http://www.capitchilog.fr

Développer en solo en mode projet

ou comment prendre les bonnes habitudes de développer en mode projet même si on est seul et sans collègue

Date	Version	Auteur(s)	Commentaires
05/10/2018	1.0	Serge COUDÉ	Première version du document

Introduction

Lorsque nous débutons la programmation, même si nous suivons des cours, que cela soit en ligne, en amphithéâtre, dans une classe, dans sa chambre, etc., nous nous retrouvons souvent seul devant notre clavier et notre écran. Si nous ne sommes pas suffisamment accompagné, nous pouvons très rapidement prendre de mauvaises habitudes sur la façon de s'organiser et/ou développer, en tout cas prendre des habitudes non conformes à celles présentes (généralement...) en entreprise. Cela peut amener un décalage important entre le développeur et le reste de l'équipe et éventuellement lui faire interrompre sa période d'essai de façon prématurée.

L'objectif de ce document est de vous présenter comment un projet est mené (de façon générale) dans une équipe composée de plusieurs personnes et vous donner des méthodes pour que vous, étant seul, puissiez mettre en œuvre ces façons de procéder pour une meilleure intégration dans l'entreprise ultérieurement.

Cette présentation propose une organisation particulière, fruit de quelques années d'expérience en ESN/SSII (Entreprise de Services Numériques / Société de Services en Ingénierie Informatique) et Startups. Elle n'est pas une norme (loin de là!) mais elle vous permettra de vous familiariser avec les organisations mises en place la plus part du temps dans les entreprises.

La première partie décrira la vie et le déroulement d'une prestation classique, la seconde vous présentera comment, en « solo », coller au plus près d'une démarche « projet » que l'on retrouve en entreprise.

Table des matières

Introduction....1
Organisation et fonctionnement d'un projet....3
La commande....3
Le Cahier des Charges....3
Constitution de l'équipe projet....3
Démarrage de la réalisation....4
La méthode SCRUM et la gestion KANBAN....4
Le Sprint....6
La Qualification....6
L'Intégration....6
Déploiement et Recette....7
Conseils pour mener en solo un projet....7
Appréciation du projet....8
Cas de la possession d'un descriptif....8
Sans indication....8
Contenu du Cahier des Charges....8
Analyse et Etude....9
Le Dossier d'Architecture Technique (DAT)....11
Organisation du développement....13
Les bonnes pratiques à mettre en œuvre....14
Persistance du code....14
Partitionner son disque....14
Utiliser un outil de « versionning »....14
Maintenabilité du code....15
Aérer son code....15
Nommer ses variables, fonctions, classes et méthodes....16
Commenter et documenter son code....17
Structurer son code....22
Utiliser des instructions pérennes dans le temps....23
Améliorer la qualité de son code....23
Utiliser le debugger....23
Les tests unitaires....24
l'intégration continue....26
Exploitabilité de l'application....27
Modification de paramètres....27
Supervision des logs....28
Scalabilité de l'application....32
Scalabilité = clusterisation....32
Scalabilité = architecture n-tiers....32
Conclusion....33

Organisation et fonctionnement d'un projet

Un projet dans une ESN/SSII est généralement le fruit d'une commande.

La commande

La commande est passée soit en interne par un autre service de l'entreprise, soit par un client extérieur. De toute façon, la méthode appliquée est la même, seule diffère la façon dont est prise en compte la demande : en interne, il pourra éventuellement y avoir moins de « pression » que dans le cas d'un client externe, la réputation de l'entreprise pouvant être mise à mal si la commande n'était pas honorée correctement auprès du client externe.

J'utiliserai dans la suite de ce document le terme de client de façon indifférente, dans le cas d'une demande interne ou externe.

Une commande est généralement conçue comme une demande d'un client qui a un besoin. Ce besoin doit être formulé pour que l'équipe projet puisse en prendre connaissance et y répondre. Pour cela, le client va, avec ou sans l'aide de l'ESN, définir son besoin qui sera transmis par écrit à cette dernière. Généralement, les commerciaux de l'ESN sont en charge des premiers échanges avec le client si celui-ci est externe à l'entreprise. A l'aide de technico-commerciaux, un nombre de jour/homme va être défini, avec divers profils impliquant des montants/jour/homme différents. Par exemple, un chef de projet devrait avoir un montant plus élevé pour une journée de prestation qu'un développeur. La somme de ces jours/hommes + un pourcentage de marge (on est là pour le business!) donnera au final le montant total que se verra proposé le client pour la réalisation de sa commande. Evidemment, il y a, avant toute décision prise, des négociations entre le client et les commerciaux...

Vient alors la rédaction du Cahier des Charges.

Le Cahier des Charges

Le Cahier des Charges peut être éventuellement rédigé par des chefs de projet ou bien des technicaux-commerciaux avant la décision prise par le client, afin de montrer à ce dernier la volonté de l'ESN à travailler avec lui. Généralement, le Cahier des Charges est rédigé a posteriori ou durant la conclusion de l'entente.

Le Cahier des Charges est un document qui va reprendre la totalité du besoin du client, en décrivant les besoins, les contraintes techniques, le périmètre et l'environnement de la prestation à réaliser, les objectifs devant être atteints (installation, fonctionnalités, livrables à fournir, etc.) et l'ensemble des solutions que l'ESN va mettre en œuvre pour répondre à la demande.

Le Cahier des Charges est un document très important, car il s'agit de la documentation servant de guide à la fois à l'équipe de développement mais aussi au client. Ce document sera signé par les deux parties, validant in fine le lancement de la prestation.

Constitution de l'équipe projet

Au moment de la négociation entre l'ESN et le client, une équipe projet commence à être définie. Elle va être constituée de différents profils en fonction du projet. Dans le cas de très gros projets, un Directeur de Projet sera désigné pour définir l'équipe qui travaillera avec lui. Plusieurs Chefs de Projet seront assignés à l'équipe pour répondre à divers problématiques.

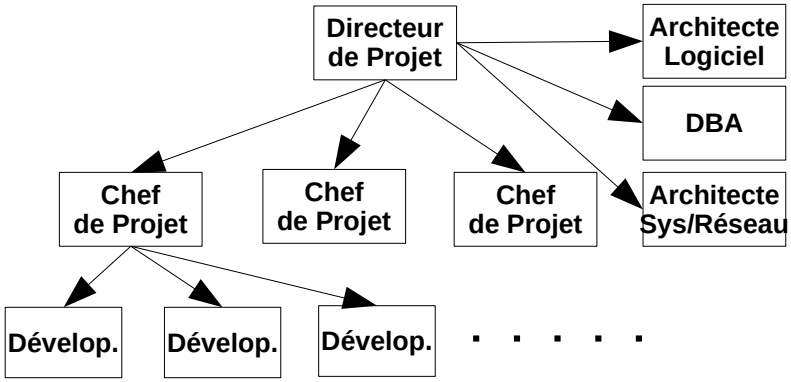
On observe là une première caractéristique d'une gestion de projet : le découpage de tâches en sous tâches

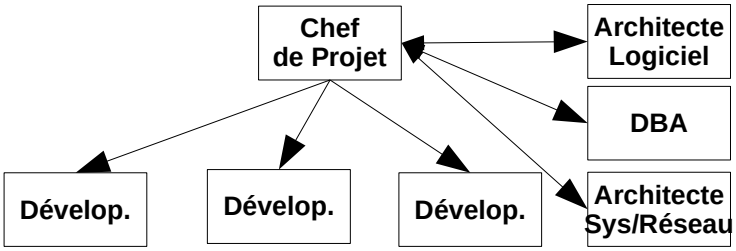
Des développeurs seront intégrés dans les diverses équipes du projet en fonction de leur expérience sur les domaines abordés.

Eventuellement, l'équipe du projet pourra avoir en support un Architecte Logiciel, un Architecte Base de Données (DBA) et un Architecte Système et/ou Réseau si le projet a besoin d'experts dans ces domaines. Ces derniers interviendront dans la rédaction du Cahier des Charges sur leur partie et conseillerons les équipes durant la phase de développement.

Sur les projets de taille standard, seul un Chef de Projet est désigné et constitue son équipe de développeurs, en faisant appel si besoin aux divers experts sus-mentionnés le cas échéant.

Le Chef de Projet peut intervenir dans la rédaction du Cahier des Charges, plus rarement les développeurs, sauf si ces derniers ont des profils « seniors » et sont reconnus pour leur expertise.





Démarrage de la réalisation

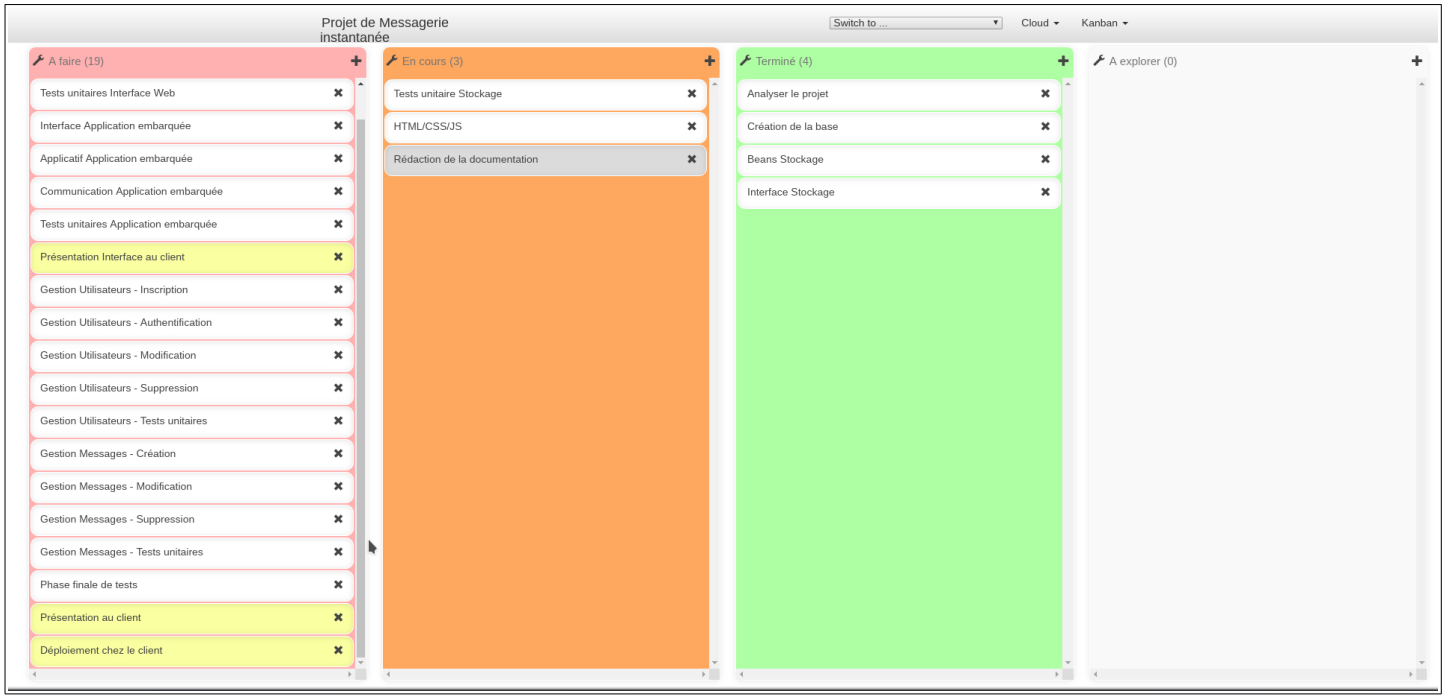
Une fois le Cahier des Charges validé par les deux parties, l'équipe du projet constituée, le Directeur de Projet ou le Chef de Projet répartit les tâches à réaliser. Prenons le cas pour être le plus simple d'un projet ne nécessitant qu'un Chef de Projet et son équipe de développeurs.

