UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PROPOSITION D'UNE MÉTHODE BASÉE SUR LA CRÉATION ET L'UTILISATION D'ARTÉFACTS AFIN DE RÉSOUDRE DES PROBLÈMES DANS LES COURS D'INTRODUCTION À LA PROGRAMMATION

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE DE GESTION

PAR
PIERRE BÉLISLE

FÉVRIER 2006

REMERCIEMENTS

À la mémoire de mon ami Daniel Delay

Je n'eusse pas cru devoir me contenter des opinions d'autrui un seul moment si je ne me fusse proposé d'employer mon propre jugement à les examiner lorsqu'il serait temps.

René Descartes

J'offre cette recherche à la mémoire de mon ami, Daniel Delay qui m'a obligé à lui promettre de faire des études universitaires. Il n'a malheureusement pas eu le temps de voir le résultat de son effort.

MÉGA ÉNORME GROS MERCI de tout mon cœur à mon épouse, Patricia, qui m'a toujours encouragé dans ce projet et qui m'a permis de le réaliser. Je remercie aussi la maman en elle, qui a dû s'occuper seule des enfants pendant mon absence. Une tâche dont elle s'est acquittée avec mention. Je désire aussi demander pardon à mes enfants, Moïra, Nali, Kaël et Méliane, qui ont dû se passer assez souvent de leur père. Même s'ils sont en assez bas âge, je sais qu'ils en ont souffert un peu, mais ils ne me l'ont jamais fait sentir, et pour cela je les en remercie.

ÉNORME MERCI à M. Robert Dupuis, qui a su me diriger et qui a dû subir la lecture de mes nombreuses versions. Je peux dire que, sans lui, cette recherche n'aurait pas eu de version finale.

GROS MERCI à Mme Louise Laforest et à M. Guy Tremblay, qui ont accepté d'être membres du jury d'évaluation de ma recherche. Plus particulièrement à M. Tremblay, qui a permis, par sa révision, de remettre une recherche d'une plus grande qualité.

MERCI SPÉCIAL à M. Richard Labonté qui m'a permis d'être accepté à la maîtrise en informatique de gestion et, ensuite qui m'a permis d'effectuer un travail que j'aime en s'engageant personnellement lors de mon embauche à l'École de technologie supérieure. Sa confiance en moi m'a beaucoup servi et je lui en serai éternellement reconnaissant.

Merci à mon grand ami, M. Jean Robert, qui m'a appuyé dans toutes mes démarches autant professionnelles que personnelles et qui m'a toujours été de bon conseil.

Merci à Messieurs Marc Bouisset, Normand Séguin, Fernand Gaudet et Louis Mailhot qui m'ont aidé à un moment donné dans ma formation.

Finalement, merci à tous les autres qui m'ont aidé de quelque façon que ce soit et que je n'ai pas cité explicitement.

RÉSUMÉ

Nous pouvons voir le développement d'un logiciel selon deux perspectives. La première concerne le développement, par un programmeur ou une petite équipe de programmeurs, d'un petit logiciel, avec un utilisateur et un seul ordinateur. C'est ce qu'on appelle de la programmation à petite échelle (*programming in the small*). Ce genre de développement s'enseigne dans les cours de programmation en informatique (*computer science*). La deuxième perspective consiste à faire de la programmation à grande échelle avec plusieurs équipes de programmeurs, plusieurs fonctionnalités, plusieurs utilisateurs et même, parfois, plusieurs ordinateurs et serveurs. C'est ce qu'on appelle le génie logiciel (*software engineering*), qui s'enseigne dans des baccalauréats en génie logiciel.

Même si beaucoup de progrès a été fait en génie logiciel, il reste qu'il existe encore des lacunes dans l'enseignement de cette discipline. Les difficultés se situent, entre autres, dans les cours d'introduction à la programmation. Il y a plusieurs façons d'enseigner le développement de logiciels. Cependant, peu importe celle qui est employée, la transition entre la programmation à petite échelle et la programmation à grande échelle n'est pas facile. Nous désirons donc créer une méthode qui favorise la résolution de problèmes, tout en facilitant la transition ultérieure vers les grands projets. Nous croyons également que la transition serait plus facile si ladite méthode de développement utilisait des principes de génie logiciel déjà utilisés et reconnus. C'est pourquoi nous avons tenu compte d'un corpus de connaissances en génie logiciel (Abran et al., 2001) dans la construction de la méthode.

LISTE DES FIGURES

Figure

INTRODUCTION

Le cycle de vie logiciel est l'ensemble des étapes qu'on doit franchir pour développer un logiciel. Les étapes du cycle de vie sont : exigences, conception, construction, intégration et tests et maintenance. Le cycle de vie d'un logiciel trace la vie du logiciel à partir du moment où il émerge d'une première idée, jusqu'à son remplacement ou son abandon (Robillard et al., 2002). Il existe plusieurs modèles de cycle de vie (cascade, spirale, par prototype, itératif, etc.). Le modèle en cascade est le plus utilisé dans l'enseignement des principes de base en programmation à petite échelle. Il est maintenant reconnu que ce modèle est valable dans de petits projets où il n'y a pas, ou très peu, de possibilité de changement d'exigences. Dans la réalité, lors d'une modification d'exigences ou lors d'une détection d'erreurs, il faut revenir sur une étape précédente du cycle et faire les correctifs sur la documentation appropriée. Une première constatation, que nous pouvons faire à ce stade-ci, est qu'il faut être en mesure de revenir à la bonne étape dans le cycle de développement pour permettre d'effectuer la modification ou la correction d'erreurs.

Dans les méthodes de développement, chaque étape du cycle de vie devrait être documentée que ce soit sous forme de texte ou de diagramme. Un des problèmes des cours d'introduction à la programmation est qu'on enseigne sommairement comment produire ces documents, mais pas comment les utiliser.

Des études ont été réalisées pour combler les lacunes de l'enseignement dans des cours d'introduction. Certaines disent qu'il est primordial d'enseigner d'abord la résolution de problèmes, et d'autres affirment que c'est le processus de développement qu'il faut enseigner en premier. Peu importe la tangente de la recherche, si elle préconise un aspect, elle en néglige un autre. Nous croyons que les deux aspects sont importants et nous voulons promouvoir l'enseignement d'une méthode de développement qui permettra de résoudre des problèmes en **utilisant** la documentation produite lors du développement de logiciel.

Même si la discipline du génie logiciel est relativement jeune dans le domaine de l'ingénierie, il existe des notions biens établies, telles que les différents modèles du cycle de vie d'un logiciel, les techniques de programmation (traditionnelles et orientées-objet) et les outils, tels que les diagrammes structurels, les diagrammes de flux de données, etc. Il existe également un corpus de connaissances en génie logiciel (Abran et al., 2001) qui fait état de ce qui est actuellement utilisé dans le monde du génie logiciel. Nous désirons utiliser ce qui est généralement reconnu en génie logiciel, en y apportant quelques modifications, afin d'en faciliter l'apprentissage dans les cours d'introduction à la programmation.

Les domaines dans lesquels se fera notre travail sont les milieux universitaire et collégial et ce, seulement pour le contenu des deux premiers cours d'introduction à la programmation, dans les cursus en informatique (*computer science curriculum*) et dans un paradigme procédural. De façon générale, les notions enseignées dans ces cours sont :

- un langage procédural (variables, constantes, boucles, sélections, procédures, fonctions, passage de paramètres, tableaux, structure de données et manipulation de fichiers)
- des principes de conception et de construction (modularité, représentation cachée, types abstraits (pile, file, arbre, etc.), généricité, approche descendante, approche ascendante)
- le cycle de vie logiciel
- la documentation de base

Ce travail vise d'abord à créer une méthode pour les cours d'introduction à la programmation de niveaux universitaire et collégial dans le but de combler certaines lacunes. Ceux qui bénéficieront principalement de ce travail sont les enseignants et les étudiants dans les cours d'introduction à la programmation. D'un autre côté, le génie logiciel pourra voir confirmer une utilisation du corpus de connaissances (Abran et al., 2001), et avoir un aperçu de son application, dans l'enseignement des cours d'introduction.

La structure de ce mémoire est la suivante :

1. Chapitres I et II : Nous démontrons, malgré les efforts de la recherche, qu'il existe encore des lacunes dans l'enseignement des cours d'introduction à la programmation. De plus, nous définissons ce qu'est une méthode de développement et nous présentons le génie logiciel et le guide au corpus de connaissances en génie logiciel (Abran et al., 2001). C'est avec l'utilisation de ce dernier que nous espérons offrir une meilleure transition entre la programmation à petite échelle et la programmation à grande échelle.
2. Chapitre III : Nous présentons l'organisation de la méthode.
3. Chapitres IV à VII : Nous retenons les connaissances du corpus (Abran et al., 2001) applicables pour construire la méthode qui décrit comment utiliser chaque étape, du cycle de vie du logiciel, et les liens à établir entre chacune d'elles.
4. Chapitre VIII : Nous avons présenté la méthode auprès d'enseignants qui ont des préoccupations semblables dans leur enseignement et qui ont une connaissance du génie logiciel. Nous analysons les commentaires des enseignants participants et nous justifions les décisions que nous avons prises par rapport à leurs commentaires.
5. Chapitre IX : La conclusion.

CHAPITRE I

PROBLÈMES AVEC LES MÉTHODES EXISTANTES

Plusieurs communications de recherche ont été effectuées au cours des années concernant les problèmes sur les méthodes employées dans les cours d'introduction à la programmation, entre autres dans le cadre du *Technical Symposium on Computer Science Education* (SIGCSE), de la *Conférence on Software Engineering Education and Training* (CSEE&T) et du *Forum for Advancing Software engineering Education* (FASE). Nous ferons ressortir, dans ce chapitre, les problèmes de développement et de résolution de problèmes en programmation qui sont identifiés ainsi que des solutions qui ont été proposées. Nous voulons montrer que les problèmes liés à l'enseignement des cours de programmation ne sont pas nouveaux et qu'ils existent encore de nos jours. Par le fait même, nous montrerons les lacunes qui ont été identifiées au cours des années et qui n'ont pas encore été résolues.

1.1. RÉSOLUTION DE PROBLÈMES OU MÉTHODE DE DÉVELOPPEMENT

C'est depuis le début des années 70 que les chercheurs en informatique, par la suite en génie logiciel, tentent de trouver ce qui ne va pas dans les cours d'introduction à la programmation. Déjà en 1974, David Gries (Gries, 1974) établit, dans un article du *Technical Symposium on Computer Science Education* (SIGCSE), que plusieurs collèges et universités avaient mal interprété le but des cours d'introduction, en enseignant la programmation simple plutôt que des techniques de résolution de problèmes. Il mentionne qu'à cette époque, peu était écrit sur la façon de développer un logiciel informatique, de trouver des idées, de structurer ses pensées et d'arriver à un programme bien structuré, bien écrit et lisible. Il suggère donc l'algorithmique comme outil de résolution de problèmes.

Les propos de Gries sont repris par une autre équipe de chercheurs en 1979 (Hyde et al., 1979) qui affirment que la méthodologie pour concevoir le plan d'une solution d'un problème est appelée « le processus de résolution de problèmes[1] », en ajoutant que l'algorithme n'est qu'une étape dans le plan. Ils démontrent qu'il est possible d'intégrer une méthode de développement et des outils pour résoudre des problèmes dans les cours d'introduction. Leur plan de développement suit les étapes suivantes : la formulation du problème, l'analyse du problème, le développement des algorithmes, la traduction dans un langage de programmation, la vérification du programme et l'écriture du document final. Nous pouvons reconnaître ici les grandes étapes du cycle de vie traditionnel.

---

[1] *Problem solving process.*

En 1986, dans l'esprit que la résolution de problèmes demeure une lacune, Peter Henderson (Henderson, 1986), un professeur de Stony Brook-New York, écrit qu'il y a plusieurs livres qui ont l'expression « *problem solving* » dans leur titre. La plupart de ces livres n'abordent pas directement la résolution de problèmes et ne font qu'enseigner un langage de programmation spécifique. Il y est mentionné que la plupart des étudiants dans les cours d'introduction sont exposés à des « *toy programs* », alors qu'ils devraient être exposés aux exigences, aux documents de conception, au code commenté et à une version exécutable. Il conclut cet article en écrivant que les étudiants éprouvent de la difficulté à comprendre le contexte et quels concepts doivent être appliqués, qu'il est nécessaire de développer des exemples pour leur permettre de concrétiser ces concepts, que c'est aux éducateurs d'établir les fondations de la discipline de l'informatique et qu'il est difficile de comprendre le génie logiciel si on ne sait pas comment programmer.

En 1987, Peter Henderson (Henderson, 1987) revient avec une proposition de cours d'introduction. Il mentionne qu'on donne trop d'importance à la programmation et pas assez aux fondements. Il écrit que plusieurs reconnaissent que la résolution de problèmes est une déficience majeure des étudiants. Il propose donc un cours où on enseigne les fondements mathématiques servant à la résolution de problèmes et, par la suite, les principes de conception d'un programme. Toujours selon Henderson, il y a trop de principes de conception qui sont enseignés sans que les étudiants ne soient guidés correctement. De plus, il ajoute que les étudiants éprouvent de la difficulté à établir des liens entre les concepts et la réalité; c'est pourquoi il faut enseigner les concepts avant le développement.

Rex E. Gantenbein (Gantenbein, 1989) introduit le cycle de vie du logiciel comme modèle de développement en programmation. Il fait ressortir que le cycle de vie devrait être enseigné à tous les niveaux de la formation en informatique. Un problème identifié par D. Bell (Bell, 1987) révèle que les étudiants ont tendance à voir les étapes du cycle de vie de façon distincte, donc qu'ils les approchent séquentiellement, alors qu'il est connu que ces étapes ne sont pas toujours linéaires et peuvent se chevaucher. Gantenbein propose quand même d'enseigner un modèle de développement de base qui reflète les étapes du cycle de vie logiciel plutôt que d'enseigner seulement un langage de programmation. Il décrit quelles en sont les étapes dans son article, mais ne décrit pas comment les réaliser. Il écrit finalement que cette méthode d'enseignement n'est pas meilleure ou pire que d'autres, mais qu'elle offre de bonnes techniques et un cadre de développement.

En 2001, un logiciel, dénommé *UNLOCK*, a été développé sous prétexte que les étudiants manquent d'outils fondamentaux de résolution de problèmes. En fait, non seulement ils ne peuvent résoudre des problèmes informatiques, mais ils ne peuvent résoudre des problèmes en

général (Beaubouef et al., 2001). Ce logiciel offre des problèmes et des indices qui permettent aux étudiants de trouver la solution, s'ils n'y arrivent pas seuls.

1.2. PROBLÈMES DES ENSEIGNANTS

Les problèmes persistant toujours, diverses solutions ont été explorées. Dans les années 90, quelques études ont été faites concernant la préparation des enseignants en programmation. L'une d'elle (Kushan, 1994) relève les problèmes déjà cités concernant la résolution de problèmes dans d'autres ouvrages (Soloway et al., 1982; Linn's, 1985; Maher et al., 1986). On y affirme que les recherches concernant la programmation et la résolution de problèmes semblent indiquer que ce n'est pas seulement ce qui est enseigné, mais la manière dont cela est enseigné qui fait la différence.

L'étude de Kushan fait aussi référence à d'autres études qui ont été conduites sur les outils de résolution de problèmes (Soloway et al., 1982; Clements et al., 1984; Clements, 1986 et 1990; Mayer et al., 1986; Soloway, 1986; Mayer, 1987). Ces études mentionnent que des techniques de résolution de problèmes peuvent être acquises, si la programmation est enseignée en utilisant des stratégies d'enseignement et de programmation spécifiques. Ces études ont démontré que l'accent est mis sur seulement huit parmi douze concepts identifiés comme étant importants, seulement cinq parmi dix stratégies de résolution de problèmes identifiées comme importantes sont soulignées et seulement quatre parmi huit stratégies d'enseignement les plus communes sont utilisées par les enseignants des cours d'introduction.

1.3. ASPECT PSYCHOLOGIQUE

Il y a eu également des études qui démontrent que l'aspect psychologique est important en pédagogie et dans l'apprentissage de la programmation. Leon E. Winslow (Winslow, 1996) fait la synthèse des études psychologiques sur l'enseignement de la programmation. Il mentionne que ces études psychologiques ont noté que plusieurs autres études antérieures sur la programmation étaient faibles quant à la méthodologie à cause, entre autres, de la complexité du processus de développement et de la pauvreté de l'expérience en conception (Sheil, 1981). Winslow affirme que les recherches en psychologie ont amélioré considérablement les méthodologies de toutes sortes. Une après l'autre, les études en psychologie en sont arrivées à la même conclusion : les programmeurs novices connaissent la syntaxe et la sémantique d'instructions individuelles, mais ils ne savent pas comment les combiner pour obtenir un programme valide, ou encore ils savent comment résoudre le problème, mais n'arrivent pas à traduire la solution dans un langage informatique. Ces études auront permis de mieux comprendre les difficultés des étudiants. Winslow mentionne que du point de vue pédagogique, les modèles utilisés sont cruciaux pour construire la compréhension des étudiants

(modèle de représentation des données, des contrôles, de développement, etc.). Il écrit que si les enseignants omettent l'enseignement de modèles, les étudiants créeront leurs propres modèles qui seront de qualité douteuse.

1.4. AUTRES PROBLÈMES

Dans un rapport du département d'informatique de l'université d'Otago en Nouvelle-Zélande (Robins et al., 2001), les auteurs constatent qu'il y a encore des problèmes d'apprentissage qui ne sont pas réglés. Ils relèvent les points importants qui ont été amenés au cours des années :

- la différence entre les novices et les experts et ce qu'il faut à un novice pour devenir expert
- la distinction à faire entre les connaissances des faits (syntaxe, mécanique des instructions, etc.) et les stratégies pour utiliser ces connaissances (à quel moment doit-on utiliser les instructions)
- la différence entre comprendre un programme et en créer un, etc.

Leur étude a pris une toute autre direction que celle d'appliquer ou d'améliorer une méthode. Ils se sont penchés sur les différentes catégories de novices et les problèmes reliés à cette catégorie. Même si nous sommes d'avis qu'il y a quelque chose à faire individuellement auprès de chaque étudiant, nous ne désirons pas traiter de cet aspect. Il reste néanmoins que c'est un rapport important qui fait une très bonne revue de la documentation sur des recherches qui ont été effectuées sur les problèmes dans l'apprentissage de la programmation pour les novices.

Tous s'entendent pour dire que l'enseignement des cours d'introduction comporte des lacunes soit dans l'aspect résolution de problèmes, soit dans l'aspect développement. Certains chercheurs tendent à favoriser la résolution de problèmes tandis que d'autres favorisent les méthodes de développement. Les études psychologiques faites sur la programmation démontrent qu'il faut des modèles pour aider les étudiants dans leur apprentissage. Chacune des variantes d'enseignement utilise des modèles sauf que si l'accent est mis sur la résolution de problèmes, les méthodes de développement sont mises de côté et vice versa. Nous désirons une méthode qui jumelle les deux aspects pour obtenir une méthode de développement qui aidera à résoudre des problèmes. De cette façon, nous espérons pouvoir enseigner en même temps ces deux aspects importants dans l'introduction à la programmation.

CHAPITRE II

PROBLÈMES DE L'ENSEIGNEMENT DU GÉNIE LOGICIEL

Dans un article du *Forum for Advancing Software engineering Education* (FASE) sur les cours d'introduction à la programmation en génie logiciel, Rick Duley (Edith Cowan university, Perth, Western Australia) écrit :

« *There should be no assumption of any prior experience in programming on the part of any first-year student. Inevitably, in a SE course, most students will have prior experience - a fact which may be a source of problems. Self-taught or badly-taught students may be fraught with bad habits which they will have to break if they are to succeed.* » [2]

Selon son affirmation, il est difficile d'enseigner le génie logiciel à des programmeurs qui ont pris de mauvaises habitudes de programmation. Nous présenterons dans ce chapitre des études qui démontrent que les aspects du génie logiciel devraient être enseignés dans les cours d'introduction. Cela nous permet de justifier l'utilisation du corpus de connaissances en génie logiciel (Abran et al., 2001) dans la construction de notre méthode.

2.1. SITUATION DU GÉNIE LOGICIEL DANS LES CURSUS EN INFORMATIQUE

Les problèmes identifiés par les programmes collégiaux ou universitaires ont suscité des discussions sur la définition même des cursus en informatique. Le but des discussions était de former plus de programmeurs ayant des connaissances en génie logiciel dans les cursus en informatique. Dans un éditorial sur ces cursus, un professeur émérite (Turner, 2001) conclut que malgré les progrès des dernières décennies, il reste encore plusieurs questions importantes comme : Quels sont les concepts de base qui doivent être inclus dans tous les programmes ? Quel matériel enseigné actuellement peut être éliminé pour faire de la place à de nouveaux concepts ? Quels sont les chemins qui permettront d'éduquer des étudiants qui proviennent de profils différents, pour qu'ils apprennent le matériel fondamental et acquièrent les capacités nécessaires pour fonctionner efficacement comme un professionnel de l'informatique ? Et comment peut-on monter un programme qui est efficace dans la préparation des étudiants pour faire face à la variété toujours grandissante des emplois en informatique ?

[2] tab.computer.org/fase/fase-archive/v12/v12n05.txt

En 1995, une table ronde (McCauley et al., 1995) de professeurs provenant de diverses universités épousent la philosophie que les principes de génie logiciel doivent être considérés comme parties intégrantes des cursus en informatique. En 2000, une autre table ronde (McCauley et al., 2000) se prononce sur la situation actuelle. Nell Dale, une des expertes, écrit que des efforts pour introduire les principes de génie logiciel ont été faits, mais que nous ne sommes pas rendus où il faudrait. Elle mentionne que le processus de tests devrait être introduit dès le premier cours de programmation et pour toutes les étapes du cycle de vie. Thomas Hilburn, un autre expert, écrit qu'il y a eu amélioration en ce qui a trait à chaque étape du développement. Malheureusement, il ne croit pas que ce soit couvert en profondeur dans la plupart des programmes. Susan Mengel, quant à elle, écrit que si les étudiants commencent tôt dans les programmes à utiliser des principes de génie logiciel, ils les trouveront utiles dans leurs projets et dans leurs futurs cours. Il ressort donc que le génie logiciel doit être introduit dans les programmes le plus tôt possible.

2.2. SUGGESTION DE NOUVEAUX CURSUS EN INFORMATIQUE

D'autres part, plusieurs versions de cursus ont été proposées par diverses universités et cela continue. Chacune de ces universités tente de proposer des façons de concevoir son programme pour que les étudiants puissent mieux se préparer au monde du génie logiciel. Dans le rapport du *Computing Curriculum* 2001 (CC 2001) de l'ACM/IEEE-CS, qui fait suite à celui de 1991, il est mentionné qu'il y a plusieurs façons acceptables de construire un cursus en informatique. Les programmes peuvent enseigner dans les premiers cours : l'approche algorithmique, l'approche objet, l'approche impérative, l'approche fonctionnelle, l'approche matériel (*hardware*) ou l'approche en profondeur. Ce rapport propose d'intégrer des cours de génie logiciel pour améliorer les programmes et cela peu importe l'approche, alors que les reproches faits sont qu'il y a une omission d'expérience dans la conception et un manque de principes du génie logiciel (Myers, 2001). Notre avis est que le fait d'ajouter des cours de génie logiciel après les cours d'introduction ne réglera pas le problème du manque de préparation des étudiants au moment où ils commencent leur formation en génie logiciel.

2.3. POURQUOI UTILISER LE GUIDE AU CORPUS DE CONNAISSANCES

C'est en préface d'une proposition de cursus pour l'université de Kaiserslautern, en Allemagne, que M. Rombach dénonce le manque de références au corpus de connaissances en génie logiciel. La plupart des cursus en génie logiciel prennent leurs bases seulement du point de vue de l'informatique (*computer science*). Les cursus ignorent l'existence du corpus de connaissances, [...]. Est-il acceptable de diplômer des étudiants en génie logiciel sans l'idée de base de modèles économiques pour le logiciel, le génie logiciel, [...]. Est-il acceptable de

diplômer des étudiants sans une compréhension du corpus de connaissances ? [...] (Rombach, 2003). [3]

Pourtant, en 1999, une équipe de chercheurs avaient préparé un guide pour l'éducation en génie logiciel (Bagert et al., 1999). Ce guide a été conçu pour offrir une assistance aux professeurs des facultés qui désirent avoir un programme de qualité en génie logiciel. Dans la préface de ce document, les chercheurs mentionnent que peu de programmes universitaires consacrent du temps à l'enseignement de la modélisation des exigences, des méthodes de conception et d'architecture, etc.

Il existe un sous-comité responsable d'identifier l'ensemble des connaissances à enseigner (*Software Engineering Education Knowledge*) qui a basé ses travaux sur SWEBOK. De même, le comité du CCSE (*Computing Curriculum Software Engineering*) stipule qu'un guide pour les cursus en génie logiciel doit être basé sur des définitions appropriées des connaissances en génie logiciel. C'est pour toutes ces raisons que nous désirons que notre méthode, qui se veut un outil d'introduction au génie logiciel par de petits projets, soit inspirée des connaissances actuelles en génie logiciel. Ces connaissances sont décrites dans le guide au corpus de connaissances que nous utilisons (Abran et al., 2001).

2.4. DÉFINITIONS

Voici les définitions que nous avons employées :

Méthode de développement de logiciel : Un ensemble ordonné de manière logique de principes, de règles, d'étapes permettant de parvenir à la création, à la réalisation et à la mise au point d'un programme, d'un logiciel, d'une application ou d'un système.

Génie logiciel : Ensemble des connaissances, des procédés et des acquis scientifiques et techniques mis en application pour la conception, le développement, la vérification et la documentation de logiciels dans le but d'en optimiser la production, le support et la qualité. (http:// www.granddictionnaire.com)

Tel que mentionné précédemment, un des problèmes que nous voulons aider à résoudre est la transition de l'informatique vers le génie logiciel. Nous avons pu voir dans ce chapitre qu'il y a un problème d'intégration des notions de génie logiciel aux premiers cours de programmation. Plusieurs institutions tentent de résoudre leurs problèmes en modifiant les cursus en informatique et en y ajoutant des cours de génie logiciel. Nous croyons que si la

[3] Ce texte est une traduction de l'anglais par l'auteur.

source du problème est dans les premiers cours de programmation, la solution devrait être envisagée à ce niveau et non pas dans des cours donnés ultérieurement. C'est pourquoi nous utiliserons le guide au corpus de connaissances (Abran et al., 2001) en génie logiciel pour cette méthode et elle devra être enseignée dès les premiers cours de programmation. Ainsi cela aidera les étudiants à mieux effectuer la transition dans les cours de génie logiciel, qu'ils soient donnés dans un baccalauréat en génie logiciel ou dans un baccalauréat en informatique.

CHAPITRE III

ORGANISATION DE LA MÉTHODE

Dans ce chapitre, nous présenterons un aperçu de ce que nous fournirons à l'issue de notre réflexion, à savoir une méthode de développement qui aidera à résoudre des problèmes dans le cadre des cours de programmation de base. Nous donnerons une description de la procédure d'analyse qui permettra de produire cette méthode. Nous décrirons comment nous utiliserons le guide au corpus de connaissances du génie logiciel (Abran et al., 2001).

Pour chaque étape du cycle de vie, l'étudiant aura à produire un document que nous appelons artéfact. Ces artéfacts doivent servir à documenter les diverses étapes des exigences jusqu'aux décisions d'implantation, en passant par les stratégies de vérification et les tests. Notre méthode décrit de quelle façon ces artéfacts doivent être construits et utilisés. Elle montre plus spécifiquement quelles sont les activités à réaliser pour arriver à produire les artéfacts de chaque étape du cycle de vie et comment utiliser l'artéfact à l'étape suivante du cycle de vie. Nous espérons que les règles de développement et les liens logiques entre les étapes du cycle de vie constitueront pour les étudiants une aide à la résolution de problèmes. Nous espérons également que cette méthode permettra de développer plus facilement des logiciels à petite échelle et offrira une transition plus facile et plus efficace vers le développement à grande échelle.

3.1. DESCRIPTION DES SECTIONS

Dans la méthode que nous proposons, il y a quatre chapitres qui décrivent comment construire les artéfacts. Chacun de ces chapitres comporte les sections suivantes :

- Une introduction,
- La terminologie employée pour la réalisation des artéfacts à produire,
- Les objectifs à atteindre,
- La production de l'artéfact associé à l'étape du cycle de vie en respectant des règles de développement,
- Les liens à établir entre les étapes en utilisant l'artéfact précédent et le suivant (s'il y a lieu),
- Un mot sur les grands projets,
- Un exemple d'artéfact.

Voici une brève description des sections contenues dans la méthode pour chaque étape du cycle de vie (excluant la maintenance).

3.1.1. Terminologie

Cette section nous permet de fournir la définition des termes employés dans la construction de l'artéfact à produire.

3.1.2. Objectifs

Cette section nous permet de définir les objectifs de l'étape et le rôle de l'artéfact à produire. Par exemple, l'artéfact pour l'étape des exigences permet entre autres de séparer les exigences fonctionnelles et les exigences non-fonctionnelles.

3.1.3. Règles de développement

Cette section nous permet de fournir l'ensemble des règles permettant la création, la réalisation et la mise au point de l'artéfact à produire. Les règles sont inspirées du corpus de connaissances (Abran et al., 2001). Nous départagerons ce qui est utilisable dans le cadre de petits projets de ce qui ne l'est pas. Par exemple, nous avons inclus dans la méthode qu'une exigence doit avoir un numéro unique parce qu'il est écrit dans le guide au corpus (et que c'est applicable) qu'une exigence se doit d'être clairement identifiée. Il est aussi mentionné que les exigences servent à découvrir comment le système sera partitionné, en identifiant quelles exigences seront allouées à quels composants. Dans le cas de programmation à petite échelle, cela ne s'applique pas, alors nous l'avons omis.

3.1.4. Description de l'artéfact à produire

Nous faisons une description des sections de chaque artéfact afin de respecter les règles de développement et fournir quelques informations additionnelles.

3.1.5. Liens entre les artéfacts

Cette section nous permet de décrire les artéfacts à travers les différentes étapes logiques, et les liens à faire entre eux. Nous espérons que cela guidera les étudiants à parvenir à une solution informatique. Nous décrivons également comment utiliser l'artéfact précédent et comment passer à l'étape suivante du cycle de vie.

3.1.6. Mot sur les grands projets

Cette section nous permet de faire référence aux aspects de génie logiciel qui ne peuvent pas être pris en considération dans de petits projets, mais qu'il faudra considérer lors de plus grands projets. Le but est de montrer aux étudiants que ce qu'ils apprendront dans cette méthode est adapté à leur contexte et qu'ils auront à élargir leur approche pour des projets de plus grandes envergures.

3.1.7. Exemple d'artéfact

Nous montrons un exemple en utilisant un contexte qui permet d'utiliser les règles de développement et de démontrer les liens entre les artéfacts.

3.2. CHOIX DU PARADIGME PROCÉDURAL

Dans le rapport du département d'informatique de l'université d'Otago en Nouvelle-Zélande (Robins et al., 2001) cité au chapitre I, il est fait mention d'études qui ont exploré les différences entre des novices qui ont appris la programmation objet et ceux qui ont appris la programmation procédurale, à leurs deux premiers semestres. Pour des petits programmes, il n'y a pas eu de différence significative dans la compréhension générale des langages utilisés (Pascal et C++), mais les programmeurs utilisant le paradigme objet avaient une meilleure compréhension de ce que faisait le programme. Les résultats ont été complètement différents avec de gros programmes. Les programmeurs utilisant le paradigme procédural ont fait mieux que ceux utilisant le paradigme objet, et ce à tous les niveaux (Wiedenbeck et al., 1999). **Conséquemment, nous croyons qu'il est préférable d'enseigner le paradigme procédural dans les cours d'introduction à la programmation et c'est celui que nous utiliserons pour la méthode.**

Dans ce chapitre, nous avons présenté nos choix en ce qui a trait à l'organisation de la méthode. La méthode est divisée en sections (terminologie, objectifs, règles de développement, liens entre les artéfacts, mots sur le génie logiciel, description de la documentation et exemple) qui décrivent ce qui est important pour l'étape du cycle de vie que l'étudiant est en train de développer. De plus, nous avons justifié notre choix du paradigme procédural.

À l'annexe C, page 73, se trouvent les pages introductives de la méthode telle qu'elle sera présentée aux étudiants.

CHAPITRE IV

EXIGENCES

Ce chapitre décrira principalement les choix qui ont été faits pour la spécification des exigences. Nous ferons mention des parties qui peuvent s'appliquer à de petits projets mais nous laisserons de côté les parties qui sont importantes dans de grands projets, puisqu'elles ne s'appliquent pas dans les cours de base en programmation. Par exemple, dans les grands projets il faut définir pour la spécification des exigences : les modèles, les acteurs, le processus de support, de gestion, de qualité et d'amélioration pour faire la planification du développement (Sawyer et al., 2001, chap. 3.1.1), il faut introduire le rôle des différents intervenants (Sawyer et al., 2001, chap. 3.1.2), etc. Tout cela n'est pas nécessaire pour de petits projets.

La spécification des exigences doit être réalisée dans un projet à grande échelle pour plusieurs catégories d'exigences. Par contre, dans le cadre d'un petit projet de programmation, les exigences sont souvent prises à partir d'un énoncé de travail pratique, ce qui diminue habituellement les catégories d'exigences à considérer.

