R5.A.08 :
Qualité de développement

Chouki Tibermacine
Chouki.Tibermacine@univ-ubs.fr

Plan prévisionnel de la ressource

1. Intro aux architectures logicielles
2. Documentation d'architectures en UML
3. Styles et patrons d'architectures
4. Architectures à microservices
5. Design & Implem de frameworks
6. Patrons de conception : Façade, Bridge, MVC et MVVM
7. Patrons de conception : Builder, Proxy et Visitor
8. Autres patrons

Plan prévisionnel de la ressource

1. Intro aux architectures logicielles
2. Documentation d'architectures en UML
3. Styles et patrons d'architectures
4. Architectures à microservices
5. Design & Implem de frameworks
6. Patrons de conception : Façade, Bridge, MVC et MVVM
7. Patrons de conception : Builder, Proxy et Visitor
8. Autres patrons

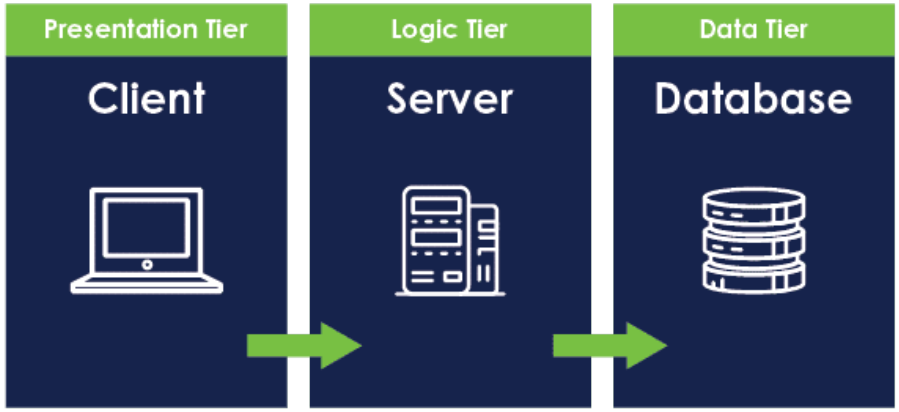
Style d'architecture

- Un vocabulaire et des contraintes d'assemblage des composants
- Exemple : le style *Pipe & Filter*
  - Vocabulaire : composants de type "Filter" et connecteurs de type "Pipe"
  - Contraintes d'assemblage : les pipes sont orientés dans le même sens
- Catégories de styles :
  - Styles d'architecture pour la modularité : "en couches", *Pipe & Filter* (ou "en *Pipelines*"), MVC, *Blackboard*, *Event Bus*, ...
  - Styles d'architecture pour la distribution : client-serveur, multi-niveaux, *Peer to Peer*, *Broker*, SOA, ...

Style d'architecture en couches (en étages)

- Chaque étage correspond à un niveau d'abstraction
- Les étages supérieurs (les plus abstraits) sont responsables des interactions avec les utilisateurs
- Les étages inférieurs sont responsables des interactions avec le matériel, les données, ...
- Les échanges ne peuvent se faire qu'entre deux étages consécutifs
- Exemple : modèle OSI des réseaux informatiques
- Avantages : les étages sont réutilisables et substituables, modularité (dépendances faibles entre étages à travers des interfaces)

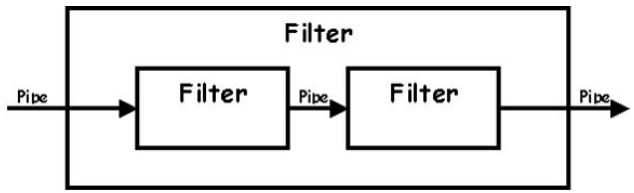
Style d'architecture en 3 couches (3-tier)



- Couche "Présentation" : affiche les interfaces aux utilisateurs
- Couche "Logique/Application" : effectue les traitements (implémente les règles métier de l'application et l'entreprise)
- Couche "Données" : gère la persistance des données (bases de données, SI, systèmes légataires, ...)

