Architecture Événementielle : Dans ces architectures, les évènements qui se produisent affectent l'état ou le comportement des objets.•**Evènement** : N'importe quelle action qui est pertinente au système(ex.un clic dans une interface graphique •**Message**: Une notification qui suit l'évènement, produite par le système ou le producteur de l'évènement et destinée à l'auditeur des évènements. (parfois messages et événements confondus)• **Bus d'évènements** : Un canal ou un tampon où les évènements sont enregistrés, temporairement stockés et transmis aux auditeurs. **Propriétés :** Capacité de parallélisation augmentée. •Le bus est commun et il peut être partagé par plusieurs producteurs et auditeurs. •Le même évènement peut être traité par plusieurs auditeurs. • Les auditeurs, en tant que processeurs des évènements, peuvent être répliqués pour paralléliser les mêmes taches. Cela augmente l'évolutivité du système. • Les évènements peuvent être agnostique à la technologie, et alors on peut définir des interfaces différentes pour des langages différents. C'est la base des architectures orientées services **Motivation** : Le matériel pas très couteux, on peut ajouter de la puissance de calcul et de traitement sans contraintes d'espace ou financières. • Les ordinateurs modernes n'ont pas un seul CPU, mais ils fonctionnent à l'aide de CPU multicœurs. Cela facilite beaucoup la parallélisation. • Les technologies de réseau ont aussi évolué. La vitesse et la capacité des réseaux ont beaucoup augmenté. • Les applications distribuées et multifilaires sont partout aujourd'hui. • Certains des avantages des systèmes distribués comprennent : • La collaboration et la connectivité : des données et des ressources distribuées sont maintenant disponibles. • La performance, l'évolutivité et la tolérance aux pannes sont augmentées. • L'économie d'échelle et les externalités de réseau améliorent la gestion des frais. **Défis(4) : 1. Accès et configuration de service** : Les services distribués peuvent être accédés en utilisant des API à divers niveaux d'abstraction (logiciel, réseau, middleware): • La communication inter-procédurale basée sur la mémoire partagée.• Les protocoles de communication (TELNET, FTP, SSH) • Les appels éloignées aux méthodes avec des intergiciels (COM+, CORBA, ou JAVA-RM).• L'évolution statique ou dynamique des services est un problème complexe qui exige des outils pour gérer : • l'évolution des interfaces et des relations entre les composantes. • La reconfiguration dynamique des ressources pour répondre aux changements de la charge. • Les clients ne doivent pas affectés pendant l'évolution. **2.Synchronisation** : Synchroniser accès ressources partagées• Acquisition et déverrouillage adéquats des verrous • Flexibilité dans les choix de mécanismes de synchronisation • Minimisation du surcoût de verrouillage et prévention de l'auto-blocage dans les appels de méthode intra-composant, • Réduction des conflits et des surcoûts pour les sections critiques rarement exécutées.**3.Concurrence** :Exécution simultanée des fils et de processus multiples requiert la gestion liées à la concurrence comme• Les blocages et l'état de concurrence. • Des API multifilaires non portables. • Des contradictions dans la sémantique des fils entre les systèmes d'exploitation. • Les problèmes fondamentaux de la conception des systèmes concurrents sont : • Le choix d'une architecture efficace qui minimise la surcharge. • La combinaison de tâches synchrones et asynchrones. • La sélection des primitives de synchronisation. • L'élimination des fils et des verrous inutiles dans les applications concurrentes en temps réel. **4.Gestion des événements** :3caractéristiques importantes distinguent les applications évènementielles des applications ayant un flux de contrôle auto-dirigé: **1**. Comportement de l'application est déclenché par des évènements internes ou externes qui se produisent en façon asynchrone.**2**. Les évènements doivent majoritairement être traités immédiatement pour éviter la pénurie de ressources et l'augmentation du temps de réponse. **3**. Les automates à état fini peuvent être nécessaires pour contrôler le traitement des évènements et détecter les transitions illégales. **Patrons de concurrence : 1.Service access and configuration patterns 1.1Wrapper Façade • Obj** : Encapsuler les fonctionnalités et les données fournies par des bibliothèques existantes (souvent non orientées-objets) par des interfaces plus cohésives, fiables, portables et facilement maintenables.• **App/ex**: Encapsuler les fonctionnalités de bas niveau (ex.du SE) dans une interface de haut niveau pour les rendre plus portables et réutilisables. L'exemple d'Android. **Consq :** + Des interfaces orientées-objets de haut niveau concis, cohésives et fiables. + Portabilité, maintenabilité, modularité et réutilisabilité augmentées. - Perte de fonctionnalité. - Performance diminuée. - Limitations imposées par les langages et les compilateurs.