Petit précis de développement informatique pour géomaticiens néophytes

Nicolas Py ◘ Forum SIG

Lors de la conférence francophone ESRI SIG2010, le ForumSIG[1] était présent pour deux interventions afin d'illustrer comment professionnaliser sa gestion d'un projet à caractère informatique[2] ainsi que ses développements[3]. Cet article vise principalement à synthétiser de manière plus littéraire les pratiques évoquées des ces présentations, en insistant sur plusieurs concepts qui peuvent être immédiatement employés avec plus-value par tout géomaticien, que ce soit pour le codage d'applications ou de scripts.

Le géomaticien est-il un informaticien ?

Dans un nombre non négligeable de petites structures, comme en attestent les offres d'emploi publiées sur le forum Géorezo[4], les compétences en géomatique doivent être complétées par celles d'informaticien, au sens commun du terme. De même, la démarche métier[5] initiée par l'association Georezo offre un aperçu du métier de géomaticien qui corrobore en partie ce constat. Pourtant, les différents sondages, études et analyses de ladite démarche métier montrent principalement une très large représentation des formations *« Géographie – Environnement »*, une grande spécialisation et une haute technicité du métier (plus de 65 % de la population ciblée a un diplôme bac+5), ainsi que le caractère pluridisciplinaire de la profession.

En parallèle, on assiste à un besoin croissant de compétences en développement. Ce besoin peut s'expliquer par plusieurs facteurs concomitants. Le premier facteur est l'évolution des outils : l'essor de l'*Open Source* a fait naitre de nouveaux logiciels, basés sur les langages de programmation actuels, souvent orientés objet[6]. Simultanément, les éditeurs de logiciels propriétaires permettent de plus en plus la personnalisation des outils en se détournant des langages propriétaires pour se rapprocher des langages de haut niveau, comme *Python*. Un deuxième facteur est l'évolution des données, augmentées en précision, résolution et quantité. Il en ressort un besoin légitime d'automatisation de leur manipulation et de leur analyse.

Si l'on s'accorde sur ce bilan, il devient alors opportun de se demander si le géomaticien est suffisamment armé pour répondre à ces attentes. Cet article se donne ainsi pour objectif de pointer du doigt quelques termes et concepts de l'industrie du développement informatique. Il ne s'agit pas de tout expliquer mais plutôt de débroussailler le champ sémantique du génie logiciel, et de permettre ainsi au géomaticien de dépasser ses éventuelles appréhensions face à un corpus nouveau. *In fine*, cet article tente également de combler au moins partiellement un manque de littérature sur le sujet, trop rarement présente voire absente des cours disponible sur internet ou des ouvrages dédiés à la programmation.

1. http://www.forumsig.org/.
2. http://www.portailsig.org/content/bien-debuter-son-developpement-pour-arcgis-partie-1-professionnaliser-sa-gestion-de-projet ou http://www.esrifrance.fr/sig2010/biendebuter_devlpt_ArcGIS_partie1.asp.
3. http://www.portailsig.org/content/bien-debuter-son-developpement-pour-arcgis-partie-2-professionnaliser-son-code ou http://www.esrifrance.fr/sig2010/Dvlpt_ArcGIS_Code_partie2.asp
4. http://georezo.net/forum/index.php
5. http://georezo.net/wiki/_media/main:formetiers:poster_metier_mathian_2003_2.pdf
6. http://fr.wikipedia.org/wiki/Programmation_orient %C3 %A9e_objet

Le génie logiciel

La page *Wikipedia* du génie logiciel[7], bien que fournissant une définition relativement claire des procédures employées en développement informatique, est trop proche d'une collection de listes pour être utile au géomaticien novice en programmation. Afin de ne pas se décourager trop rapidement face à ces techniques qui peuvent se percevoir comme complexes, rigides ou inutiles, il convient d'en énumérer leurs atouts.