Style d'architecture du *Pipe & Filter*

- Des données sont émises en entrée d'une série de filtres (*Filters*) reliés par des tubes (*Pipes*)



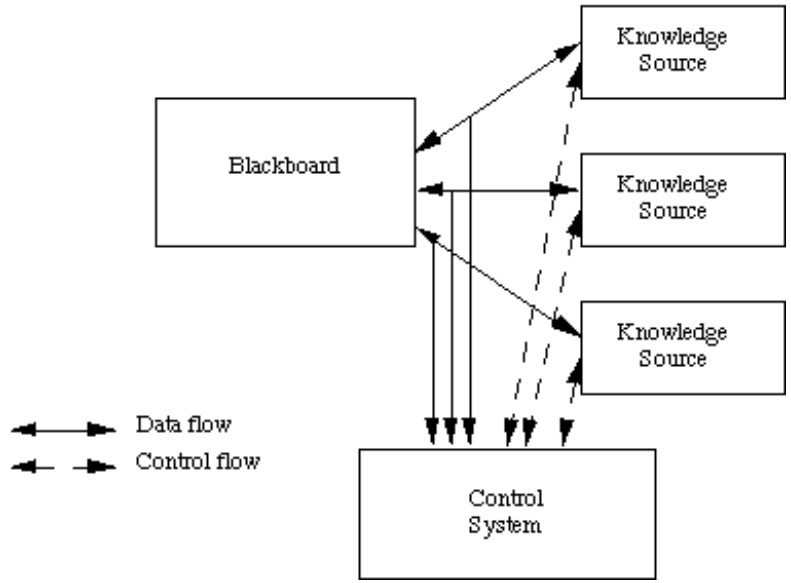
- Un filtre transforme la donnée reçue en entrée et l'émet en sortie vers le filtre suivant
- Un pipe véhicule la donnée et peut servir à synchroniser, bufferiser, ...
- Exemples : 1) dans les processeurs modernes plusieurs pipelines/étages *Fetch* > *Decode* > *Execute*, 2) les architectures d'apprentissage automatique/profond, 3) les compilateurs, ...

Avantages du style d'architecture du *Pipe & Filter*

- Efficacité : possibilité de parallélisme
- Extensibilité et adaptabilité : facilité d'ajouter des filtres ou de remplacer un filtre ou un pipe existant pour mettre en place une nouvelle transformation ou un nouveau mode de communication
- Puissance de calcul : possibilité de rendre l'architecture recursive

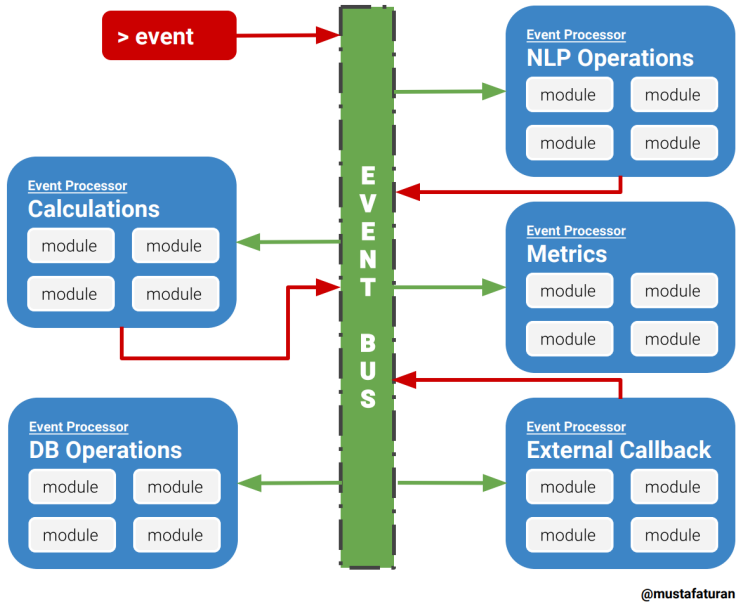
Style d'architecture du *Blackboard*

- Un composant central (le *Blackboard*) centralise les données communes à divers petits traitements
- Un contrôleur de ce "tableau noir" (un autre composant) notifie différents autres composants écouteurs (*KSs : Knowledge Sources*) des changements d'état sur les données
- Avantages : passage à l'échelle et modifiabilité



