

IFT 2255 – Génie logiciel

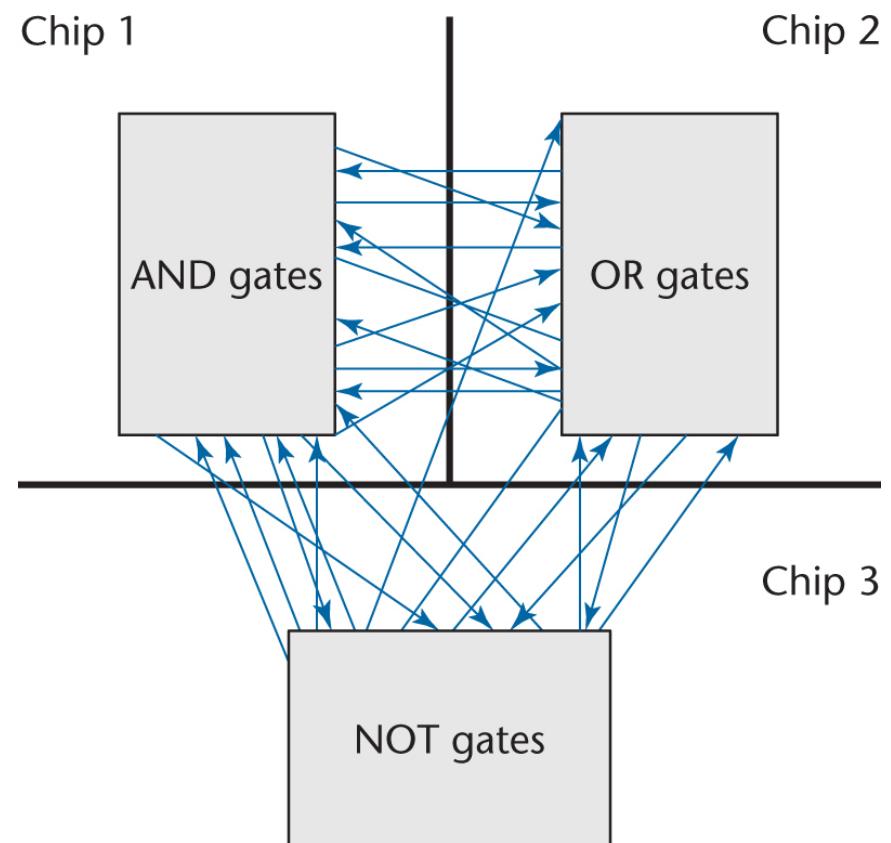
Architecture

Conception architecturale

- Aperçu haut niveau du système
 - Composants principaux, leur propriétés et leurs collaborations
 - Exigences **non-fonctionnelles** dirigent l'architecture
 - Portabilité, fiabilité, robustesse, maintenabilité, sécurité, etc.
- Conception des **interfaces utilisateurs** du logiciel
- Conception des **bases de données**
- Conception des **points de contrôle** du logiciel
 - Correction, sécurité, tolérance de fautes, protection des données
- Conception du **réseau**: communication entre processus distribués
- Allocation de chaque composant aux **ressources matérielles**

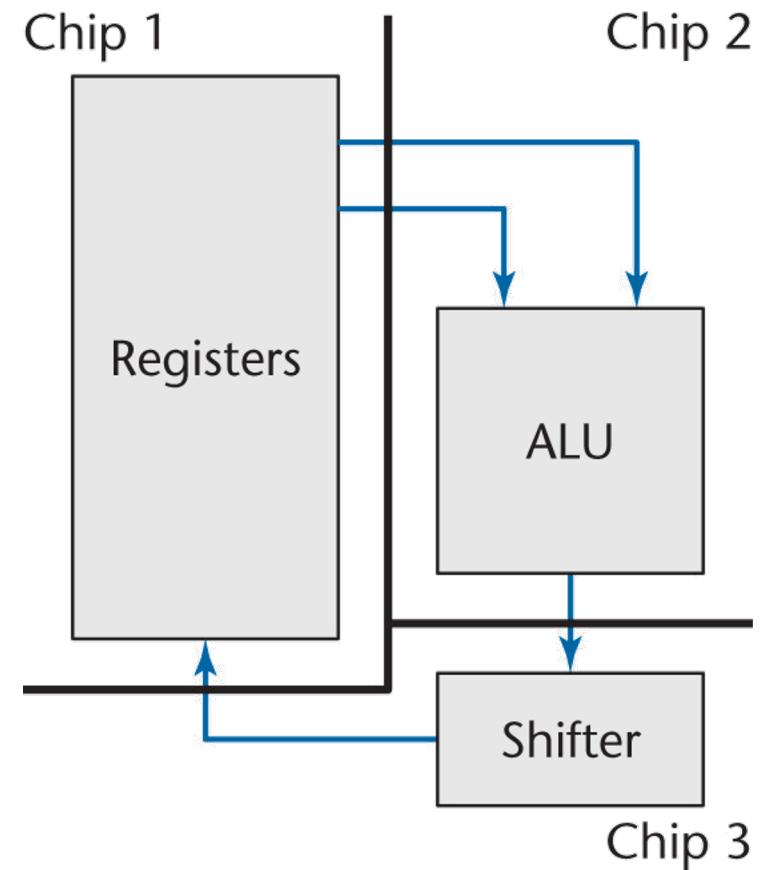
Conception d'un ordinateur

- Un architecte incompetent décide de le construire avec des fonctions ET, OU et NON
 - Reconçoit avec un type de fonction par puce
 - Résultat: le chef-d'œuvre!



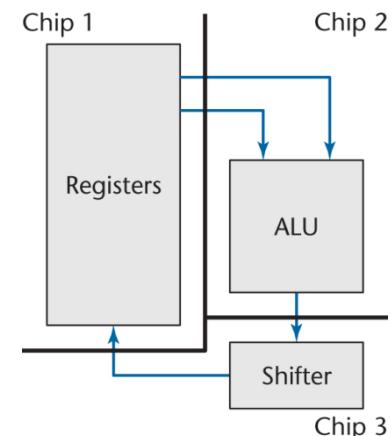
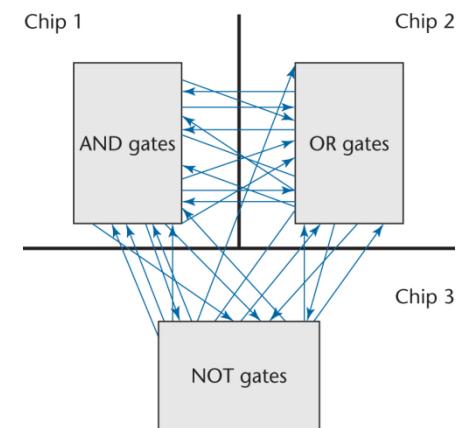
Conception d'un ordinateur