D'une manière générale, leur premier intérêt est de garder une trace des besoins, problèmes rencontrés, et réponses apportées (historique). On note ensuite les possibilités offertes en termes de maintenance des logiciels (mise à jour, correction d'erreurs) mais également de partage de code. Les travaux complexes, à plusieurs développeurs, s'étalant sur temps importants sont également facilités. Plus simplement, l'emploi des pratiques du génie logiciel augmente les chances que le projet aboutisse et en améliore la qualité.

Quelle que soit la méthode employée (en cascade, *Quick and Dirty, Agile, Scrum, Extrem Programming, La Rache*…) et en simplifiant, on peut distinguer trois étapes successives. La première est la phase d'analyse : elle permet l'analyse des besoins, la recherche et la comparaison des technologies utilisables et la définition de l'architecture générale de l'application ; la seconde est la phase de programmation, le cœur du développement ; la dernière est la phase de livraison : on vérifie que le logiciel fonctionne ailleurs que sur l'ordinateur du développeur (et donc de préférence sur celui de l'utilisateur), et le client valide en fonction de ses attentes.

La suite du document reprend et précise chacune de ces trois phases.

La phase d'analyse

Lors de cette étape, il s'agit de prendre le temps de réfléchir, d'ébaucher la solution, de la confronter avec d'autres, avant de débuter le développement. Elle ne nécessite pas forcément d'investir financièrement dans un logiciel couteux. De simples éditeurs de texte et de diagrammes suffisent dans la majorité des cas.

On y trouve d'abord, *a minima*, les spécifications : description formelle et exhaustive du produit à réaliser. En parallèle, des cahiers de tests peuvent être rédigés. Ces documents présentent des cas d'utilisation, c'est-à-dire les manipulations utilisateurs, avec les résultats attendus.

S'y ajoute également l'architecture informatique, qui décrit la structuration d'un système informatique en termes de composants et d'organisation de ses fonctions. Celle-ci peut se décliner physiquement (machines utilisateur, serveur physique de base de données…) mais également logiquement, puisqu'un même serveur physique peut héberger plusieurs serveurs logiques (serveur logique de base de données, serveur logique web…). Une autre analyse de l'architecture pourrait employer une segmentation par composants : logiciels, les composants matériels, réseaux, etc. Ces deux approches ne sont cependant pas incompatibles !

Enfin, plus technique, la conception. Celle-ci permet de décrire de manière non ambiguë, le plus souvent en utilisant un langage de modélisation (UML, *Merise*…), le fonctionnement futur du système, afin d'en faciliter la réalisation. On y trouve les diagrammes de classes, de cas d'utilisation, de séquence, etc. Spécifier est le socle de base de la phase d'analyse, incontournable. Cela permet de savoir où et pourquoi l'on développe, afin de vérifier la pertinence de la création d'un outil et de la faire correspondre aux besoins.

La phase de développement

Depuis que le développement logiciel existe en tant que profession et secteur marchand, de nombreuses méthodes ont été mises au point. À l'heure actuelle, on peut même y distinguer des familles, dont les objectifs semblent parfois apparemment contradictoires (les extrêmes pouvant être la recherche d'absence de *bug*, en aviation par exemple ; contre une solution rapide et fonctionnelle pour un utilisateur unique dans un besoin ponctuel).

Plutôt que d'en lister les plus connues et de tenter une synthèse avec avantages et inconvénients, on retiendra que pour la majorité des géomaticiens (dans le cas de petites structures, projets impliquant un faible nombre de développeurs), un développement selon son bon sens suffira.

Néanmoins, l'emploi des principes de base du développement et de certains outils *« professionnels »* peut très rapidement et pour un faible investissement se révéler très rentable, sur de nombreux plans (chances d'aboutir du projet, qualité du code, confort du développeur, coût de développement…).

Phase de développement : des principes

De manière très générale, dès la phase d'architecture du logiciel comme dans le code d'une classe, trois principes majeurs sont à retenir :

1. Faire au plus simple, quitte à affiner, complexifier une fois l'ébauche fonctionnelle (*KISS : Keep It Small & Simple*) ;

2. Ne pas se répéter. En clair, factoriser le code qui peut l'être, c'est-à-dire le rendre commun ou générique afin de pouvoir le réemployer (*DRY : Don't Repeat Yourself*) ;

7. http://fr.wikipedia.org/wiki/G %C3 %A9nie_logiciel.