Dans les pages qui suivent, nous présenterons ce qui est généralement reconnu dans le guide au corpus de connaissances (Abran et al., 2001) pour la spécification des exigences. Nous définirons ce qu'est une exigence et les objectifs que sa spécification doit atteindre. Nous décrirons les points que nous considérons réalisables et que nous voulons voir dans la méthode. Nous poursuivrons en présentant la méthode telle qu'elle est décrite dans le chapitre précédent avec les objectifs, les règles de développement, les liens à établir avec l'étape suivante, dans ce cas-ci, l'étape de conception et un exemple de l'artéfact à produire.

NOTE : Nous présentons chaque thème concernant les exigences et nous mettons en caractères gras ce que nous retenons pour la méthode.

4.1. EXIGENCES LOGICIEL

Le guide au corpus de connaissances (Abran et al., 2001) est divisé en champs de connaissances (*knowledge area*). Il y en a un pour chaque étape du cycle de vie en plus de ceux qui décrivent la gestion du processus logiciel (*software process*). Un champ de connaissance est divisé en différents thèmes. Les thèmes qui concernent le champ de connaissances des exigences sont : le processus d'ingénierie (*engineering process*), la cueillette (*elicitation*), l'analyse, la

spécification, la validation et la gestion (*management*)[4]. Voici un petit résumé des objectifs de chacun de ces champs et ce que nous avons retenu pour la méthode :

1. Le processus d'ingénierie des exigences permet de documenter la procédure qui doit être suivie lors de l'étape des exigences. Cette section décrit :

- Le modèle de description des exigences, utilisé dans le projet.
- Les personnes impliquées dans le processus.
- Les ressources requises en termes de coûts, de ressources humaines, de formations et d'outils.
- Les mécanismes pour assurer la qualité et les améliorations du logiciel.

2. La cueillette des exigences permet d'identifier la source de chaque exigence et d'utiliser des techniques de cueillette d'informations comme : des entrevues, des scénarios, des prototypes, etc. (Sawyer et al., 2001, chap. 3.2).

3. L'analyse des exigences permet de détecter les conflits entre les exigences, de découvrir les limites du système et comment il doit interagir avec son environnement et de transformer les exigences système en exigences logiciel (Sawyer et al., 2001, chap. 3.3).

4. La spécification des exigences s'effectue à l'aide de documents tels que le *System Requirements Definition Document,* le *Software Requirements Spectifications* (SRS), le *Document Structure ans Standards* et le *Document Quality.* Dans ce thème, il est mentionné que la documentation des exigences est une précondition des plus fondamentales pour le succès du projet (Sawyer et al., 2001, chap. 3.4).

5. La validation des exigences vise à relever les problèmes avant que les ressources ne soient engagées pour poursuivre le travail. Elle vise également à s'assurer que les différents documents décrivent bien le logiciel à produire (Sawyer et al., 2001, chap. 3.5).

6. La gestion des exigences permet de décrire comment procéder lorsqu'il y a des modifications en ce qui concerne les exigences ou autres. Cette section du corpus de connaissances décrit chacune des décisions prises tout au long du cycle de vie du logiciel à produire. Par exemple : quelle classification a été choisie lors de l'analyse des exigences, quelles sont les méthodes d'acceptation et de vérification des plans de tests, quelle est la source de chacune des exigences ? (Sawyer et al., 2001, chap. 3.6).

Nous pouvons voir l'utilité de décrire le processus d'ingénierie, de cueillette, de validation et de gestion des exigences dans de grands projets mais moins dans des projets du niveau des cours

[4] Les termes ont quelque peu changé dans la version 2004 du corpus de connaissance, mais les modifications qui ont été apportées n'affectent en rien les décisions que nous avons prises.

d'introduction à la programmation. C'est pourquoi **nous n'avons considéré que l'analyse et la spécification des exigences que nous décrivons dans les paragraphes suivants.**

4.2. DÉFINITION D'UNE EXIGENCE

1. « Propriété de ce qui doit être exposé, dans l'ordre, pour résoudre un problème du monde réel. » (Sawyer et al., 2001, chap. 2.1; Pfleeger 1998; Kotonya et Sommerville, 2000; Sommerville 2001; Thayer et Dorfman, 1997)
2. La condition ou capacité nécessaire à un utilisateur pour résoudre un problème ou atteindre un objectif. (Davis, 1993; IEEE 830-1983)
3. Les exigences sont toujours dérivées de faits existants qui nécessitent d'être organisés en ordre pour décrire le système. (Robillard et al., 2002)

Dans ces définitions, nous avons retenu pour la méthode que :

- **les exigences doivent être reliées au problème (non à la solution),**
- **la notion d'ordre dans les exigences existe.**

4.3. CUEILLETTE DES EXIGENCES

Dans un petit projet de programmation, plus précisément dans le cadre d'un cours, les exigences se trouvent souvent dans des énoncés de travaux pratiques, donc il n'est pas nécessaire de développer sur les entrevues, les scénarios, les prototypes, etc. Par contre, nous y ferons référence dans la section « mot sur les grands projets » de la méthode.

Pour que notre méthode soit efficace, il faudra que les enseignants s'assurent que les informations essentielles, à la réalisation du projet, se trouvent dans les énoncés de travaux pratiques. C'est indispensable pour que les étudiants puissent suivre la démarche que nous proposons.

4.4. ANALYSE DES EXIGENCES

Les étudiants devront être en mesure de retrouver les exigences à partir de l'énoncé de travail pratique.

L'analyse des exigences sert à :

- détecter les contradictions entre les exigences,
- découvrir les limites du système et comment il doit interagir avec son environnement,

- transformer les exigences système en exigences logiciel (Sawyer et al., 2001, chap. 3.3).

Pour y parvenir, l'analyse des exigences se fait en plusieurs étapes : la classification, la modélisation, la conception architecturale et la négociation.

Nous ne conserverons que la classification pour la méthode puisque les autres étapes nécessitent plus d'interactions entre plusieurs intervenants, ce qui n'arrive pas dans les cours de programmation de base sinon avec l'enseignant.

4.5. CATÉGORIES D'EXIGENCES POUR LA CLASSIFICATION

Voici différentes façons de classer les exigences (Sawyer et al., 2001, chap. 3.3.1) :

- fonctionnelles ou non-fonctionnelles.
- par processus.
- par buts.
- par contraintes.
- volatiles ou stables.
- autres (selon le contexte).

Nous avons choisi de n'utiliser que les exigences fonctionnelles et non-fonctionnelles puisqu'il s'agit de la distinction la plus fondamentale (Sawyer et al., 2001, chap.2.1; Kotonya et Sommerville, 2000; Somerville, 1997).

Voici la définition des ces deux catégories d'exigences :

- fonctionnelle : Exigences en terme de capacité. Le programme doit être en mesure de faire quelque chose. Par exemple : formater du texte, moduler un signal, etc.
- non-fonctionnelle : Exigences qui contraignent le programme. Par exemple : des contraintes de qualité, de conception, de performance, de sécurité, etc.

Dériver les exigences non-fonctionnelles peut s'avérer difficile particulièrement parce que cette catégorie d'exigences n'est pas couverte par la plupart des méthodes d'ingénierie des exigences (Robillard et al., 2002). **C'est pour cela que nous tenons à ce que les étudiants commencent à les distinguer dès le début.**

4.6. DOCUMENTATION À PRODUIRE

Dans notre méthode, la description des exigences sera présentée dans un document que nous appellerons « Spécification des exigences ». Ce document est inspiré du *Software Requirements Specifications* (IEEE, 830-1998), mais il n'a pas la prétention d'être aussi complet. Le SRS décrit dans un langage naturel la liste des exigences et peut être appuyé par des outils qui le supportent (diagramme d'états, diagramme de flux de données, diagramme de flux d'opérations, etc.). L'utilisation de ces outils dépend de la taille et de la complexité du projet donc nous n'en aurons nullement besoin dans notre méthode.

Dans un document comme le *Software Requirements Specifications* (IEEE, 830-1998), il faut habituellement retrouver :

- la description des fonctionnalités (ce que le logiciel doit faire),
- les interactions avec les gens, les autres systèmes, etc.
- les contraintes de performance (vitesse, temps de réponse, etc.),
- les attributs : considérations de portabilité, de sécurité, etc.,
- les contraintes de conception (langage de programmation, système d'exploitation, base de données, etc.)[5].

Nous décrirons dans notre méthode les fonctionnalités (exigences fonctionnelles), les attributs (exigences non-fonctionnelles) par ordre d'importance et nous ajouterons deux sous-catégories adaptées aux cours visés soit : les contraintes particulières concernant le projet qui ne sont pas des exigences logiciel (par exemple une contrainte de l'enseignant) et les validations à effectuer.

Les exigences doivent être :

- claires et non-ambigües,
- vérifiables,
- complètes,
- consistantes,
- clairement identifiées (numéro unique),
- modifiables,
- retraçables.

Dans la méthode, nous voulons mentionner toutes ces caractéristiques.

---

[5] Ces contraintes peuvent être mises dans le document de conception.

Les exigences servent essentiellement à :

- exprimer les besoins et les contraintes du programme d'un point de vue externe au programme (utilisateur),
- découvrir comment le système sera partitionné, en identifiant quelles exigences seront allouées à quels composants,
- aider à établir une base de communication entre les utilisateurs et les concepteurs du système.

Nous n'écrirons rien sur la partition du système mais les étudiants auront à concevoir un diagramme hiérarchique de la solution qu'ils proposent lors de l'étape de la conception architecturale (Voir p. 31). Cette étape est habituellement difficile pour les étudiants. C'est pourquoi nous avons ajouté dans la méthode une vue des dépendances. Cela est habituellement réalisé lors de l'étape de traçabilité des exigences à l'aide d'un graphe complexe (Sawyer et al,, 2001, chap. 3.6.3). **Nous présentons une version simplifiée de ces dépendances et nous croyons que cela aidera à effectuer l'étape de conception architecturale plus facilement.**

Dans le domaine du génie logiciel, il existe une panoplie de normes pour établir la structure de la documentation sur les exigences (Sawyer et al., 2001, chap. 3.4.3). Nous n'avons utilisé aucune norme intégralement puisque celles-ci sont habituellement dédiées à de grands projets et les documents associés sont longs à produire. Nous avons tout de même gardé l'esprit du *Software Requirements Specifications* (IEEE, 830-1998) cité dans le guide au corpus de connaissances (Abran et al., 2001). Nous avons donc créé notre propre document intitulé « Spécification des exigences » qui permet de faire quand même la transition entre les exigences système et les exigences logiciel.

Dans le cadre d'un cours de base en programmation, les exigences sont souvent fournies par un énoncé de travail pratique. Ce qu'il est important de faire ressortir dans notre méthode est que la spécification des exigences découlera de cet énoncé.

Nous avons également vu différentes catégories d'exigences, entre autres qu'une exigence a un objectif et qu'elle se doit d'être identifiée et classifiée par ordre d'importance. L'artéfact que nous produisons pour l'étape des exigences se devait au moins de mettre en valeur ces points. Nous croyons important de spécifier dans la section « un mot sur les grands projets » qu'il n'y a pas qu'une catégorie d'exigences et qu'il y a plusieurs façons d'en faire la cueillette.

À l'annexe C, page 76, se trouve la description de l'étape des exigences telle qu'elle sera présentée aux étudiants.

CHAPITRE V

CONCEPTION

L'étape de conception d'un logiciel est « Le processus de définition de l'architecture, des composants[6], des interfaces et autres caractéristiques d'un système ou d'un composant et le résultat de ce processus » (IEEE 610.12-1990).

L'étape de conception peut s'avérer difficile à effectuer correctement pour les étudiants, à cause de leur manque d'expérience. Nous voulons dans notre méthode simplifier la tâche des étudiants tout en leur permettant de développer des habiletés qui leur serviront lors du développement de plus grands projets. Comme pour les autres chapitres, nous décrirons les modèles de conception employés en génie logiciel selon le guide au corpus de connaissances (Abran et al., 2001), nous retiendrons ceux qui sont applicables pour de petits projets et nous les adapterons pour notre méthode.

NOTE : Nous présentons chaque thème concernant la conception et nous mettons en caractères gras ce que nous retenons pour la méthode.

5.1. CONCEPTS DE BASE

Le résultat du processus de conception est un ensemble de modèles et d'artéfacts qui décrivent les décisions majeures qui auront été prises (Tremblay, 2001, chap. 3:3; Budgen,1994; IEEE 1016-1998; Liskov et Guttag, 2001; Pressman, 1997). Dans le domaine du génie logiciel plusieurs notions clés sont considérées comme fondamentales pour l'étape de conception telles que : l'abstraction, le couplage et la cohésion, la décomposition et la modularité, l'encapsulation (représentation cachée), la séparation de l'interface et de l'implantation ainsi que la complétude du système (Tremblay et al., 2001, chap. 3.3). Il existe également un certain nombre d'éléments qui doivent être considérés dans la conception du logiciel, entre autres : le contrôle et la manipulation des flux de données, l'approche choisie pour interagir avec les utilisateurs des composants, la gestion d'exceptions et la tolérance aux fautes, la durée de vie des données (persistance), la concurrence des composants et la distribution du logiciel (Tremblay, 2001, chap. 3.3 :II).

---

[6] Les composants sont des entités de conception comme des procédures, des fonctions et des modules.

Une fois les exigences comprises, organisées, formalisées et exprimées, l'étape suivante est d'utiliser ces informations pour l'analyse et la conception (Robillard et al., 2002 ; chap. 5 p. 104). La pratique recommandée pour décrire l'étape de conception présentée dans la norme IEEE 1016 est le *Software Design Description*. C'est un document qui montre comment le système (peu importe la taille ou la complexité) doit être structuré pour satisfaire aux exigences identifiées dans le *Software Requirement Specification* (IEEE 830-1998). C'est une traduction des exigences en une description des structures, des composants, des interfaces et des données requises pour l'étape de construction. (IEEE 1016-1998).

Dans la méthode, nous voulons insister sur le fait que l'étape de conception doit permettre de satisfaire aux exigences soulevées lors de l'étape de spécification des exigences et permettre de passer à l'étape de construction. Nous désirons aussi que la méthode permette de décrire la décomposition et la modularité, l'encapsulation (représentation cachée), la séparation de l'interface et de son implantation, le contrôle et la manipulation des flux de données et la gestion d'exception. Par contre, la description de la complétude du système, de la tolérance aux fautes, de la durée de vie des données (persistance), de la concurrence des composants, de la distribution du logiciel, du couplage et de la cohésion ne fera pas partie de la méthode.

Étant donné que certaines notions théoriques nécessaires pour l'étape de conception ne seront pas acquises lors du premier cours de programmation, nous avons séparé la description des modules de celle des composants pour décrire la conception architecturale. Conséquemment un document décrira la décomposition en composants (excluant les modules) et un autre document décrira la composition des modules. Ainsi il sera possible ainsi de n'utiliser que la décomposition des composants pour le premier cours et d'utiliser les deux documents pour le deuxième cours.

Dans les paragraphes suivants se trouvent les descriptions des thèmes provenant du corpus de connaissances (Abran, et al., 2001) pour le processus de conception. Les différents thèmes sont : les points de vue, les styles architecturaux, les patrons de conception et les familles de logiciels, l'analyse et l'évaluation de la conception, les notations et les stratégies et méthodes. La plupart sont assez complexes et c'est pour cela qu'ils seront ignorés ou adaptés pour la méthode. Nous concentrons nos efforts sur les notations, les stratégies et les méthodes, sans trop insister sur les points de vue, les styles architecturaux, etc. bien que nous les décrivions sommairement.

5.2. POINTS DE VUE

Habituellement les systèmes doivent être décrits et documentés par différentes vues. Une vue représente « un aspect partiel de l'architecture du logiciel qui montre des propriétés spécifiques du logiciel » (Buschman et al., 1996; p.562). Les différentes vues sont : la vue des spécifications *(logical view)* qui décrit la satisfaction des exigences fonctionnelles, la vue du processus (*process view*) qui décrit la concurrence, la vue physique (*physical view*) qui décrit la distribution et la vue du développement (*development view*) qui décrit le découpage en unité d'implantation (Tremblay, 2001; chap. 3; III).

La concurrence et la distribution du logiciel ne s'appliquent pas dans la mesure où les logiciels développés dans un premier cours de programmation sont simples et ne comportent pas plusieurs sous-systèmes. Dans la méthode nous ne considérerons que la vue des spécifications, pour laquelle nous décrirons les exigences fonctionnelles à l'aide d'un diagramme hiérarchique et la vue du développement.

5.3. STYLES ARCHITECTURAUX

Il existe différents styles architecturaux qui sont « un ensemble de contraintes sur une architecture qui définit un ensemble ou une famille d'architectures qui les satisfont (les contraintes).» (Tremblay, 2001; chap. 3; III, Bass et al., 1998, chap. 2). Les différents styles peuvent être organisés de la façon suivante : structure générale (ex : tuyau et filtre), systèmes distribués (ex : client-serveur), systèmes interactifs (ex : modèle vue contrôleur), système adaptables (ex : micro-kernel) et autres styles (ex : interpréteur, lot) (Tremblay, 2001; chap. 3; III).

Le style que nous utiliserons dans la méthode est celui qui est associé au paradigme procédural et que certains appellent le style *call-return*. Les autres styles plus complexes cités dans le guide au corpus de connaissances (Abran et al., 2001) **ne sont pas nécessaires pour un cours d'introduction à la programmation. Nous ne ferons aucune mention dans la méthode de ces styles architecturaux.**

5.4. PATRONS DE CONCEPTION (*DESIGN PATTERNS*) ET FAMILLES DE PROGRAMME

Un patron est « une solution commune à un problème commun dans un contexte donné. Les styles architecturaux peuvent être vus comme un patron décrivant l'organisation de haut niveau du logiciel ». (Booch, et al., 1999, p.447). Les patrons de conception sont utilisés pour décrire les détails d'un plus bas niveau (plus local) de l'architecture. (Tremblay, 2001; chap. 3 ; III).

Il est possible aussi de catégoriser les logiciels dans des familles de produits pour permettre la réutilisation de composants lors de développement d'autres logiciels de même famille.

Il ne sera pas fait mention des familles de logiciels et des patrons de conception dans la méthode. Les patrons devraient faire partie de l'enseignement en conception mais ils ne sont pas suffisants en soi (Robillard et al., 2002). **Alors même si certains patrons pouvaient être utilisés dans le cadre des cours visés par la méthode (itérateur, *template* ou autres), nous considérons qu'il n'est pas nécessaire de le mentionner explicitement car il faudrait montrer d'autres concepts et cela occasionnerait une surcharge de matière à acquérir dans ces cours.**

5.5. ANALYSE DE QUALITÉ ET ÉVALUATION DE LA CONCEPTION

Il est possible de vérifier la qualité de la conception par des techniques et des outils prévus à cet effet. La vérification des attributs, la révision d'artéfacts et l'analyse statique de la conception servent, entre autres, à vérifier la capacité fonctionnelle (aptitude, exactitude, etc.), la fiabilité, la maintenabilité, la portabilité, la traçabilité et la robustesse d'un logiciel. Elles servent aussi à mettre en application la révision architecturale, la révision et l'inspection de la conception (couplage et cohésion), les techniques basées sur des scénarios et la vérification de la traçabilité des exigences. Finalement, elles permettent d'évaluer la conception du point de vue de la performance ou de la faisabilité (Tremblay, 2001, chap. 3; IV).

Nous croyons qu'il est très important de nous préoccuper de la qualité et cela à tous les niveaux. Nous voulons préparer les étudiants à s'en soucier dès le début. Cependant, pour ne pas surcharger la matière offerte dans les cours visés et pour respecter ce que nous avons dit en 5.1 (l'étape de conception doit permettre de satisfaire aux exigences soulevées lors de l'étape de spécification des exigences), **nous inclurons dans la méthode la vérification de la traçabilité des exigences. Cela devrait permettre de vérifier à nouveau la spécification des exigences avant de passer à l'étape de construction. Même s'il peut être intéressant d'initier les étudiants aux concepts de couplage et de cohésion[7] pour mesurer leur conception architecturale, cela nous semble prématuré. Il en est de même pour les considérations de portabilité, de maintenabilité, etc. Pour ce qui est des autres concepts concernant la qualité, nous croyons qu'ils devront être enseignés ultérieurement dans un cours sur la vérification et l'assurance qualité de logiciels.**

[7] Il y a une bonne description de ces mesures (couplage et cohésion) dans les livres de Yourdon et Constantine (Yourdon et al., 1979) et de Meilir Page-Jones (Page-Jones, 1988).

5.6. NOTATIONS

Il existe plusieurs façons de représenter les artéfacts pour l'étape de conception. La plupart sont graphiques et permettent de représenter les différents composants et leurs relations. Selon le *standard for information technology-software life cycle processes* (IEEE, 12207-1996), qui décrit le processus du cycle de vie logiciel, l'étape de conception se divise en deux activités : conception architecturale[8] et conception détaillée. La première décompose et décrit les composants du logiciel et la seconde décrit chacun de ces composants suffisamment pour permettre la traduction en code (Tremblay, 2001 ; chap. 3). **Nous décrirons la conception architecturale et la conception détaillée dans la méthode.**

Il existe deux catégories de notations à considérer, celles qui décrivent la structure statique et celles qui décrivent le comportement dynamique du logiciel. Tout cela a pour but commun de définir le squelette du logiciel (Robillard et al., 2002).

Voici une liste des différentes notations reconnues pour décrire la structure statique (Tremblay, 2001 ; chap. 3 ; V) :

- ADL (*Architecture Description Languages*) : décrit l'architecture en terme de composants et de connecteurs,
- diagramme de classes et d'objets : montre les classes et leurs relations,
- diagramme de composants : montre les composants physiques et leurs relations,
- cartes CRC : décrit le nom, la responsabilité et la collaboration des composants,
- diagramme de déploiement : modèle l'aspect physique du système,
- diagramme entité-relation : définit le modèle conceptuel des données,
- IDL (*Interface Description Langages*) : définit les interfaces des composants,
- JSD (*Jackson Structure Diagrams*) : décrit les structures de données,
- diagramme structuré (*Structure Charts*) : décrit la structure d'appel des sous-programmes.

Nous avons choisi le diagramme hiérarchique qui fait partie de la catégorie des diagrammes structurés. Ce diagramme est d'usage courant dans les cours d'introduction à la programmation et il permet de décrire la hiérarchie, la modularité, le couplage et la cohésion pour les composants (Yourdon et al., 1979). Nous voulons nous assurer que le diagramme sera construit à partir du document « Spécification des exigences » car ensuite

[8] Certains auteurs, tels P. Robillard (Robillard et al., 2002) et M. Pages-Jones (Pages-Jones, 1988), appellent cette étape la conception préliminaire (*preliminary design*).

nous ferons référence aux numéros des exigences pour chaque composant. Comme nous l'avons indiqué précédemment, il ne sera pas fait mention explicitement de couplage et de cohésion. Nous présenterons tout de même quelques pratiques reconnues qui permettront de réduire le couplage et d'augmenter la cohésion.

Nous voulons également décrire une interface[9] pour les composants. Une interface sert à décrire le nom, le comportement d'un composant et à en définir les entrées et les sorties pour l'utilisateur de ce composant. Cela permet de faire une séparation claire entre la vue externe et la vue interne du composant (Booch et al., 1999). Nous croyons que d'introduire ce concept dès le premier cours facilitera la compréhension de la séparation de l'interface et de l'implantation (voir 5.1) lors du deuxième cours de programmation.

Voici une liste des différentes notations reconnues pour le comportement dynamique (Tremblay, 2001) :

- diagramme d'activité : montre le flux des contrôles des activités,
- diagramme de collaboration : montre les interactions entre des groupes d'objets,
- diagramme de flux de données : montre le flux des données entre les composants,
- table de décisions : représente des combinaisons complexes de conditions et d'actions,
- diagramme de flux de contrôles : représente les flux de contrôles et les actions à effectuer,
- langages de spécification formelle : langages rigoureux à base mathématique qui définissent le comportement des composants,
- pseudo-code et PDL (*Program Design Language)* : décrit en détail, sous forme de pseudo langage informatique, le comportement des composants,
- diagramme de séquence : montre les interactions entre groupes d'objets ordonnés par le temps d'ordonnancement des événements,
- diagramme d'état : montre le flux de contrôles d'état en état.

Nous avons choisi le pseudo-langage structuré pour la conception détaillée. C'est un langage structuré (en français pour nous) qui comprend dans son vocabulaire des itérations, des sélections et des phrases simples (Pages-Jones, 1988; chap. 4 ; p. 52). **Le pseudo-langage est similaire au pseudo-code, qui lui est un langage de programmeurs et de concepteurs. Tandis que le pseudo-langage est celui des utilisateurs et des analystes**

[9] Nous n'utiliserons pas la notation IDL pour la description des interfaces puisqu'elle exige la connaissance de UML (Booch, et al., 1999). Nous utiliserons plutôt la notation de M. Page-Jones (Page-Jones, 1988) qui est plus textuelle.

(Pages-Jones, 1988; chap. 4 ; p. 52). **Nous espérons que les étudiants seront en mesure de faire l'étape de conception avant l'étape de construction (et non l'inverse) en distinguant clairement la différence entre un langage d'analyse et un langage de programmation.**

5.7. STRATÉGIES ET MÉTHODES DE CONCEPTION

Les stratégies de conception décrites dans le guide au corpus de connaissances (Tremblay, 2001 ; chap. 3 ; VI) sont les suivantes : générales, orientées-fonction, orientées-objet, orientées-donnée et autres. Pour chacune de ces stratégies, il y a plusieurs méthodes qui sont plus spécifiques et qui ne seront pas décrites ici.

Nous avons choisi la stratégie orientée-fonction qui est une des plus classiques et dont les méthodes associées sont celles que nous avons décrites dans les premiers chapitres soit : méthode diviser pour régner, raffinement successif, approche descendante et ascendante, analyse et conception structurée.

5.8. CONSTRUCTION DE L'ARTÉFACT

Même si nous n'envisageons que l'approche procédurale, nous utiliserons des outils de conception qui tiendront compte de la conception pour la programmation utilisant des langages basés-objets par opposition à des langages orientés-objets. Les langages basés-objets permettent le découpage en modules et l'encapsulation des données, mais ne permettent pas l'héritage et le polymorphisme dynamique.

Dans les paragraphes suivants, nous décrirons comment nous comptons présenter l'étape de conception à l'aide de l'artéfact que les étudiants doivent produire. Nous donnerons les détails sur le contenu de chaque partie de l'artéfact qui respectera les décisions prises que nous avons décrites précédemment dans ce chapitre.

5.8.1. Conception architecturale : description des diagrammes hiérarchiques

Le diagramme hiérarchique est connu et fortement employé dans les cours de programmation. La modification que nous voulons proposer à la conception architecturale pour la méthode est que les étudiants fassent deux diagrammes hiérarchiques. Un premier diagramme que nous appellerons « diagramme des exigences »[10] conçu à partir de la liste des dépendances (Voir

[10] Le « diagramme des exigences » ne fait pas partie de l'étape de conception décrite dans la littérature mais nous prétendons que cela permettra de faire plus facilement la transition de l'étape d'exigences à l'étape de conception vers l'étape de construction.

chapitre IV). Nous voulons que les étudiants l'utilisent comme base pour continuer le découpage. Dès ce moment, les étudiants devraient voir la notion de hiérarchie et les liens entre les composants qu'ils auront à développer. De plus nous espérons que lorsqu'il y aura plusieurs références à la même exigence, cela leur donnera l'intuition de réutiliser.

Il est fréquent dans le genre de cours visés, à cause du manque d'expérience des étudiants, que les étudiants fassent toute l'étape de codage avant de produire le diagramme hiérarchique. Ils imaginent que ce diagramme doit seulement décrire l'organisation finale des composants, donc ils peuvent attendre à la fin pour le produire. Pour notre méthode, nous désirons que les étudiants se servent de la production du diagramme hiérarchique pour trouver les composants et les liens qui les unissent. C'est pourquoi nous voulons leur faire produire un autre diagramme que nous appellerons « diagramme de la solution » qui reflètera la solution finale du découpage. Nous espérons ainsi que les étudiants réalisent qu'il y a une démarche entre la spécification des exigences et la version finale d'un diagramme hiérarchique et qu'il y aura des décisions intermédiaires à prendre.

Nous présenterons aux étudiants la notation décrite dans *The practical guide to structured system designed* (Page-Jones, 1988) pour la production des diagrammes hiérarchiques. Cela se fait à l'aide de rectangles reliés par des flèches où chaque rectangle est un composant ayant un nom décrivant son comportement. Les flèches décrivent les appels entre composants dans le style *call-return* (voir 5.3). Il est possible de mettre des détails tels que les itérations et les sélections mais nous n'en utiliserons pas. Par contre, nous illustrerons les flux de données.

5.8.2. Conception architecturale : description des modules

Pour les étudiants d'un deuxième cours de programmation, il faudra décrire les modules utilisés et la composition de ces modules. Habituellement ces regroupements en modules sont visibles dans le diagramme hiérarchique (Yourdon et al., 1979 ; chap. 14). Nous ferons un document à part qui illustrera les composants faisant partie des modules.

5.8.3. Conception détaillée : description des interfaces et des algorithmes

Les interfaces (numéro de référence aux exigences inclus) doivent décrire le comportement des composants (sans utiliser de pseudo-code) et la stratégie détaillée (ou les algorithmes utilisés) sera décrite pour chaque composant. Nous croyons qu'en introduisant la notion d'interfaces dès le premier cours, cela aidera à la compréhension de la séparation de l'interface et de l'implantation pour le deuxième cours. Pour décrire les entrées/sorties, nous n'avons pas pris de modèles de flux de données conventionnel puisqu'ils sont beaucoup trop longs à produire

pour le peu de données à traiter dans les cours qui nous intéressent. Les entrées et les sorties nécessaires à chaque composant seront décrites en texte avec leur type, leur provenance (clavier, fichier, ...) et leur destination (écran, fichier, ...). Cette façon de faire a déjà été employée à l'Université du Québec à Montréal, dans certains cours d'introduction à la programmation, et elle nous semble suffisante. Nous ajouterons à cela les messages d'exception pour décrire les erreurs possibles lors de l'exécution du logiciel.

Un problème connu est que les étudiants attendent d'avoir fini le code pour le traduire en pseudo-langage (ou en pseudo-code) alors que ce devrait être l'inverse. Les stratégies employées pour résoudre un problème font partie de la conception détaillée. C'est donc ici que nous ferons la transition entre les exigences et la solution. Pour chacune des interfaces, nous décrirons les algorithmes détaillés en pseudo-langage qui seront traduits en code par la suite dans un langage de programmation. De toute évidence, pour réaliser cette section, les étudiants devront comprendre minimalement ce qu'est une saisie (clavier ou fichier), une écriture (écran ou fichier), une sélection (si), une itération (tant que), une affectation (une flèche de droite vers la gauche ou := ou =), les opérateurs arithmétiques de base possibles (+, -, *, /, modulo) et les opérateurs relationnels (<, >, <=, >=, =, ≠).

Il faut éviter que le pseudo-langage ressemble trop au code d'autant plus qu'il doit être absolument indépendant du langage de programmation utilisé. Nous décrirons à l'aide d'exemples dans la méthode comment décrire les algorithmes en y introduisant les itérations, les sélections, les affectations et les opérations qui rendront la traduction des algorithmes possible sans avoir déjà écrit tout le code.

Il y a moins de points de vue à considérer lors de l'étape de conception dans les cours de programmation de base que dans de grands projets. Les objectifs à atteindre sont pour la plupart reliés à la description des composants, de l'architecture, des interfaces et des liens entre les spécifications et le code (Tremblay, 2001, chap. 3.3:III). Nous avons quand même atteint ces objectifs en utilisant des méthodes et des techniques déjà éprouvées. Nous avons utilisé le diagramme hiérarchique classique mais nous en avons prévu deux soit le « diagramme des exigences » et « le diagramme de la solution ». Nous avons prévu un document pour les interfaces (description comportementale, entrées/sorties, exception) et les algorithmes en pseudo-langage et un document décrivant la modularité pour les concepts de programmation un peu plus avancés.

Nous voulons conserver les outils utilisés habituellement pour la conception. Nous avons changé l'ordre de présentation que nous avons adapté. Par exemple, nous avons décrit des interfaces dans les premiers cours en espérant que cela facilite l'apprentissage de l'encapsulation (des données) montrée dans le deuxième cours. Nous faisons concevoir également deux diagrammes hiérarchiques plutôt qu'un seul. Notre objectif, à part celui d'obtenir une bonne conception, est que cette phase du cycle de vie soit réalisée avant de commencer l'étape de construction et que certains concepts abstraits de la programmation (découpage, encapsulation) s'assimilent plus aisément.

À l'annexe C, page 92, se trouve la description de l'étape de conception telle qu'elle sera présentée aux étudiants.

CHAPITRE VI

CONSTRUCTION

Le contenu de ce chapitre ne s'éloigne pas de ce qui est enseigné habituellement dans les cours de programmation visés par notre méthode. C'est probablement même la matière la plus susceptible d'être présentée. Les techniques qui y sont enseignées sont connues et généralement uniformes dans les universités. Ce sont souvent les raisons qui motivent leur mise en application (incluant les bonnes pratiques de codage) qui ne sont pas suffisamment retenues par les étudiants. Nous avons fait un effort pour améliorer, par la pratique, la rétention des principes à retenir. Ce chapitre décrira les quatre thèmes du corpus de connaissances (Abran et al., 2001) concernant le processus de construction, les techniques employées et ce que nous utiliserons pour la méthode.

NOTE : Nous présentons chaque thème concernant la construction et nous mettons en caractères gras ce que nous retenons pour la méthode.

