reçoit des données à chaque 30 minutes. On doit avoir une architecture qui supporte bien le temps réel, mais pas que. Il faut des réponses rapides et il faut un système qui permet de supporter plusieurs mises à jour de façon efficace. **Q**. Nommez un défi de la modularité basée sur les bibliothèques. Comment les architectures orientées services résolvent-elles ce défi? **R**. Un des problèmes de la modularité par des librairies est qu'il y a le partage de code. Ainsi, il y a des dépendances technologiques et temporelles. **Q**. Décrivez dans vos propres mots l'antipatron de la loi sur le drapeau rouge (Red Flag Law). **R**. Pour représenter cet antipatron, on peut penser à des personnes qui devaient marcher devant des automobiles avec des drapeaux rouges. En effet, les automobiles devaient circuler à la même vitesse que les piétons. Au niveau des projets informatiques, cela signifie que les développeurs n'utilisent pas les bonnes pratiques des microservices. En effet, tout ce qui a attrait au processus en cascade, aux tests manuels et aux commits pendant des moments de maintenance fait en sorte que la vitesse d'efficacité est très lente. Effectivement, le but des microservices et de travailler sur différentes tâches afin d'être plus efficace et de mieux séparer les tâches. Pour qu'on puisse utiliser une architecture en microservice, il faut utiliser des pratiques DevOps pour apporter un côté d'automatisation (plutôt que de faire des tests manuellement), organiser les équipes adéquatement et utiliser le modèle Strangler Application. **Q**. Nommez un avantage et un désavantage des services SOAP. **R**. un grand avantage d'utiliser SOAP est qu'il est indépendant du protocole de communication. Par contre un désavantage de ce type de service est qu'il n'est pas efficace. **Q**. Comment les microservices sont-ils conçus pour l'échec? **R**. Les microservices sont conçus pour l'échec en faisant en sorte d'être capable de réduire l'impact sur l'utilisateur si un microservice échoue. De plus, on peut effectuer la pratique de l'injection aléatoire, inattendue et intentionnelle de pannes dans les services, les nœuds ou même les centres de données pour voir comment les différents microservices réagissent. Cela permet de vérifier le comportement en cas d'échec et de s'assurer que l'impact de l'echec sera minimale sur l'utilisateur. **Q**. Vous développez un système de gestion de la base de données d'une librairie électronique. Le système sera utilisé par les administrateurs et les utilisateurs. Un administrateur peut uniquement ajouter, supprimer ou modifier des livres et leurs informations. Un utilisateur peut parcourir des catalogues, rechercher des livres par catégorie ou par auteur, commander ou réserver un livre. Concevoir et décrire une architecture SOA pour ce système. Vous pouvez combiner plusieurs styles architecturaux pour votre implémentation. **R**. *Pour ce système*, nous pouvons utiliser une architecture REST qui aura plusieurs routes pour obtenir les informations souhaités. Avec REST, nous pouvons aussi effectuer de la pagination pour la recherche de livres. Donc, ce système aura un client avec un serveur REST ayant ces différentes routes*. Pour l'administrateur*: api/v1/book 🡪Pour cette route, les verbes GET et POST seront supportés. GET permet d'avoir l'ensemble des livres. Cette route supporte la pagination pour parcourir le catalogue de livres. POST permet de créer une nouvelle entrée dans la base de données. api/v1/book/{id} 🡪Pour cette route, les verbes GET, PATCH et DELETE seront supportés. GET permet d'obtenir un livre précis dans la base de données. DELETE permet d'effacer un livre en particulier. PATCH permet de modifier/remplacer un livre précis dans la base de données. *Pour l'utilisateur* : api/v1/book 🡪 Pour cette route, les verbes GET seront supportés. GET permet d'avoir l'ensemble des livres. Cette route supporte la pagination pour parcourir le catalogue de livres. Pour cette route, il y aura des paramètres optionnels qui permettront de filtrer la liste des livres à retourner: author pour filtrer selon l'auteur et category pour filtrer par categorie. api/v1/book/{id}🡪Pour cette route, les verbes GET seront supportés. GET permet d'obtenir un livre précis dans la base de données. api/v1/book/{id}/reserve 🡪 Ici, que le verbe POST sera supporté. Celui-ci enverra toutes les informations nécessaire pour réserver le livre en question. api/v1/book/{id}/order 🡪 Ici, que le verbe POST sera supporté. Celui-ci enverra toutes les informations nécessaire pour acheter le livre en question Il faut aussi noter qu'il faudra ajouter un système d'authentification au sein du système et aussi d'implémenter cela dans l'architecture REST. C'est-à-dire, si un utilisateur effectue une rêquete, de vérifier que cette personne a les droits pour effectuer la requête avant de l'executer. Sinon, retourner le code d'erreur HTTP approprié. L'interface utilisateur pourrait être RPC/SOAP. **Exemple d’examen final (question 4)** Vous concevez une application Web éducative sur l'histoire du monde. L'application présente aux utilisateurs une carte du monde et une chronologie. En faisant défiler la chronologie et en cliquant sur un endroit spécifique sur la carte, l'utilisateur peut accéder à certains événements historiques qui se sont produits à cet endroit à ce moment-là. L'application explore YouTube et Wikipedia pour trouver des vidéos pertinentes (marquées avec «histoire» ou «éducation») et des articles pour les événements spécifiques. Les ressources sont ensuite accessibles via des fenêtres dans le cadre de l'application ("in-frame windows"). Les liens vers les articles et les liens vers les vidéos sont gérés par différentes fenêtres dans le cadre. Les utilisateurs peuvent enregistrer leur propre chronologie avec les événements qui les intéressent et recevoir une notification des nouvelles vidéos et articles, lorsqu'ils deviennent disponibles. L'application propose également un forum de discussion où les utilisateurs peuvent échanger des opinions sur des événements historiques de manière civile. Un utilisateur peut également recommander l'ajout d'une ressource (vidéo ou article) sur un événement spécifique. La recommandation est examinée par un modérateur et si elle est acceptée, elle est ajoutée à la carte. Bien que YouTube et Wikipédia soient actuellement les principales ressources, nous souhaitons que l'application soit conçue de manière à ce que d'autres ressources puissent être facilement ajoutées à l'avenir. Nous souhaitons également que notre application soit accessible par plusieurs appareils (ordinateurs personnels, tablettes et téléphones portables). En ce qui concerne les ressources, bien que certaines d'entre elles soient externes (comme les vidéos YouTube ou les articles Wikipédia), les liens vers ces ressources ainsi que d'autres ressources propriétaires (y compris les vidéos et articles produits en interne) doivent tous être gérés par l'application avec leurs relations les uns avec les autres et avec les événements pertinents sur le plan temporel et géographique. 1) Quel ou quels styles architecturaux utiliseriez-vous pour mettre en œuvre ce système? Justifiez et expliquez vos choix. Utilisez également des arguments relatifs à la qualité pour étayer votre décision. **R**.MVC pour l’interaction web avec la carte, microservices (carte, forum, ajout des ressources) pour gérer les accès différents. Les qualités ciblées : utilisabilité, portabilité maintenabilité. CBA (MVC) et SOA (microservices) sont acceptés (100%). 3) Mentionnez au moins deux patrons (patrons de conception ou patrons événementiels) que vous utiliseriez dans votre système. **R**. Patron State (pour YouTube et Wikipedia). Il permet l’extensibilité et de traiter la présentation en façon différente selon le type. Patron Observer (pour l’enregistrement aux évènements). Patron Reactor (pour l’ajout des ressources). 4) Comment implémenteriez-vous la partie données de votre système? Commentez l'organisation des données et si vous utiliserez des solutions spéciales pour stocker, ingérer ou analyser les données. **R**. Une BD pour les données de l’application : NoSQL Graph pour capturer les relations complexes. Une BD pour le forum : NoSQL Document/Wide-column, traiter les fils de forum en tant que documents. Une BD « personnel » pour la carte de chaque utilisateur : NoSQL Wide-column/relationnelle, peut-être pas beaucoup des données. Microservices => Une BD par service.

