

TP INF 4027 : GÉNIE LOGICIEL

GROUPE 6 : ARCHITECTURE DES LOGICIELS

I. Principes de la conception architecturale

La conception du système est le processus de création d'une conception permettant au système de répondre aux exigences. La conception du système est le processus de conception de l'architecture, des composants, des modules, des interfaces et des données d'un système afin de satisfaire aux exigences spécifiées. Cela implique la conception de l'architecture du système, des composants, des modules, des interfaces et des données.

pour une bonne conception il faut respecter certains points tels que :

- *la compréhension* la problématique de recherche
- Identification des **éléments** de conception et leurs relations
- Évaluation la conception architectural
- Transformation la conception de l'architecture

Ainsi une conception architecturale obei a plusieurs principes a savoir :

a-Séparation des responsabilités :

Le principe de séparation des responsabilités vise à organiser un logiciel en plusieurs sous-parties, chacune ayant une responsabilité bien définie.

Ainsi construite de manière modulaire, l'application sera plus facile à comprendre et à faire évoluer. Au moment où un nouveau besoin se fera sentir, il suffira d'intervenir sur la ou les sous-partie(s) concernée(s). Le reste de l'application sera inchangée : cela limite les tests à effectuer et le risque d'erreur. Une construction modulaire encourage également la réutilisation de certaines parties de l'application.

B-Réutilisation :

Un bâtiment s'édifie à partir de morceaux de bases, par exemple des briques ou des moellons. De la même manière, une carte mère est conçue par assemblage de composants électroniques.

Longtemps, l'informatique a gardé un côté artisanal : chaque programmeur recréait la roue dans son coin pour les besoins de son projet. Mais nous sommes passés depuis plusieurs

années à une ère industrielle. Des logiciels de plus en plus complexes doivent être réalisés dans des délais de plus en plus courts, tout en maintenant le meilleur niveau de qualité possible. Une réponse à ces exigences contradictoires passe par la réutilisation de briques logicielles de base appelées librairies, modules ou plus généralement composants.

En particulier, la mise à disposition de milliers de projets *open source* via des plateformes comme Git hub ou des outils comme Nuget , composer ou nmp a permis aux équipes de développement de faire des gains de productivité remarquables en intégrant ces composants lors de la conception de leurs applications. A l'heure actuelle, il n'est pas de logiciel de taille significative qui n'intègre plusieurs dizaines, voire des centaines de composants externes.

Déjà testé et éprouvé, un composant logiciel fait simultanément baisser le temps et augmenter la qualité du développement. Il permet de limiter les efforts nécessaires pour traiter les problématiques *techniques* afin de se concentrer sur les problématiques *métier*, celles qui sont en lien direct avec ses fonctionnalités essentielles.

c-Encapsulation maximale :

Ce principe de conception recommande de n'exposer au reste de l'application que le strict nécessaire pour que la sous-partie joue son rôle. Au niveau d'une classe, cela consiste à ne donner le niveau d'accessibilité publique qu'à un nombre minimal de membres, qui seront le plus souvent des méthodes.

Au niveau d'une sous-partie d'application composée de plusieurs classes, cela consiste à rendre certaines classes privées afin d'interdire leur utilisation par le reste de l'application.

d-Couplage faible :

La définition du couplage est la suivante : "une entité (sous-partie, composant, classe, méthode) est couplée à une autre si elle dépend d'elle", autrement dit, si elle a besoin d'elle pour fonctionner. Plus une classe ou une méthode utilise d'autres classes comme classes de base, attributs, paramètres ou variables locales, plus son couplage avec ces classes augmente.

Au sein d'une application, un couplage fort tisse entre ses éléments des liens puissants qui la rend plus rigide à toute modification (on parle de "code spaghetti"). A l'inverse, un couplage faible permet une grande souplesse de mise à jour. Un élément peut être modifié (exemple : changement de la signature d'une méthode publique) en limitant ses impacts. Raison pour laquelle il serait préférable de privilégier un couplage faible.

e- Cohésion forte :

Ce principe recommande de placer ensemble des éléments (composants, classes, méthodes) ayant des rôles similaires ou dédiés à une même problématique. Inversement, il déconseille de rassembler des éléments ayant des rôles différents.