Plusieurs méthodes de démarche Projet sont envisageables, comme par exemple la démarche en V (qui a longtemps était la démarche mise en œuvre de façon classique) ou bien une approche « agile » comme SCRUM, largement répandue maintenant. Prenons le cas de SCRUM.

La méthode SCRUM et la gestion KANBAN

Le Chef de Projet va devoir, après analyse du Cahier des Charges (éventuellement aidé des développeurs pour cette analyse) définir les tâches à réaliser. Il va découper le projet en tâches distinctes. Chaque tâche se verra également découpée en sous-tâches, elles-mêmes découpées en sous-sous-tâches, etc. jusqu'à arriver à un niveau que pourra réaliser un développeur dans un temps relativement court. Ces tâches, sous-tâches, etc. ont potentiellement des dépendances entre-elles, par exemple la tâche T04 doit être réalisée avant la tâche T05 car cette dernière dépend de la T04. Il arrive que la T05 soit réalisée avant la T04 et pour des besoins de tests on créé rapidement une maquette de la T04 qui renvoie des valeurs bidons, dans ce cas on parle de *bouchonnage*. Certaines des tâches peuvent être réalisées en parallèle, comme par exemple la tâche qui demande le développement du connecteur aux données et la tâche qui demande la création de l'interface web.

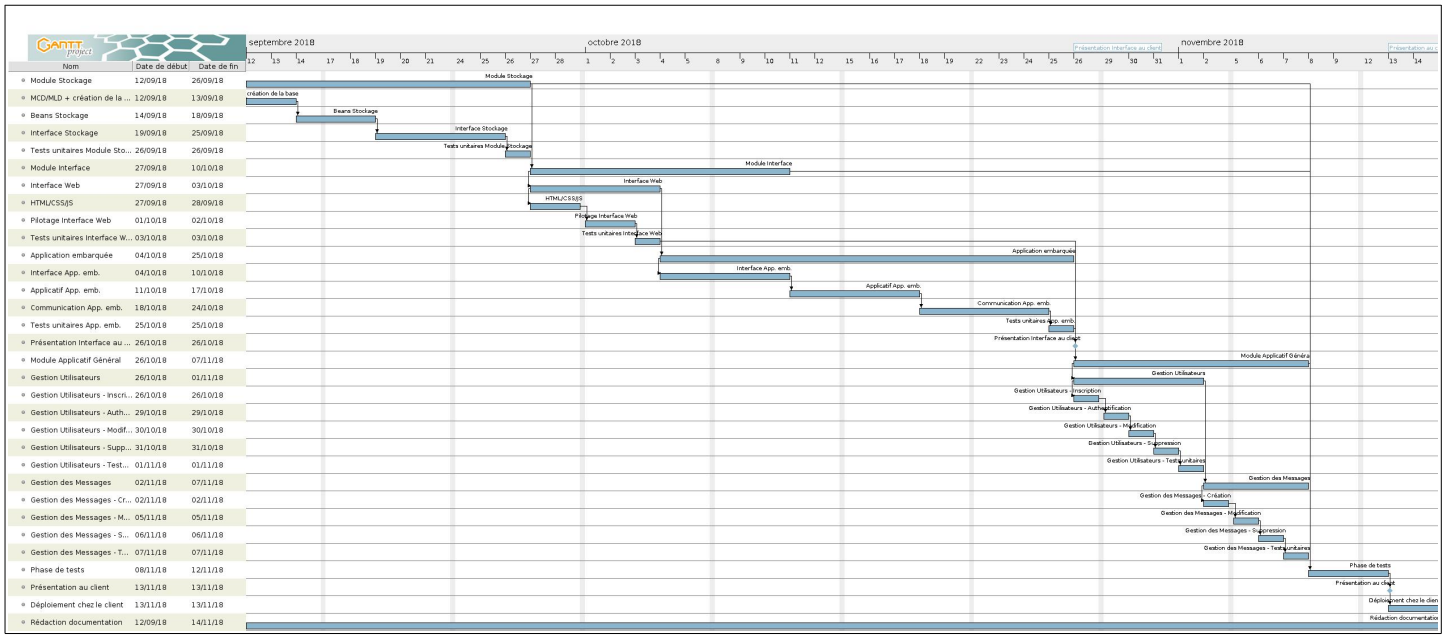
Ces tâches, sous-tâches, etc. vont être posées sous forme d'étiquettes sur un tableau de type Kanban. Ce tableau comporte des colonnes « A faire », « En stand-by », « En cours », « Alerte », « Réalisée », etc. dans lesquelles les étiquettes « tâches » sont placées en fonction de leur état. Chaque jour, le Chef de Projet réunit son équipe et ensemble déplacent les tâches en fonction de l'avancé des réalisations de chacun.



Exemple de tableau Kanban

Exemple de logiciel de méthode Kanban libre et gratuit très simple à installer et à utiliser : My-personal-kanban (https://greggigon.github.io/my-personal-kanban/)
Page de téléchargement : sur la page du site
Tutoriel : vidéo sur la page du site

En même temps, le Chef de Projet rédige la planification des différentes tâches dans le temps. Il peut utiliser un diagramme de Gantt pour avoir une bonne vision de l'avancement du projet. Ainsi, il est en mesure de piloter la réalisation des tâches en fonction du temps de réalisation défini au départ, qui ne sera pas forcément le temps réel passé au final ! Cela permet également de proposer une date de livraison finale au client dès le début du projet, date qui, la plus part du temps est... reculée !!!



Exemple de diagramme de Gantt d'un projet

Exemple de logiciel de gestion de projet libre et gratuit simple à utiliser : ganttProject (https://www.ganttproject.biz/)
Page de téléchargement : https://www.ganttproject.biz/download/free
Tutoriel : http://eduscol.education.fr/sti/sites/eduscol.education.fr.sti/files/ressources/pedagogiques/3364/3364-tutoriel-gantt-project-version-26-vers-17janv2014.pdf

Le Sprint

Dans un projet, il se peut que des parties de la prestation puissent être livrées avant la finalisation du projet ou bien que l'équipe ait à faire à un moment donné un point global sur le projet avec des développement qui sont « fonctionnels ».

Dans le cas d'une livraison partielle, le client voit l'avancé de sa demande et peut faire évoluer « à la marge » les développements à venir si besoin en faisant bien attention de ne pas modifier trop le projet car sinon tout est à re-évaluer (durée, découpage, équipe, et donc... tarification !).

Dans le cas d'un point, l'équipe arrête les développements et analyse les réalisations déjà faites.

Dans le jargon SCRUM, la période de développement d'une partie de la prestation se nomme un *sprint*. Elle a pour durée environ 2 à 4 semaines en moyenne. Ainsi un projet est découpé en « sprints », que l'on retrouve dans le diagramme de Gantt.

Une fois que le sprint est terminé, il faut livrer au client. Plusieurs étapes peuvent (et devraient...) être mises en œuvre avant cette livraison. Tout d'abord, il y a l'étape de « qualification ».

La Qualification

Des développeurs (idéalement ne faisant pas partie de l'équipe du projet) passent en revue l'ensemble des développements du sprint pour vérifier si les fonctionnalités définies dans le Cahier des Charges sont bien présentent dans les livrables et si le code fonctionne comme il a été défini dans le cahier.

Techniquement, ils déploient la partie de l'application sur des serveurs ou sur des machines distinctes de celles des développeurs et testent le bon fonctionnement de ce qui doit être livré.

Ils rédigent ensuite un document qui récapitule tous les tests effectués et les remarques éventuelles de non fonctionnement.

Dans le cas de dysfonctionnements constatés, les développeurs de l'équipe projet corrigent leur code en fonction des remarques faîtes dans le document. Une fois les corrections apportées, on repasse une nouvelle phase de qualification, etc. jusqu'à ce qu'il n'y ait plus d'erreur...

Une fois la phase de qualification passée avec succès, le code passe à l'étape d'Intégration.

L'Intégration

Des informaticiens (par forcément développeur, cela peut être des administrateurs Système/réseau) récupèrent ensuite le code fourni par l'équipe avec la documentation (souvent succincte...) d'installation que cette dernière leur fourni. Ils installent le code sur des machines possédant les caractéristiques les plus proches de celles du client, le paramètre si besoin (changement des identifiant, adresse du serveur de bases de données, etc.) et vérifient qu'il fonctionne correctement.

Dans le même temps, ils rédigent une documentation officielle du modus operandi de l'installation et de la configuration ainsi que des documentations à destination des exploitants et des utilisateurs du client. On parlera alors du manuel de l'utilisateur, du manuel de l'exploitant, etc.

Le manuel d'installation décrira l'ensemble des actions à mener pour déployer le code, avec un focus particulier sur les pré-requis éventuels comme par exemple la version de PHP, Java ou Tomcat, la quantité de mémoire vive des serveurs, leur capacité de stockage, la puissance de processeur, l'architecture réseau à mettre en place

au préalable (par exemple des répartiteurs de charge [loadbalancer]), etc. Il précisera les fichiers de configuration à modifier pour s'adapter au client.

Le manuel d'exploitation décrira l'ensemble des procédures que l'exploitant aura à effectuer pour maintenir l'application en état de fonctionnement (MCO [Maintient en Condition Opérationnelle]). Cela sera par exemple les instructions à suivre pour lancer et arrêter l'application, les sources de données à monitorer/surveiller (logs), les données à sauvegarder et comment lancer les sauvegardes, à quelle fréquence, les services à superviser, etc. Il précisera les divers cas de problèmes potentiels et les actions curatives à mener pour les résoudre.

Le manuel de l'utilisateur montrera l'ensemble des fonctionnalités apporté par l'application et comment y avoir accès. Il ne sera pas orienté « technique informatique » comme les deux précédents, seulement « fonctionnel » et comportera (éventuellement) des captures d'écran pour une meilleure compréhension par les utilisateurs futurs.

Une fois toutes ces documentations rédigées et le mode opératoire de l'installation validé, on procède à l'installation chez le client et à sa « validation ».

Déploiement et Recette

Deux cas de figures peuvent se présenter, soit l'installation se fait chez le client par des employés du client, soit elle est effectuée par l'équipe d'intégration voire l'équipe du projet.

Dans les deux cas, les informaticiens suivent les consignes du document d'installation.

Une fois le déploiement effectué, avant la mise en production, des tests sont réalisés, il s'agit de la « recette ». La recette est une nouvelle phase de tests effectuée chez le client cette fois-ci et qui permet de vérifier que l'application est fonctionnelle dans l'environnement du client, qu'elle propose toutes les fonctionnalités demandées et qu'il n'y a pas de bug avéré. L'ensemble des tests est consigné dans un document que l'on nomme « Cahier de Recette ». Ce document est ensuite signé par les deux parties pour valider la livraison.

Dans le cas où des problèmes apparaissent au moment du déploiement, des « réserves » sont émises : il s'agit de décrire les problèmes apparus et les informations sont renvoyées à l'équipe de développement et/ou à l'équipe d'intégration, pour correction. Normalement, tout le cycle décrit préalablement est à refaire avant de relivrer. Tant que la recette n'est pas validée, cela peut entraîner des « retards de paiement » de la part du client, ce qui peut être préjudiciable pour l'ESN...

Une fois la livraison finale validée par le client (la totalité des sprints), une nouvelle phase peut alors éventuellement commencer, une phase de maintenance de garantie : l'ESN se porte garant du fonctionnement de son code durant une période donnée. En cas de bug découvert ultérieurement, elle s'engage à le corriger durant cette période. Cela implique de pourvoir mobiliser éventuellement une nouvelle équipe ou d'un ensemble de développeurs qui doivent au mieux se replonger dans le code car ce sont eux qui ont assuré le développement initial, au pire devoir s'approprier les développements réalisés par autrui. Et là, **il est primordial que les développements initiaux aient été réalisés dans les règles de l'art** (façon de coder, documentation, etc. cf annexe) afin que la reprise du code se fasse la plus rapidement et le plus facilement possible.