6.1. PHASE CONSTRUCTION

La construction se fait par la combinaison du codage, de la validation et des tests unitaires[11] faits par un (des) programmeur(s) (Bollinger et al., 2001; chap. 4; 2). C'est comme la mise en chantier du plan fait lors de la conception. Il faut maintenant concrétiser une solution aux différents problèmes reliés à la satisfaction des exigences. L'étape de construction se situe entre celle de conception et celle de test pour le modèle en cascades. La conception est l'entrée de la construction et la construction est l'entrée des tests (Bollinger et al., 2001; chap. 4; 2). **De là l'importance dans la méthode de se servir des artéfacts produits lors de l'étape de conception et de les maintenir à jour lorsqu'il y a des modifications apportées pendant l'étape de construction**.

Les objectifs qu'il faut viser lors de la construction sont les suivants : améliorer la productivité et la qualité du système, évaluer le logiciel, sélectionner les standards de développement du logiciel, choisir les langages de programmation, de configuration et les techniques de construction (manuelles ou automatisées) (Bollinger et al., 2001). Tout cela peut se faire en utilisant des outils existants ou en produisant soi-même ses propres outils. Les principes qui affectent le plus la construction sont : réduire la complexité du système, anticiper le changement, structurer en fonction de la validation et utiliser des normes externes pour permettre de partager des données avec le monde extérieur (Bollinger et al., 2001; chap. 4).

---

[11] La description de ces tests est faite dans le chapitre VII.

Une fois de plus, tous ces objectifs ne peuvent pas être atteints dans les premiers cours de programmation. D'une part, les langages de programmation et les standards de développement sont habituellement choisis (plate-forme, BD, ...). D'autre part, il y a rarement des soucis de productivité. **Nous n'envisagerons la construction qu'avec des objectifs de réduction de la complexité, d'anticipation au changement et de structuration pour la validation. Nous laisserons de côté tout ce qui regarde les choix à faire sur les logiciels, les plates-formes, les compilateurs (et autres) et l'utilisation de normes externes.**

6.2. STYLES DE CONSTRUCTION

Il existe trois styles de construction : Le style linguistique, le style formel et le style visuel (Bollinger et al., 2001, chap. 4; 3.2). Le premier repose sur l'utilisation d'un langage naturel ou d'un pseudo-code pour décrire la construction, le deuxième sur l'utilisation d'un langage formel et le dernier sur l'utilisation d'images et d'objets visuels. **Nous utiliserons le style linguistique lors de l'étape de construction puisqu'il utilise un langage que les étudiants connaissent déjà. Nous croyons que le style formel et le style visuel nécessitent l'apprentissage de connaissances (langage formel, outil de création d'interfaces utilisateur) qui surchargerait le contenu des cours visés par notre méthode.**

6.3. PRINCIPES DE CONSTRUCTION

Dans ce chapitre nous présentons les principes que nous voulons promouvoir dans la méthode soit : réduire la complexité du système, anticiper le changement et structurer en fonction de la validation. Nous présenterons aussi les techniques utilisées pour y parvenir et les raisons qui en motivent l'utilisation. Nous retiendrons pour la méthode les considérations que doivent avoir en tête les étudiants pendant toute l'étape de construction. Nous sommes conscients que la description de l'étape de construction dans la méthode chevauchera la théorie enseignée dans les cours. Nous ne voulons pas que cette section serve de notes de cours pas plus que nous ne voulons qu'elle contraigne la liberté pédagogique des enseignants. C'est pourquoi dans la méthode nous ne ferons qu'énumérer ces techniques en insistant sur les objectifs qu'elles doivent atteindre sans en expliquer les concepts.

6.3.1. Réduire la complexité

La raison la plus importante pour créer des composants est de réduire la complexité du logiciel (McConnell, 1993). Il existe trois techniques pour réduire la complexité d'un système (Bollinger et al., 2001, chap. 4; 3.1).

1. Éliminer ce qui n'est pas nécessaire au système. Cela revient aussi à dire de ne pas ajouter au logiciel des fonctionnalités qui ne sont pas exigées.
2. Automatiser les tâches complexes du système, soit en utilisant des outils existants, soit en écrivant nos propres outils. Les tâches dont il est question sont au niveau du développement du logiciel et non au niveau des fonctionnalités du logiciel à produire.
3. Isoler un problème complexe et le découper en petits composants regroupés en modules, plus facilement compréhensibles. Le découpage en composants n'est pas suffisant, il est aussi important de les localiser au même endroit (modules).

Il ne sera pas fait mention dans la méthode de l'automatisation des tâches complexes du développement, nous glisserons un mot pour avertir de ne rien ajouter qui n'est pas explicitement demandé et nous mentionnerons que la localisation des composants en module est une façon de réduire la complexité.

Les techniques employées en utilisant la méthode linguistique pour réduire la complexité sont les suivantes (Bollinger et al., 2001; chap. 4 ; 3.3.1.1) :

- les patrons de conception (*design patterns*),
- les composants (procédure, fonction, module générique ou non),
- l'encapsulation,
- les types de données abstraits, les bibliothèques de composants (et *framework*), les fichiers et les bibliothèques,
- les inspections formelles,
- les objets et l'utilisation de langages spécifiques au domaine.

Étant donné le niveau des cours visés, il ne sera pas fait mention d'objets, d'inspections formelles, de bibliothèques, de patrons de conception et de langages spécifiques au domaine. Par contre, nous prévoyons citer les autres techniques dans la méthode (découpage en composants, encapsulation et TDA).

6.3.2. Anticiper le changement

Il faut prévoir les changements possibles durant la vie d'un logiciel et les faciliter. Il existe trois techniques principales pour assouplir la construction d'un système et en faciliter le changement (Bollinger et al., 2001; chap. 4).

1. Généraliser plutôt qu'être spécifique. Il faut abstraire le plus possible.
2. Collecter le plus de données liées à l'application pour permettre de généraliser.
3. Isoler ce qui est appelé à changer pour avoir le moins d'endroit à modifier lors d'un changement.

Les techniques employées en utilisant la méthode linguistique pour anticiper le changement sont les suivantes (Bollinger et al., 2001 ; chap. 4 ; 3.3.2.1) :

- utiliser l'abstraction et l'encapsulation (information cachée),
- faire une documentation solidement commentée,
- avoir un ensemble de méthodes complètes et suffisantes,
- utiliser une méthode objet,
- utiliser des fichiers de configuration,
- favoriser la réutilisation,
- faire des logiciels auto-descriptifs (*plug and play*).

Nous mentionnons dans la méthode qu'il faut généraliser le plus possible et isoler les composants qui sont appelés à changer. Nous citerons l'abstraction, l'encapsulation, la réutilisation et la documentation commentée comme étant des méthodes pour parvenir à anticiper le changement. Les autres techniques étant associées à la programmation objet ou à de grands projets elles ne seront pas citées sinon dans la section « mot sur les grands projets ».

6.3.3. Structurer pour la validation

Les techniques employées en utilisant le style linguistique pour structurer en fonction de la validation sont les suivantes (Bollinger et al., 2001 ; chap. 4 ; 3.3.3.1):

- modularité,
- programmation structurée,
- raffinement successif.

Le but que ces techniques veulent atteindre est de contraindre l'interprétation du code source (par les utilisateurs ou l'ordinateur) pour ne pas perdre la structure et la précision et en assurer

la consistance (Bollinger et al., 2001, chap. 4). **Nous mentionnerons toutes ces techniques dans la méthode.**

6.3.4. Créer des composants

Le concept premier qui relie tous les principes et les techniques énumérés précédemment est la création de composants. Habituellement la théorie concernant les bonnes pratiques de création des composants est donnée dans les cours de programmation. Les paragraphes précédents mentionnent les raisons de créer des composants mais **nous désirons aussi donner certaines caractéristiques de qualité d'un composant (directement ou indirectement). Par exemple, nous ne mentionnerons rien concernant le couplage et la cohésion directement mais nous leur rappellerons qu'une procédure ou une fonction doit faire seulement la tâche que son nom indique (cohésion) aussi qu'il est préférable de passer en paramètre seulement les champs nécessaires à une fonction plutôt que de passer tout un enregistrement (couplage). Nous désirons aussi rappeler les bonnes pratiques de codage. La plupart des bonnes pratiques que nous décrirons dans la méthode (voir annexe F) pour la création de composants (commenter le code, donner des noms significatifs, etc.) proviennent du livre *Code Complete* de Steve McConnell (McConnell, 1993) et elles sont généralement reconnues.**

La construction du code se situe entre la conception et les tests. Nous avons choisi le style linguistique pour la méthode parce qu'il est le plus approprié pour les cours de programmation de base, dans le paradigme procédural. Nous avons mis de côté tout ce qui touche à l'utilisation de normes (*standard*) externes, au choix des langages de programmation, au choix des outils qui améliorent la productivité, au choix des plates-formes et au paradigme objet. Tout cela se retrouvera dans la section « mot sur les grands projets » de la méthode.

Nous avons retenu ce qui s'applique dans les cours visés pour permettre de réduire la complexité, d'anticiper les changements et de structurer pour la validation. Notre méthode mentionne aussi les concepts de découpage en sous-programmes, de modularité, d'encapsulation, d'abstraction, de généricité, de type de données abstrait, de raffinement successif, de généralisation et de documentation de code. Ces concepts sont habituellement enseignés dans les cours visés par notre méthode, c'est pourquoi nous ne les décrivons pas mais nous les citons avec les objectifs qu'ils permettent d'atteindre. Finalement, nous donnons des caractéristiques de qualité et quelques bonnes pratiques de codage lors de la création de composants.

À l'annexe C, page 105, se trouve la description de l'étape de construction telle qu'elle sera présentée aux étudiants.

CHAPITRE VII

LES TESTS

Dans ce chapitre, nous décrirons en quoi consiste le processus de test selon le guide au corpus de connaissances (Abran et al., 2001). Nous ferons un bref résumé des différentes catégories de tests et nous prendrons ce qui, selon nous, est applicable et réalisable pour notre méthode et à quelle étape du développement ces tests doivent se faire.

NOTE : Nous présentons chaque thème concernant la construction et nous mettons en caractères gras ce que nous retenons pour la méthode.

7.1. INTRODUCTION

Il faut tester (ou vérifier) à toutes les étapes du processus de développement (Bertolino et al., 2001, chap. 5; 1). Nous avions le choix, pour la méthode, de présenter la planification des tests, et les tests, à l'intérieur des chapitres sur la conception et la construction OU de faire un chapitre spécifiquement sur les tests. Nous avons choisi cette dernière solution pour que les chapitres de la méthode reflètent les étapes du cycle de vie.

Même si nous utilisons le modèle en cascade, la majorité de ce qui sera décrit à l'intérieur de ce chapitre devra être fait pendant les autres étapes de développement que celle des tests. Les seuls tests qui seront effectivement faits à l'étape des tests seront les tests fonctionnels et les tests système, qui vérifient que la version exécutable du logiciel se comporte tel que désiré et qu'elle satisfait aux exigences fonctionnelles et non-fonctionnelles (voir chapitre sur les exigences).

7.2. CONCEPTS DE BASE ET DÉFINITIONS DES TESTS

Définition :

Tester un logiciel consiste à effectuer une vérification *dynamique* du comportement du programme sur un ensemble *fini* de cas tests, convenablement *sélectionnés* à partir du domaine d'exécution habituellement infini, relativement au comportement spécifié *désiré* (Bertolino et al., 2001, chap. 5; 2).

Les termes en italiques dans cette définition sont importants pour la direction que doivent prendre les tests.

- *Dynamique* : parce que les tests se font durant l'exécution du logiciel. Il faut donc faire des tests sur un produit fonctionnel.
- *Fini et sélectionné* : parce qu'il y a une infinité de tests possibles et qu'il est impossible de les faire tous. Il faudra donc en choisir.
- *Désiré* : parce qu'il faudra déterminer le comportement attendu.

C'est donc sur le choix des tests et sur les techniques permettant d'y parvenir que les décisions doivent se prendre, dans le but de satisfaire, selon le guide au corpus de connaissances (Bertolino et al., 2001, chap. 5; 2.1), des objectifs précis que voici :

- tester à toutes les étapes du processus de développement,
- exposer les erreurs,
- déterminer les tests cibles et les objectifs des tests à différentes étapes,
- assurer la qualité du logiciel,
- mesurer la fiabilité,
- évaluer l'utilisabilité,
- obtenir l'acceptation du client.

Dans la méthode nous planifions et exécutons les tests aux étapes de construction et de test dans le but d'exposer les erreurs et d'assurer une certaine qualité du logiciel produit. La mesure de fiabilité, l'évaluation de l'utilisabilité et l'acceptation du client sont peu applicables dans les cours visés mais ils seront mentionnés dans la section « mot sur les grands projets» de la méthode.

7.3. FONDEMENTS THÉORIQUES

Le *IEEE Standard for software unit testing* (IEEE 1008-1987) précise que le processus de test commence par la planification des tests. La littérature fournit un certain nombre de fondements théoriques pour la planification des tests que voici (Bertolino et al., 2001, chap. 5) :

- la sélection des critères de test : consiste à décider quels critères permettront de sélectionner les ensembles de tests que doit passer le logiciel (Bertolino et al., 2001; Pfleeger, 1998; Zhu et al., 1997; Weyuker, 1983; Weyuker et al., 1991),

- la sélection des modèles de test : peut être guidée par différents objectifs (Bertolino et al., 2001; Beizer, 1990; Perry, 1995; Frank et al., 1998),
- l'identification des défectuosités : il est préférable de penser à des tests pour trouver des erreurs plutôt qu'à des tests qui prouvent le bon fonctionnement du logiciel (Bertolino et al., 2001; Beizer, 1990; Kaner et al., 1999),
- la prédiction des résultats attendus pour l'acceptation ou le refus d'un test (*Oracle problem*) (Bertolino et al., 2001; Beizer, 1990; Weyuker, 1982),
- les limitations théoriques et pratiques des tests : vérifie la théorie sur laquelle sont basés les tests, pour établir un niveau de confiance des tests (Bertolino et al., 2001; Kaner et al., 1999),
- le problème des chemins impossibles (*infeasible paths*) : consiste à vérifier les chemins des flux de contrôle qui ne seront pas parcourus par des données (Beizer, 1990).

Il est évident que de sélectionner des critères de test, en considération de grands projets, ne peut pas être laissé aux étudiants. Cependant la méthode présentera une sélection de certains tests à effectuer en plus de spécifier à quelles étapes faire ces tests. Nous voulons laisser transparaître que la sélection des critères et des modèles de tests a déjà été faite, que les tests ont été pensés pour trouver des erreurs et que la prédiction des résultats attendus doit être faite. En d'autres termes, nous ferons une liste des considérations dont nous devons tenir compte pour mettre en place une bonne sélection des critères de tests mais nous ne leur demanderons pas de le faire.

Le plan que nous utiliserons est inspiré du IEEE *Standard for Software Unit Testing* (IEEE 1008-1987). Ce plan prévoit décrire l'approche générale du plan des tests, déterminer ce qui doit être testé et concevoir l'ensemble des tests à effectuer avant de les implanter et des les exécuter. Nous décrirons dans la méthode l'approche générale et la sélection des critères de tests à faire. L'ensemble des tests à concevoir, l'implantation et l'exécution des tests seront laissés aux étudiants.

7.4. DIFFÉRENTS NIVEAUX DE TESTS.

Il existe trois niveaux pour les tests d'un logiciel :

test unitaire (*Unit testing*) : Vérifie un logiciel en morceaux, unité par unité. Cette étape débute lorsque le code d'une unité compile correctement (Bertolino et al., 2001; Beizer, 1990; Perry, 1995; Pfleeger, 1998; IEEE 1008-1987),

test d'intégration : Vérifie l'intégration des unités dans le système. Cette étape est faite après les tests unitaires. On procède traditionnellement selon la hiérarchie du logiciel, soit du bas vers le haut, soit l'inverse. Dans les approches plus modernes on cherche plutôt à tester l'intégration des unités fonctionnelles (Bertolino et al., 2001; Jorgensen, 1995; Pfleeger, 1998),

test du système : Vérifie le comportement global du logiciel dans son environnement. C'est aussi à ce moment que sont vérifiées les exigences non-fonctionnelles (sécurité, performance,...) (Bertolino et al., 2001; Jorgensen, 1995; Pfleeger, 1998).

Nous décrirons dans la méthode ces trois niveaux de tests mais principalement les tests unitaires. Certains tests d'intégration se feront de façon incrémentale par la technique de l'échafaudage (*stub*) (pour les conteneurs par exemple) et nous considérerons l'intégration pour la vérification des exigences fonctionnelles et non-fonctionnelles comme faisant partie des tests système.

7.5. DIFFÉRENTES FAÇONS DE TESTER

Pour les niveaux de tests cités précédemment, les tests dépendent des objectifs à atteindre. La liste des objectifs reconnus en génie logiciel est assez longue et exhaustive et peut être atteinte par différentes sortes de tests : acceptation/qualification, installation, Alpha/beta, conformité, fonctionnel, exactitude, fiabilité et évaluation, régression, performance, utilisabilité, Stress, *Back to back*, recouvrement et configuration [12] (Bertolino et al., 2001; chap. 5 ; B2)

Pour les cours de programmation de base nous restreindrons les tests à ceci :

acceptation/qualification : Le but est de vérifier si le logiciel respecte les exigences du client (Bertolino et al., 2001; Perry, 1995; Pfleeger, 1998; IEEE 12207 -1996),

---

[12] Le but de ce chapitre n'est pas de montrer ces notions théoriques mais de choisir celles qui seront utilisées dans la méthode. Pour la mise en application voir p.114.

test de conformité, test fonctionnel et test d'exactitude : Le but est de vérifier si le comportement obtenu est conforme au comportement attendu (Kaner et al., 1999; Perry, 1995; Wakid et al., 1999),

test de performance : Vérifie que le logiciel respecte les exigences non-fonctionnelles de performance (Bertolino et al., 2001; Perry, 1995; Pfleeger, 1998; Wakid et al., 1999). Dans le cas des cours de programmation de base, il se peut que ce soit des contraintes d'efficacité sur les types de données abstraits, par exemple.

La plupart des autres tests demandent trop de temps ou de ressources et ils ne peuvent pas être mis en application dans les cours visés. Par exemple les tests de régression devraient être effectués à chaque fois qu'il y a une modification pour tester de nouveau le système ou les composants (Bertolino et al., 2001; chap. 5; B2). Les tests Alpha/Beta nécessitent des utilisateurs internes (employés) et externes (clients). Évidemment les tests d'installation, d'utilisabilité et autres ne s'appliquent pas. Habituellement pour les tests Acceptation/qualification le client est impliqué. Nous utiliserons ce terme même s'il n'y aura pas d'implication du client et nous ferons faire une revue du document sur la spécification des exigences.

7.6. DIFFÉRENTES TECHNIQUES DE TEST

Le guide au corpus de connaissances (Bertolino et al., 2001) décrit différentes techniques, pour tester un logiciel, selon deux classifications.

La première classification :

- intuition et expérience du testeur,
- basée sur les spécifications,
- basée sur le code,
- basée sur la recherche d'erreurs,
- basée sur l'utilisation,
- basée sur la nature de l'application.

La deuxième classification :

- boîte noire (*black box*) : tests basés sur les spécifications des exigences et les erreurs,
- boîte blanche ou transparente (*white box* ou *glass box*) : tests basés sur le code,
- sélection et combinaison de techniques : autres tests.

Nous n'avons pas l'intention de mentionner les différentes classifications dans la méthode mais les techniques que nous croyons les plus facilement applicables dans le cadre de cours visés sont basées sur les spécifications des exigences et la recherche d'erreurs (*black box*) et le code (*white box*). Nous laisserons tomber les tests basés sur l'intuition et l'expérience pour des raisons évidentes. Nous n'utiliserons pas non plus les tests basés sur l'utilisation puisqu'il n'y aura pas de transition et sur la nature de l'application parce que ce ne seront pas des applications très complexes.

7.6.1. Détail sur la première classification :

1. Tests basés sur les spécifications. Ces tests ont pour but de s'assurer que le logiciel satisfait aux exigences (Bertolino et al., 2001 ; chap. 5 ; C). Ils peuvent être faits par différentes techniques :

- identifier les classes d'équivalence,
- analyser les valeurs limites,
- faire des tables de décision,
- faire des automates finis,
- faire des spécifications formelles,
- faire des tests générés aléatoirement.

2. Tests basés sur le code. Ces tests ont pour but de vérifier l'interaction et les parcours entre les unités, la définition et l'utilisation des variables du programme (Bertolino et al., 2001 ; chap. 5 ; C). Ils peuvent être faits à l'aide de :

- diagramme de décisions,
- diagramme de flux de données,
- révisions formelles.

3. Tests basés sur la recherche d'erreurs. Ces tests ont pour but de relever des erreurs prédéfinies (Bertolino et al., 2001 ; chap. 5 ; C). Ils peuvent être faits grâce à des techniques de :

- prévisions d'erreurs (*error guessing*),
- mutations.

Pour atteindre tous les objectifs cités précédemment il fallait prendre des décisions sur les stratégies à employer pour mettre en place les différentes techniques de tests que nous avons choisies. Il est nécessaire de sélectionner et de combiner quelques-unes de ces techniques.

C'est ce que nous avons fait pour fournir aux étudiants plusieurs façons de vérifier leur travail selon des objectifs prédéterminés. Comme nous ne pouvons pas utiliser dans un seul cours toutes ces techniques et toutes ces sortes de diagrammes, **nous avons opté pour un style plus linguistique pour la description des tests à effectuer. Une stratégie de vérification devra, lors de la conception, prévoir l'analyse des valeurs limites, la recherche d'erreurs et la vérification des exigences (fonctionnelles ou non).**

7.7. DOCUMENTER LES TESTS

L'objectif d'un test, les valeurs choisies pour effecteur ce test (si possible) et les résultats attendus doivent faire partie de la documentation. Un document décrivant la stratégie de vérification sera inclus avec la documentation de l'étape de conception. Ce document décrira tous les tests à effectuer. Un autre document montrant le résultat des tests décrira les **résultats obtenus** en mettant en application la stratégie de vérification décrite à l'étape de conception pour l'évaluation des tests unitaires et des tests vérifiant la satisfaction aux exigences. Nous voulons ainsi que les étudiants voient que la planification des tests peut se faire dès la conception en même temps que les algorithmes. Cela devrait éliminer un certain nombre d'erreurs dès la conception. Nous voulons également que les étudiants aient un comparatif entre les résultats attendus (préparés par eux) et les résultats obtenus (fournis par le logiciel).

Puisque nous ne ferons qu'un document décrivant les tests à effectuer et un document montrant les tests effectivement faits, nous ne respecterons intégralement aucun des documents décrits dans le IEEE *Standard for Software Test Documentation* (IEEE 829-1983). Nous ferons plutôt un modèle adapté qui contiendra tous les tests mais qui respectera les objectifs et les techniques citées dans les paragraphes précédents. **La plupart des techniques de tests que nous décrirons dans la méthode sont connues et proviennent du livre *Software Engineering Theory and Practice (second edition)* de Shari Lawrence Pfleeger (Pfleeger, 2001).**

Dans ce chapitre, nous avons vu qu'il y a plusieurs objectifs différents qui peuvent être atteints lors des tests et que le point de vue des tests peut être différent (spécification, code, recherche d'erreurs, etc.). Nous avons vu aussi qu'il y a des techniques différentes pour tester selon le point de vue adopté. Pour notre part, nous désirons ouvrir l'esprit des étudiants sur les différents points de vue que peuvent prendre les tests. Également, pour chaque point de vue que nous leur présentons, nous voulons que les étudiants voient que le genre de tests dépendra des objectifs à atteindre. De plus, nous voulons que soit bien perçue la différence entre les résultats attendus et les résultats obtenus. C'est pourquoi notre méthode offre dès la conception de faire une stratégie de vérification qui décrit l'analyse des valeurs limites, la vérification des

interactions entre les unités du programme et la prévision d'erreurs avec des valeurs concrètes lorsque c'est possible.

À l'annexe C, page 115, se trouve la description de l'étape des tests telle qu'elle sera présentée aux étudiants.

CHAPITRE VIII

ÉVALUATION DE LA MÉTHODE

Pour évaluer la méthode, nous avons conçu un questionnaire (annexe A) que nous avons fait parvenir à cinq professeurs et maîtres de conférence d'une université française, l'Université de Pau et des pays de l'Adour. Le questionnaire pose des questions autant sur le point de vue des étudiants qui utiliseraient la méthode que du point de vue des enseignants qui auraient à l'utiliser. Les participants y ont répondu et dans ce chapitre nous analysons les commentaires que nous avons reçus et nous répondons à ces commentaires.

8.1. INTRODUCTION

Nous avons fait parvenir un questionnaire (annexe A) à des évaluateurs qui ont accepté généreusement de commenter la méthode que avons conçue. Nous avons séparé ce questionnaire selon deux points de vue : étudiant et enseignant. Cependant, seulement deux des cinq participants ont répondu aux questions du point de vue de l'enseignant et cela venait recouper les commentaires du point de vue des étudiants. Nous tiendrons donc compte des commentaires sur le point de vue de l'étudiant pour les cinq participants. De plus, trois des participants ont répondu en commun, ce qui fait que nous traiterons les commentaires comme s'il n'y avait eu que trois répondants.

Une autre constatation est que nous avons mis des notes au lecteur dans la méthode qui a été évaluée. Ces notes étaient supposées aider les évaluateurs à comprendre les choix qui ont été effectués lors de la construction de la méthode. La plupart des participants ont interprété ces notes comme faisant partie de la méthode. Certains commentaires sont faits sur les notes aux lecteurs et nous n'en tiendront pas compte puisque ces notes ne font pas partie de la méthode.

8.2. QUESTIONS ET RÉPONSES POINTS DE VUE DE L'ÉTUDIANT

Vous trouverez dans le texte qui suit une synthèse des commentaires reçus ainsi que notre évaluation de ces commentaires et ce que nous en ferons. Les commentaires et un résumé de notre évaluation se retrouvent à l'annexe B sous forme de tableau Excel.

Question 1.

La méthode insiste-t-elle suffisamment sur la provenance, la classification par priorité et la numérotation des exigences ?

Commentaires 1 à 7

À cette question un des participants a trouvé que la méthode insistait suffisamment sur la classification et la numérotation. Cependant, concernant la provenance des exigences et des spécifications, il aurait aimé que nous puissions développer davantage la partie description des catégories d'interlocuteurs possibles. Il aurait aimé que nous présentions également une technique et/ou démarche de cueillette d'information. Il mentionne aussi que nous employons « spécifications du client » plutôt que « spécifications de l'enseignant ». Un autre participant trouve qu'il n'y a aucun problème et que la méthode insiste suffisamment sur la provenance, la classification par priorité et la numérotation des exigences.

Un dernier participant trouve que nous insistons assez sur les éléments mentionnés dans la question, mais se questionne néanmoins. Concernant les notions d'« exigence » et de « spécification », il ne voyait pas trop la frontière entre l'une et l'autre. Il a perçu que les exigences relèvent d'aspects « extérieurs » de l'application, c'est-à-dire du comportement attendu de l'application vis-à-vis de l'utilisateur. L'autre (spécification) relève aussi parfois du comportement attendu, mais aussi de propriétés de codage de certains éléments. Le participant se pose la question si la relation de « détail » existant entre ces éléments n'est pas difficile à percevoir pour un programmeur débutant. Le même participant se questionne lorsque le texte parle de « spécifications de l'enseignant », qui ne sont pas les mêmes que les « spécifications » (tout court). Finalement, lorsque la méthode fait référence à la source des exigences, le participant se demande si le fait d'établir la liste des exigences et des spécifications à partir de l'énoncé donné par l'enseignant ne posera pas de problème à l'étudiant lorsque le problème qui lui est posé n'est plus posé par un enseignant. Il suggère d'aider à trouver les questions qu'il faut poser pour compléter le sujet.

Nous avons noté que l'ensemble des participants trouve que la méthode insiste suffisamment sur la provenance, la classification par priorité et la numérotation des exigences. Concernant la provenance des exigences, nous avions pris en compte le travail de Peter Henderson (Henderson, 1986). Ce travail écrit qu'il y a trop de principes de conception qui sont enseignés sans que les étudiants ne soient guidés correctement et que les étudiants ont de la difficulté à faire les liens entre les concepts et la réalité. C'est pourquoi nous avons limité la provenance des exigences à l'énoncé de travail pratique fourni par un enseignant. Dans le même ordre d'idée, nous faisons une distinction entre les contraintes de l'enseignant et les spécifications liées au problème justement pour que l'étudiant perçoive que la source des exigences n'est pas toujours la même. Nous avons ajouté une note de bas de page dans la méthode pour clarifier ce détail. Pour ne pas surcharger cette partie, nous n'ajouterons pas de technique de cueillette d'informations.

Nous avons tenu compte de la suggestion de changer « spécifications de l'enseignant » pour « spécifications du client » et nous avons fait la modification à la méthode. Nous pensons que cela rejoint notre objectif pour que l'étudiant voit que les exigences peuvent provenir d'ailleurs que de l'énoncé de travail pratique.

Question 2.

La méthode insiste-t-elle suffisamment sur l'utilisation des documents sur les exigences et les spécifications, pour l'étape de la conception ?

Commentaires 8 à 12

Un des participants trouve que l'utilisation des documents sur les exigences et les spécifications, pour l'étape de la conception décrit dans la méthode, est assez claire. Par contre, un autre participant trouve que cette partie est un peu lourde pour de petits projets auxquels seront confrontés les étudiants et qu'elle ne semble pas assez formelle pour de gros projets.

Un des participants soulève que, d'après lui, la cueillette d'informations doit s'intéresser aux tâches de l'utilisateur, afin que le logiciel ait un comportement compatible avec les modes de travail de ses utilisateurs. Les spécifications données dans l'exemple semblent s'attacher uniquement aux propriétés intrinsèques de l'application, sans tenir compte du comportement attendu vis-à-vis de l'utilisateur. Un autre de ces commentaires mentionne qu'il semble que le diagramme des exigences et spécifications ait été bâti uniquement par regroupement des exigences listées précédemment, et classées par ordre de priorité, sans tenir compte des fonctionnalités 'dictées' par les tâches de l'utilisateur. Il faudrait selon lui que la phase d'analyse mette à jour ces informations. Ce même participant mentionne qu'il ne voit pas la justification qui motive un deuxième diagramme (diagramme de la solution).

D'abord, nous notons que tous les participants trouvent que cette section est assez claire. Pour ce qui est de la lourdeur de l'étape de la conception, nous savons que le lien entre le quoi et le comment est rempli de subtilité et que cela peut paraître lourd. Rex. E. Gantenbein (Gantenbein, 1989) a écrit que le cycle de vie devrait être enseigné à tous les niveaux de la formation en informatique et nous avons intégré la conception dans la méthode. C'est une version simplifiée, c'est pourquoi cela n'est pas assez formel pour de gros projets. Nous croyons que la version que nous avons ne pourrait pas être moins lourde pour de petits projets et que nous ne pourrions pas la rendre plus formelle sans l'alourdir encore plus.

Pour l'aspect dynamique et comportemental, nous n'en avons pas tenu compte puisque très peu applicable dans le modèle en cascade. C'est plus dans un modèle par prototypage que l'on

sollicite d'emblée l'utilisateur. Dans le cas qui nous intéresse, le seul utilisateur sera l'enseignant. C'est pourquoi nous n'avons pas décrit ce genre de préoccupation dans les exigences. Nous espérons que l'enseignant fera un énoncé clair et qu'il décrira bien ses exigences sur le résultat final à obtenir.

Enfin, nous avons expliqué dans le mémoire en quoi consistait la différence entre le diagramme des exigences et le diagramme de la solution et la justification. Elle doit avoir échappé aux lecteurs.

Question 3.

La méthode insiste-t-elle suffisamment sur la différence entre les exigences et la solution (quoi et comment), pour l'étape de la conception ?

Commentaires 13 à 17

Un des participants trouve que l'insistance de la méthode sur la différence entre le quoi et le comment est assez claire. Ce qui est moins clair dans son esprit est la documentation à produire. Un autre participant mentionne qu'il ne voit aucun problème là-dessus. Le dernier participant indique que, d'après lui, les itérations et les sélections (donc la solution) demanderaient une analyse préalable plus approfondie et devraient commencer à la partie 1 de la conception et non dans la partie sur les algorithmes.

Nous notons que tous les participants trouvent que cet aspect est assez clairement défini dans la méthode. Nous ne comprenons pas l'allusion d'un participant qui n'écrit pas ce qu'il ne trouve pas clair dans la documentation. Pour ce qui est de l'analyse à faire dans la partie 1 de l'étape de la conception, dans le guide au corpus de connaissances (Abran et al., 2001) on écrit que le résultat du processus de conception est un ensemble de modèles et d'artéfacts qui décrivent les décisions majeures qui auront été prises (Abran et al., 2001, chap. 3:3; Budgen, 1994; IEEE 1016-1998; Liskov et Guttag, 2001; Pressman, 1997). À notre avis, que ce soit fait dans la partie un de l'étape de la conception ou dans une autre partie ne compte pas vraiment, en autant que ce soit fait à cette étape. Nous acceptons donc le commentaire sans toutefois faire de modification à la méthode. Ce qui semble incompris est que nous proposons d'utiliser les boucles et les sélections comme base pour tenter de trouver une solution après avoir déterminé l'ordre de traitement des exigences.

Question 4.

La méthode insiste-t-elle suffisamment sur l'utilisation des approches de développement (descendante et ascendante) ?