**Cours1:Principes SOLID et Patrons de conception.** **Q**. Vous travaillez pour un bureau de comptabilité et vous développez un logiciel pour calculer la taille totale du système de fichiers. Le taille (size) d’un dossier (Directory) est le somme des tailles des fichiers (File) dans le dossier. **R**. ***Composite****.* Vous avez des objets qui doivent être traités de la même manière (c'est-à-dire ajouter de la taille). Mais certains d'entre eux sont des composites contenant d'autres plus simples. (Les dossiers contiennent des fichiers). **Q**. Nous avons une configuration où nous envoyons et recevons des données sur le réseau. À un moment donné, le flux de données en entrée commence à être encrypté. Comment doit-on changer le code client? **R**. **Proxy**. Vous devez contrôler l'accès à un objet donné (RegularChannel). Vous fournissez un espace réservé (Channel) qui détermine dynamiquement comment l'objet sera accédé.

**Q**. Vous développez un tableur qui permet de calculer automatiquement les cellules en fonction des formules, qui dépendent d'autres cellules. **R**. **Observer**. Un objet (Cellule) doit changer d'état en fonction de ce qui arrive à un autre objet (Formule). L'objet qui change est le sujet et l'objet qui doit changer est un observateur qui souscrit au sujet. **Q**. Vous souhaitez développer un lecteur de fichier capable de lire un fichier, qui peut être (a) compressé, (b) encrypté (c) compressé et encrypté ou (d) encrypté puis compressé et encrypté à nouveau. **R**. **Decorator**. Vous avez des objets qui peuvent avoir des comportements ou des états différents (tels que compressés ou encryptés). En plus de cela, vous voulez que ces états soient combinés de toutes les manières possibles. ––––**Q**. Vous avez un système qui imprime deux types de factures, un pour les individus et un autre pour les compagnies, qui diffèrent entre eux par le graphisme d'en-tête et de pied de page. Le contenu au milieu est une liste de tous les éléments de la facture, leurs prix et le total. **R**. **Template method**. Vous avez un comportement complexe (createInvoice()) qui change selon les différents scénarios (IndividualInvoice, CompanyInvoice). Cependant, certaines parties du comportement restent les mêmes quel que soit le scénario (header(), footer()). **Q**. Vous avez plusieurs bases de données qui ont chacune leur façon de s’initialiser et vous souhaitez que votre application puisse construire et utiliser chacune d’entre elles de façon interchangeable (db1, db2, ...) au choix de l'utilisateur. **R**. **Factory method**. Vous disposez d'un objet (Database) avec différentes implémentations (MySQLDB, HBaseDB). Une implémentation n'est créée que sur le choix d'un client. . **Q**. Vous êtes embauchés par MathWorks (les développeurs de Matlab). La compagnie voudrait fournir des logiciels aux écoles primaires, mais ils veulent réutiliser le code existant. On vous fournit le code pour une calculatrice scientifique et votre tâche est de développer une calculatrice pour les enfants avec juste les opérations de base. **R**. **Adapter/Facade**--> adapter le code, le réutiliser. Vous disposez d'une interface complexe (ScientificCalculator) et d'un objet (Client) qui ne peut pas la comprendre en grande partie. L'adaptateur traduit une interface en une autre comprise par le client. Façade simplifie une interface complexe ou en fusionne plusieurs pour ne publier que certaines opérations. **Cours2 : Styles d’architecture**

**Q**. Nous recherchons un système de données distribué à haute tolérance aux fautes et à haute disponibilité. La base de données contient de nombreuses répliques et les nœuds communiquent entre eux pour identifier des erreurs. **R**.**Peer to peer** car les noeuds communiquent entre eux pour identifier des erreurs. Pas centralisée, robuste aux échecs des participans = tolérance aux fautes. **Q**. Nous recherchons un système d’analyse de données. Un travail analytique est soumis et il est répliqué sur plusieurs nœuds. Chaque nœud travaille sur une partie des données. Les résultats sont assemblés et agrégés avant d’être retournés. **R**. **Master-slave** car il n'y a qu'une seule tâche étant répliquée et exécutée en parallèle par plusieurs serveurs. Le fait que la tâche est appliquée sur des partitions différentes des données est ce qui produit des résultats frucueux et pas répétés. Si la partition de données était seule et les tâches étaient différentes, nous parlerions de Blackboard ou de pipe-filter. **Q**. Nous recherchons un système pour stocker et analyser les déclarations de revenus. La capacité de calcul et de stockage ne sont pas des problèmes et le nombre de déclarations de revenus est à peu près prévisible. **R**. **Monolithique** car la sécurité est de la plus haute importance! L'adaptabilité n'est pas un enjeux, donc on peut déployer le mainframe directement sur une infrastructure qui ne va pas changer. Une sorte d'interface peut être nécessaire pour soumettre les relevés des impôts (maintenant, le web aussi), mais l'analyse se produit localement et à l'interne. **Q**. Nous recherchons un système de données pour vendre les produits d’un magasin en ligne. Chaque produit peut être identifié uniquement. L’utilisateur peut simplement voir un produit, ajouter un produit, supprimer un produit ou mettre à jour un produit. **R**. **REST** (GET, PUT, POST, DELETE) car nous sommes certain qu'on cherche pour une sorte d'architecture web, donc plus probablement client-serveur. Plus spécifiquement, c'est un bon cas pour le SOA ou multi-serveur (GUI, service, BD). Basé sur les identifiants uniques des produits, nous sommes amenés vers une solution SOA particulière; le REST qui fonctionne avec des URIs. Basé sur l'ensemble des opérations spécifiques (CRUD), SOAP serait trop complexe pour cette application. La différence enntre Microservices et REST peut être perçue comme l'une entre une architecture et son implémentation. **Q**. Nous recherchons un système d’analyse de données. Il y a plusieurs algorithmes pour analyser les données. Chaque algorithme change les données et les prépare pour l’algorithme suivant. R. **Pipe-filter** si l’ordre est prédéterminé, **Blackboard** si l’ordre est déterminé sur la base des résultats les plus récents.

**Cours5 : Mégadonnées** 🡪 [Une tâche **MAP** représente un travail d’analyse (p.ex., un tri, une addition, une cherche etc.) Une tâche **REDUCE** représente un travail de sommation, qui va agréger les résultats des tâches map individuelles.] **1**. Chaque serveur de données peut exécuter une réplique de l'algorithme. **2**. Les tâches map consistent à trouver le maximum dans un seul serveur. **3**. Les tâches reduce consistent à trouver le maximum par province et une autre tâche reduce pour trouver le maximum global du pays.

**Cours6 : CAP, ACID, BASE, NoSql**

**Q**.Nous développons une application pour une entreprise commerciale. L'entreprise gère les contrats et les factures des transactions commerciales. Les données doivent être stockées de manière sécurisée et persistante pendant une longue période. L'entreprise fonctionne au niveau mondial avec un grand nombre de transactions. **R**. **Graph**🡪Il y a des liens entre les pages. Les liens peuvent aussi représenter des relations complexes (hiérarchie, taxonomie, partonomie), on a un réseau des pages. OU **Document** 🡪Les contrats et les factures sont des fichiers. La quantité de données nécessite une solution NoSQL. **Q**. Wikipedia! **R**. Document🡪On peut traiter chaque page en tant que document, on peut récupérer une page de sa clé ou de ses métadonnées. **Q**. Nous avons un système de monitoring des ressources cloud. À des intervalles assez fréquents et rapides, le système envoie des mesures (CPU, mémoire, disque, réseau) pour chaque ressource (machine virtuelle). **R**. Key-value 🡪Nous avons besoin d'une efficacité accrue. La structure des données est assez simple. Il est possible de profiter de la mémoire pour une ingestion de données rapide et efficace. **Q**. Pour le système de monitorage précèdent, quelle architecture de traitement des données choisiriez-vous? **R**. **Kappa**. Si on avait seulement l’ingestion des données. Si on utilise une méthode « **push** » : les ressources envoient leurs mesures au système de monitorage. MAIS ! **Lambda** Si nous avons aussi des analyses au données du monitorage ou c’est le système qui demande les mesures des ressources (méthode « **pull** »), nous avons des problèmes. On a « l’effet de l’observateur » : en demandant les mesures, le monitorage affecte les mesures elles-mêmes (parce qu’on exécute du code supplémentaire). **Q**. Les bases de données pour les données biologiques existent depuis un certain temps déjà. Ils contiennent des données sur les gènes, les protéines, les organismes. Les entités ont des attributs, mais il est possible de découvrir de nouveaux attributs dans le futur. Diverses analyses et outils existent déjà pour nous aider à étudier le monde naturel. **R**. **Wide-column**. La flexibilité de la structure est réquise. Il est possible que des bases des données rélationnelles existent déjà. Il est certain que des clients de ces bases de données existent déjà, qui assume l’existence d’un schema.