Il en va des mêmes obligations de « best practices » (bonnes pratiques) de codage dans le cas d'une nouvelle commande du client qui souhaite ajouter de nouvelles fonctionnalités à son application car il apprécie celle que l'ESN lui a déjà réalisée.

Conseils pour mener en solo un projet

Pour bien « apprécier » les divers conseils qui vont suivre, il est opportun d'avoir lu comment se déroulait un projet dans une entreprise et avoir bien retenu la succession des étapes. L'ensemble des phases présentées précédemment sont dépendantes les unes des autres, et une mauvaise réalisation ou l'absence de l'une d'entre-elles peut être préjudiciable à la bonne réalisation de la globalité du projet.

Appréciation du projet

Deux cas de figures peuvent de présenter à vous :

- vous avez un descriptif (la commande ou expression de besoin) qui vous est fournie
- vous n'avez aucune indication

Dans les deux cas, il vous faut rédiger un Cahier des Charges. Sa rédaction vous permettra également d'avoir des informations à proposer dans le cas éventuel d'une soutenance durant votre formation, voire à le fournir pour une explication de votre réalisation.

Cas de la possession d'un descriptif

Si vous avez un descriptif, alors amorcez la rédaction du Cahier des Charges après avoir bien étudié le document fourni, puis une fois le premier jet effectué, discutez avec le client pour valider certains points qui ne seraient pas bien compris entre les deux parties. Une fois le document finalisé, faites-en une présentation « officielle » au client.

Sans indication

Il va falloir aller à la pêche aux informations. Cette situation est la plus délicate des deux car il va falloir que vous soyez très diplomate, pédagogue et persévérant : le client a une idée peut-être flou de son besoin et c'est à vous de savoir coucher sur le papier la description de ce qu'il souhaite. Il y aura certainement beaucoup d'aller-retours entre lui et vous. Eventuellement, si vous voyez que des salariés de l'entreprise peuvent être des personnes ressources, n'hésitez pas à les « interroger » en ayant préalablement demandé l'accord à votre correspondant principal.

Au fur et à mesure, vous allez vous faire une idée de plus en plus précise de ses attentes et la rédaction du Cahier des Charges va pouvoir commencer. Comme pour le cas n°1, réalisez une présentation du cahier des charge avant sa validation pour être sûr que tout est clair et carré entre vous et le client.

Contenu du Cahier des Charges

Le Cahier des Charges doit présenter l'ensemble de la prestation réalisée.

Très important : vous devez absolument « versionner » votre document dès le départ. Cela peut se faire sous forme d'un tableau en début de document avec la date, le n° de version, le(s) rédacteur(s), les modifications apportées. Ainsi, il sera aisé à tout lecteur de savoir ce qui a été ajouté depuis sa dernière lecture.

Généralement, le Cahier des Charges commence par une présentation du contexte d'intervention :

- présentation du client : histoire et/ou domaine d' évolution du client (entreprise dans la peinture automatisée, etc.)
- contexte de la prestation : pourquoi le client fait appel à vous
- objet de la prestation : ce que le client vous demande de réaliser
- cadre de la prestation : dans quel environnement se déroule la prestation (environnement technique, etc.) et limites de la réalisation (on parlera du « scope » de la prestation)

Si des contraintes sont données, alors elles pourront apparaître dans le document de façon bien identifiées.

Une fois cette partie rédigée, vous pouvez alors décrire l'ensemble des protagonistes. Vous présenterez les différentes personnes côté client avec leur nom, email, téléphone et fonction.

Puisque vous êtes seul, vous vous noterez seulement vous à la suite, avec les mêmes informations que pour le client. Mais dans une équipe, vous présenterez l'ensemble des vos collègues (toute personne de l'ESN qui interviendra, même ponctuellement sur le projet).

Une fois l'ensemble des informations données, le Cahier des Charges rentre dans le vif du sujet.

Avant de pouvoir rédiger la partie suivante, vous devez avoir analysé en détail les besoins et trouvé les solutions techniques pour répondre à la demande. (voir chapitre suivant pour une proposition de méthode)

Vous présenterez d'abord les différents modules fonctionnels de l'application, avec si possible des maquettes d'interfaces (mookup) pour que le client se projette dans l'application plus facilement. Les différents modules doivent répondre complètement à la demande du client : ils doivent couvrir l'ensemble du « scope » des fonctionnalités.

Vous présenterez ensuite les différents découpages techniques que vous avez décidé (par exemple la partie interface web, la partie stockage des données, la partie moteur de l'application, etc.). Si besoin, vous présenterez également l'architecture réseau définie avec les différents composants comme un répartiteur de charge, des pare-feux, des commutateurs, un cluster de serveurs web, un ou plusieurs serveurs de base de données, les serveurs applicatifs etc.

Vous approfondirez la description des briques logicielles mises en œuvre, comme par exemple la présence d'un serveur LAMP ou Java/Tomcat. Vous devez à ce stade déjà présenter les briques que vous allez développer dans les technos que vous aurez choisies. Par exemple, expliciter le module de stockage que vous allez implémenter, son interface d'utilisation, le module d'affichage des résultats, comment il est structuré, les interfaces éventuelles entre l'application et d'autres applications externes (échanges de fichiers, connexion FTP, interrogation de Services Web, pages web, etc.).

Si vous étiez dans une équipe de développement, vos collègues devraient pouvoir savoir s'organiser pour démarrer le développement à la seule lecture de ce document.
Utilisez un maximum de synoptiques pour que le client visualise bien les architectures applicative, système et réseau que vous allez mettre en œuvre. Un dessin vaut très souvent mieux qu'un long discourt ou une longue lecture !

Une fois cette grosse partie rédigée, vous pouvez si besoin ajouter une sitographie en indiquant les divers sites des applications que vous allez utiliser pour mettre en place votre prestation, comme par exemple un lien vers Apache, Nginx, Node.js, MySQL, Oracle, etc. Le client, s'il en a les compétences et les connaissances, pourra apprécier vos choix effectués.

Voilà, votre Cahier des Charges est rédigé, vous pouvez maintenant le présenter au client lors d'une réunion. Je vous conseille de lui envoyer quelques jours auparavant pour qu'il puisse en prendre connaissance et que la présentation soit alors un moment d'échange plutôt que de découverte.

Il se peut que vous ayez à effectuer des retouches selon les cas, donc veillez bien à versionner le document ! Une fois que le client et vous êtes d'accord, vous pouvez acter du démarrage du projet.

Analyse et Etude

Pour rédiger le Cahier des Charges, il est nécessaire d'avoir parfaitement en tête ce que vous allez réaliser. C'est, selon moi, une des parties les plus « excitantes » du projet : on crée, on imagine, on conçoit réellement !

Il faut pour cela partir d'une vision globale puis au fur et à mesure aller vers plus de précision. On peut voir cela comme une pelure d'oignon : on commence par les parties générales (les premières choses que les utilisateurs « voient », que les développeurs pensent) puis on va vers le cœur du projet.

Imaginons que nous devions programmer une application de messagerie instantanée tout périphérique possible (ordinateur, tablette, smartphone).

Dans ce cas de figure, il faut dans un premier temps découper fonctionnellement l'application.

Première chose que l'utilisateur va voir, c'est *l'interface*. Elle sera de deux ordres : web pour ordinateur et tablette, application embarquée pour les smartphones. On aura donc une couche « frontend » composée de deux modules différents.

Ensuite, cette application, de part ce qu'elle propose (les messages), doit stocker les informations qu'elle contient. Il va y avoir donc un module de *stockage* qui va faire partie de la couche « backend », celle que l'utilisateur ne va pas voir.

Pour faire la liaison entre l'interface et les données, un troisième module « intermédiaire » est nécessaire, le module *applicatif général* faisant partie lui aussi de la couche « backend ». Ce sera lui qui sera « le cerveau » de l'application. Toute la partie « intelligente » (algorithmie) est fournie par lui.

On est en présence d'un modèle de découpage n-tiers (ici 3-tiers) :

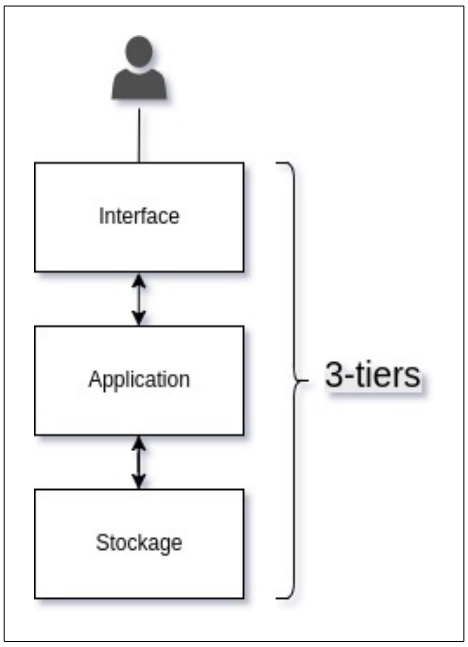


Schéma de découpage en modèle n-tiers de l'application (ici 3-tiers)

Il s'agit d'un schéma très classique. On peut (un peu) le rapprocher du paradigme MVC que les développeurs doivent bien connaître !

Une fois ce premier découpage effectué, il faut réitérer le processus sur chaque module.

Ainsi pour le module *interface*, il va y avoir de facto deux sous modules dans notre cas : le module *interface web* et le module *application embarquée*.

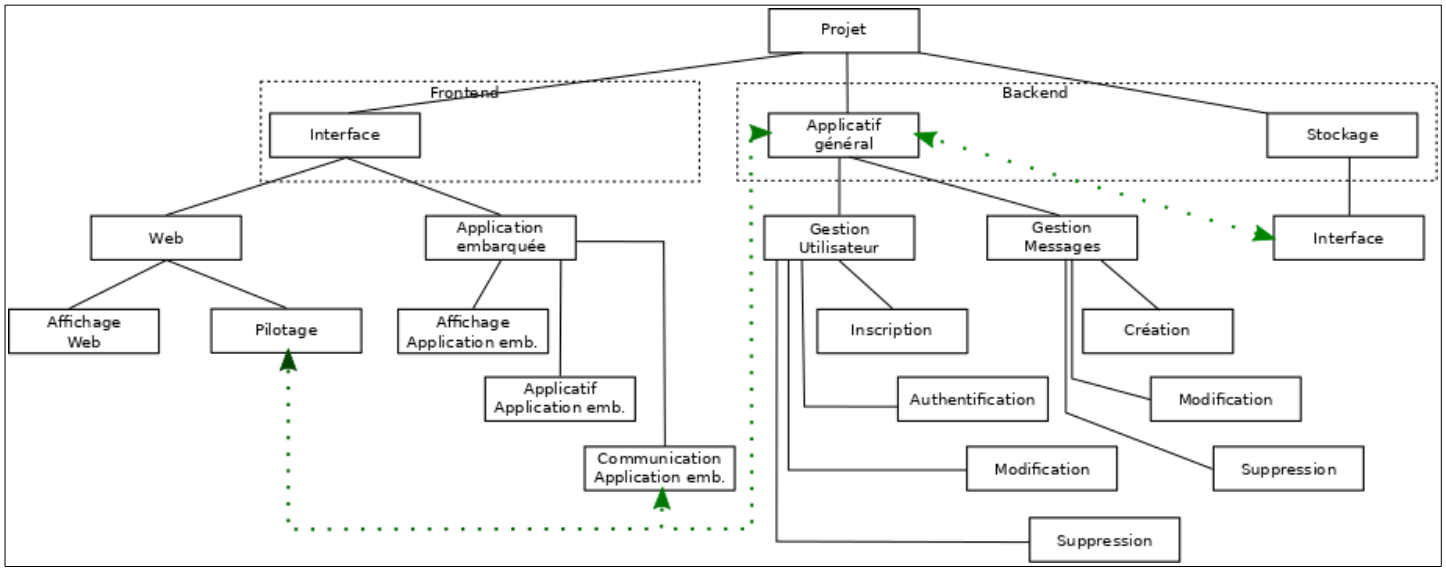
Le module *interface web* va contenir un module *affichage web* et un module *pilotage* qu'utilisera le module *applicatif général*.