Exemple : ajouter une méthode de calcul métier au sein d'un composant lié aux données (ou à la présentation) est contraire au principe de cohésion forte.

f- DRY :

DRY est l'acronyme de **Don't Repeat Yourself**. Ce principe vise à éviter la redondance au travers de l'ensemble de l'application. Cette redondance est en effet l'un des principaux ennemis du développeur. Elle a les conséquences néfastes suivantes :

- Augmentation du volume de code.
- Diminution de sa lisibilité.
- Risque d'apparition de bogues dûs à des modifications incomplètes.

La redondance peut se présenter à plusieurs endroits d'une application, parfois de manière inévitable (réutilisation d'un existant). Elle prend souvent la forme d'un ensemble de lignes de code dupliquées à plusieurs endroits pour répondre au même besoin.

Le principe DRY est important mais ne doit pas être appliqué de manière trop zélée. Vouloir absolument éliminer toute forme de redondance conduit parfois à créer des applications inutilement génériques et complexes. C'est l'objet des deux prochains principes.

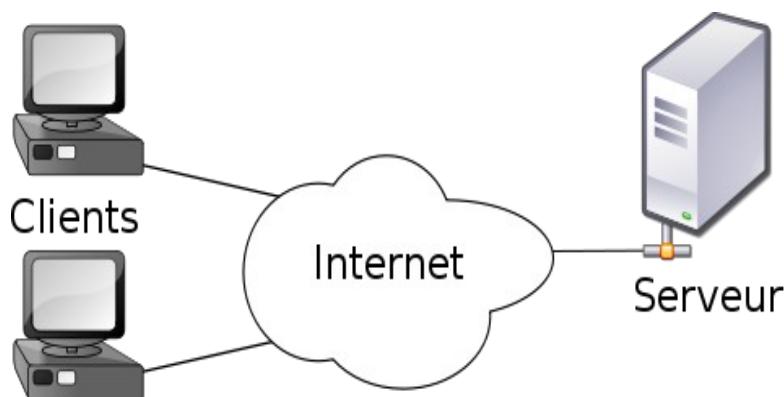
g- KISS :

KISS est un autre acronyme signifiant **Keep It Simple, Stupid** et qu'on peut traduire par "Ne complique pas les choses". Ce principe vise à privilégier autant que possible la simplicité lors de la construction d'une application.

1- MODÈLES D'ARCHITECTURE COURANTS :

- Architecture client/serveur :

L'architecture client/serveur caractérise un système basé sur des échanges réseau entre des clients et un serveur centralisé, lieu de stockage des données de l'application.



Le principal avantage de l'architecture client/serveur tient à la centralisation des données. Stockées à un seul endroit, elles sont plus faciles à sauvegarder et à sécuriser. Le serveur qui les héberge peut être dimensionné pour pouvoir héberger le volume de données nécessaire et répondre aux sollicitations de nombreux clients. Cette médaille a son revers : le serveur constitue le nœud central du système et représente son maillon faible. En cas de défaillance (surcharge, indisponibilité, problème réseau), les clients ne peuvent plus fonctionner.

On peut classer les clients d'une architecture client/serveur en plusieurs types :

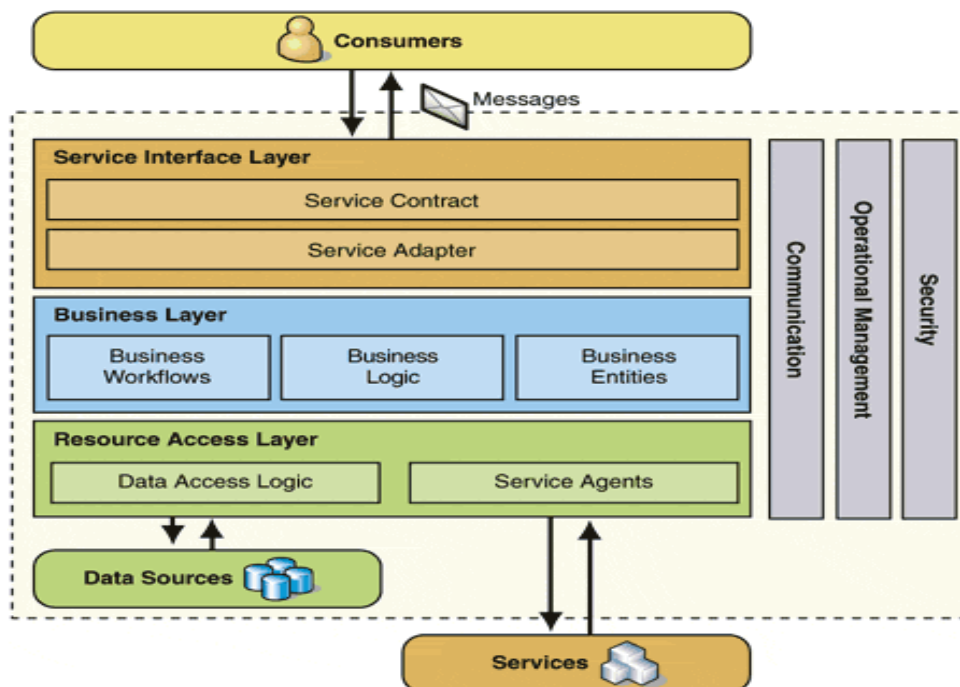
Client léger, destiné uniquement à l'affichage (exemple : navigateur web).