**Cours7 : Architecture Événementielle 🡪 Monitor Object, Wrapper Façade, Component configurator, Interceptor, Extension Interface**

**Q**. Nous sommes dans un restaurant de fast food. Les clients passent leur commande à la caisse. Si la commande ne peut pas être servie immédiatement, les clients se retirent et attendent. Lorsque la commande est prête, ils reculent devant la file pour recevoir leur repas de la caisse. **R**. **Monitor Object**. Un client à la fois peut gérer les commandes, on gère de facon synchrone. Le caissier est l'objet moniteur. Les clients sont les requêtes synchronisées. Un seul client peut accéder à la caisse à un moment donné. Les clients « attendent » pendant que le caissier prépare une commande ou sert d'autres clients. Le caissier informe le client suivant lorsqu'il est prêt. **Q**. Chrome utilise des plugins pour permettre la présentation de certains types de médias, comme Flash. Lorsqu'une page contenant de tels médias doit être chargée, le navigateur vérifie si le plug-in approprié est installé. Si tel est le cas, le navigateur appelle automatiquement le plugin pour gérer le média. **R**. **Interceptor**. Le navigateur est le framework dont les fonctionnalités peuvent être étendues. Les plugins sont les intercepteurs. Le répartiteur identifie dynamiquement quel est le meilleur plugin à ajouter. **Q**. Vous avez probablement un téléphone mobile associé à un numéro de téléphone spécifique. Les gens peuvent appeler ce numéro et vous joindre. Lorsque quelqu'un le fait, votre téléphone sonne et vous pouvez choisir de répondre et d'engager la conversation avec la personne de l'autre côté de la ligne. **R**. **Reactor or acceptor/connector**. Le fournisseur de télécommunications est le Reactor. Votre numéro de téléphone est le Handle. Vous (avec votre téléphone portable) êtes l'EventHandler. Acceptor/Connector peut être une autre réponse, mais généralement l'acceptation d'un appel vient après la partie connexion, donc Reactor est une meilleure réponse. Dans le cas d’un centrer d’appel, Acceptor/Connector est la meilleure solution. **Q**. Taxis dans un aéroport. **R**. **Leader/Followers**. La ligne de taxis est le pool de fils. Les taxis sont les gestionnaires de l'événement. Les passagers sont les sources de l'événement.

**Cours11 : AOP**

**Q**. On veut implémenter un système de sécurité qui utilise plusieurs dispositifs (cameras, détecteurs de mouvement, serrures) pour surveiller et assurer la sécurité d’un building. Selon la journée (jour de semaine ou de fin de semaine) et l’heure (jour vs. nuit), le système utilise différents dispositifs (e.g. la nuit uniquement les caméras).Quel patron de conception utiliseriez-vous pour implémenter ce système ? Comment peut-on transformer cette conception en utilisant l’AOP ? **R**. **Patrons Strategy**. 1. Le Contexte garde une référence vers une des stratégies concrètes et communique avec cet objet uniquement au travers de l’interface stratégie.2. L’interface Stratégie est commune à toutes les stratégies concrètes. Elle déclare une méthode que le contexte utilise pour exécuter une stratégie. 3. Les Stratégies Concrètes implémentent différentes variantes d’algorithmes utilisées par le contexte. 4. Chaque fois qu’il veut lancer un algorithme, le contexte appelle la méthode d’exécution de l’objet stratégie associé. Le contexte ne sait pas comment la stratégie fonctionne ni comment l’algorithme est lancé. 5. Le Client crée un objet spécifique Stratégie et le passe au contexte. Le contexte expose un setter qui permet aux clients de remplacer la stratégie associée au contexte lors de l’exécution.

Chaque stratégie concrète réduit les dispositifs à surveiller à un sous-ensemble de l’ensemble original. En cours d’exécution on vérifie l’heure et on choisit la stratégie appropriée. ***Solution AOP***🡪La stratégie abstraite est remplacée par un aspect abstrait StrategyProtocol, qui conserve une paire de Contexte (temps) et Stratégie et un aspect. L’AOP réduit la profondeur de l’héritage and libère les stratégies concrètes, comme classes du domaine, pour étendre d’autres classes. ***Impact sln AOP***🡪 Réduit l’enchevêtrement du code, la dispersion du code, La profondeur de l’héritage, la compléxité. Accroit Le couplage, cohésion, NLC car stratégie est enchevêtrée avec le patron Observer.

Principes SOLID : 1) **« Parce que tu peux ne signifie pas que tu dois. »**Une classe ne devrait avoir qu’une seule responsabilité. Une classe ne devrait avoir qu’une seule raison de changer. Haute cohésion + Faible couplage ➔ Moins de responsabilités, moins de raisons de changer.2) **La chirurgie à cœur ouvert :** Les entités logicielles doivent être ouvertes aux extensions, mais fermées aux modifications. L’héritage et le polymorphisme favorisent l’extension. Refactoring : Remplacer la vérification de type par du polymorphisme (si la hiérarchie existe déjà). Refactoring : Remplacer la vérification de type par le patron « Stratégie ». (si la hiérarchie n’existe pas encore). **« Si cela ressemble à un canard, sonne comme un canard, mais a besoin de piles, vous avez probablement la mauvaise abstraction. »** Les fonctions qui utilisent des références à des classes de base doivent pouvoir utiliser des objets de classes dérivées sans le savoir. Les méthodes dérivées ne peuvent pas attendre plus que les méthodes de base. Les méthodes dérivées ne peuvent pas fournir moins que les méthodes de base. **« Tu veux que je mette ça où? »** De nombreuses interfaces spécifiques aux clients valent mieux qu’une seule interface. **« Souderiez-vous une lampe directement sur le câblage électrique dans le mur? »** Dépend des abstractions. Ne dépend pas des classes concrètes. Si on dépend de bibliothèques logicielles, notre système est enfermé dans un langage spécifique. (DIP violé). Si on dépend des APIs REST, notre système reste indépendant des langages. (DIP garanti).

**Architecture vs. Conception**🡪 **Architecture** : On veut une couche de GUI, une couche d’analyse et une couche de stockage des données, réfère à l’environnement d’exécution/déploiement de l’application. L’application sera disponible comme un service déployé en infonuagique, réfère à l’environnement d’exécution/déploiement de l’application. On a besoin d’une base de données NoSQL avec un taux de disponibilité élevé, réfère à une couche/niveau spécifique de l’application. Il mentionne aussi une exigence non-fonctionnelle. NoSQL est une technologie spécifique, mais pas une implémentation particulière. **Conception :** Toutes les nouvelles applications doivent étendre l’interface Application, réfère à un détail d’implémentation. Il requiert des modifications au niveau de la classe. Une méthode prend le type d’un objet comme paramètre et retourne une instance de ce type en appelant le constructeur privé de la classe correspondante, réfère à un détail d’implémentation. Il requiert des modifications au niveau de la classe. Il mentionne aussi un patron de conception (Singleton).