Style d'architecture de l'*Event Bus*

- Un composant principal (le bus) traite et diffuse des événements de façon asynchrone
- Autres composants autour : *publishers* et *subscribers*
- Le bus peut filtrer les événements (*RabbitMQ*) ou non (*Redux*), peut ordonner la diffusion (*Kafka* de LinkedIn) ou non, ...
- Avantages : passage à l'échelle et réactivité

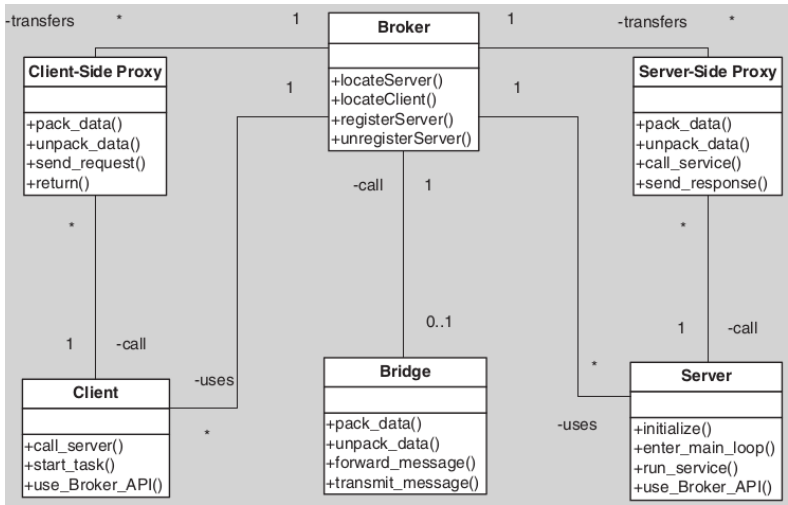


Style d'architecture du *Peer to peer*

- Un ensemble de composants qui sont à la fois clients et serveurs (initient et répondent aux requêtes)
- Un client qui accède à une ressource trop sollicitée devient son serveur (parfois serveur de segments de ressources)
- Pas d'asymétrie comme dans le style client-serveur
- Exemple : les logiciels torrent
- Avantages : disponibilité et perf. (réduction de charge sur les serveurs)

Style d'architecture du *Broker*

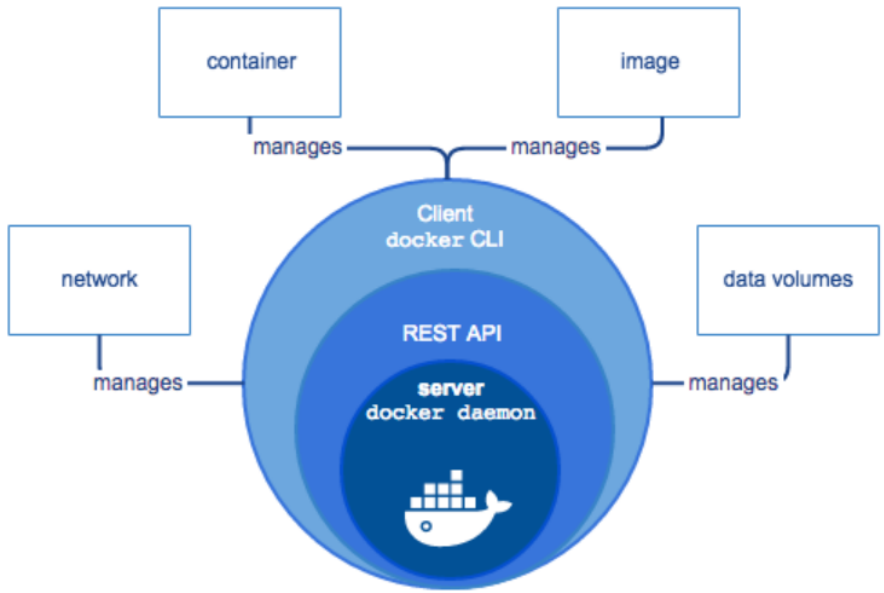
- Un composant courtier (*Broker*) fait l'intermédiaire entre les composants client et les différents composants serveurs dont ils ont besoin



- Avantages : modifiabilité, disponibilité (possibilité de changer un serveur défaillant facilement) et perf. (répartition de charge)