- Un architecte décide de construire une unité de logique arithmétique (ALU) avec 16 registres à décalage en utilisant des fonctions NON-ET ou NON-OU
- 3 puces en silicium



Conception modulaire vs. monolithique

- Les deux designs sont équivalent d'un p.d.v. fonctionnel
- Le 2^e est difficile à
 - Comprendre
 - Localiser les fautes
 - Étendre ou améliorer
 - (impossible à) Réutiliser dans un autre produit
- Les modules doivent être comme le premier design
 - **Maximiser** les relations **au sein** du module
 - **Minimiser** les relations **entre** les modules



Couplage et cohésion

- Décomposer le design en modules de sorte que
 - On maximise les interactions au sein du module
 - ET
 - On minimise les interactions entre les modules
- **Cohésion** d'un module
 - Degré d'interaction au sein du module
- **Couplage** d'un module
 - Degré d'interaction entre les modules

Couplage et cohésion dans le code

```
// Deux choses se passent, pas évident
// Couplage fort
```

```
int x = lireVecteur(v);
```

```
// Compréhensible mais pas réutilisable
int x = lireVecteurEtCalculerSomme(v);
```

```
// Deux actions séparées et avec des bons noms
// Forte cohésion
```

```
var V = lireVecteur(v);
int x = calculerSomme(V);
```

Couplage et cohésion en orienté-objet

```
Driver john = new Driver();
Car ford = new Car("Ford", "red");

public class Driver {
    Car myCar;
    public void goFaster(int speed) {
        myCar.speed += speed;
    }
}
```

- Car permet d'accéder à son attribut speed: couplage fort
- Comment résoudre ce problème ?
 - Ajouter une méthode faster(:int) dans Car
 - Changer speed via une interface publique

Bonne cohésion

- Module effectue des **actions**,
 - chacune ayant son **propre point d'entrée**,
 - avec du **code indépendant** pour chaque action,
 - toutes effectuées sur la **même structure de donnée**
- Type de donnée abstrait
 - Inné au bon usage du paradigme orienté-objet

```
class EmployeeData {  
    String name;  
    int eId;  
    String position;  
    double salary;  
}
```

```
class EmployeeManager {  
    HashMap<int, EmployeeData> data;  
    void add(EmployeeData ed) {  
        data.put(ed.eid, ed);  
    }  
    void remove(EmployeeData ed) {  
        data.remove(ed.eid);  
    }  
    void update(EmployeeData ed) {  
        if (data.get(ed.eid) != null)  
            data.replace(ed.eid, ed);  
        else throw Exception();  
    }  
}
```

Bon couplage

- Ne pas baser la dépendance sur des structures complexes mais sur des paramètres homogènes
 - Simples paramètres ou structures de données dans lesquelles tous les éléments sont utilisé par le module invocateur
- Exemples
 - `afficherHeureArrivée(noVol);`
 - `Multiplication(no1, no2);`
 - `chercherTâchePrioritaire(fileTâches);`
- Conséquence d'un couplage fort entre modules M et N
 - Changer M nécessite de changer N
 - Si le changement dans N n'est pas fait, il y aura des fautes dans M

Grouper par couche ou fonctionnalité

- Paquets par fonctionnalité
 - com.app.doctor
 - com.app.drug
 - com.app.patient
 - com.app.prescription
 - com.app.report
 - com.app.security
 - com.app.webmaster
 - com.app.util
- Paquets par couche
 - com.app.action:
DoctorAction, DrugAction,
PatientAction
 - com.app.model:
Doctor, Drug, Patient
 - com.app.dao:
DoctorDAO, DrugDAO,
PatientDAO
 - com.app.util

Chaque paquet a la même structure:

- DoctorAction.java:
objet d'action ou de contrôle
- Doctor.java:
objet de modèle
- DoctorDAO.java:
objet d'accès aux données
- Items de base de donnée (SQL)
- Items du GUI (Javascript)

QUESTION

Laquelle de ces deux décompositions est-elle meilleure : par fonctionnalités ou par couches ?

- Par couche: chaque implémentation est étalée à travers plusieurs paquets/réertoires qui représentent des « préoccupations d'implémentation ». Chaque paquet contient des modules qui ne sont pas fortement reliés entre eux
 - Faible cohésion
 - Fort couplage entre paquets
 - Peu modulaire: composants ne peuvent pas être réutilisés tels quels dans d'autres systèmes
 - Modifier une fonctionnalité se fait à plusieurs endroits (ubiquitous changes)
 - Meilleure réutilisation et maintenance de l'architecture ; ex: remplacer la base de donnée par un service sur le cloud

QUESTION

Laquelle de ces deux décompositions est-elle meilleure : par fonctionnalités ou par couches ?

- Par fonctionnalité: Forte cohésion, bonne modularité, couplage faible entre les paquets
 - Facilite la compréhension et navigation dans le code
 - Séparation entre couche présente, mais en séparant les classes
 - Plus proche du niveau d'abstraction des objets du monde perceptible (bonne conception OO)
 - Meilleure réutilisation et maintenance des éléments du domaine ; ex: ajouter des infirmières

Propriétés d'une conception de bonne qualité

- **Faible couplage et forte cohésion**
- **Réutilisation**
 - Réutilisation d'un composant d'un produit pour faciliter le développement d'un autre produit ayant des fonctionnalités différentes
- **Conception modulaire**
 - Décomposition en modules indépendants et interchangeables qui contiennent tout ce qui leur est nécessaire pour exécuter un aspect d'une fonctionnalité
 - Modules cohésifs faiblement couplés
 - Modules réutilisables
- **Maintenabilité**
 - Minimiser l'effort d'apporter des modifications au logiciel après le développement initial

Propriétés d'une mauvaise conception

- **Rigidité**
 - Logiciel difficile à modifier car chaque changement impacte beaucoup trop de parties du système
 - Diminuer le couplage
- **Fragilité**
 - Quand on effectue une modification, des parties imprévues du système ne fonctionnent plus
 - Construire des modules indépendants
- **Immobilité**
 - Difficile de réutiliser un composant dans une autre application car on ne peut la démêler de l'application courante
 - Conception et programmation modulaire

Réutilisation logiciel

Réutilisation logiciel

- Réutilisation **opportuniste** (accidentelle)
 1. Construction du système
 2. Durant le développement, des parties du systèmes sont identifiées comme réutilisables et sont utilisées dans le produit
- Réutilisation **systématique** (délibérée)
 1. Construction de composants réutilisables
 2. Système est construit en assemblant ces composantes
 - **Programmation orientée composants** (*Component-based design*)

Quand réutiliser ?

- A-t-on des composants **développés à l'interne** disponibles pour implémenter cette exigence ?
- Y a-t-il des composants ou librairies **en vente libre** disponibles pour implémenter cette exigence ?
- Les **interfaces** pour les composants disponibles sont-elles **compatibles** avec l'architecture du système ?