**1.2 Component configurator Obj** : Permettre à une application de lier et de dissocier les implémentations de ses composantes au moment de l'exécution en évitant de modifier, de recompiler ou de relier de façon statique l'application. En plus, le patron permet la reconfiguration de composantes dans des processus distincts sans devoir terminer et de relancer les processus actifs. • **App** : Le patron est appliqué lorsque 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. **Consq** : + Uniformité de la configuration et contrôle de l'interface + Administration centralisée + Modularité, testabilité et réutilisabilité augmentées + Configuration dynamique + Optimisation augmenté grâce à plusieurs possibilités de configuration - Accroissement de l'incertitude à cause de toutes les interactions possibles entre les différentes composantes configurées dynamiquement - Sécurité et fiabilité réduites - Complexité augmentée et performance réduite **1.3 Interceptor Obj**: Permet à des services d'être ajoutés à un cadriciel de manière transparente et d'être déclenchés automatiquement lorsque certains évènements se produisent. • **App** : Lorsqu'un cadriciel doit être capable d'enregistrer et de déclencher de nouveaux services qui n'était pas planifiés originalement. Aussi, pour permettre à des applications de contrôler le comportement et la fonctionnalité du cadriciel. **Consq** : +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 les évènements qui doivent être interceptés et à séparer les interfaces des intercepteurs - Les intercepteurs sont des points d'insertions de vulnérabilités ou de fautes - Possibilité d'interception en cascade à cause des changements au cadriciel **1.4 Extension Interface Objectif** : 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. • **App** : Lorsque on s'attend qu'il faudra changer les interfaces des composantes de façon impossible à anticiper après la livraison de la composante et son intégration dans des applications. Lorsque l'ajout de nouvelles interfaces pour accommoder les besoins de nouveaux clients ne doit pas affecter les clients existants. **Consq** +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 augmenté pour la conception et l'implémentation des composantes - Complexité des clients augmentées - Indirection additionnelle et surcoût à l'exécution **2**. **Synchronization patterns 2.1 Scoped locking Obj** : Assurer qu'un verrou est acquis lorsque le contrôle entre dans une portée spécifique et qu'il est libéré automatiquement lorsque celui-ci quitte la portée, quel que soit le chemin emprunté pour quitter la portée. • **App** : 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. • **Prob** : Du code qui doit s'exécuter concurremment 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. • **Sol** : 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 méthode ou de bloc définissant des sections critiques. **Consq** :+ 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 appelés en cas de terminaison inattendue (exit ou abort) ou de constructions particulières du langage (p.ex. longjmp() en C). **2.2 Strategized locking Obj** : Paramétrer les mécanismes de synchronisation qui protègent les sections critiques du code contre des accès simultanés. • **App** : Lorsqu'un système a des composantes qui doivent s'exécuter efficacement dans une variété d'architectures concurrentes. • **Prob** : 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. • **Sol** : Paramétrer les aspects de synchronisation d'une composante en 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. **Consq** : + Flexibilité et personnalisation augmentées + Effort réduit pour la maintenance des composantes + Réutilisabilité augmentée - Verrouillage intrusif - Sur ingénierie **2.3Thread-safe interface Obj** : Minimiser la surcharge de verrouillage et s'assurer que les appels de méthode intra-composante ne sont pas «auto-bloqués» en essayant de réacquérir un verrou déjà détenu par la composante. • **App** : 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 intra-composante ou récursivement. • **Prob** : 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. **Consq** : + 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 **2.4 Double-checked locking optimization Obj** : 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. • **App** : Lorsqu'une composante a une section critique qui doit être protégé contre les accès simultanés, dans le code d'initialisation, qui s'exécute une seule fois, et lorsque scoped locking n'est pas efficace. **Prob** : 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. Moyen courant de protéger les sections critiques consiste à utiliser scoped locking, mais peut représenter une surcharge inacceptable lorsque le code à protéger ne doit être exécuté qu'une seule fois. **Sol** : 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. **Consq** + 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 **3.Concurrency patterns 3.1 Active object Obj** : 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. • **App** : 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. **Consq** + Améliore la concurrence et simplifie la complexité de la synchronisation + Exploite le parallélisme disponible de manière transparente + L'ordre d'exécution des méthodes peut être différent de l'ordre d'invocation des méthode - Surcoût de performance - Complexe à déboguer **3.2 Half-sync/half-async Obj**: Découpler le traitement synchrones et asynchrones des services dans les systèmes concurrents pour simplifier le développement sans réduire la performance. • **App** : Lorsqu'un système concurrent exécute des services synchrones et asynchrones qui doivent communiquer. **Consq** : + 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 **3.3Monitor