3. N'ajouter une fonctionnalité que si elle est nécessaire ou demandée par les utilisateurs, afin de se focaliser sur l'important à implémenter (*YAGNI : You Ain't Gonna Need It*).

Il faut également penser à employer les patrons de conception (*design patterns*), qui sont utilisés dans le monde de la programmation orientée objet (*.Net, Java*, C++) afin de fournir des solutions efficaces à des problèmes récurrents. Autrement dit, ils décrivent des solutions standard pour répondre à des problèmes d'architecture dans le monde objet, comme par exemple n'autoriser qu'un seul et unique objet d'un certain type pour tout le programme (*« singleton »*). Leur lecture et compréhension aidera de plus le développeur débutant à bien appréhender les subtilités du monde objet.

L'uniformisation du *« nommage »* est une règle à appliquer. Les bonnes pratiques de nommage permettent de standardiser le code écrit ; elles portent notamment sur la façon de baptiser les variables, les classes, les objets. Cela rend plus lisible le code produit par différents développeurs, facilite la documentation de son code, apporte des métadonnées facilement compréhensibles (par exemple '*LayersListButton*', un objet de type bouton destiné à récupérer la liste des couches de la carte) et permet l'automatisation de fonctions avancées dans l'éditeur (mise en page du code par exemple).

La documentation[8] est un point à ne pas négliger. Elle porte classiquement sur trois domaines distincts et complémentaires : le projet, le code et le logiciel.

La documentation du projet consiste à rassembler les informations liées à la vie du projet. On y retrouve par exemple l'expression des besoins (cahier des charges), les entrées et sorties (input et output) prévues pour le programme, les documents d'architecture, les comptes rendus de réunion…

Pour documenter le code, on s'attachera simultanément à trois points d'entrée. Le premier, relativement général, porte sur les attributs, propriétés, fonctions, procédures, classes, *namespaces*. Cela apporte généralement l'aide à la saisie via l'auto complétion (*Intellisense* dans l'univers *Microsoft*). Le second s'attache à expliciter le code écrit à l'intérieur des propriétés, fonctions et procédures. Il correspond plutôt au raisonnement qu'au pseudo-code, autrement dit le pourquoi plutôt que le comment. Le dernier, plus classique et ancien, consiste à documenter le fichier dans lequel le code est écrit, au travers d'un en-tête standardisé. Suivant les outils employés (gestion de version notamment, cf. ci-après), il doit être plus ou moins étoffé. À noter, à une classe correspond généralement un fichier de code.

Dans la documentation du logiciel enfin, on regroupe les documents permettant la reprise du projet par un développeur ainsi que l'aide utilisateur. Côté développeur, décrire les problèmes rencontrés, les pistes explorées ainsi que la solution employée permet de garder une trace des réflexions engagées. Les manuels d'installation et de maintenance aideront l'administrateur système à déployer votre logiciel. Rassembler les bibliothèques tierces employées (ainsi que leur code source s'il est disponible) palliera une défaillance éventuelle de leur développeur. Un dictionnaire des classes (*Library reference*) vous aidera à clarifier et réemployer votre code, qui, s'il est bien commenté, en permettra la génération automatique. Enfin, le manuel utilisateur explicitera à l'utilisateur le fonctionnement et les subtilités de votre programme.

