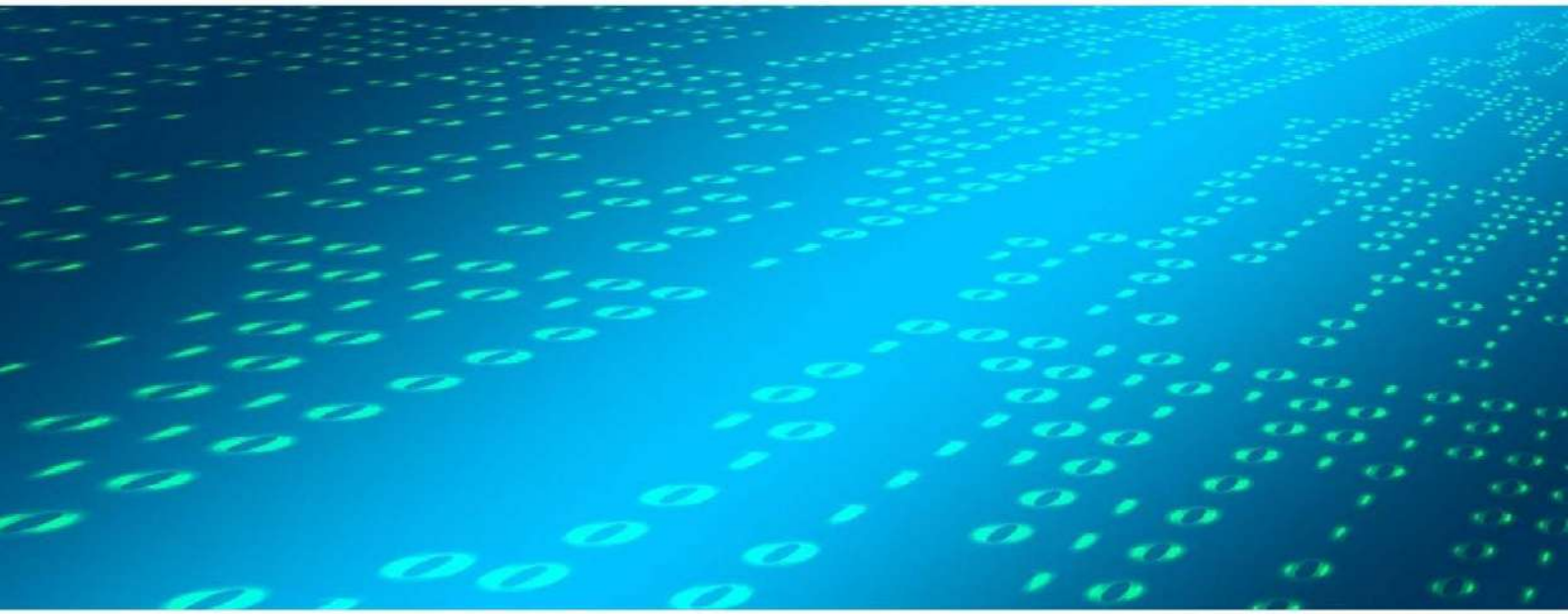
Bordeaux INP
ENSC

Génie Logiciel



Baptiste Pesquet

Table des matières

Génie logiciel

Ce livre est un support de cours à l'Ecole Nationale Supérieure de Cognitique.

Résumé

Ce livre constitue une introduction au **génie logiciel**. Il présente les grands enjeux et les bonnes pratiques liés à l'activité de réalisation de logiciels :

- Notion d'architecture logicielle.
- Principes de conception.
- Patrons logiciels.
- Production du code source.
- Gestion des versions.
- Travail collaboratif.
- Tests.
- Documentation.

> Le point de vue adopté par ce livre est essentiellement technique. Les aspects organisationnels (gestion de projet) et méthodologiques ne sont pas étudiés ici.

Compléments

Un projet écrit en langage C# et utilisant la technologie WinForms illustre certaines notions d'architecture et de test présentées dans ce livre. Son code source est disponible en ligne.

> Au besoin, consultez les livres Programmation orientée objet en C# et Programmation évènementielle avec les WinForms pour pouvoir étudier ce projet.

Contributions

Merci d'avance et bonne lecture !

Le génie logiciel

L'objectif de ce chapitre est de présenter le génie logiciel, ses enjeux et ses dimensions.

Introduction

Le **génie logiciel** (*software engineering*) représente l'application de principes d'ingénierie au domaine de la création de logiciels. Il consiste à identifier et à utiliser des **méthodes**, des **pratiques** et des **outils** permettant de maximiser les chances de réussite d'un projet logiciel.

Il s'agit d'une science récente dont l'origine remonte aux années 1970. A cette époque, l'augmentation de la puissance matérielle a permis de réaliser des logiciels plus complexes mais souffrant de nouveaux défauts : délais non respectés, coûts de production et d'entretien élevés, manque de fiabilité et de performances. Cette tendance se poursuit encore aujourd'hui.

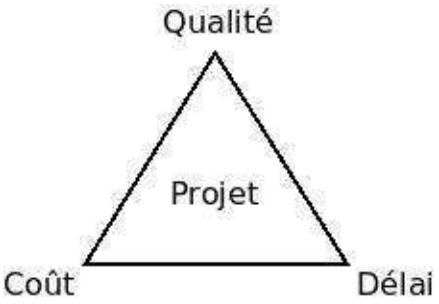
L'apparition du génie logiciel est une réponse aux défis posés par la complexification des logiciels et de l'activité qui vise à les produire.

Enjeux

Le génie logiciel vise à rationaliser et à optimiser le processus de production d'un logiciel. Les enjeux associés sont multiples :

- Adéquation aux besoins du client.
- Respect des délais de réalisation prévus.
- Maximisation des performances et de la fiabilité.
- Facilitation de la maintenance et des évolutions ultérieures.

Comme tout projet, la réalisation d'un logiciel est soumise à des exigences contradictoires et difficilement conciliables (triangle coût-délai-qualité).



La qualité d'un logiciel peut s'évaluer à partir d'un ensemble de facteurs tels que :

- Le logiciel répond-il aux besoins exprimés ?
- Le logiciel demande-t-il peu d'efforts pour évoluer aux regards de nouveaux besoins ?
- Le logiciel peut-il facilement être transféré d'une plate-forme à une autre ?
- ... (voir norme ISO 9126)

Sans être une solution miracle, le génie logiciel a pour objectif de maximiser la surface du triangle en tenant compte des priorités du client.

Dimensions

Le génie logiciel couvre l'ensemble du cycle de vie d'un logiciel. Il étudie toutes les activités qui mènent d'un besoin à la livraison du logiciel, y compris dans ses versions successives, jusqu'à sa fin de vie.

Les dimensions du génie logiciel sont donc multiples :

- Analyse des besoins du client.
- Définition de l'architecture du logiciel.
- Choix de conception.
- Règles et méthodes de production du code source.
- Gestion des versions.
- Test du logiciel.
- Documentation.
- Organisation de l'équipe et interactions avec le client.
- ...

Ce livre présente plus en détail certaines de ces activités.

Architecture logicielle

L'objectif de ce chapitre est de présenter ce qu'est l'architecture logicielle.

Définition

Le Petit Robert définit l'architecture comme étant **"l'art de construire les édifices"**. Ce mot est avant tout lié au domaine du génie civil : on pense à l'architecture d'un monument ou encore d'un pont.

Par analogie, l'architecture logicielle peut être définie comme étant **"l'art de construire les logiciels"**.

Selon le contexte, l'architecture logicielle peut désigner :

- **L'activité d'architecture**, c'est-à-dire une phase au cours de laquelle on effectue les grands choix qui vont structurer une application : langages et technologies utilisés, découpage en sous-parties, méthodologies mises en oeuvre...
- **Le résultat de cette activité**, c'est-à-dire la structure d'une l'application, son squelette.

Importance

Dans le domaine du génie civil, on n'imagine pas se lancer dans la construction d'un bâtiment sans avoir prévu son apparence, étudié ses fondations et son équilibre, choisi les matériaux utilisés, etc. Dans le cas contraire, on va au-devant de graves désillusions...

Cette problématique se retrouve dans le domaine informatique. Comme un bâtiment, un logiciel est fait pour durer dans le temps. Il est presque systématique que des projets informatiques aient une durée de vie de plusieurs années. Plus encore qu'un bâtiment, un logiciel va, tout au long de son cycle de vie, connaître de nombreuses modifications qui aboutiront à la livraison de nouvelles versions, majeures ou mineures. Les évolutions par rapport au produit initialement créé sont souvent nombreuses et très difficiles à prévoir au début du projet.

> Exemple : le logiciel VLC n'était à l'origine qu'un projet étudiant destiné à diffuser des vidéos sur le campus de l'Ecole Centrale de Paris. Sa première version remonte à l'année 2001.

Objectifs

Dans le domaine du génie civil, les objectifs de l'architecture sont que le bâtiment construit réponde aux besoins qu'il remplit, soit robuste dans le temps et (notion plus subjective) agréable à l'oeil.

L'architecture logicielle poursuit les mêmes objectifs. Le logiciel créé doit répondre aux besoins et résister aux nombreuses modifications qu'il subira au cours de son cycle de vie. Contrairement à un bâtiment, un logiciel mal pensé ne risque pas de s'effondrer. En revanche, une mauvaise architecture peut faire exploser le temps nécessaire pour réaliser les modifications, et donc leur coût.

Les deux objectifs principaux de toute architecture logicielle sont la réduction des coûts (création et maintenance) et l'augmentation de la **qualité** du logiciel.

La qualité du code source d'un logiciel peut être évaluée par un certain nombre de mesures appelées **métriques de code** : indice de maintenabilité, complexité cyclomatique, etc.

Architecture ou conception ?

Il n'existe pas de vrai consensus concernant le sens des mots "architecture" et "conception" dans le domaine du développement logiciel. Ces deux termes sont souvent employés de manière interchangeable. Il arrive aussi que l'architecture soit appelée "conception préliminaire" et la conception proprement dite "conception détaillée". Certaines méthodologies de développement incluent la définition de l'architecture dans une phase plus globale appelée "conception".

Cela dit, la distinction suivante est généralement admise et sera utilisée dans la suite de ce livre :

- **L'architecture logicielle (*software architecture*)** considère le logiciel de manière globale. Il s'agit d'une vue de haut niveau qui définit le logiciel dans ses grandes lignes : que fait-il ? Quelles sont les sous-parties qui le composent ? Interagissent-elles ? Sous quelle forme sont stockées ses données ? etc.
- **La conception logicielle (*software design*)** intervient à un niveau de granularité plus fin et permet de préciser comment fonctionne chaque sous-partie de l'application. Quel logiciel est utilisé pour stocker les données ? Comment est organisé le code ? Comment une sous-partie expose-t-elle ses fonctionnalités au reste du système ? etc.

La perspective change selon la taille du logiciel et le niveau auquel on s'intéresse à lui :

- Sur un projet de taille modeste, architecture et conception peuvent se confondre.
- A l'inverse, certaines sous-parties d'un projet de taille conséquente peuvent nécessiter en elles-mêmes un travail d'architecture qui, du point de vue de l'application globale, relève plutôt de la conception...

L'activité d'architecture

Définition

Tout logiciel, au-delà d'un niveau minimal de complexité, est un édifice qui mérite une phase de réflexion initiale pour l'imaginer dans ses grandes lignes. Cette ohase correspond à l'activité d'architecture. Au cours de cette phase, on effectue les grands choix structurant le

futur logiciel : langages, technologies, outils... Elle consiste notamment à identifier les différents éléments qui vont composer le logiciel et à organiser les interactions entre ces éléments.

Selon le niveau de complexité du logiciel, l'activité d'architecture peut être une simple formalité ou bien un travail de longue haleine.

L'activité d'architecture peut donner lieu à la production de **diagrammes** représentant les éléments et leurs interactions selon différents formalismes, par exemple UML.

Place dans le processus de création

L'activité d'architecture intervient traditionnellement vers le début d'un projet logiciel, dès le moment où les besoins auxquels le logiciel doit répondre sont suffisamment identifiés. Elle est presque toujours suivie par une phase de conception.

Les évolutions d'un projet logiciel peuvent nécessiter de nouvelles phases d'architecture tout au long de sa vie. C'est notamment le cas avec certaines méthodologies de développement itératif ou agile, où des phases d'architecture souvent brèves alternent avec des phases de production, de test et de livraison.