Pourquoi réutiliser ?

- Mettre les produits le plus rapidement sur le marché
 - Pas besoin de concevoir, implémenter, tester et documenter une composant réutilisé
- En moyenne, seul 15% du nouveau code contribue à un objectif original
 - En principe, 85% pourrait être standardisé et réutilisé
 - En pratique, le taux de réutilisation ne dépasse pas les 40%
- La réutilisation affecte la maintenance plus que le développement

Réutilisation et maintenance

- Considérons un logiciel d'une durée de vie totale de 15 ans
- Supposons que 40% du produit consiste en des composants réutilisés d'autres produits
 - Documents de spécifications, modèles de conception, code, tests, manuels
- Est-ce que ça veut dire qu'on aura 40% de moins d'effort de dév ?
 - Intégration des composants
 - Adaptation au nouveau système ⇒ changement des composants réutilisés
 - Révision des tests d'intégration
- Supposons que 30% du produit réutilisé demeure inchangé et 10% de ce qui est réutilisé est changé
- Épargne durant la maintenance est 2x plus importante que durant le développement

Activity	Percentage of Total Cost over Product Lifetime	Percentage Savings over Product Lifetime due to Reuse
Development	33%	9.3%
Postdelivery maintenance	67%	17.9%

Réutilisation et maintenance

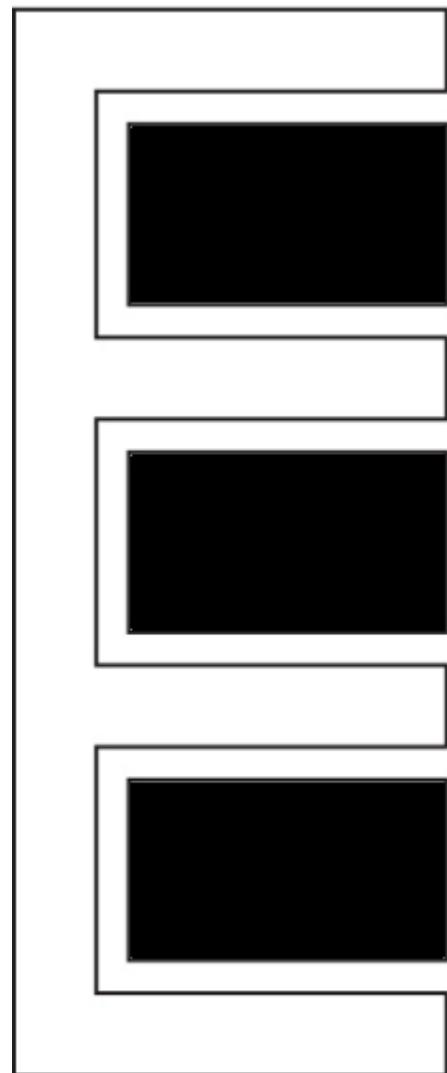
- But de la réutilisation: **faciliter la maintenance!**
- But d'une architecture et conception de bonne qualité: faciliter la maintenance
 - Extensions
 - Modifications
 - Corrections

Obstacles à la réutilisation

- Syndrome du « **pas inventé ici** »
- Crainte de **défauts** dans des routines potentiellement réutilisables
- Questions relatives à la **gestion d'une librairie** de composants réutilisables
- **Coût de la réutilisation**
 - Coût de rendre un module réutilisable
 - Coût de le réutiliser
 - Coût de définir et d'implanter un processus de réutilisation
- Problèmes **légaux**
- Manque **d'ouverture du code source** des composants commerciales disponibles

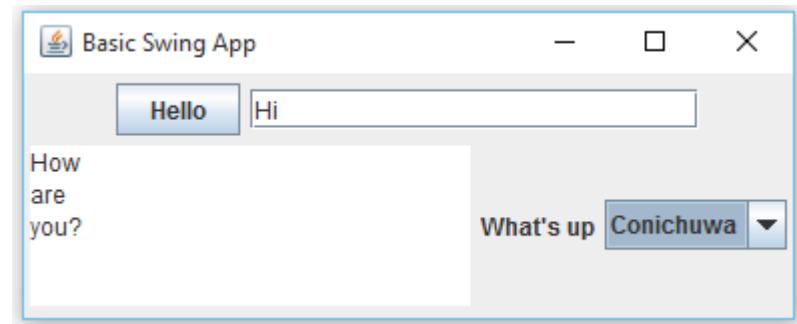
Librairie

- Ensemble de programmes/modules réutilisables
 - Ex: logiciel scientifique, librairie de classes pour le GUI
- Utilisateur de la librairie est responsable de la logique de contrôle (partie blanche)
- Une librairie fournit une API pour l'utiliser
 - Votre code fait des appels au code de la librairie



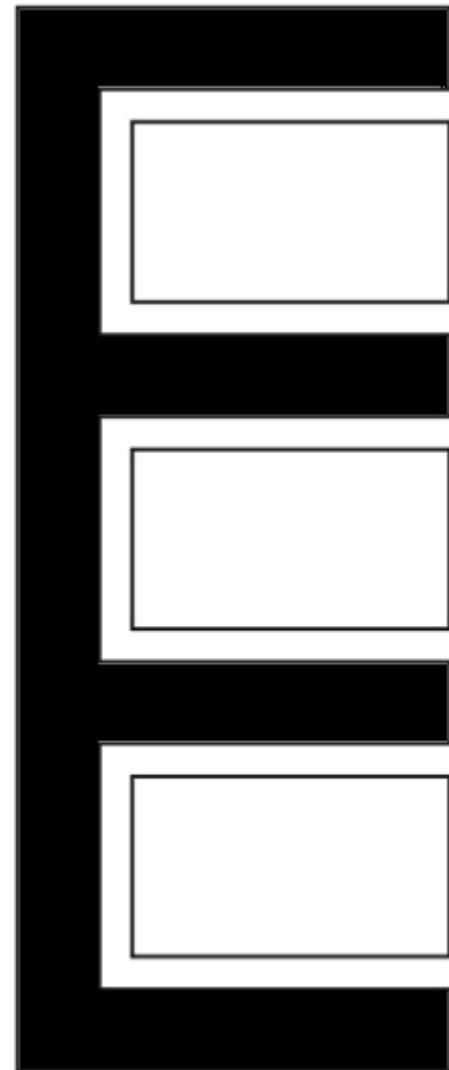
Exemple: librairie Swing

```
import javax.swing.*;  
  
public class ExempleSwing extends JFrame{  
    private static final long serialVersionUID = 1L;  
    JPanel p=new JPanel();  
    JButton b=new JButton("Hello");  
    JTextField t=new JTextField("Hi",20);  
    JTextArea ta=new JTextArea("How\\nare\\nyou?",5,20);  
    JLabel l=new JLabel("What's up");  
    String choices[]={ "Hallo", "Bonjour", "Conichuwa" };  
    JComboBox<String> cb=new JComboBox<String>(choices);  
  
    public ExempleSwing() {  
        super("Basic Swing App");  
        setSize(400,300);  
        setResizable(true);  
        p.add(b);  
        p.add(t);  
        p.add(ta);  
        p.add(l);  
        p.add(cb);  
        add(p);  
        setVisible(true);  
    }  
    public static void main(String[] args) {  
        new ExempleSwing();  
    }  
}
```