Phase de développement : des outils et des pratiques

Parmi les outils utilisés dans l'industrie du développement logiciel, et plus particulièrement en environnement collaboratif, le concept de forge logicielle[9] mérite qu'on s'y intéresse. Sous cette appellation sont en fait regroupés différents outils, principalement de gestion (de version, de tâches, de *bugs*…) mais également d'interaction et de communication (messagerie, Wiki, discussion en ligne, etc.) liés entre eux. Souvent accessibles en ligne, les plus connus se nomment *SourceForge, Codeplex, Hosting on Google Code*…

La gestion de version est l'outil à employer en géomatique qui fournira le retour sur investissement le plus rapide ainsi que les usages les plus variés. Dans le développement (d'un logiciel ou d'un script), on modifie le code au cours du temps (par des remplacements, des ajouts, des suppressions). La gestion de version enregistre ces modifications et y associe commentaires, dates, auteurs, *tags*. Cet outil permet alors, dans un usage minimal de ses capacités, outre une sauvegarde distante, la restauration d'un projet à une étape donnée. Un des logiciels le plus connu et fournissant seul cette fonctionnalité est *Subversion*[10] (SVN), distribué sous licence *Apache* et *BSD*. Depuis le 14 février 2010, SVN est devenu officiellement un projet de la Fondation *Apache*, prenant le nom d'*Apache Subversion*. On peut y adjoindre un client s'intégrant à l'explorateur *Windows* et facilitant son utilisation, comme par exemple *TortoiseSVN*[11] .

Un second outil d'une forge utile au géomaticien est le gestionnaire de tâches, que l'on peut déjà connaître en gestion de projet classique. Son emploi permet notamment de segmenter un projet en tâches, étapes élémentaires, pour y adjoindre des ressources (humaines,

8. http://fr.wikipedia.org/wiki/Documentation_logicielle.
9. http://fr.wikipedia.org/wiki/Forge_ %28informatique %29.

matérielles…). En y renseignant un état d'avancement, le suivi du projet peut être effectué et communiqué. Son emploi peut également aider à accumuler l'expérience, en gardant une trace des problèmes rencontrés, du temps passé sur chaque tâche, expérience pouvant s'avérer utile lors de la planification d'un projet ultérieur. Néanmoins, l'usage de la gestion de tâches peut être compliqué, et pas seulement dans sa mise en place : il nécessite de la rigueur dans le renseignement des actions. Cet outil est donc à réserver pour des projets où la complexité dépasse un géomaticien seul, autrement dit lorsque le projet est trop complexe, trop étalé dans le temps pour qu'une seule personne l'appréhende facilement dans sa globalité, ou dès lors qu'il y a une collaboration étroite.

Le troisième et dernier outil dont nous parlerons ici est le gestionnaire de demande (assimilable au gestionnaire de *bugs*). Son emploi permet de centraliser les problèmes, demandes, ou *bugs* rencontrés accompagnés de leurs caractéristiques : description, commentaires, fonctionnalité touchée, capture d'écran. Ils peuvent être également hiérarchisés (de mineurs à bloquants) et archivés lors de leur résolution. L'emploi de cette fonctionnalité s'impose quand le dialogue entre l'utilisateur et le développeur est complexe : utilisateurs extérieurs à l'entreprise, en nombre élevé, peu ou pas à même de décrire le problème par téléphone…

À travers ces trois outils principaux, le périmètre fonctionnel de l'ensemble professionnel et collaboratif de développement logiciel qu'est la *« forge »* est relativement clair. Il conviendrait cependant d'ajouter que sa force principale est de lier ses outils entre eux. Une demande de fonctionnalité, un *bug* peuvent être reliés à une tâche, elle-même en relation avec une version. Autrement dit, toute modification du code correspond à un auteur pour une date donnée en réponse à une tâche issue d'une demande…

À part ces outils, quelques pratiques sont encore à prendre en considération. La principale d'entre elles concerne les *logs* de votre application. Si les studios de développement permettent à l'heure actuelle d'employer un mode *debug*, utile à l'investigation et à la recherche d'erreurs, il est nécessaire de conserver un mécanisme de *logs*. Bien que pouvant être perçu comme coûteux, de part l'ajout de lignes de codes qu'il nécessite, sa mise en place facilitera la recherche d'erreurs ou la compréhension de celles-ci *a posteriori*. Ainsi, dès le début du projet, son intégration doit être pensée, et ce d'autant plus qu'il existe de nombreuses bibliothèques pour la quasi-totalité des langages. Concrètement, son emploi permet d'enregistrer des événements (*instanciation* de classe, valeur entrée par l'utilisateur…) avec un statut (informatif, avertissement, erreur) au sein d'une base de donnée (fichier, SGBD, mail, etc.). Idéalement, le mécanisme de *logs* doit pouvoir être activé ou désactivé une fois le logiciel déployé.