object Obj**: 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. • **App**: Lorsque plusieurs fils d'exécution accèdent au même objet de façon concurrente. • **Prob** : Plusieurs applications contiennent des objets dont les méthodes sont invoquées concurremment par plusieurs fils d'exécution. Ces méthodes modifient souvent l'état de leurs objets. Pour que de telles applications s'exécutent correctement il est nécessaire de synchroniser et planifier l'accès aux objets. • **Sol** : Synchroniser l'accès aux méthodes d'un objet de façon à ce qu'une seule méthode puisse s'exécuter à la fois. Chaque objet qui doit être accédé de façon concurrente par plusieurs fils clients est défini comme un Monitor Object. Les clients ne peuvent accéder aux méthodes définies par le Monitor Object seulement au travers des méthodes synchronisées. Pour éviter les conditions de concurrence sur l'état interne de l'objet seulement une méthode synchronisée à la fois peut s'exécuter dans le Monitor Object. La sérialisation est effectuée à l'aide d'un Monitor Lock. Les méthodes peuvent déterminer les situations dans lesquelles elles suspendent ou reprennent leur exécution en se basant sur des Monitor Conditions. **Consq** : + 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 **3.4Leader/followers Obj** : 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. • **App** : Dans une application basée sur la programmation événementielle où un grand nombre de requêtes de service arrivent sur un ensemble de sources d'évènements, qui doivent être traitées efficacement par plusieurs fils d'exécution qui partagent les sources d'évènements. • **Prob** : L'utilisation de plusieurs fils d'exécution est une technique courante pour implémenter des applications qui doivent traiter de nombreux évènements de façon concurrente. Il est cependant difficile d'implémenter des serveurs haute-performance ayant plusieurs fils d'exécution. Ces applications doivent souvent traiter des grands nombres d'évènements de différents types qui arrivent tous simultanément. **Sol**: Structurer une queue de fils d'exécution pour partager de façon efficace un ensemble de sources d'évènements et démultiplexer à tour de rôle les évènements qui arrivent de ces sources, puis envoyer de façon synchrone les évènements aux services applicatifs pour qu'ils soient traités. **Consq** : + 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. **3.5 Thread-specific storage Objectif** : Patron de conception qui permet à plusieurs fils d'exécution d'utiliser un point d'accès « logiquement global » pour récupérer un objet local à un fil d'exécution, sans encourir un surcoût de verrouillage à chaque accès à l'objet. • **App** : 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. • **Prob** : À cause du surcoût associé aux verrous, la performance des applications comportant plusieurs fils d'exécution n'est souvent pas meilleure que celles des applications utilisant un seul fil. **Sol** : Introduire un point d'accès global pour chaque objet spécifique à un fil d'exécution mais maintenir le vrai objet dans un espace de stockage local de chaque fil. S'assurer que les applications manipulent les objets spécifiques aux fils d'exécution uniquement en utilisant les points d'accès globaux. **Consq** : + 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. - Camouffle la structure du système. - Restreint les options d'implémentation. **3.Event Handling patterns 3.1. Reactor Obj** : Le patron architectural Reactor permet à des applications évènementielles démultiplexer et d'envoyer des requêtes de service qui sont soumises à une application par un ou plusieurs clients. • **App** : Une application événementielle qui reçoit de nombreuses requêtes de service simultanément, mais qui les traite de façon synchrone et en série. • **Prob** : Les applications événementielles dans un système distribué, et particulièrement les serveurs, doivent être prêtes à traiter de nombreuses requêtes simultanément, même si c'est requêtes sont ultimement traitées séquentiellement par l'application. L'arrivée de chaque requête est identifiée par un évènement spécifique d'indication. Avant d'exécuter séquentiellement un service spécifique, l'application doit démultiplexer et envoyer les évènements d'indication arrivant de façon concurrente aux implémentations appropriées des services. **Sol**: Attendre de façon synchrone l'arrivée d'évènement d'indication venant d'une ou plusieurs sources, comme p.ex. d'identificateurs de connexion réseau (socket handle) . Intégrer un mécanisme qui démultiplexe et envoie les évènements aux services qui doivent les traiter. Découpler ces mécanismes de démultiplixage et d'envoie des mécanismes de traitement des évènements d'indication dans les services, qui sont spécifique à une application. Pour chaque service qu'offre une application, introduire un gestionnaire d'évènement (event handler) distinct qui traite certains types d'évènements provenant de certaines sources. Les gestionnaires d'évènement s'enregistrent auprès du Reactor, qui utilise un démultiplexeur synchrone (synchronous demultiplexer) pour attendre qu'arrivent des évènements d'indication d'une ou de plusieurs sources. Lorsque des évènements arrivent, le démultiplexeur synchrone avertit le Reactor, qui déclenche de façon synchrone le gestionnaire d'évènement associé au service afin qu'il