Répartition des problématiques

De manière très générale, un logiciel sert à automatiser des traitements sur des données. Toute application informatique est donc confrontée à trois problématiques :

- Gérer les interactions avec l'extérieur, en particulier l'utilisateur : saisie et contrôle de données, affichage. C'est la problématique de **présentation**.
- Effectuer sur les données des opérations (calculs) en rapport avec les règles métier ("business logic"). C'est la problématique des **traitements**.
- Accéder et stocker les informations qu'il manipule, notamment entre deux utilisations. C'est la problématique des **données**.

> Dans certains cas de figure, l'une ou l'autre problématique seront très réduites (logiciel sans utilisateur, pas de stockage des données, etc).

La phase d'architecture d'une application consiste aussi à choisir comment sont gérées ces trois problématiques, autrement dit à les répartir dans l'application créée.

L'architecture d'un logiciel

Résultat de l'activité du même nom, l'architecture d'un logiciel décrit sa structure globale, son squelette. Elle décrit les principaux éléments qui composent le logiciel, ainsi que les flux d'échanges entre ces éléments. Elle permet à l'équipe de développement d'avoir une vue d'ensemble de l'organisation du logiciel, et constitue donc en elle-même une forme de documentation.

On peut décrire l'architecture d'un logiciel selon différents points de vue. Entre autres, une vue **logique** mettre l'accent sur le rôle et les responsabilités de chaque partie du logiciel. Une vue **physique** présentera les processus, les machines et les liens réseau nécessaires.

Principes de conception

Ce chapitre présente les grands principes qui doivent guider la création d'un logiciel, et plus généralement le travail du développeur au quotidien.

> Certains étant potentiellement contradictoires entre eux, il faudra nécessairement procéder à des compromis ou des arbitrages en fonction du contexte du projet.

Séparation des responsabilités

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

> C'est sans doute le principe de conception le plus essentiel.

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.

Le **principe de responsabilité unique** (*single responsibility principle*) stipule quant à lui que chaque sous-partie atomique d'uun logiciel (exemple : une classe) doit avoir une unique responsabilité (une raison de changer) ou bien être elle-même décomposée en sous-parties. "A class should have only one reason to change" (Robert C. Martin).

Exemples d'applications de ces deux principes :

- Une sous-partie qui s'occupe des affichages à l'écran participe ne devrait pas comporter de traitements métier, ni de code en rapport avec l'accès aux données.
- Un composant de traitements métier (calcul scientifique ou financier, etc) ne doit pas s'intéresser ni à l'affichage des données qu'il manipule, ni à leur stockage.
- Une classe d'accès à une base de données (connexion, exécution de requêtes) ne devrait faire ni traitements métier, ni affichage des informations.
- Une classe qui aurait deux raisons de changer devrait être scindée en deux classes distinctes.

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 aommes 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 plates-formes comme GitHub ou des outils comme NuGet, Composer ou npm 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.

Voici parmi bien d'autres quelques exemples de problématiques techniques adressables par des composants logiciels :

- Accès à une base de données (connexion, exécution de requêtes).
- Calculs scientifiques.
- Gestion de l'affichage (moteur 3D).
- Journalisation des évènements dans des fichiers.
- ...

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.

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.

Le couplage peut également désigner une dépendance envers une technologie ou un élément extérieur spécifique : un SGBD, un composant logiciel, etc.

Le principe de responsabilité unique permet de limiter le couplage au sein de l'application : chaque sous-partie a un rôle précis et n'a que des interactions limitées avec les autres sous-parties. Pour aller plus loin, il faut limiter le nombre de paramètres des méthodes et utiliser des **classes abstraites** ou des **interfaces** plutôt que des implémentations spécifiques ("Program to interfaces, not to implementations").

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.

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, comme dans l'exemple suivant.

```
function A() {
  // ...

  // Code dupliqué

  // ...
}

function B() {
  // ...

  // Code dupliqué

  // ...
}
```

La solution classique consiste à factoriser les lignes auparavant dupliquées.

```
function C() {
  // Code auparavant dupliqué
}

function A() {
  // ...

  // Appel à C()

  // ...
}

function B() {
  // ...

  // Appel à C()

  // ...
}
```

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.

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.

Il part du constat que la complexité entraîne des surcoûts de développement puis de maintenance, pour des gains parfois discutables. La complexité peut prendre la forme d'une architecture surdimensionnée par rapports aux besoins (*over-engineering*), ou de l'ajout de fonctionnalités secondaires ou non prioritaires.

Une autre manière d'exprimer ce principe consiste à affirmer qu'une application doit être créée selon l'ordre de priorité ci-dessous :

1. *Make it work*.
2. *Make it right*.
3. *Make it fast*.

YAGNI

Ce troisième acronyme signifie **You Ain't Gonna Need It**. Corollaire du précédent, il consiste à ne pas se baser sur d'hypothétiques évolutions futures pour faire les choix du présent, au risque d'une complexification inutile (principe KISS). Il faut réaliser l'application au plus simple et en fonction des besoins actuels.

Le moment venu, il sera toujours temps de procéder à des changements (refactorisation ou *refactoring*) pour que l'application réponde aux nouvelles exigences.

Patrons logiciels

Ce chapitre présente les patrons logiciels (*software patterns*), qui sont des modèles de solutions à des problématiques fréquentes d'architecture ou de conception.

Patrons d'architecture

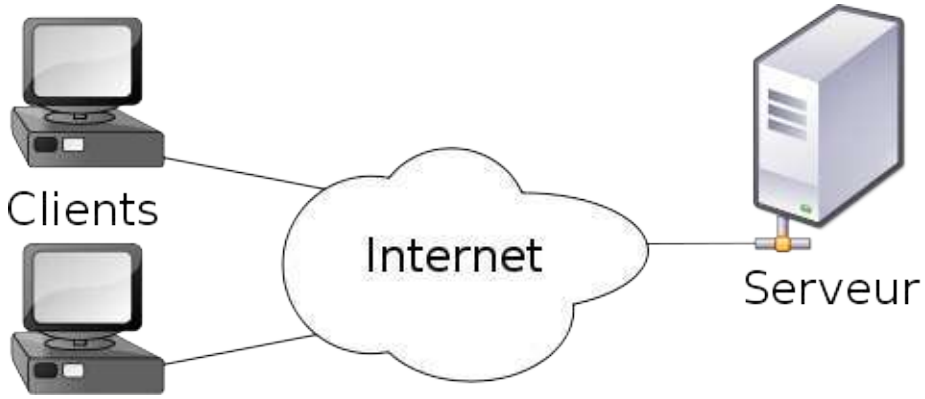
Il n'existe pas une architecture logicielle parfaite qui s'adapterait à toutes les exigences.

Au fil du temps et des projets, plusieurs architectures-types se sont dégagées. Elles constituent des patrons d'architecture (*architecture patterns*) qui ont fait leurs preuves et peuvent servir d'inspiration pour un nouveau projet.

> Vous trouverez une description détaillée des principaux styles architecturaux à cette adresse.

Architecture client/serveur

L'architecture client/serveur caractérise un système basé sur des échanges réeeau 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 noeud central du système et représente son maillon faible. En cas de défaillance (surcharge, indisponibilité, problème réseau), les clients ne peuvent plus fonctionner.

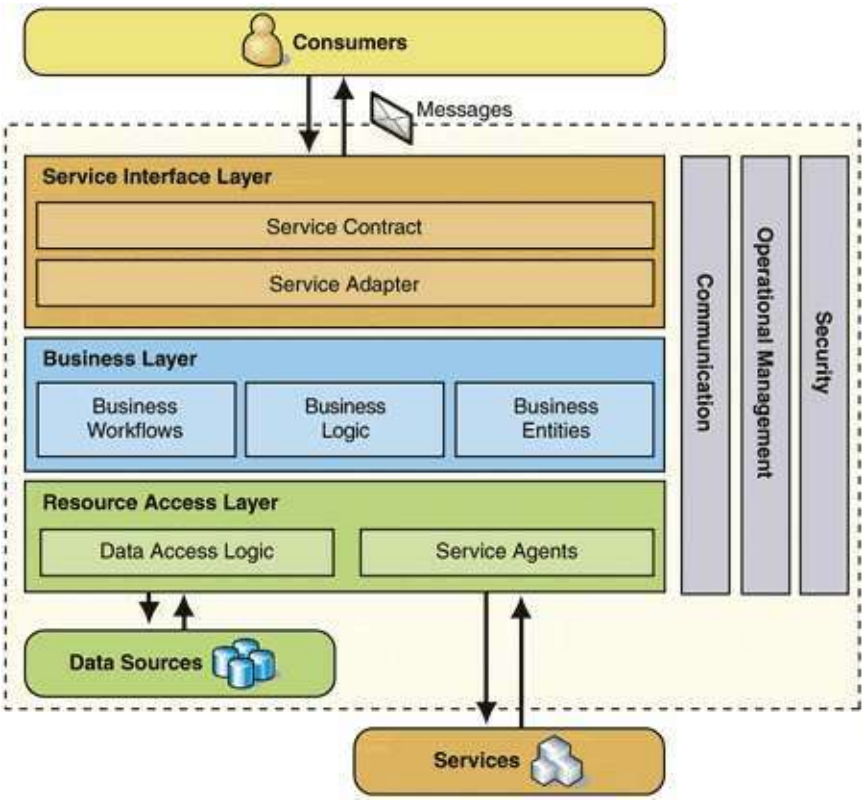
On peut classer les clients d'une architecture client/serveur en plusieurs types :

- **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

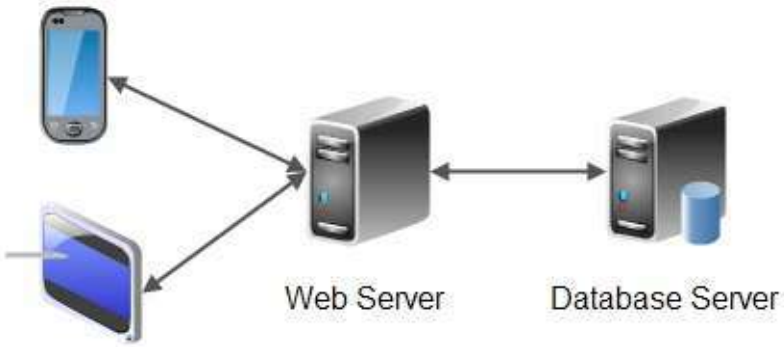
Une architecture en couches organise un logiciel sous forme de couches (*layers*). Chaque couche ne peut communiquer qu'avec les couches adjacentes.



Cette architecture respecte le principe de séparation des responsabiltés 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.

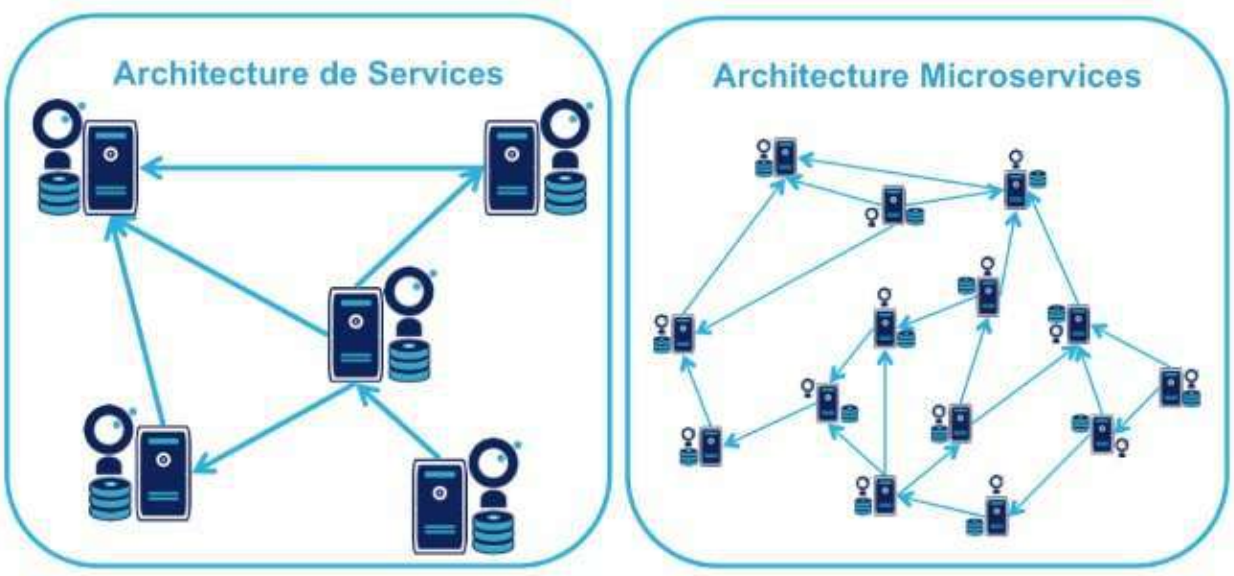
Un navigateur web accédant à des pages dynamiques intégrant des informations stockées dans une base de données constitue un exemple classique d'architecture 3-tiers.