Une fois le code écrit, il est souhaitable de lui appliquer les étapes de *refactoring* et d'optimisation. Le *refactoring* consiste à réviser et modifier régulièrement le code écrit, sans en changer le comportement avec comme objectif principal de faciliter la maintenance. On s'attachera notamment à éclater les fonctions et méthodes trop longues, à corriger le nommage inconsistant et à éliminer le code mort (inutile). L'optimisation, quant à elle, intervient aux environs de la sortie de la version *bêta* de l'application. Là encore, on change le code sans en modifier le comportement, dans le but notamment de réduire le temps d'exécution d'une fonction, de diminuer l'espace occupé par les données et le programme ou la consommation d'énergie.

Dernier point, les tests que passera l'application. Le plus simple et le plus répandu correspond souvent à la version *bêta* de l'application ; on le nomme *« test holistique »*, ce qui signifie tout simplement un test en situation d'utilisation. Il vérifie que l'ensemble des composants de l'application fonctionnent correctement, que l'installation sur des machines de configurations parfois différentes s'effectue sans erreur. Les scénarios d'utilisation sont joués, avec l'emploi de données réelles, parfois plus volumineuses que celles utilisées pendant le développement. L'action des utilisateurs peut également être étudiée, afin de corriger l'ergonomie ou de pallier des manipulations non prévues.

Plus complexes et plus récents, les tests unitaires peuvent être utilisés par des développeurs aguerris. Ils consistent à écrire des scenarii, sous forme de code, afin de vérifier le code principal (!).On s'assure ainsi du fonctionnement d'une partie circonscrite du logiciel. De part la répétition qu'ils permettent, ce sont de véritables argument qualité, leur usage permet également de tester la non-régression (vérifier l'impact d'une modification du code sur le reste du logiciel) et aide au suivi de l'avancement du projet.

Exemple pour le langage .NET

Concernant les principes de développement et notamment la documentation, *.NET* utilise un fichier XML, lié aux fichiers contenant le code. Dans *Visual Studio*, le menu contextuel d'une classe, fonction, méthode, propriété ou attribut laisse apparaître l'item *« Insérer un commentaire »*. Au clic, l'IDE[12] générera le squelette du commentaire. Un *plug-in* gratuit appelé GhostDoc[13] permet d'aller plus

10. http://subversion.tigris.org/.
11. http://fr.wikipedia.org/wiki/Forge_ %28informatique %29.

loin en pré-remplissant de façon beaucoup moins sommaire les commentaires. De plus amples détails sont disponibles facilement en ligne, comme par exemple l'article *« Bien commenter et documenter son code»* sur le site Developpez[14].

La *Library reference* peut ensuite être relativement facile à construire, en employant l'utilitaire *SandCastle*, qui a également fait l'objet d'un article clair sur *Developpez*[15].

Afin de coller au plus prés des recommandations de *Microsoft*[16], les outils *FxCop* et *StyleCop* peuvent être employés. *FxCop* est de préférence employé sur le code compilé et axe ses investigations sur la sécurité, les conventions de nommage, la performance, l'interopérabilité et les bonnes pratiques. *StyleCop* s'intéresse quand à lui plutôt au code source et s'attache au respect des bonnes pratiques de présentation, documentation, ordonnancement, facilité de lecture. Ces deux logiciels supportent l'insertion d'attributs dans le code (sur les *namespaces*, classes, fonctions…) afin d'indiquer que l'on déroge à une règle en toute conscience.

Un des premiers outils génériques pour *.NET* à employer est un *plug-in* pour *Visual Studio* appelé *CodeRush Xpress*[17], qui aide à la lecture du code en renforçant notamment la lisibilité de sa structure via l'emploi de lignes de tabulations et en soulignant les occurrences au survol d'un objet. Le *refactoring* au quotidien est également facilité sur les changements des signatures, une aide sur les expressions et des propositions de changements.

