**Les principes SOLID**

**S**: Single Responsibility principle: Une classe doit avoir une seule responsabilité

**Quelques principes GRASP**

Haute cohésion: Les méthodes d’une classe utilisent d’autres membres de la classe elle-même Faible couplage: Les méthodes d’une classe utilisent peu de membres d’autres classes Haute cohésion + Faible couplage =­> Moins de responsabilité, moins de raisons de changer

**O**: Open closed Principle: Ouvert à l’extension, fermé à la modification Solution: Polymorphisme

**L**: Liskov substitution principle

Les fonctions qui utilisent des références à des classes de base doivent pouvoir utiliser des objets de classes dérivées sans le savoir/ Les postconditions ne peuvent pas être renforcées dans le type dérivé/ Les postconditions ne peuvent pas être plus faibles dans le type dérivé.

**I**: Interface segregation principle: De nombreuses interfaces spécifiques aux clients valent mieux qu’une seule interface

**D**: Dependency inversion Principle: Dépend des abstractions. Ne dépend pas de classes concrètes.

**Styles d’architecture:**

1- Monolithique 2- Modulaire: Pipe-filter, Layered, MVC, Blackboard, Event-bus 3- Distribué: Client-server, Multitier, Master-slave, peer-to-peer, Broker, SOA, REST

**Cadriciels**: Architectures à moitié implémentées et génériques.Dans un cadriciel il y a des modules abstrait (hot spots) que les développeurs doivent étendre et des modules concrets (frozen spots) qu’il ne faut pas changer OCP. Propriétés: Inversion de contrôle: Les modules concrets du cadriciel dictent le flux de contrôle et non les modules étendus par le programme. (DIP)// L’extensibilité et la réutilisabilité sont augmentées, mais la modifiabilité est réduite. AngularDjango

**Plugiciels** (plugins): propriétés Le système de base est atomique et independent/ Les plug-ins ajoutent des fonctionnalités supplémentaires Contrairement aux cadriciels, les plug-ins améliorent l’expérience de l’utilisateur plutôt que celle du développeur.Eclipse

SRP est le principe le plus pertinent pour les plug-ins

**Écosystèmes logiciels**: Collection de modules qui ont quelque chose en commun: NodeJS, Android(les 3)

**Changement de rupture:** À cause des dépendances entre des modules qui sont développés indépendamment, des changements peuvent se propager à travers l’écosystème.

**Négociations de l’impact des changements de rupture**: La stratégie des développeurs par rapport à qui sera responsable de s’adapter aux changements.

Éclipse: Peu de changements de rupture

R/CRAN: Les paquets dans CRAN se synchronisent chaque mois. Changements de rupture en nombre moyen

Node.js: C’est la responsabilité de l’utilisateur de mettre à jour son paquet basé sur les changements d’autres paquets. Beaucoup de changements de rupture.

**Comment casser l’écosystème?**

Eclipse: Pas possible R/CRAN: Communiquer avec les développeurs affectés Node.js: Incrémenter le numéro majeur de la version.

**Les métriques**

Taille: Ligne de code (LOC) Complexité de l’interface: Nombre d’attributs et de méthodes (SIZE2) et Nombre de méthodes locales (NOM)Complexité structurelle: Complexité cyclomatique de McCabe (CC), Nombre pondéré de méthode (WMC) et Réponse pour une classe (RFC)Héritage: Profondeur de l’arbre d’héritage (DIT), Nombre d’enfants (NOC)Couplage: Couplage afférent (Ca), couplage entre les objets (CBO)Cohésion: Manque de cohésion entre les méthodes (LCOM), cohésion de classe serrée (TCC)

Documentation: Manque de documentation (LOD)

Métriques augmentées: Qualité diminué? Qualité amélioré

LOC, SIZE2, NOM: Compréhensibilité, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d’apprentissage, Stabilité, Remplaçabilité? Attraction, Interopérabilité, sécurité, Tolérance aux pannes et récupérabilité CC, WMC, RFC: Compréhensibilité, Opérabilité, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d’apprentissage, Stabilité, Remplaçabilité? Attraction, Interopérabilité, Sécurité, Tolérance aux pannes, Récupérabilité DIT: Compréhensibilité, Opérabilité, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d’apprentissage, Stabilité, Remplaçabilité? Attraction, Interopérabilité, Sécurité, Tolérance aux pannes, Récupérabilité NOC: Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Stabilité, Remplaçabilité? Compréhensibilité,pérabilité, Attraction, Facilité d’apprentissage CBO: Compréhensibilité, Opérabilité, Attraction, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d’apprentissage? Interopérabilité Ca: Stabilité, Remplaçabilité, Interopérabilité, Sécurité, Tolérance aux pannes? Opérabilité, Attraction, Facilité d’apprentissage