puisse répondre à la requête. **Consq** : + Séparation des responsabilités: + Modularité, réutilisabilité et configurabilité:. + Portabilité:. + Contrôle de la concurrence à gros grain:- Applicabilité limitée. - Absence de préemption. - Complexe à débugger et tester. **3.2. Proactor Objectif** : Le patron architectural Proactor permet à des applications évènementielles de démultiplexer et d'envoyer efficacement des requêtes de service déclenchées par la complétion d'opérations asynchrones, afin de bénéficier des bénéfices en performance de la concurrence sans encourir certaines de ses handicaps. • **App** : Une application évènementielle qui reçoit et traite de façon asynchrone de nombreuses requêtes de service. • **Prob** : La performance des applications évènementielles, particulièrement des serveurs, dans un système distribué peut souvent être améliorée en traitant les requêtes de service de façon asynchrone. Lorsque le traitement asynchrone est complété, les applications doivent gérer les évènements de terminaison transmis par le système d'exploitation pour indiquer la fin des calculs asynchrones. **Sol** : Séparer les services applicatifs en deux parties: 1) les opérations longues qui s'exécutent de façon asynchrone et 2) les gestionnaire de terminaison qui traitent les résultats de ces opérations lorsqu'elles sont terminées. Intégrer le démultiplexage des évènements de terminaison, qui sont envoyés lorsque l'opération est terminée, avec le déclenchement du gestionnaire qui traite les résultats. Découpler les mécanismes de démultiplexage et de déclenchement des gestionnaires des traitement des résultats, liés à des applications spécifiques. Pour chaque service offert par une application, introduire des opérations asynchrones qui initient le traitement des requêtes de service de façon proactive, via un identificateur (handle), conjointement avec un gestionnaire de terminaison qui traite les évènements de terminaison contenant les résultats de ces opérations asynchrones. Une opération asynchrone est déclenchée, dans une application, par un initiateur pour, par exemple, accepter des requêtes de connexion entrante provenant d'applications distantes. L'opération est exécutée par un processeur d'opération asynchrone. Lorsqu'une opération se termine, le processeur d'opération asynchrone insère un évènement de terminaison contenant les résultats du traitement dans une queue d'évènements de terminaison. Cette queue est monitorée par une démultiplexeur asynchrone d'évènements, appelé par un Proactor. Lorsque le démultiplexeur retire une évènement de la queue, le Proactor démultiplexe et envoie l'évènement au gestionnaire de terminaison spécifique pour l'application. Ce gestionnaire

traite les résultats et peut déclencher d'autres opérations asynchrones selon le même schéma. **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. **3.3 Asynchronous Completion Token Objectif** : Permet à une application de démultiplexer et traiter efficacement les réponses retournées par les opérations asynchrones qu'elle invoque sur des services.• **App** : Un système événementiel dans lequel les applications invoquent des opérations asynchrones sur des services et doivent ensuite traiter les évènements de terminaison associés.• **Prob** : Lorsqu'une application client invoque une requête applicative sur un ou plusieurs services de façon asynchrone, chaque service retourne sa réponse à l'application par un évènement de terminaison. L'application doit alors démultiplexer les événements vers les gestionnaires appropriés (fonction ou objet) qu'elle utilise pour traiter le résultat de l'opération contenu dans l'évènement de terminaison. **Sol**: En association avec chaque opération asynchrone qu'un client initiateur invoque sur un service, transmettre de l'information qui identifie comment l'initiateur devrait traiter la réponse du service. Retourner cette information à l'initiateur lorsque l'opération est complétée afin qu'elle soit utilisée pour démultiplexer la réponse efficacement, permettant ainsi à l'initiateur de la traiter. Pour chaque opération asynchrone qu'un client initiateur invoque sur un service, créer un ACT. Cet ACT contient de l'information qui identifie de façon unique le gestionnaire de terminaison, qui est la fonction ou l'objet responsable de traiter la réponse de l'opération. Passer l'ACT au service avec l'opération, qui conserve mais ne modifie pas l'ACT. Lorsque le service répond à l'initiateur, la réponse inclut l'ACT. L'initiateur peut alors utiliser l'ACT pour identifier le gestionnaire de terminaison qui doit traiter la réponse. **Consq** : + 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. **3.4 Acceptor-Connector Obj**: Il découple la partie connexion et initialisation de la partie de traitement dans un système distribué de services pairs à pairs qui collaborent. • **App**: 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.