Commentaires 18 à 20

Un des participants n'a pas fait la distinction entre les deux approches. Un autre participant trouve que le dernier alinéa de la page 20 est redondant avec le premier de la page 21. Le dernier participant se demande s'il ne fallait pas expliquer davantage. Illustrer avec un autre exemple ? Être plus précis, plus directif, pour aider l'étudiant à construire le diagramme hiérarchique ?

Nous notons que deux des trois participants trouvent que c'est suffisant, tandis que le troisième n'a pas vu la distinction que nous faisons entre les deux approches. Cette distinction est faite dans le chapitre sur la construction. Nous avons effectivement constaté qu'il y avait redondance entre le dernier alinéa de la page 20 et le premier de la page 21. Nous avons fusionné les deux alinéas. Concernant les exemples du diagramme hiérarchique, nous adhérons à la demande, mais les modifications seront faites dans une version future, grâce à une étude de cas qui sera annexée à la méthode.

Question 5.

La méthode insiste-t-elle suffisamment sur les techniques permettant de concevoir un logiciel pour réduire sa complexité, faciliter les changements et penser en fonction de la validation ?

Commentaires 21 à 25

Un des participants trouve que l'exemple des dates valides est intéressant par rapport à la réduction de la complexité et à la réutilisation mais qu'il manque un exemple pour illustrer abstraction et encapsulation. Il se demande également si chaque élément du diagramme de solution correspond à un bloc de programme particulier ? Un autre participant a simplement répondu oui à la question sans apporter de commentaires supplémentaires. Le dernier participant pense que ces préoccupations peuvent déjà intervenir à un niveau plus haut de l'étude. Nous mentionnons dans la méthode que la meilleure façon de commenter est de le faire au fur et à mesure. De l'avis du dernier participant, la meilleure façon de commenter un programme 'au fur et à mesure' est de **commencer** par la documentation.

« Dictionnaire de données et dictionnaire des actions. La première version du corps du programme est un code source contenant la description de chaque action à développer, en terme de : éléments nécessaires (entrées : cf. page 12), NomDeLAction, éléments produits-modifiés (sorties : cf. page 12), but de l'action. » (sic).

« La première version du programme est alors un programme complet, compilable, qui contient les **objectifs** de la programmation à réaliser. Il pourra être développé de façon modulaire (action par action) et sera compilable à tout moment. » (sic).

Nous notons que tous les participants trouvent que nous insistons assez sur les techniques mentionnées dans la question. Pour ce qui est de la suggestion d'insérer ces techniques avant dans la méthode, nous serions prêt à le considérer, mais il aurait fallu élaborer sur l'endroit le plus approprié. Finalement, nous avons déjà une difficulté par rapport à la charge de travail que la documentation apporte à ce genre de cours, nous ne croyons pas pouvoir ajouter les dictionnaires à la méthode sans alourdir davantage la documentation. Nous sommes d'accord que cela doit être fait, mais ce devrait être fait dans d'autres cours.

Question 6.

La méthode insiste-t-elle suffisamment sur l'utilisation des diagrammes de solution faits lors de l'étape de la conception ?

Commentaires 26 à 29

Un participant demande de préciser quelles décisions d'implantation particulières ont été prises. Un autre participant écrit qu'à son avis, c'est dans cette partie que l'approche ascendante prend son intérêt. Il faudrait la mettre plus en avant. Finalement, le dernier participant ne voit pas l'intérêt du diagramme de la solution après les algorithmes, mais il comprend d'avantage l'utilité du diagramme des modules.

Ici, il faut mentionner que le diagramme de la solution est décrit à l'étape de la conception, mais est conçu seulement à l'étape de la construction. La justification du diagramme de la solution est faite à la page 19 du mémoire. Nous croyons que la solution finale peut différer de la solution prédéfinie au diagramme des exigences, selon le langage employé. C'est pourquoi nous voulons avoir un plan de la solution finale. Pour ce qui est de préciser les décisions d'implantation, nous croyons que cela mérite considération, mais il faudrait en évaluer la mise en application. Dans les cours de programmation visés, une section sur les structures de données demande un certain niveau de compréhension. Nous n'avions pas cette section dans la méthode, mais nous devrons vérifier la validité d'une telle section dans les cours visés.

Question 7.

La méthode insiste-t-elle suffisamment sur la détection de différents types d'erreurs à l'étape des tests ?

Commentaires 30 à 33

Un des participants trouve que la démarche de tests sur les spécifications, le comportement et la détection d'erreurs, est claire et bien illustrée. Il suggère de développer un lexique pour les termes de génie logiciel qui abondent dans cette partie. Un autre participant écrit que la méthode insiste suffisamment, mais qu'il faudrait insister plus sur les jeux de tests et les choix des jeux de tests. Le dernier participant trouve cette partie simple et intéressante, il comprend bien la nature des tests à effectuer à la lecture du document.

Nous notons que tous les participants trouvent que la méthode insiste suffisamment sur ce point. Pour ce qui est d'ajouter des exemples, cela devrait être fait avec l'ajout d'une étude de cas annexée à la méthode.

Question 8.

La méthode insiste-t-elle suffisamment sur la création d'une stratégie de vérification pour avoir une comparaison avec les résultats obtenus par le logiciel ?

Commentaires 33 à 35

Un des participants écrit que la démarche de tests sur les spécifications, le comportement et la détection d'erreurs sont clairs et bien illustrés pour des petits programmes. Il se demande toutefois quelle stratégie conseiller dans un programme de grande taille. Un autre participant commente en disant que la stratégie n'est pas suffisamment décrite, mais qu'il ne sait pas comment décrire une méthode de stratégie de vérification « générique » alors que par définition elle est spécifique aux problèmes rencontrés. Finalement, le dernier participant trouve cette partie simple et intéressante, il comprend bien la nature des tests à effectuer à la lecture du document (comme à la question précédente).

Nous notons que deux des trois participants trouvent le tout très clair et que l'autre comprend pourquoi il n'y a pas plus de détails. Pour ce qui est de conseiller une stratégie pour les gros projets, une mention sur l'automatisation de certains tests a été ajoutée dans la section « mot sur le génie logiciel » du chapitre sur les tests.

Question 9.

La méthode insiste-t-elle suffisamment sur l'importance de la documentation ?

Commentaires 36 à 38

Un des participants demande des canevas (squelette) de documents très précis à rédiger pour chaque étape. Un autre participant trouve que la méthode insiste peut-être trop sur la documentation, qu'il est difficile pour des élèves de voir l'utilité de toute cette documentation demandée pour de petits projets, car elle ne rencontre sa pleine utilité que lors de « gros » projets. Le dernier participant trouve qu'il n'y a rien sur la gestion de l'évolution de la documentation. Il admet tout de même que cela peut être fait dans d'autres cours.

Nous avons noté que deux des trois participants trouvent que la méthode insiste suffisamment sur ce point. Pour ce qui est du commentaire sur l'utilité de la documentation dans les petits projets, nous en sommes conscients. Susan Mengel écrit que si les étudiants commencent tôt dans les programmes à utiliser des principes de génie logiciel, ils les trouveront utiles dans leurs projets et dans leurs futurs cours (McCauley et al., 1995). C'est dans cette optique que nous avons insisté sur la documentation. Finalement, l'ajout d'une étude de cas devrait donner un modèle de ce qui doit être produit et devrait répondre à la demande d'ajouter des canevas à la méthode.

Question 10.

La méthode insiste-t-elle suffisamment sur la différence entre un petit projet et un projet en génie logiciel ?

Commentaires 39 à 41

Un des participants répond positivement mais trouve que certaines références à des démarches et techniques de GL seront à expliquer par l'enseignant. Un autre participant écrit que la question est plus : « Comment montrer un gros projet de génie logiciel avec des projets à l'échelle étudiante ? ». Le dernier participant répond aussi positivement mais il ne pense pas que les étudiants débutants puissent faire la différence entre l'un et l'autre. Le domaine est bien trop abstrait et sa comparaison avec le génie logiciel est également basée sur des aspects abstraits.

8.3. QUESTIONS ET RÉPONSES : POINTS DE VUE DE L'ENSEIGNANT

Nous sommes assez satisfait du commentaire qui écrit que les enseignants devront montrer des concepts de génie logiciel pour pouvoir utiliser la méthode. Cela prouve que notre méthode rencontre l'objectif d'aider les enseignants à connaître ce qui doit être enseigné dans ce genre de cours, concernant le génie logiciel. Pour les autres commentaires, nous croyons qu'il faudra tester la méthode pour pouvoir dissiper les doutes en ce qui concerne les résultats.

Cette partie visait à vérifier si le texte de la méthode indique aux enseignants sur quelles étapes et sur quels éléments ils devraient insister. Nous avons demandé dans ce qui suit d'évaluer si l'importance accordée à chaque élément est montrée assez clairement, sinon de suggérer des améliorations. De plus, dans tous les cas, nous avons demandé de commenter sur la présentation des éléments, du point de vue de l'aide apportée aux enseignants.

Comme il n'y a que deux participants qui ont répondu à cette section et que les commentaires recoupent ceux du point de vue de l'étudiant, nous les avons mis en vrac. Tous les commentaires se résument par la création d'un lexique sur les termes de génie logiciel, la création d'un tableau synthétique à la fin de chaque étape, la création d'une section sur les décisions d'implantation (structure de données) et l'ajout d'une étude de cas complète faite en utilisant la méthode. Voici les commentaires tels quels :

a) *Quelle démarche préconiser auprès de l'étudiant pour la cueillette d'informations : exigences et spécifications. Existe-t-il des catégories type d'interlocuteur à interviewer : futurs utilisateurs, direction hiérarchique immédiate, direction générale, etc.*
b) *Le lien est clairement établi. Par contre, ici, il y a un gros travail d'expertise à transmettre aux étudiants en ce qui concerne la construction de ce diagramme hiérarchique à partir des exigences et spécifications. Pouvez-vous donner des éléments à l'enseignant ? Existe-t-il des consignes, règles très précises, pour le construire ?*
c) *Vous ne faites jamais référence à la modélisation des données. En France, la méthode Merise (plutôt ancienne), mais d'autres également, prévoient la description des traitements mais aussi celle des données (Modèle Conceptuel de Données Merise, diagramme de classes dans le langage de modélisation UML). Pourquoi ne parlez-vous pas de la structuration des données nécessaires à tous ces traitements ?*
d) *On comprend bien qu'il faut décomposer des problèmes., donc des algorithmes autant de fois que nécessaire par souci de simplification.*

e) *L'enseignant devra expliquer différence entre diagramme hiérarchique et diagramme de solution. Pas facile.*

f) *Ici, description fonctionnelle de l'application. Pour chaque grande fonction, montrer les blocs de programmes qui implémentent ces fonctions ? Éventuels appels à des sous-programmes ?*

g) *Des canevas (squelettes) de documents très précis à rédiger pour chaque étape seraient peut-être plus rassurants pour l'apprenant mais aussi l'enseignant.*

h) *Certaines références à des démarches et techniques de GL seront à expliquer par l'enseignant.*

i) *La méthode fait parfois référence à des démarches et techniques existantes non connues de l'apprenant. L'enseignant devra porter des informations à la connaissance de l'élève. Pour l'aider, pourriez-vous rédiger un lexique : modèle en cascade, approche ascendante, descendante, style architectural, structurogramme, organigramme, librairie de composants, d'objets, d'inspection formelle, [...], différents diagrammes et différentes techniques de tests, etc.*

j) *Pourriez-vous également proposer un cas d'étude relativement complet sur lequel l'enseignant pourrait s'appuyer pour traiter ses propres cas avec les étudiants ?*

k) *La différence entre un petit projet et un projet en génie logiciel ? Commentaire : Elle n'est pas suffisamment mise en évidence. Il faudrait arriver à des mesures, chiffres ce qui en pratique relève de l'impossible...sauf peut être avec des exemples de projets enrichis chaque année.*

l) *Pour qu'un enseignant réalise un cours, la méthode est suffisamment détaillée*

m) *Je pense que la méthode manque d'une fiche de rappel synthétique par rubrique à l'attention des étudiants.*

Nous avons noté, et nous sommes d'accord, que la méthode bénéficierait d'un lexique, d'une étude de cas et d'un tableau synthétique pour chaque étape de la méthode. Toutes ces modifications pourraient être faites dans une prochaine version. Concernant une section sur la modélisation de données, il faudrait vérifier l'application qui peut en être faite dans ce genre de cours.

8.4. COMMENTAIRES SUPPLÉMENTAIRES

Deux participants ont ajouté des commentaires qui ne faisaient pas partie du questionnaire. Nous les mettons ici intégralement, mais nous ne répondrons pas à ces commentaires parce que

cela a déjà été fait ou que cela concerne la forme du document. À noter que le participant numéro un aimerait utiliser notre méthode pour un de ses cours.

Participant numéro 1

Je trouve le projet très intéressant. En effet, j'anime un cours d'initiation à la programmation destiné à des étudiants spécialisés en gestion. Ce ne sont donc pas de futurs informaticiens ; ils ont d'autant plus besoin de méthode pour guider leurs développements (cas d'école relativement simples).

Votre proposition pourrait très bien constituer une première partie de ce cours d'initiation à la programmation, avec d'autres parties comme l'algorithmique et la programmation. Si vous me donnez l'autorisation, je veux bien essayer de la tester sur cet échantillon d'étudiants.

Vous m'avez demandé de critiquer votre travail. Avant de me lancer, je me permets d'insister encore sur le fait que je l'ai beaucoup apprécié. Je vais donc vous faire part de mes remarques qui ne sont d'ailleurs pas celles d'un spécialiste du Génie Logiciel.

La lecture pourrait être facilitée encore davantage par un ***plan*** *marqué de parties numérotées et un sommaire :*

1. *INTRODUCTION*
    - *1.1. LE CYCLE DE VIE DU LOGICIEL*
    - *1.2. DESCRIPTION SOMMAIRE DE LA METHODE*
    - *1.3. DESCRIPTION DETAILLEE DE LA METHODE*
2. *L'ETAPE DES EXIGENCES*
    - *2.1. TERMINOLOGIE*
    - *2.2. OBJECTIFS*
    - *2.3. REGLES DE DEVELOPPEMENT*
    - *2.4. ...*
3. *L'ETAPE DE LA CONCEPTION*
4. *...*

D'autre part, pour l'enseignant que je suis et surtout pour les étudiants qui débutent, il serait intéressant de proposer un ***lexique*** *décrivant les dizaines de termes du Génie Logiciel auxquels vous faites référence dans le document et qui ne sont pas précisés. Par exemple : modèle en cascade, approche ascendante, descendante, style architectural, structurogramme, organigramme, librairie de composants, d'objets, d'inspection formelle, ..., différents diagrammes et différentes techniques de tests, ...*

*La mise à disposition d'un **cas d'étud**e développé à l'aide de votre méthode serait un plus. Pourriez-vous ainsi proposer un cas d'étude relativement complet sur lequel l'enseignant pourrait s'appuyer pour traiter ses propres cas avec les étudiants ?*

Participant numéro 2

Cette méthode a une approche assez abstraite : l'étudiant doit adhérer à une préoccupation méthodologique pour avancer (ce qui n'est pas le cas de tous les étudiants...).

*Néanmoins, elle est à mon sens très intéressante, car elle s'inscrit dans une tentative de décrire une **démarche** permettant de trouver les exigences d'un problème, définir et organiser une solution. La littérature dans ce domaine est très rare, elle se cantonne souvent à la seule phase de programmation.*

Le document décrivant la méthodologie doit donc être particulièrement facile à lire afin de ne pas ajouter de surcharge cognitive supplémentaire à l'étudiant.

Il gagnerait en lisibilité si on pouvait distinguer visiblement :

- *les paragraphes qui correspondent aux étapes de la méthodologie*
- *les paragraphes qui correspondent à un apport supplémentaire d'information (par exemple, la comparaison de la méthode au GL)*
- *les paragraphes qui sont destinés aux enseignants*

Moyens préconisés :

- *paragraphes numérotés, figures numérotées, polices distinctes, des encadrés, bref, une localisation physique plus aisée des différents types de rubriques aiderait sans doute.*
- *peut-être une séparation physique des paragraphes destinés aux étudiants de ceux destinés au lecteur ?*

Toutes mes remarques ont été regroupées dans la première partie de ce questionnaire. Je crains de n'avoir mélangé les points de vues étudiants-enseignants : Veuillez m'en excuser.

Nous avons eu des préoccupations similaires dans le cadre de nos enseignements en bases de la programmation du département informatique de l'IUT de Bayonne. Il en est sorti une méthodologie de la même famille que la vôtre, sans doute plus simple car moins rattachée aux principes du génie logiciel. Elle couvre les premières 60 heures d'enseignement des bases de la programmation.

Voici un résumé des commentaires dont nous n'avons pas tenu comptes et la justification :

- Rejet car ne s'applique pas dans les cours visés par la méthode.
    - Ajouter des techniques de cueillette d'information.
    - Ajouter les exigences de l'utilisateur final à l'étape des exigences.
    - Ajouter des dictionnaires (données et actions).
- Rejet car il nous manque de détails.
    - Clarifier la documentation à produire.
    - Faire intervenir la réduction de la complexité et la validation à un niveau plus haut de l'étude.
- Rejet car nous avions déjà justifié les raisons qui nous avaient fait prendre ces décisions.
    - Faire ressortir les itérations et les sélections à la partie 1 de l'étape de la conception.
- Rejet car la méthode contient ce qui a été demandé (cela a simplement échappé au lecteur) ou que le commentaire contredit l'opinion des autres participants.
    - Mettre en avant l'approche ascendante.
    - Commenter l'utilité d'avoir un diagramme de la solution.
    - Manque de jeux de tests.
    - Stratégie de vérification insuffisamment décrite.

Voici un résumé des commentaires acceptés avec modification immédiate :

- Modifier quelques fautes typographiques identifiées par un des participants.
- Changer « spécifications de l'enseignant » par « spécifications du client » à l'étape des exigences.
- Fusionner le dernier alinéa de la page 20 avec le premier alinéa de la page 21 qui étaient redondants.
- Ajouter une clarification pour la distinction entre les spécifications de l'enseignant et les spécifications tout court.
- Parler des techniques automatisées de tests à l'étape des tests, dans la section *mot sur le génie logiciel*.
- Ajouter une table des matières.

Voici un résumé des commentaires acceptés avec modification à venir :

- Faire un lexique des termes de génie logiciel.

  - Nous avons su qu'il existe une version française d'un des livre de Ian Sommerville (Sommerville, 1997 ou Sommerville, 2001) qui offre un lexique des termes employés en génie logiciel. Nous avons l'intention de consulter ce manuel et d'employer le lexique qui s'y trouve.
- Faire un tableau synthétique à la fin de chaque étape.
  - Cette tâche devrait être faite sous peu puisque nous avons l'intention de poursuivre notre étude en mettant en application la méthode dans des cours de programmation.
- Évaluer la pertinence et la mise en application d'une section sur les structures de données.
  - Étant donné le niveau des cours visés, l'aspect structure de données nous semble un peu prématuré. Nous devrons vérifier la pertinence d'une telle section avant de déterminer si nous en ajoutons une. Si nous croyons que c'est le cas, une section devra être ajoutée à la méthode. Pour ce faire, nous respecterons ce qui est écrit dans le guide au corpus de connaissances (Abran et al., 2001) à ce sujet.
- Annexer une étude de cas complète faite en utilisant la méthode.
  - Cette étude aussi devra être faite avant la mise en application de la méthode. Nous avons déjà fait ce genre de travail dans un cours de programmation. Il ne nous resterait qu'à adapter l'exemple pour qu'il respecte la méthode. Nous croyons également qu'il faudra plus d'un exemple car les connaissances des étudiants évoluent dans ces cours et il y a des connaissances qu'ils n'ont pas pour leurs premiers travaux pratiques mais qu'ils acquièrent pour les autres travaux.

Des commentaires supplémentaires faits par les participants mentionnent leur intérêt pour la méthode. Nous les avons mis puisqu'ils confirment que notre méthode remplit les objectifs que nous espérions atteindre. Nous n'avons pas répondu aux commentaires additionnels puisqu'ils touchent plus à la mise en page du document qu'à son contenu. Nous avons, dans ce chapitre, pris tous les commentaires des participants à l'évaluation de la méthode et nous avons répondu à chacun des commentaires en spécifiant ce que nous ferions de ces commentaires. Il en ressort que les participants ont trouvé la méthode intéressante. Il y a quelques commentaires pour lesquels la tâche qui en résulte est trop grosse pour qu'il soit réalisable de modifier la méthode dans les délais requis pour ce mémoire. Il s'agit de : faire un lexique de tous les termes de génie employés dans la méthode, faire un tableau synthétique à la fin de chaque chapitre pour

aider aux enseignants et aux étudiants qui auraient à utiliser la méthode, évaluer la possibilité de faire une section pour la modélisation de données et faire une étude de cas utilisant la méthode. Pour les autres commentaires, soit qu'ils ont été rejetés parce qu'ils ne s'appliquent pas, soit qu'ils ont été acceptés et les modifications ont été effectuées ou finalement qu'ils ont été notés comme étant des commentaires satisfaisants qui confirment ce que nous espérions obtenir de cette méthode.

CHAPITRE IX

CONCLUSION

9.1. SYNTHÈSE DE LA RECHERCHE

Nous avons commencé cette recherche car nous croyions qu'il y avait des difficultés lors de l'apprentissage mais aussi au niveau de l'enseignement des cours de programmation de base en informatique, au niveau universitaire et collégial. En faisant une revue de la documentation sur le sujet, nous avons eu la confirmation que des difficultés existaient et qu'il n'y avait toujours pas de solution satisfaisante. Un de nos objectifs était donc de proposer une méthode qui permettrait d'aider à l'apprentissage de la programmation. Nous avons entrepris pour cela de concevoir une méthode de développement qui serait utilisée pour la résolution de problèmes. En plus d'aider les utilisateurs dans la résolution de leurs problèmes informatiques, nous prétendons qu'utiliser des principes de génie logiciel dès le départ de l'apprentissage de la programmation dans de petits projets serait bénéfique pour les utilisateurs de cette méthode, lors de leur passage à de gros projets. C'est pourquoi cette méthode de développement a été construite en utilisant le guide au corpus de connaissances en génie logiciel (Abran et al., 2001).

Comme autre objectif, nous désirions que cette méthode soit un bon outil pour aider des enseignants qui n'ont pas toutes les connaissances nécessaires en génie logiciel à guider leurs étudiants.

La méthode que nous avons produite reflète quatre étapes du cycle de vie en cascades d'un logiciel (exigences, conception, construction, intégration et tests). La description de chaque étape a été divisée en sept parties :

1. Terminologie
2. Objectifs
3. Règles de développement
4. Lien entre les artéfacts
5. Mot sur les grands projets
6. Description de l'artéfact
7. Exemple d'artéfact

Lorsque la méthode a été complétée, nous avons demandé à des personnes, que nous avions choisies, d'évaluer la méthode et de la commenter. Un questionnaire (annexe A) leur a été fourni et les participants ont été appelés à faire des commentaires du point de vue d'un étudiant et ensuite du point de vue d'un enseignant. Il y a eu un malentendu avec la partie du point de vue de l'enseignant, mais cela n'a pas nuit à ce que nous voulions que ces commentaires apportent à la méthode. Par la suite, nous avons compilé les commentaires par question et nous avons défini ce que nous entendions faire avec ces commentaires. Certains commentaires ont été acceptés et les modifications demandées ont été apportées immédiatement, certains ont été notés sans plus, d'autres ont été refusés (avec justification) et finalement certaines suggestions ont été acceptées, mais seront traitées dans une future version de la méthode.

Nous espérons que la méthode que nous avons produite aidera à l'enseignement et à l'apprentissage de la programmation ainsi qu'à la résolution de problèmes, dans les cours de base. Nous espérons aussi que l'insistance que nous avons mise sur le génie logiciel dans cette méthode permettra un passage plus facile à des projets de plus grande envergure. Nous avons tout de même eu la confirmation, grâce aux commentaires des participants, que notre méthode était justifiée et qu'il y avait de la place pour ce genre de méthode dans les cours de programmation de base.

9.1.1. Objectifs et apport

Un des participants a mentionné qu'il était intéressé à utiliser notre méthode, pour un cours d'introduction à la programmation. Un autre participant trouve que l'approche est intéressante et que c'est bien de proposer aux étudiants une démarche. Il mentionne également qu'une étude moins poussée l'avait amené à créer un genre de méthode moins formelle (avec des considérations similaires), dans les cours de programmation qu'il enseigne. Nous considérons donc que nous avons atteint les objectifs visés en ce qui concerne la pertinence, l'utilité et l'intérêt d'avoir une méthode comme la nôtre dans les cours de programmation de base.

9.1.2. Forces

La force principale de cette méthode à notre avis est que tout ce qui y est inclus est appuyé par un corpus de connaissances (Abran et al., 2001).

9.1.3. Principales limites de notre étude

- Les objectifs à long terme comme le passage au génie logiciel, une meilleure résolution de problème de la part des étudiants et l'aide à l'enseignement ne sont pas testés.
- Les commentaires obtenus proviennent d'un nombre limité de participants.

9.1.4. Objectifs personnels

Nos objectifs personnels étaient de produire une méthode, inexistante jusqu'à maintenant, qui pourrait servir dans les cours d'introduction à la programmation. De plus, nous avions comme objectif de parfaire nos connaissances en génie logiciel. L'utilisation du corpus de connaissances (Abran et al., 2001) nous a permis d'approfondir amplement nos connaissances. Sans l'ombre d'un doute, nous pouvons dire que nos objectifs personnels ont été atteints.

9.1.5. Travaux futurs

- Il faudra modifier la méthode pour tenir compte des modifications qui ont été acceptées et qui n'ont pas été faites avant sa mise en application.
- Il faudrait mettre la méthode en application dans des cours de programmation et vérifier si effectivement cela rencontre les objectifs visés. Cela ne sera pas une mince tâche, mais nous croyons qu'il serait important de le faire.
- Les buts que nous avions en faisant évaluer notre méthode étaient de faire confirmer la pertinence et l'utilité d'une telle méthode par des enseignants qui ont les mêmes préoccupations que nous, ainsi que d'avoir une opinion sur les améliorations immédiates ou futures qui pourraient être apportées à la méthode. Nous sommes satisfaits des commentaires des répondants qui nous ont permis de faire certaines modifications immédiatement, de confirmer la justesse de certaines décisions que nous avions prises et d'avoir une opinion des modifications à apporter dans le futur.

ANNEXE A

QUESTIONNAIRE D'ÉVALUATION

Vous trouverez ici le questionnaire d'évaluation qui a été envoyé au participant.

Nous vous serions reconnaissant de nous donner vos commentaires sur la méthode que nous vous avons présentée. Pour vous aider dans votre réflexion, nous vous avons présenté une liste de questions pour lesquelles nous aimerions avoir vos commentaires. Naturellement, tout autre point que vous voudriez soulever sera le bienvenu. Merci à l'avance pour le temps que vous m'accordez.

Commentaires

1) Commentez les points suivants de cette méthode dans l'optique qu'ils pourraient aider des étudiants, non seulement dans leurs premiers cours de programmation de base, mais aussi lors de la création de plus gros projet.
La méthode insiste suffisamment sur la provenance, la classification par priorité et la numérotation des exigences ?
La méthode insiste suffisamment sur l'utilisation des documents sur les exigences et les spécifications, pour l'étape du design ?
La méthode insiste suffisamment sur la différence entre les exigences et la solution (quoi et comment), pour l'étape du design ?
La méthode insiste suffisamment sur l'utilisation des approches de développement (descendante et ascendante) ?

La méthode insiste suffisamment sur les techniques permettant de concevoir un logiciel pour réduire sa complexité, faciliter les changements et penser en fonction de la validation ?
La méthode insiste suffisamment sur l'utilisation des diagrammes de solution faits lors de l'étape du design ?
La méthode insiste suffisamment sur l'importance de la documentation ?
La méthode insiste suffisamment sur la détection de différents types d'erreurs lors de l'étape des tests ?
La méthode insiste suffisamment sur la création d'une stratégie de vérification pour avoir une comparaison avec les résultats obtenus par le logiciel ?
La méthode insiste suffisamment sur la différence entre un petit projet et un projet en génie logiciel ?

2) Commentez les points suivants de cette méthode dans l'optique qu'ils pourraient aider des enseignants pour la préparation de cours.
L'enseignant voit qu'il pourrait insister sur la provenance, la classification par priorité et la numérotation des exigences ?
L'enseignant voit qu'il pourrait insister sur l'utilisation des documents sur les exigences et les spécifications, pour l'étape du design ?

L'enseignant voit qu'il pourrait insister sur la différence entre les exigences et la solution (quoi et comment), pour l'étape du design ?
L'enseignant voit qu'il pourrait insister sur l'utilisation des approches de développement (descendante et ascendante) ?
L'enseignant voit qu'il pourrait insister sur les techniques permettant de concevoir un logiciel pour réduire sa complexité, faciliter les changements et penser en fonction de la validation ?
L'enseignant voit qu'il pourrait insister sur l'utilisation des diagrammes de solution faits lors de l'étape du design ?
L'enseignant voit qu'il pourrait insister sur l'importance de la documentation ?
L'enseignant voit qu'il pourrait insister sur la détection de différents types d'erreurs lors de l'étape des tests ?
L'enseignant voit qu'il pourrait insister sur la création d'une stratégie de vérification pour avoir une comparaison avec les résultats obtenus par le logiciel ?
L'enseignant voit qu'il pourrait insister sur la différence entre un petit projet et un projet en génie logiciel ?
3) Commentez si vous croyez ou non qu'il serait possible d'utiliser cette méthode, pour l'enseignement de la programmation dans les cours de base en génie logiciel ? (trop court, trop long, trop technique, pas assez technique, etc.)
Commentaires additionnels

La méthode insiste suffisamment sur la provenance, la classification par priorité et la numérotation des exigences ?

ANNEXE B

Identification	Questions	Réponses	Justification	Commentateurs
1	Oui, pour la classification et la numérotation	Noté		Christian Sallaberry
2	Concernant la provenance des exigences et des spécifications, pourriez-vous développer davantage la partie description des catégories d'interlocuteurs possibles ?Pourriez-vous également présenter une technique et/ou démarche de cueillette d'information ?	Refusé	Étant donné que le seul client est l'enseignant, et que ce sont de petit projet, nous croyons que cela ne s'applique pas à ce stade.	
3	Parler des spécifications de l'enseignant (P6) ne correspond pas à une étude de cas réelle. Ne peut-on parler du client ?	Accepté	La modification a été apportée	
4	Aucun problème là dessus	Noté		Philippe Roose
5	Sur les notions d' « exigence » et de « spécification ». Je ne vois pas trop la frontière entre l'une et l'autre. Je crois comprendre que l'une (exigence) relève bien de aspects « extérieurs » de l'application, c'est à dire du comportement attendu de l'application vis à vis de l'utilisateur L'autre (spécification) relève aussi parfois du comportement attendu (S5, S6), mais aussi, de propriétés de codages de certains éléments (un S4 . un numéro de plaque est composé de 3 caractères et de 3 chiffres). La relation de « détail » existant entre ces éléments n'est-elle pas difficile à percevoir pour un programmeur débutant ?	Noté	Absolument, c'est pourquoi nous l'introduisons au départ	Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret
6	Par moments, le texte parle de « spécifications de l'enseignant » (page 6), qui ne sont pas les mêmes que les « spécifications » (tout court).	Accepté	Nous faisons une distinction car les contraintes de l'enseignant ne sont pas des spécifications habituellement liées au problème à résoudre et nous voulons que ce soit transparent. Nous avons ajouté une note de bas de page dans la méthode pour clarifier ce détail.	
7	Dans la partie « Résolution de problèmes –La source » (page 7) : A la lecture de ce paragraphe, je comprends que la liste des exigences et des spécifications s 'établit àpartir de l'énoncé donné par l'enseignant. Ok, mais que fait l'étudiant pour établir les exigences lorsque le problème qui lui est posé n'est plus posé par un enseignant ? Autrement dit, un énoncé est incomplet la plupart du temps. La méthodologie doit donc aussi l'aider à trouver les questions qu'il faut poser pour compléter le sujet.	Accepté partiellement	La source des exigences provient de l'enseignant et des énoncés de travaux. Ils doivent aller à la source Nous espérons que si la source change, ils s'adapteront Cependant, nous avons mis une note de bas de page dans la méthode pour illustrer ce cas.	

La méthode insiste suffisamment sur l'utilisation des documents sur les exigences et les spécifications, pour l'étape du design ?

Identification	Commentaires	Réponses	Justification	Commentateurs
8	Assez clair	Noté		Christian Sallaberry
9	Cette partie est à mon sens un peu lourde pour de petits projets auxquels seront confrontés les étudiants. Elle ne me semble pas assez formelle pour de gros projets	Noté		Philippe Roose
10	La cueillette d'informations doit aussi s'intéresser aux tâches de l'utilisateur, afin que le logiciel ait un comportement compatible avec les modes de travail de ses utilisateurs. Les spécifications données dans l'exemple et faisant partie du document produit à cette étape semblent s'attacher uniquement aux propriétés intrinsèques de l'application, sans tenir compte du comportement attendu vis à vis de l'utilisateur.	Noté	Nous sommes en accord, sauf que ce n'est pas applicable puisqu'il n'y a pas d'utilisateur final dans les cours visés.	Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret
11	Le diagramme des exigences et spécifications (page 15) a été bâti uniquement par regroupement des exigences listées précédemment (démarche ascendante), et classées par ordre de priorité, sans tenir compte des fonctionnalités 'dictées' par les tâches de l'utilisateur.	Noté	Voir commentaire suivant	
12	Le texte parle d'un second diagramme (solution), qui représente les décisions sur le découpage en sous-exigences et fonctionnalités, mais je n'ai pas trouvé d'indications sur les raisons qui motiveraient un autre découpage. C'est sans doute lâqu'interviennent les tâches de l'utilisateur, les aspects plus 'dynamiques/comportementaux' de l'application. mais pour cela il faut que la phase d'analyse ait mis à jour ces informations.	Refusé	Nous avions mis une note dans le mémoire pour expliquer la différence et la justification du choix de ces 2 diagrammes.	