LCOM, TCC: /? Maturité, Compréhensibilité, Attraction, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d’apprentissage, Stabilité, Remplaçabilité Documentation: Compréhensibilité, Opérabilité, Attraction, Maturité, Analysabilité, Possibilité de modification, Testabilité, Adaptabilité, Facilité d’apprentissage, Stabilité, Remplaçabilité? /

Sélection du type d’architecture: voir annexe

**Antipatrons d’architecture:** Architecture by implication, (Réutilisation inappropriée d’architecture) ; Autogenerated Stovepipe (migration vers une version distribuée), Cover your assets (Documentation trop détaillée et exhaustive), Design by committee, Jumble (Lorsque les éléments horizontaux sont mélangés avec les éléments verticaux), Reinvent the wheel , Spaghetti code, Stovepipe system ( L’antipatron se produit lorsqu’on veut intégrer des sous-systèmes), Swiss Army knife (Une interface très complexe qui expose beaucoup de fonctionnalité: violation de SRP et DIP), Vendor Lock-in (Dépendance directe de logiciels ou de matériel propriétaires)

**Antipatrons de conception:**Missing abstraction (Lorsque des groupes de données se produisent ensemble souvent); Multifaceted abstraction (Une abstraction a plus d’une responsabilité); Duplicate abstraction (Deux abstractions ont le même nom ou la même implémentation); Deficient encapsulation (Lorsque une abstraction donne plus de permissions qu’il est nécessaire); Unexploited encapsulation (L’utilisation des vérifications de types explicites lorsque le polymorphisme est disponible); 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); Insufficient modularization (Une abstraction a beaucoup de membres publiques ou des méthodes très complexe); Cyclically dependent modularization (Deux abstractions utilisent beaucoup de membres entre elles); Unfactored hierarchy (Dans une hiérarchie, il existe de la duplication entre les classes dérivées ou les classes dérivées et la classe de base); Broken hierarchy (La classe de base et les classes dérivées n’ont pas une relation est un); Cyclic hierarchy (Une classe de base a une association avec une ou plusieurs de ses classes dérivées)**Antipatrons de code:**Duplicated code (Le même code existe à plusieurs endroits dans le système); Long Method (Une méthode a une longue implémentation qui fait beaucoup de choses); Large class (Une classe qui a beaucoup de membres ou beaucoup de responsabilité); Long parameter List (Une méthode a une longue liste de paramètres); Divergent change (Une classe change beaucoup, plusieurs fois, ou chaque fois qu’il y a un changement aux exigences); Shotgun surgery (Lorsqu’il y a un changement, on doit changer le code à plusieurs endroits); Feature envy (Une méthode utilise plus de membres d’autres classes que de la classe à laquelle elle appartient); Switch statement (Une méthode est longue et complexe au point de contenir plusieurs instructions <Switch> ou conditionnelles; Message chains (Il existe de longues chaînes d’invocations de méthodes en cascade); Inappropriate intimacy (Une classe a beaucoup d’associations avec les membres privés d’une autre classe)

Refused Bequest (Une classe dérivée n’a pas besoin d’hériter de certains membres de la classe de base)

**Architecture des mégadonnées (Entrée et analyse des données)** Advanced Message Queuing protocol (AMQP)

Les entités selon l’AMQP: Courtier (Permet l’asynchronicité, achemine les messages entre producteurs et consommateurs, peut transformer les messages, responsable de garantir la sécurité), Producteur, Consommateur, Échange: Algorithmes d’échanges:

Direct: Achemine le message à une clé spécifiée par une clé

Fanout: Broadcast du message; Thème: Le message est acheminé selon la clé et un paramètre supplémentaire pour permettre la multidiffusion; En-tête: achemine le message à une queue spécifié par plusieurs paramètres qu’une clé