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 (source) illustre la défférence entre ces deux approches.



Architecture Modèle-Vue-Contrôleur

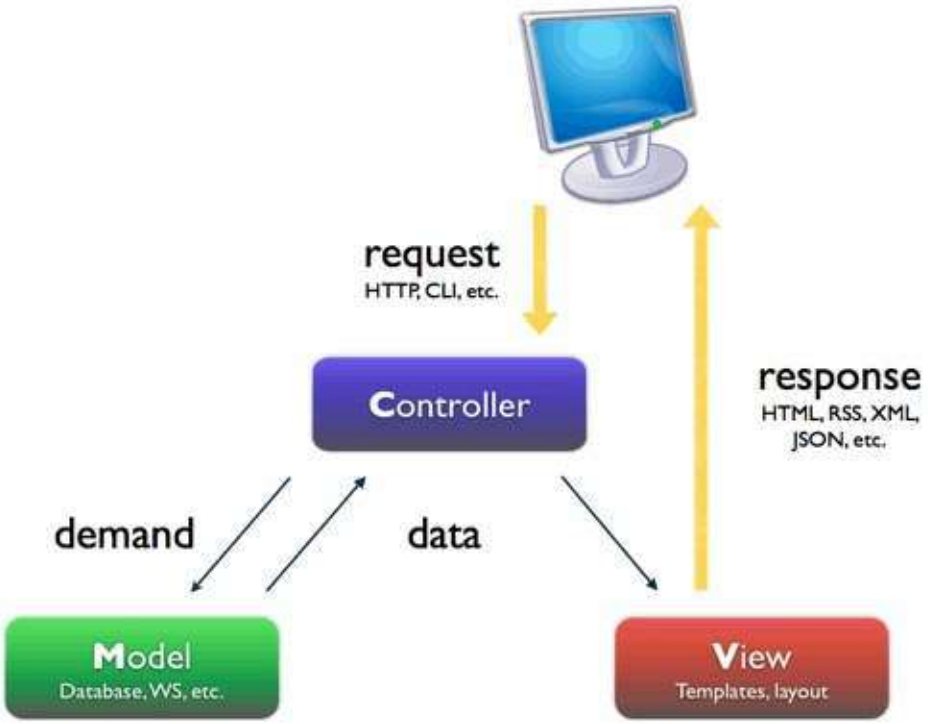
La patron Modèle-Vue-Contrôleur, ou **MVC**, décompose une application en trois sous-parties :

- 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 (extrait de la documentation du framework PHP Symfony) résume les relations entre les composants d'une architecture web MVC.

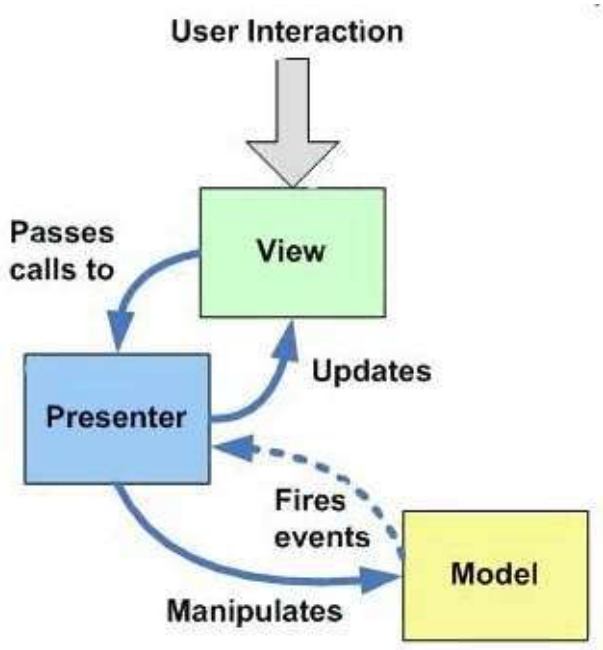


> Attention à ne pas employer le terme de "couche" à propos d'une architecture MVC. Dans une architecture en couches, chaque couche ne peut communiquer qu'avec les couches adjacentes. Les parties Modèle, Vue et Contrôleur ne sont donc pas des couches au vrai sens du mot.

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*, Vuet et Modèle sont couplées et les modifications du Modèle déclenchent la mise à jour de la Vue.

Patrons de conception

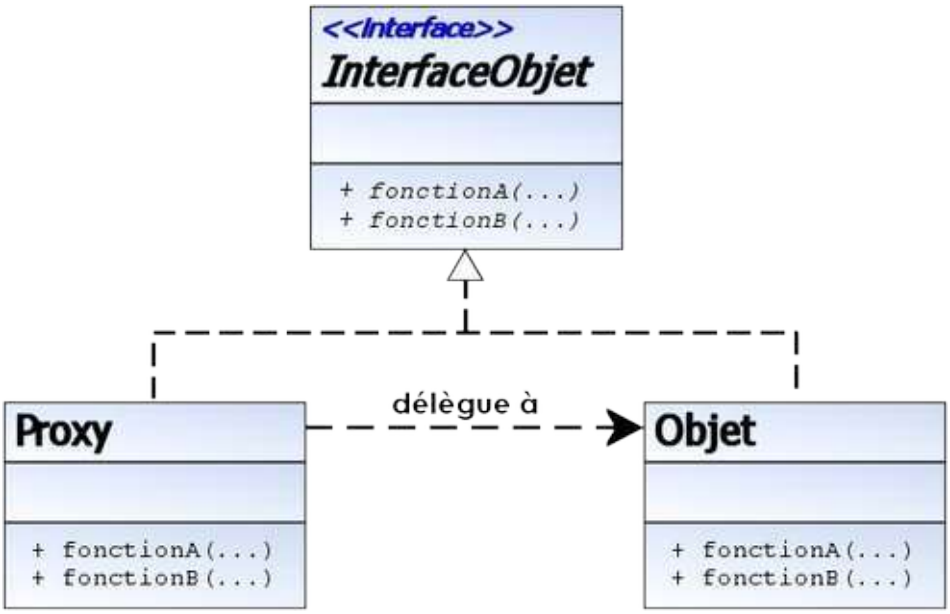
Un **patron de conception** (*design pattern*) est une solution standard à un problème de conception. L'ensemble des patrons de conception constitue un catalogue de bonnes pratiques issu de l'expérience de la communauté. Leurs noms forment un vocabulaire commun qui permet d'identifier immédiatement la solution associée.

> On rencontre également le terme de "motif de conception".

Les patrons de conception ont été popularisés par le livre *Design Patterns – Elements of Reusable Object-Oriented Software* sorti en 1995 et co-écrit par quatre auteurs (le *Gang Of Four*, ou GoF). Ce livre décrit 23 patrons, auxquels d'autres se sont rajoutés depuis.

Chaque patron décrit un problème à résoudre puis les éléments de sa solution, ainsi que leurs relations. La formalisme graphique utilisé est souvent un diagramme de classes UML.

L'exemple ci-dessous décrit le patron de conception **Proxy**, dont l'objectif est de substituer un objet à un autre afin de contrôler l'utilisation de ce dernier.



Il existe également un catalogue d'**anti-patterns**. Comme son nom l'indique, un anti-pattern est un exemple de mauvaise pratique à ne surtout pas suivre.

> Certains patterns originaux du GoF sont maintenant plutôt considérés comme des anti-patterns. C'est par exemple le cas du pattern **Singleton** (explications).

Production du code source

L'objectif de ce chapitre est de présenter les enjeux et les solutions liés à la production du code source d'un logiciel.

Introduction

Le code source est le coeur d'un projet logiciel. Il est essentiel que tous les membres de l'équipe de développement se coordonnent pour adopter des règles communes dans la production de ce code.

L'objectif de ces règles est l'uniformisation de la base de code source du projet. Les avantages liés sont les suivants :

- La consultation du code est facilitée.
- Les risques de duplication ou d'erreurs liées à des pratiques disparates sont éliminés.
- Chaque membre de l'équipe peut comprendre et intervenir sur d'autres parties que celles qu'il a lui-même réalisées.
- Les nouveaux venus sur le projet mettront moins longtemps à être opérationnels.

> Il est important de garder à l'esprit qu'un développeur passe en moyenne beaucoup plus de temps à lire qu'à écrire du code.

Convention de nommage

Une première série de règle concerne le nommage des différents éléments qui composent le code. Il n'existe pas de standard universel à ce sujet.

La convention la plus fréquemment adoptée se nomme **camelCase** (ou parfois *lowerCamelCase*). Elle repose sur deux grands principes :

- Les noms des classes (et des méthodes en C#, pour être en harmonie avec le framework .NET) commencent par une lettre majuscule.
- Les noms de tous les autres éléments (variables, attributs, paramètres, etc) commencent par une lettre minuscule.
- Si le nom d'un élément se compose de plusieurs mots, la première lettre de chaque mot suivant le premier s'écrit en majuscule.

Voici un exemple de classe conforme à cette convention.

```
class UneNouvelleClasse {
  private int unAttribut;
  private float unAutreAttribut;

  public void UneMethode(int monParam1, int monParam2) { ... }
  public void UneAutreMethode(string encoreUnParametre) { ... }
}
```

On peut ajouter à cette convention une règle qui impose d'utiliser le pluriel pour nommer les éléments contenant plusieurs valeurs, comme les tableaux et les listes. Cela rend le parcours de ces éléments plus lisible.

```
List<string> clients = new List<string>();
// ...
foreach(string client in clients) {
  // 'clients' désigne la liste, 'client' le client courant
  // ...
}
```

> On peut aussi utiliser des noms de la forme `listeClients`.

Langue utilisée

La langue utilisée dans la production du code doit bien entendu être unique sur tout le projet.

Le français (`idClientSuivant`) et l'anglais (`nextClientId`) ont chacun leurs avantages et leurs inconvénients. On choisira de préférence l'anglais pour les projets de taille importante ou destinés à être publiés en ligne.

Formatage du code

La grande majorité des IDE et des éditeurs de code offrent des fonctionnalités de formatage automatique du code. A condition d'utiliser un paramétrage commun, cela permet à chaque membre de l'équipe de formater rapidement et uniformément le code sur lequel il travaille.

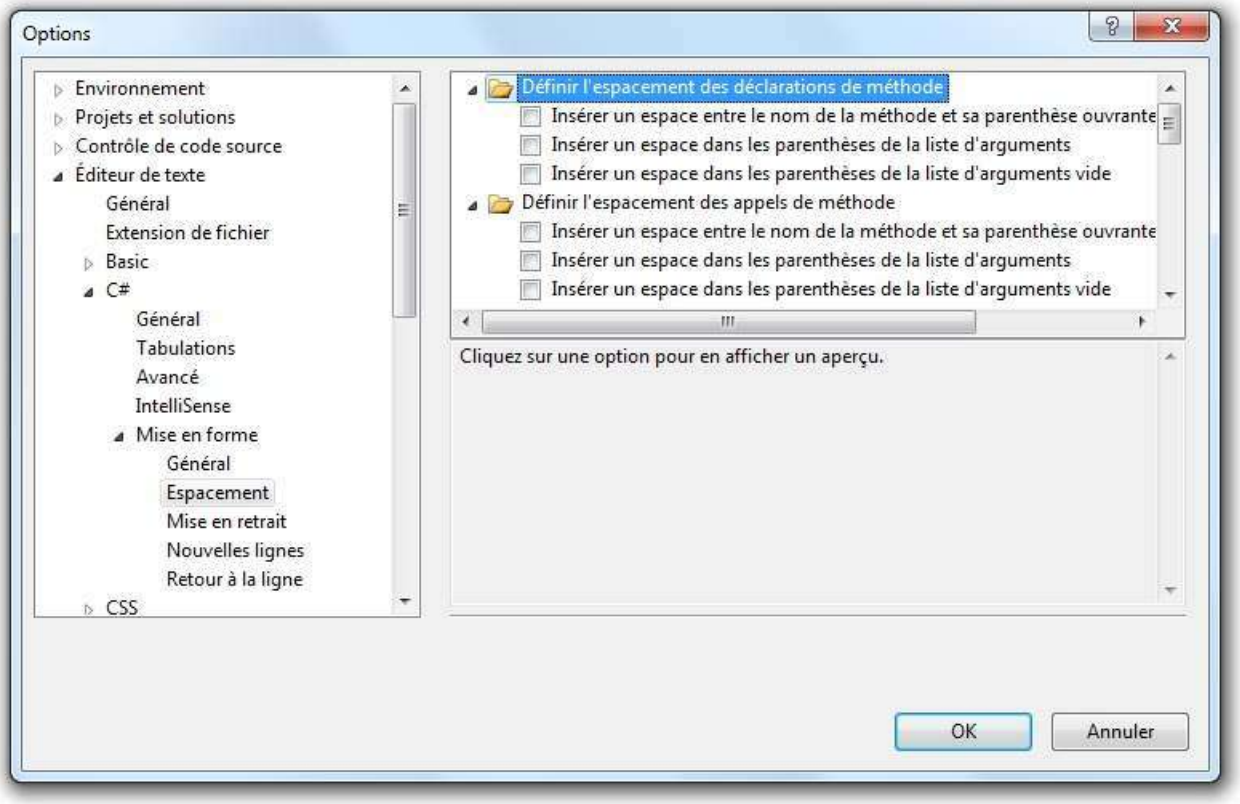
Les paramètres de formatage les plus courants sont :

- Taille des tabulations (2 ou 4 espaces).
- Remplacement automatique des tabulations par des espaces.
- Passage ou non à la ligne après chaque accolade ouvrante ou fermante.
- Ajout ou non d'un espace avant une liste de paramètres.