Le module *application embarquée* aura lui un module *affichage application*, un module *applicatif application* « léger » et un module *communication* entre l'application embarquée et le module *applicatif général*.

Etc., etc, etc.

On s'aperçoit bien vite qu'un schéma est beaucoup plus parlant !

Schéma d'un découpage du projet sous forme de modules



Une fois ce découpage effectué et rédigé dans le Cahier des Charges, ce dernier peut servir de base de discussion entre le client et vous.

Le Dossier d'Architecture Technique (DAT)

Le Cahier des Charges validé ou en passe de l'être, un document très précieux va devoir être rédigé... par vous (ou le Chef de Projet de l'équipe!) : le DAT. Il s'agit du document qui reprend la partie technique du Cahier des Charges, mais cette fois-ci en détaillant l'ensemble des modules, classe par classe, méthode par méthode, table de la base de données par table, web services (avec les ports d'écoute et les requêtes), etc. si possible. C'est ce document que les développeurs utiliseront quotidiennement pour réaliser le projet.

Le DAT sera structuré en fonction des divers modules, des plus généraux aux plus spécifiques. Prenons l'exemple du module « Stockage » (le plus simple à décrire) :

le DAT devra indiquer pour ce module :

- le type de stockage (système de fichiers, base de données [dans notre cas], services distants, etc.) avec ses caractéristiques (serveur de base de données [mySQL, Oracle, PostgreSQL,etc.] avec les identifiants, paramètres , etc.)
- les MCD (Modèle Conceptuel de Données) / MLD (Modèle Logique de Données) de la base de données
- les classes « systèmes » présentes dans le langage choisi (PDO pour PHP, JDBC pour Java, etc.)
- les classes et les interfaces du projet et leurs méthodes (classe StockageBDD implémentant l'interface Stockage, ainsi l'application pourra « switcher » facilement de système de stockage en utilisant les méthodes de l'interface Stockage...)
- les paramètres du module (données à externaliser le plus possible du code, comme l'adresse du serveur hébergeant le serveur de base de données, les identifiants, les mots de passe, le nom des bases, etc.)

Cette documentation se fera au préalable en utilisant par exemple la méthode UML qui permettra de poser la structure des classes, les méthodes, etc. (Cf cours d'OpenClassrooms : https://openclassrooms.com/fr/courses/2035826-debutez-lanalyse-logicielle-avec-uml)

Vous pouvez décrire ces informations sous forme de tableau pour chaque classe avec sa description, le nom des méthodes, leurs paramètres, ce qu'elles renvoient, etc.

Les MCD / MLD seront plutôt présentés graphiquement pour une lisibilité facilitée. Les développeurs utiliseront quasiment constamment le MLD. Eventuellement un descriptif de chaque table sera également fourni dans le DAT avec les types des champs, leur raison d'être etc.

In fine, un développeur doit (malheureusement !) pouvoir développer l'application juste en suivant les informations du DAT, sans obligatoirement savoir de quoi il en retourne... Vous entendrez peut-être parler dans votre entourage professionnel des « pisseurs de code »... Avec la prolifération des développeurs et les tâches qu'on leur confie, vous comprenez peut-être pourquoi les salaires ne sont plus ce qu'ils étaient...

Reprenons notre exemple du module de Stockage avec l'exemple de la description de l'interface et de la classe :

Interface Stockage					
Type	Méthode	Entrée	Sortie	Exception	Commentaire
statique	getStockage	néant	Objet Stockage	néant	Crée un singleton d'un objet implémentant l'interface Stockage et le retourne
	connecte	néant	booléen / entier	néant	Tente la connexion au système de stockage et renvoie soit vrai si bien connecté soit un code erreur interne à l'application : 1 stockage non accessible, 2 identification incorrecte, etc.
	deconnecte	néant	booléen / entier	néant	Déconnecte le système de stockage et renvoie vrai si bien réalisé soit un code erreur interne à l'application : 1 stockage non accessible, etc.
	authentification	Chaîne identifiant Chaîne mot de passe	Objet Utilisateur / entier	néant	Authentifie l'utilisateur dont l'identifiant et le mot de passe sont passés en argument. Si l'authentification est correcte, renvoie un objet Utilisateur avec ses informations ou bien renvoie un code erreur interne à l'application : 1 stockage inaccessible, 2 identifiant inconnu, 3 mot de passe erroné, etc.

Classe StockageBDD (implémente l'interface Stockage)					
Type	Méthode	Entrée	Sortie	Exception	Commentaire
statique	getStockage	néant	Objet Stockage	néant	Crée un singleton/objet de la classe Stockage et le retourne
	connecte	néant	booléen / entier	néant	Tente la connexion à la base de données et renvoie soit vrai si bien connecté soit un code erreur interne à l'application : 1 base non accessible, 2 identifiant / mot de passe incorrect etc.

Classe StockageBDD (implémente l'interface Stockage)					
	deconnecte	néant	Booléen / entier	néant	Déconnecte la base de données et renvoie vrai si bien réalisé soit un code erreur interne à l'application : 1 base de données non accessible, etc.
	authentification	Chaîne identifiant Chaîne mot de passe	Objet Utilisateur / entier	néant	Compare dans la table Utilisateur le mot de passe associé à l'identifiant au mot de passe fourni en argument. Pour le retour, cf la méthode de l'interface.

Avec cette documentation, tout développeur doit être en mesure d'écrire le code de l'application.

Pour les habitués des bases de données, on peut voir cette description comme le MCD d'une base et le codage comme les scripts implémentant le MCD...

Une fois le DAT rédigé, il n'y a plus qu'à... s'organiser !

Organisation du développement

A cette étape, nous avons une vision globale du projet et l'ensemble des codages à effectuer. Oui, mais comment s'organiser, par quel code (euh bout) commencer ?

Comme vu précédemment dans la vie d'un projet, nous allons pouvoir mettre en pratique par exemple la méthode SCRUM. Rappelez-vous, le diagramme de Gantt, le tableau Kanban, les sprints, etc. !

En vous basant sur les modules, les sous-modules, etc., vous allez pouvoir planifier votre projet. Par exemple

vous pouvez commencer par ce qui est le plus nécessaire, selon moi le stockage de données. Avec un peu de chance, vous aurez utilisé un logiciel de modélisation de schéma de base de données qui vous générera le code SQL de votre base. Vous n'aurez plus qu'à injecter ce code dans le moteur de base de données pour avoir votre base prête à l'emploi !

Votre base étant installée, vous pouvez vous atteler à « inscrire » (dans votre planning, pas tout de suite !) le codage des différents composants (classes, interfaces, etc.).

Ensuite vous pouvez passer peut-être à l'interface web pour pouvoir présenter quelque chose au client rapidement.

Puis au final vous consacrer au module principal, l'applicatif.

Ainsi, vous allez définir l'ensemble des tâches à réaliser regroupées par module, sous-module, etc. A vous de voir vers quelle finesse vous descendez dans la définition des tâches planifiées dans le diagramme de Gantt (cf 5) . Plus c'est grossier moins vous avez le sentiment d'avancer et plus votre « dead line » sera floue ; plus c'est fin, plus vous avez des risques de voir constamment vos tâches modifiées et donc avoir un travail de mise à jour chronophage... Pour chaque tâche, vous allez devoir « pronostiquer » une durée de réalisation. Au début, elles seront plutôt fausses (souvent sous-évaluées), mais avec le temps et l'expérience, vous saurez définir une durée plus proche de la réalité !

Une fois les tâches définies et placées sur un diagramme de Gantt, vous pouvez les créer dans votre tableau Kanban (cf 5) et les positionner dans la colonne « A réaliser ».

Ensuite, à vous de vous atteler aux développements !

A chaque tâche réalisée, pensez obligatoirement à :

- documenter / commenter votre code
- rédiger la documentation correspondante à la tâche réalisée (éventuellement dans le manuel d'installation, d'exploitation, etc. selon le cas)
- déplacer l'étiquette du tableau Kanban dans la bonne colonne
- mettre à jour l'avancement de la tâche (% de réalisation) ou du groupe de tâches dans le diagramme de Gantt

En suivant cette méthode, vous vous retrouvez à 80 % en situation de travail en équipe...

Les bonnes pratiques à mettre en œuvre

Lorsque vous développez, et surtout au début, il est très facile de prendre de mauvaises habitudes, surtout lorsque vous êtes seul ! Eventuellement, dans une équipe, vous allez pouvoir « mimer » les façons de faire de vos autres collègues. Mais, même cela, peut poser problème : vos collègues n'ont peut-être pas forcément eux non plus de bonnes façons de faire (cas vécu !).

Il existe plusieurs domaines dans les bonnes pratiques : il existe plusieurs domaines dans le fait de coder mais tous se regroupent en un seul général : la pérennité du code.

Cette pérennité se décompose en sous-domaines :

- persistance du code : en bref, sa sauvegarde !
- maintenabilité du code : codage simple, clair et efficace, commentaires et documentations
- exploitabilité de l'application : un exploitant doit pouvoir exploiter votre application sans savoir coder, en lisant seulement votre documentation d'exploitation
- scalabilité de l'application : votre application doit pouvoir être installée sur plusieurs serveur et vue comme une seule et même application ou alors pouvoir être « découpée / répartie » sur plusieurs serveurs

Persistance du code

Il n'est (malheureusement...) pas rare que votre ordinateur fasse des siennes, de telle façon que votre cher code se retrouve perdu (plus ou moins à jamais...). Il en va de même de tous vos documents !

Partitionner son disque

Une première bonne pratique qui n'est aucunement liée au code est de partitionner votre disque dur, avec au minimum une partition système et une partition données. Ainsi, en cas de plantage d'un célèbre système d'exploitation (avec l'ancien écran bleu), vous pouvez au moins récupérer vos données. Evidemment, si c'est le disque qui est défaillant, deux partitions ne vont pas changer la donne. Mais le premier cas est le plus fréquent...

Utiliser un outil de « versionning »

Un outil de versionning permet de sauvegarder la totalité des modifications que vous apportez à votre code. Il vous permet alors de revenir « en arrière », de faire des tests ou essais, éventuellement de les supprimer ou de les intégrer dans votre développement principal.

Deux outils sont très utilisés, SVN et Git. SVN a été le premier il y a longtemps, (pas dans une galaxie lointaine , très lointaine, juste sur Terre...) et depuis pas mal d'années maintenant, Git a la préférence des équipes de développement.

Ces outils sont installés sur votre machine ou plus généralement sur un serveur distant (interne à l'entreprise ou dans le cloud [exemple github ou gitlab]). Vous avez un client installé sur votre machine de développement et (comme les shadocks) vous commitez, commitez, commitez... En fait, à chaque fois que vous terminez une

méthode ou plutôt un ensemble qui peut être fonctionnel, alors vous commitez : vous demandez au serveur SVN/Git de prendre en compte l'ensemble des modifications réalisées depuis la précédente prise en compte. C'est une sorte de sauvegarde différentielle. A chaque commit, une version est créée. Vous pouvez alors « naviguer » dans vos commits/versions pour éventuellement revenir en arrière si votre dev ne fonctionne pas ou que vous vous êtes engagé dans une mauvaise piste.

Donc il est quasi obligatoire d'installer ce type d'outil, au pire sur votre ordinateur (mais la panne système ou matérielle sera également fatale), au mieux sur un serveur distant bien sauvegardé. Des services en ligne fournissant ce type de serveurs sont fréquents, vous n'aurez que l'embarras du choix, à moins que vos développements ne soient très privés, dans ce cas un serveur interne est requis !

Pour maîtriser un outils de versionning, pensez au cours (https://openclassrooms.com/fr/courses/2342361-gerez-votre-code-avec-git-et-github)!