La méthode insiste suffisamment sur la différence entre les exigences et la solution (quoi et comment), pour l'étape du design ?

Identification	Commentaires	Réponses	Justification	Commentateurs
13	Clair également	Noté	voir commentaire numéro 8	Christian Sallaberry
14	Aucun problème là dessus	Noté		Philippe Roose
15	Oui – avec les remarques de la questions numéro 2	Déjà notées		Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret
16	Ce qui est moins clair dans mon esprit est la documentation à produire (listée page 14-Description de la documentation).	Refusé	Nous ne comprenons pas l'allusion ici. Qu'est-ce qui n'est pas clair dans la documentation?	
17	L'étape 1.- de la méthode de résolution d'informations indique qu'il faut préciser la stratégie en pseudo code en introduisant les itérations (boucles) et sélections (si). Or, il me semble que les itérations et sélections sont le résultat d'une analyse préalable des actions que le logiciel devra réaliser ainsi que des éléments de l'application sur lesquels ces actions vont porter (s'agit-il de valeurs simples, composées, multiples…). Il me semble que les étapes 3,4 et/ou 5 de la phase de design (page 12) ne devraient pas donc se contenter de lister les éléments de l'application, mais les étudier de façon un peu plus approfondie, ainsi que les actions auxquelles ils se rapportent, afin de pouvoir traduire (étape 1, page 15) cette complexité en termes d'itérations et/ou de sélections.	Accepté sans modification	Nous sommes d'accord. C'est dans le Design que ce fait l'analyse approfondi des spécifications. Ce qui semble incompris est que ce que nous proposons est d'utiliser les boucles et les sélections comme base pour tenter de trouver une solution. Nous tenterons de faire transparaître ce point.	

La méthode insiste suffisamment sur l'utilisation des approches de développement (descendante et ascendante) ?

Identification	Commentaires	Réponses	Justification	Commentateurs
18	On y trouve simplement des références à l'approche descendante (décomposition de pb. en ss-pb.). Faut-il expliquer davantage ? Illustrer avec un autre exemple ?Une difficulté pour l'étudiant : construire le diagramme hiérarchique àpartir des exigences. Peut-on être plus précis, plus directif, pour l'aider ?Ensuite, le cours d'algo. permet de détailler chaque élément de ce diagramme	Accepté sans modification	Le découpage en sous exigences est difficile, mais nous croyons que les multiples possibilités nous empêchent de montrer tous les cas. C'est à force d'expérience que les étudiants sauront ce qui nécessite du découpage et ce qui ne le nécessite pas. C'est en analysant qu'on devient analyste.	Christian Sallaberry
19	J'avoue ne pas avoir vu cette distinction d'approches dans cette partie	Noté	Effectivement, cette distinction est faite à l'étape suivante	Philippe Roose
20	Il me semble que le dernier alinéa de la page 20 est redondant avec le premier de la page 21 : il s'agit de la même idée.	Accepté	La modification a été apportée. Les 2 alinéas ont été fusionnés.	Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret

La méthode insiste suffisamment sur les techniques permettant de concevoir un logiciel pour réduire sa complexité, faciliter les changements et penser en fonction de la validation ?

Identification	Commentaires	Réponses	Justification	Commentateurs
21	L'exemple des dates valides est intéressant par rapport à la réduction de la complexité et à la réutilisation.	Noté		Christian Sallaberry
22	Il manque un exemple pour illustrer abstraction et encapsulation	Refusé	C'est une notion théorique que nous présumons qui doit être montrée dans un cours.	
23	Est-ce que chaque élément du diagramme de solution correspond à un bloc de programme particulier ?	Accepté	Nous avons ajouté une note pour spécifier ce point.	
24	Il me semble toutefois que cette préoccupation peut déjà intervenir à un niveau plus haut de l'étude.	Refusé	Nous serions prêt à débattre là dessus, mais il aurait fallu élaboré.	Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret
25	Concernant la remarque page 25. La meilleure façon de commenter un programme 'au fur et à mesure' est de **commencer** par la documentation. a) dictionnaire de données b) dictionnaire des actions. La première version du corps du programme est un code source contenant la description de chaque action à développer, en terme de : éléments nécessaires (entrées cf page 12), NomDeLAction, éléments produits-modifiés (sorties cf page 12), but de l'action. La première version du programme est alors un programme complet, compilable, qui contient les **objectifs** de la programmation à réaliser. Il pourra être développé de façon modulaire (action par action), et sera compilable à tout moment.	Refusé	Nous avons déjà une difficulté par rapport à la charge de travail que la documentation apporte à ce genre de cours. Nous ne croyons pas pouvoir ajouter les dictionnaires à la méthode.	

La méthode insiste suffisamment sur l'utilisation des diagrammes de solution faits lors de l'étape du design ?

Identification	Commentaires	Réponses	Justification	Commentateurs
26	A chaque élément du diagramme hiérarchique correspond au moins un élément du diagramme de solution ? Niveau Design : Diagramme hiérarchique et enchaînement des algorithmesNiveau Conception et Implantation : enchaînement de blocs de programmes correspondants à ces algo. Préciser quelles décisions d'implantation particulières peuvent être prises ici.	Noté	Voir numéro 12	Christian Sallaberry
27	C'est dans cette partie que l'approche ascendante prends de son intérêt. Il faudrait la mettre plus en avant.	Accepté	L'exemple est fait dans la partie conception et implantation.	Philippe Roose
28	Je ne vois pas l'intérêt de faire ce diagramme APRES le codage des algorithmes. En effet, autant je comprends l'intérêt de distinguer problème et solution, autant je pense que faire un diagramme de la solution après le codage des algorithmes n'est pas en accord avec une démarche descendante. Le code doit produit doit être conforme àune solution préalablement définie. Ce n'est pas ce que je comprends à la lecture de la page 26.	Refusé	La justification de ce diagramme est faire à la page 27 et nous croyons que la solution finale peut différer de la solution prédéfinie selon le langage employé.	Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret
29	Je comprends davantage le diagramme des librairies, qui indique le rangement PHYSIQUE des programmes dans des fichiers/paquetages/librairies.	Noté	voir commentaire prédécent	

La méthode insiste suffisamment sur la création d'une stratégie de vérification pour avoir une comparaison avec les résultats obtenus par le logiciel ?

Identification	Commentaires	Réponses	Justification	Commentateurs
33	Démarche de tests sur les spécifications, le comportement et la détection d'erreurs, claire et bien illustrée pour des petits programmes. Quelle stratégie conseiller (automatisation ?, ...) pour test de toutes les spécifications, ... dans un programme de grande taille	Accepté	Nous avons noté que le participant trouve les le tout est clair et bien illustré et nous avons ajouté la phrase " plusieurs outils automatisés" dans le mot sur le génie logiciel de ce chapitre qui parle des différentes façon de faire le tests.	Christian Sallaberry
34	La stratégie n'est pas suffisamment décrite, mais comment décrire une méthode de stratégie de vérification de stratégie « générique » alors que par définition elle est spécifique aux problèmes rencontrés.	Noté		Philippe Roose
35	Simple et intéressante, on comprend bien la nature des tests à effectuer à la lecture du document.	Noté		Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret

La méthode insiste suffisamment sur l'importance de la documentation ?

Identification	Commentaires	Réponses	Justification	Commentateurs
36	Oui, toutefois, des canevas (squelette) de documents très précis à rédiger pour chaque étape seraient peut-être plus rassurant pour l'apprenant mais aussi l'enseignant	Refusé	Nous croyons que les gabarits dans ce genre de méthode donnent un carcan aux étudiants qui ne leurs seraient pas profitables.	Christian Sallaberry
37	Oui, trop peut être. Il est difficile pour des élèves de voir l'utilité de toute cette documentation demandée pour de petits projets. Elle ne rencontre sa pleine utilité que lors de « gros » projets	Noté	Nous en sommes conscient, c'est pour cela que nous voulons y préparer les étudiants dès le début de leur formation et que le moment venu, ils y voient une utilité.	Philippe Roose
38	Une des difficultés dans la maintenance de la documentation est la gestion des versions de ces documents, c'est à dire quelles sont les techniques pour gérer l'évolution de la documentation ? (nomenclature de documents, notion de numéro de version, table des modifications, ….). Des notions succinctes sur ces techniques (extérieures au domaine de la programmation et qui peuvent être vues en dehors des cours de programmation) peuvent néanmoins aider les étudiants maîtriser l'évolution de la documentation)	Noté	Parfaitement en accord. Cela devra être enseigné éventuellement.	Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret

La méthode insiste suffisamment sur la différence entre un petit projet et un projet en génie logiciel ?

Identification	Commentaires	Réponses	Justification	Commentateurs
39	Oui, Certaines références à des démarches et techniques de GL seront à expliquer par l'enseignant	Noté		Christian Sallaberry
40	La question est plus : comment montrer un gros projet de génie logiciel avec des projets à l'échelle étudiante ?	Noté	C'est effectivement un problème, mais que nous ne pourront pas régler dans ce genre de cours.	Philippe Roose
41	Oui, Néanmoins, je ne pense pas que les étudiants débutants puissent faire la différence entre l'un et l'autre. Le domaine est bien trop abstrait, et sa comparaison avec le génie logiciel est également basée sur des aspects abstraits.	Noté	C'est ce qu'il faudrait tester éventuellement dans une recherche plus poussée sur les résultats produits par l'utilisation de cette méthode	Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret

La méthode insiste suffisamment sur la différence entre un petit projet et un projet en génie logiciel ?

Identification	Commentaires	Réponses	Justification	Commentateurs
39	Oui, Certaines références à des démarches et techniques de GL seront à expliquer par l'enseignant	Noté		Christian Sallaberry
40	La question est plus : comment montrer un gros projet de génie logiciel avec des projets à l'échelle étudiante ?	Noté	C'est effectivement un problème, mais que nous ne pourront pas régler dans ce genre de cours.	Philippe Roose
41	Oui, Néanmoins, je ne pense pas que les étudiants débutants puissent faire la différence entre l'un et l'autre. Le domaine est bien trop abstrait, et sa comparaison avec le génie logiciel est également basée sur des aspects abstraits.	Noté	C'est ce qu'il faudrait tester éventuellement dans une recherche plus poussée sur les résultats produits par l'utilisation de cette méthode	Thierry Nodenot Philippe Lopisteguy Pantxika Dagorret

ANNEXE C

MÉTHODE DE DEVELOPPEMENT DE LOGICIEL

Introduction

Il y a plusieurs façons de concevoir un logiciel, mais il y a peu de méthodes provenant de recherches rigoureuses qui expliquent les étapes de développement et les liens à faire entre chacune des étapes.

La méthode suivante est construite sur des bases de faits reconnus dans le monde du génie logiciel, provenant du *Guide to the SoftWare Engineering Body Of Knowledge* (SWEBOK). Le but de cette méthode est d'apprendre à développer de petits projets, avec une perspective de grands projets.

Finalement, cette méthode offre une explication des liens à faire avec chaque étape du développement, le but étant ici d'utiliser la méthode de développement pour résoudre des problèmes.

L'approche suggérée insiste, lors du développement du logiciel, sur les points suivants :

- connaissance des objectifs précis de l'étape actuelle et comment y parvenir,
- connaissance de l'étape précédente (s'il y a lieu),
- connaissance de l'étape suivante (s'il y a lieu),
- comment passer d'une étape à l'autre,
- connaissance de la documentation à produire pour chaque étape.

Cette méthode n'est pas un cours théorique de programmation. C'est une liste des choses à faire pour agencer des concepts théoriques appris dans les cours de programmation, pour chacune des étapes du cycle de vie d'un logiciel. L'étape de maintenance, elle, commence lorsque le logiciel est chez le client. C'est pour cette raison que nous ne décrirons pas l'étape de maintenance dans ce document.

Modèle en cascade (« *waterfall* »)

Le cycle de vie d'un logiciel est constitué de processus (exigences, conception, construction, intégration et tests, maintenance) qu'on doit franchir pour développer un programme informatique. Il existe plusieurs modèles du cycle de vie (cascade, prototypale, spirale, etc.). Cette méthode utilise le modèle en cascade dans lequel il est stipulé que nous devons compléter

une étape du cycle de vie avant de passer à la suivante. Nous considérons que ce modèle est le plus approprié pour la programmation à petite échelle.

Différentes approches (descendante et ascendante)

Idéalement, dans le modèle en cascade, nous utilisons *l'approche descendante*, où on divise le logiciel en plusieurs parties et un cycle de vie est effectué sur chacune de ces parties, ce qui facilite les étapes de tests et de maintenance. Chaque problème à résoudre est subdivisé en plus petits problèmes jusqu'à ce qu'il soit possible de les solutionner individuellement en peu de ligne de code. Chaque étape est complétée avant de passer à la suivante. Il est aussi possible de se créer des outils (procédures, fonctions et modules) qui aident pour les problèmes récurrents (ou semblables) de projet en projet. Cela fait partie d'une **approche ascendante**. Nous utilisons ces deux approches dans la méthode.

Documentation

Pour chaque étape du cycle de vie, il y aura un document à produire. Cette méthode décrit de quelle façon ces documents doivent être construits. Elle montre quelles sont les règles à suivre pour arriver à produire les documents de chaque étape du cycle de vie et comment utiliser ce document pour passer à l'étape suivante du cycle de vie. Nous espérons fortement que la production de ces documents serve de démarche pour résoudre des problèmes plutôt que de servir seulement à documenter des solutions déjà trouvées.

Description sommaire de la méthode

Dans cette méthode il y a autant de documents qu'il y a d'étapes du cycle de vie (à l'exception de l'étape de maintenance). La description des étapes comporte les sections suivantes :

- l'introduction,
- la terminologie employée pour la réalisation de l'artéfact[13] à produire,
- les objectifs à atteindre,
- la documentation à produire (en respectant des règles de développement),
- les liens entre les étapes en utilisant le document précédent et le suivant (lorsque cela s'applique),
- les exemples illustrant la documentation à produire.

[13] Artéfact : c'est un produit réalisé au cours d'une étape. La documentation, le code, les tests et la version exécutable d'un programme sont tous des artéfacts.

Description détaillée de la méthode

Terminologie

Nous donnons la définition des termes employés dans la construction de l'artéfact à produire, lorsque nous croyons que cela nécessite des précisions.

Objectifs

Nous définissons les objectifs de l'étape et le rôle de l'artéfact à produire. Par exemple, l'artéfact pour l'étape des exigences permet entre autres de décrire les exigences fonctionnelles et les exigences non-fonctionnelles.

Règles de développement

Nous fournissons un ensemble des règles permettant la création, la réalisation et la mise au point de l'artéfact à produire. Les règles seront inspirées du *Guide to the SWEBOK.*

Description de la documentation

Nous faisons une description de la documentation à produire afin de respecter les règles de développement et de fournir quelques informations additionnelles utiles.

Les liens entre les artéfacts

Nous décrivons les artéfacts à travers les différentes étapes logiques, et les liens à faire entre eux. Nous espérons que cela guidera les étudiants pour arriver à une solution informatique. Nous décrivons également comment utiliser l'artéfact précédent et comment passer à l'étape suivante du cycle de vie.

Mot sur les grands projets

Nous faisons référence aux aspects de génie logiciel qui ne peuvent être pris en considération dans de petits projets, mais qu'il faudra considérer lors de plus grands projets. Nous voulons sensibiliser l'étudiant au fait que ce qu'il apprend n'est pas perdu, il y aura des certes des ajustements à faire mais l'idée de base sera là.

Exemple de documentation

Nous montrons un (ou des) exemple(s) de document, en utilisant un contexte qui permet d'utiliser les règles de développement et de démontrer les liens entre les artéfacts.

Les exigences

Terminologie

Exigence fonctionnelle : Exigence en terme de capacité. Le programme doit être en mesure (capable) de faire quelque chose. Par exemple : formater du texte, moduler un signal, ajouter une entrée dans une base de données, calculer les racines d'un polynôme, etc.

Exigence non-fonctionnelle : Exigence qui contraint le programme. Par exemple : contraintes de qualité, de performance, de sécurité, etc.

Objectifs

Vous devez trouver les exigences de votre projet à partir de l'énoncé du problème et produire un document que nous appellerons « Spécification des exigences ». Ce document est inspiré du IEEE *Recommended Practice for Software Requirements Specifications*.

Les objectifs du document que vous devez construire ici sont :

- de trouver les exigences fonctionnelles
- de trouver les exigences non-fonctionnelles
- d'exprimer clairement les exigences du programme
- de classer en ordre d'importance les exigences retenues
- d'assurer que les exigences retenues sont : [14]
    - claires et non ambiguës
    - vérifiables
    - complètes et consistantes
    - clairement identifiées (numéro unique)
    - modifiables
    - retraçables

Règles de développement

---

[14] Les définitions complètes de ces termes sont disponibles dans le IEEE *Recommended Practice for Software Requirements Specifications*.

Revoir plusieurs fois l'énoncé du problème pour y retrouver :

- le but principal du projet
- les exigences fonctionnelles
- les exigences non-fonctionnelles
- les validations
    - valeurs limites de nombres (inférieures et supérieures)
    - validation de non-existence d'une valeur avant son ajout
    - validation d'existence d'une valeur avant son retrait
    - autres
- les contraintes particulières[15]
    - les fichiers à utiliser
    - les contraintes sur la casse des caractères en entrée pour un nom, un prénom, etc. (minuscule ou majuscule ou sans importance)
    - le format d'affichage des résultats
    - autres

Ensuite vous devez :

- identifier les exigences avec un numéro unique clairement identifiable (par exemple : EF10, EF20,...)
- mettre les exigences par ordre d'importance
- vous assurer de la véracité des faits provenant des énoncés. Par exemple, s'assurer qu'il n'y pas d'erreurs dans une formule fournie
- vous assurer qu'il n'y a pas de conflits entre les différentes exigences
- comparer vos listes de contraintes et d'exigences avec l'énoncé du problème pour s'assurer de leur validité et s'assurer qu'elles en expriment clairement la demande
- concevoir une liste des dépendances entre les exigences. Cela revient à mettre les exigences principales au premier niveau et mettre les exigences qui en découlent aux niveaux inférieurs et ainsi de suite (voir exemple d'artéfact plus bas)

[15] Habituellement ces contraintes sont décrites dans une section à part dans le document.

Description de la documentation

La documentation décrit les exigences (par catégorie) et leur objectif. Dans notre méthode nous commencerons par la ***définition du projet***. Il s'agit d'être capable de décrire, dans un texte relativement court, ce qu'est le projet à réaliser. Ce texte n'est pas formel et n'est aucunement relié à la solution, mais à la problématique. N'importe qui, autre qu'un programmeur, devrait pouvoir lire et comprendre ce qui doit être fait.

Par la suite concevoir une ***liste des exigences*** classées et numérotées. Il s'agit de faire une liste la plus exhaustive possible des exigences du projet, **par ordre d'importance**. L'importance peut être défini comme étant l'ordre dans lequel vous devrez vous préoccuper des exigences lors de la conception. Pour chaque exigence, nous devons retrouver un numéro **unique** d'exigence, la justification en terme de besoin du client (si nécessaire) et la catégorie (fonctionnelle ou non-fonctionnelle). Si des exigences s'ajoutent en cours de route, il faudra mettre à jour ce document au fur et à mesure que de nouvelles exigences surviendront.

Une exigence décrit le comportement d'un logiciel qui permettra de satisfaire les demandes des utilisateurs et des clients (d'un point de vue externe au logiciel). Vous devrez faire une section spéciale pour les **contraintes particulières** qui ne sont pas explicitement des exigences logiciel (voir exemple plus bas). Il faudra mettre aussi les limites, les contraintes ou les détails importants qui concernent les exigences; faites une section spécifique seulement pour les **validations**.

Les liens entre les artéfacts

Dans la section des exigences, ce n'est **pas comment** respecter toutes les exigences que nous désirons trouver. Nous désirons seulement produire une liste des exigences. En fait, nous voulons simplement savoir **quoi** faire et établir une liste de priorités. Il faut voir les exigences d'un point de vue externe au logiciel (point de vue utilisateur).

La source : L'outil le plus important est l'énoncé du problème. Tout ce qui fait partie de la liste des exigences doit être dans l'énoncé[16].

La documentation : Le document « Spécification des exigences » à produire doit bien décrire toutes les exigences trouvées en suivant les règles de développement. Le document

[16] Lorsque l'étudiant sera en situation de cas réel, il faudra ajuster la source selon le ou les clients.

« dépendance des exigences » illustrera les liens entre les exigences, les validations et les conditions particulières.

L'étape suivante : Lorsque vous saurez ce que vous avez à faire, il faudra commencer à penser **comment** vous allez le faire. Voici un aperçu de l'étape suivante qu'on appelle la conception. Cette étape nous servira à faire une transition entre les exigences et la réalisation.

- Découper les problèmes soulevés, pour satisfaire les exigences, en plus petits problèmes.
- Introduire les stratégies utilisées pour satisfaire aux exigences.
- Décrire la provenance et le type des informations qui sont nécessaires en entrée et qui seront fournies en sortie.

Un mot sur les grands projets

Lors d'un projet en génie logiciel, les exigences représentent environ 25 % de l'effort et 40 % du temps y est consacré. Cette étape est inévitable et essentielle. Lors de grands projets, les exigences à considérer sont plus larges. Il faut définir les modèles, les acteurs, le processus de support, de gestion, de qualité et d'amélioration. Cela permet de faire les spécifications des exigences du logiciel à développer et d'introduire le rôle des différents intervenants. La cueillette de ces informations se fait via des techniques de cueillette d'informations, comme des entrevues, des scénarios, des prototypes, etc. De plus l'étape des exigences sert à découvrir comment le système sera partitionné, en identifiant quelles exigences seront allouées à quels composants. Il faut aussi souvent travailler avec plusieurs équipes. C'est pourquoi la rigueur dans la documentation est primordiale.

Il existe aussi plusieurs méthodes et outils permettant de décrire les exigences (diagramme d'états, diagramme de flux de données, diagramme de flux d'opérations, etc.). Ces outils vous seront utiles lorsque la taille et la complexité du projet seront plus grandes. Ils devraient vous être enseignés plus tard au cours de votre formation.

EXEMPLE 1 :

ÉNONCÉ D'UN PREMIER TRAVAIL PRATIQUE D'UN PREMIER COURS DE PROGRAMMATION.

Vous devez écrire une calculatrice qui permet de faire des opérations sur les nombres rationnels. Les opérations permises sont l'addition, la soustraction, la multiplication et la division.

Le logiciel commence par demander à l'utilisateur d'entrer un nombre entier pour le numérateur, un nombre entier pour le dénominateur et un caractère ('+','-','*','/') pour l'opérateur. Le logiciel valide l'opérateur (tous les autres caractères que ceux permis sont refusés et le logiciel redemande un nouvel opérateur). Un dénominateur ne peut pas être égal à zéro (0). Le logiciel effectue l'opération voulue avec le résultat obtenu de la dernière opération, dans le but de ne pas avoir à entrer de nouveau la fraction qui vient juste d'être calculée. Naturellement, la première fois l'opération se fera avec le chiffre zéro (0). Le logiciel affiche ensuite l'opération et le résultat simplifié au maximum. Lorsque l'utilisateur désire quitter, il entre zéro (0) au numérateur. Votre logiciel affiche alors un message de fin. Les nombres négatifs sont permis pour les entiers, à vous de vous assurer que les résultats respectent les règles des opérations sur les fractions. Si le résultat est zéro (0) vous devez écrire 0 et non 0/0. Même chose pour le dénominateur égal à un (1). Donc on verra 2 et non 2/1 et naturellement partout où cela s'applique.

Si l'utilisateur entre un caractère plutôt qu'un nombre entier, votre logiciel redemandera un nouveau nombre. Vous n'avez pas à valider s'il entre des nombres réels. Si c'est le cas, votre logiciel ne considérera que la partie entière et ne tiendra pas compte de la partie décimale. Par exemple l'utilisateur entre 3.5, votre logiciel considérera que l'utilisateur a entré 3. Vous n'avez pas à vous préoccuper des débordements. Ce qui signifie que si les nombres entrés ou le résultat du calcul donnent un nombre plus grand que le maximum pour les entiers, le logiciel ne se comportera pas comme il se doit.

Lorsque vous avez terminé, n'hésitez pas à vous assurer que vous n'avez pas le même code qui se répète à plusieurs reprises. C'est un indice que ce code peut être amélioré. Un sous-programme (incluant le programme principal) ne devrait pas dépasser 50 lignes de code (incluant aération et commentaires). Si cela arrive, découpez en sous-programme. Mettez une seule instruction par ligne. N'oubliez pas de commenter chaque paramètre formel, chaque

variable et chaque en-tête de fonction. Écrivez du code facile à lire. Commentez à mesure que vous écrivez le code.

Un nombre rationnel est simplifié au maximum lorsque le plus grand commun diviseur (PGCD) = 1. Il faudra trouver le PGCD pour réussir à simplifier une fraction.

Voici l'algorithme d'Euclide pour trouver le PGCD.

On veut le PGCD de N et M

```
Reste := N modulo M    // Reste est de type entier
tantque Reste est différent de 0
    N = M
    M = Reste
    Reste = N modulo M
Fin
Après la boucle M est le PGCD entre N et M
```

***Vous devez remettre la spécification des exigences et la conception avant de commencer le code. La date de remise est fixée au 30 octobre.

EXEMPLE D'ARTÉFACT

Spécification des exigences

Définition du projet

Il s'agit d'écrire une calculatrice qui permet de faire des opérations sur les nombres rationnels et d'afficher les résultats sous forme rationnelle. Les opérations permises sont l'addition, la soustraction, la multiplication et la division.

Liste des exigences fonctionnelles par ordre de priorité

EF10. Le logiciel doit permettre l'addition de fractions.

EF20. Le logiciel doit permettre la soustraction de fractions.

EF30. Le logiciel doit permettre la multiplication de fractions.

EF40. Le logiciel doit permettre la division de fractions.

EF50. Les résultats sont affichés sous forme rationnelle (ex : 7/12).

EF60. Toutes les fractions affichées doivent l'être dans le format le plus simplifié possible. Justification : l'utilisateur doit voir les résultats simplifiés.

Liste des exigences non-fonctionnelles

Aucune

Validation

V10. Le logiciel doit demander un numérateur, un dénominateur et un opérateur dans cet ordre.

V20. Le numérateur et le dénominateur doivent être validés. Justification : ne pas permettre de caractères ou la valeur zéro (0) au dénominateur.

V30. Les opérateurs permis à l'utilisateur doivent être un des caractères suivants : '+',' –',' *' ou '/'. Dans le cas d'une erreur l'opérateur est redemandé. Justification : utiliser les caractères usuels pour ces opérations.

V40. Le dénominateur ne peut pas être zéro. Dans le cas d'une erreur le dénominateur est redemandé. Justification : valeur non définie.

V50. Si l'utilisateur entre un caractère plutôt qu'un entier, le logiciel redemande un nouveau nombre. Justification : respecter le type des données des fractions.

Contraintes particulières

CP10. Il n'est pas nécessaire de se préoccuper des valeurs entrées avec une partie décimale. Si c'est le cas, les résultats pourraient être erronés.

CP20. Il n'est pas nécessaire de se préoccuper des débordements. Si c'est le cas, les résultats pourraient être erronés.

CP50. Le numérateur et le dénominateur peuvent être négatifs en entrés.

CP60. Une opération se fait toujours avec la fraction entrée par l'utilisateur et le dernier résultat calculé. La première fois le dernier résultat calculé est zéro. Justification : pouvoir faire plusieurs opérations consécutives sans avoir à entrer de nouveau la dernière fraction calculée.

CP70. Si le résultat est zéro le logiciel doit afficher 0 et non 0/0.

CP80. Si le dénominateur affiché est 1, il devra être omis par exemple 2/1 donnera 2.

CP90. Le signe d'une fraction résultante sera toujours au numérateur.

CP100. Un sous-programme ne devra pas dépasser 50 lignes de code incluant aération et commentaires.

CP110. Une seule instruction par ligne de code.

CP120. Il faut commenter : en-têtes de sous programme, paramètres formels et variables

Dépendances des exigences

- EF10, EF20, EF30, EF40 Additionner, soustraire, multiplier et diviser une fraction.
    - V10 Demander dans l'ordre le numérateur, le dénominateur et l'opérateur.
        - V20 Valider numérateur et dénominateur.
            - CP10 Ne pas se préoccuper de la partie décimale.
            - CP20 Il n'est pas nécessaire de se préoccuper des débordements.
            - V40 Le dénominateur ne peut pas être zéro.
            - V50 Pas de caractère.
            - CP50 Le numérateur et le dénominateur peuvent être négatifs.
        - V30 Valider l'opérateur.
    - CP60 Utiliser le dernier résultat calculé.
- EF50 Afficher résultat sous forme fractionnaire.
    - EF60 Résultat simplifié.
    - CP20 Ne pas se préoccuper des débordements.
    - CP70 Si le résultat est zéro alors afficher '0' et non '0/0'.
    - CP80 Si le dénominateur affiché est un 1 alors ne pas l'afficher.
    - CP90 Le signe doit être au numérateur.

EXEMPLE 2 :

ÉNONCÉ D'UN DEUXIÈME TRAVAIL PRATIQUE D'UN DEUXIÈME COURS DE PROGRAMMATION.

Il s'agit d'écrire un programme qui permettra l'ajout, la modification, la suppression et l'affichage d'informations sur des véhicules automobiles et leur(s) propriétaire(s). Les informations nécessaires pour les véhicules sont : le constructeur (Ford, Chrysler, etc.), le modèle, la couleur, le numéro de série et l'année. Celles pour les propriétaires sont : le nom, l'adresse complète (numéro civique, rue, ville, code postal) et le numéro de permis de conduire.

Un propriétaire peut avoir plusieurs véhicules, mais un véhicule ne peut avoir qu'un seul propriétaire. Lors de l'affichage d'un propriétaire, le programme doit afficher la liste de ses véhicules. Lors de l'affichage d'un véhicule, le programme doit afficher le nom du propriétaire actuel et de tous ses anciens propriétaires (s'il y a lieu). Le programme doit permettre la recherche d'un véhicule par : numéro de série, numéro de plaque, constructeur, modèle et année. Le programme doit permettre la recherche d'un propriétaire par : nom, numéro de permis de conduire et numéro de plaque d'un véhicule. Il peut avoir plusieurs véhicules ou plusieurs propriétaires qui correspondent à un critère de recherche (à l'exception du numéro de série, du numéro de plaque et du numéro de permis de conduire où la solution sera unique). Le programme doit alors permettre la visualisation des informations correspondant au critère de recherche sur plusieurs pages écran.

Lors de l'ajout d'un propriétaire toutes les informations sont obligatoires et le numéro de permis de conduire ne doit pas déjà exister dans le système. Lors de l'ajout d'un véhicule toutes les informations sont obligatoires et le numéro de série ne doit pas déjà exister dans le système. Le programme doit générer un numéro de plaque minéralogique automatiquement lors de l'ajout d'un véhicule. Ce numéro ne doit pas déjà avoir été attribué. Le numéro de plaque doit avoir trois lettres suivies de trois chiffres.

Le logiciel ne peut pas être accessible à tous les employées, il faudra un mot de passe de 13 caractères exactement (correspond au numéro d'employé) pour pouvoir l'utiliser. Chaque option devra avoir un temps de réponse en deçà de 30 secondes. L'affichage de toutes les informations devra être en minuscules même si elles ont été entrées en majuscules. À

l'exception de la première lettre du nom et prénom des propriétaires et du constructeur du véhicule.

Le logiciel doit être écrit en Ada et fonctionner sous Windows. Il faut compléter et utiliser le module générique d'index dont la partie spécification (.ads) est fournie avec l'énoncé. Le logiciel doit offrir les options sous forme de menus. Le choix entré par l'utilisateur devra être validé. Vous devez remettre les documents de la spécification des exigences et de la conception avant de commencer à écrire le code. La date de remise des documents est fixée à deux semaines après le retour des vacances de Noël.

EXEMPLE D'ARTÉFACT

Spécification des exigences

Définition du projet

Il s'agit d'écrire un programme de gestion des plaques minéralogiques pour la Société de l'Assurance Automobile du Québec (S.A.A.Q). Le programme devra permettre l'ajout, la modification, la suppression et la recherche de propriétaires et de véhicules et générer automatiquement des numéros de plaques minéralogiques uniques pour chaque véhicule.

Liste des exigences fonctionnelles

Le logiciel doit :

EF10. Permettre l'ajout de propriétaires. Justification : l'utilisateur doit pouvoir ajouter les informations sur de nouveaux propriétaires.

EF20. Permettre la modification de propriétaires. Justification : l'utilisateur doit pouvoir corriger les informations d'un propriétaire existant.

EF30. Permettre la suppression de propriétaires. Justification : l'utilisateur doit pouvoir supprimer un propriétaire du système.

EF40. Permettre la recherche d'un propriétaire par : numéro de permis de conduire, nom et numéro de plaque d'un véhicule. Justification : l'utilisateur doit pouvoir retrouver un propriétaire selon plusieurs critères.

EF50. Permettre l'affichage de plusieurs propriétaires correspondant à un critère de recherche. Justification : l'utilisateur doit avoir accès à plusieurs résultats de recherche facilement.