- ...

Sous Visual Studio, la commande de formatage automatique du code est **Edition->Avancé->Mettre le document en forme**.

Voici quelques exemples de paramétrages possibles du formatage (menu **Outils->Options**).



Commentaires

L'ajout de commentaires permetde faciliter la lecture et la compréhension d'une portion de code source. L'ensemble des commentaires constitue une forme efficace de documentation d'un projet logiciel.

Il n'y a pas de règle absolue, ni de consensus, en matière de taux de commentaires dans le code source. Certaines méthodologies de développement agile (*eXtreme Programming*) vont jusqu'à affirmer qu'un code bien écrit se suffit à lui-même et ne nécessite aucun ajout de commentaires.

Dans un premier temps, il vaut mieux se montrer raisonnable et commenter les portions de code complexes ou essentielles : en-têtes de classes, algorithmes importants, portions atypiques, etc. Il faut éviter de paraphraser le code source en le commentant, ce qui alourdit sa lecture et n'est d'aucun intérêt.

Voici quelques exemples de commentaires inutiles : autant lire directement les instructions décrites.

```
// Initialisation de i à 0
int i = 0;

// Instanciation d'un objet de la classe Random
Random rng = new Random();

// Appel de la méthode Next sur l'objet rng
int nombreAlea = rng.Next(1, 4);
```

Voici comment le code précédent pourrait être mieux commenté.

```
int i = 0;
Random rng = new Random();

// Génération aléatoire d'un entier entre 1 et 3
int nombreAlea = rng.Next(1, 4);
```

Gestion des versions

L'objectif de ce chapitre est de découvrir ce qu'est la gestion des versions d'un logiciel, en utilisant comme exemple l'outil Git.

Introduction

Nous avons déjà mentionné qu'un projet logiciel d'entreprise a une durée de vie de plusieurs années et subit de nombreuses évolutions au cours de cette période. On rencontre souvent le besoin de livrer de nouvelles versions qui corrigent des bogues ou apportent de nouvelles fonctionnalités. Le code source du logiciel "vit" donc plusieurs années. Afin de pouvoir corriger des problèmes signalés par un utilisateur du logiciel, on doit savoir précisément quels fichiers source font partie de quelle(s) version(s).

En entreprise, seuls une petite minorité de logiciels sont conçus par un seul développeur. La grande majorité des projets sont réalisés et/ou maintenus par une équipe de plusieurs personnes travaillant sur la même base de code source. Ce travail en parallèle est source de complexité :

- Comment récupérer le travail d'un autre membre de l'équipe ?
- Comment publier ses propres modifications ?
- Comment faire en cas de modifications conflictuelles (travail sur le même fichier source qu'un ou plusieurs collègues) ?
- Comment accéder à une version précédente d'un fichier ou du logiciel entier ?

Pour les raisons précédentes, tout projet logiciel d'entreprise (même mono-développeur) doit faire l'objet d'une **gestion des versions** (*Revision Control System* ou *versioning*). La gestion des versions vise les objectifs suivants :

- Assurer la pérennité du code source d'un logiciel.
- Permettre le travail collaboratif.
- Fournir une gestion de l'historique du logiciel.

> La gestion des versions est parfois appelée gestion du code source (**SCM**, *Source Code Management*).

La gestion des versions la plus basique consiste à déposer le code source sur un répertoire partagé par l'équipe de développement. Si elle permet à tous de récupérer le code, elle n'offre aucune solution aux autres complexités du développement en équipe et n'autorise pas la gestion des versions.

Afin de libérer l'équipe de développement des complexités du travail collaboratif, il existe une catégories de logiciels spécialisés dans la gestion des versions.

Les logiciels de gestion des versions

Principales fonctionnalités

Un logiciel de gestion des versions est avant tout un **dépôt de code** qui héberge le code source du projet. Chaque développeur peut accéder au dépôt afin de récupérer le code source, puis de publier ses modifications. Les autres développeurs peuvent alors récupérer le travail publié.

Le logiciel garde la trace des modifications successives d'un fichier. Il permet d'en visualiser l'**historique** et de revenir à une version antérieure.

Un logiciel de gestion des versions permet de travailler en parallèle sur plusieurs problématiques (par exemple, la correction des bogues de la version publiée et l'avancement sur la future version) en créant des **branches**. Les modifications réalisées sur une branche peuvent ensuite être intégrées (*merging*) à une autre.

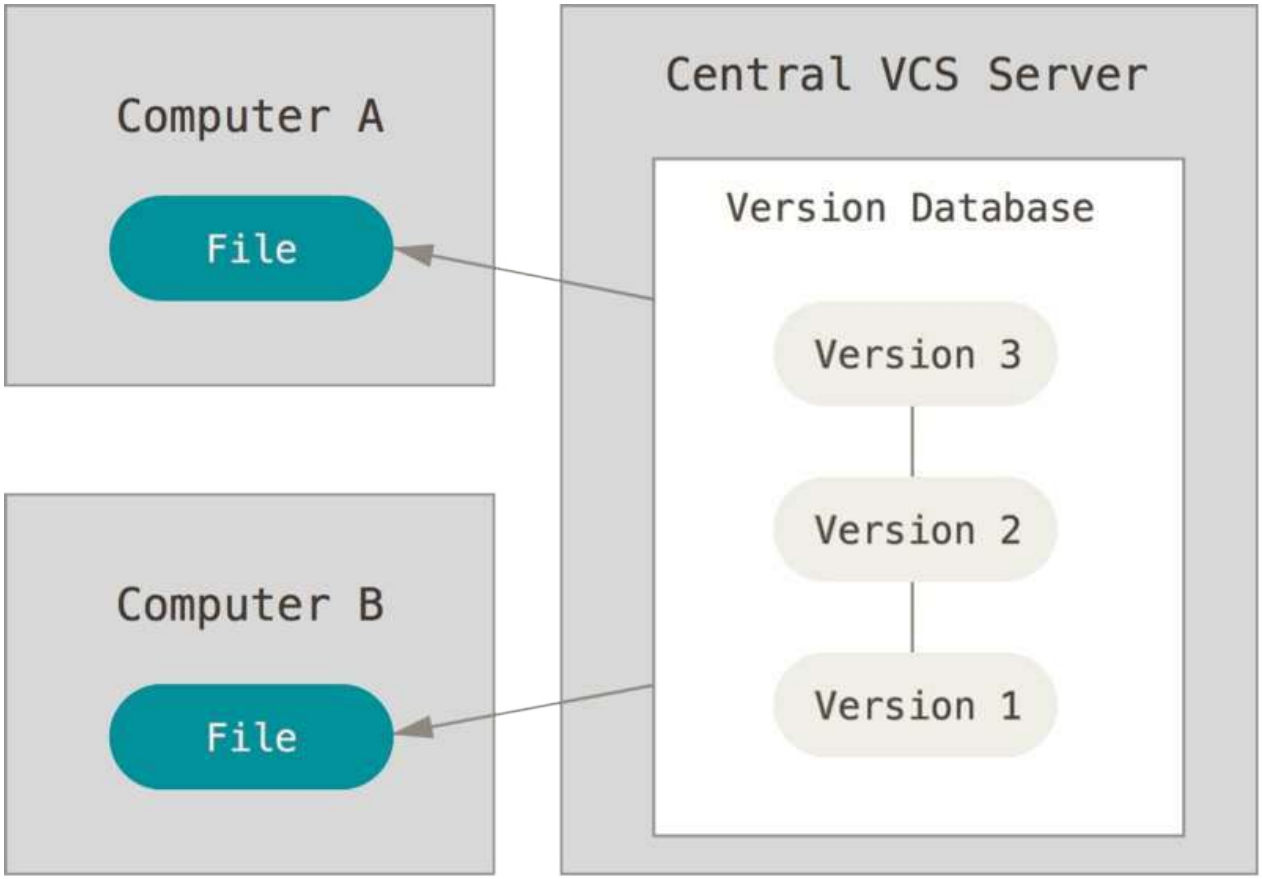
En cas d'apparition d'un **conflit** (modifications simultanées du même fichier par plusieurs développeurs), le logiciel de gestion des versions permet de comparer les versions du fichier et de choisir les modifications à conserver ou à rejeter pour créer le fichier fusionné final.

Le logiciel de gestion des versions permet de regrouper logiquement des fichiers par le biais du *tagging* : il ajoute aux fichiers source des tags correspondant aux différentes versions du logiciel.

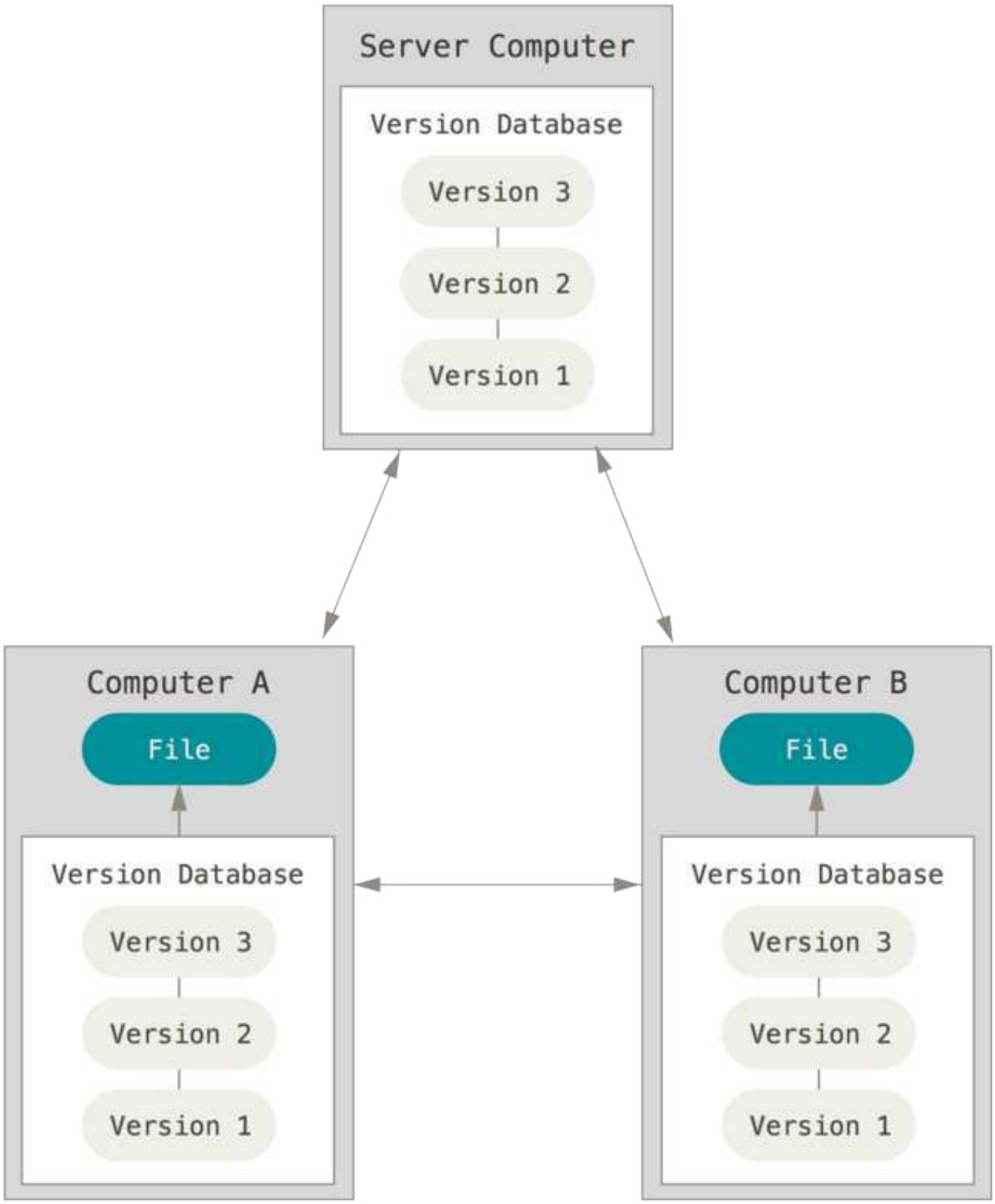
Gestion centralisée Vs gestion décentralisée

On peut classer les logiciels de gestion des versions en deux catégories.