• **Prob**: Les applications dans un système distribué orienté connexion contiennent souvent une quantité significative de code de configuration qui établit les connexions et initialise les services. Ce code de configuration est largement indépendant du traitement effectué par les services sur les données échangées entre les points terminaux de transport. Coupler étroitement le code de configuration avec le code de traitement n'est donc pas souhaitable. **Sol**: Découpler la partie connexion et initialisation des services pairs de la partie de traitement que ces services effectuent une fois qu'ils sont connectés. Encapsuler les services applicatifs dans des gestionnaires de services pairs. Chaque gestionnaire de service implémente la moitié d'un service point-à-point dans une application distribuée. Connecter et initialiser les gestionnaires de services pairs à l'aide de deux usines: Acceptor et Connector. Les deux usines collaborent pour créer l'association complète entre deux gestionnaires de services et les deux points terminaux de transport par lesquels ils sont connectés, chacun encapsulé dans un gestionnaire de transport. L'usine Acceptor établit des connexions de façon passive au nom du gestionnaire de service auquel elle est associée lorsqu'arrive une requête de connexion émise par un gestionnaire de service distant. De la même façon, l'usine Connector établit une connexion de façon active vers un service distant désigné au nom du gestionnaire de service auquel elle est associée. Une fois la connexion établie, l'Acceptor et le Connector initialise les gestionnaires de service qui leur sont associés et leur passent les identificateurs de transport. Les gestionnaires de service effectuent alors les traitements applicatifs, en utilisant les identificateurs de transport pour échanger des données au travers de la connexion. En général, les gestionnaires de service n'interagissent plus avec les usines Acceptor et Connector une fois qu'ils sont connectés et initialisés. **Conséquences** : + Efficacité. + Réutilisabilité, portabilité, extensibilité. + Robustesse. - Indirections supplémentaires. - Complexité additionnelle. **SOA modularité:** +Reutilisabilité, +confiance à l'expertise des autres, +découpler les composantes. -inclut le partage de code. -dépendances technologiques -des dépendances temporelles(mise à jour manuelle). **CORBA** : méthode pour partager des objets. courtier (ORB) achemine les requêtes aux objets indépendamment du langage ou SE. Partage objet application langage différents.**CORBA** (problème implémentation, conception (par comité = bcp conflit, manque norme), transparence de la localisation (objet traité même façon indépendamment localisation = problème performance) **Services Web : Service** = composante logicielle indépendante, atomique(produit résultats complets), publique(expose expertise/fonctionnalité ou donnée de facons explicite), service publie une inetrface (boite noire pour client), services component architectures modulaires et distribuées. **SOA Entités 1.Fournisseur** fournit service, responsible pour son implementation et publication de son interface. expose ses fonctionnalités et ses données propriétaires (avec contrôle explicite), assure bon fonctionnement du service, déploiement, disponibilité et qualité. Assure bonne conception de l'interface du service. **2.Clients:** utilise service, développe application-client consommé données fournies par service, s'attend niveau de qualité promis par fournisseur, suit terme utilsiation impose par fournisseur, doit générer stub dans le langage de l'application locale. Responsible adapter application avec évolution service. **Réseau:** mécanisme de communication entre service et app-clients, on parle internet, mais aussi reseau privé dans entreprise, fournisseur responsible exposer service sur reseau et cient responsible maintient connexion à ce reseau, défaillance du reseau = responsabilité fournisseur (affecte la SLA) **SOA: Service et Application:** développement service et application independent (langage different). Pour être déployées comme service, implementation doiit avoit annotation spéciale ou dépendre librairies pour permettre à outil de générer interfaces et code pour le déploiement. Des librairies nécessaires à l'application-client pour accéder aux service sur un reseau. Librairies = spécifique au langage de programmation. **SOA: Interface du service:** Spécifie ce qui est réalisable par le service, spécifie operation qui peuvent être appelés par client (nom, paramètres), les données qui sont disponible sou qui sont utilisées par les operations (Nom, type, structure), l'endroit ou se trouve le service, la version, dépendances potentielles et autres méta-informations. Parfois interface service = un document formel qui peut être consommé par outils automatique(generation stubs, découverte service, identification differences entre version). Interface spécifiée dans langage abstrait(XML, HTML). **SOA: Service stub:** Code qui rend possible l'accès au service, généré automatiquement ou disponible en tant que librairie dans langage l'app client. Peut être complexe, mais pas necessaire de le comprendre ou modifier. Information définies dans interface du service suffisantes pour générer stub, outils de generation complètent ces infos avec des configurations pour l'accès reseau. **SOA:Protocole de communication** Dicte la facons de communiquer entre modules sur reseau, données qui transite, format et type connexion entre les modules. Type de protocols (HTTP=données, SMTP=courriels, FTP=fichier). Il y a a des types de service fonctionne avec plusieurs protocole (SOAP) ou REST. **SOA: SLA:** Entente fournisseur et consommateur service concerne livraison et qualité du service. Dans document(types services dispo, performance désirée, méthodes