Propriétés de l’échange: Nom, durabilité, suppression automatique et arguments supplémentaires – Queue - Message: Attributs: Type et encodage du contenu, Clé d’acheminement, Persistent ou non, Priorité, ID du producteur, Horodatage et temps d’expiration du message

RabbitMQ: Implémentation directe d’AMQP

Kafka: Service distribué de streaming, transmission selon des thèmes, toutes les données sont persistants sur disque

Kafka Zookeeper: ZooKeeper est un service de découverte qui peut gérer le groupe

RabbitMQ // Kafka

Courtier intelligent/ Consommateurs simples // Courtier simple/ Consommateurs intelligents Fiabilité et découplage sont les priorités // Streaming permet l’utilisation d’une seule connexion avec un taux d’entrée augmentée

Plusieurs types d'échange sont supportés // Seul l’échange par thème est supporté Transmission directe ou multidiffusion // Transmission par diffusion

Sécurité est augmentee // Sécurité peut être un problème

La haute performance est possible mais avec des ressources significatives // La performance est augmentée

**Avantages et désavantages des intergiciels orientés messages: A: Flexibilité Découplage/Interopérabilité Fiabilité Sécurité Évolutivité D: Performance Complexité de l’infrastructure et de l’architecture Découplage n’est pas toujours désiré**

**MapReduce: Avantages et désavantages**

**A: Efficacité grâce au parallélisme Disponibilité Tolérance aux pannes Mobilité minimale des données**

**D: Region hotspotting Ce ne sont pas tous les types de traitement qui peuvent être implémentés par MapReduce**

Hadoop et Spark: implementation directe de MapReduce

Hadoop:Le cluster représente l’infrastructure physique ou virtuelle pour le traitement des données YARN: Le système de gestion du cluster HDFS: Système de fichiers distribué S3: Le stockage physique où le HDFS existe

Spark: Les RDDs expliquent la puissance de Spark, Spark est plus efficace que Hadoop (100x plus rapide sans disque, 10x olus rapide avec disque)

**Hadoop vs Spark**

Performance O//+ utilisabilité/portabilité + // ++ Frais O // -/+ Compatibilité + Traitement de données - // + Tolérance aux pannes + // ++ Évolutivité ++ // + Sécurité + // o

Streaming Analytics:

Spark streaming vs kafka streaming

La définition des flux et l’entrée des données est basée sur le temps en spark et sur les évènements en kafka

Spark est une fausse analyse stream, car en réalité il analyse les données en petits lots; Kafka est plus efficace et meilleur pour réagir aux évènements; Spark est plus stable et potable, et il est compatible avec toutes les technologies hadoop

**Architectures des mégadonnées (Base de données)**

Le théorème CAP: En faveur de la disponibilité et de la performance, les systèmes distribués sacrifient souvent la cohérence

**C**onsistency: Tous les clients voient la même vue des données **A**vailability: Tous les clients peuvent obtenir une copie des données **P**artition tolerance: Le système continue à fonctionner même après l’échec d’un noeud

ACID vs BASE: ACID: Atomique, cohérent, Durable

BASE: Basic-Availability, Soft-state, Eventual Consistency

**Implémentation de bases distribuées:**

Key-Value: Systèmes très efficace pour des applications simples, Possibilité de stockage en mémoire (Efficacité augmentée) et supporte des données complexes

Implémentations: BerkeleyDB, Redis, Memcached, Riak et Voldemort; Document: Valeurs opaques, mais demande des documents en fonction de leurs attributs; Implémentations: CouchDB, MongoDB, RavenDB; Graph: La puissance et la capacité du système dépendent de l’implémentation sous-jacente Implémentations: Neo4j, OrientDB, Titan

Wide Column: Plus proche de RDB: Facilite la migration des bases de données rationnelles vers les systèmes distribués, Fonctionne avec les systèmes distribués d’analyses des mégadonnées tels que MapReduce et Hadoop

Implémentations: HBase, Accumulo, Cassandra

**Sélection de la solution NoSQL**

key-Value: Mise en cache, Lorsqu’on a des données séquentielles, Pour supporter des systèmes publish/subscribe, pour maintenir un état

Document:Avec des données très complexe et imbriquées Lorsqu’on a des clients en JavaScript

Graph: Lorsque les relations entre les données sont très complexes, Pour travailler avec des hiérarchies et des taxonomies

Wide-Column: Lorsqu’on a des données sans structure, qui ne changent pas beaucoup et qui doivent être stockées pendant longtempsLorsqu’on a besoin d’un stockage évolutif