**Styles Architectureaux** : \*\*Architectures Monolithique :Un seul programme. Un seul endroit de changement. Il n’y a pas de distinction entre l’IU, la logique, les données etc. Les éléments du programme sont fortement liés. Le programme est développé et maintenu seulement par une petite équipe de développeurs/experts. Connaissance absolue et concentrée. Il y a juste un propriétaire du système entier. Contrôle absolu. La logique et les données existent dans un seul lieu. La propriété est bien protégée. Chaque copie du programme s’exécute sur un seul ordinateur. Les frais d’exécution sont réduits. **Avantages :** Sécurité maximale! Aucune opportunité pour une attaque externe. Le programme est souvent efficace, parce que tout est chargé en mémoire et il n’y a pas de délais liés au réseau ou au chargement de modules. Grâce à la taille de l’équipe, la communication et l’efficacité sont optimisées. Le développement, la maintenance et l’évolution du système sont la responsabilité d’une seule entité. **Désavantages :** Pas de réutilisabilité. Il est impossible de réutiliser des parties du système, à moins de les copier en entier. Cela crée des clones. Pas d’évolutivité ou d’extensibilité. Difficile à maintenir! Difficile d’ajouter de nouveaux développeurs. Frais croissants pour la maintenance du système.

\*\*Architectures Modulaires : Le système comprend plusieurs modules (Plusieurs programmes). Les fonctionnalités et les responsabilités sont plus distinctes. Les modules sont développés et compilés séparément. Les modules sont plus petits et plus facile à gérer. Le développement est aussi séparé entre plusieurs plus petites équipes. Ces architectures implémentent les principes de SRP et DIP. La propriété du système appartient toujours à une seule partie. **Avantages :** Le développement est parallélisé, donc il est plus efficace. La maintenance est généralement plus facile, car les changements locaux et internes n’affectent pas les autres modules. Les modules sont aussi autonomes que possible au niveau des tests (unitaires). Il est plus facile d’ajouter du nouveau personnel. Les modules peuvent être réutilisés et changés. **Désavantages :** La distribution du travail augmente le besoin de communication entre les équipes. La connaissance du système entier est perdue. Il y a moins de dépendances, mais elles sont peut-être plus importantes. On doit investir plus d’efforts dans la gestion du système(Tests, débogage, monitoring, maintenance)

1) Pipe-filter🡪écomposition d’une fonctionnalité en composantes successives appliquant des transformations incrémentales sur les données. Un producteur envoie des données par des tuyaux et à travers de filtres à un consommateur. Les tuyaux sont les canaux de communication et les connecteurs entre les filtres et entre les producteurs et les consommateurs. Ils peuvent aussi fonctionner en tant que des agrégateurs de données ou des synchronisateurs. Les filtres sont des transformations appliquées progressivement aux données. **Avantages :** Efficace : Les filtres peuvent être parallélisés. Les filtres sont indépendants des producteurs et des consommateurs et ils peuvent être réutilisés. Il est possible d’ajouter ou de supprimer des tuyaux et des filtres en cours d’exécution. S’il n’y a pas suffisamment de données dans les tuyaux, les filtres attendent tout simplement. Il est possible d’introduire de la récursion et des compositions. **Désavantages :** Il est possible d’avoir des cas de dépassement de capacité ou de blocage (deadlock), si un filtre doit attendre pour toutes les données. Les tuyaux acceptent souvent seulement des type de données simples. Par conséquent, les filtres doivent faire plus de travail pour pousser et extraire les données dans les tuyaux. Si on implémente des tuyaux pour différents types, il n’est plus possible de connecter n’importe quel tuyau à n’importe quel filtre.

2)Layered🡪La division d’architecture en couches. La couche est une notion abstraite et elle peut inclure des groupes de méthodes, de classes, de paquets. Chaque couche capture d’une responsabilité bien définie et distincte[Présentation (IU), Application (Service), Logique (Domain), Accès aux Données (Persistance)]. Les éléments d’une couche ne peuvent utiliser que des modules de la couche précédente ou suivante. Il est possible d’avoir plusieurs couches au même niveau, mais elles ne peuvent pas communiquer. **Avantages :** La maintenance est facile et peu coûteuse. On peut exploiter l’expertise des développeurs. Les couches peuvent être réutilisées et partagées parmi plusieurs autres couches. **Désavantages :** Il est interdit (ou vraiment compliqué) d’utiliser une couche pas immédiatement adjacente. L’architecture est sensible à la conception des interfaces.

3)MVC🡪Couche Modèle : responsable des données. Couche Vue : responsable de la présentation. Couche Contrôleur : responsable de la logique. Un seul modèle peut être présenté par plusieurs vues[Le modèle est responsable de connaître toutes les vues et de les avertir lors de changements. (patron Observer)]. Le modèle ne doit pas faire partie du système. Un lien à une base de données et la fonctionnalité pour accéder à la base sont suffisants. Le contrôleur réagit aux évènements dans la vue et il sait comment changer le modèle. Le contrôleur accepte une action par l’utilisateur. **Alternatifs :** MVP🡪L’utilisateur interagit avec le présenter, pas la vue. Un présenter par vue. MVVM🡪ViewModel peut contrôler plusieurs vues. MVA🡪Le modèle et la vue ne sont pas connectés. PAC🡪Si le modèle est modulaire. Les composantes sont organisées dans une hiérarchie. **Avantages :** Séparation des responsabilités. Le modèle peut être partagé parmi divers vues et contrôleurs. Il y a déjà plusieurs cadriciels prêts à étendre pour implémenter le MV. **Désavantages :** L’architecture peut augmenter la complexité du système. Il y a la possibilité de plusieurs mises à jours des composantes, pas vraiment nécessaires. Les diverses alternatives et définitions peuvent créer de la confusion.

4)Blackboard🡪Le tableau représente un grand problème qui peut être divisé en petites parties. Il fournit la structure de données partagée. La structure peut être complexe avec divers représentations, des données décomposées et organisées en différents niveaux. Chaque source de connaissance ajoute à la solution du problème.Le professeur est le planificateur qui décide quelle expertise est nécessaire selon le travail qui est déjà complété. Il réagit aux changements sur le tableau pour mettre les décisions. **Avantages :** Il peut fonctionner avec des problèmes grands, mais décomposables. Il peut accommoder plusieurs « expertises ». Il est facile d’ajouter de nouvelles sources de connaissance et d’étendre les structures de données. **Désavantages :** Le planificateur peut être lourd et complexe. La synchronisation et le contrôle d’accès sont nécessaire. Il est difficile de modifier les structures de données

5)Event-bus🡪Si on a plusieurs modules qui veulent communiquer entre eux, le bus d’évènements peut gérer la communication. Les modules peuvent soumettre des messages directs, diffuser des messages ou attendre des messages d’un type spécifique (Publish-Subscribe). Les messages ont une en-tête, qui peut spécifier le receveur et le type du message, et un corps. Ex:systèmes de notification, monitoring, interfaces graphiques. Un peu comme le patron Observer. **Avantages :** On peut ajouter ou supprimer des modules en cours d’exécution. Les modules ne sont pas responsables de la livraison des messages. **Désavantages :** Problèmes d’évolutivité : Il y a un risque de dépassement de capacité, s’il y a beaucoup de messages pour un seul bus. La synchronisation est aussi une considération critique.

\*\*Architectures Distribuées : Ce sont aussi des architectures modulaires, mais dans ce cas, les modules sont distribués parmi plusieurs fournisseurs et situés à différents endroits. L’importance du réseau est critique. Le développement est partagé parmi divers organisations, ce qui a un impact sur les aspect économiques liés au logiciel. Le logiciel devient plus flexible et dynamique.**Avantages :**Non-transparence :Plusieurs détails sont cachés, ce qui facilite le développement. Partage des ressources (matériel et logiciel). Ouverture : Flexibilité accrue pour utiliser divers fournisseurs de matériel et de logiciel. Traitement concurrent pour améliorer la performance.Évolutivité:Capacité augmentée en ajoutant de nouvelles ressources. Tolérance aux fautes : Capacité de continuer à fonctionner après une la détection et la résolution d’une faute. **Désavantages :** Complexité augmentée. Sécurité diminuée. L’effort de gestion est augmenté.