Maintenabilité du code

La maintenabilité du code, c'est le fait que « reprendre » votre code soit des plus aisés. Plusieurs cas de reprises existent : vous avez arrêté votre développement il y a un an, et vous devez vous replonger dedans ; vous n'avez pas le temps de continuer vos développements et une ou plusieurs autres personnes vont le récupérer ; vous intégrez une équipe projet et devez rapidement être au « parfum » de ce qui a déjà été fait, dans quel style, etc. ; on vous affecte à un projet réalisé il y a quelques années et qui demande une évolution (fonctionnelle, système, etc.) et, toujours, vous devez rapidement « renter dedans » !

Pour cela, des bonnes pratiques existent, mais ne sont pas tout le temps mises en pratique...

Aérer son code

Il est primordiale que votre code soit aéré. Exemple :

code opaque :

```
function chargeJoursChomes($dateDeb, $dateFin) {
        $res = array();
          foreach(new SimpleXMLElement(file_get_contents(FICHIER_ABS_JOURS_CHOMES))->jour
as $xmlJour) {
            $d = $xmlJour["date"]->__toString();
            if (strlen($d) <= 5)  {
                $annee = substr($dateDeb, 0, 4); $nd = $annee . "-" . $d;
                if ($nd < $dateDeb || $nd > $dateFin) $d = ($annee+1) . "-" . $d;
                else $d = $nd;
            }
                 if ($d >= $dateDeb && $d <= $dateFin) $res[$d] = $xmlJour["libelle"]-
>__toString();
        }
        ksort($res); return $res;
}
```

Nous pouvons constater que lire et comprendre ce code lors de la première lecture n'est pas évident. On ne voit pas facilement où commencent les lignes, où elles se terminent, etc. Et ne parlons pas d'informations qui ne sont pas présentes pour nous faire comprendre d'un seul trait ce que fait cette fonction...

code aéré :

```
function chargeJoursChomes($dateDeb, $dateFin) {
```

```
        $res = array();
        $xmlJours = new SimpleXMLElement(file_get_contents(FICHIER_ABS_JOURS_CHOMES));

        foreach($xmlJours->jour as $xmlJour) {

            $d = $xmlJour["date"]->__toString();

            if (strlen($d) <= 5)  {

                $annee = substr($dateDeb, 0, 4);
                $nd = $annee . "-" . $d;
                if ($nd < $dateDeb || $nd > $dateFin) {
                    $d = ($annee+1) . "-" . $d;
                } else {
                    $d = $nd;
                }

            }

            if ($d >= $dateDeb && $d <= $dateFin) {

                $res[$d] = $xmlJour["libelle"]->__toString();

            }
        }

        ksort($res);
        return $res;
}
```

La lecture et la compréhension de ce code sont déjà plus aisées : la lecture est facilité. Les différentes parties du code sont plus évidentes, on voit bien les conditions, les boucles, etc. Il manque encore des commentaires et de la documentation pour comprendre du premier coup ce que cette fonction réalise.

Nommer ses variables, fonctions, classes et méthodes

Lorsque vous codez, vous utilisez des variables, des fonctions, des classes et vous lisez continuellement votre code. Des personnes ont trouvé un moyen d'écrire du code qui sera facile à lire pour les variables, fonctions, etc. Il s'agit du ***Camel case*** : la casse du chameau ! Non, on ne « dégomme » pas un chameau, il s'agit, comme le chameau qui a plusieurs bosses, de mettre des majuscules sur les mots composant le nom d'une variable ou d'une fonction. Ainsi, cette variable que nous pourrions écrire `$nombredecoupsavantfindevie` s'écrira plutôt comme ceci `$nombreDeCoupsAvantFinDeVie`. Vous voyez la facilité obtenue... Ok, on ne va pas forcément écrire des variables à « rallonge » comme celle-ci, mais des noms de fonction, si !

La convention veut que pour les noms de variables, le premier mot soit en minuscule puis ensuite avec une majuscule à chaque mot. Il en va de même pour les fonctions et les méthodes. On écrira par exemple :

public `function getUserLogin() {` au lieu de `public function getuserlogin() {`

Concernant les classes et les interfaces ce sera le même principe à l'exception du premier mot qui lui aussi sera en majuscule :

class MaSuperClasse implements MaSuperInterface {

au lieu de

class masuperclasse implements masuperinterface {

Lorsque vous utilisez des constantes, prenez l'habitude de les écrire en majuscule avec les mots séparés par des « underscores » (le fameux « tiret du 8 ») :

define ("MA_SUPER_CONSTANTE", "42") ;

Une autre convention qu'il peut être utile de mettre en pratique, lorsque vous définissez des champs de classes qui ont une visibilité « privée », placez un « underscore » devant le nom du champs :

```
class MaSuperClasse implements MaSuperInterface {
      private $_monChampPrive; // Champ qui ne sera accessible que par cette classe
```

Dernière proposition sur ce sujet, malheureusement il est de règle de coder tout en... anglais. Deux raisons à cela :

- on ne sait jamais si votre projet ne va pas être repris par des personnes étrangères et donc tout le monde devrait pouvoir lire votre code. J'ai pourtant fait Allemand première langue, mais un code en allemand, c'est plutôt ardu à lire...
- les mots anglais n'ayant pas d'accent, le code est plus lisible qu'avec des mots français et cela supprime une bonne partie de risques de confusion ou de mauvaise interprétation. Exemple (ok, un peu capillotracté, je vous l'accorde !) :

```
function estDiffuse() { // le contenu de l'émission n'est pas encore bien établit
function estDiffusee() { // l'émission est en cours de diffusion
```

qui donne en anglais

```
function isDiffused() { // the content of the program is not yet established
function isBroadcast() { // the radio program is being broadcast
```

Pas de mauvaise interprétation possible en anglais...

Commenter et documenter son code

Deux types d'informations peuvent être adjointes à votre code : les commentaires et la documentation, deux choses bien distinctes avec chacune leur rôle.

Le commentaire :

Le commentaire est écrit dans le code et ne sera pas pris en compte au moment de l'interprétation ou bien de la compilation de votre code. Le commentaire est disposé dans les fonctions ou méthodes de classes. Il a pour objectif de faciliter la compréhension de l'algorithme mis en œuvre dans la fonction ou la méthode. Il est placé à des endroits bien spécifiques : au niveau des boucles, des embranchements (conditions, etc.), de certaines instructions qui ne seraient pas forcément évidentes à comprendre rapidement. Il ne doit pas être très long, juste au maximum 2 lignes éventuellement, la plus part du temps se sera plutôt 1 ligne.

Vous placez le commentaire où cela vous semble le plus « logique » : juste avant une boucle pour expliquer ce qu'elle va faire, ou à côté (après) d'une condition ou d'une instruction pour indiquer le cas pris en compte ou ce que fait l'instruction.

Exemples de commentaire :

```
function chargeJoursChomes($dateDeb, $dateFin) {

           $res = array(); // Crée un tableau qui sera renvoyé par la fonction comme résultat
        // charge en mémoire le contenu du fichier et le «parse» comme étant du XML
        $xmlJours = new SimpleXMLElement(file_get_contents(FICHIER_ABS_JOURS_CHOMES));

        // On ne prend en compte que les jours non travaillés qui sont entre
        // $dateDeb et $DateFin
        foreach($xmlJours->jour as $xmlJour) {

               $d = $xmlJour["date"]->__toString(); // On transforme la date qui est un objet PHP en chaîne de caractères

            if (strlen($d) <= 5)  { // Il s'agit d'une date fixe chaque année
                // elle est de la forme "MM-JJ"
                $annee = substr($dateDeb, 0, 4); // On récupère l'année en cours
                 $nd = $annee . "-" . $d; // et la date qui devrait être prise en compte serait l'année ainsi que le mois et le jour
                if ($nd < $dateDeb || $nd > $dateFin) { // cette date ne fait pas partie de la plage de jours prise en compte
                     $d = ($annee+1) . "-" . $d; // la date est alors "définie à l'année suivante"
                } else { // la date fait partie de la plage de jours
                          $d = $nd; // cette date est alors utilisée pour la suite du traitement
                }

            } // Fin du cas d'une date fixe chaque année
            // A ce niveau là, la date présente dans la variable $d est de la forme AAAA-MM-JJ
             if ($d >= $dateDeb && $d <= $dateFin) { // Si la date est bien dans la plage de jours prise en compte

                  $res[$d] = $xmlJour["libelle"]->__toString(); // On ajoute cette date dans le tableau à la clé $d et valeur le libellé du jour chomé

            }
        }

        ksort($res); // On trie le tableau en fonction de ses clés (les dates)
        return $res; // la fonction renvoie un tableau associatif
}
```

Une fois les commentaires placés, il est beaucoup plus aisé de comprendre ce que fait la fonction. Il suffit de lire les commentaires sans faire attention au code ! Dans cet exemple, il y a un peu trop de commentaires. Cette foison est plutôt à but pédagogique ! Un exemple de commentaires corrects (ni pas assez ni trop) :

```
function chargeJoursChomes($dateDeb, $dateFin) {

        $res = array();
        // parse le contenu XML du fichier de données
        $xmlJours = new SimpleXMLElement(file_get_contents(FICHIER_ABS_JOURS_CHOMES));

        // On ne prend en compte que les jours non travaillés qui sont entre
        // $dateDeb et $DateFin
        foreach($xmlJours->jour as $xmlJour) {

            $d = $xmlJour["date"]->__toString();

            if (strlen($d) <= 5)  { // Il s'agit d'une date fixe chaque année
                // elle est de la forme "MM-JJ", on la transforme au format AAAA-MM-JJ
                $annee = substr($dateDeb, 0, 4);
                $nd = $annee . "-" . $d;
                 if ($nd < $dateDeb || $nd > $dateFin) { // cette date ne fait pas partie
de la plage de jours prise en compte
                      $d = ($annee+1) . "-" . $d; // la date est alors "définie à l'année
suivante"
                } else { // la date fait partie de la plage de jours
                    $d = $nd;
                }

            }

             if ($d >= $dateDeb && $d <= $dateFin) { // Si la date est bien dans la plage
de jours prise en compte
                   $res[$d] = $xmlJour["libelle"]->__toString(); // On ajoute cette date
dans le tableau
            }
        }

        ksort($res);
        return $res;
}
```

La documentation :

Une autre façon d'informer le lecteur de votre code est de rédiger de la documentation. La documentation est située en « entête » des classes, des méthodes, des fonctions. Généralement, elle adopte une syntaxe particulière et précise pour être traitée de façon automatique. Il existe des programmes qui vont analyser vos fichiers et prendre en compte les informations que vous y avez insérées. Une série de programmes souvent utilisées ont leur nom se terminant par « doc ». Ainsi pour Java vous avez Javadoc, pour php phpDoc, etc.

Ils savent analyser une syntaxe assez commune, ce qui permet d'en apprendre qu'une au lieu d'une syntaxe par langage. En « sortie », vous avez à disposition des documents sous divers formats (HTML, PDF, etc.) qui énumèrent l'ensemble des classes de votre application, l'ensemble des méthodes, ce qu'elles font, les arguments qu'elles attendent et ce qu'elles renvoient (enfin, le programme n'invente rien, il ne met qu'en forme ce que vous avez écrit !).

Lorsque vous êtes développeur et que vous avez accès à toute cette documentation (car elle aura été réalisée dans les règles de l'art !), vous allez pouvoir utiliser sereinement toutes ces informations pour coder de façon beaucoup plus rapide et efficace. Pas besoin d'aller ouvrir tel fichier source pour savoir comment utiliser telle classe ou telle méthode, vous lisez la doc, point barre !