**Architectures de traitement:**

**Lambda**: How to beat the CAP theorem: Séparation des données volatiles et immuables. Pour les données volatiles on peut garantir la disponibilité. Pour les données immuables on peut garantir la coherence Avantages: On peut garantir que certaines des données sont cohérentes et disponibles On garantit la performance des analyses en temps reel On garantit la précision des analyses sur la plupart de données Une architecture évolutive et tolérante aux pannes Un bon équilibre entre performance et fiabilité. Désavantages:

Le codage peut être un processus assez compliqué

Le modèle des données peut être difficile à migrer ou à maintenir La répétition du traitement de données (streaming et batch) crée une surcharge de performance

Kappa: Questioning the lambda architecture: Le retraitement des données n’est pas toujours nécessaire, il peut être fait lorsque la logique des analyses change. Il est possible de prévoir et de planifier les retraitements

Avantages: Flexibilité augmentée pour le codage

Simplicité augmentée pour le modèle des données

Facilité de développement des systèmes d’apprentissage en ligne qui n’exigent pas de stockage persistent Utilisation de la mémoire pour une efficacité augmentée

Désavantages:Tolérance aux pannes et capacité de récupérer réduites

Sélection de l’architecture de traitement

Lambda: Les demandes des utilisateurs doivent être servies sur une base de données ad-hoc en utilisant le stockage de données immuable , Des réponses rapides sont nécessaires et le système doit pouvoir gérer diverses mises à jour sous la forme de nouv flux de données, Aucune des données stockés ne doivent être effacées et cela devrait permettre l’ajout de mises à jour et de nouv données à la base de données

Kappa :Plusieurs événements ou requêtes sur les données sont mises en file d’attente pour être traités en fonction d’un stockage dans un système de fichiers distribué

L’ordre des événements et des requêtes n’est pas prédéterminé. Les plateformes de traitement de flux peuvent interagir avec la base de données à tout moment

Nécessité de résilience et de haute disponibilité car la gestion de téraoctets de stockage est requise pour chaque noeud du système afin de supporter la replication

**Blockchain**: An immutable (permanent and unalterable), consensus-driven (trust verification), decentralized (network copies) and transparent public record of data using a P2P network on multi-clouds

Database vs Blockchain: Centralized, Permissioned and requires admin/Decentralized, permissionless and no admin

The Blockchain trilemma: Scalability, Consistency and decentralisation (Pick two): sacrifice scalability

What makes Blockchain tamper-proof? Consensus protocol: There is a unanimous acceptance between participants of the network for every change or new submission; Cryptographic fingerprint: Called hash, it takes a lot of computing power and energy to generate initially.

Merkle Tree enables users to authenticate for themselves when a transaction has been inserted to the network blocks

In order to authenticate and verify the transaction A is added to the block, the user only need few hashes including merkle root Byzantine Generals Problem: (3f+1) is the number of generals where f represents the traitors

Blockchain Consensus protocol: Proof of work: Incentives, Randomness, The nodes doesn,t have to be persistent

Key Characteristics of blockchain: Permissionless, Trustless, Transparent, Immutable and secure (Unhackable)

Smart Contract: Smart contracts define the rules and penalties around an agreement in the same way that a traditional contract does, but automatically enforce the obligations A smart contract is a piece of code and data that is deployed to a blockchain it can perform calculations, store information, and automatically send funds to other accounts

The characteristics of smart-contracts: Deterministic Immutable Verifiable, they need to be triggered, it can decide: oh it the first day of the month I need to do this

**Défis de l’adoption d’une architecture évènementielle:**

Service access and configuration: Wrapper façade, component configurator, interceptor, Extension interface

Synchronisation: Scoped locking, strategized locking, Thread-safe interface, Double-checked locking optimization

Concurrency patterns: Active objects, Half-sync/ Half-async, Monitor object, leader/followers, Thread specific storage