de monitorage, comment rapporter problemes, temps répondre et résoudre probleme, repercussion et pénalité pour fournisseur si qualité pas conforme) Métriques (temps de réponse pour une demande, nombre demandes servies par unite temps, taux de disponibilité. **Simple Object Access Protocol(SOAP) et WS-*services:** SOAP =protocole de messages, independent langage et du protocole de communication, utilise XML pour specifier données, include le message(XML) et detail de communication. Service utilise SOAP utilise norme WS (Web Service Definition Langage, WS-Security, WS-Politicy) **WSDL:** interface de SOAP définies en WSDL = langage base sur XML. Fichier WSDL spécifie operation, types retour, type parametres, spécifie localization du service et comment y accéder(liaison et protocole de communication) **SOAP: Avantage:** independent protocole de communication, base sur XML=>tous outils langage disponible, possible definer plusieurs norms et specifications à nouveau. **Désavantage**: pas efficace, interface de service et formats des messages longs et complexes, parcourir fichiers est couteux. **Representational State Transfert (REST):** style architectural pour créer services web, base essentiellement sur HTTP, pour accéder aux ressources dispo sur web, identificateur unique pour toutes les ressources, on peut accéder à une resource spécifique ou à une collection de ressources par URL.5opérations(GET,PUT,POST,PATCH,DELETE), interfaces des services parfois publiées comme page HTML, requêtes des clients au service sont atomiques (une response ou exception par requete) **REST Avantage:** interface simple, performance est priorité, modificabilité, fiabilité, évolutivité, portabilité augmentés. **Désavantage**: Manque normes et specifications formelles ajoute défis aux taches périphériques pour maintenance des systems et service, fonctionnalités plus complexes, pas tirs dispo, il faut organization des données. **Remote procedure calls (XML/JSON RPC):** ce nest pas du REST, utilise n'importe quell methode, format standat pour le transfert donn/e, avantage/d/savantage Presque meme REST avec plusieurs capcit/s niveau methode et sans imposer organisation des données. **Microservice**: version petite de SOA, chaque service = une operation avec une responsabilité, garde propriété de SOA(réutilisabilité, portabilité, interopétabilité). Plus facile composer et remplacer, déployés et gérés indépendamment, technologie fundamental pour Devops avec conteneur, promeut automatization, évolution et livraison continue. Défis(transfert charge cognitive) **Movitation vers microservice:** Réutilisabilité, gestion données décentralisée, déploiement automatique, évolutivité. Processus développemetn: organization équipe suit architecture système, combine expert pour chaque microservice, une équipe-coeur pour gérer logiciel au niveau système. **Avantage monolithe:** une équipe développeur, facile developer(plupart IDE et outil développement = monolithe), comprehensible et modificabilité accrues, facile tester, permet déploiement continu, possible mettre en echelle horizontale, bénéficier cadriciels et caractéristiques langage existent.**Désavantage monolithe:** + grossit, plus problemes, compréhensibilité reduite, integration difficile, facile briser modularité, plus base code grande et plus IDE et conteneur d'app Web surcharge, maintenabilité et évolutivité difficile, nécessote engagement long terme envers pile technologique, peut entrainer un verouillage du fournisseur (vendor lockin) **Application web microservice:** Un module pour chaque composant logique de l'app serveur, utilization équilibreur de charge/de la passerelle/ de l'Appstore pour decoupler demandes, BD dédiées par module. **De Monolithe a microservices: Avantage:** amelioration maintenabilité et testabilité (modules et équipe + petit, équipe expert), amelioration productivité, isolement d'anomalie (panne n'affecte pas tous), éliminer l'engagement à long terme envers les technologies **Désavantages**: Compléxité supplémentaire(développement, Testing, Déploiement) Augmentation consummation mémoire (chaque service reserve mémoire) **Scaling Monolithe vs Microservice:** au lieu mettre à l'échelle ensemble de l'app, on réplique que les composantes surcharges, facilite maintenabilité et déploiement continu (redéploie les modules concernés par changement, on peut conserver simultanément différentes versions modules tout en passant à une nouvelle = deployment Canary) **Caractéristiques Architecture Microservice: 1. Division en composantes via services:** composant = unite logiciel remplacable indépendamment et evolutive, deux facons de faire des composant: Bibliothèques: composants liés à un programme via des appels de function en mémoire, + Frontières de modularité plus douces, appel en mémoire rapide moins cher. Services: composant hors processus avec capacités de communication à distance, + Deploiement independent, interfaces publiques et explicites **2. Décomposition des affaires et du développement (Business oriented, development oriented) Décomposition du développement:** expert pour technologies et couches de l'application (expert GUI, BD, du service). Puisque chaque équipe devra travailler avec tous les composants, l'archiecture est plus monolithique. Peut entrainer reunions fréquente entre équipe, conflits, violation budget et délai. Peut conduire injection de logique dans toutes couches. 