Client lourd, application native spécialement conçue pour communiquer avec le serveur (exemple : application mobile).

Client riche combinant les avantages des clients légers et lourds (exemple : navigateur web utilisant des technologies évoluées pour offrir une expérience utilisateur proche de celle d'une application native).

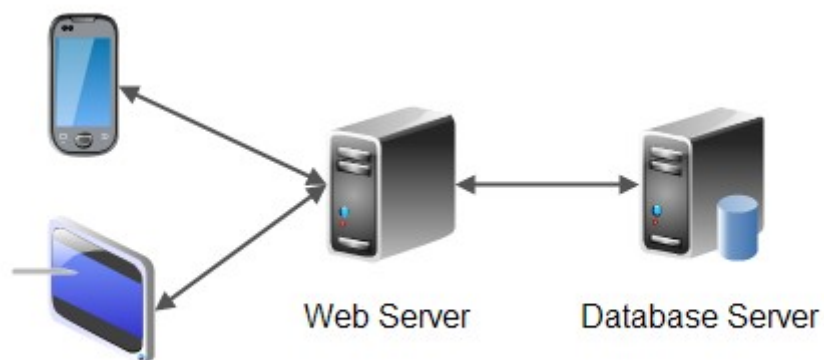
Le fonctionnement en mode client/serveur est très souvent utilisé en informatique. Un réseau Windows organisé en domaine, la consultation d'une page hébergée par un serveur Web ou le téléchargement d'une application mobile depuis un magasin central (App Store, Google Play) en constituent des exemples.

-Architecture en couches :



Cette architecture respecte le principe de séparation des responsabilités et facilite la compréhension des échanges au sein de l'application. Lorsque chaque couche correspond à un processus distinct sur une machine, on parle d'architecture **n-tiers**, n désignant le nombre de couches.

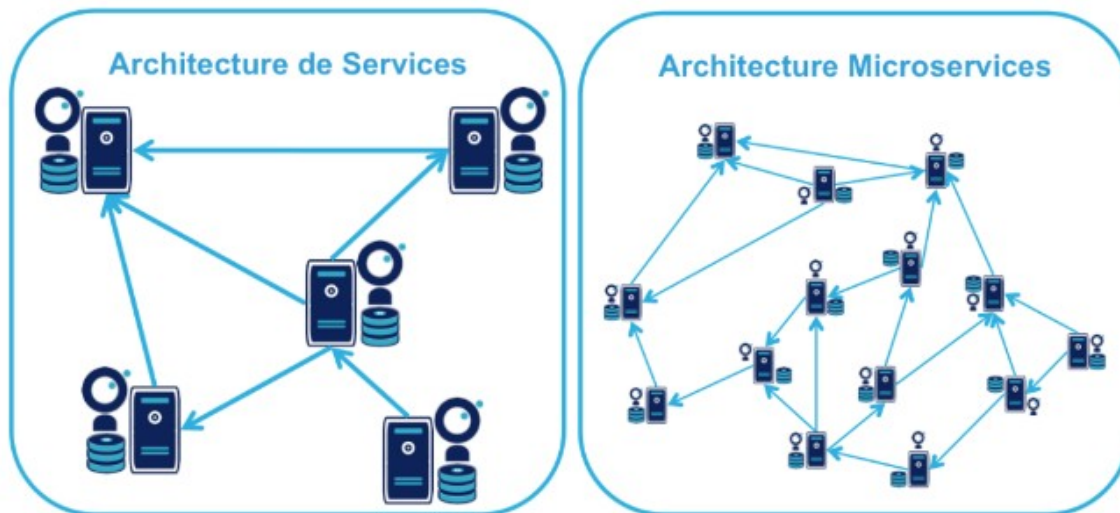
Un navigateur web accédant à des pages dynamiques intégrant des informations stockées dans une base de données constitue un exemple classique d'architecture 3-tiers.