EF60. Rendre accessible à l'utilisateur la liste de tous ses véhicules lorsqu'un propriétaire est trouvé selon un critère de recherche. Justification : permettre à l'utilisateur d'obtenir tous les véhicules d'un propriétaire.

EF70. Permettre l'ajout de véhicules. Justification : l'utilisateur doit pouvoir ajouter les informations sur de nouveaux véhicules.

EF80. Générer automatiquement le numéro de plaque d'un véhicule lors de l'ajout. Justification : assurer l'unicité des numéros.

EF90. Permettre la modification des informations sur les véhicules. Justification : l'utilisateur doit pouvoir modifier les informations sur des véhicules existants.

EF100. Permettre le changement de propriétaire pour un véhicule et garder la trace de tous ses anciens propriétaires. Justification : permettre le suivi des tous les propriétaires d'un véhicule.

EF110. Permettre la suppression de véhicules. Justification : l'utilisateur doit pouvoir enlever un véhicule du système.

EF120. Permettre la recherche de véhicules par : numéro de série, numéro de plaque, constructeur, modèle, année et numéro de permis de conduire d'un propriétaire. Justification : l'utilisateur doit pouvoir retrouver un véhicule selon plusieurs critères.

EF130. Permettre d'afficher plusieurs véhicules correspondant à un critère de recherche. Justification : l'utilisateur doit avoir accès à plusieurs résultats de recherche facilement.

EF140. Rendre accessible à l'utilisateur la liste de tous les propriétaires lorsqu'un véhicule est trouvé selon un critère de recherche. Justification : Permettre à l'utilisateur d'obtenir tous les propriétaires d'un véhicule.

Liste des exigences non-fonctionnelles

ENF10. Le logiciel doit demander un mot de passe pour l'utilisation du logiciel. Justification : l'accès n'est pas autorisé à tous les employés.

ENF20. Le logiciel doit avoir un temps de réponse en moins de 30 secondes. Justification : l'utilisateur ne doit pas attendre trop pour effectuer une opération.

Validation

V10. Le numéro de plaque doit avoir exactement trois lettres suivies de trois chiffres. Justification : respecter la norme concernant les numéros de plaque de la S.A.A.Q.

V20. Le mot de passe pour un utilisateur doit avoir exactement 13 caractères. Justification : correspondre au numéro d'employé.

V30. Il sera impossible d'ajouter un propriétaire déjà existant. Justification : conserver l'intégrité du système.

V40. Il sera impossible de détruire un propriétaire inexistant ou un propriétaire qui a déjà possédé un véhicule. Justification : conserver l'intégrité du système et la liste des propriétaires d'un véhicule.

V50. Il sera impossible de détruire un véhicule inexistant. Justification : conserver l'intégrité du système.

V60. Un véhicule ne peut avoir qu'un seul propriétaire actuel. Justification : avoir un seul propriétaire actuel par véhicule.

V70. Les informations obligatoires pour les propriétaires sont : le nom, l'adresse complète (numéro civique, rue, ville, code postal) et le numéro de permis de conduire. Justification : avoir l'information minimum requise dans le système.

V80. Les informations obligatoires pour les véhicules sont : le constructeur (Ford, Chrysler, etc.), le modèle, la couleur, le numéro de série et l'année. Justification : avoir l'information minimum requise dans le système.

V90. Le choix du menu entré par l'utilisateur doit être validé.

Contraintes particulières

CP10. Le logiciel doit être écrit en Ada et fonctionner sous Windows.

CP20. Il faut compléter et utiliser le module générique d'index dont la partie spécification (.ads) est fournie avec l'énoncé.

CP30. Le logiciel doit offrir les options sous forme de menus.

CP40. L'affichage des informations doit toujours être en minuscules, même si elles ont été saisies en majuscules. À l'exception de la première lettre du nom et prénom des propriétaires et du constructeur du véhicule. Justification : conserver une uniformité pour l'affichage des informations.

Dépendance des exigences

- EF10 Ajout de propriétaires
    - V70 Les informations obligatoires sont : le nom, l'adresse complète (numéro civique, rue, ville, code postal) et le numéro de permis de conduire.
    - V30 Il sera impossible d'ajouter un propriétaire déjà existant (ref : EF20).
- EF20 Modification de propriétaires.
    - V70 Les informations obligatoires sont : le nom, l'adresse complète (numéro civique, rue, ville, code postal) et le numéro de permis de conduire.
- EF30 Suppression de propriétaires.
    - V40 Il sera impossible de détruire un propriétaire inexistant ou un propriétaire qui a déjà possédé un véhicule.
- EF40 Recherche de propriétaires par plusieurs critères de recherche.
    - V10 Le numéro de plaque doit avoir exactement trois lettres suivies de trois chiffres.
    - EF50 Afficher plusieurs propriétaires correspondant à un critère de recherche.
    - EF60 Afficher la liste de tous ses véhicules.
    - CP40 L'affichage des informations doit toujours être en minuscules, même si elles ont été saisies en majuscules. À l'exception de la première lettre du nom et prénom des propriétaires et du constructeur du véhicule.
- EF70 Ajout de véhicules.
    - V60 Un véhicule ne peut avoir qu'un seul propriétaire actuel.
    - V80 Les informations obligatoires pour les véhicules sont : le constructeur (Ford, Chrysler, etc.), le modèle, la couleur, le numéro de série et l'année.
    - EF80 Le numéro de plaque d'un véhicule est généré automatiquement.
        - V10 Le numéro de plaque doit avoir exactement trois lettres suivies de trois chiffres.
- EF90 Modification des informations sur les véhicules.
    - V80 Les informations obligatoires pour les véhicules sont : le constructeur (Ford, Chrysler, etc.), le modèle, la couleur, le numéro de série et l'année.
    - EF100 Changer de propriétaire actuel et garder la trace de l'ancien.
- EF110 Suppression de véhicules.
    - V10 Le numéro de plaque doit avoir exactement trois lettres suivies de trois chiffres.
    - V50 Il sera impossible de détruire un véhicule inexistant.
- EF120 Recherche de véhicules par plusieurs critères de recherche.

- V10 Le numéro de plaque doit avoir exactement trois lettres suivies de trois chiffres.
- EF130 Le logiciel doit permettre d'afficher plusieurs véhicules correspondant à un critère de recherche.
- EF140 Lorsqu'un véhicule est trouvé selon un critère de recherche, la liste de tous les propriétaires l'ayant possédé doit être accessible à l'utilisateur.
- CP40 L'affichage des informations doit toujours être en minuscules, même si elles ont été saisies en majuscules. À l'exception de la première lettre du nom et prénom des propriétaires et du constructeur du véhicule.

- ENF10 mot de passe.
  - V20 Le mot de passe pour un utilisateur doit avoir exactement 13 caractères.
- ENF20 moins de 30 secondes par opération.
- CP10. Le logiciel doit être écrit en Ada et fonctionner sous Windows.
- CP20. Il faut compléter et utiliser le module générique d'index fourni.
- CP30. Le logiciel doit les options sous forme de menus.
  - V90. Le choix du menu entré par l'utilisateur doit être validé.

La conception

Terminologie

Conception : « Processus de définition de l'architecture, des composants, des interfaces et autres caractéristiques d'un système ou d'un composant et le résultat de ce processus » (IEEE 610.12-1990).

Composants : Entités de conception comme des procédures, des fonctions et de modules.

Conception architecturale : Décomposition et description des composants du logiciel.

Conception détaillée : Description de chacun de ces composants suffisamment en détail pour en permettre la traduction en code.

Algorithme : Ensemble des règles opératoires qui permettent la résolution d'un problème par l'application d'un nombre fini d'opérations de calcul à exécuter en séquence (www.granddictionnaire.com).

Objectifs

La conception sert à faire le lien entre les exigences et le code. Il s'agit de décrire le comportement spécifique de composants qui satisferont aux exigences. Ces descriptions seront éventuellement traduites dans un langage de programmation. Nous désirons avoir une description globale et une description plus détaillée qui permettront l'implantation de chaque composant. L'étape de conception, c'est de faire le **plan** qui représente **les décisions qui ont été prises** pour satisfaire aux exigences.

Règles de développement

Il faut faire la conception en deux étapes :

Commencer à concevoir la solution en décrivant la conception architecturale. Cela sert, entre autres, à finaliser la compréhension des exigences à satisfaire (le quoi).

Faire la conception détaillée (le comment).

Conception architecturale :

- Prenez la liste de dépendances des exigences qui se trouvent avec le document « Spécification des exigences ». Vous devriez pouvoir y retrouver l'ordre d'importance, les numéros de référence et les dépendances (liens et données) que vous devrez décrire.
- Prenez chaque exigence fonctionnelle et chaque considération de validation (voir chapitre précédent) pour lesquelles vous devez fournir :

    - Un nom (les règles de choix du nom sont décrites avec le diagramme hiérarchique plus bas).
        - Une description sommaire de ce que doit accomplir le composant.
        - Une liste des informations nécessaires à la réalisation (les entrées) et des informations qui seront fournies (les sorties) lorsque le composant sera implanté.
        - Une liste des erreurs qui peuvent survenir lors de l'exécution. Nous les appellerons des « messages d'exception ».

        NOTE : Décrit en détail plus loin dans le « document sur les interfaces ».

    - Regrouper les composants en modules si cela s'applique (selon le niveau du cours). Les composants peuvent être regroupés pour faire des modules utilitaires (*toolbox*) par exemple les composants utilitaires de mathématiques, d'entrées/sorties pour le clavier et l'écran, d'entrées/sorties pour les fichiers, de validation, etc. Les composants regroupés peuvent aussi décrire des types de données (étudiants, immeubles, autres) ou des types abstraits (pile, file, liste arbre, etc.) que vous avez à implanter.

Conception détaillée :

- décrivez les stratégies en pseudo-langage (décrit plus bas), qui incluent les itérations (boucles), les sélections (si) et les traitements à effectuer pour chaque composant

Note :

- Si un composant traite un problème spécifique, qui ne provient pas directement des exigences, il faudra quand même ajouter son interface et son algorithme dans les documents prévus à cet effet. Ce composant apparaîtra dans un diagramme qui s'appellera « diagramme de la solution » (détail plus bas).
- Les algorithmes de la conception détaillée sont décrits dans un langage qui est différent d'un langage de programmation. Vous ne devez donc jamais faire référence à des mots réservés d'un langage de programmation en particulier.
- S'il y a des modifications au niveau des exigences, des composants, des algorithmes et/ou des interfaces, il faudra mettre à jour tous les documents pour qu'ils reflètent la réalité en tout temps.

Description de la documentation

La documentation se composera de deux diagrammes et de deux documents : Un diagramme pour représenter graphiquement les dépendances (appels et données) entre les exigences. Un document pour décrire les interfaces (nom, comportement, description des entrées/sorties) et les algorithmes, un diagramme qui décrira les décisions sur le découpage en composants (diagramme de la solution) et un document pour le regroupement des composants en modules.

Pour le **diagramme des exigences** nous représenterons les exigences (fonctionnelles et non-fonctionnelles) selon leur ordre d'importance, en les nommant et en illustrant les dépendances (appels et données), les validations, les conditions particulières et les numéros de référence du document « Spécification des exigences »[17]. Le but est d'avoir une vue graphique de l'ordre et des liens entre les **futurs** composants. Si un composant pour satisfaire une exigence décrite ne peut pas se réaliser en peu de ligne de code, vous devez le décomposer en composants plus petits. Refaites cette étape jusqu'à ce que les composants puissent se réaliser en peu de lignes de code. Les composants que vous ajoutez devront se retrouver dans le « diagramme sur la solution » mais ne doivent pas faire partie du diagramme des exigences.

[17] Dans cette méthode le diagramme des exigences est une étape intermédiaire ajoutée pour en arriver au diagramme de la solution. Le diagramme des exigences ne sera plus nécessaire lorsque vous aurez pris de l'expérience.

Pour le **diagramme de la solution** nous illustrerons tous les composants en incluant ceux qui se sont ajoutés et qui ne font pas parties du diagramme sur les exigences. Ce diagramme devra être maintenu à jour pendant l'étape de construction. Le but est d'avoir une image en temps réel des composants.

Pour le **document sur les interfaces et les algorithmes** nous ferons une description du comportement, des entrées/sorties et des exceptions pour chaque élément du diagramme de la solution. Le but est d'avoir une description de la façon d'utiliser le composant. Donc pour chacune des entrées et des sorties, écrivez à quelle catégorie de données elles se rapportent (entier, réel, chaînes de caractères, autres.). **Pour simplifier la description des entrées/sorties, s'il n'y a pas de précisions particulières, nous présumerons que c'est de la catégorie des chaînes de caractères.**

Pour chaque entrée : indiquez quelle est sa provenance (clavier, via paramètres). Pour chaque sortie : indiquez quelle est sa destination (écran, via paramètres). S'il n'y a pas de précisions particulières, nous présumerons que c'est un paramètre.

Les algorithmes décrivent en pseudo-langage les stratégies employées pour chaque interface. Le but est de décrire les stratégies choisies qui seront utilisées pour l'implantation de chaque composant.

Pour le **document sur les modules** nous donnons un nom à chaque module et nous faisons la liste des composants qui en font partie.

Les liens entre les artéfacts

Le but du processus de conception est de nous faire passer du **quoi** vers le **comment**. Le **quoi** est synonyme d'exigences et le **comment** est synonyme de solution. La conception est représentée par les documents de conception architecturale et détaillée. Nous voulons, grâce à ces documents, avoir un **plan** de plusieurs points de vue différents pour la réalisation d'un projet.

Le but de cette documentation est de savoir quels composants sont affectés par un changement que ce soit au niveau de l'interface ou de l'algorithme d'un composant. Par exemple si une interface est changée pour un composant (entrées/sorties ou le nom), le diagramme de la solution nous donnera le nom de tous les composants qui se servent du composant modifié. Il sera plus facile ensuite de modifier le code car vous saurez quels composants sont affectés par

vos modifications. Un autre exemple, si vous cherchez un composant vous pouvez consulter le document sur les modules pour savoir à quel endroit il se trouve dans votre code.

La source : Il faut utiliser le document fait à l'étape précédente qui fournit la spécification des exigences et les dépendances entre elles. Il ne faut pas oublier que ce sont ces exigences que nous désirons satisfaire en premier lieu.

La documentation : Les diagrammes illustreront graphiquement l'ordre d'importance des exigences et leurs liens entre elles (données et appels). Des documents décrivent textuellement les interfaces (les descriptions comportementales des composants, les entrées nécessaires, les sorties et les messages d'exception fournies pour chaque composant) et de façon plus détaillée les algorithmes (en pseudo-langage) qui serviront à implanter chaque composant.

L'étape suivante : La prochaine étape est celle de construction qui décrira des tests unitaires, des tests d'intégration et la traduction des algorithmes dans un langage de programmation. Les tests serviront à faire la vérification des interfaces et des algorithmes pour chaque composant avant d'en faire la traduction.

Note :

- La stratégie de vérification (voir chapitre sur les tests) de l'étape de construction peut être faite simultanément avec la conception d'algorithme. Elle peut même parfois aider à déterminer un algorithme.
- Il est possible de faire tout le cycle de vie sur un composant avant de passer à un autre. Cela signifie que lorsque le diagramme des exigences est terminé, vous faites l'interface, l'algorithme et le diagramme de la solution de ce composant, la stratégie de vérification, la traduction dans un langage de programmation et les tests. Vous pourrez ainsi identifier des fonctionnalités déjà écrites qui pourront vous servir pour répondre à d'autres exigences

Mot sur les grands projets

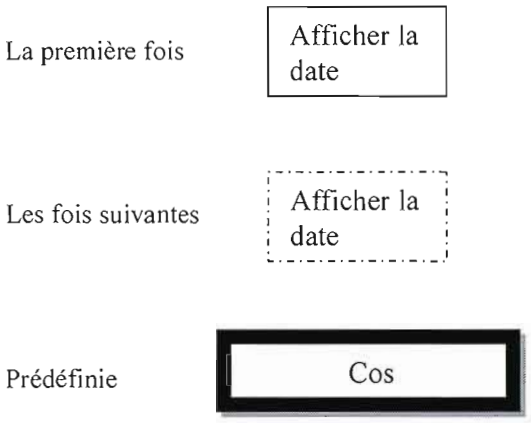
En génie logiciel, la conception sert à décrire (graphiquement ou non) les liens qu'il y a entre les composants, les processus, les interfaces, etc. C'est le plan du système qui sera exprimé de différents points de vue avec plusieurs styles architecturaux, plusieurs stratégies et différentes méthodes selon les stratégies. Ces outils sont importants pour passer du **quoi** vers le **comment**, mais aussi pour obtenir une **vision d'ensemble** de **différents points de vue** des **décisions** qui ont été **prises** pour satisfaire aux exigences.

Il existe un certain nombre d'éléments clés qui doivent être considérés dans la conception du logiciel que nous n'avons pas décrits pour l'étape de conception. Entre autres, des graphiques illustrant le contrôle et la manipulation des flux de données et des documents décrivant l'approche choisie pour interagir avec les utilisateurs, la tolérance aux fautes, la durée de vie des données (persistance), la concurrence des composants et la distribution du logiciel.

Notation pour le diagramme hiérarchique :

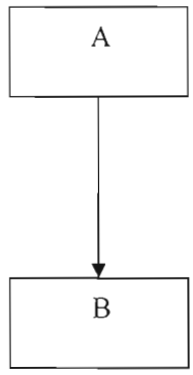
Un composant est représenté par un rectangle avec le nom du composant. Le nom du composant doit refléter son comportement. Habituellement c'est un verbe à l'infinitif suivi d'un mot reflétant ce sur quoi le composant agit (ex : valider la date). Pour les fonctions il est possible de donner un nom représentant la valeur retournée par la fonction (ex : cos, sin, moyenne d'age). Le rectangle est en pointillé si un composant a été défini antérieurement. Le rectangle aura une bordure plus foncée si c'est un composant disponible dans l'environnement de programmation.

Exemple :



Les composants sont reliés par une flèche qui décrit les liens d'appel entre les composants.

Exemple : A appelle B



Si le découpage d'un diagramme devient trop gros, il est possible de poursuivre le diagramme séparément en ajoutant une référence dans un cercle.

Exemple :

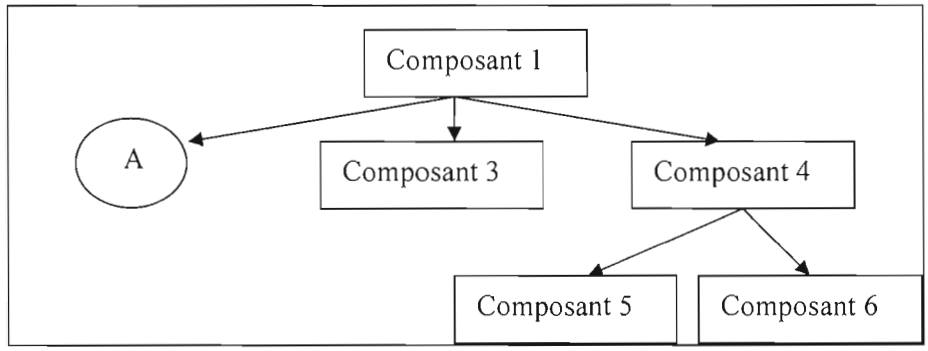


figure E.1 Diagramme hiérarchique

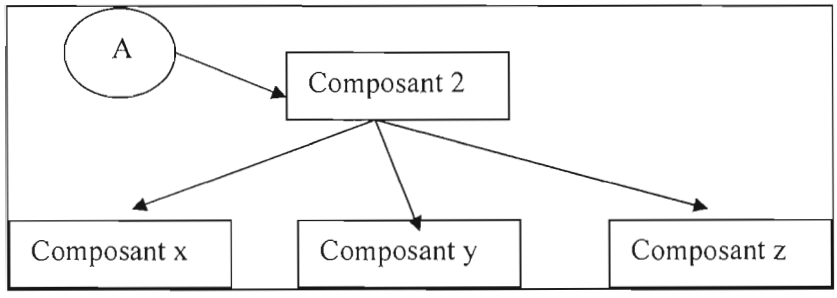


figure E.2 Diagramme hiérarchique description composant A

Les données reçues et fournies en paramètres à un composant sont décrites par des petites flèches reliant les composants et un identificateur des données. Si une information est interne (de contrôle) il faut lui donner un signe différent de celui des données.

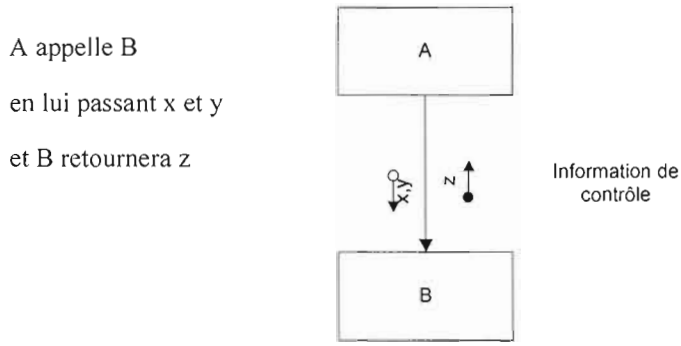


figure E.3 Flux d'appels et de données

Les exemples suivants décrivent une petite partie de la conception du deuxième exemple de l'annexe D.

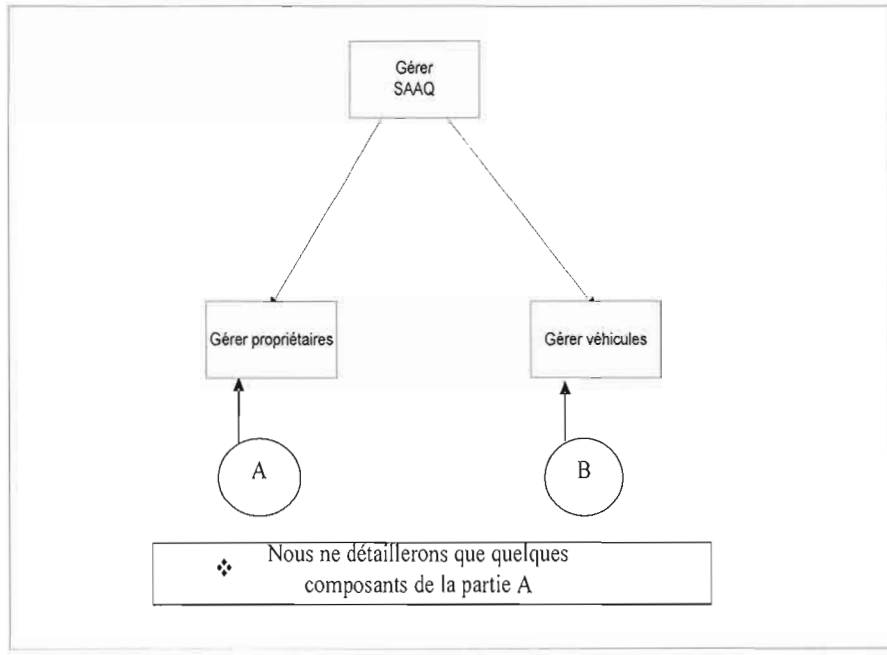


figure E.4 Diagramme des exigences

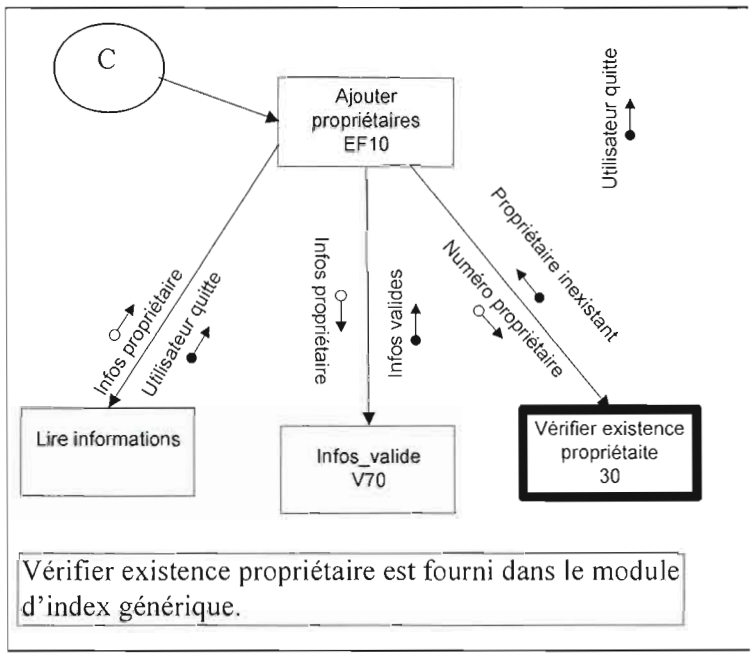


figure E.5 Diagramme des exigences, description du composants C

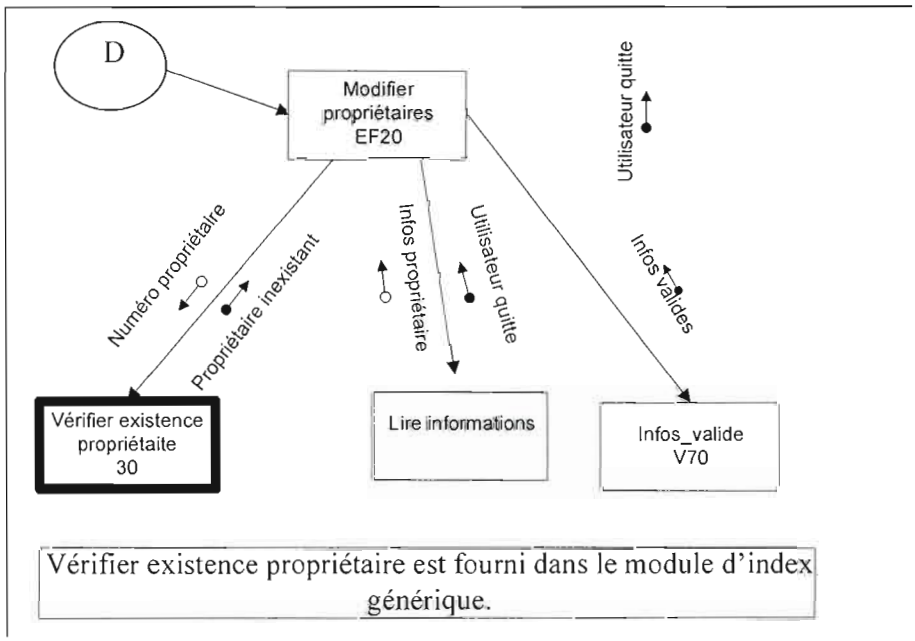


figure E.6 Diagramme des exigences, description du composants D

Ici un exemple devrait être suffisant pour montrer la différence entre la dépendance des exigences (p.90) et le diagramme de la solution. Le diagramme des exigences ne détaille pas ce composant.

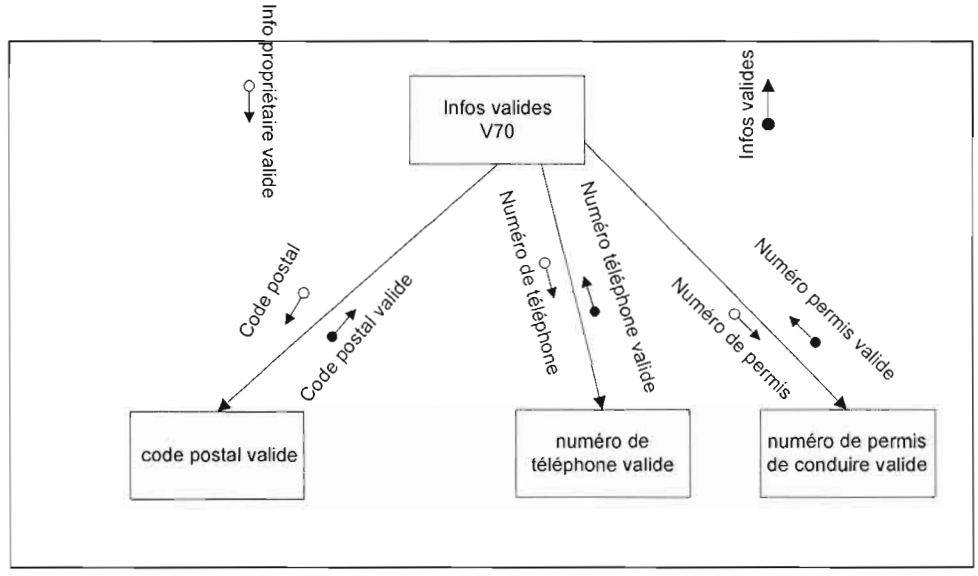


figure E.7 Diagramme de la solution

Exemple de document sur les interfaces et les algorithmes

info sur le propriétaire : nom, adresse complète (numéro civique, rue, ville, code postal) et numéro de permis de conduire.

Nom : Gérer propriétaires

Référence : Sous menu (CP30)

Description : S'occupe de faire afficher le menu et de gérer le choix de l'utilisateur.

Entrées : Aucune

Sorties : Utilisateur quitte (booléen)

Message d'exception : Aucun

Algorithme : Afficher un menu offrant les opérations possibles et, selon le choix de l'utilisateur, faire l'une des opérations suivantes : ajouter un propriétaire, modifier un propriétaire, supprimer un propriétaire, rechercher un propriétaire ou quitter. Se termine seulement si l'utilisateur choisit de quitter.

Nom : Ajouter propriétaires

Référence : EF10

Description : S'occupe de l'ajout d'un propriétaire en faisant valider les informations d'un propriétaire.

Entrées : Aucune

Sorties : Utilisateur quitte (booléen)

Message d'exception : Aucun

Algorithme : Faire valider les infos sur le propriétaire en entrés et faire vérifier si le propriétaire existe déjà. Si c'est le cas, aviser l'utilisateur et redemander un nouveau propriétaire. Si le propriétaire est inexistant il faut faire mettre à jour les données permanentes. Se termine si l'utilisateur quitte ou que le propriétaire est ajouté.

Nom : Valider les informations en entrées

Référence : V70

Description : S'occupe de faire saisir les champs obligatoires et de les faire valider

Entrées : Aucune

Sorties : Utilisateur quitte (booléen)

Info sur le propriétaire valide

Message d'exception : Aucun

Algorithme : Faire saisir les champs obligatoires, faire valider le code postal, le numéro de téléphone et le numéro de permis de conduire du propriétaire. Se termine si l'utilisateur quitte ou que tous les champs sont entrés et valides.

Nom : code postal valide

Référence : Utilitaire

Description : Valide que le code postal a le format **LCL CLC.**

Entrées : Code postal

Sorties : Code postal valide (booléen)

Message d'exception : Aucun

Algorithme :

```
valide = nombre de caractères du code postal = 6

I = 1
Tant que i <= 6  et que c'est valide
        valide = i est impair et le ième caractère est alphabétique ou i est pair et
                 le ième caractère est un chiffre
        i = i + 1
fin

retourne valide
```

Nom : numéro de téléphone valide

Référence : Utilitaire

Description : Valide que le numéro de téléphone a le format **999-999-9999**

Entrées : Numéro de téléphone

Sorties : Numéro de téléphone valide (booléen)

Message d'exception : Aucun

Algorithme : valide = nombre de caractères du numéro de téléphone = 10

```
I = 1
Tant que i <= 10  et que c'est valide
        valide = le ième caractère est un chiffre
        i = i + 1
fin

retourne valide
```

Etc.

Exemple de document sur les modules

Module de validation

- Valider code postal
- Valider numéro de téléphone
- Valider numéro de permis de conduire
- Etc.

Module propriétaire

- Ajouter propriétaire
- Modifier propriétaire
- Supprimer propriétaire
- Rechercher propriétaire

Module d'index générique

- Vérifier existence propriétaire
- Etc.

Etc.

La construction

Objectifs

Voici des objectifs[18] à atteindre lors du codage d'un logiciel et, très sommairement, la façon d'y parvenir :

1. Réduire la complexité du logiciel[19]

    - découper en composants
    - regrouper en modules
        - les composants utilitaires
        - les types de données
        - les types abstraits
        - les problèmes plus complexes.
    - ne pas ajouter au logiciel de fonctionnalités qui ne sont pas décrites dans la conception et qui ne sont pas nécessaires pour satisfaire aux exigences désirées

2. Anticiper les changements

    - généraliser
        - En collectant le plus de données (et de scénarios) liées à l'application.
        - En créant des composants plus génériques qui serviront à traiter des sous-problèmes semblables qui doivent être résolus à plusieurs reprises.
    - isoler ce qui est appelé à changer dans des modules

---

[18] Certains des objectifs ne seront réalisables que lorsque vous aurez suivi un cours de programmation plus avancé. Cependant, rien ne vous empêche d'en avoir quelques-uns en tête lorsque vous commencez à programmer.

[19] Cet objectif est en majeure partie atteint lors de la conception, mais d'autres découpages en composants et d'autres regroupements en modules peuvent survenir lors de la construction.

3. structurer pour la validation
    - programmer en modules[20]
    - utiliser la programmation structurée et le raffinement successif

Idéalement, il faut avoir ces trois grands objectifs **toujours** en tête lors de la conception et de la construction d'un logiciel.