Event handling patterns: Reactor, Proactor, Asynchronous completion Token, Acceptor-connector **Wrapper façade**: Encapsuler les fonctionnalités de bas niveau dans une interface de haut niveau.Conséquences : + Des interfaces orientées-objets de haut niveau concises, cohésives et fiable + Portabilité, maintenabilité modularité et réutilisabilité augmentées - Perte de fonctionnalités - Performance diminuée - Limitations imposées par les langages et les compilateurs **Component Configurator**: lorsqu’on veut qu’une composante change son implémentation dynamiquement, mais en évitant de charger toutes les configurations possibles en mémoire au démarrage de l’application.Conséquences : + Uniformité de la configuration et contrôle de l’interface. + Administration centralisée + Modularité, testabilité et ré-utilisabilité augmentées + Configuration dynamique + Optimisation augmenté grâce à plusieurs possibilités de configurations - Accroissement des incertitudes à cause de toutes;es interactions possibles entre les différentes composantes configurées dynamiquement - Sécurité et fiabilité réduites - Complexité augmentée et performance réduites **Interceptor**: Lorsque un cadriciel doit être capable d’enregistrer et de déclencher de nouveaux services qui n’était pas planifiés originalement. Aussi pour permettre à d’autres applications de contrôler le comportement et la fonctionnalité du cadriciel. Ex : Google chrome, les plugins sont les interceptors et le navigateur est le framework. Conséquences :+ Extensibilité et flexibilité du cadriciel augmentées+ Séparation des responsabilités entre les services+ Capacité de monitorer et de contrôler les cadriciels+ Réutilisabilité des intercepteurs- Difficulté à anticiper;es évènements qui doivent être interceptés et à séparer les interfaces des intercepteurs- Les intercepteurs sont des points d’insertion de vulnérabilités ou de fautes

- Possibilité d’interception en cascade à cause des changements au cadriciel **Extension interface**: Permettre à une composante d’exposer plusieurs interfaces pour garantir le principe de la ségrégation d’interface, éviter d’encombrer les interfaces et d’affecter les clients lorsque les développeurs étendent ou modifient une composante. Conséquences :+ Fournit des mécanismes d’extension des composantes + Promeut la séparation des responsabilités entre les rôles + Supporte du polymorphisme entre des classes qui ne sont pas liées hiérarchiquement + Découple les composantes de leurs clients + Supporte l’agrégation et la délégation d’interfaces - Effort augmentée pour la conception et l’implémentation des composantes- Complexité des clients augmentées - Interdiction additionnelle et surcoût à l’exécution. **Scoped locking**:**:** Assure qu’un verrou est acquis lorsque le contrôle entre dans une portée spécifique et qu’il est libéré automatiquement lorsque celle-ci quitte la portée, quel que soit le chemin emprunté pour quitter la portée. **Application :** lorsqu’une section critique d’une méthode doit être protégée par un mécanisme de verrouillage. Pour éviter de sortir d’une méthode par un return ou parce qu’une exception a été lancée sans libérer le verrou. **Problème :** Du code qui doit s’exécuter concurrement doit être protégé par un verrou qui est acquis et libéré lorsque le contrôle entre et sort d’une section critique. Si les développeurs doivent acquérir et libérer des verrous explicitement, il est difficile de s’assurer que les verrous seront libérés dans tous les chemins définis par le code. **Solution :** Définir une classe de garde dont le constructeur acquiert automatiquement un verrou lorsque le contrôle entre dans la portée et dont le destructeur libère automatiquement le verrou lorsque le contrôle quitte la portée. Instancier la classe de garde pour acquérir/libérer des verrous dans des portées de méthodes ou de bloc définissant des sections critiques.Conéquences : +Robustesse et fiabilité augmentées- Blocage potentiel lorsque le patron est utilisé récursivement.- Limitations imposées par la sémantique du langage. Il assume des destructeurs qui ne sont pas toujours appelées en cas de terminaison inattendu (exit ou abort) ou des constructions particulières du langage.