Plateforme applicative (*framework*)

- La plateforme incorpore la **logique de contrôle générique**
- Utilisateur personnalise les fonctionnalités génériques pour les besoins de l'application
 - Configuration
- Utilisateur insert des programmes spécifiques à l'application (blanc dans la figure)
 - Plug-in
- Une plateforme fournit une API à laquelle votre application doit se conformer
 - La plateforme **fait appel** à votre code



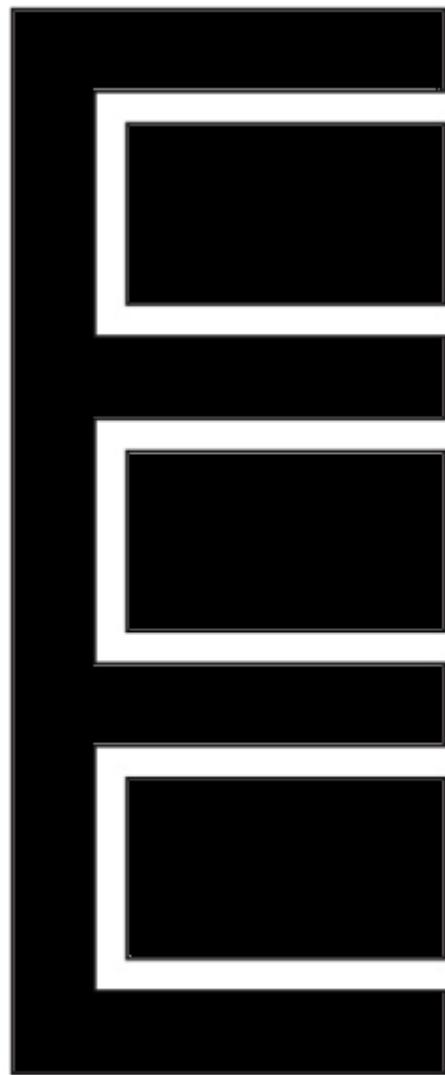
Exemple: système de gestion de contenu

- Content Management System (CMS) web
- Famille de logiciels pour concevoir et mettre à jour des sites web dynamiques
 - Workflow pour mettre en ligne le contenu
 - Opérations de gestion de la forme et du contenu
 - Structurer le contenu
 - Hiérarchiser les utilisateurs et les rôles
 - Gestion de version
- Souvent emploie une base de donnée et une architecture MVC

The screenshot shows the Joomla! Article Manager interface. The top navigation bar includes links for System, Users, Menus, Content, Components, Extensions, and Help. The main title is "Article Manager: Articles". Below the title is a toolbar with buttons for New, Edit, Publish, Unpublish, Featured, Archive, Check In, Trash, and Batch. A search bar and search tools are also present. The main content area displays a table of articles with columns for Status, Title, Access, Author, Language, Date, Hits, and ID. The table lists several core Joomla! modules, such as "Administrator Components", "Archive Module", "Article Categories Module", "Articles Category Module", "Australian Parks", "Authentication", and "Banner Module". Each row includes a checkbox and a star icon.

Programmation orientée composants

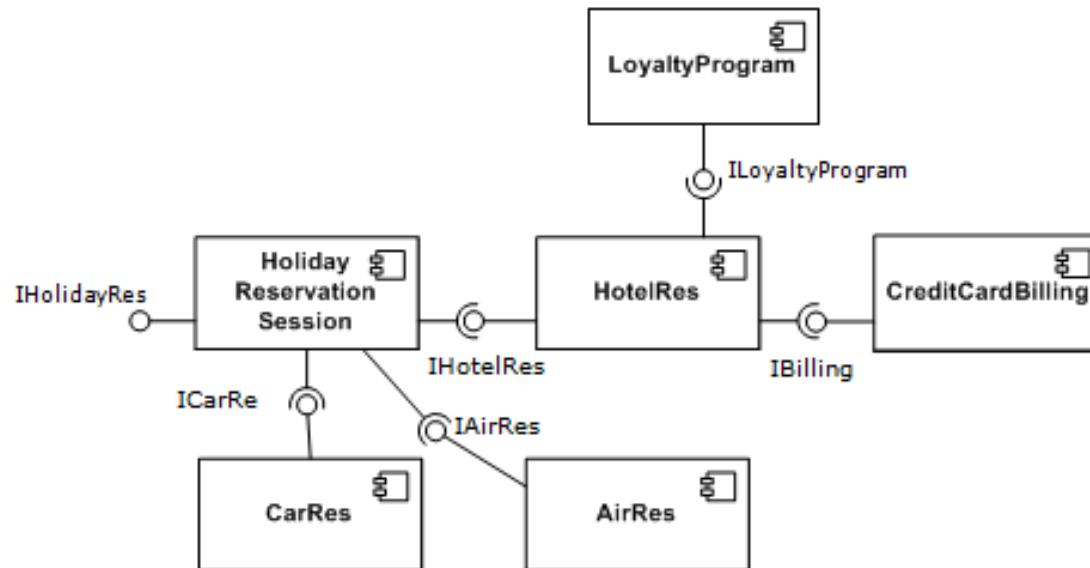
- Combine les deux approches
- Mise en place d'une plateforme applicative
- Identification des librairies réutilisables
- Tâche est d'emballer les librairies dans du code que la plateforme peut invoquer



Programmation orientée composants

Décomposition horizontale

- Concevoir en assemblant des composants fortement encapsulés avec une interface concise et rigoureuse
- But: maximiser la **réutilisation**



Choix d'un composant existant

- Son interface de programmation (API)
- Outils de développements qu'elle requiert
- Ses exigences d'exécution
 - Utilisation des ressources (mémoire, disque)
 - Vitesse et temps
 - Protocoles réseau
- Dépendance d'autres composants
- Hypothèses de conception
- Caractéristiques de sécurité
 - Contrôle d'accès
 - Protocoles d'authentification
- Gestion des erreurs