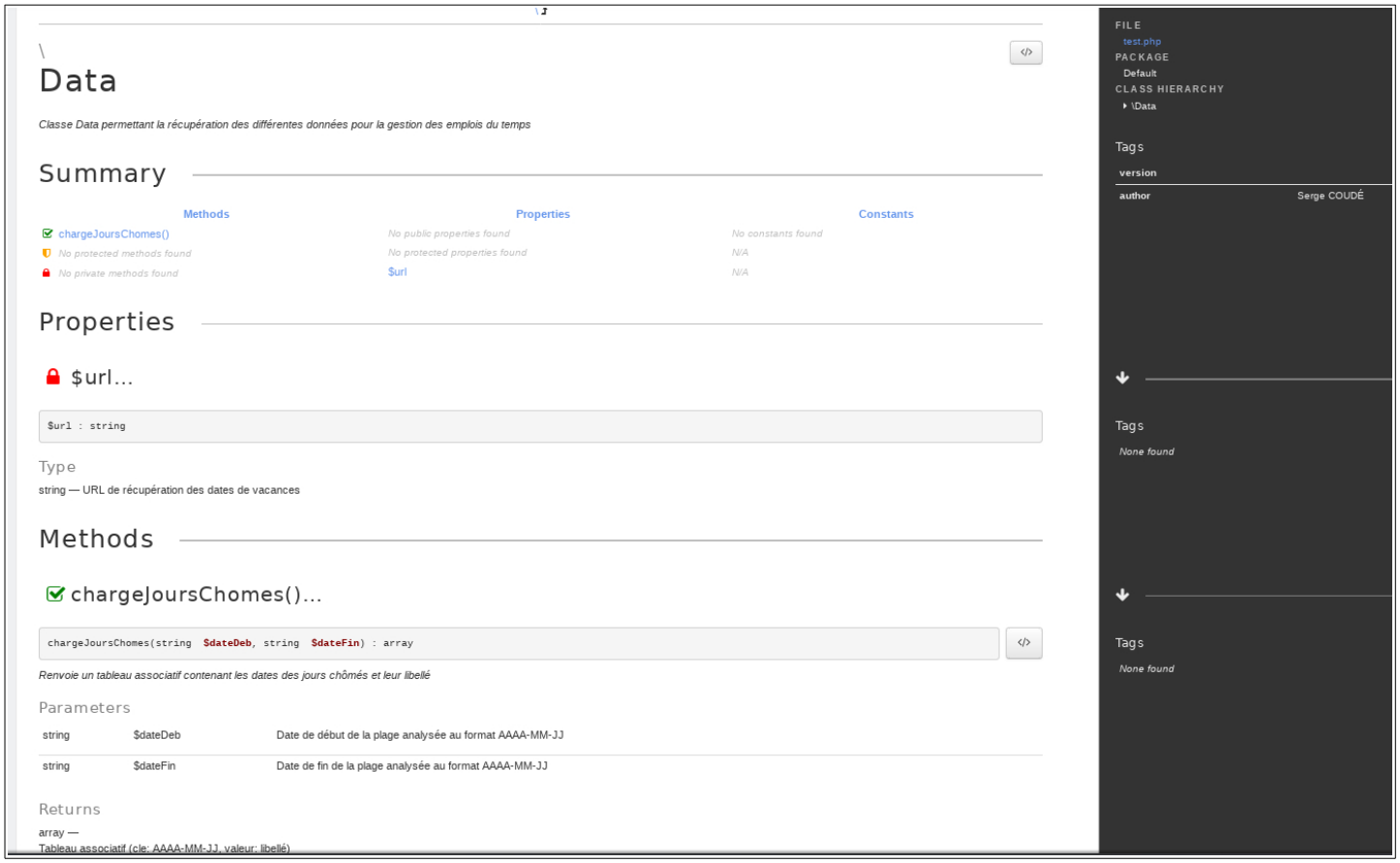
Evidemment, lorsque vous apportez des modifications à du code déjà existant, pensez obligatoirement à documenter vos modifications et re-générer la documentation pour qu'elle soit à jour !

Si vous utilisez la syntaxe de phpDoc (https://docs.phpdoc.org/references/phpdoc/index.html), par exemple, cela donnera ce style de code :

```
/**
  * Classe Data permettant la récupération des différentes données pour la
  * gestion des emplois du temps
  *
  * @version 1.0
  * @author Serge COUDÉ
  */
class Data {

      /**
       * @var string $url URL de récupération des dates de vacances
       */
      private $url;

/**
  * Renvoie un tableau associatif contenant les dates des jours chômés et leur libellé
  *
  * @param string $dateDeb Date de début de la plage analysée au format AAAA-MM-JJ
  * @param string $dateFin Date de fin de la plage analysée au format AAAA-MM-JJ
  * @return array Tableau associatif (cle: AAAA-MM-JJ, valeur: libellé)
  */
function chargeJoursChomes($dateDeb, $dateFin) {

        $res = array();
        // parse le contenu XML du fichier de données
        $xmlJours = new SimpleXMLElement(file_get_contents(FICHIER_ABS_JOURS_CHOMES));

        // On ne prend en compte que les jours non travaillés qui sont entre
        // $dateDeb et $DateFin
        foreach($xmlJours->jour as $xmlJour) {

            $d = $xmlJour["date"]->__toString();
```

```
            if (strlen($d) <= 5)  { // Il s'agit d'une date fixe chaque année
                // elle est de la forme "MM-JJ", on la transforme au format AAAA-MM-JJ
                $annee = substr($dateDeb, 0, 4);
                $nd = $annee . "-" . $d;
                 if ($nd < $dateDeb || $nd > $dateFin) { // cette date ne fait pas partie
de la plage de jours prise en compte
                      $d = ($annee+1) . "-" . $d; // la date est alors "définie à l'année
suivante"
                } else { // la date fait partie de la plage de jours
                    $d = $nd;
                }
            }

             if ($d >= $dateDeb && $d <= $dateFin) { // Si la date est bien dans la plage
de jours prise en compte
                  $res[$d] = $xmlJour["libelle"]->__toString(); // On ajoute cette date
dans le tableau
            }
        }

        ksort($res);
        return $res;
}
...
```

En installant phpDocumentor, vous allez pouvoir obtenir ce genre de sortie en une seule commande :



Donc très important : dans tous vos développements, commentez et documentez !

Structurer son code

Un code bien conçu passe également par son découpage. Au début, vous avez tendance à coder dans un seul et même fichier. Rapidement, il va devenir complexe et fastidieux à maintenir, « scroller » de haut en bas pour aller modifier une méthode ou un champ va vous agacer ! Il est préférable de découper votre code en différents fichiers et aller également créer une arborescence de fichiers.

Votre arborescence va s'adapter au découpage de votre application que vous aurez réalisé. Exemple, si vous avez une partie Administration, le code lié à cette fonctionnalité sera plutôt mis dans un répertoire *admin*. Dans le cas du stockage, vous allez mettre le code affilié dans un répertoire *bdd*, etc. Mais si vous utilisez le paradigme MVC, vous allez découper votre code avec un répertoire *modèle*, un répertoire *contrôleur* et un répertoire *vue*. Si votre partie « admin » est elle-même en MVC, alors ces 3 répertoires peuvent se trouver dans le répertoire « parent » *admin*... Si vous avez des configurations, alors vous pouvez créer un répertoire *config*. Si vous avez un « paquet » de fonctions utiles, vous pouvez créer un répertoire *lib* (comme « librairies »), etc.

Une fois votre arborescence posée, vous allez créer un fichier par classe si vous faîtes de la POO. Si vous programmez en façon « procédurale » ou « fonctionnelle », vous regrouperez vos fonctions dans des fichiers en fonction de leur « thématique » : toutes les fonctions travaillant avec la base de donnée seront placées dans un fichier *bdd.inc.php* par exemple (pourquoi le « inc » ? Il s'agit juste d'un moyen mémo-technique de se rappeler que ce fichier ne sera pas appelé directement par l'utilisateur mais sera inclus ou utilisé par d'autres scripts).

En utilisant cette façon de découper et structurer votre code, vous aurez plus de facilité à vous « promener » dans votre code pour y faire des ajouts ou de modifications...

Exemple d'arborescence :

```
admin
......view
```

```
            auth.html
......controler
            auth.class.php
......model
            user.class.php
app
......view
            article.html
            footer.html
            main.html
      ...
```

Utiliser des instructions pérennes dans le temps

Si votre code a pour vocation de durer dans le temps (plusieurs années !), faites en sorte d'utiliser des instructions qui seront encore utilisables dans quelques années. On ne peut prédire de quoi sera fait demain, mais si en utilisant des instructions vous voyez dans leur documentation qu'elles sont *deprecated* ou en passe de l'être dans les prochaines versions du langage que vous utilisez, évitez au maximum de les utiliser !

De même, lorsque vous utilisez des librairies externes à votre programme, vérifiez qu'elles sont mises à jour régulièrement et qu'une bonne équipe de développeurs l'accompagne : une librairie qui n'est pas mis à jour depuis 2 ans risque dans les années à venir de ne plus être utilisable : version de langage obligatoire qui ne sera plus compatible avec la version en cours actuellement, dépendances à d'autres librairies qui seront obsolètes, etc. C'est d'autant plus flagrant avec les librairies de cryptographie qui évoluent tous les semestres !

Améliorer la qualité de son code

Utiliser le debugger

Lorsque vous codez, vous vous retrouvez rapidement le « nez dans le guidon » et il devient rapidement facile de se perdre dans son code, surtout au début de son apprentissage. Prenez régulièrement du recul (passez à autre chose, changez vous les idées) afin d'avoir un regard neuf lorsque vous vous y replongez. Si votre code est complexe (ou tout du moins l'est pour vous ;-)) vous allez souvent essayer de le faire « tourner » instruction par instruction pour vérifier si le fonctionnement passe bien par cette partie de votre code, si la condition est bien vérifiée, etc. Il y a plusieurs méthodes pour cela :

- la plus naturelle (pour un débutant !) : le « print « coucou » avant chaque instruction que l'on souhaite vérifier
- la plus professionnelle (évidemment à appliquer !) : l'utilisation d'un outil de debuggage

La première vous permettra de vous familiariser avec l'enchaînement des instructions de votre code, mais elle devra être rapidement remplacée par la seconde !

Dans les temps anciens (fin du 20ème siècle ;-)), les outils de debuggage n'étaient pas aussi pratiques qu'aujourd'hui... De nos jours, les IDE utilisés pour développer embarquent tous au moins un outil pour réaliser cette tâche. Eclipse, Netbeans, etc. permettent de faire tourner votre application instruction par instruction, de pouvoir consulter le contenu de vos variables et autres données renvoyées par vos méthodes / fonctions, etc., de vous arrêter à un endroit précis et éventuellement changer la valeur d'une variable « à chaud ». Même pour des langages qui ne sont pas normalement exécutés par votre IDE comme le PHP, il est possible via des outils à installer sur votre serveur de debugger votre application (exemple xDebug avec PHP à installer avec votre serveur Apache).

L'utilisation d'un debugger vous permettra d'optimiser le temps de votre développement en vous évitant de chercher un peu partout où cela peut « coincer » !

Les tests unitaires

Une source d'erreur fréquente est la mise à jour de son code. Vous pensez mettre à jour un algorithme, une fonction, une classe pour l'optimiser. Vous effectuez un test ou deux pour vérifier que votre modification fonctionne et vous passez à une autre partie de votre code. Lorsque vous refaites tourner votre application, vous vous apercevez qu'elle ne réagit pas comme vous l'attendez et vous êtes obligé de vous replonger dans votre code pour savoir d'où provient le dysfonctionnement.

Il est intéressant de mettre en place, au moment où vous développez la première version de votre code, des tests unitaires. Il s'agit de fonctions ou bien d'instructions supplémentaires qui vont être exécutées avec un outil spécifique. Lors de la compilation pour produire le code binaire final, elles ne seront évidemment pas prises en compte.