**Strategized locking**: Paramétrer les mécanismes de synchronisation qui protègent les sections critiques du code contre des accès simultanés. **Application :** Lorsque un système a des composantes qui doivent s’exécuter efficacement dans une variété d’architectures concurrentes. **Problème :** Les composantes qui s'exécutent dans des environnements multifilaires doivent protéger leurs sections critiques contre l'accès simultané par plusieurs clients. L’intégration des mécanismes de synchronisation aux fonctionnalités des composantes doit trouver un juste équilibre entre la nécessité de respecter les besoins des différentes applications (mutex, verrous de lecture / écriture, sémaphores) et la nécessité d’éviter la duplication. **Solution :** Paramétrer les aspects de synchronisation d’une composante en faisant des types enfichables (plugins). Chaque type représente une stratégie de synchronisation particulière. Définir des occurrences de ces types de plug-in en tant qu'objets contenus dans une composante pouvant utiliser les objets pour synchroniser efficacement ses implémentations.Conéquences :+ Flexibilité et personalisation augmentées+ Effort réduit pour la maintenance des composantes+ Réutilisabilité augmentée- Verrouillage intrusif - Sur ingénierie **Thread-safe interface**:Minimiser la surcharge de verrouillage et s’assurer que les appels de méthodes intra-composante ne sont pas auto bloqués en essayant de réacquérir un verrou déjà détenu par la composante. **Application :** Lorsque le code critique d’une composante qui doivent être protégées de l’accès simultané est répandu parmi plusieurs méthodes qui appellent d’autres méthodes de façon intra-composante ou récursivement. **Problème :** Des composantes multifilaires contiennent parfois plusieurs méthodes publiques et privées qui peuvent changer l’état de la composante. Les méthodes peuvent s’appeler l’une l’autre pour faire leurs calculs. Dans ce cas, l’invocation des méthodes doit être conçue afin d’éviter l’auto-blocage et de minimiser le surcoût de verrouillage. **Solution :** Structurer toutes les composantes qui traitent les invocations intra-composante selon deux conventions de conception : C’est les méthodes d’interfaces qui contrôlent les verrous. Les méthodes d’implémentation font confiance aux méthodes publiques pour le contrôle des verrous.

Cnséquences : + Robustesse, fiabilité, performance augmentées +Simplification du logiciel- Indirection additionnelle et méthodes additionnelles- Blocage inter-composante potentiel- Blocage intra-composante potentiel entre différents objets- Surcoût potentiel **Double-checked locking optimization**: Réduire les conflits et le surcoût de synchronisation lorsque les sections critiques d’un code doivent acquérir un verrou une seule fois durant l’exécution d’un programme. **Application :** Lorsque une composante a une section critique qui doit être protégée contre les accès simultanés, dans le code d’initialisation, qui s’exécute une seule fois et lorsque scoped locking n’est pas efficace. Problème : Les applications concurrentes doivent s'assurer que certaines parties de leur code s'exécutent en série pour éviter les situations de concurrence lors de l'accès aux ressources partagées et de leur modification. Un moyen courant de protéger les sections critiques consiste à utiliser scoped locking, mais cela peut représenter une surcharge inacceptable lorsque le code à protéger ne doit être exécuté qu'une seule fois. Solution : Introduire une variable booléenne indiquant s’il est nécessaire d’exécuter une section critique avant d’acquérir le verrou qui la protège. Si ce code n'a pas besoin d'être exécuté, la section critique est ignorée, ce qui évite le surcoût de verrouillage inutile. Conéquences : + Minimiser le surcoût de verrouillage + Prévention des situations de concurrence - L’usage de mutex additionnel - 2 problèmes potentiels liés à certaines architectures: - Assignation non atomique des entiers et des pointeurs- Cohérence des caches sur les systèmes multi-processeurs **Active object**:Découpler l’exécution de méthode de l’invocation de méthode pour améliorer la concurrence et simplifier l’accès simultané aux objets qui résident dans leurs propres fils de contrôle. **Application :** Pour améliorer la performance lorsque des clients accèdent à des objets s’exécutant dans des fils de contrôle séparés. Lorsque des clients ne doivent pas être liés à des détails spécifiques de synchronisation, de sérialisation ou de planification de méthodes.Conséquences : + Améliore la concurrence et simplifie la complexité de la synchronisation+ Exploite le parallélisme disponible de manière transparente+ L’ordre d’exécution des méthode peut être différent de l’ordre d’invocation des méthode- Surcoût de performance- Complexe à déboguer **Half-sync/Half-async**: Découpler le traitement synchrones et asynchrones des services dans les systèmes concurrents pour simplifier le développement sans réduire la performance. Lorsqu’un système concurrent exécute des services synchrones et asynchrones qui doivent communiquer. **Application :** Lorsque un système concurrent exécute des services synchrones et asynchrones qui doivent communiquer.Conséquences : + Simplification et performance+ Séparation des responsabilités+ Centralisation de la communication inter-couches- Possibilité de pénalité en traversant les frontières entre les couches