La première catégorie offre une gestion centralisée du code source. Dans ce cas de figure, il n'existe qu'un seul dépôt qui fait référence. Les développeurs se connectent au logiciel de gestion des versions suivant le principe du **client/serveur**. Cette solution offre les avantages de la centralisation (administration facilitée) mais handicape le travail en mode déconnecté : une connexion au logiciel de SCM est indispensable.



Une seconde catégorie est apparu il y a peu. Elle consiste à voir le logiciel de gestion des versions comme un outil individuel permettant de travailler de manière décentralisé (hors ligne). Dans ce cas de figure, il existe autant de dépôts de code que de développeurs sur le projet. Le logiciel de gestion des versions fournit heureusement un service de synchronisation entre toutes ces bases de code. Cette solution fonctionne suivant le principe du **pair-à-pair**. Cependant, il peut exister un dépôt de référence contenant les versions livrées.



Principaux logiciels de gestion des versions

Il existe de très nombreux logiciels de gestion des versions (Wikipedia). Nous n'allons citer que les principaux.

Assez ancien mais toujours utilisé, **CVS** (*Concurrent Versioning System*) fonctionne sur un principe centralisé, de même que son successeur **SVN** (*Subversion*). Tous deux sont souvent employés dans le monde du logiciel libre (ce sont eux-mêmes des logiciels libres).

Les logiciels de SCM décentralisés sont apparus plus récemment. On peut citer **Mercurial** et surtout **Git**, que nous utiliserons dans la suite de ce chapitre. Ce sont également des logiciels libres.

Microsoft fournit un logiciel de SCM développé sur mesure pour son environnement. Il se nomme **TFS** (*Team Foundation Server*) et fonctionne de manière centralisée. TFS esr une solution payante.

> Il est tout à fait possible de gérer le code source d'un projet .NET avec un autre outil de gestion des verions que TFS.

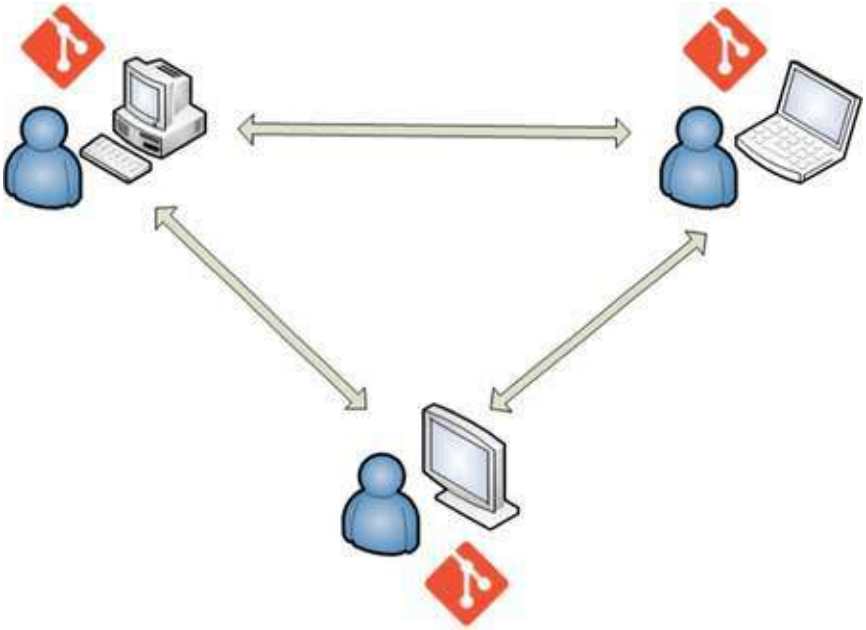
Présentation de Git

Git est un logiciel libre de gestion des versions. C'est un outil qui permet d'archiver et de maintenir les différentes versions d'un ensemble de fichiers textuels constituant souvent le code source d'un projet logiciel. Créé à l'origine pour gérer le code du noyau Linux, il est multi-langages et multi-plateformes. Git est devenu à l'heure actuelle un quasi-standard.

> Le nom "Git" se prononce comme dans "guitare" et non pas comme dans "jitsu".

Fonctionnement

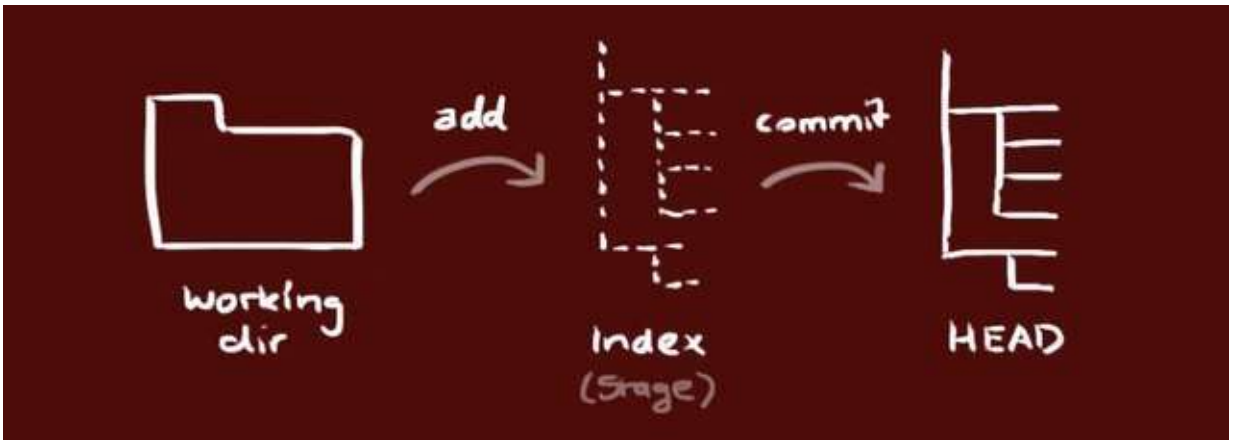
Git rassemble dans un **dépôt** (*repository* ou *repo*) l'ensemble des données associées au projet. Il fonctionne de manière décentralisée : tout dépôt Git contient l'intégralité des données (code source, historique, versions, etc). Chaque participant au projet travaille à son rythme sur son dépôt local. Il existe donc autant de dépôts que de participants. Git offre des mécanismes permettant de synchroniser les modifications entre tous les dépôts.



Un dépôt Git correspond physiquement à un ensemble de fichiers rassemblés dans un répertoire `.git`. Sauf cas particulier, il n'est pas nécessaire d'intervenir manuellement dans ce répertoire.

Lorsqu'on travaille avec Git, il est essentiel de faire la distinction entre trois zones :

- Le **répertoire de travail** (*working directory*) correspond aux fichiers actuellement sauvegardés localement.
- L'**index** ou *staging area* est un espace de transit.
- **HEAD** correspond aux derniers fichiers ajoutés au dépôt.



Principales opérations

Bien qu'il existe de nombreux plugins et autres interfaces graphiques, Git est à la base prévu pour être utilisé sous forme de commandes textuelles. Les principales commandes sont :

- `git init` : crée un nouveau dépôt vide à l'emplacement courant.
- `git status` : affiche les différences entre le répertoire de travail, l'index et HEAD.

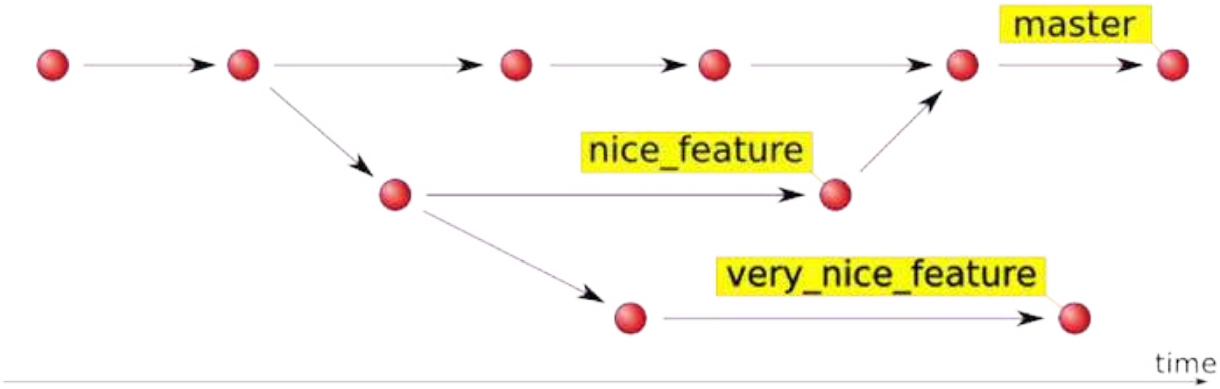
- `git add` : ajoute des fichiers depuis le répertoire de travail vers l'index.
- `git commit` : ajoute des fichiers depuis l'index vers HEAD.
- `git clone` : clone un dépôt existant local ou distant.
- `git pull` : récupère des modifications depuis un dépôt distant vers HEAD.
- `git push` : publie des modifications depuis HEAD vers un dépôt distant.

Pour plus de détails sur les autres opérations possibles, consultez ce mémento.

Notion de branche

Une branche permet de travailler de manière isolée sur une problèmatique particulière. Leur gestion par Git est particulièrement simple et efficace.

Tout dépôt Git possède une branche par défaut nommée `master`. On peut ensuite créer de nouvelles branches (`git branch`), y effectuer des modifications, puis fusionner cette branche avec la branche par défaut (`git merge`).



Le fichier `.gitignore`

Lorsqu'il est présent à la racine d'un répertoire géré par Git, le fichier `.gitignore` permet d'ignorer certains fichiers qui n'ont pas à être versionnés. C'est typiquement le cas des fichiers créés lors de la génération du logiciel ou des composants externes téléchargés par un gestionnaire de dépendances.

La plate-forme GitHub maintient en ligne une liste de fichiers .gitignore pour de nombreux types de projets. Voici un extrait du fichier `.gitignore` pour l'environnement Visual Studio.

```
# User-specific files
*.suo
*.user
*.userosscache
*.sln.docstates

# User-specific files (MonoDevelop/Xamarin Studio)
*.userprefs

# Build results
[Dd]ebug/
[Dd]ebugPublic/
[Rr]elease/
[Rr]eleases/
x64/
x86/
bld/
[Bb]in/
[Oo]bj/
[Ll]og/

# ...
```

Travail collaboratif

L'objectif de ce chapitre est de présenter les enjeux du travail collaboratif dans le cadre de la réalisation d'un logiciel.

Les enjeux du travail collaboratif

La très grande majorité des projets logiciels sont menés par des équipes de plusieurs développeurs. Il est de plus en plus fréquent que ces développeurs travaillent à distance ou en mobilité.

Le travail en équipe sur un projet logiciel nécessite de pouvoir :

- Partager le code source entre membres de l'équipe.
- Gérer les droits d'accès au code.
- Intégrer les modifications réalisées par chaque développeur.
- Signaler des problèmes ou proposer des améliorations qui peuvent ensuite être discutés collectivement.

Pour répondre à ces besoins, des plates-formes de publication et de partage de code en ligne sont apparues. On les appelle parfois des **forges logicielles**. La plus importante à l'heure actuelle est la plate-forme GitHub.

Présentation de GitHub

GitHub est une plate-forme web d'hébergement et de partage de code. Comme son nom l'indique, elle se base sur le logiciel Git.

Le principal service proposé par GitHub est la fourniture de dépôts Git accessibles en ligne. Elle offre aussi une gestion des équipes de travail (*organizations* et *teams*), des espaces d'échange autour du code (*issues* et *pull requests*), des statistiques, etc.

GitHub est utilisée par pratiquement toutes les grandes sociétés du monde du logiciel, y compris Microsoft, Facebook et Apple. Pour un développeur, GitHub peut constituer une vitrine en ligne de son travail et un atout efficace pour son employabilité.

Modèle économique

Le *business model* de GitHub est le suivant :

- La création d'un compte et de dépôts publics sont gratuites.
- La création de dépôts privés est payante.

Ce modèle économique est à l'origine de l'énorme popularité de GitHub pour la publication de projets *open source*.