-Architecture orientée services :

Une architecture orientée services (SOA, *Service-Oriented Architecture*) décompose un logiciel sous la forme d'un ensemble de services métier utilisant un format d'échange commun, généralement XML ou JSON.

Une variante récente, l'architecture microservices, diminue la granularité des services pour leur assurer souplesse et capacité à évoluer, au prix d'une plus grande distribution du système. L'image ci-dessous illustre la différence entre ces deux approches :



3-PATRONS DE CONCEPTION ARCHITECTURAL

- *Architecture Modèle-Vue-Contrôleur* :

La patron Modèle-Vue-Contrôleur, ou **MVC**, décompose une application en trois sous-parties :

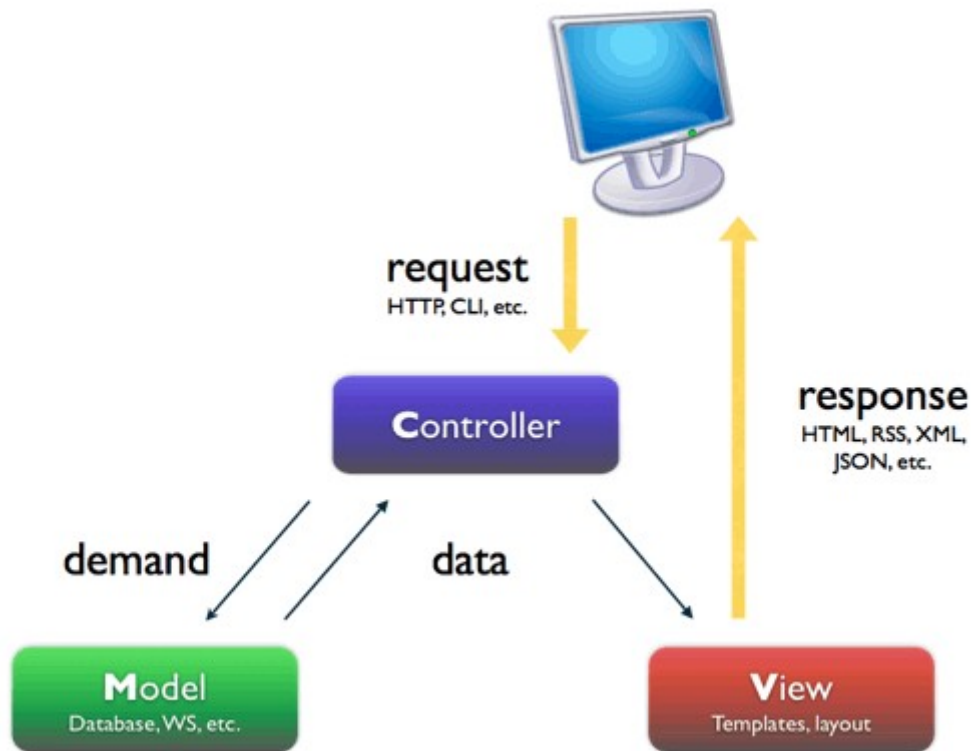
La partie **Modèle** qui regroupe la logique métier ("business logic") ainsi que l'accès aux données. Il peut s'agir d'un ensemble de fonctions (Modèle procédural) ou de classes (Modèle orienté objet). ;

La partie **Vue** qui s'occupe des interactions avec l'utilisateur : présentation, saisie et validation des données ;

La partie **Contrôleur** qui gère la dynamique de l'application. Elle fait le lien entre les deux autres parties.