1) Client-Serveur : Un serveur, qui fournit la fonctionnalité (un service) et le client, qui est un consommateur du service. Le client fait des demandes et le server répond. Le server peut servir plusieurs clients. La communication se passe par un réseau. Client-léger: juste le rendu de l’interface graphique. Aucune logique. Terminal. Client-lourd: Le rendu de l’interface graphique et quelque partie de la logique. On ne peut pas parler d’avantages ou de désavantages parce que l’architecture est vraiment générique.

2) Multitier : Au lieu de deux niveaux (client, serveur), on peut en avoir plusieurs. Présentation (interface), Application (plusieurs niveaux), Base des données. Les niveaux peuvent communiquer indépendamment entre eux. Les niveaux peuvent être distribués sur plusieurs ordinateurs ou même entre plusieurs propriétaires. Les modules sont développés complètement indépendamment et ils sont très spécialisés.Ex : Applications de commerce électronique, analyse de données, apprentissage automatique.Uber utilise module maps de google (avec sa prope bd), base de donnée commandes. **Avantages :** Les technologies existent depuis longtemps et elles sont robustes. Les modules sont réutilisables. Ils ne sont plus autonomes, mais vraiment indépendants. L’échange dynamique des modules est possible et facile. **Désavantages :** Les modules peuvent être spécifiques à des technologies particulières et cela diminue la flexibilité. La fonctionnalité et la disponibilité dépendent du réseau. La sécurité devient complexe et réduite.

3) Master-Slave : On a un grand travail, qui peut être divisé et partagé. Le « maître » est responsable de diviser le travail et de distribuer les parties aux esclaves. Les « esclaves » exécute le travail sur des parties des données. En fait, les esclaves ont des programmes identiques. **Avantages :** La parallélisation augmente l’efficacité. La présence du maître augmente la fiabilité du système. Le résultat est garanti. **Désavantages :** L’architecture est sensible à la fiabilité du maître. Les esclaves sont isolés. Il y a un risque de redondance. L’architecture n’est pas utilisable sans une capacité suffisante de calcul. N’est pas applicable à des problèmes qui ne sont pas décomposables.

4) Peer-to-peer : Les données ne sont pas centralisées, mais tout le monde peut avoir toutes les données. Les données sont partagées par des connections directes entre les participants du réseau. Une structure centrale est possible, mais juste pour les adresses des participants ou pour conserver l’information sur qui possède quoi. **Avantages :** Le système est robuste aux échecs des participants. L’architecture offre une évolutivité augmentée. **Désavantages :** La qualité, la performance et la sécurité ne sont pas garanties. Ils peuvent être garanties avec des protocoles et outils ajoutés comme le fait les bloc chains.

5) Broker : On sait tout ce qu’on veut faire, mais on ne sait pas qui peut le faire et comment on peut les contacter. Le « courtier » est un service de découverte qui connait tous les fournisseurs et leurs services, dont la description est publiée. Il est nécessaire pour les serveurs et les clients de soumettre les fonctionnalités de leurs services et les exigences respectivement dans un format spécifique. C’est le prédécesseur de l’architecture orientée service. **Avantages :** Il est possible de dynamiquement ajouter, modifier ou supprimer des serveurs. La distribution est transparente pour le développeur. Le serveur et le client sont découplés. **Désavantages :** Les transactions et les exceptions doivent être gérées.

6) SOA : Tout est un service! Les services sont des logiciels autonomes et atomiques. Ils rendent possible l’exposition de l’expertise (fonctionnalité) et des données. Le protocole est accompagné de plusieurs standards. Pour la sécurité, la composition, le web sémantique. Le service est publié en tant qu’une interface XML. L’interface spécifie la fonctionnalité et les types des données disponibles. L’entente au niveau du service (SLA) spécifie les propriétés non-fonctionnelles. **Avantages :** La technologie promeut la composition, la réutilisation, l’interopérabilité. Les clients ne dépendent pas d’une technologie spécifique. Il y a une abondance d’options et c’est possible d’échanger les services dynamiquement. **Désavantages :** SOAP est très lourd. Plusieurs technologies n’ont jamais marché.

7) REST : Tout est une ressource! En théorie, les services REST exposent juste des données et des structures des données (identifiées uniquement) et permettent des opérations simples sur ces structures (GET, PUT, POST, DELETE). La technologie utilise plusieurs formats de données (texte, XML, JSON). Une interface explicite n’est pas nécessaire. Une communication vers HTTP est requise.

SOAP vs. REST : SOAP🡪Standardisé. Langage, plateforme, communication indépendant. Support des outils augmenté. Gestion des fautes et des exceptions fournie. REST🡪Plus facile. Plus simple. Plus performant.

**Cadriciels et Systèmes Distrib** : Cadriciels🡪 architectures à moitié implémentées et génériques. fournissent un squelette avec une fonctionnalité de base prévue pour être étendue. il y a des modules abstraits (hot spots), que les développeurs doivent étendre, et des modules concrets (frozen spots), qu’il ne faut pas changer. cadriciels implémentent le principe OCP par définition. patron Template Method. Les méthodes abstraites appelées par la méthode template constituent la partie qui sera étendue.**Propriétés :** Inversion de contrôle-Les modules concrets du cadriciel dictent le flux de contrôle et non les modules étendus par le programme. L’extensibilité et la réutilisabilité sont augmentées, mais la modifiabilité est réduite. **Plugiciels** : suivent le type d’architecture microkernel. **Propriétés :** Le système de base est atomique et indépendant. Ça veut dire que le système peut fonctionner sans plug-in. Les plug-ins ajoutent des fonctionnalités supplémentaires. les plug-ins améliorent l’expérience de l’utilisateur plutôt que celle du développeur. SRP est le principe le plus pertinent pour les plugiciels(La fonctionnalité est séparée et la logique est isolée). La flexibilité et l’extensibilité sont augmentées, mais la maintenabilité peut souffrir. **Écosystèmes** : une collection de modules qui ont quelque chose en commun, p.ex. un langage, un cadriciel. un dépôt de modules logiciels partagé par une communauté de développeurs. Les modules, ou les « paquets », contribuent des fonctionnalités. Les développeurs soumettent des paquets dans un dépôt. **Propritétés :** Le modèle principal pour les systèmes à code source ouvert. Les contributions sont gérées par la communauté, qui spécifie les règles et les habitudes de développement et de contribution. Les fonctionnalités sont séparées et isolées. La modularité, la testabilité et la réutilisabilité sont augmentées. La maintenabilité souffre comme dans le cas des plugiciels.

**Qualité de conceptionSOA=Architecture** orienté service(comprendre, produire, outils qui existent déjà, changements, projet innovant. 🡪Pertinence, Maintenabilité, Compatibilité) **MDA = Architecture** orienté modèle(adaptatif et facile à changer, interface simple, éviter erreurs, performance, blackboard et pipefilter, pls étapes pour identifier les mots le syntaxe🡪Utilisabilité, Maintenabilité, Fiabilité). **CBA = Architecture des composantes** (financier, exigences bien définies, système disponible 100% du temps, données doivent rester privées, maitenir intégrité. 🡪Conformité aux exigences, la sécurité et la fiabilité). **EDA = Architecture évènementielle**(notifier, en temps réel, intégré 🡪Efficacité,Portabilité,Compatibilité)**AOA= Architecture orienté aspect**(autonome, en tout temps, pas faire d'erreurs, plus haute priorité, système fonctionne avec n'importe quelle voiture🡪Efficacité, Fiabilité, Sécurité, Portabilité).