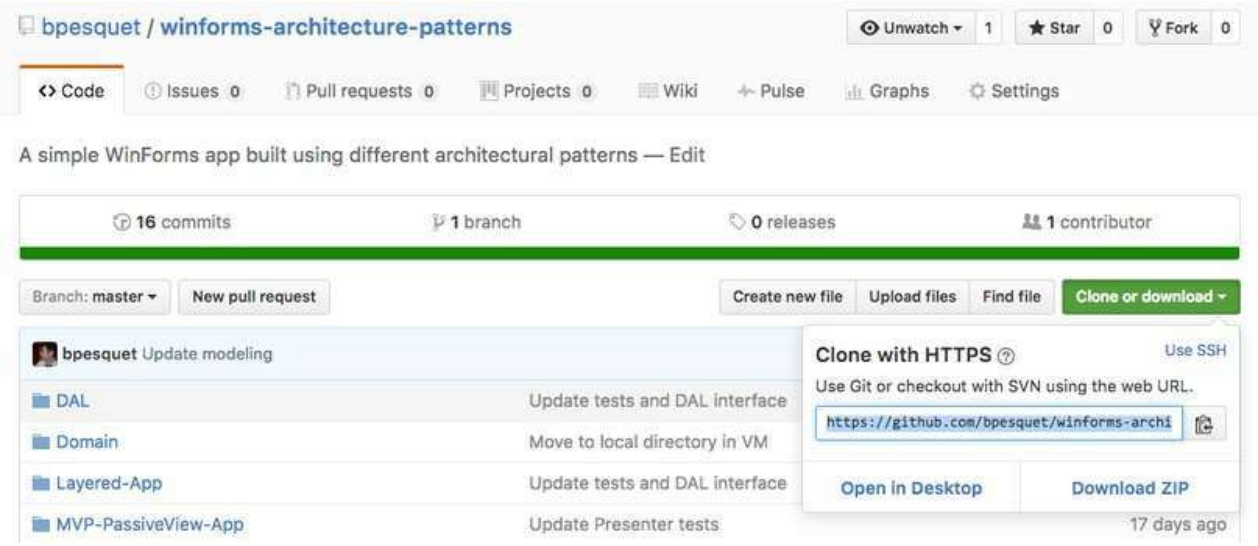
> GitHub peut offrir des dépôts privés gratuits aux étudiants et aux établissements scolaires.

D'autres plates-formes alternatives se basent sur un modèle différent :

- BitBucket autorise les dépôts privés gratuits mais limite la taille de l'équipe à 5 personnes.
- GitLab est une solution *open core* pour installer sur ses propres serveurs une plate-forme similaire à GitHub.

Fonctionnement

Sur GitHub, un dépôt est associé à un utilisateur individuel ou à une organisation. Il est ensuite accessible via une URL de la forme `https://github.com/.../nomDuDepot.git` .

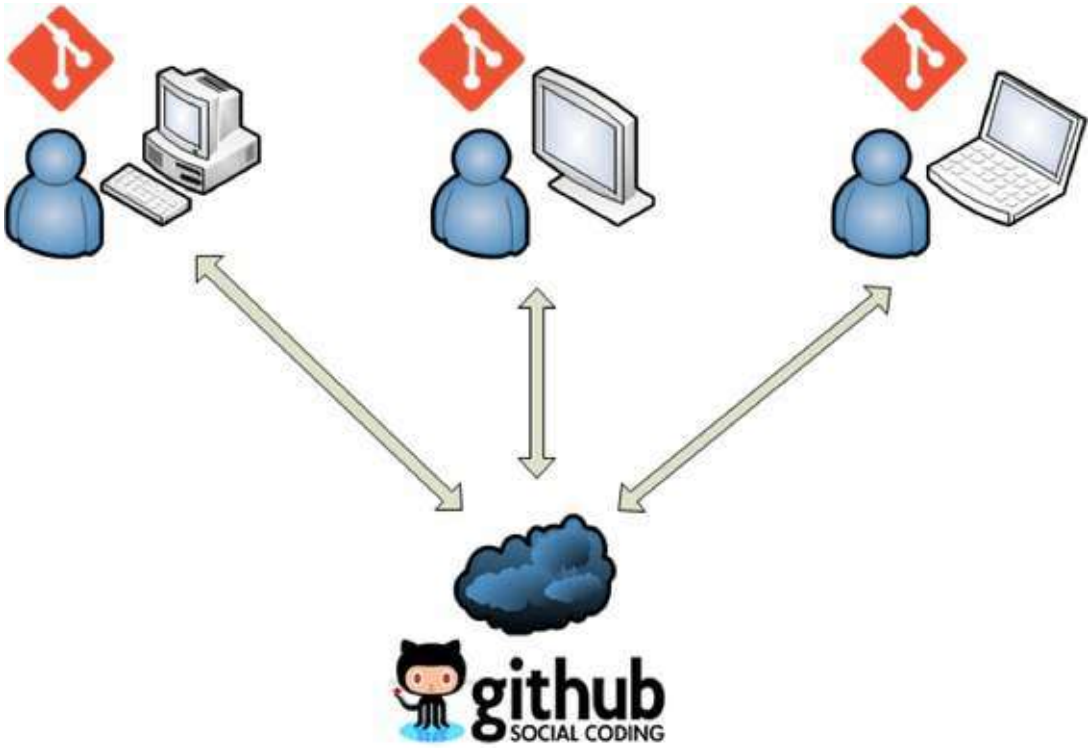


L'organisation du travail en équipe autour de GitHub peut se faire suivant deux modèles distincts.

Modèle "dépôt partagé"

Ce modèle de travail est bien adapté aux petites équipes et aux projets peu complexes.

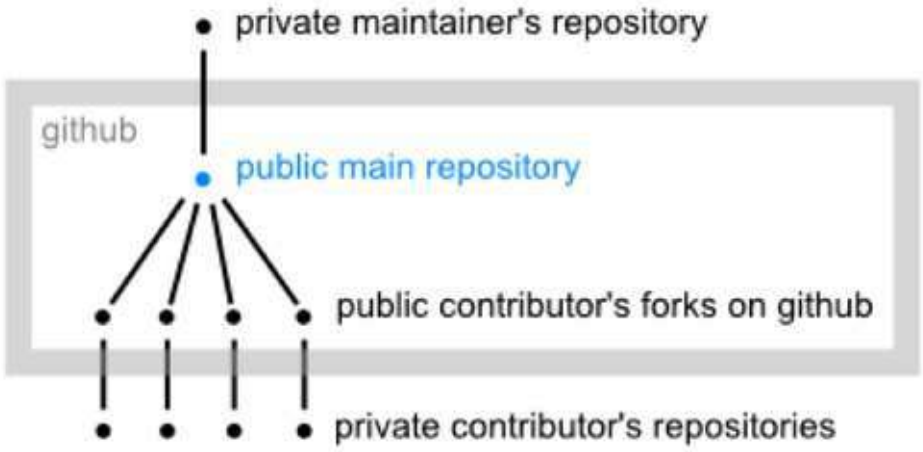
Un dépôt GitHub fait office de dépôt central pour l'équipe. Après avoir obtenu les droits nécessaires, chaque développeur clone ce dépôt (`git clone`) sur sa machine de travail. Ensuite, chacun peut envoyer ses modifications locales vers le dépôt commun (`git push`) et récupérer celles des autres membres de l'équipe (`git pull`).



Modèle "fork and pull"

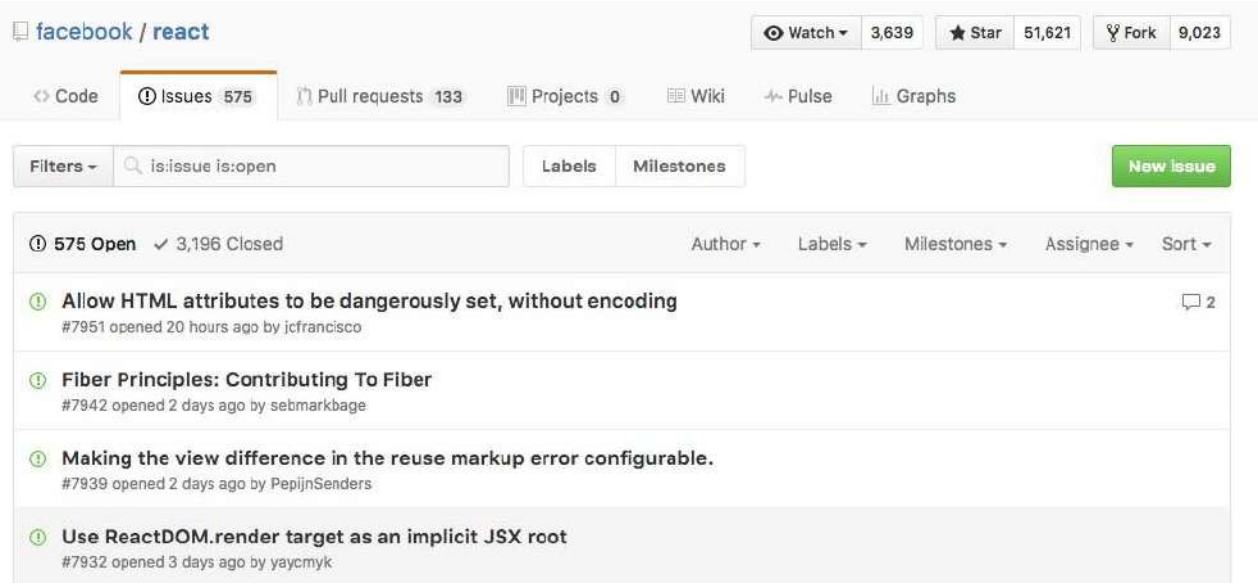
Ce modèle plus complexe est utilisé sur des projets de taille importante.

Un dépôt GitHub principal est *forké* par chaque développeur pour en obtenir une copie exacte sur son compte GitHub, puis cloné sur sa machine locale. Les modifications sont *pushées* sur GitHub, puis le développeur émet une demande d'intégration (*pull request*) pour signaler au responsable du dépôt commun qu'il a effectué des améliorations de son côté. Ce responsable étudie le code ajouté et décide de l'intégrer ou non dans le dépôt principal.



Issues

Parmi les autres services proposés par GitHub, les *issues* ("sujets de discussion") permettent de communiquer autour du projet. Elles sont souvent utilisées pour signaler des problèmes ou proposer des idées.



Tests

L'objectif de ce chapitre est de présenter le rôle des tests dans le processus de création d'un logiciel.

Pourquoi tester ?

La problématique des tests est souvent considérée comme secondaire et négligée par les développeurs. C'est une erreur : lorsqu'on livre une application et qu'elle est placée en production (offerte à ses utilisateurs), il est essentiel d'avoir un maximum de garanties sur son bon fonctionnement afin d'éviter au maximum de coûteuses mauvaises surprises.

Le test d'une application peut être manuel. Dans ce cas, une personne effectue sur l'application une suite d'opérations prévue à l'avance (navigation, connexion, envoi d'informations...) pour vérifier qu'elle possède bien le comportement attendu. C'est un processus coûteux en temps et sujets aux erreurs (oublis, négligences, etc).

En complément de ces tests manuels, on a tout intérêt à intégrer à un projet logiciel des tests automatisés qui pourront être lancés aussi souvent que nécessaire. Ceci est d'autant plus vrai pour les méthodologies agiles basées sur un développement itératif et des livraisons fréquentes, ou bien lorsque l'on met en place une intégration continue.

Comment tester ?

On peut classer les tests logiciels en différentes catégories.

Tests de validation

Ces tests sont réalisés lors de la recette (validation) par un client d'un projet livré par l'un de ses fournisseurs. Souvent écrits par le client lui-même, ils portent sur l'ensemble du logiciel et permet de vérifier son comportement global en situation.

De part leur spectre large et leur complexité, les tests de validation sont le plus souvent manuels. Les procédures à suivre sont regroupées dans un document associé au projet, fréquemment nommé plan de validation.

Tests d'intégration

Dans un projet informatique, l'intégration est de fait d'assembler plusieurs composants (ou modules) élémentaires en un composants de plus haut niveau. Un test d'intégration valide les résultats des interactions entre plusieurs composants permet de vérifier que leur assemblage s'est produit sans défaut. Il peut être manuel ou automatisé.

Un nombre croissant de projets logiciels mettent en place un processus d'**intégration continue**. Cela consiste à vérifier que chaque modification ne produit pas de régression dans l'application développée. L'intégration continue est nécessairement liée à une batterie de tests qui se déclenchent automatiquement lorsque des modifications sont intégrées au code du projet.