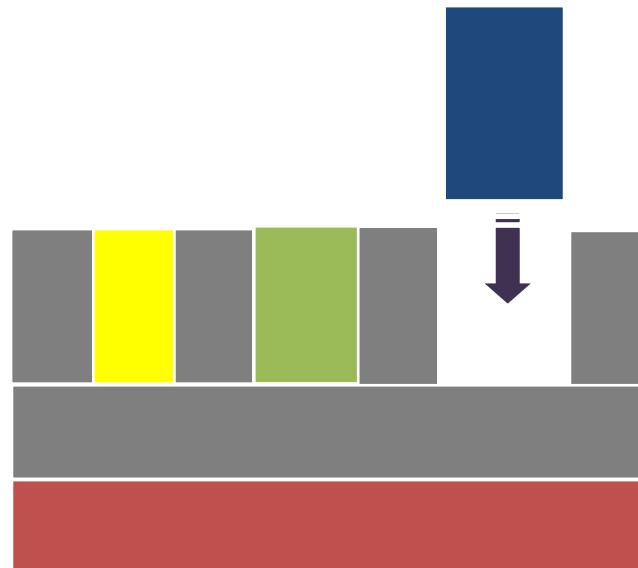
Intégration de composants

- Quelle est la difficulté d'**intégrer** ce composant dans le système actuel
 - **Adaptation** entre système et nouveau composant
- **Interface** avec l'architecture et l'environnement externe
- **Échange des données** compatible entre les composants
- Déterminer les **activités communes** à plusieurs composants
 - Manipulation et gestion des données
- Méthode de gestion et accès aux **ressources matérielles** cohérente pour tous les composants de la librairie

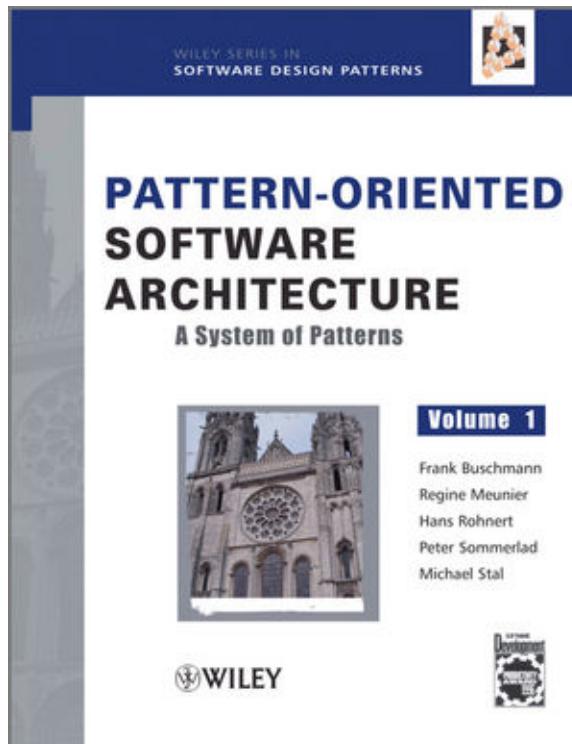


Prévoir de nouvelles intégrations

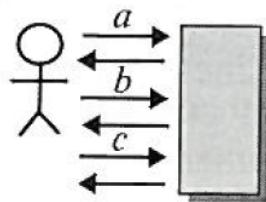
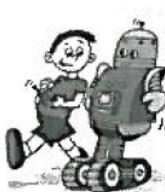
- Introduction d'un composant qui encapsule l'implémentation d'une **fonctionnalité** ou d'un **service**
- Système doit offrir une série de **points d'ancrage** qui permettent d'adapter le système et le faire évoluer
- **Minimiser** les modifications au système