**Mauvaise Conception**:

Antipatrons d’architecture: **1)Architecture by implication:** Les architectes ont beaucoup d’expérience dans le développement de systèmes semblables et ils ne préparent pas de spécifications architecturales suffisantes: Réutilisation inappropriée d’une architecture existante/connue. Manque de planification et de spécification de l’architecture. Ignorances de nouvelles technologies.Risques cachés qui émergent en cours de projet: manque de connaissance du domaine, des technologies et complexité inattendue. **Causes** : Absence de gestion de risque. Excès de confiance des gestionnaires, architectes et développeurs. Dépendance envers des expériences antérieures qui différent sous certains aspects critiques. **Refactoring** : Modulariser la spécification de l’architecture en vues et points de vue. Chaque vue représente une partie prenante pertinente (architecte, développeur, client etc.). Exprimer l’architecture à l’aide d’objectifs et de questions (GQA): objectifs, questions, vues, analyse des vues, intégration des vues, correspondance entre les vues et les exigences, raffinement des vues, communication de l’architecture, validation de l’implémentation. **Exceptions** : Si les différences entre 2 projets sont petites ou localisées. Pour un projet exploratoire. 2) **Autogenerated Stovepipe** : Le problème se produit lorsque nous migrons un système vers une version distribuée. Les interfaces du système original peuvent être inappropriées pour le système distribué. Les opérations locales font souvent diverses hypothèses sur l'emplacement, p.ex. l’accès au système de fichiers local. **Refactoring** : Définir les interfaces distribuées à nouveau. L’interopérabilité et la stabilité doivent être une haute priorité. **3) Cover your Assets** : La documentation est trop de détaillée et tente de couvrir toutes les options et les exceptions. Elle devient très compliquée. **Refactoring** : Utiliser des abstractions, comme des diagrammes et des tableaux, pour communiquer l’architecture de façon plus efficace. **4) Design by Committee** : Tous les membres de l’équipe peuvent influencer les décisions d’architecture et de conception. Le résultat est de la documentation très complexe avec tous les caractéristiques possibles. **Symptômes et conséquences :** Des réunions fréquentes, mais pas organisées et sans objectif. Beaucoup de conflits parmi les architectes et les développeurs. Beaucoup d’efforts requis pour interpréter et développer les spécifications. **Causes** : Les rôles ne sont pas bien définis. Un effort pour satisfaire tout le monde. Réunions longues et mal organisées. **Refactoring** : Définir des rôles clairs pour les membres de l’équipe. Désigner un architecte principal pour le projet qui va prendre les décisions finales. Prioriser les exigences. Organiser les réunions et diminuer leur durée. Possiblement diviser les équipes pour mieux organiser les réunions (DevOps). **Exceptions** : Lorsque les équipes sont petites.**5) Jumble** : orsque les éléments horizontaux se mélangent avec les éléments verticaux. Les éléments horizontaux sont les couches d’une architecture générale. Les éléments verticaux sont les tiers de l’architecture spécifiques à une application. (comme une violation de DIP). **Symptômes et conséquences**: Réutilisabilité et stabilité réduites. **Causes** : Manque de communication entre les développeurs et les architectes. **Refactoring** : Traiter les éléments horizontaux en tant qu’abstractions et les éléments verticaux en tant qu’implémentations. Ajouter des éléments verticaux pour la fonctionnalité spécialisée ou pour la performance. **6) Reinvent the Wheel** : Lorsqu’un système est conçu à nouveau. Les architectures et les exemples existants ne sont pas pris en considération. **Symptômes et conséquences** : La réutilisabilité et l’interopérabilité sont diminuées. L’effort est augmenté. Des architectures fermées, pertinentes uniquement pour un système spécifique (comme une violation d’OCP). Réplication inutile des conceptions existantes. **Causes** : le système doit être conçu à nouveau. Nous sommes les premiers qui font quelque chose comme ça. Connaissance insuffisante d’autres projets. **Refactoring** : Faire le minage des architectures pour identifier les exemples pertinents. Mettre en place un processus logiciel en spirale, agile et itératif. Raffiner les exigences et les décisions de conception, et identifier des prototypes d’architectures dans d’autres systèmes. **Exceptions** : Lorsque la communication est difficile et que l’on veut minimiser les frais de la coordination. 7) Spaghetti Code : Le logiciel manque de structure et de cohérence. **Symptômes et conséquences** : La compréhensibilité et la réutilisabilité sont diminuées. L’effort pour la maintenance est augmenté. **Causes** : Manque d’expérience. **Refactoring** : Séparation des responsabilités, encapsulation. Utilisation des patrons et de bonnes pratiques de conception. **Exceptions** : Du code spaghetti est tolérable lorsqu’il se produit dans l’implémentation et pas dans les interfaces publiques.**8) Stovepipe System** : L’antipatron se produit lorsqu’on veut intégrer des sous-systèmes. Chaque intégration est spéciale et il manque un plan d’intégration générale pour le système entier. **Symptômes et conséquences** : La maintenabilité et l’interopérabilité sont diminuées. L’implémentation ne correspond pas à la documentation. La portabilité est réduite. **Causes** : Manque d’abstraction. L’interface de chaque sous-système est unique. Fort couplage. **Refactoring** : Définition d’une interface commune (SOA). **9) Swiss Army Knife** : Une interface très complexe qui expose beaucoup de fonctionnalités. (une violation de SRP et d’ISP). **Refactoring** : Extraire des méthodes de l’interface ou de la classe. Utiliser les patrons façade ou adapter. Documenter l’usage et l’implémentation de l’interface. **10)Vendor Lock-in** : Lorsque le système dépend directement de logiciels ou de matériel propriétaires. **Symptômes et conséquences** : Les mises à niveau des produits commerciaux pilotent le cycle de maintenance des logiciels d'application. Des échecs dans les applications se produisent à cause des délais induits par les produits commerciaux. Des violations de DIP et d’OCP. Portabilité est réduite. **Causes** : Il n’existe pas un standard universel. Le produit commercial ne se conforme pas aux standards ouverts. **Refactoring** : Architecture multi-niveaux. Utilisation d’abstractions.

**Antipatrons de conception** : **1)Missing abstraction** : Lorsque des groupes de données se produisent ensemble souvent. ex. telephoneNumber, areaCode, countryCode. **Symptômes et conséquences :** Violation de SRP. Complexité augmentée. Maintenabilité et compréhensibilité réduites. **Refactoring** : Extraire de la classe. 2) Multifaceted abstraction : Une abstraction a plus d’une responsabilité. **Symptômes et conséquences** : Violation de SRP. **Refactoring** : Extraire de la classe ou de la superclasse. 3) Duplicate abstraction : Deux abstractions ont le même nom ou la même implémentation (cloning). **Causes** : Copier-coller. Manque de communication. **Refactoring** : Extraire de la classe ou de la superclasse. 4) **Deficient encapsulation** : Lorsque une abstraction donne plus de permissions qu’il est nécessaire, p.ex. lorsque tous les attributs sont déclarés comme publiques. **Symptômes et conséquences** : Maintenabilité et sécurité réduites. **Causes** : Maintenance négligente. **Refactoring** : Encapsuler les données et fournir des méthodes d’accès. **5) Unexploited encapsulation** : L’utilisation des vérifications de types explicites lorsque le polymorphisme est disponible. **Symptômes et conséquences** : Complexité augmentée. Maintenabilité et compréhensibilité réduites. **Causes** : Pensée procédurale dans le contexte orienté-objet. Échec d’application des principes orientés-objet. **Refactoring** : Remplacer les vérifications de types par du polymorphisme.**6) Broken modularization** : Des données ou des méthodes qui doivent être ensemble, par rapport à la similarité sémantique ou à l’utilisation, font parties de plusieurs abstractions. **Symptômes et conséquences** : Haut couplage, faible cohésion. Maintenabilité est réduite. Performance se détériore. **Causes** : Manque de connaissance de la conception existante. **Refactoring** : Déplacer des méthodes ou des attributs. Imbriquer des classes.**7) Insufficient modularization** : Une abstraction a beaucoup de membres publiques ou des méthodes très complexes. **Symptômes et conséquences** : Violation de SRP et d’ISP. **Causes** : Contrôle centralisé. **Refactoring** : Extraire des méthodes ou attributs de la classe ou de l’interface. **8)Cyclically-dependent modularization** : Deux abstractions utilisent beaucoup de membres entre elles. **Symptômes et conséquences** : Haut couplage. **Causes** : Passage du pointeur « this ». L’abstraction n’est pas conçu correctement. **Refactoring** : Déplacer des méthodes ou des attributs. Imbriquer des classes. **9) Unfactored hierarchy** : Dans une hiérarchie, Il existe de la duplication entre les classes dérivées ou les classes dérivées et la classe de base. **Symptômes et conséquences** : Maintenabilité est réduite. **Causes** : Duplication de code (Cloning). Copier-coller. **Refactoring** : Extraire une superclasse. Remonter des méthodes ou des attributs.**10) Broken hierarchy**: La classe de base et les classes dérivées n’ont pas une relation « est-un ». **Symptômes et conséquences** : Violation de LSP. **Causes** : L’héritage est implémentée pour des raisons d’implémentation, pas de conception. **Refactoring** : Remplacer l’héritage par de la délégation. Inverser la relation d’héritage. **11) Cyclic hierarchy** : Une classe de base a une association avec une ou plusieurs de ses classes dérivées. **Symptômes et conséquences** : Haut couplage. Compréhensibilité réduite. **Causes** : L’héritage n’est pas conçu correctement. **Refactoring** : Extraire une classe. Déplacer des méthodes. Imbriquer des classes.