Tests unitaires

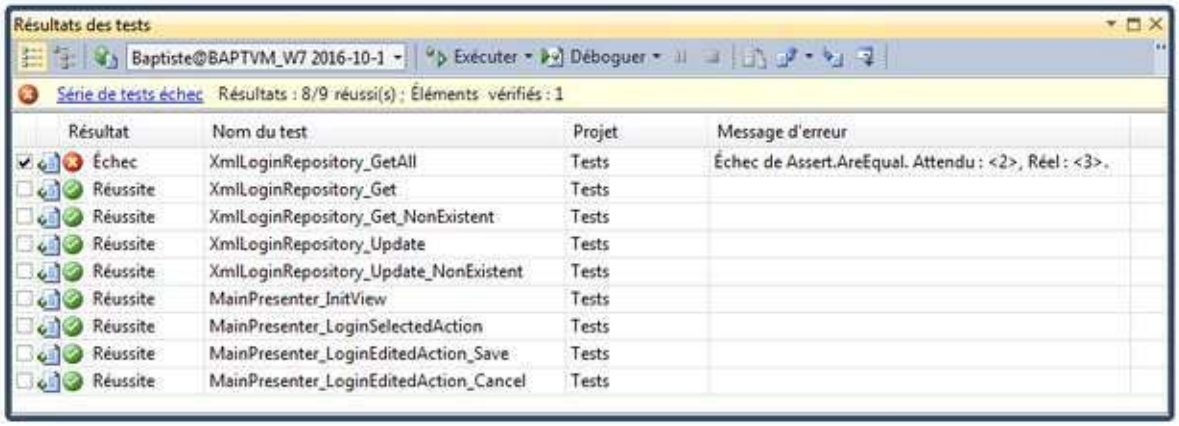
Contrairement aux tests de validation et d'intégration qui testent des pans entiers d'un logiciel, un test unitaire na valide qu'une portion atomique du code source (exemple : une seule classe) et est systématiquement automatisé.

Le test unitaire offre les avantages suivants :

- Il est facile à écrire. Dédié à une partie très réduite du code, le test unitaire ne nécessite le plus souvent qu'un contexte minimal, voire pas de contexte du tout.
- Il offre une granularité de test très fine et permet de valider exhaustivement le comportement de la partie du code testée (cas dégradés, saisie d'informations erronées...).
- Son exécution est rapide, ce qui permet de le lancer très fréquemment (idéalement à chaque modification du code testé).
- Il rassemble les cas d'utilisation possibles d'une portion d'un projet et représente donc une véritable documentation sur la manière de manipuler le code testé.

L'ensemble des tests unitaires d'un projet permet de valider unitairement une grande partie de son code source et de détecter le plus tôt possibles d'éventuelles erreurs.

L'image ci-dessous illustre le résultat du lancement de tests unitaires sous Visual Studio.



> Certaines méthodologies de développement logiciel préconisent l'écriture des tests unitaires avant celle du code testé (TDD, *Test Driven Developement*).

Création de test doubles

En pratique, très peux de parties d'un projet fonctionnent de manière autonome, ce qui complique l'écriture des tests unitaires. Par exemple, comment tester unitairement une classe qui collabore avec plusieurs autres pour réaliser ses fonctionnalités ?

La solution consiste à créer des éléments qui simulent le comportement des collaborateurs d'une classe donnée, afin de pouvoir tester le comportement de cette classe dans un environnement isolé et maîtrisé. Ces éléments sont appelés des **test doubles**.

> Un couplage faible au sein du projet (voir ce chapitre) facilite l'écriture de *test doubles*.

Selon la complexité du test à écrire, un *test double* peut être :

- Un **dummy**, élément basique sans aucun comportement, juste là pour faire compiler le code lors du test.
- Un **stub**, qui renvoie des données permettant de prévoir les résultats attendus lors du test.
- Un **mock**, qui permet de vérifier finement le comportement de l'élément testé (ordre d'appel des méthodes, paramètres passés, etc).

Documentation

L'objectif de ce chapitre est de découvrir les différents aspects associés à la documentation d'un logiciel.

Introduction

Nous avons déjà mentionné à plusieurs reprises qu'un logiciel a une durée de vie de plusieurs années et subit de nombreuses évolutions au cours de cette période. En entreprise, seuls une petite minorité de logiciels sont conçus par un seul développeur. La grande majorité des projets sont réalisés et/ou maintenus par une équipe de plusieurs personnes travaillant sur la même base de code source. Il est fréquent que les effectifs changent et que des développeurs soient amenés à travailler sur un logiciel sans avoir participé à sa création. L'intégration de ces nouveaux développeurs doit être aussi rapide et efficace que possible.

Cependant, il est malaisé, voire parfois très difficile, de se familiariser avec un logiciel par la seule lecture de son code source. En complément, un ou plusieurs documents doivent accompagner le logiciel. On peut classer cette documentation en deux catégories :

- La documentation technique
- La documentation utilisateur.

La documentation technique

Rôle

Le mot-clé de la documentation technique est "comment". Il ne s'agit pas ici de dire pourquoi le logiciel existe, ni de décrire ses fonctionnalités attendues. Ces informations figurent dans d'autres documents comme le cahier des charges. Il ne s'agit pas non plus d'expliquer à un utilisateur du logiciel ce qu'il doit faire pour effectuer telle ou telle tâche : c'est le rôle de la documentation utilisateur.

La documentation technique doit expliquer **comment** fonctionne le logiciel.

Public visé

La documentation technique est écrite par des informaticiens, pour des informaticiens. Elle nécessite des compétences techniques pour être comprise. Le public visé est celui des personnes qui interviennent sur le logiciel du point de vue technique : développeurs, intégrateurs, responsables techniques, éventuellements chefs de projet.

Dans le cadre d'une relation maîtrise d'ouvrage / maîtrise d'oeuvre pour réaliser un logiciel, la responsabilité de la documentation technique est à la charge de la maîtrise d'oeuvre. Bien sûr, le ou les document(s) associé(s) sont fournis à la MOA en fin de projet.

Contenu

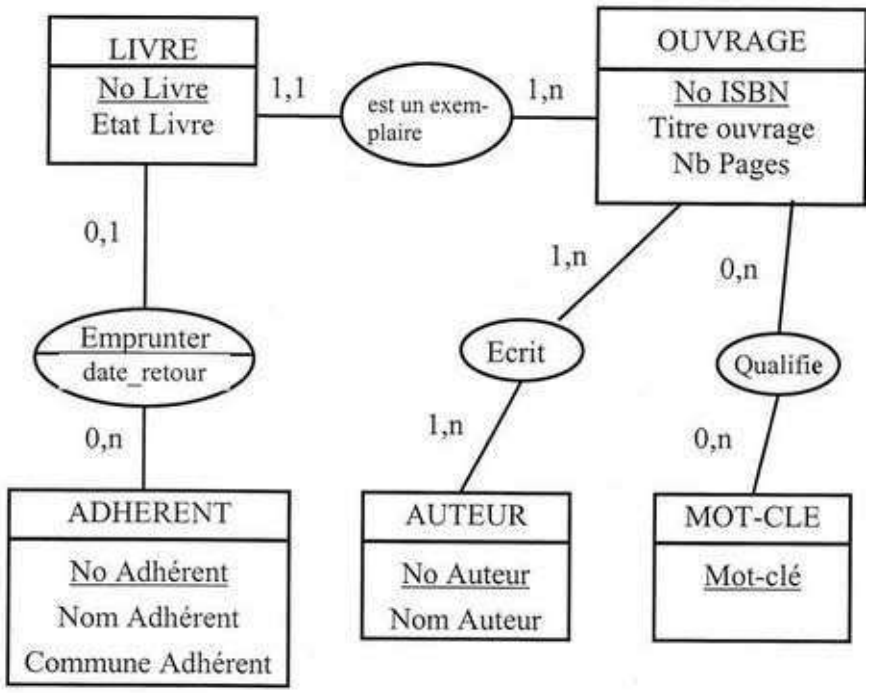
Le contenu de la documentation technique varie fortement en fonction de la structure et de la complexité du logiciel associé. Néanmoins, nous allons décrire les aspects qu'on y retrouve le plus souvent.

> Les paragraphes ci-dessous ne constituent pas un plan-type de documentation technique.

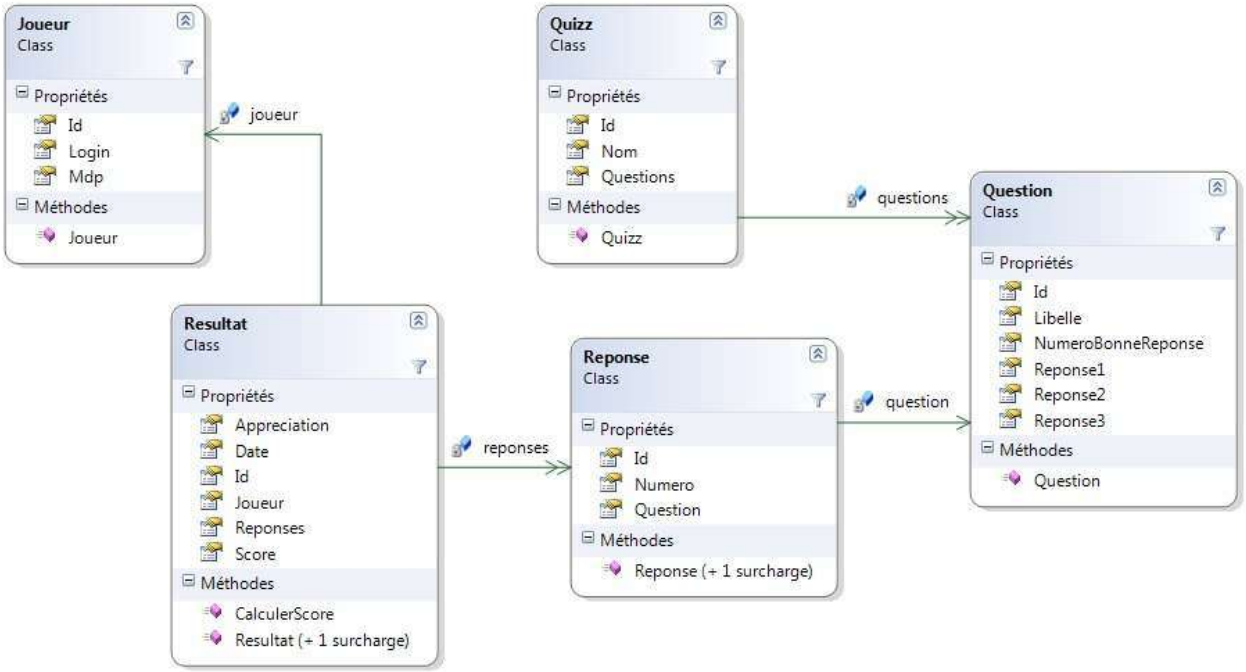
Modélisation

La documentation technique inclut les informations liées au domaine du logiciel. Elle précise comment les éléments métier ont été modélisées informatiquement au sein du logiciel.

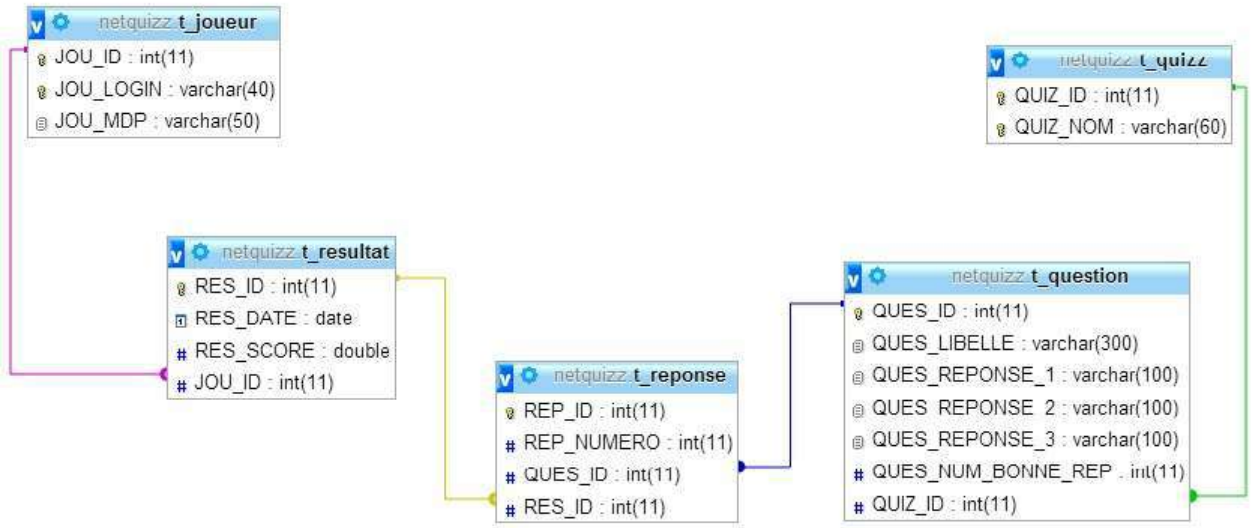
Si le logiciel a fait l'objet d'une modélisation de type entité-association, la documentation technique présente le modèle conceptuel résultat.