4 **Décomposition des affaires**: équipe organisées par module, équipes interfonctionneles pour inclure tous experts, architecture reflète organization de l'équipe de développement, meilleure maintenabilité et productivité) ***3. Produits et non projets:*** développement orienté projet fournis ensemble de fonctionnalité et projet repris par une équipe de maintenance et d'exploitation. Dans développement orienté produit, développeur sont aussi responsible de l'exécution du système, le logiciel en production. Mentalité produit mieux liée aux capacité de l'entreprise, mentalité produit peut être appliqué aux monolithes, mais composants plus petit et ciblé pour ces objectifs commerciaux spécifiques sont mieux adaptés. **4.Points de terminaison intelligents et tuyaux stupides:** SOA traditionnelle utilise moyens lourd pour communication entre service (protocole SOAP, Buse de service d'entreprise), ces moyens on tune logique importante pour permettre formatage message, rassemblement et le démarshallling, le routage et la chorégrapgie. Microservices privilégient moyen plus simples (HTTP ou middleware orienté message avec moins logique). La logique sur comment gérer message = service récepteur, logique sur comment preparer message = service expéditeur. Services deviennent + complexex, mais grace modularité et specialization, on a + flexibilité sur les types de messages qu'on peut traiter. **5. Gouvernance décentralisée:** Chaque service est responsable de lui-même, concerne l'utilisation de la technologie (chaque service peut utilizer sa propre pile technologique et éviter blocage du fournisseur pour le systeme), "vous le construisez, vous l'exécutez" (chaque équipe responsible du fonctionnement de son propre service) **6.Gestion décentralisée des données:** une entité peut avoir différentes vues au sein grande application (client pour vente, client pour support), selon domain-driven design(DDD), on peut divier domaine complexe en plusieurs domaines délimités, ceci à égalité entre decomposition orienté business et gouvernance décentralisée. **7. Automatisation de l'infrastrucure**: concentrer sur taches répétitives à chaque changement (testing, déploiement), peuvent être automatisé même dans monolithe**. 8. Concevoir pour l'échec**: tout peut échouer a tout moment, minimiser impact avec client, monitorage est la clé (sein service et entre service, monitorage technique (throughout des demandés) et monitorage commercial (throughput des commandes), ingénierie du chaos (pratique injection aléatoie, inatendue intentionnelle panes pour voir comment échec est résolu) **9. Conception evolutive:** Possible de réecrire ou remplacer completement composans sans affecter ses dépendants => développez système en tant que cadriciel, même logique lors migration monolithe vers un composant de microservice (Strangler Application = monolithe reste comme système central qui expose API, nouvelle fonctionnalités ajouté en tant que nouveaux services, remplace progressivement fonctionnalités de base par d'autres microservices), Antipatron ShotgunSurgery si on modifie a plusieurs reprises deux service ensemble on devrait les fusionner. **Antipatrons d'adoption des architectures microservice**: **Magic pixie dust** Une pincée de microservice résoudra tous nos problemes l'adoption d'une architecture doit s'accompagner des principes, patrons, philosophies cultures appropriés, utilisez-vous pratiques appropriés pour architecture de microservices? Avant d'adopter architecture microservice, pensez: cause problème? **Microservices as the goal:** Responsable annonce initiative a transformer tous les systems en microservices, avant adopter architecture essayez de réparer et preparer le monolithe lui-même, adoptez tests et déploiement automatisés, élimier travail inutile et améliorez monitorage. **Scattershot adoption:** Une consequence de microservice as the goal, annpnce l'adoption de microservices, mais jamais specifier de plan ou stratégie de faire. Chaque équipe adopte sa propre strategie, donc manque general de coordination. **Trying to fly before you can walk:** adoptez architecture avancée, comme microservices avant d'adopter les bonnes pratiques de développement logiciel et les principes de conception logicielle, enforcer assurance automatisée de la qualité du code dans cadre pipeline déploiement, adoptex processus de développement axé sur les tests pour enforcer des tests automatisés dans cadre pipeline integration continue, utiliser principe de conception ou de conception base sur le domaine pour faciliter la decomposition du monolithe, refactorisez le monolithe avant migrer. **Focusing on Technology** C'est pas parce qu'on adopte une technologie spéciale que la migration (ou architecture de microservic) fonctionnera, emphase sur la technologie peut rapidement conduire à un vendor lockin, Avant de choisir une pile technologique:Mettre l'emphase sur la conception (décomposition du monolithe en services), Mettre l'emphase sur l'organisation (une équipe par service), Mettre l'emphase sur les objectifs et les capacités de l'entreprise. **More the merrier** Microservice n'implique pas littéralement petit. Prendre cela à la lettre peut conduire à un anti-modèle appelé nanoservice.Des services plus petits impliquent beaucoup plus de services. Plus de dépendances, communication