Ce patron a été imaginé à la fin des années 1970 pour le langage Smalltalk afin de bien séparer le code de l'interface graphique de la logique applicative. On le retrouve dans de très nombreux langages : bibliothèques Swing et Model 2 (JSP) de Java, frameworks PHP, ASP.NET MVC, etc.

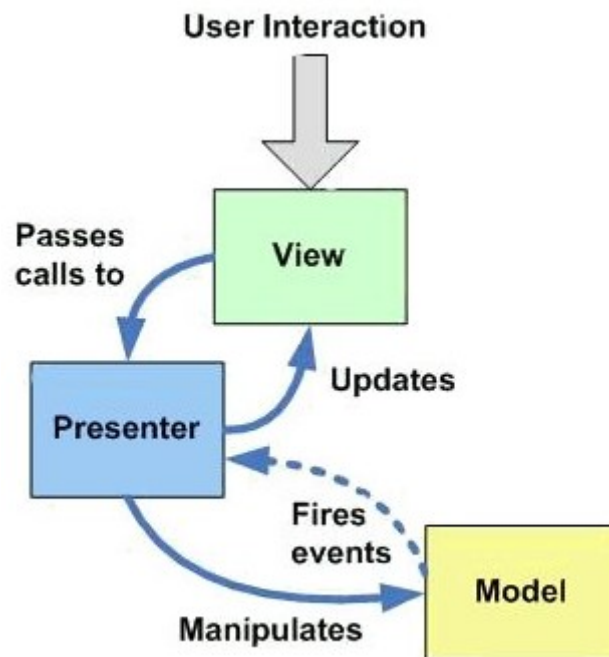
Le diagramme ci-dessous résume les relations entre les composants d'une architecture web MVC :



-Architecture Modèle-Vue-Présentation :

La patron Modèle-Vue-Présentation, ou **MVP**, est un proche cousin du patron MVC surtout utilisé pour construire des interfaces utilisateurs (UI).

Dans une architecture MVP, la partie Vue reçoit les évènements provenant de l'utilisateur et délègue leur gestion à la partie Présentation. Celle-ci utilise les services de la partie Modèle puis met à jour la Vue.

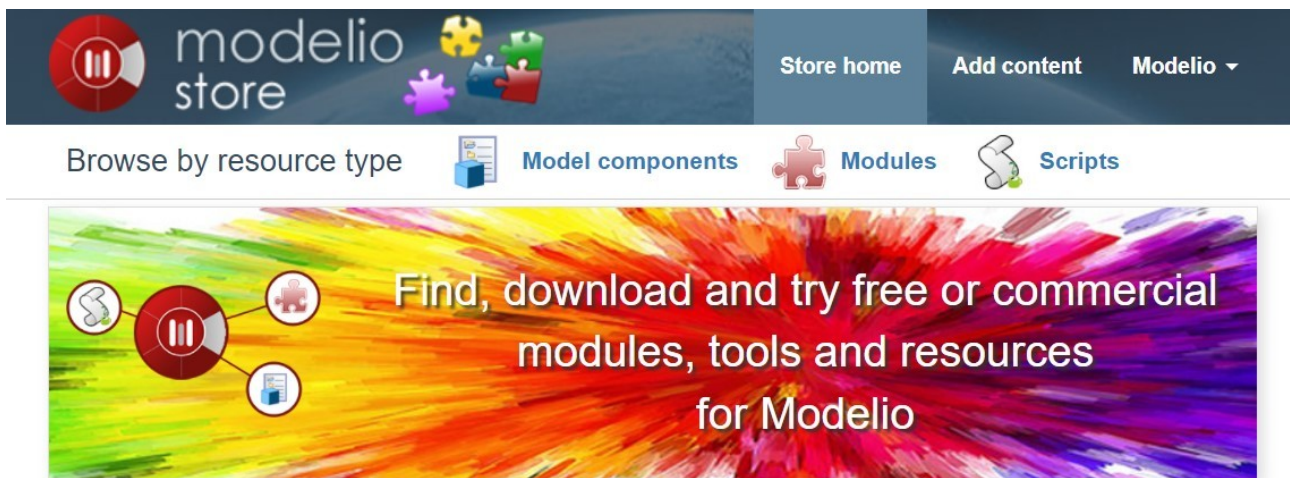


Dans la variante dite *Passive View* de cette architecture, la Vue est passive et dépend totalement du contrôleur pour ses mises à jour. Dans la variante dite *Supervising Controller*, Vue et Modèle sont couplées et les modifications du Modèle déclenchent la mise à jour de la Vue.