Si le logiciel a fait l'objet d'une modélisation orientée objet, la documentation technique inclut une représentation des principales classes (souvent les classes métier) sous la forme d'un diagramme de classes respectant la norme UML.



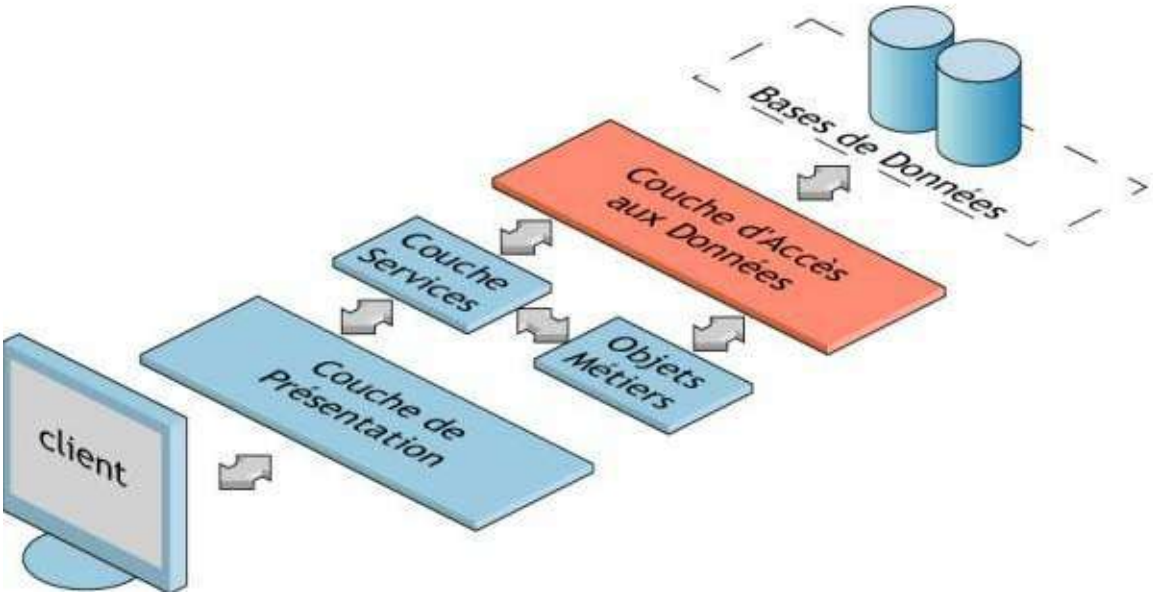
Si le logiciel utilise une base de données, la documentation technique doit présenter le modèle d'organisation des données retenu, le plus souvent sous la forme d'un modèle logique sous forme graphique.



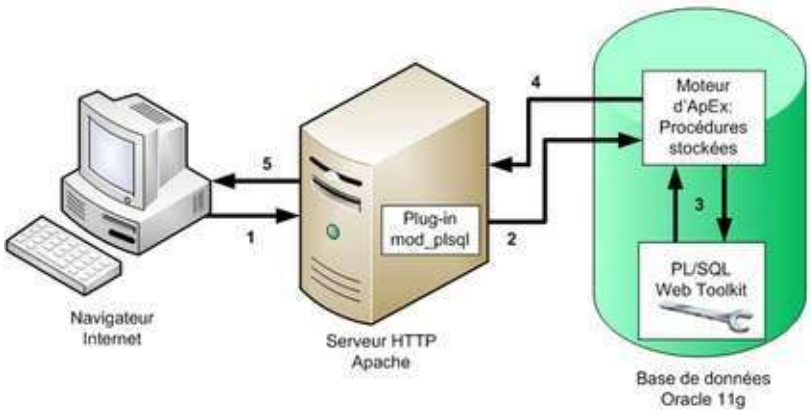
Architecture

La phase d'architecture d'un logiciel permet, en partant des besoins exprimés dans le cahier des charges, de réaliser les grands choix qui structureront le développement : technologies, langages, patrons utilisés, découpage en sous-parties, outils, etc.

La documentation technique doit décrire tous ces choix de conception. L'ajout de schémas est conseillé, par exemple pour illustrer une organisation logique multicouches.



L'implantation physique des différents composants (appelés parfois tiers) sur une ou plusieurs machines doit également être documentée.



Production du code source

Afin d'augmenter la qualité du code source produit, de nombreux logiciels adoptent des normes ou des standards de production du code source : conventions de nommage, formatage du code, etc. Certains peuvent être internes et spécifiques au logiciel, d'autres sont reprises de normes existantes (exemples : normes PSR-x pour PHP).

Afin que les nouveaux développeurs les connaissent et les respectent, ces normes et standards doivent être présentés dans la documentation technique.

Génération

Le processus de génération ("build") permet de passer des fichiers sources du logiciel aux éléments exécutables. Elle inclut souvent une phase de compilation du code source.

Sa complexité est liée à celle du logiciel. Dans les cas simples, toute la génération est effectuée de manière transparente par l'IDE utilisé. Dans d'autres, elle se fait en plusieurs étapes et doit donc être documentée.

Déploiement

La documentation technique doit indiquer comment s'effectue le déploiement du logiciel, c'est-à-dire l'installation de ses différents composants sur la ou les machines nécessaire(s). Là encore, cette étape peut être plus ou moins complexe et nécessiter plus ou moins de documentation.

Documentation du code source

Il est également possible de documenter un logiciel directement depuis son code source en y ajoutant des commentaires (voir plus haut). Certains langages disposent d'un format spécial de commentaire permettant de créer une documentation auto-générée

Le langage Java a le premier introduit une technique de documentation du code source basée sur l'ajout de commentaires utilisant un format spécial. En exécutant un outil du JDK appelé **javadoc**, on obtient une documentation du code source au format HTML.

Voici un exemple de méthode Java documentée au format javadoc.

```
/**
 * Valide un mouvement de jeu d'Echecs.
 *
 * @param leDepuisFile   File de la pièce à déplacer
 * @param leDepuisRangée Rangée de la pièce à déplacer
 * @param leVersFile     File de la case de destination
 * @param leVersRangée   Rangée de la case de destination
 * @return vrai si le mouvement d'échec est valide ou faux si invalide
 */
boolean estUnDeplacementValide(int leDepuisFile, int leDepuisRangée, int leVersFile, i
nt leVersRangée)
{
   // ...
}
```

L'avantage de cette approche est qu'elle facilite la documentation du code par les développeurs, au fur et à mesure de son écriture. Depuis, de nombreux langages ont repris l'idée.

Voici un exemple de documentation d'une classe C#, qui utilise une syntaxe légèrement différente.

```
/// <summary>
/// Classe modélisant un adversaire de Chifoumi
/// </summary>
public class Adversaire
{
    /// <summary>
    /// Générateur de nombres aléatoires
    /// Utilisé pour simuler le choix d'un signe : pierre, feuille ou ciseau
    /// </summary>
    private static Random rng = new Random();

    /// <summary>
    /// Fait choisir aléatoirement un coup à l'adversaire
    /// Les coups possibles sont : "pierre", "feuille", "ciseau"
    /// </summary>
    /// <returns>Le coup choisi</returns>
    public string ChoisirCoup()
    {
       // ...
    }
}
```

La documentation utilisateur

Rôle

Contrairement à la documentation technique, la documentation d'utilisation ne vise pas à faire comprendre comment le logiciel est conçu. Son objectif est d'apprendre à l'utilisateur à se servir du logiciel.

La documentation d'utilisation doit être :

- Utile : une information exacte, mais inutile, ne fait que renforcer le sentiment d'inutilité et gêne la recherche de l'information pertinente ;
- Agréable : sa forme doit favoriser la clarté et mettre en avant les préoccupations de l'utilisateur et non pas les caractéristiques techniques du produit.

Public visé

Le public visé est l'ensemble des utilisateurs du logiciel. Selon le contexte d'utilisation, les utilisateurs du logiciel à documenter peuvent avoir des connaissances en informatique (exemples : cas d'un IDE ou d'un outil de SCM). Cependant, on supposera le plus souvent que le public visé n'est pas un public d'informaticiens.

Conséquence essentielle : toute information trop technique est à bannir de la documentation d'utilisation. Pas question d'aborder l'architecture MVC ou les design patterns employés : ces éléments ont leur place dans la documentation technique.

D'une manière générale, s'adapter aux connaissances du public visé constitue la principale difficulté de la rédaction de la documentation d'utilisation.

Formes possibles

Manuel utilisateur

La forme la plus classique de la documentation d'utilisation consiste à rédiger un manuel utilisateur, le plus souvent sous la forme d'un document bureautique. Ce document est structuré et permet aux utilisateurs de retrouver les informations qu'ils recherchent. Il intègre très souvent des captures d'écran afin d'illustrer le propos.

Un manuel utilisateur peut être organisé de deux façons :

- **Guide d'utilisation** : ce mode d'organisation décompose la documentation en grandes fonctionnalités décrites pas à pas et dans l'ordre de leur utilisation. Exemple pour un logiciel de finances personnelles : création d'un compte, ajout d'écritures, pointage...

Cette organisation plaît souvent aux utilisateurs car elle leur permet d'accéder facilement aux informations essentielles. En revanche, s'informer sur une fonctionnalité avancée ou un détail complexe peut s'avérer difficile.

- **Manuel de référence** : dans ce mode d'organisation, on décrit une par une chaque fonctionnalité du logiciel, sans se préoccuper de leur ordre ou de leur fréquence d'utilisation. Par exemple, on décrit l'un après l'autre chacun des boutons d'une barre de boutons, alors que certains sont plus "importants" que d'autres. Cette organisation suit la logique du créateur du logiciel plutôt que celle de son utilisateur. Elle est en général moins appréciée de ces derniers.

Tutoriel

De plus en plus souvent, la documentation d'utilisation inclut un ou plusieurs tutoriel(s), destinés à faciliter la prise en main initiale du logiciel. Un tutoriel est un guide pédagogique constitué d'instructions détaillées pas à pas en vue d'objectifs simples.

Le tutoriel a l'avantage de "prendre l'utilisateur par la main" afin de l'aider à réaliser ses premiers pas avec le logiciel qu'il découvre, sans l'obliger à parcourir un manuel utilisateur plus ou moins volumineux. Il peut prendre la forme d'un document texte, ou bien d'une vidéo ou d'un exercice interactif. Cependant, il est illusoire de vouloir documenter l'intégralité d'un logiciel en accumulant les tutoriels.

FAQ

Une Foire Aux Questions (en anglais *Frequently Asked questions*) est une liste de questions/réponses sur un sujet. Elle peut faire partie de la documentation d'utilisation d'un logiciel.

La création d'une FAQ permet d'éviter que les mêmes questions soient régulièrement posées.

Aide en ligne

L'aide en ligne est une forme de documentation d'utilisation accessible depuis un ordinateur. Il peut s'agir d'une partie de la documentation publiée sur Internet sous un format hypertexte.

Quand une section de l'aide en ligne est accessible facilement depuis la fonctionnalité d'un logiciel qu'elle concerne, elle est appelée aide contextuelle ou aide en ligne contextuelle. Les principaix formats d'aide en ligne sont le HTML et le PDF. Microsoft a publié plusieurs

formats pour l'aide en ligne des logiciels tourant sous Windows : HLP, CHM ou encore MAML.

Un moyen simple et efficace de fournir une aide en ligne consiste à définir des infobulles (*tooltips*). Elles permettent de décrire succinctement une fonctionnalité par survol du curseur.

Conseils de rédaction

Structure

Qu'elle soit technique ou d'utilisation, toute documentation doit absolument être écrite de manière **structurée**, afin de faciliter l'accès à une information précise.

Structurer un document bureautique signifie :

- Le décomposer en paragraphes suivant une organisation hiérarchique.
- Utiliser des styles de titre, une table des matières, des références...

Niveau de langage

Comme dit plus haut, le public visé n'a pas forcément d'expérience informatique : il faut bannir les explications trop techniques et penser à définir les principaux termes et le "jargon" utilisé.

Une documentation doit être rédigée dans une langue simple, pour être compris de tous, y compris de personnes étrangères apprenant la langue.