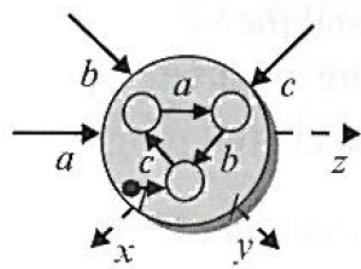
Styles d'architecture



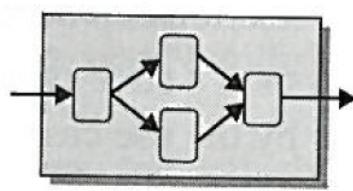
Différents types de systèmes



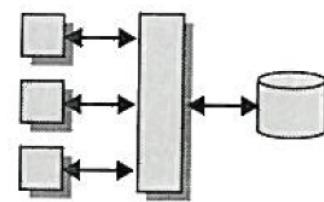
(a) Interactive subsystem



(b) Event-driven subsystem



(c) Transformational subsystem



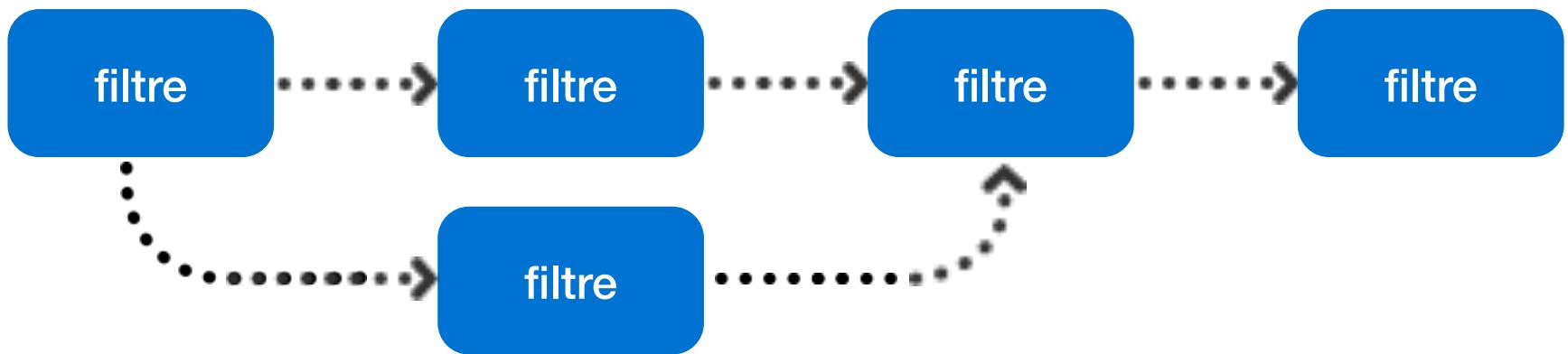
(d) Database subsystem

FIGURE 6.2 Four types of systems and behavior

Systèmes hybrides

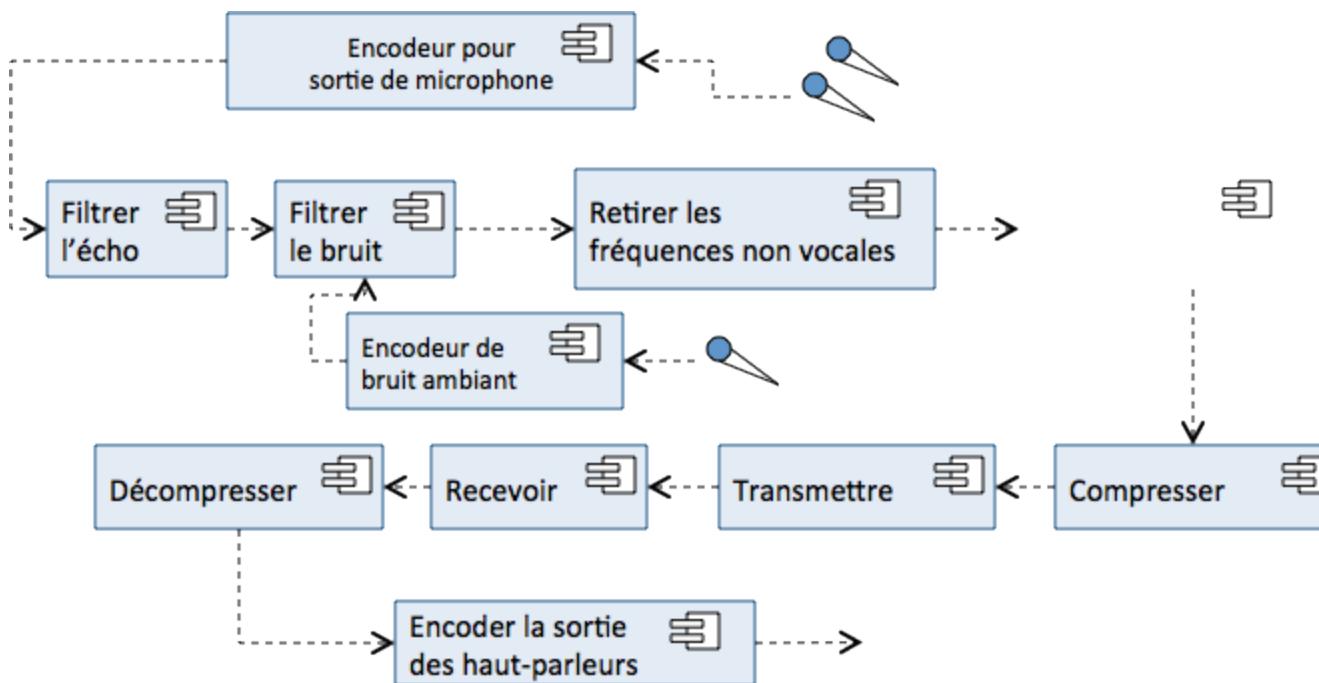
- Le type du système complet est généralement une combinaison des différents types de ses sous-systèmes
- Exemple, logiciel MSG :
 - Système **interactif** globalement (application)
 - Sous-système **dirigé par événements** (mises à jour automatiques de la compagnie de crédit)
 - Sous-système **transformationnel** (calcul des fonds hebdomadaires)
 - Sous-système de **base de données** (gestion des actifs)

Pipes et filtres

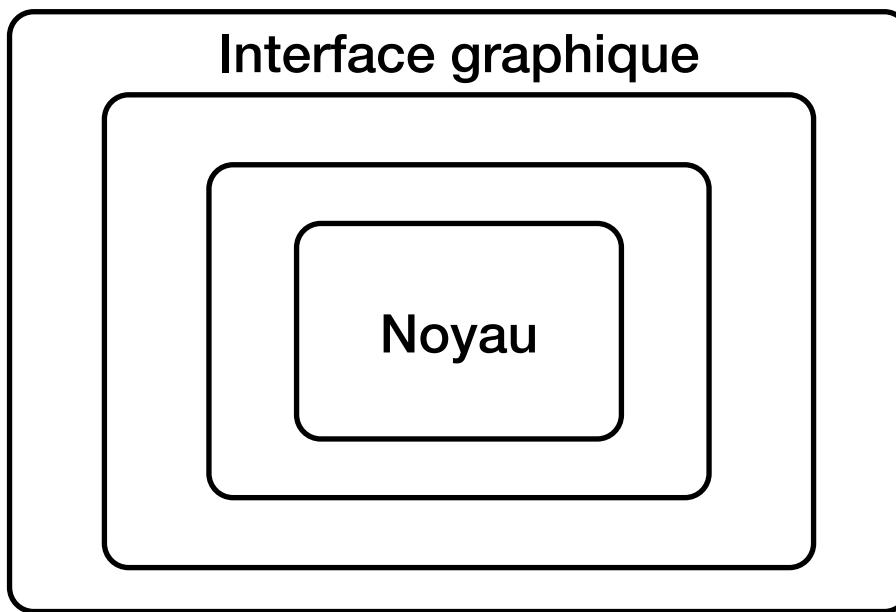


- Modules organisés en filtres communiquant via des pipes
 - Communication locale
- Traitement batch en plusieurs étapes
- Exécution concurrente de filtres, synchronisation des flux parfois nécessaire

Exemple d'architecture en pipes et filtres



Architecture en couche



- Système organisé en couches hiérarchiques
- Chaque couche fournit un service à la couche supérieur et sert de client à la couche inférieure

Exemples d'architecture en couche

Application

Présentation

Transformation des données (encryption, etc.)