**Antipatrons de code** : **1) Duplicated Code** : Le même code existe à plusieurs endroits dans le système. **Refactoring** : Extraire des méthodes. Construire une classe générique. Extraire des classes. Enchaîner les constructeurs. **Détection** : SEC = Size of exact clone. LB = Line Bias. SDC = Size of Duplication Chain. **Détection** : SEC = Size of exact clone. LB = Line Bias. SDC = Size of Duplication Chain.**2) Long Method** : Une méthode a une longue implémentation qui fait beaucoup de choses. **Refactoring** : Extraire des méthodes. Décomposer les énoncés conditionnels. Assembler une méthode (Compose Method). **Détection** : NOAV = Number of Accessed Variables. MAXNESTING = Maximum Nesting Level. **3)Large Class** : Une classe qui a beaucoup de membres ou beaucoup de responsabilités. **Refactoring** : Extraire une/des classes/superclasses/sous-classes. **Détection** : ATFD = Access to Foreign Data. **4) Long Parameter List** : Une méthode a une longue liste des paramètres. **Refactoring** : Remplacer des paramètres par une méthode. Introduire des objets en paramètres.**5) Divergent Change** : Une classe change beaucoup, plusieurs fois, ou chaque fois qu’il y a un changement aux exigences. **Refactoring** : Extraire une/des classes. **6) Shotgun Surgery** : Lorsqu’il y a un changement, on doit changer le code à plusieurs endroits. **Refactoring** : Déplacer des méthodes. Imbriquer des classes. **Détection** : CM = Changing Methods. CC = Changing Classes. **7) Feature Envy** : Une méthode utilise plus de membres d'autres classes que de la classe à laquelle elle appartient. **Refactoring** : Déplacer des méthodes. Extraire et déplacer des attributs ou des méthodes. **Détection** : LAA = Locality of Attribute. FDP = Foreign Data Providers.8) Switch Statements : Une méthode est longue et complexe au point de contenir plusieurs instructions « switch » ou conditionnelles. **Refactoring** : Remplacer des vérifications de types par des sous-classes. Remplacer des verifications de types par un instance du patron Stratégie ou État. Remplacer des énoncés conditionnels par du polymorphisme. **9)Message Chains** : Il existe de longues chaînes d’invocations de méthodes en cascade. **Refactoring** : Extraire et déplacer des méthodes. **10) Inappropriate Intimacy** : Une classe a beaucoup d’associations avec les membres privés d’une autre classe. **Refactoring** : Déplacer des méthodes ou des attributs. Extraire une/des classe(s). Cacher les délégations. **11) Refused Bequest**: Une classe dérivée n’a pas besoin d’hériter de certains membres de la classe de base. **Refactoring** : Descendre certaines méthodes ou attributs dans les sous-classes. **Détection** : NProtM = Number of ProtectedbMember. BUR = Base-class Usage Ratio. BOvR = Base-class Overriding. AMW = Average Method Weight.

**Architectures des Megadonnées** : **1)Intergiciels orientés messages** : Dans une architecture orientée message, les modules sont vraiment découplés et interopérables. L’entité principale de l’architecture est le message. Les messages correspondent aux données qui doivent être communiquées entre les modules. Les modules se séparent entre producteurs et consommateurs de messages. **Avantages :** Flexibilité. Découplage/Interopérabilité. Fiabilité. **Désavantages :** Performance. Complexité de l’infrastructure et de l’architecture.**2) MapReduce** : Dans une implémentation MapReduce, le développeur doit définir deux tâches : Une tâche map qui représente un travail d’analyse (p.ex., un tri, une addition, une cherche etc.). Une tâche reduce qui représente un travail de sommation, qui va agréger les résultats des tâches map individuelles. 3) **Théorème CAP :** En faveur de la disponibilité et de la performance, les systèmes distribués sacrifient souvent la cohérence. Consistency : Tous les clients voient la même vue des données, même après une mise jour. Availability : Tous les clients peuvent obtenir une copie des données. Partition Tolerance : Le système continue à fonctionner, même après l’échec d’un nœud. La tolérance au partitionnement est la propriété de base des systèmes distribués, alors elle est garanties par définition. Alors on doit prendre une décision entre la cohérence et la disponibilité. **4) ACID vs BASE** : ACID🡪 **Atomique** : Toutes les opérations d'une transaction réussissent ou l’ensemble de la transaction est annulée. **Cohérente** : À la fin d'une transaction, la base de données est structurellement correcte. **Isolée** : Les transactions ne se font pas concurrence. Les accès litigieux aux données sont arbitrées par la base de données, de sorte que les transactions semblent s'exécuter de manière séquentielle. **Durable** : Les résultats de l'application d'une transaction sont permanents, même en cas d'échec. BASE🡪 **Basic Availability** : La base de données semble fonctionner la plupart du temps. **Soft-state** : Les bases n'ont pas besoin d'être cohérentes en écriture, pas plus que les différentes répliques doivent être cohérentes les unes avec les autres. **Eventual Consistency** : Les bases présentent une cohérence à un moment ultérieur (par exemple, paresseusement au moment de la lecture). **Implémentations des bases distribués** : NoSQL🡪 Elles n’exposent ni interfaces SQL ni interfaces normalisées(Cela augmente l’efficacité et la flexibilité dans la définition des requêtes). Elles utilisent aussi plusieurs types d’organisation de données(Cela augmente beaucoup l’efficacité du système). Types d’organisation des données : **1)Key-value**🡪 Les données sont organisées en pairs de clés et de valeurs. Les valeurs suivent les notions d’objets, comme dans le cas de la programmation orientée objet(Elles peuvent être assez complexes). Alors on peut avoir des attributs différents ou des attributs manquants entre les objets du même type(Cela offre une flexibilité augmentée). Les données peuvent être partitionnées et partagées parmi les nœuds en utilisant les clés. **Avantages :** Ces systèmes sont très efficaces pour des applications simples. Il est possible de stocker toutes les données en mémoire, ce qui augment l’efficacité. Ils peuvent supporter des données complexes en offrant une efficacité dans la cherche des valeurs basée sur les clés. **2)Document** 🡪 Ces systèmes traitent les données en tant que fichiers. On peut demander des documents à l’aide de clés. Les documents peuvent être de divers tailles et complexité. **Avantages :** Ces systèmes sont appropriés pour les applications où les données sont des documents (bien sûr!), mais aussi lorsqu’on a des données ayant une structure variable. Contrairement au KV, où les valeurs sont opaques, ici on peut demander des documents en fonction de leurs attributs. On peut aussi définir des métadonnées pour faciliter la récupération des fichiers. **3)Graph**🡪 Dans ces systèmes, les données sont représentées en tant que graphes. Ils répondent au besoin de combiner une modélisation graphique des données et les tableaux relationnels de RDB. Les nœuds représentent des entités et les arêtes des relations. Les nœuds peuvent avoir des attributs. Ils permettent de stocker n’importe quelle relation de façon pratique et efficace. **Avantages :** Parfois, on a besoin de stocker les données des graphes en préservant leur forme, en tant que graphe. Il existe plusieurs algorithmes pour traverser ou analyser les graphes qui sont déjà efficaces. Certaines requêtes ne sont pas possibles avec les langages traditionnels des requêtes. **4)Wide column**🡪 Ce sont les systèmes les plus proches de bases de données relationnelles, au moins au niveau conceptuel. Ils conservent les concepts de tableaux, de rangées et de colonnes. Cependant, les concepts relationnels forment une interface pour les clients(Au niveau de l’implémentation, les systèmes sont basés sur des systèmes de fichiers distribués). La définition des colonnes est très flexibles. On groupe plusieurs attributs dans des familles de colonnes. **Avantages :** Ce type de BD facilite la migration des bases de données relationnelles vers les systèmes distribués. Ils peuvent supporter d’énormes quantités de données. Les systèmes wide-column sont conçus pour fonctionner avec les systèmes distribués d’analyse des mégadonnées tels que MapReduce et Hadoop. Ils sont conçus pour offrir : une disponibilité et une sécurité augmentées; des lectures et des écritures rapides.