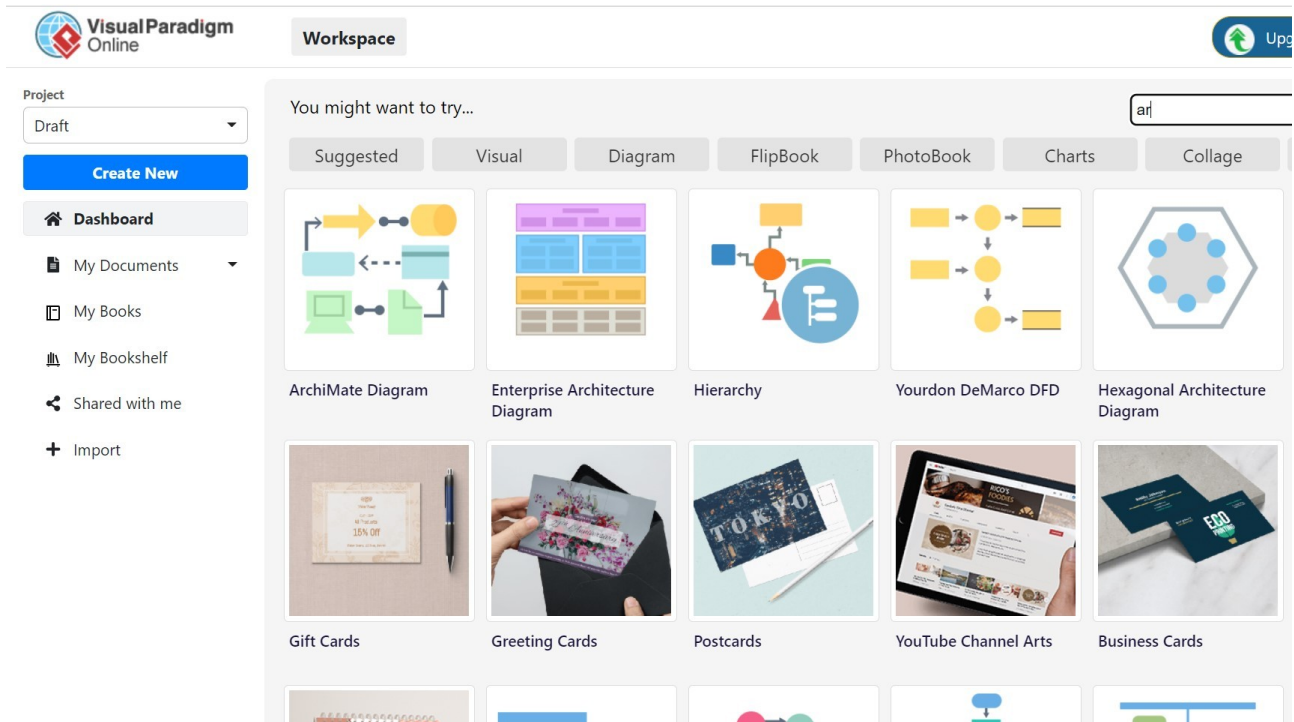
II.MODELISATION ARCHITECTURALE

a-OUTILS DE MODÉLISATION ARCHITECTURALE

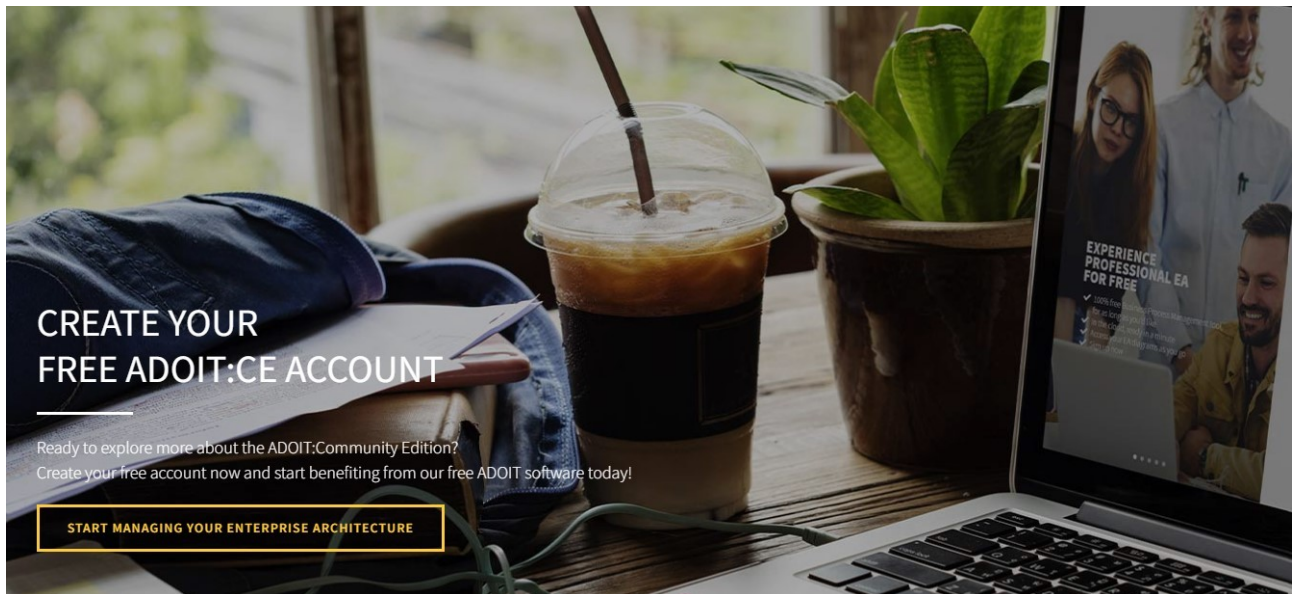
- **Modelio** : le seul outil open source fonctionnant sur Windows, Linux, Mac OS, supportant ArchiMate, TOGAF, BPMN, UML, SysML, MDA et offrant, pour tous ces artefacts de modélisation, un référentiel commun assurant la traçabilité depuis la stratégie jusqu'à la couche technologique. Modelio offre une étude de cas exhaustive illustrant l'architecture d'une agence de voyages fictive. L'éditeur Softeam, société de conseils et de services, est français, c'est une filiale de Docaposte.



-**Visual Paradigm (online)** : sa gratuité, sa facilité d'installation en mode SaaS ne demandant aucune connaissance technique, la possibilité de partager son écran à des coéquipiers et les nombreux exemples ArchiMate, BPMN, UML, prêts à être adaptés, font de Visual Paradigm un excellent outil pédagogique pour mettre en pratique l'art de la modélisation.



- **ADOIT:CE** : rien à installer, toujours la dernière version en ligne, interface utilisateur relativement simple, support complet d'ArchiMate, 35 vues différentes proposées, nombreuses fonctionnalités, gestion de portefeuille d'application, Master Data Management, un contenu de démonstration de l'architecture d'une banque fictive, etc.