- Les services de haut niveau ne bénéficient pas toujours de l’efficacité de l’I/O asynchrone.**Monitor object**: Synchroniser l’exécution de méthodes concurrentes pour s’assurer qu’une seule méthode à la fois s’exécute dans un objet. Permettre aussi aux méthodes d’un objet de planifier de façon collaborative la séquence de leur exécution. **Application :** Lorsque plusieurs fils d’exécution accèdent au même objet de façon concurrente. Ex Caissier qui sert les clients. Conséquences+ Simplification du contrôle de la concurrence + Simplification de la planification de l’exécution des méthodes - Complexité d’extension lié au couplage entre la fonctionnalité de l’objet et les mécanismes de synchronisation des méthodes - Possibilité de verrouillage dans le cas de Monitor Objects imbriqués **Leader/ Followers:** Patron architectural qui fournit un modèle efficace de concurrence où plusieurs fils d’exécution partagent à tour de rôle un ensemble de sources d’évènements pour détecter, démultiplexer, soumettre et exécuter des requêtes de service qui arrivent d’une source d’évènements.**Application :** Dans une application basée sur la programmation événementielle où un grand nombre de requêtes de services arrivent sur un ensemble de sources d’évènements, qui doivent être traitées efficacement par plusieurs fils d’exécution qui partagent les sources d’évènements. Ex : les voyageurs et les taxis Conséquences : + Amélioration de la performance: Améliore l’affinité des CPUs avec la cache et élimine l’allocation dynamique et le partage de tampons entre les fils d’exécution. Minimise le surcoût lié aux verrous en n’échangeant pas de données entre les fils d’exécution. Permet de minimiser les inversions de priorité en éliminant les queues supplémentaires d’évènements. Ne nécessite pas de changement de contexte pour traiter chaque évènement.

+ Simplification du modèle de programmation du traitement d’évènements concurrents. - Complexité d’implémentation: la promotion des fils de leader à follower et l’inverse doit être implémenté avec des opérations atomiques. - Manque de flexibilité: difficulté de prioriser les évènements en l’absence de queue. **Thread- specific storage**: Patron de conception qui permet à plusieurs fils d’exécution d’utiliser un point d’accès global pour récupérer un objet local à un fil d’exécution, sans encourir un surcoût de verrouillage à chaque accès à l’objet. Utilisé dans les applications ayant plusieurs fils d’exécution qui ont besoin d’accéder fréquemment à des données ou des objets qui sont stockés logiquement de façon globale mais dont l’état devrait être physiquement local à chaque fil d’exécution. Conséquence : + Efficacité: peut être implémenté de façon à ce que des verrous ne soient pas nécessaires pour accéder aux données spécifiques à un fil d’exécution + Réutilisabilité: Ce patron fournit du code qui peut être réutilisé en collaboration avec

d’autres patrons comme Wrapper Façade. + Facilité d’utilisation: une fois encapsulé dans un Wrapper Façade, le Thread-Specific Storage est relativement facile à utiliser pour les programmeurs d’applications. + Portabilité: le stockage spécifique aux fils d’exécution est disponible sur la grande majorité des systèmes d’exploitation. - Encourage l’utilisation d’objets globaux. - Camoufle la structure du système. -Restreint les options d’implémentation.**Reactor**: permet à des applications événementielles de démultiplexer et d’envoyer des requêtes de service qui sont soumises à une application par un ou plusieurs clients. Ce patron est utilisé quand une application événementielle qui reçoit de nombreuses requêtes de services simultanément, mais qui les traite de façon synchrone et en série. Ex : Appel téléphonique Conséquences : + Séparation des responsabilités: + Modularité, ré-utilisabilité et configurabilité:. + Portabilité: + Contrôle de la concurrence à gros grain:. - Applicabilité limitée.- Absence de préemption.- Complexe à débugger et tester. **Proactor**: Le patrons architectural proactor permet à des applications évènementielles de démultiplexer et d’envoyer efficacement des requêtes de service déclenchées par la complétion des opérations asynchrones afin de bénéficier des bénéfices en performance de la concurrence sans encourir certains handicaps. Ce patron est utilisé quand une application évènementielle reçoit et traite de façon asynchrone de nombreuses requêtes de services. Conséquences : + Séparation des responsabilités: + Portabilité+ Encapsulation des mécanismes de concurrence. + Découplage des fils d’exécution et de la concurrence: + Performance. + Simplification de la synchronisation des applications:. - Applicabilité limitée.- Complexe à débugger et tester. - Planification, contrôle et annulation des opérations s’exécutant de façon asynchrone. **Asynchronous Completion token (ACT)**:permet à une application de démultiplexer et traiter efficacement les réponses retournées par les opérations asynchrones qu’elle invoque sur des services. **Application :** ce patron est utilisé à l’intérieur d’un système événementiel dans lequel les applications invoquent des opérations asynchrones sur des services et doivent ensuite traiter les évènements de terminaison associés ex : FedEx Conséquence : + Simplification des structures de données de l’initiateur + Efficacité dans l’acquisition d’état+ Efficacité en espace. + Flexibilité. + Politiques de concurrence non dictatoriales - Possibilité de fuite de mémoire. - Authentification. - Invalidation suite à des déplacements en mémoire. **Acceptor-Connector**:découple la partie connexion et initialisation de la partie de traitement dans un système distribué de services pairs à pairs qui collaborent. **Application :** Ce patron est utilisé dans une application ou un système distribué dans lequel des protocoles orientés-connexion sont utilisés pour communiquer entre des services pairs connectés par des points terminaux de transport. Ex : Appel téléphonique dans un centre d’appel Conséquence :