complexe entre les services. Augmentation de la complexité du développement et communication accrue, définir un service par équipe. Ne pas ajouter de nouveau servic, sauf si résout problemes. Divisez les services trop volumineux, divisez les équipes (et leurs services) qui deviennent trop grandes. **Red Flag Law:** lorsqu'une organisation continue d'utiliser des pratiques incompatibles avec les microservices:Processus en cascade, tests manuels, commits pendant des fenêtres de maintenance périodique prédéterminées Équipes d'experts à travers le système, il n'y a aucun avantage à migrer vers une architecture de microservices. Avant d'adopter une architecture de microservice: Adoptez les méthodes DevOps pour permettre l'automatisation et l'intégration entre les équipes. Réorganisez vos équipes, des équipes d'experts aux équipes transversales par service, on peut adopter le modèle«Strangler Application» pour appliquer ces modifications de manière incrémentielle.**Programmation orienté aspect(AOP) :** + concret –> + abstrait (langage machine, procéduraux, orienté object, orienté aspect) **orienté objet(OO) avantage** : puissants, réutilisation, séparation interface/impl/mentation, typage. **Désavantage OO:** difficulté modéliser fonctionnalités transversales(crosscutting concerns), éparpillement code (code scaterring) et confusion de code (code tangling), mène classes et module moins indépendants et moins réutilisable, AOP fournit solution pour modéliser fonctionnalités transversales sans affecter classes . **AOP combine technologie existante**: AOP utilize avec orienté objet et même C, modélise fonctionnalités transversales (CCC) avec des aspects, code weaving(permet convertir aspect code compilable par un compilateur OO standard) **Avantage AOP:** diminue éparpilement code (code fonctionnalité transversal localisé à un endroit précis, plus facile comprehension et maintenance code), diminue confusion code(classes issues concepts du domaine plus forte cohesion, pas pollué par code de foncitonnalité transversal, classes plus réutilisables) **AOP fonctionnement:** définit nouvelle construction(=aspect) permet de modéliser fonctionnalités transversales, permet 2 types de crosscutting sur code **Crosscuting dynamique:** intercepte évenement dans déroulement programme, execute operation avant/apres/pendant evenement, type + frequent. **Pointcuts:** permet capturer des points d'exécution(join point) dans déroulement programme, **Advice:** permet executer code lorsqu'un pointcut est actuf, peut executer code avant ou après activation pointcut ou modifier environnement execution pointcut **Crosscuting statique:** modifie structure statique des classes (ajout attributs ou fonctions à une classe, modification hiéarchie classes, ajout contrainte compilation), utilié pour implanter crosscutting dynamique. Permet introduire des members à une classes, Modifier hiéarchie des classes, définir messages à la compilation **AspectJ: aspect** = construction semblable à classe permet implanter fonctionnalité transversales, similarités avec classes (aspect peut inclure attributs et méthodes, regles de visibilité public…, peut être abstrait, peut dériver classe…, aspect embarqué nested aspect) Différence à classe = peut pas directement construire instance, pas faire dériver aspect d'un aspect concert, aspect en mode privileged peut accéder aux éléments privés classes qu'il affecte. **Trace d'exécution**: exemple + connu AOP = logfile, si veut garder trace d'exécution on doit ajouter code chacune fonciton = long, ardus, propice erreur. Avec AOP, ajoute quelques ligne claires place dans un aspect pour enregistrer trace d'exécution de facons non invasive, pas besoin de modifier chacune des fonctions indendamment. **AOP : réalité :** flot de contrôle difficile à suivre, on veut un niveau abstraction plus élevé, AOP ne regle pas nouveau probleme, on résout probleme actuel + efficace, AOP permet corriger probleme difficilement corrigible en OO, binen conception OO des classes issues du domaine du probleme reste necessaire, AOP augmente modularité, compréhensibilité et tracabilité, AOP brise encapsulation mais facons systématique et controlee, AOP ne remplace pas OO, concept issu du domaine du probleme continuent d'être modelisés avec des object, AOP pour fonctions transversal modélisée par aspect,AspectJ parfaitement intégré à plusieurs IDE, AspectC++ dispo, grand nombre langage. Champs application très varies. Champs application: design pattern(implantation de certains patrons de conception avec AOP est avantageuse). Patrons observer: solution invasive, code éparpillé, il faut modifier classes issues du domaine Navigateur et Noeud, modifie ou retire patron on doit modifier hiéarchie statique classes, classes on perdu cohesion et -réutilisable, Avec AOP défini dans aspect et les classes issues domaine Noeud et Navigateur pas affectés, 2 aspects: un aspect abstrait définissant éléments communs, un aspect concert concrétise utilisation patron dans context particulier, patron appliqué de l'extérieur. Avantage: diminue éparpillement code, classes Noeud et navigateur pas affectées par instanciation patron (+ cohesion, facilite instanciation et désinstanciation patron, transparence composition patrons). AOP réduit: enchevetrement code, dispersion, profondeur heritage, compléxité. Accroit: couplage, cohesion, NLC.