Session

Identifier et authentifier une connexion

Transport

Transmission (point à point) fiable des données

Réseau

Transmission des paquets

Application

Ligne de données

Transmission des data frames sans erreurs

Swing

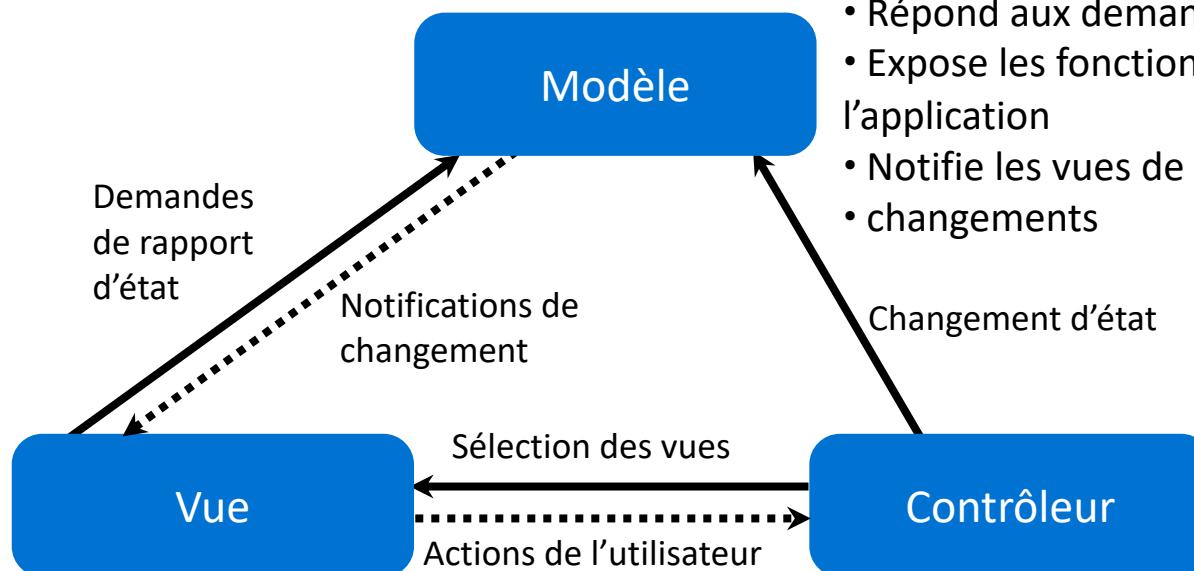
Interface matérielle du réseau

AWT

Physique

X11

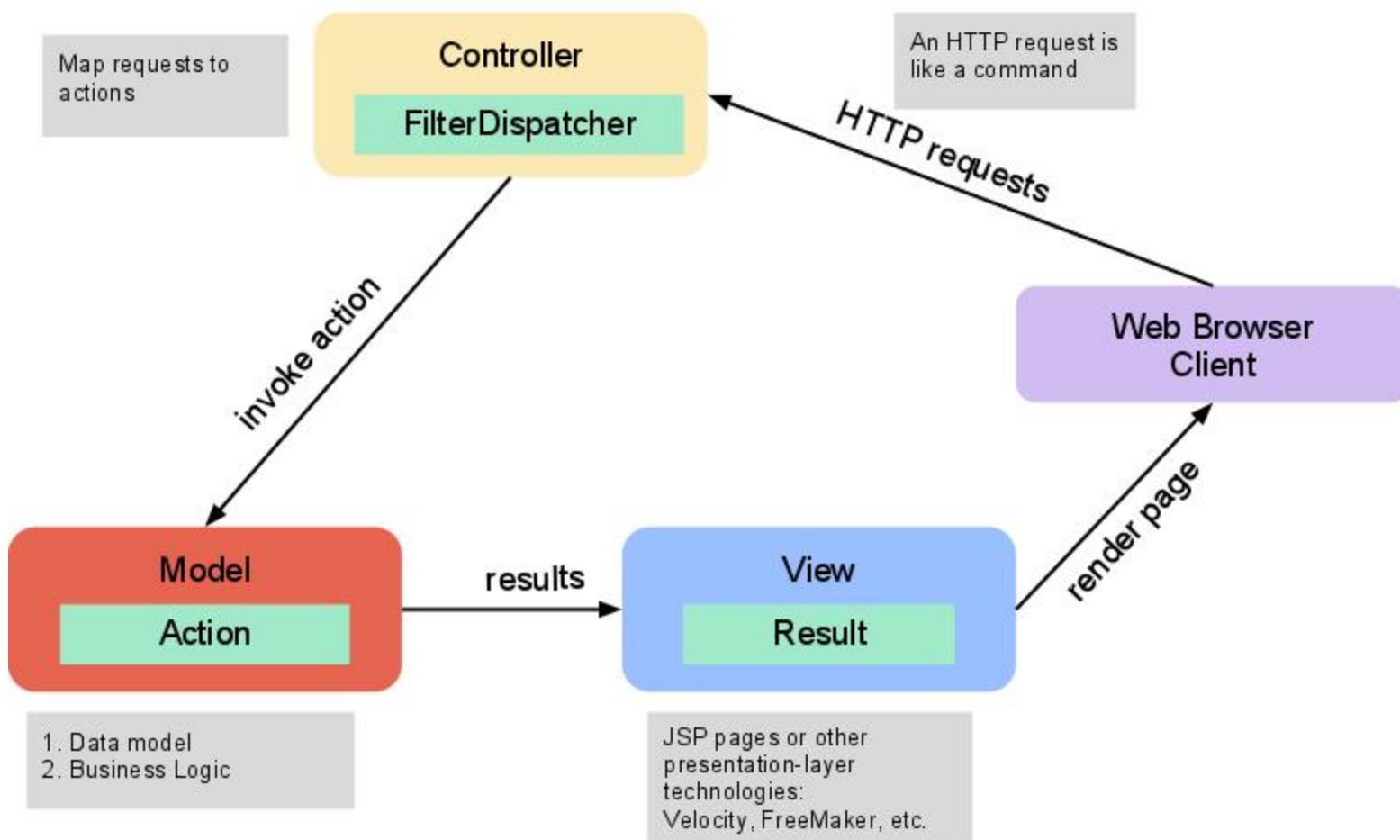
Modèle-vue-contrôleur (MVC)



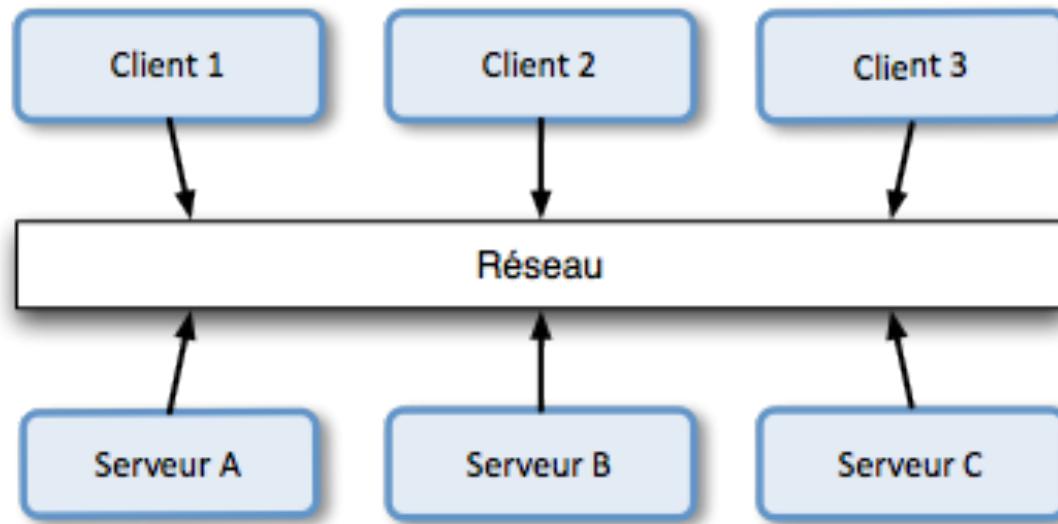
- Fait le rendu du modèle
- Demande des mises à jour du modèle
- Envoie des actions de l'utilisateur au contrôleur
- Permet au contrôleur de sélectionner une vue

- Définit le comportement de l'application
- Mappe les actions de l'utilisateur aux mises à jour du modèle
- Sélectionne une vue pour la réponse