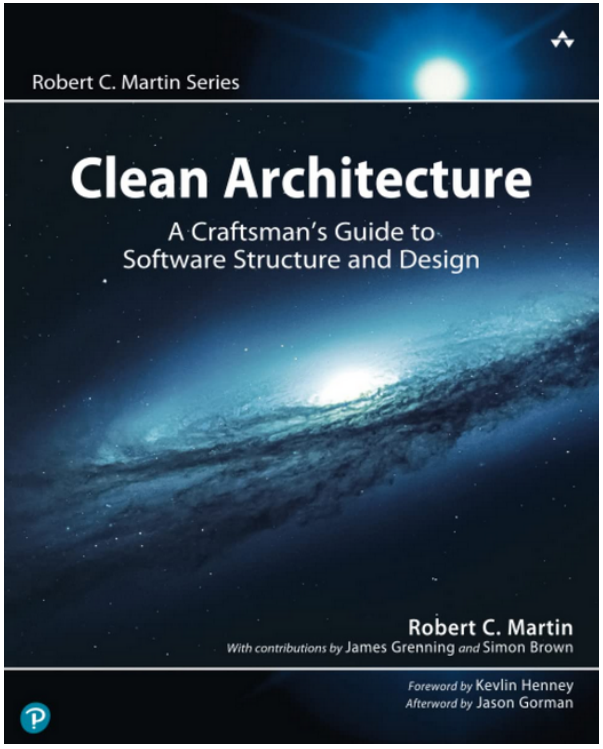
Style d'architecture du *Microkernel* ou Oignon

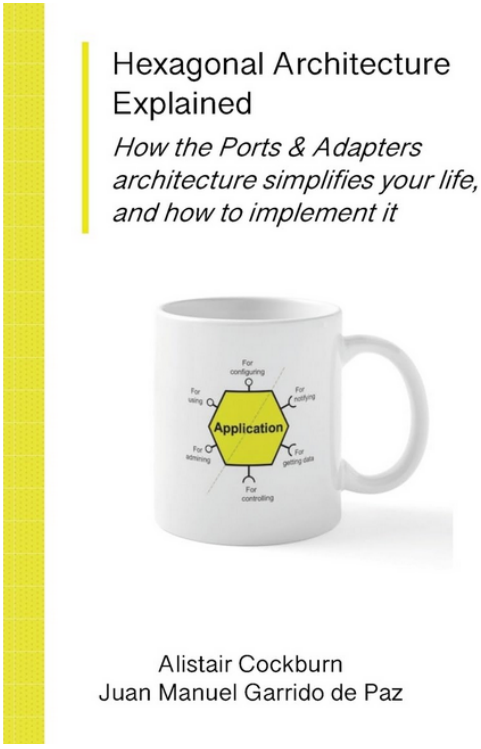
- Structure en couches sous forme de disques concentriques
- Le noyau (minimaliste) encapsule les services de base
- Les couches supérieures fournissent les services externes
- Exemples : Machine virtuelle Java, systèmes Unix, Docker, ...



Style d'architecture du *Microkernel* ou Oignon

- Avantages : extensibilité et portabilité
- Instances particulières (variantes) :
  - Clean Architecture (*Robert C. Martin – Uncle Bob*)
  - Architecture hexagonale (*Alistair Cockburn*)





Style d'architecture du *SOA*

- Des composants fournissent des services et d'autres les consomment, sans connaître l'implém des fournisseurs
- D'autres composants intermédiaires peuvent exister :
  - ESB (*Enterprise Service Bus*) : un courtier jouant le rôle d'adaptateur format/protocole, de contrôleur de sécurité, de transactions, ...
  - Annuaire (*Registry*) : pour découvrir des fournisseurs de services à l'exécution
  - Orchestrateur : gérer différents consommateurs et fournisseurs et déclencher l'exécution de scripts (processus/workflows)
- Avantages : interopérabilité, portabilité et modifiabilité

Style d'architecture des microservices

- Les microservices sont des modules :
  - traitant un domaine métier précis (gestion des commandes, par ex) et pouvant couvrir plusieurs couches techniques (présentation, logique métier, persistence des données, ...) ;
  - encapsulent les données qu'ils manipulent (gèrent leurs propres bases de données) ;
  - peuvent être livrés, construits et déployés de façon indépendante (s'exécutent dans des processus indépendants).
- Ils communiquent en s'appuyant sur des mécanismes IPC (*Inter-Process Com.*) : HTTP (sockets), files de messages, ...
  Pas de dépendances statiques entre microservices
- Avantages : modifiabilité, passage à l'échelle et livraison rapide