Ces instructions sont là pour vérifier si une méthode ou une fonction renvoie bien des résultats corrects. Vous définissez un « jeu » de paramètres à passer en tant qu'arguments à la méthode/fonction (si la méthode/fonction en a besoin !) et le résultat qu'elle doit retourner (si elle doit retourner quelque chose !). Ce jeu de paramètres / résultat sera « rejoué » à chaque fois que vous lancerez l'exécution de vos tests unitaires.

Ainsi, vous aurez, à chaque modification de code ou au moins de façon régulière, « l'obligation » de lancer vos tests unitaires pour vérifier qu'il n'y ait pas de « ***régression*** » (c'est à dire que votre code qui fonctionnait avant votre modification ne fonctionne plus normalement et provoque une erreur).

Exemple de test unitaire sur une fonction très simple, la fonction factorielle :

Nous avons installé préalablement l'application phpUnit (https://phpunit.de/), moteur de tests unitaires libre et gratuit.

Notre code initial (qui sera livré) est composé de 2 fichiers, un fichier lib.inc.php contenant notre fonction factorielle et un fichier factorielle.php qui est à exécuter pour avoir le résultat de la factorielle du nombre passé en paramètre :

fichier lib.inc.php :

```php
<?php

    /**
     * Calcule la factorielle d'un nombre entier
     *
     * @param int $n Nombre entier dont on veut calculer la factorielle
     * @return int factorielle du nombre entier passé en argument
    */
    function factorielle($n) {
        if ($n <= 1) {
            return 1;
        } else {
            return $n * factorielle($n-1);
        }
    }
```

fichier factorielle.php :

```php
<?php
```

```php
/**
 * Programme calculant la factorielle de l'argument passé en paramètre du script
 *
 */

require(__DIR__ . "/lib.inc.php");

/**
 * Affiche la façon d'utiliser le script
 *
 * @param array $argv Tableau contenant le nom du script et les paramètres fournis
 */
function usage($argv) {
    die("Usage : php " . $argv[0] . " <nombre entier>\n");
}

if ($argc != 2) { // Si il n'y a pas un et un seul argument, on affiche l'usage
    usage($argv);
}

$nb = $argv[1];

echo "factorielle(" . $nb . ") => " . factorielle($nb) . "\n";
```

Nous créons un troisième fichier (afin de ne pas interférer avec notre code déjà existant et qui bien évidemment ne sera pas livré au client, sauf cas exceptionnel) qui va contenir les tests à effectuer : test.php

```php
<?php

use PHPUnit\Framework\TestCase;

require(__DIR__ . "/lib.inc.php");

class FactorielleTest extends TestCase {

    public function testFactorielle() {

        $this->assertSame(120, factorielle(5));
        $this->assertSame(3628800, factorielle(10));
        $this->assertSame(1, factorielle(0));
    }

}
```

On peut constater qu'il n'y a ni commentaire ni documentation. Dans l'absolu cela devrait être fait, mais si déjà vous le faite pour votre code source livré, cela sera très suffisant !

Nous pouvons observer dans ce fichier test.php la définition d'une méthode qui effectue les appels à la fonction factorielle avec des paramètres définis et qui vérifie que le résultat obtenu est bien celui attendu (exemple, la fonction factorielle doit retourner la valeur 120 si on lui passe la valeur 5 en paramètre).

Si on lance phpunit sur ce fichier, cela donne :

```
$ phpunit test.php
PHPUnit 6.5.5 by Sebastian Bergmann and contributors.
.                                                                   1 / 1 (100%)
Time: 120 ms, Memory: 4.00MB
OK (1 test, 3 assertions)
```

Nous constatons qu'un test a été effectué (appel de la méthode) et 3 assertions ont été vérifiées (les 3 appels à la méthode assertSame de la classe FactorielleTest héritant de la classe TestCase. Si une assertion n'est pas « remplie », alors un message d'erreur sera affiché, vous permettant de savoir exactement où aller modifier votre code.

Je ne peux que vous conseiller le cours d'OpenClassrooms sur les tests unitaires avec PHP : https://openclassrooms.com/fr/courses/4087056-testez-et-suivez-letat-de-votre-application-php/4419446-premiers-pas-avec-phpunit-et-les-tests-unitaires

De « l'invention » du principe des tests unitaires est apparue une méthode de programmation appelée « ***développement piloté par les tests*** ». Il s'agit tout simplement de commencer avant tout développement par définir l'ensemble des tests que doit passer votre code et petit à petit, au fur et à mesure de « corriger » votre code, c'est à dire dans un premier temps rédiger les instructions puis ensuite les optimiser. Si vous avez défini l'ensemble correcte des tests, vous n'avez plus qu'à « boucher les trous » ! C'est un peu comme lorsque vous rédigez un document, normalement vous faites en premier votre plan, puis ensuite, partie par partie vous rédigez...

l'intégration continue

L'ensemble des différentes méthodes présentées précédemment doit vous permettre d'avoir un code le plus propre, efficace, maintenable possible. Il faut les avoir à l'esprit à chaque seconde, mais cela peut être un peu prise de tête à la longue. Heureusement, des développeurs ont réalisé des applications visant à vous aider à ne pas devoir vous rappeler de tout ! L'utilisation de ces outils vous permet de mettre en œuvre le principe d'***intégration continue*** avec donc des « **serveurs d'intégration** ».

Imaginez un tapis roulant où vous seriez situé à une des extrémités. Vous déposez votre code dessus et il part sur le tapis. Au bout du tapis, vous avez votre client qui récupère votre code, mais pendant le trajet sur le tapis, votre code a été trituré, vérifié, etc. Et si des problèmes ont été constatés, vous avez un bras articulé qui vous ramène votre code pour que vous le corrigiez !

Ces outils vont faire passer votre code à la « moulinette » de différentes vérifications avec entre autres celles de l'application des bonnes pratiques mentionnées dans les paragraphes précédents ! Vous pourrez avoir des vérification sur :

- le bon nommage de vos variables, classes ou méthodes
- la bonne indentation de votre code
- la bonne aération et la faible complexité (pas trop d'instructions dans une méthode, utilisation des conditions à bon escient, etc.)

- la présence de commentaires et documentations
- l'absence de méthodes ou fonctions qui ne doivent plus être utilisées
- etc.

Ces outils pourront exécuter également de façon automatique les tests unitaires que vous aurez défini.

Tout cet enchaînement pourra être réalisé à chaque fois que vous « commiterez » sur votre dépôt de version...

Ainsi, si des erreurs ou avertissements sont remontés, vous en êtes informé immédiatement... Cela vous permet de vous « dégager » la tête de toutes les vérifications à réaliser. Mais que cela ne vous empêche pas d'essayer de faire du code le plus propre du premier coup, ce sera toujours plus efficace que les aller-retours...

Exemple d'outils utilisés pour mettre en œuvre ce principe : Jenkins (ex Hudson), CruiseControl, Travis ou TreeHerder, des outils libres et gratuits...

Exploitabilité de l'application

On appelle exploitabilité d'une application sa propension à pouvoir être paramétrée et exploitée sans connaître le code qui la compose.

Un exploitant est un informaticien qui n'a pas forcément de compétences en développement et surtout pas obligatoirement dans le langage utilisé. Son rôle est de maintenir l'application en condition de fonctionnement optimal (MCO : Maintient en Condition Opérationnelle).

Pour cela, il a comme « outils » :

- la modification de paramètres
- la supervision des logs

Modification de paramètres

Si pour modifier un paramètre il est obligé de recompiler l'application ou modifier un code source d'une application interprétée, il peut potentiellement sans le vouloir insérer un bug dans l'application. Il faut donc « externaliser » tous les paramétrages susceptibles d'être modifiés, dans des fichiers (texte si possible) ou bien dans des champs de tables de bases de données (même si les fichiers sont plus facilement utilisables et à privilégier !).

Ainsi, il n'accède qu'à une partie précise de l'application et votre application peut éventuellement effectuer des tests de bon paramétrage à son lancement avant de prendre en compte la nouvelle configuration.

Exemples de fichier de configuration :

dans un fichier texte (le mieux, à privilégier, indépendant du langage utilisé !)

```
appname=ModuleTDT
appversion=1.2
appauthor=Serge COUDÉ
dbhost=127.0.0.1
dbname=moduletdt
dbuser=expmoduletdt
dbpassword=tototiti
```

dans un fichier dans le langage utilisé (moins bien car dépendant du langage, mais souvent utilisé... Ici en PHP)

```
<?php
```

```
/**
 * Fichier de configuration de l'application moduleTDT
 *
 */
$appName = "ModuleTDT";
$appVersion = "1.2";
$appAuthor = "Serge COUDÉ";
$dbHost = "127.0.0.1";
$dbName = "moduletdt";
$dbUser = "expmoduletdt";
$dbPassword = "tototiti";
```

Supervision des logs

Un exploitant a besoin d'avoir des informations sur le bon fonctionnement de votre application. Elle peut tourner 24/24 ou bien juste être lancée de temps en temps. Si votre application rencontre des problèmes, elle doit pouvoir « communiquer » ces problèmes. Plusieurs techniques peuvent être utilisées :

- des informations insérées au fur et à mesure dans un fichier (les logs)
- des informations envoyées à un système local de gestion des logs (syslog par exemple)
- des informations envoyées à un webservice ou un système d'agrégation distant
- des informations mises à disposition suivant un protocole précis (SNMP)
- des informations affichées au fur et à mesure dans un terminal/console (peu souvent le cas !)
- etc.

Pour faciliter la tâche du développeur qui aurait tendance à penser que son code est fiable et que cela prend de toute manière trop de temps de générer les logs, des équipes ont développé des librairies « prêtes à l'emploi » pour que le développeur n'ait plus qu'à se concentrer sur son code. Comme pour la documentation, des librairies « standards » se sont développées pour plusieurs langages différents : les log4xxxx !

Ainsi pour Java, vous pouvez utiliser log4j, pour PHP log4php, etc. Ces librairies fonctionnent sur le même principe et il vous est donc aisé, une fois maîtrisée leur utilisation, de les utiliser toutes en fonction de votre langage.

Généralement, vous allez devoir définir des « niveaux » de logs, dont voici les plus courants du plus verbeux au plus critique :

- debug
- trace
- info
- warning
- errror
- critical

Lorsque vous codez, il faut toujours avoir à l'idée que l'instruction que vous codez ou encore la méthode peut « se planter ». Il faut donc prévoir le cas ou les cas d'erreurs et générer les bonnes informations.

Vous n'êtes pas obligé de prendre en compte tous ces niveaux, mais au minimum warning et error doivent être implémentés !

Comment choisir le niveau d'un log ? C'est selon votre application ! Un message indiquant que votre application s'est bien lancée sera plutôt de niveau *info*. La connexion d'un utilisateur à votre application pourra être au niveau *trace*. Un mauvais mot de passe saisie pourra soit être au niveau *trace*, soit *warning* suivant vos besoins. L'inaccessibilité d'une ressource pas très importante pourra être de niveau *warning*, alors que la connexion impossible à une base de données sera de niveau *error* voire *critical* si toute votre application en dépend ! On utilise le niveau *critical* lorsque votre application ne peut plus fonctionner ou bien en mode tellement dégradé qu'il faut une intervention urgente de l'exploitant.

Pourquoi utiliser ces niveaux ? Plus on « descend » vers le niveau *debug*, plus il y a une quantité importante d'informations qui sont données, et plus il est compliqué à l'exploitant de trier le grain de l'ivraie. L'exploitant aura à sa disposition un outil d'analyse de log qui lui permettra de définir ses niveaux d'alertes ou d'affichage d'informations (filtrage). Il pourra alors se concentrer sur les informations les plus pertinentes à ses yeux. En fonction de la période, ses niveaux d'alertes pourront être modifiés sans devoir toucher au code de votre application. Par exemple, si il constate une tentative d'intrusion dans l'application, il pourra demander pendant quelques heures à avoir également les informations du niveau *trace* pour savoir qui tente de se connecter.