Le second, dénommé *Reflector*, permet le désassemblage d'un code compilé (*exe* ou *dll*) ce qui conduit, au minimum, à pallier l'absence de documentation et, au mieux (sauf si le code à été crypté), autorise la lecture du code source. De nombreux *add-ins* lui ont été adjoints, comme *FileDisassembler* qui autorise l'export/désassemblage dans le langage de son choix d'une *assembly*. Il vient cependant de changer de modèle économique (de gratuit, il devient payant dans ses versions à venir).

Enfin, dans les pratiques, on retiendra le *framework de logs Log4Net*, simple d'appréhension et pourtant complet et efficace. Ultime astuce, pour laisser temporairement une remarque ou une idée de côté sans l'oublier, il est possible d'insérer en commentaire un *« TO-DO »* ou tout autre mot clé défini en jeton. Cela permet de retrouver, à la façon d'un marque-page, ce commentaire (ainsi que le fichier de code dans lequel il se trouve, accompagné de la ligne concernée). On bénéficie alors également directement dans *Visual Studio* d'une liste de ces marques pages (tâches personnelles), dans la fenêtre *« liste des tâches »*.

La phase de livraison

Cette phase s'effectue une fois l'écriture du code terminé. Il s'agit de tout rassembler pour préparer le transfert du logiciel terminé vers les utilisateurs. On y retrouve le *packaging*, qui comprend notamment la compilation du code ainsi que la mise en relation de l'ensemble des composants du projet : logiciels, modules, documentation… Vient ensuite le déploiement, qui consiste notamment à installer l'application chez l'utilisateur ainsi qu'à l'aider à la prise en main de l'outil. Ultime étape, la réception. On y fait généralement constater au client que l'ensemble des livrables (exécutables, manuels, …) a été fourni, que les fonctionnalités demandées ont été implémentées, et qu'en l'état actuel (sans garantir l'absence totale de *bugs* !), le logiciel est utilisable en production. La réception peut s'accompagner du passage en commun et de façon contradictoire des cahiers de tests, s'ils ont été rédigés.

Conclusion

Les principes, outils et pratiques explicitées dans cet article sont utilisables par tout géomaticien amené à développer pour une application ou un script. Cela ne sous-entend cependant pas leur adoption irraisonnée et systématique. Volontairement synthétique, leur description devrait cependant vous permettre d'en cerner leurs atouts ou de susciter la curiosité et un complément d'information, afin de ne sélectionner que les plus utiles.

Avec une mise en place réussie, il sera plus simple d'obtenir une assurance sur la qualité du logiciel produit tout en facilitant son élaboration et la gestion du projet.

Remerciements

Je remercie Le Doc, Lud, Warg, David, ClaireLk du *ForumSIG*, ainsi que Christelle, Benoît, et les autres pour leurs relectures et commentaires sur cet article et pour nos échanges géomatiques et informatiques toujours fructueux. ❍

12. Integrated Development Environment : outil de développement tout en un regroupant éditeur, compilateur, débogueur, ainsi que des fonctions annexes (par exemple, versionneur.
13. http://submain.com/products/ghostdoc.aspx .
14. http://vincentlaine.developpez.com/tuto/dotnet/comdoc/.
15. http://philippe.developpez.com/articles/*SandCastle*/.
16. Les bases dans « Design Guidelines for Class Library Developers », http://msdn.microsoft.com/en-us/library/czefa0ke %28VS.71 %29.aspx, mises à jour pour le .Net Framework (2.0, 3.0, 3.5 et 4.0) dans « Design Guidelines for Developing Class Libraries », http://msdn.microsoft.com/en-us/library/ms229042 %28v=VS.90 %29.aspx (version du .Net Framewok 3.5).
17.http://www.devexpress.com/Products/Visual_Studio_Add-in/CodeRushX/.