- **Archi (archimatetool)** : convient parfaitement au consultant pour des missions ponctuelles. L'absence de mode collaboratif finalisé, nécessitant des installations locales aussi simples soient-elles, prive Archi de pouvoir être utilisé dans une équipe d'architectes d'entreprise. Alors que Modelio, son concurrent direct comme outil open source de modélisation pour l'Architecture d'Entreprise, est en français, supporte en plus d'ArchiMate, BPMN, UML, SysML, MDA avec un référentiel commun et fourni une étude de cas pédagogique, Archi n'intègre qu'ArchiMate et donc n'est pas archi complet.




- **GenMyModel** : est une plate-forme française de conception logicielle basée sur les normes UML pour la modélisation de systèmes, BPMN pour les processus d'entreprise, DMN pour les règles métier et AchiMate pour l'Architecture d'Entreprise. Complètement accessible dans le cloud en mode SaaS, il fournit des générateurs de code intégrés pour Java, SQL, Spring...


 **GenMyModel**
by Axellience

[Des produits](#) [Assistance/Contact](#) [À propos de](#)


La modélisation en ligne simplifiée


ArchiMate®



Cartographie du trajet



La modélisation des données


BPMN


UML


Organigramme

1 210 622
UTILISATEURS


1 454 059
DES MODÈLES


+180
DES PAYS