Règles de développement

Voici un peu plus en détails les règles à suivre pour réduire la complexité, anticiper le changement et structurer pour la validation :

1. Poursuivre l'utilisation du raffinement successif. Cette technique est déjà employée pour la conception, mais il faut poursuivre lors de la construction. Par exemple pour valider une date entre 1900 et 2100, il faut valider l'année, valider le mois et ensuite valider le jour. Nous venons de diviser un problème en trois plus petits problèmes. Chacun de ces problèmes pourra être résolu comme les exigences l'ont été lors de la conception (interface et algorithme). L'important est que les sous-problèmes de dernier niveau ne résolvent **qu'un seul et unique problème**. Si ce n'est pas le cas, ils devraient être découpés encore plus.
2. Séparer le code en plusieurs modules pour regrouper les fonctionnalités qui s'occupent des mêmes choses. Par exemple, mettez tous les sous-programmes qui s'occupent de saisie de données au clavier dans le même module, tout ce qui fait référence à une structure de données dans un autre (ex : véhicule, propriétaire, ...), tout ce qui manipule un fichier en particulier, etc.
3. Isoler les problèmes plus complexes dans des modules à part.
4. Écrire et utiliser des types de données abstraits (piles, listes, files, arbres, ...) dans des modules séparés du code de l'application que vous êtes en train de construire. Cela permet de séparer les structures de données utilisées du problème à résoudre et en permet la réutilisation de projet en projet.

[20] Vous pouvez remarquer que la modularité sert aux trois objectifs.

5. Séparer les spécifications des modules de leur implantation (encapsulation), lorsque le langage le permet (ads, adb en Ada, .h et .cpp en C++).

6. Écrire des modules génériques (TEMPLATE en C++, GENERIC en Ada). Les modules génériques permettent également de généraliser en faisant de l'abstraction, mais cette fois-ci avec des types plutôt que des valeurs. Par exemple, les opérations (recherche, tri, insertion, suppression,...) d'un tableau d'entiers et celles d'un tableau de réels sont identiques. Dans un langage qui ne permet pas la généricité, il faut écrire deux fois le même code. Ce qui change, c'est seulement le type de données. Avec la généricité, le code est écrit une seule fois, c'est le type de données qui est passé en paramètre. Lorsque le langage le permet, il est préférable d'utiliser la généricité. Si le langage ne permet pas la généricité, il est parfois possible de la simuler (classe OBJECT en Java). Si c'est le cas, il est valable de le faire.

7. Utiliser les modules déjà existants qui sont fournis avec le langage que vous utilisez. Mieux vous connaissez ces modules, moins vous avez à écrire de code.

8. Généraliser (ou abstraire) plutôt que d'être spécifique. L'algèbre est un bon exemple de généralisation. Nous pouvons dire que $3 + 6 = 9$, que $4 + 8 = 12$ et que $5 + 10 = 15$. Nous pouvons, de façon plus générale, dire que $x + 2x = 3x$. Généraliser, c'est décrire de façon générale tous les scénarios possibles.

9. Utiliser les constantes, les paramètres et les variables dans la construction de vos sous-programmes plutôt que de mettre des valeurs littérales liées à l'application que vous êtes en train de construire.

10. Réutiliser le code lorsque cela est possible. Préférablement, le code est écrit, de prime abord, dans le but d'être réutilisé. Les sous-programmes, les modules (générique ou non) et les types de données abstraits servent à la réutilisation.

11. Utiliser l'approche ascendante pour éliminer le code répétitif dans les cas semblables. Par exemple, si vous avez du code qui se répète avec seulement quelques différences, il est préférable d'écrire une fois le code dans un sous-programme et de passer des paramètres. Cette étape se fait habituellement après avoir écrit les algorithmes.

    Exemple : Reprenons la validation de la date citée au numéro 1. Nous pouvons constater que valider l'année, valider le mois et valider le jour revient à peu près au même, du point de vue de l'algorithme. Voici l'algorithme pour chacun d'eux :

Algorithme pour valider l'année

```
Début
   Saisir une année
   Tant que l'année n'est pas entre 1900 et 2100
         Afficher un message d'erreur
         Saisir une autre année
    Fin boucle
Fin
```

Algorithme pour valider le mois

```
Début
   Saisir un mois
   Tant que le mois n'est pas entre 1 et 12
         Afficher un message d'erreur
         Saisir un autre mois
    Fin boucle
Fin
```

Algorithme pour valider le jour

```
Début
      Saisir un jour
      Tant que le jour n'est pas entre 1 et le nombre de jours maximum
      permis pour un mois
          Afficher un message d'erreur
          Saisir un autre jour
      Fin boucle
 Fin
```

Nous pouvons voir que ces algorithmes se ressemblent beaucoup. Nous voulons généraliser et tenter de trouver un algorithme qui règle les trois cas. Il faut garder les parties d'algorithme qui sont identiques et les parties différentes deviendront des paramètres. Dans cet exemple les paramètres nécessaires sont les bornes et le message d'erreur. Alors voici ce que cela pourrait donner :

Algorithme pour valider un entier

```
Début
      Saisir un entier
      Tant que l'entier n'est pas entre la borne minimale et la borne maximale
           Afficher le message d'erreur reçu
          Saisir un autre entier
       Fin boucle
 Fin
```

Cette approche est ascendante, puisque c'est la création d'un outil plus général qui aidera pour plusieurs parties qui ont été dessinées par l'approche descendante. De plus, cet utilitaire pourra resservir dans d'autres applications qui exigent le même genre de validation. Vous pouvez voir le code de cet utilitaire à la page 113.

12. Ne passez en paramètre que les données nécessaires à un sous-programme. Par exemple si vous avez un enregistrement contenant 15 champs dont un numéro de téléphone et que vous n'avez besoin que de ce dernier dans un sous-programme, ne passez pas tout l'enregistrement mais seulement le numéro de téléphone.

13. Documenter le code.

    - Pour chaque variable, expliquer brièvement à quoi servira cette variable que vous êtes en train de définir.

      Exemple[21] :
      'Sert à calculer le nombre de jours maximum permis pour un mois[22]
      Dim Nbr_jour_max As Integer

    - Pour chaque sous-programme, il faut décrire l'interface soit de décrire le problème à résoudre, les paramètres, les conditions qui doivent êtres respectés avant l'exécution du sous-programme (antécédent) et les changements qu'apportent le sous-programme sur le système après son exécution (conséquent). Il faut aussi mettre les messages d'exception levés par le sous-programme et le numéro de référence aux exigences (ex : EF10) si cela s'applique. Si c'est un sous-programme utilitaire il faut l'indiquer.

      Exemple :

      Procédure d'insertion d'une donnée dans un tableau
      Description : Il s'agit d'insérer en ordre une donnée reçue en paramètre dans un tableau également reçu en paramètre.

      Algorithme : Nous utilisons une fonction pour trouver l'endroit où insérer. Nous décalons les données

[21] Écrit en Visual Basic

[22] L'apostrophe en VB commence un commentaire

d'une case vers la droite et nous insérons la donnée à sa place.

Entrées : La donnée et le tableau
Sorties : Le tableau

Antécédent : Le tableau n'est pas plein et il est ordonné
Conséquent : La donnée est dans le tableau, à sa place, en ordre croissant.

Message d'exception : aucun
Référence : Sous-programme utilitaire

- Pour le corps du programme (les instructions qui implantent l'algorithme), il est préférable d'insérer des commentaires pour indiquer à quelle étape de votre algorithme vous êtes rendu et guider le lecteur de votre code.

  Exemple :

```
'Nous recherchons l'emplacement où insérer la donnée
Endroit = Ou_inserer(LaDonnee, LeTableau)

'Nous décalons les cases vers la droite
For I = (N – 1) to Endroit step -1
    Tableau(I+1) = Tableau(I)
Next I

'Nous insérons la donnée
Tableau(Endroit) = LaDonnee

'Un élément de plus dans le tableau
N = N + 1
```

- Établir des normes de programmation. Comment écrivez-vous le nom de vos constantes (majuscules ou minuscules) ? Comment écrivez-vous le nom de vos variables : première lettre de chaque mot en majuscule, un petit souligné entre chaque mot ? Comment écrivez-vous les mots réservés ? Comment indentez-vous les blocs de code (boucle, sous-programme, ...). Où mettez-vous les commentaires dans le code, avant le bloc, après le bloc,

à coté du bloc ? Le plus important est d'être constant dans vos choix et de suivre les normes imposées par vos enseignants.

Remarque

Le code d'un sous-programme n'est écrit qu'une seule fois. Le reste du temps, ce code est lu et modifié. Pour donner un aperçu, nous pouvons dire que le code est lu des centaines de fois plus souvent qu'il n'est écrit. Il est donc essentiel, pour faciliter le changement, qu'il soit bien écrit dès la première fois. L'erreur la plus fréquente est d'écrire les commentaires, d'indenter et d'aérer après que le code soit terminé. **Si vous commentez au fur et à mesure, les commentaires vous guideront tout au long du développement de vos programmes**, ce qui ne peut pas être le cas si vous ajoutez les commentaires à la fin.

Description de la documentation

La documentation que vous aurez pour l'étape de construction dans cette méthode se résume au listage du programme **commenté** et d'un plan visuel du code. Pour le plan, nous avons le diagramme de la solution, qui est une image des composants qui ont été réalisés et des liens qui les unissent, et un document sur l'organisation des composants en modules. Ces diagrammes ont été initialement faits lors de la conception et maintenus à jour pendant l'étape de construction.

Liens entre les artéfacts

L'acte de construction est fondamental. Nous désirons traduire en code tous les algorithmes qui ont été faits lors de l'étape de conception dans le but d'obtenir une solution exécutable pour satisfaire aux exigences du logiciel. Vous avez abordé ce qu'il y avait à faire à l'étape des exigences, vous avez dessiné comment vous le feriez à l'étape de conception maintenant il faut le réaliser. Il faut suivre le plan fait lors de la conception. Vous avez pris la peine de décrire ce qu'il y avait à faire, maintenant il faut le faire. La construction se termine lorsque les algorithmes sont traduits dans un langage de programmation et qu'il existe une stratégie de vérification pour chaque composant[23].

[23] Les tests sont abordés dans le chapitre suivant.

La source : La conception est la principale source pour l'implantation. Il faut suivre l'ordre d'importance et les algorithmes qui ont été choisis. Il est fort possible que des composants s'ajoutent lors de cette étape. Si c'est le cas il faut les aborder de la même façon que l'ont été les autres composants. Il faudra alors maintenir à jour les documents sur les interfaces et les algorithmes et celui sur les modules. Il faudra aussi ajouter ces composants au diagramme de la solution.

La documentation : Le code est documenté le plus souvent sous forme de commentaires qui y sont incorporés. Nous avons également un plan de l'organisation de la solution et des modules grâce aux documents faits lors de la conception et maintenus à jour lors de la construction.

L'étape suivante : La prochaine étape est celle des tests. Lorsqu'un composant est codé, il faut le tester et lorsqu'il est testé il faut l'intégrer avec les autres composants déjà testés.

Mot sur les grands projets

Les objectifs de la construction en génie logiciel sont plus larges que ceux qui vous ont été présentés. Il faut prévoir, dans cette phase à améliorer la productivité et la qualité du système, à évaluer le logiciel, à sélectionner les standards de développement du logiciel, à choisir les langages de programmation, de configuration et les techniques de construction (manuelles ou automatisées). Tout cela peut se faire en utilisant des outils existants ou en produisant soi-même ses outils. Il faut penser à utiliser des normes externes pour permettre de partager des données avec le monde extérieur. Dans de grands projets il est aussi recommandé d'utiliser une méthode objet avec un ensemble de méthodes complètes et suffisantes, d'utiliser des fichiers de configuration et de faire des logiciels auto-descriptifs (*plug and play*) pour rencontrer les objectifs qui vous ont été présentés dans cette méthode. Cet exemple est écrit en Ada.

```
procedure Valider_Entier (Msg_Sollicitation : In  String ;
                          Borne_Min         : In  Integer ;
                          Borne_Max         : In  Integer ;
                          Msg_Erreur        : In  String ;
                          Zero_Permis       : In  Boolean ;
                          Entier            : Out Integer);
procedure Valider_Entier (Msg_Sollicitation : In  String ;

                          Borne_Min         : In  Integer ;
                          Borne_Max         : In  Integer ;
                          Msg_Erreur        : In  String ;
                          Zero_Permis       : In  Boolean ;
                          Entier            : Out Integer);
```

--Description: Il s'agit de valider un entier entré au clavier qui doit être entre deux bornes ou
-- égal à 0.
--
--Entrées : **Msg_Sollic** est le message de sollicitation à afficher.
-- **Borne_Min, Borne_Max** sont les limites acceptables pour l'entier.
-- **Zero_Permis** est vrai si la valeur 0 est permise et faux sinon.
-- **Msg_Erreur** est le message d'erreur à utiliser si l'entier n'est pas entre les bornes.

--Sorties : **Entier** est la valeur à retourner.
--
--Antécédent : aucun.
--Conséquent : borne_min <= entier >= borne_max ou entier = 0
--Message d'exception : data_error si la valeur entrée est de type caractère.
--
--Mise en garde : Seule la partie entière d'un nombre réel entré sera prise en compte
-- exemple (3.5 retournera 3)
--
--Exemple d'appel :
--
-- valider_entier(« Entrez votre age (0 pour annuler) », 1,120, « age invalide », true,age);
--
--Référence : Sous-programme utilitaire

```
BEGIN --Valider_Entier
     LOOP
         --affichage du message de sollicitation
         Put(Msg_Sollic);

         --saisie de l'entier au clavier
         Es_Entiers.Get(Entier);
         Ada.Text_IO.Skip_Line;

         --vérification de la validité de l'entier
         EXIT WHEN Entier IN Borne_Min..Borne_Max OR
                   (Entier = 0 AND Zero_Permis);

         --affichage du message d'erreur
         Put(Msg_Erreur);

      END LOOP;
END Valider_Entier ;
```

Les tests

Objectifs

Pour faire la sélection de bons tests, il faut avoir en tête les objectifs que l'on veut atteindre. Voici une liste de quelques objectifs possibles :

- Tester (ou réviser) toutes les étapes du processus de développement.
- Exposer les erreurs.
- Déterminer les tests cibles et les objectifs des tests pour les différentes étapes du processus.
- Assurer la qualité du logiciel.

Pour y parvenir nous vous suggérons de faire une stratégie de vérification lors de la conception des algorithmes. Cette stratégie doit permettre de vérifier que les exigences sont satisfaites, que les interactions entre les différents sous-programmes se font correctement, que les données se transmettent correctement entre chaque sous-programme et finalement que la vérification des erreurs prévisibles sera faite. Cette stratégie de vérification sera mise en application lors de l'étape de construction et de test. Nous ferons deux sortes de tests : les tests unitaires et les tests systèmes. Les tests systèmes nous serviront à tester l'intégration des composants développés lors de la construction et à tester si les exigences (fonctionnelles et non-fonctionnelles) sont satisfaites.

Règles de développement

- sélectionner des critères de tests. Cela qui consiste à décider quels critères permettront de sélectionner les ensembles de tests que doit passer le logiciel. Dans le cas de petits projets, nous pouvons considérer les tests qui vérifient si les exigences sont satisfaites (conformité) par rapport à la spécification des exigences, que les erreurs prévisibles ont été testées (fiabilité) et que les sous-programmes s'intègrent sans erreur (validité). Voici la description de ce qu'il faut faire pour y parvenir :

- parcourir les fonctionnalités implantées et s'assurer que les entrées/sorties sont valides et que les interactions entre les fonctionnalités respectent le diagramme hiérarchique. Il faut baser les tests sur les spécifications des exigences et sur la recherche d'erreurs. Cette méthode de tests est appelée technique de la boîte noire (*black box*).

- évaluer la structure du code implantant les fonctionnalités. Il faut baser les tests sur la détection d'erreurs dans le code. Cette méthode de tests est appelée technique de la boîte blanche (*white box*).
- identifier les défectuosités possibles et prévisibles (ajout d'un client existant, retrait d'un client inexistant, ouverture d'un fichier inexistant, etc.). Il est préférable, voir même indispensable, de penser à des tests pour trouver des erreurs qu'à des tests qui prouvent le bon fonctionnement du logiciel.
- prévoir des résultats attendus concrets pour l'acceptation ou le refus d'un test. Il faudra vérifier toutes les contraintes du document « Spécifications des exigences » et faire référence aux numéros des exigences testées.

Les liens à faire entre les artéfacts

Les tests sont comme le logiciel et il faut les planifier. Idéalement, il faut prévoir l'objectif et le résultat attendu d'un test. Il faut tester le logiciel du point de vue des exigences, du code et de la recherche d'erreurs.

La source : Le document « Spécifications des exigences », le diagramme de la solution et le code sont les principales sources des tests. Lorsque la révision du code est effectuée (voir chapitre sur la construction) il faut s'assurer que les exigences sont satisfaites et que la solution respecte les relations du diagramme hiérarchique de la solution. Il faut aussi vérifier **au moins une fois** que toutes les lignes de code d'un composant ont été exécutées, que chaque condition des sélections et des boucles réagissent correctement, que tous les scénarios distincts possibles de l'algorithme d'un composant ont été envisagés et que toutes les utilisations possibles du logiciel ont été testées.

La documentation : Le document sur la stratégie de vérification sert à décrire les tests qui devront être effectués. Il faudra un autre document qui décrit les tests lorsqu'ils ont été effectués, afin de pouvoir comparer les résultats attendus à ceux qui ont été obtenus lors du test réel. Ce dernier est habituellement fait lors de l'exécution du logiciel et il existe des outils pour en permettre la production automatiquement.

L'étape suivante : Dans un premier temps, l'erreur que l'on détecte se trouve dans le code. Il faut toujours identifier la portion de code qui produit l'erreur. Cela peut être causé par une exigence mal énoncée (5% du temps), un oubli dans le diagramme hiérarchique de la solution ou une erreur de logique dans le code (50% du temps), une erreur dans la stratégie de vérification, dans le test final (11% du temps), dans la documentation (19% du temps) ou d'autres (15%). Peu importe, il faudra réparer l'erreur à la source et refaire les étapes du cycle

de vie à partir de l'endroit où le problème a été détecté. **Il est très important est de maintenir la documentation à jour.**

Mot sur les grands projets

En génie logiciel, des tests sont prévus pour différents types d'application et de programmation. Il existe plusieurs autres techniques de tests que celles que nous vous avons présentées. Par exemple des tests de régression devraient être faits à chaque fois qu'il y a une modification pour tester de nouveau le système ou les composants, des tests Alpha/Beta qui nécessitent des utilisateurs internes (employés) et externes (clients), des tests d'installation et des tests d'utilisabilité.

Pour vérifier que les exigences ont été satisfaites il existe aussi des tests en séparant par classes d'équivalence, en utilisant des tables de décision, des automates finis, des spécifications formelles et des tests générés aléatoirement. Plusieurs diagrammes peuvent aussi être utilisés lorsque les tests sont basés sur le code (diagramme de décisions, diagramme de flux de données et révisions formelles). D'autres techniques sont utilisées lorsque les tests sont basés sur la recherche d'erreurs (Prévisions d'erreurs (*error guessing*), mutations).

Finalement des normes existent pour le développement de tests unitaires (*IEEE Standard for Software Unit Testing*) et pour la documentation des différentes techniques de tests (*IEEE Standard for Software Test Documentation*). En résumé, les tests ne sont pas une mince tâche et il ne faut pas les prendre à la légère.

Description de la documentation

Préférablement, vous devez décrire votre stratégie de vérification en fournissant la description du test, son objectif, quelles sont les exigences qui seront satisfaites, les valeurs concrètes que vous utiliserez pour effectuer les tests ainsi que les résultats attendus (lorsque cela s'applique). Cette stratégie devrait être faite simultanément avec les algorithmes lors de l'étape de conception.

Dans cette méthode nous diviserons la stratégie de vérification en trois sections :

- tests de satisfaction des exigences fonctionnelles
- tests pour la détection d'erreurs
- tests système (cela inclut les tests d'intégration et les tests sur les exigences non-fonctionnelles)

Les tests d'intégration peuvent se faire avec plusieurs approches :

- approche descendante (*top-down*) : L'implantation des composants se fait par la technique de l'échafaudage (*stub*). Un composant fait appel à d'autres composants dont seulement l'interface a été codée avec une valeur de retour bidon ce qui permet de faire les tests voulus. Le corps de ces composants sera implanté plus tard. Un avantage est que des sections du logiciel peuvent être opérationnelles sans que le logiciel soit complété. Un désavantage est qu'il est possible d'avoir besoin d'écrire plusieurs interfaces avant de pouvoir tester un composant.
- approche ascendante (*bottom-up*) : Les composants de plus bas niveaux (utilitaires) sont codés et testés individuellement. Ensuite les composants de plus hauts niveaux sont codés en utilisant les composants complétés. Un avantage est que les composants de bas niveau sont codés et réutilisables dès le départ. Un désavantage est que les composants les plus importants (ceux qui satisfont les exigences) sont faits et testés en dernier.
- approche système (*big bang integration*) : Tous les composants sont testés individuellement et l'intégration se fait d'un seul coup (possible seulement dans de petits projets). Cette approche est déconseillée même si elle est encore parfois utilisée.
- approche par couches (*sandwich integration*). C'est un mélange de l'approche descendante et de l'approche ascendante. Le système est vu en trois couches (comme un sandwich). L'approche descendante est utilisée pour les composants au dessus et l'approche ascendante pour les composants en dessous. Cette approche combine les avantages des deux autres approches. **C'est cette approche que nous préconisons dans la méthode.**

Pour les tests, la description se fera de la façon suivante :

Test (numéroter le test)

But :

Démarche :

Comportement ou résultats attendus :

Références :

Exemple de documentation :

Stratégie de vérification

Tests pour les exigences fonctionnelles

Ces tests vérifient que les exigences fonctionnelles sont satisfaites.

Test 1 : Vérifier que l'année acceptée est seulement entre 1900 et 2100 inclusivement.
But : S'assurer que le programme redemande une année si l'entrée est invalide.

Démarche :Entrer les valeurs 1899, 1900, 2000, 2100, 2101.

Comportement ou résultats attendus : 1900, 2000 et 2100 sont valides et 1899, 2101 sont invalides.

Références : V8 (numéro hypothétique).

Test 2 : Vérifier que le mois accepté est seulement entre 1 et 12 inclusivement.
But : S'assurer que le programme redemande un mois si l'entrée est invalide.

Démarche : Entrer les valeurs 1, 12, 0, 13, 5.

Comportement ou résultats attendus : 1,5, 12 sont valides et 0, 13 sont invalides.

Références : V9 (numéro hypothétique).

***note : Le code qui permettra de valider l'année est possiblement dans une procédure utilitaire qui s'appelle valider_entier de même que pour la validation du mois et du jour. Cela n'empêchera pas de faire les tests fonctionnels pour le mois et le jour quand même. Cela permettra de vérifier si l'appel de la procédure utilitaire a été fait correctement.

Test 8 : Quitter au début de l'application.

But : S'assurer que le programme se termine correctement.

Démarche : Entrer 'Q' au menu principal.

Comportement ou résultats attendus : Le programme se termine.

Références : Utilitaire.

Test 18 : Entrer différents choix au menu principal.

But : S'assurer que la validation des choix s'effectue correctement.

Démarche : Entrer 0, 3, 8, 1, 7.

Comportement ou résultats attendus : 1,7 et 8 sont valides et 0,3 sont invalides.

Références : EF9, V10 (numéro hypothétique).

Exemple d'une stratégie de vérification moins intéressante

Test : Entrer différents choix au menu principal.

But : S'assurer que la validation des choix s'effectue correctement.

Démarche : Entrer des valeurs valides et invalides pour tester.

Lors de ce test il n'y aura aucune comparaison possible entre la stratégie de vérification et le test puisqu'il n'y a pas de valeurs concrètes décrites.

Truc :

- Lorsque vous faites la stratégie de vérification, imaginez que ce sera une autre personne qui fera les tests et indiquez quelles valeurs vous aimeriez qu'elle entre et quels sont les résultats attendus.
- Dans le cas d'une vérification de valeurs **consécutives** (numériques ou caractères), cinq valeurs suffisent. Les deux valeurs limites (1900 et 2100 dans notre exemple), les deux premières valeurs hors bornes (1899 et 2101) et une valeur entre les bornes (2000). Naturellement, ceci ne tient pas compte des erreurs d'incompatibilité de type. Ce genre d'erreur se gère différemment selon le langage de programmation utilisé.
- Pensez toujours à vérifier les cas extrêmes. Par exemple, un fichier vide pour la manipulation de fichier. Une structure pleine (pile, file, tableau,...) lors de l'insertion de données, une structure vide lors de suppression ou une recherche de données inexistantes, etc.

Tests pour la détection d'erreurs

Ces tests tentent de trouver des erreurs prévisibles.

Test 36 : Entrer le même numéro de plaque à deux propriétaires différents.

But : S'assurer que le programme rejette deux fois le même numéro de plaque.

Démarche : Ajouter un véhicule et laisser le système attribuer un numéro de plaque. Prendre ce numéro en note et ajouter deux propriétaires avec ce numéro de plaque en référence.

Comportement ou résultats attendus : Un message avise que le numéro de plaque a déjà été attribué.

Références : V123 (numéro hypothétique)

Tests système :

Tests pour les exigences non-fonctionnelles :

Test 80 : Vérifier que l'insertion dans la liste ordonnée se fait en O(n).

But : S'assurer que le programme n'insère pas en plus de O(n).

Démarche : Insérer 100 valeurs dans la liste et vérifier qu'il n'y a pas plus de 100 éléments qui sont visités lors de la recherche de l'emplacement.

Comportement ou résultats attendus : Écrire un programme test qui compte le nombre d'éléments parcourus dans la liste lors d'une insertion.

Références : ENF110 (numéro hypothétique)

Tests d'intégration

- commencez par l'approche descendante pour le premier niveau du diagramme de la solution (en écrivant les interfaces de chaque sous-composant). Ce sont habituellement les composants qui satisfont les exigences fonctionnelles. La descente peut se terminer lorsque les composants du dernier niveau sont atteints.

- écrivez les composants de plus bas niveau en les intégrant par l'approche ascendante. C'est à ce moment que les tests unitaires peuvent être faits sur les composants complétés du premier au dernier niveau.

- recommencez la même chose pour tous les composants du premier niveau en faisant les tests d'intégration finaux.

- faites des tests unitaires pour chaque opération et testez l'intégration avec un programme test, indépendant du logiciel, afin de tester des modules implantant une structure de données (pile, file, liste, arbre, ...).

INSÉRER TOUT LE CODE QUI A SERVI AUX TESTS D'INTÉGRATION DANS VOTRE DOCUMENTATION.

CONCLUSION

Nous avons vu dans cette méthode les étapes importantes du développement de logiciels. Nous vous avons suggéré des documents à produire qui reflèteront les décisions que vous avez prises. Ce que nous espérons est que vous utiliserez cette documentation non seulement pour décrire ces décisions mais bien pour vous aider à les prendre. Nous croyons que si vous prenez la peine de faire ces documents dans l'ordre avec les préoccupations que nous vous avons mentionnées, vous serez en mesure de développer de meilleurs logiciels et cela de plus en plus rapidement. De plus vous aurez une vision de certaines considérations que vous aurez à prendre lors de la réalisation de grands projets.

ANNEXE D

PREMIÈRE VERSION DE LA MÉTHODE

Nous mettons ici une version de la méthode qui avait été annotée spécialement pour les évaluateurs participants. La version finale de la méthode est une version modifiée suite aux commentaires des membres du jury à partir de la version annotée. La version annotée est présentée ici avec les erreurs qu'elle contenait.

Note au lecteur

La méthode qui suit s'adresse à des étudiants commençant leur formation en programmation. Ce que nous espérons développer chez les étudiants est l'utilisation d'un plan de travail pour le développement de petits logiciels, mais aussi l'utilisation de ce plan comme outil de résolution de problèmes. Un autre objectif est d'habituer les étudiants à ce qui les attendra en génie logiciel lors de la réalisation de plus grands projets. Nous prétendons que l'on peut développer de petits projets de façon similaire à de plus grands projets. Nous prétendons aussi que cette méthode pourrait aussi aider les enseignants à donner ces cours de base, puisque les enseignants auront un modèle des notions importantes qui devraient y être enseignées.

Chacune des décisions prises pour la construction de cette méthode ont été prises en prenant en considération ce qui est écrit dans le guide au corpus de connaissances en génie logiciel (abran et al., 2001) et respecte des principes fondamentaux du génie logiciel (Dupuis et al., 1999). Nous vous suggérons de lire la méthode suivante comme si vous étiez un enseignant qui doit donner ces cours ou bien comme un étudiant qui doit apprendre à utiliser cette méthode. Ce dernier est un peu plus difficile, puisqu'il vous demande d'oublier des connaissances que vous avez déjà probablement acquises. Par la suite, vous aurez des petits cadres appelés **note au lecteur.** Ces notes serviront à vous communiquer la provenance et les objectifs des éléments qui sont proposés dans la méthode.

Vous retrouvez, dans les titres des pages suivantes, les grands objectifs que veut atteindre cette méthode :

1. Offrir une méthode de développement.
2. Utiliser cette méthode comme outil de résolution de problèmes.
3. Montrer un lien avec de plus grands projets.
4. Aider dans la préparation des cours de programmation de base, pour les enseignants de ces cours.

Bonne lecture!

INTRODUCTION

Il y a plusieurs façons de concevoir un logiciel, mais il y a peu de méthodes provenant de recherches rigoureuses qui expliquent les étapes de développement et les liens à faire entre chacune des étapes.

La méthode suivante utilise des principes fondamentaux en génie logiciel définis par une équipe de chercheurs de l'Université du Québec à Montréal (Dupuis et al., 1999). De plus, cette méthode est construite sur des bases de faits reconnus dans le monde du génie logiciel, provenant du corpus de connaissances en génie logiciel (Abran et al., 2001). Le but de cette méthode est d'apprendre à développer de petits projets, avec une perspective de grands projets.

Finalement, nous offrons une explication des liens à faire avec chaque étape du développement, le but étant ici de permettre d'utiliser la méthode de développement comme méthode de résolution de problèmes.

L'approche suggérée est d'être conscient, lors du développement du logiciel, des points suivants :

Connaissance des objectifs précis de l'étape actuelle et comment y parvenir.

Connaissance de l'étape qui vient avant (s'il y a lieu).

Connaissance de l'étape qui vient après.

Comment passer d'une étape à l'autre.

Connaissance de la documentation à produire pour chaque étape.

Cette méthode n'est pas un cours théorique de programmation. C'est une liste des choses à faire pour agencer des concepts théoriques appris dans les cours de programmation, pour chacune des étapes du cycle de vie d'un logiciel. L'étape de maintenance, elle, commence lorsque le logiciel est chez le client. C'est pour cette raison que nous ne décrirons rien concernant la maintenance dans ce document.

Modèle en cascade (« waterfall ») :

Le cycle de vie d'un logiciel est constitué d'étapes (exigences, conception, construction, intégration et tests, maintenance) qu'on doit franchir pour développer un programme informatique. Il existe plusieurs modèles du cycle de vie. Cette méthode utilise le modèle en cascade, car c'est celui que nous considérons le plus approprié pour la programmation à petite échelle.

Différentes approches (descendante et ascendante) :

Idéalement, dans le modèle en cascade, nous utilisons *l'approche descendante*, où chaque problème à résoudre est subdivisé en plus petits problèmes jusqu'à atteindre une solution. Chaque étape est complétée avant de passer à la suivante. Dans la réalité, il faut souvent revenir sur une étape précédente, lors de la détection d'erreurs dans le cycle. Dans l'approche descendante, on divise le logiciel en plusieurs parties et un cycle de vie est effectué sur chacune de ces parties, ce qui facilite les étapes de tests et de maintenance.

Il est possible de créer de petits utilitaires qui aident dans la résolution d'un problème. Cette approche est *l'approche ascendante*. Nous utiliserons les deux approches dans cette méthode.

DOCUMENTATION

Pour chaque étape du cycle de vie, il y aura un document à produire. Cette méthode décrit de quelle façon ces documents doivent être construits. Elle montre quelles sont les règles à suivre pour arriver à produire les documents de chaque étape du cycle de vie et comment utiliser ce document pour passer à l'étape suivante du cycle de vie.

DESCRIPTION SOMMAIRE DE LA MÉTHODE

Nous ferons autant de documents qu'il y a d'étapes du cycle de vie. Dans chaque document, nous avons fait une introduction. Par la suite, nous avons divisé le document en sections où chacune :

Décrit la terminologie employée pour la réalisation de l'artéfact[24] à produire.

Explique les objectifs à atteindre.

Décrit comment produire la documentation associée à l'étape du cycle de vie (en respectant des règles de développement).

Décrit comment faire les liens entre les étapes en utilisant le document précédent et le suivant (lorsque cela s'applique).

Donne de petits exemples illustrant la documentation à produire.

[24] Artéfact : c'est un produit réalisé au cours d'une étape. La documentation, le code, les tests et la version exécutable d'un programme sont tous des artéfacts.

DESCRIPTION DÉTAILLÉE DE LA MÉTHODE

Terminologie

Nous ferons la définition des termes employés dans la construction de l'artéfact à produire, si nous croyons que cela nécessite des précisions.

Objectifs

Dans cette section, nous définirons les objectifs que doit atteindre l'artéfact à produire. Par exemple, l'artéfact sur les exigences sert, entre autres, à différencier les exigences fonctionnelles des exigences non-fonctionnelles.

Règles de développement

Cette section sert à fournir un ensemble des règles permettant de parvenir à la création, la réalisation et la mise au point de l'artéfact à produire. Les règles seront puisées dans le guide au corpus de connaissances (Abran et al., 2001).

Résolution de problèmes

Dans cette section, nous décrirons les étapes logiques et les liens qui sont nécessaires pour parvenir à suivre les règles et les principes de développement et arriver à une solution informatique. Nous décrirons brièvement comment utiliser l'artéfact précédent et comment passer à l'étape suivante du cycle de vie.