+ Efficacité. + Réutilisabilité, portabilité, extensibilité. + Robustesse. - Indirections supplémentaires. - Complexité additionnelle.

**Architectures orientées services :** Un service est une composante logicielle qui est : indépendantes, atomique et publique Un service publie une interface Les services composent des architectures modulaires et distribuées

Patrons de composition : Chorégraphie et orchestration

SOA composantes : Entites : - Fournisseur -Client -Réseau

Artéfacts :- Implémentation service - Interface service - Implémentation client - service stub- Protocole de communication- Entente de niveau de services (SLA)

**Simple object Access Protocole (SOAP) et les WS-\* services**

SOAP est un protocole de messages. Il est indépendant du langage mais aussi du protocole de communication

SOAP utilise XML pour spécifier les données. SOAP incluse le message (en XML) et les détails de la communication (comme une demande HTTP, par exemple) Les services qui utilise SOAP utilisent les normes WS. WSDL : Les interfaces de services SOAP sont définies en WSDL **Avantages et désavantages :** **A :** Indépendant du protocole de communication, basé sur XML donc tous les outils du langage sont disponibles, il est possible de définir plusieurs normes et spécifications à nouveau. **D:** SOAP n’est pas efficace, Les interfaces de services et les formats des messages sont longs et complexes. Parcourir les fichiers est couteux.

**REST : Avantages:** Une interface simple (juste 5 méthodes). La performance est la priorité. Modificabilité, Fiabilité, évolutivité, portabilité augmentées. **Désavantages :** Manque de normes et de spécifications formelles, Les fonctionnalités plus complexes ne sont pas toujours disponibles, Il faut une organisation des données.

**Architecture de Microservices :** mini version de SOA

**Antipatrons d’adoption des architectures microservices**

- Magic pixie dust- Microservices as a goal - scattershot adoption(pas de plan) - Trying to fly before you can walk - Focussing on technology - More the merrier - Red flag law

AOP (Aspect oriented programing) fournit une solution pour modéliser les fonctionnalités transversales ex logfile

Avantage :diminue Éparpillement de code (*scattering*) et Confusion du code (*tangling*)

Ca permet 2 types de crosscutting : dynamique et statique.

Un aspect est une construction semblable à une classe permettant d’implanter des fonctionnalités transversales.

• Apect vs classe: Un aspect peut inclure des attributs et des méthodes.  Les même règles de visibilité (public, private, protected, packaged).  Un aspect peut être abstrait.  Un aspect peut dériver d’une classe, d’une interface ou d’un autre aspect (abstrait).  On peut faire des aspects embarqués tout comme avec les classes (nested aspect). Cependant: On ne peut pas directement construire une instance d’un aspect. On ne peut pas faire dériver un aspect d’un aspect concret. Un aspect peut être en mode « privileged » et accéder aux éléments privés des classes qu’il affecte.

Pontcuts : Permettent de capturer un (des) point(s) d’exécution (join point) dans le déroulement d’un programme

Advices : Les advices permettent d’exécuter du code lorsqu’un pointcut est actif.