Quelques technologies favorisant des architectures à composants explicites

Java EE (devenu Jakarta EE – 2018)

- Un composant = unité de déploiement ⇒ des classes + ressources empaquetées ensemble (archives JAR, WAR et EAR)
- Gestion transparente de la distribution
- SOA (objets distribués RMI et services Web) + annuaire (JNDI)
- Architectures multi-niveaux (n-tiers) et mapping objet-relationnel
- Modularité limitée : pas de dépendances ni de structures internes explicites
- Site Web :

  `https://www.oracle.com/technetwork/java/javaee/overview/index.html`

Spring MVC/Boot

- Spring : un framework Java pour simplifier la programmation d'applications d'entreprise
- Composants Spring (beans) de petite taille
- Un composant = unité modulaire avec une architecture explicite
- Un composant = une classe Java annotée, instanciée automatiquement, et dont les dépendances sont résolues automatiquement à l'exécution
- Site Web : `http://spring.io/`

OSGi

- OSGi : un framework Java pour modulariser les applications de grande taille
- Un composant (bundle) = unité de déploiement ⇒ une archive JAR avec des dépendances explicites (interfaces requises sous la forme d'une liste de packages requis)
- Gestion fine du cycle de vie d'un composant : installation, démarrage, mise-à-jour (remplacement à chaud), arrêt et désinstallation
- *SOA in a JVM* avec un annuaire de services
- Applications connues développées avec OSGi : Eclipse, Netbeans et Jitsi (`https://jitsi.org/`)
- Site Web : `https://www.osgi.org/`

Modules Java 9+

- Depuis la version 9 du JDK (sept. 2017), Oracle a introduit un système de modules largement inspiré d'OSGi
- Un composant/module = un ensemble de packages, avec des dépendances explicites (interfaces requises et fournies déclarées dans un descripteur de modules)
- Résolution statique des dépendances : détection de dépendances cycliques, ...
- *SOA in a JVM* sans annuaire (*registry*) de services

Projets/Modules Maven ou Gradle

- Projet = un composant avec des dépendances explicites
- Un projet comporte une configuration (pom.xml ou build.gradle) qui décrit les phases du cycle de vie pour automatiser le build (compilation, tests, déploiement, exécution, ...)
- Dans cette config, on indique les noms/versions d'autres projets utilisés (élément *dependencies*)
- Un outil de build résout de façon transparente (après téléchargement des dépendances depuis des dépôts, comme Maven Central) pendant le build
- Application pouvant être structurée en composants (modules Maven, par ex) inter-dépendants

Sous-systèmes conteneurisés (dockerisés)

- Un sous-système conteneurisé = un composant avec des dépendances explicites
- Un orchestrateur de conteneurs, comme Docker-Compose, Docker-Swarm ou Kubernetes, permet de résoudre les dépendances déclarées (élément *depends_on* entre services dans docker-compose.yml par ex)
- Cet orchestrateur va :
  - télécharger (si ce n'a pas été déjà fait) une image du conteneur (avec la bonne version) depuis un dépôt comme le Docker Hub
  - démarrer des services (isolés) par conteneur, dans le bon ordre
  - et leur permettre de dialoguer (s'envoyer des messages via des réseaux virtuels isolés)

Sous-modules Git

- Pour la gestion de dépôts de code de modules indépendants ;
- Une fonctionnalité de Git qui permet d'inclure un dépôt comme sous-dossier d'un dépôt existant ;
- Utile pour référencer une version précise (stable) d'un module dans le dépôt d'un autre : un sous-module Git pointe vers un commit précis d'un dépôt externe ;
- Peuvent s'avérer utiles dans les projets multi-repo d'applications avec une archi à MS ;
- Mais attention : workflow plus complexe (commandes++, *detached HEAD state* & synchronisation avec la dernière version pour toute l'équipe)

`https://git-scm.com/book/fr/v2/Utilitaires-Git-Sous-modules`