**Microservice vs SOA**

**Microservices :** Une version …plus petite de SOA. Chaque service est juste une opération avec une seule responsabilité. Les microservices gardent toutes les autres propriétés de SOA : réutilisabilité, portabilité, interopérabilité. Ils sont plus faciles à composer et à remplacer. Les microservices sont déployés et gérés indépendamment. Ils promeuvent l’automatisation, l’évolution et la livraison continue. **Défis** : Transfert de charge cognitive, Le logiciel est plus simple, mais les connexions dans le réseau deviennent beaucoup plus nombreuses et complexes.

AOP : permet de modéliser des fonctionnalités qui, autrement, seraient invasives. Fonctionnalités transversales ou crosscutting concerns. AOP ne vise pas à remplacer les langages OO, mais à les compléter. Les concepts issus du domaine sont toujours modélisés avec des classes. Les fonctionnalités transversales sont modélisées avec des aspects. Réduit la confusion et l’éparpillement du code (code tangling et code scattering). **Avantages :** Le code associé à l’implantation du patron est localisé à un endroit bien identifié.( Diminue l’éparpillement de code). Les classes Noeud et Navigateur ne sont pas affectées par l’instanciation du patron(Augmente la cohésion des classes issues du domaine, Facilite l’instanciation et la « désinstanciation » du patron). Transparence à la composition de patrons (Les classes Noeud et Navigateur pourraient participer simultanément à plusieurs patrons de façon complètement transparente). On peut définir une librairie de patrons de conception réutilisable en regroupant les aspects communs à toutes les instanciations des patrons dans des aspects abstraits.

**Comment choisir BD? KV:** mise en cache, donnée séquentielles (listes, queue), Peut supporter systems publish/subscribe. **Document:** Avec données tres complexe et imbriquées. Client en JS**. Graph:** relation entre données sont tres complexe, travailer avec hiéarchie et taxonomie. **Wide-column:** donnée sans structure qui ne changent pas bcp et qui doivent être stocké lgt. Lorsqu’on a besoin d’un stockage évolutif. **Architecture traitement: Architecture Lambda:** garantit disponibilité pour donnée volatile. Analyse incrémentale sur petite quantité donnée, analyse en temps reel (streaming). Pour donnée immutable = on garantit coherence. Avec résultats des analyses en temps reel on rend donnée immuables, applique analyse + complexe et couteuse sur ces donnée, favorise precision plutot que performance. Guarantit certaine donnée dispo et cohérente, guarantit performance des analyse en temps reel. Garantit precision analyses, architecture evolutive tolérante aux pannes, bon équilibre entre performance et fiabilité – codage = compliqué, modèle donnée difficile à migrer ou maintenir, repartition du traitement des donnée créer surcharge de performance **Architecture.Kappa:** garde juste niveau streaming, système orienté message pour gérer entré des donnée et faciliter retraitement. Flexibilité augmenté par rapport a maintenance et évolution des algorithme d’analyse.

On peut changer algorithme et retraiter donnée avec nouvelle analyse (Canary deployment = remplace service ou algo nouvelle ebrsion en temps reel en déployent les deux version et en redirigant demandes). Flexibilité augmenté pour codage, simplicité augmenté pour modèle des données, facilite développement des système apprentissage en ligne qui n’exige pas stockage persistent, utilization mémoire pour efficacité augmenté. – Tolérance pannes et capacité récupérer réduite. **Lambda vs Kappa:** Lamda (demande utilisateurs doivent servis sur base ad-hoc en utilisant stockage donnée immutable, réponse rapide necessaire et système doit gérer diverses mise a jour sour forme de nouveaux flux, aucune des données stocké efface, ajout mise a jour et donnée à la BD), Kappa (événements ou requetes sur données mis en file attente pour traitement en fonction système stockage dans un système distribué. Ordre des evenement et requetes pas predetermine. Plateforme de traitement flux peuvent interagir avec BD tt le temps. Nécessite resilience et haute disponibilité, car gestion téraoctet de stockage requise por chaque noeud système pour supporter réplication) Blockchain: Consistency, Scalability, Decentralisation. **SOA Entités** • Fournisseur • Client • Réseau • **Artefacts** • Implémentation service • Interface service • Implémentation client • Service stub • Protocole de communication • Entente de niveau de service (SLA). Indépendant protocole communication, base sur XML (otuil langage disponible), possible définir plusieurs norme et specifications à nouveau. – Pas efficace, interface service et format message = long complexe, parcourir fichier = couteux. **REST: style architectural créer services web, base sur HTTP, toute resource identificateur unique, accès resource par URL, 5 opérations. Modificabilité, fiabilité, évolutivité augmenté. Manque norme et specifications formelle, fonctionnalité + complexe pas tjrs dispo, il faut organization des donnée. Microservice: un module = un composant logique serveur, une BD par module, possibilité équilibreur charge.**

**Antipatrons d’adoption microservices**: **Magic pixie dust**: «Une pincée de microservices résoudra tous nos problèmes de développement. L'adoption d'une architecture doit s'accompagner de l'adoption des principes, patrons, philosophies et cultures appropriés. Développement et opérations intégrés (DevOps)? **Microservice as the goal**: Un responsable intervient et annonce une initiative visant à transformer tous les systèmes en microservices. «Faites des microservices!» . Les développeurs sont-ils préparés et capables de ce changement? Adoptez les tests et le déploiement automatisés, éliminez le travail inutile, améliorez le monitorage **Scattershot adoption**: consequence du patron precedent. la direction a annoncé l'adoption de microservices, mais elle n'a jamais spécifié de plan ou de stratégie sur la façon de le faire. chaque équipe de l'organisation a adopté sa propre stratégie et il y a un manque général de coordination. **Trying to fly before you can walk**: Essentiellement, adoptez une architecture avancée, comme des microservices, avant d'adopter d'abord les bonnes pratiques de développement logiciel et les principes de conception logicielle. Refactorisez le monolithe avant de migrer. **Focusing on Technology**: Ce n'est pas parce que vous adoptez une technologie spéciale que la migration (ou l'architecture de microservice elle-même) fonctionnera. L’emphase sur la technologie peut rapidement conduire à un vendor lockin **More the merrier:** Microservice n'implique pas littéralement petit. Des services plus petits impliquent beaucoup plus de services. **Red Flag Law**: lorsqu'une organisation continue d'utiliser des pratiques incompatibles avec les microservices.