Exemple MVC

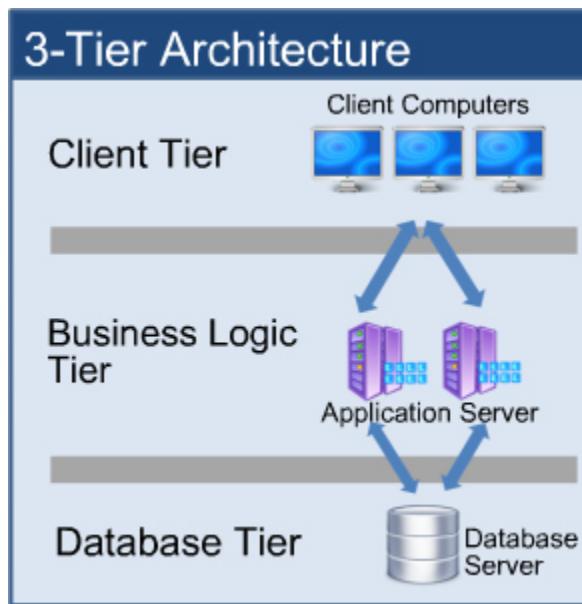


Architecture client-serveur



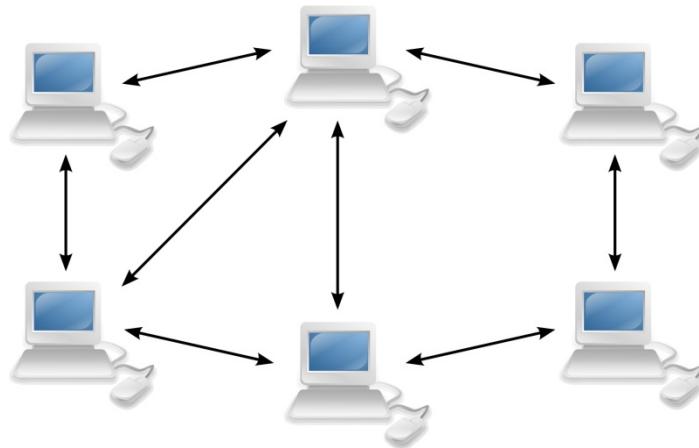
- Clients
 - Reçoivent des services des serveurs
 - Doivent connaître comment contacter les serveurs
 - Ex: adresse IP, port, etc.
- Serveurs
 - Fournissent des services aux clients et autres serveurs
- Extensibilité par ajout de serveurs

Architecture 3-tier



- Client léger: présentation (ex: HTML + JavaScript)
- Logique d'application (ex: PHP, Java)
- Couche des données (ex: SQL)
- Architecture multi-tier

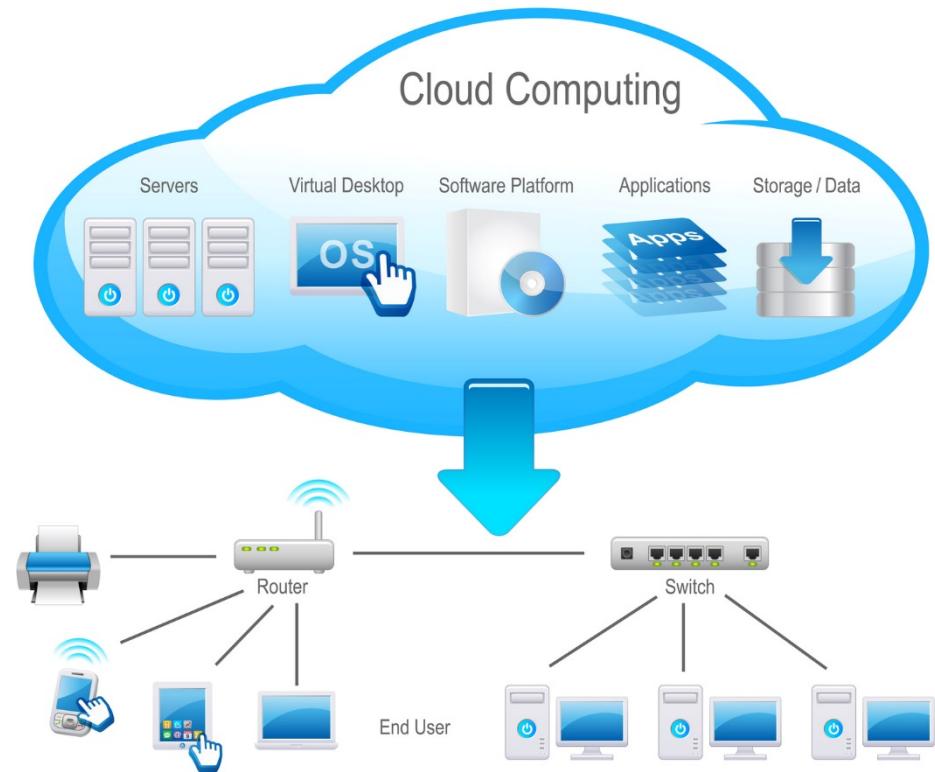
Architecture peer-to-peer



- Chaque nœud joue le rôle à la fois du client et du serveur
 - Aucun serveur central
- Modèle hybride : un serveur central conserve l'information au sujet des pairs
- Avantages
 - fiabilité (tolérance aux pannes)
 - mise à l'échelle facile (grandit avec le nombre de pairs)
- Inconvénients
 - complexité, rôle plus lourd des pairs

Cloud computing (micro-services)

- Serveur est géré par fournisseur de service, accessible par internet
 - Ex: Amazon Web Services
- Avantages
 - Facilité (sécurité, fiabilité)
 - Mise à l'échelle
- Inconvénient
 - Contrôle limité



Architecte logiciel

- Concevoir l'architecture requiert une personne spécialisée:
l'architecte logiciel
- Doit faire des **compromis**
 - La conception doit satisfaire ses **exigences fonctionnelles** (CU)
 - L'architecture doit satisfaire les **exigences non-fonctionnelles**
 - Le tout dans les **contraintes** du budget et des échéanciers
- L'architecte doit **conseiller** le client en exposant les compromis à faire

Compromis du client

- Souvent impossible de satisfaire toutes les exigences et contraintes
- Client a le choix de
 - Revoir certaines exigences
 - Augmenter le budget
 - Repousser la date de livraison

Importance de l'architecture

- L'architecture du logiciel est cruciale au produit
 - Le workflow des exigences peut être corrigé durant le workflow d'analyse
 - Le workflow d'analyse peut être corrigé durant le workflow de conception
 - Le workflow de conception peut être corrigé durant le workflow d'implémentation
- Mais il n'y a pas moyen de surmonter une architecture non-optimale plus tard
 - Il faut absolument reconcevoir l'architecture immédiatement !