Exemple d'utilisation de log4php :

```
<?php

require("log4php/Logger.php");
define("FICHIER_ABS_JOURS_CHOMES", "jourschomes.xml");

/**
  * Classe Data permettant la récupération des différentes données pour la
  * gestion des emplois du temps
  *
  * @version 1.0
  * @author Serge COUDÉ
  */
class Data {

      /**
       * @var string $url URL de récupération des dates de vacances
       */
      private $url;

/**
  * Renvoie un tableau associatif contenant les dates des jours chômés et leur libellé
  *
  * @param string $dateDeb Date de début de la plage analysée au format AAAA-MM-JJ
  * @param string $dateFin Date de fin de la plage analysée au format AAAA-MM-JJ
  * @return array Tableau associatif (cle: AAAA-MM-JJ, valeur: libellé)
  */
function chargeJoursChomes($dateDeb, $dateFin) {

        $logger = Logger::getLogger("main");
```

```
        $logger->debug("chargeJoursChomes() : debut");
        $res = array();
        if (!file_exists(FICHIER_ABS_JOURS_CHOMES)) {
            $logger->error(FICHIER_ABS_JOURS_CHOMES . " n'est pas accessible ou pas
présent");
        } else {
            try {
                // parse le contenu XML du fichier de données
                $xmlJours = new
SimpleXMLElement(file_get_contents(FICHIER_ABS_JOURS_CHOMES));

                // On ne prend en compte que les jours non travaillés qui sont entre
                // $dateDeb et $DateFin
                foreach($xmlJours->jour as $xmlJour) {

                    $d = $xmlJour["date"]->__toString();

                    if (strlen($d) <= 5)  { // Il s'agit d'une date fixe chaque année
                            // elle est de la forme "MM-JJ", on la transforme au format
AAAA-MM-JJ
                        $annee = substr($dateDeb, 0, 4);
                        $nd = $annee . "-" . $d;
                        if ($nd < $dateDeb || $nd > $dateFin) { // cette date ne fait pas
partie de la plage de jours prise en compte
                            $d = ($annee+1) . "-" . $d; // la date est alors "définie à
l'année suivante"
                        } else { // la date fait partie de la plage de jours
                            $d = $nd;
                            }
                    }

                    if ($d >= $dateDeb && $d <= $dateFin) { // Si la date est bien dans
la plage de jours prise en compte
                        $res[$d] = $xmlJour["libelle"]->__toString(); // On ajoute cette
date dans le tableau
                        }
                }
            } catch(Exception $e) {
                $logger->error($e);
            }
        }

        ksort($res);
        $logger->debug("chargeJoursChomes() : fin : (" . count($res) . " jours
chargés)");
        return $res;
}

}
```

```
$data = new Data();
$data->chargeJoursChomes("2018-09-03", "2019-09-02");
```

On peut constater la simplicité d'utilisation : on instancie un singleton de la classe Logger et on utilise ses méthodes en fonction du niveau de l'information que l'on souhaite faire « remonter ».

Sortie lorsque le fichier xml est présent :

```
DEBUG - chargeJoursChomes() : debut
DEBUG - chargeJoursChomes() : fin : (11 jours chargés)
```

Sortie lorsque le fichier xml n'est pas correctement formaté :

```
DEBUG - chargeJoursChomes() : debut
ERROR - Exception: String could not be parsed as XML in
/home/serge/Documents/openclassrooms/cours/tests/log4php/test.php:37
Stack trace:
#0 /home/serge/Documents/openclassrooms/cours/tests/log4php/test.php(37):
SimpleXMLElement->__construct('<?xml version="...')
#1 /home/serge/Documents/openclassrooms/cours/tests/log4php/test.php(73): Data-
>chargeJoursChomes('2018-09-03', '2019-09-02')
#2 {main}
DEBUG - chargeJoursChomes() : fin : (0 jours chargés)
```

Sortie lorsque le fichier xml est absent :

```
DEBUG - chargeJoursChomes() : debut
ERROR - jourschomes.xml n'est pas accessible ou pas présent
DEBUG - chargeJoursChomes() : fin : (0 jours chargés)
```

Dans cet exemple, les logs sont toujours affichés, quelque soit leur niveau. Pour modifier les niveaux pris en compte et paramétrer la gestion des logs, vous pouvez définir des options que vous placez dans un fichier de configuration au format texte que vous nommerez ***maconfig.ini*** par exemple :

```
log4php.appender.default = LoggerAppenderEcho
log4php.appender.default.layout = LoggerLayoutSimple
log4php.rootLogger = WARN, default
```

Cette config indique que seules les infos ayant au minimum le niveau *warning* seront prises en compte et cela se ferra sous forme d'affichage avec un modèle « simple ».

Vous prendrez en compte cette configuration dans votre script avec l'instruction php :

```
require("log4php/Logger.php");
Logger::configure('maconfig.ini');
define("FICHIER_ABS_JOURS_CHOMES", "jourschomes.xml");
```

Cela donnera dans le cas d'un fichier xml manquant la sortie suivante :

```
ERROR - jourschomes.xml n'est pas accessible ou pas présent
```

On voit bien que les traces du niveau debug qui avant étaient présentes ont maintenant disparu.

Les sorties peuvent se faire vers la console, des fichiers, des envois de mail, etc. Il suffit de définir les bonnes options de log4php...

Des configurations bien plus complexes peuvent être mises en place, à vous de découvrir cela en lisant les nombreux tutoriels qui sont présents sur Internet !

Scalabilité de l'application

Je regroupe deux définitions concernant la scalabilité :

- On appelle scalabilité la possibilité de déployer une application sur plusieurs serveurs en même temps et que toutes les instances soient vue comme une seule et même application (on parlera alors de cluster).
- On peut également appréhender la scalabilité sous le fait que des parties de votre application puissent être déployées sur différents serveurs.

Scalabilité = clusterisation

Imaginons que notre application de messagerie instantanée connaisse un grand succès au point que notre pauvre serveur l'hébergeant soit à deux doigts de rendre l'âme... Il va falloir penser à répartir notre application sur plusieurs serveurs et que les utilisateurs utilisent soit le premier, soit le deuxième, troisième (etc.) serveur, sans se douter qu'ils n'accèdent pas tout le temps au même serveur.

Au niveau architecture système et réseau, il n'y a pas de problème : on installe 2, 3, 4 serveurs si besoin et en amont, on place un répartiteur de charge (loadbalancer). Ainsi le loadbalancer envoi les demandes des utilisateurs au premier serveur, au deuxième, au troisième, etc. (cas de l'algorythme roundrobin) ou bien il demande quel est le serveur le moins utilisé et lui envoi la demande...

Mais au niveau de notre application, il va y avoir un problème : comment un utilisateur va pouvoir toujours interroger le même serveur pour ne pas perdre « le fil » ?

Il y a plusieurs solutions techniques, mais généralement on va mettre en œuvre des variables de session. Il faut au moment du développement de l'application penser à ajouter une information permettant de « distinguer » le serveur répondant à la demande d'un client. Cette information peut être une variable de session hébergée sur le serveur, spécifique à l'utilisateur et au serveur, qui sera aussi transmise avec chaque URL demandée par l'utilisateur. Ainsi le répartiteur de charge, bien configuré, analysera les URL, trouvera la bonne variable passée et aiguillera vers le bon serveur.

Quelque fois, le répartiteur de charge fera lui même le travail en analysant les flux entrant et sortant et en les modifiant « à la volée », mais pas toujours...

Scalabilité = architecture n-tiers

Il se peut (et c'est souvent le cas !) que votre application ne soit pas déployée chez le client comme dans votre « labo » de développement (un pc et c'est tout !).

Dans les grandes entreprises, vous avez des « fermes » de serveurs (plusieurs serveurs regroupés par fonctionnalité, par département, etc.). Ainsi il y aura certainement déjà des serveurs de bases de données installés. Et ne comptez pas sur les exploitants pour vous dire : « ok, on va rajouter un nouveau serveur de base de données sur le serveur qui va héberger votre application » ; ils ont déjà suffisamment de travail de superviser ceux en place ! Ils « intégreront » votre base de données dans un de leurs serveur déjà existant. Et votre partie applicatif sera, elle, sur un autre serveur, voire plusieurs autres (cf chapitre précédent)...

Il faut donc que la partie applicative soit en mesure de communiquer avec votre partie stockage hébergée sur un autre serveur. On oublie donc les connexions à la base de données via un descripteur de fichier (certes plus rapide mais pas adaptable à toutes les situations), **on utilisera obligatoirement une connexion tcp/ip** ! … Avec les paramétrages adéquats bien définis dans un fichier externalisé pour une modification aisée par les exploitants (cf chapitre précédent « Exploitabilité de l'application »...).

Vous pouvez même imaginer que votre base de données sera présente sur plusieurs serveurs de bases de données, accessibles via un répartiteur de charge spécial « base de données » (on appellera ça souvent un « proxy bdd »...). L'objectif de ce type d'architecture est la robustesse : si un serveur de bases de données tombe en panne, un autre prend le relai de façon transparente pour l'utilisateur ou bien si le premier serveur est trop sollicité, un second prend le relai.

Une autre partie de l'application peut être également déployée sur des serveurs distincts : le front-end. Et le front-end ne sera en général pas accessible directement depuis Internet (entre autre pour des raisons de sécurité !). L'utilisateur passera d'abord souvent par un pare-feu, puis par un serveur mandataire inversé (plus connu sous le nom reverse-proxy!) et ce dernier, enfin, attendra le front-end. Le reverse-proxy va « masquer » à l'utilisateur l'accès direct au front-end, parfois en modifiant les urls de votre application. **Il est donc très important que les pages générées par votre application ne proposent que des urls relatives**, c'est à dire de la forme *css/styles.css* ou *app/ctrl/getuser.php?key=value*, et pas d'url absolue de la forme */monapp/css/styles.css* ou *http://localhost/app/ctrl/getuser.php?key=value*.

Vous ne savez jamais à l'avance dans quelles conditions votre application va être déployée et surtout si ces conditions ne vont pas changer dans le temps.

Pour vérifier cela, faite le test de vous créer dans votre environnement de développement un autre site web (dans le cas d'une application web, évidemment !) en utilisant par exemple des hôtes virtuels sous Apache avec un nom et un chemin différent et vérifiez comment votre application se comporte. Si elle fonctionne de la même manière, alors il y a des chances qu'elle puisse s'adapter facilement. Une des erreurs qui pourra arriver est la gestion des cookies et des domaines auxquels ils se rapportent. Attention donc !

Exemple :

Application en développement : http://localhost/

Application accessible à partir d'un autre « site » : http://chezmoi.fr/monapp/

Ces deux sites utiliseront le même code source, seul les hôtes virtuels permettront le « routage ». A vous de modifier votre fichier *hosts* pour faire « pointer » chezmoi.fr vers le 127.0.0.1 !

Conclusion

En mettant en pratique l'ensemble de tout ce qui a été décrit dans ce document, vous devriez être en mesure d'intégrer une équipe projet sans trop de difficulté et surtout en ne découvrant pas pour la première fois son mode de fonctionnement.

Attention, toutes les équipes ne mettent pas forcément en pratique toutes ces bonnes façons de faire, et vous pourrez alors « tenter » (et je pèse ce verbe ;-)) de leur en faire adopter quelques unes... Mais attention, prenez des pincettes car le sujet est souvent sensible : vous le débutant n'êtes pas spécialement vu comme un expert d'entrée de jeu, et ces méthodes augmentent un peu le temps du projet et donc de facto le coût direct. Les personnes peuvent avoir du mal à appréhender le gain à plus long terme (moins d'aller-retours au moment de la recette client, plus grande rapidité lors d'ajouts de fonctionnalités ultérieurement, etc.) Mais, ça, c'est « après »...

Vous découvrirez peut-être aussi de nouvelles bonnes façons de procéder, donc soyez ouvert à tout, tout en gardant un regard critique (dans le bon sens) !

Bon développement !!!