Mot sur le génie logiciel

Nous ferons référence aux aspects de génie logiciel qui ne peuvent être pris en considération dans de petits projets, mais qu'il faudra considérer lors de plus grands projets.

Description de la documentation

Nous ferons une description des sections à mettre dans la documentation, pour respecter les règles de développement qui auront été données, et pour fournir quelques informations additionnelles.

Exemple de la documentation

Nous montrerons un ou plusieurs exemples de document, en utilisant un contexte qui permet d'utiliser les règles de développement et les liens de la résolution de problèmes.

Note au lecteur

Le but, ici, est de montrer aux étudiants que ce qu'ils apprendront dans cette méthode est simplifié et qu'ils auront à élargir leur approche en génie logiciel.

Description de la documentation

Nous ferons une description des sections à mettre dans la documentation, pour respecter les règles de développement qui auront été données et pour fournir quelques informations additionnelles.

Exemple de la documentation

Nous montrerons un ou plusieurs exemples de document, en utilisant un contexte qui permet d'utiliser les règles de développement et les liens de la résolution de problèmes.

Note au lecteur

Nous avons utilisé les définitions suivantes :

Principe : Proposition admise comme base de raisonnement (Larousse, 1999).

Méthode : Ensemble ordonné de manière logique de principes, de règles, d'étapes permettant de parvenir à un résultat (Larousse, 1999).

Résolution : Logique pour enchaîner des règles. Si A est vrai et si on sait que A entraîne B alors B est vrai. Techniquement, procédure par laquelle on déduit une nouvelle clause en résolvant deux clauses (Office de la langue française, 1995).

Développement : Création, réalisation et mise au point d'un programme, d'un logiciel, d'une application ou d'un système (Office de la langue française, 2001) .

En se basant sur ces définitions, nous pouvons dire qu'une méthode de développement est :

Un ensemble ordonné de manière logique de propositions admises comme base de raisonnement, de règles, d'étapes permettant de parvenir à ***la création, la réalisation et la mise au point d'un programme, d'un logiciel, d'une application ou d'un système.***

Nous pouvons dire aussi qu'une méthode de résolution de problèmes est :

Un ensemble ordonné de manière logique de propositions admises comme base de raisonnement, de règles, d'étapes permettant de parvenir à une logique pour ***enchaîner des règles*** *pour* ***résoudre des problèmes.***

Note au lecteur concernant les exigences

Définition d'exigences :

1) « Propriété de ce qui doit être exposé, dans l'ordre, pour résoudre un problème du monde réel.» (Abran et al., 2001, chap. 2.1; Pfleeger 1998; Kotonya et Sommerville, 2000; Sommerville 2001; Thayer et Dorfman, 1997).

2) Condition ou capacité nécessaire à un utilisateur pour résoudre un problème ou atteindre un objectif (Davis, 1993; IEEE 830-1983).

Le guide au corpus de connaissances nous a servi à utiliser ce qui est actuellement reconnu en génie logiciel et qui est **utilisable** dans le cadre de petits projets. Par exemple, il est mentionné que les exigences servent à découvrir comment le système sera partitionné, en identifiant quelles exigences seront allouées à quelles composants. Dans le cas de programmation à petite échelle, cela ne s'applique pas.

Entre autres, il est écrit qu'une exigence se doit d'être clairement identifiée. Alors, dans nos règles de développement, nous avons inclus qu'une exigence doit avoir un numéro unique.

Nous avons retenu que les exigences doivent être **reliées au problème** (non à la solution) et qu'il existe déjà une **notion de séquence** dans les exigences. Il faut aussi définir que **l'ordre de priorité** des exigences retenues est important dans un projet.

Principes fondamentaux du génie logiciel:

- Séparer en plusieurs perspectives ce que doit faire le logiciel.
- L'artéfact à produire est défini dans la méthode.
- L'ajout et la modification d'exigences et de spécifications sont prévus et facilement applicables.
- Faire la liste des exigences et établir leur priorité, afin de choisir lesquelles doivent être implantées en premier. Cela permettra de produire le logiciel par étapes.
- Parce que l'artéfact à ce niveau offre la possibilité de faire la gestion de configurations.

LES EXIGENCES

Terminologie

Exigence fonctionnelle: Ce sont des exigences en terme de capacité. Le programme doit être en mesure (capable) de faire quelque chose. Par exemple : formater du texte, moduler un signal, ajouter une entrée dans une base de données, calculer les racines d'un polynôme, etc.

Exigence non-fonctionnelle: Ce sont des exigences qui contraignent le programme. Par exemple : des contraintes de qualité, de performance, de sécurité, etc.

Objectifs

Vous devez retirer de l'énoncé des travaux pratiques, les exigences de vos enseignants. S'il y a des clarifications nécessaires, vous devrez suivre le processus de cueillette d'informations prévu par vos enseignants. Les objectifs du document que vous devez construire ici sont :

De classifier et de séparer les exigences fonctionnelles des non-fonctionnelles.

De définir l'ordre de priorité des exigences retenues.

D'assurer que les exigences retenues sont :

- claires et non-ambiguës.
- vérifiables.
- complètes.
- consistantes.
- clairement identifiées (numéro unique).
- modifiables.
- retraçables.

D'exprimer clairement les besoins et les spécifications du programme.

Note au lecteur

D'après le guide au corpus de connaissances, il existe plusieurs façons de classer les exigences.

Certaines des autres possibilités sont :

- Par processus.

- Par buts.
- Par contraintes.
- Volatiles vs stables.
- Autres (selon le contexte).

Nous avons choisi le classement fonctionnel et non fonctionnel, car il est plus facilement utilisable dans de petits projets, puisque ce sont les deux grandes catégories et qu'elles regroupent les autres.

Règles de développement

- Lire à plusieurs reprises l'énoncé du travail à effectuer pour y retrouver :

Le but principal du projet.

Les exigences fonctionnelles.

Les exigences non-fonctionnelles.

Les spécifications de validation

  - valeurs limites de nombres (inférieures et supérieures).
  - validation de non-existence d'une valeur avant son ajout.
  - validation d'existence d'une valeur avant son retrait.
  - etc.

Les spécifications de l'enseignant[25]

  - fichiers à utiliser.
  - documents à remettre.
  - date de remise.
  - longueur maximum des chaînes de caractères.
  - contrainte sur la casse des caractères en entrée pour un nom, un prénom, etc. (minuscule ou majuscule ou sans importance).
  - etc.

- Classer par ordre de priorité les exigences.
- Identifier les exigences avec un numéro unique clairement identifiable (par exemple : E1, E2,...).
- Identifier les spécifications avec un numéro unique clairement identifiable (par exemple: S1, S2,...).
- Associer les spécifications avec les exigences.

[25] C'est une catégorie de spécifications, mais ce sont des spécifications au même titre que les autres.

- S'assurer de la véracité des faits provenant des énoncés. Par exemple, s'assurer qu'il n'y pas d'erreurs dans une formule fournie.
- S'assurer qu'il n'y a pas de conflits entre les différentes exigences et les différentes spécifications.
- Comparer vos listes de contraintes et d'exigences avec l'énoncé pour s'assurer de leur validité et s'assurer qu'elles en expriment clairement les besoins.

Résolution de problèmes

Dans la section des exigences, ce n'est pas **comment** nous respecterons toutes les exigences que nous désirons trouver. Nous désirons seulement en faire ressortir une liste des exigences et des spécifications. En fait, nous voulons simplement savoir **quoi** faire et établir une liste de priorités.

La source : L'outil le plus important est l'énoncé du travail pratique. Tout ce qui fait partie de la liste des exigences et de la liste des spécifications doit être dans l'énoncé[26].

La documentation : Le document à produire doit bien décrire toutes les exigences trouvées en suivant les règles de développement.

L'étape suivante : Lorsque vous saurez ce que vous avez à faire, il faudra commencer à penser **comment** vous allez le faire. Voici un aperçu de l'étape suivante qu'on appelle la conception. Cette étape nous servira à faire une transition entre les exigences et la réalisation.

Illustrer à l'aide d'un graphique l'ordre dans lequel les exigences et les spécifications seront réalisées.

Découper les exigences les plus grosses en plus petites exigences.

Découper les spécifications les plus grosses en plus petites spécifications.

Introduire les stratégies utilisées pour répondre aux exigences et aux spécifications.

Décrire la provenance et le type des informations nécessaires en entrées et qui seront fournies en sortie.

[26] Lorsque l'étudiant sera en situation de cas réel, il faudra ajuster la source selon le ou les clients.

Note au lecteur

Un problème identifié par D. Bell (Bell, 1987) est que les étudiants ont tendance à voir les étapes du cycle de vie de façon distincte et les approchent séquentiellement. Nous avons détecté le même problème et c'est ce qui justifie ce rappel sur les objectifs actuels, la provenance et la prochaine étape. C'est la partie résolution de problèmes que nous avons incluse dans la méthode de développement.

Mot sur le génie logiciel

Lors d'un projet en génie logiciel, les exigences représentent environ 25 % de l'effort et 40 % du temps y est consacré. Cette étape est inévitable et est essentielle. Lors de grands projets, les exigences à considérer sont plus larges. Il faut définir les modèles, les acteurs, le processus de support, de gestion, de qualité et d'amélioration. Ceci permet de faire les spécifications du logiciel à développer et d'introduire le rôle des différents intervenants. La cueillette de ces informations se fait via des techniques de cueillette d'informations, comme des entrevues, des scénarios, des prototypes, etc. Il faut aussi souvent travailler avec plusieurs équipes. C'est pourquoi la rigueur dans la documentation est si importante.

Note au lecteur

Ici vous avez la liste de ce qui est énuméré dans SWEBOK et qui doit être pris en compte pour de plus gros projets. Nous croyons que d'introduire les termes et une vision de plus grands projets permettra une meilleure transition en génie logiciel.

Description de la documentation

La documentation se décomposera en deux parties. La partie décrivant les exigences et la partie spécifiant ces exigences. C'est dans le ***Document sur les exigences*** qu'il y aura la ***définition du projet***. Il s'agit d'être capable de décrire, dans un texte relativement court, ce qu'est le projet à réaliser. Ce texte n'est pas formel et n'est aucunement relié à la solution, mais à la problématique. N'importe qui, autre qu'un programmeur, devrait pouvoir lire et comprendre ce qui doit être fait. Par la suite, il faut faire une ***liste des exigences*** classées et numérotées. Il s'agit de faire une liste la plus exhaustive possible des exigences du projet, par ordre de priorité. Pour chaque exigence, nous devons retrouver un numéro d'exigence, l'objectif en terme de besoin du client et la catégorie (fonctionnelle ou non-fonctionnelle). Si des exigences s'ajoutent en cours de route, il faudra mettre à jour ce document au fur et à mesure que de nouvelles exigences surviendront.

C'est dans le ***Document sur les spécifications*** qu'apparaîtra la liste des spécifications fournissant les précisions sur les exigences. Elles aussi seront numérotées. La spécification

devra fournir une description et le numéro d'exigence à laquelle elle se rapporte. Une spécification décrit le comportement d'un logiciel qui permettra de satisfaire les exigences des utilisateurs et clients. Par exemple une limite, une contrainte ou un détail important qui concerne des exigences. Cela se résume souvent à ce que le programme devrait valider et/ou devrait considérer. Une validation implique souvent la saisie, l'analyse et l'avertissement d'erreur (optionnel) pour une valeur et cela répétitivement jusqu'à ce que la valeur respecte les contraintes imposées par la spécification.

Truc : Écrire votre document immédiatement dans un logiciel de traitement de texte, cela est plus facile à modifier par la suite.

Note au lecteur

Diviser le document en deux devrait permettre aux étudiants de voir une différence entre une exigence et une précision sur une exigence. De plus, il est écrit dans SWEBOK (Abran et al., 2001) que l'artéfact sur les exigences doit contenir absolument un descriptif du but du projet, ce que nous avons inclus dans la méthode.

Exemple d'artéfact

Document sur les exigences

Définition

Il s'agit d'écrire un programme de gestion des plaques minéralogiques de la Société de l'Assurance Automobile du Québec (S.A.A.Q). Le programme devra permettre l'ajout, la modification, la suppression et la recherche de propriétaires de véhicules et générer automatiquement des numéros de plaques minéralogiques uniques pour chaque véhicule.

Liste des exigences fonctionnelles

E1. Le logiciel doit permettre l'opération d'ajout de véhicules. L'utilisateur devra pouvoir ajouter les informations sur de nouveaux véhicules.

E2. Le logiciel doit permettre l'opération d'ajout de propriétaires. L'utilisateur devra pouvoir ajouter les informations sur de nouveaux propriétaires.

E3. Le logiciel doit permettre d'associer un véhicule à son propriétaire et vice versa. L'utilisateur devra pouvoir, lors d'un ajout de véhicule ou de propriétaire, faire une association entre les deux.

...

E19. Le logiciel doit permettre la suppression d'un propriétaire.

...

Liste des exigences non-fonctionnelles

...

E412. Le programme devra permettre de spécifier les utilisateurs permis pour l'utilisation du logiciel, pour chaque option. L'accès aux options du logiciel doit être sécurisé selon le département.

...

Document sur les spécifications

S1. Le numéro du propriétaire doit être généré automatiquement par le logiciel.

S2. Le numéro de plaque doit être généré automatiquement par le logiciel.

S3. Le numéro de plaque généré doit être unique.

S4. Le numéro de plaque minéralogique entrés par un utilisateur doit avoir exactement trois caractères suivi de trois chiffres (ref : E1, E2,...).

S5. Il sera impossible d'ajouter un propriétaire déjà existant (ref : E1).

S6. Il sera impossible de détruire un propriétaire inexistant. (ref : E19).

S7. Les informations sur un véhicule sont les suivantes : la marque, le modèle, la couleur et le numéro de série.

S8. Les informations sur un propriétaire sont les suivantes : le nom, l'adresse complète, le numéro de téléphone et le numéro de permis de conduire.

S9. Une référence à un numéro de plaque ou à une référence d'un propriétaire, entrée au clavier, doit toujours être validée.

...

S20. Le numéro d'accès pour un utilisateur ne doit pas dépasser 13 caractères (ref : E412).

...

Note au lecteur

Nous retrouvons dans cet artéfact des directives habituelles du génie logiciel en ce qui concerne les exigences. Ce qui nous importe le plus est le cheminement fait pour y parvenir. Nous espérons que l'étudiant verra bien la provenance des exigences et ce qu'il doit en faire.

Note au lecteur concernant la conception

Les systèmes doivent être décrits et documentés selon différentes vues (Abran et al., 2001, chap. 3 ; 3:III). La vue des spécifications (*logical view*), la vue des processus (*process view*), la vue physique (*physical view*) et la vue du développement (*development view*). Pour ce travail, nous n'avons pris que la vue des spécifications et du développement. Il existe également différents styles architecturaux et tout un processus de vérification de la qualité de la conception. Nous n'avons rien concernant ces styles, puisqu'ils ne s'appliquent pas

Le résultat du processus de conception est un ensemble de modèles et d'artéfacts qui décrivent les décisions majeures qui auront été prises. (Abran et al., 2001, chap. 3; Budgen, 1994 ; IEEE 1016-1998 ; Liskov et Guttag, 2001 ; Pressman, 1997). Dans le domaine du génie logiciel, la conception est importante pour l'abstraction, le couplage et la cohésion, la décomposition et la modularité, l'encapsulation, la séparation des interfaces de leur implantation et la complétude du système (Abran et al., 2001 ; chap. 3:3). René Descartes, dans son *Discours de la méthode*, utilisait déjà le découpage de problèmes en de plus petits problèmes et cela reste encore reconnu dans le monde du logiciel, que le problème soit petit ou grand. C'est pourquoi nous avons considéré la décomposition et la modularité dans la conception de petits logiciels.

Nous retrouverons dans notre artéfact sur la conception des directives habituelles à de petits projets. Ce document sera probablement le plus difficile à produire pour les étudiants, puisqu'ils sont à découvrir un langage de programmation en même temps. Le plus important est que les étudiants puissent comprendre que nous passons du QUOI vers le COMMENT et que la conception est l'outil qui est utilisé pour y parvenir, indépendamment du langage de programmation utilisé.

Nous n'avons pas senti le besoin de changer l'aspect architectural déjà employé dans les cours de programmation. Les modèles permettant entre autres d'illustrer les relations entre les fonctionnalités sont le diagramme hiérarchique, le structurogramme détaillé ou l'organigramme. Nous avons choisi le diagramme hiérarchique parce que ce modèle nous permettra de représenter l'ordre dans lequel les fonctionnalités seront développées. En plus, il permet de représenter les exigences fonctionnelles avec un découpage qui représentera aussi les spécifications.

PRINCIPES FONDAMENTAUX

- Séparer les problèmes en plus petits problèmes permet à l'étudiant de réutiliser des solutions à des problèmes déjà résolus.

- Séparer la conception en plusieurs documents offre plusieurs perspectives.
- Les artéfacts à produire sont définis dans la méthode.
- La documentation permet les ajustements assez facilement; seulement quatre étapes pour effectuer les modifications.
- Les artéfacts à mettre à jour offrent un processus rigoureux pour les modifications aux exigences.
- Les artéfacts à produire forment les étapes à réaliser.
- Parce que l'artéfact à ce niveau offre la possibilité de faire la gestion de configurations.

LA CONCEPTION

Terminologie

Conception : Processus de définition de l'architecture, des composants, des interfaces et autres caractéristiques d'un système ou d'un composant et le résultat de ce processus » (IEEE 610.12-1990)

Objectifs

La conception sert à faire le lien entre les exigences et le code. Il s'agit de décrire le comportement spécifique des composants système qui par leur comportement rempliront les exigences et les spécifications, pour éventuellement être traduit dans un langage de programmation. Nous désirons avoir une stratégie globale pour répondre à chaque exigence et une ou plusieurs stratégies plus détaillées, ce qui permettra la réalisation du projet. La conception, c'est le **plan** qui représente **les décisions qui ont été prises** pour répondre aux exigences et aux spécifications.

Règles de développement

Il faut faire la conception en deux étapes :

- Finaliser la compréhension des exigences et des spécifications à remplir (le quoi)
- Commencer à concevoir la solution (le comment)

Pour finaliser la compréhension des exigences et des spécifications à remplir, vous devez :

1) Prendre les exigences qui se trouvent dans le document des exigences et les spécifications, et illustrer à l'aide d'un diagramme l'ordre de priorité et les relations. Ce diagramme peut avoir plusieurs formes, le plus important est que l'ordre y soit clairement indiqué. Vous devez mettre la référence aux numéros d'exigence et de spécification.

2) Pour chaque fonctionnalité (par ordre de priorité), décrire textuellement (dans un document séparé) la stratégie globale qui sera employée pour y répondre. En programmation, les stratégies utilisées s'appellent des *algorithmes*.

3) Pour chaque exigence, faire la liste des informations nécessaires à sa réalisation (les *entrées)* et quelles informations seront fournies lorsque l'exigence sera réalisée (les *sorties*). Pour chacune des informations, dire quelle est sa catégorie de données (entier, réel, chaînes de caractères, ...). S'il n'y a pas de spécifications particulières, nous présumerons que c'est de la catégorie des chaînes de caractères.

4) Pour chaque entrée : indiquer quelle est la provenance (clavier, fichier, paramètres).

5) Pour chaque sortie : indiquer quelle est la destination (écran, fichier, paramètres).

Note au lecteur

Encore une fois, ce sont des directives habituelles pour la réalisation de petits projets. Ici l'apport est que nous insistons encore pour que les étudiants séparent le quoi du comment.

Pour commencer à concevoir la solution, vous devez :

1) À la suite des entrées/sorties de chaque exigence, décrire la stratégie en pseudo-code en introduisant les itérations (boucle) et les sélections (si).

2) Considérer chaque étape de la stratégie spécifique d'une exigence comme une sous-exigence ou une spécification.

3) Faire une stratégie globale, une liste des entrées et des sorties et le pseudo-code pour chacune des sous-exigences et des spécifications.

4) Si une sous-exigence ou une spécification le nécessite, écrire une fonctionnalité qui règle des problèmes spécifiques. Une fonctionnalité sera traitée comme une sous-exigence. Il faudra donc écrire les algorithmes et les entrées/sorties.

Note : La conception est indépendant du langage de programmation employé. Vous ne devez donc jamais faire référence à des mots réservés d'un langage de programmation.

Note au lecteur

Les stratégies employées pour résoudre un problème font partie de la conception. C'est dans cette partie que nous faisons la transition entre les exigences et la solution.

Nous sommes conscients que ce n'est pas une mince tâche que d'assimiler toutes ces directives théoriques et leur sens réel. Nous espérons que l'exemple d'artéfact viendra concrétiser tout cela. N'oublions pas que le but est aussi de préparer les étudiants à de plus gros projets.

Résolution de problèmes

Nous voulons avoir un **plan** de travail pour la réalisation d'un projet. Le but est de nous faire passer du **quoi** vers le **comment**. Le quoi correspond aux exigences et aux spécifications. Le comment est la stratégie que nous allons employer pour répondre à ces exigences et ces spécifications en séparant les exigences en sous-exigences et en fonctionnalités.

La source : Il faut utiliser le document fait dans l'étape précédente qui fournit la liste des exigences et la liste des spécifications.

La documentation : Le document illustrera graphiquement l'ordre de priorité des exigences et le lien avec les spécifications. Il décrit textuellement les stratégies globales qui seront employées pour chaque exigence. Il décrit également les entrées nécessaires et les sorties fournies pour chaque exigence, sous-exigence et/ou fonctionnalité. Il décrit de façon plus détaillée (en pseudo-code) l'algorithme qui servira à réaliser chaque fonctionnalité.

L'étape suivante : L'étape suivante sera de préparer une stratégie de vérification des algorithmes pour chaque fonctionnalité et de faire la traduction des algorithmes dans un langage de programmation.

Note :

1) La stratégie de vérification peut être faite simultanément avec la conception d'algorithme. Elle peut même parfois aider à déterminer un algorithme.

2) Il est possible de faire tout le cycle de vie sur une exigence avant de passer à une autre. Ceci signifie que lorsque le diagramme des exigences est terminé, vous faites la stratégie globale, les entrées/sorties, le pseudo-code, le diagramme de la solution de cette exigence, la stratégie de vérification, la traduction dans un langage de programmation et les tests. Vous pourrez ainsi détecter des fonctionnalités déjà écrites qui pourront vous servir pour répondre à d'autres exigences.

Mot sur le génie logiciel

Il existe un certain nombre d'événements clés qui doivent être considérés dans la conception du logiciel. Entre autres, le contrôle et la manipulation des flux de données, l'approche choisie pour interagir avec les utilisateurs, la tolérance aux fautes, la durée de vie des données

(persistance), la concurrence des composants, la distribution du logiciel, etc. (Abran et al., 2001 ; chap. 3 :3:II).

En génie logiciel, la conception sert aussi à décrire graphiquement les liens qu'il y a entre les composants, les processus, les interfaces, etc. C'est un outil important pour passer du **quoi** vers le **comment**, mais aussi pour obtenir une **vision d'ensemble** des **décisions** qui ont été **prises** pour répondre aux exigences. C'est le plan du système qui sera exprimé de différents points de vue.

Note au lecteur

Dans le cadre d'un cours de programmation, il était possible de présenter une architecture des fonctionnalités et des flux de données sans plus. Les interactions entre systèmes et composants sont inexistantes dans ces cours, c'est pourquoi nous n'en avons parlé que brièvement dans le mot sur le génie logiciel.

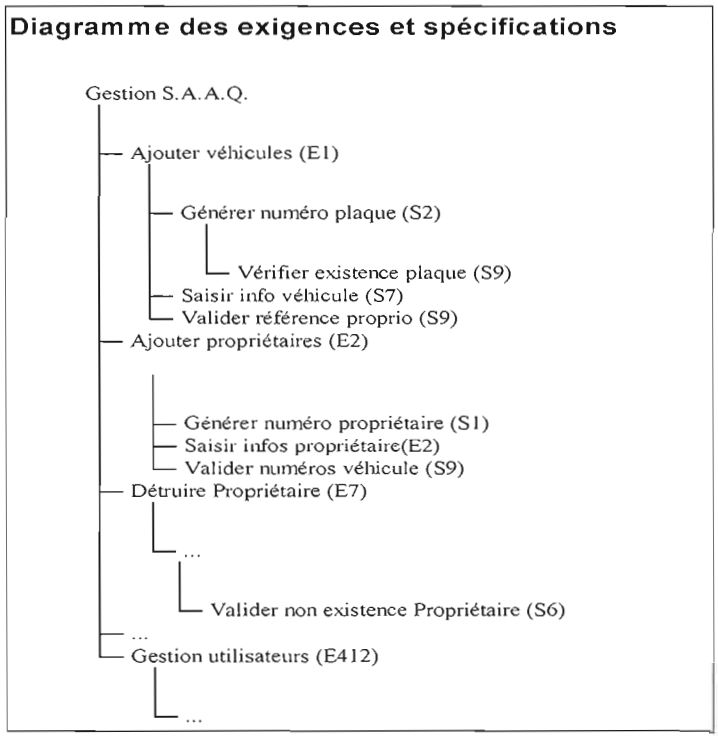
Description de la documentation

La documentation se décomposera en plusieurs parties. Un diagramme pour représenter les exigences et les spécifications. Une partie sur les stratégies globales, une partie sur les entrées/sorties, une partie sur les algorithmes, et un deuxième diagramme qui représentera les décisions sur le découpage en sous-exigences et en fonctionnalités.

Pour le premier diagramme, que nous appellerons ***diagramme des exigences et spécifications,*** nous représenterons les exigences (fonctionnelles ou non), selon leur priorité. Les niveaux inférieurs représenteront les spécifications de chaque exigence. Chaque élément aura le numéro de l'exigence ou de la spécification à laquelle il fait référence. Nous ferons également une partie sur ***les algorithmes***. Cette section fera référence aux exigences et décrira textuellement la stratégie globale pour satisfaire l'exigence. Dans cette section, il n'y a pas de pseudo-code, que des phrases explicatives de la solution envisagée et cela pour chacune des exigences et des spécifications (numéro de référence inclus).

Une section servira à décrire les entrées/sorties de chaque fonctionnalité, la catégorie de données de cette information (entier, réel, chaîne de caractères et booléen), ainsi que la source ou la destination (écran, clavier, fichier,...). Il va de soit que cette partie doit être faite à partir des diagrammes et des listes des exigences et des spécifications. Une partie contiendra les algorithmes en pseudo-code de chaque fonctionnalité qui sert à répondre aux exigences et aux spécifications.

Exemple d'artéfact



À noter :

- Le diagramme hiérarchique précédent a été construit à partir des documents sur les exigences et les spécifications faits au chapitre précédent.
- La lecture d'un tel diagramme se fait de haut en bas, pour l'ordre, et de gauche à droite, pour les dépendances. Il existe des variantes acceptables.
- Les exigences les plus en haut sont les plus prioritaires.

Note au lecteur

Le diagramme hiérarchique est connu et fortement employé dans les cours de programmation. La variante que nous voulons proposer et celle que nous croyons préférable est de faire deux diagrammes. Un premier diagramme pour représenter les exigences et les spécifications qui provient des artéfacts précédents (exigences et spécifications) et un second qui représentera le découpage en sous-exigences et en fonctionnalités, qui représente la solution. Ce dernier est une image du découpage qui sera fait dans le code. Le but est que les étudiants voient mieux la

différence entre les exigences et la solution.

Document sur les algorithmes

Ajouter des véhicules (E1)

Il s'agit de générer un numéro de plaque, de saisir les informations sur un véhicule, de saisir le numéro de référence du propriétaire du véhicule et de vérifier son existence. S'il est inexistant, un message sera affiché et l'utilisateur devra entrer un autre numéro de référence. S'il existe, alors les données seront mises dans un fichier binaire.

Entrées : marque, modèle, numéro de série, couleur d'un véhicule (clavier)

No du propriétaire (clavier)

Sorties : No de plaque généré (écran, fichier texte)

Marque, modèle, numéro de série, couleur d'un véhicule et

référence sur le proprio (fichier binaire)

Début

Tant que l'utilisateur n'a pas terminé

Générer un no de plaque (S2)

Saisir au clavier marque, modèle, couleur et no série (S7)

Valider existence no référence propriétaire(S5)

Si l'utilisateur n'a pas annulé alors

Insérer données dans fichier binaire

Fin si

Fin boucle

Fin

Valider existence no référence propriétaire (S9)

Il s'agit ici de valider que le numéro de propriétaire fourni existe dans le système avant de lui attribuer un véhicule. Tant que le numéro n'existera pas, l'utilisateur aura la possibilité d'en entrer un nouveau ou d'annuler.

Entrées : no propriétaire (clavier)

Sorties : infos véhicule + référence proprio (fichier binaire)

Début

Tant que le no de propriétaire n'existe pas et que l'utilisateur n'a pas annulé

Saisir no de propriétaire

Vérifier existence du propriétaire (S9)

Si le no existe

Ref_proprio du véhicule = no de propriétaire (E4)

Sinon

Aviser que le no est invalide

Fin si

Fin boucle

Fin

Vérifier existence du propriétaire

Nous traverserons le fichier binaire contenant les informations sur les propriétaires pour s'assurer que le numéro fourni existe. Si nous atteignons la fin du fichier, c'est que le numéro est inexistant.

Entrées : No de propriétaire (via paramètre)

Num proprio (fichier binaire)

Sorties : Existe ou pas de type booléen (via paramètre)

Début

Tant que le no de propriétaire fourni est différent du num proprio dans le fichier et

que ce n'est pas la fin du fichier

Lire un autre no de propriétaire dans le fichier

Fin boucle

Le proprio n'existe pas, si c'est la fin de fichier; sinon, il existe.

Fin

Générer numéro de plaque (S2)

Il faut générer 3 caractères suivis de 3 chiffres aléatoirement et tant que ce numéro n'a pas déjà été attribué. Pour pouvoir vérifier l'existence des numéros, nous mettrons les nouveaux numéros générés dans un fichier texte. La vérification se fera à partir de ce fichier.

Entrées : Un fichier texte contenant les no

Sorties : Un nouveau no de plaque qui sera mis dans un fichier texte

Début

Tant que le no généré existe dans le fichier

Générer trois caractères et trois chiffres aléatoirement (S4)

Vérifier l'existence du no dans le fichier texte (S3)

Si le no n'existe pas

L'insérer dans le fichier

Fin si

Fin boucle

Fin

Vérifier l'existence numéro plaque(S3)

Nous traverserons le fichier tant que le numéro cherché n'a pas été trouvé ou que la fin de fichier est atteinte. Si c'est le cas, c'est que le numéro n'existe pas.

Entrées : Le no nouvellement généré

Sorties : Existe ou non de type booléen

Début

Tant que le no généré est différent du no dans le fichier texte et

que ce n'est pas la fin du fichier

Lire un no dans le fichier

Fin boucle

Si c'est la fin du fichier, alors

Le no n'existe pas

Sinon

Le no existe

Fin si

Fin

...

Note au lecteur

Pour les entrées/sorties, nous n'avons pas pris de modèle de flux de données conventionnel, puisque beaucoup trop long à produire pour le peu de données à traiter dans le genre de cours qui nous intéresse. Nous avons choisi de prendre un modèle plus littéral qui ne fait que décrire les entrées et sorties en texte. Ce document ne décrit que les données, en entrées et en sorties, nécessaires à chaque fonctionnalité, ainsi que leur type, leur provenance (clavier, fichier, ...) ou leur destination (écran, fichier, ...). Cette méthode est déjà employée à l'Université du Québec à Montréal, dans un cours d'introduction à la programmation, et nous semble suffisante.

Note au lecteur concernant la construction

Il existe trois styles de construction d'après le guide au corpus de connaissances (Abran et al., 2001, chap 4; 3.2). Le style linguistique, le style formel et le style visuel. Le premier est l'utilisation d'un langage naturel pour décrire la construction. Le deuxième est l'utilisation de langage formel et le dernier est l'utilisation d'images et d'objets visuels. Nous avons employé le style linguistique lors de l'étape de la conception et c'est celui qui est le plus employé dans les cours de programmation de base dans le paradigme procédural.

Il existe, toujours selon le guide au corpus de connaissances (Abran et al., 2001, chap 4; 3.1), trois techniques pour réduire la complexité d'un système.

1. Éliminer ce qui n'est pas nécessaire au système.
2. Automatiser les tâches complexes du système, en utilisant des outils existants ou en écrivant nos propres outils.
3. Isoler un problème complexe et le découper en petits modules plus facilement

compréhensibles.

Les concepts utilisables pour réduire la complexité en utilisant la méthode linguistique sont : les modules génériques, les sous-programmes (procédure et fonction), l'encapsulation, les types de données abstraits, les librairies de composants (*framework*), les fichiers et les modules, les inspections formelles, les objets et l'utilisation de langages spécifiques au domaine. Étant donné le niveau des cours visés, nous n'utiliserons pas de bibliothèques de composants, ni d'objets, ni d'inspections formelles, ni de langages spécifiques au domaine. Par contre, nous prévoyons décrire les autres concepts dans notre méthode.

Il faut prévoir les changements possibles durant la vie d'un logiciel et les faciliter. Il existe trois techniques pour assouplir la construction d'un système et en faciliter le changement.

1. Généraliser plutôt qu'être spécifique. Il faut abstraire le plus possible.
2. Collecter le plus de données liées à l'application pour permettre de généraliser.
3. Isoler ce qui est appelé à changer pour avoir le moins d'endroits à modifier lors d'un changement (idée objet).

Les concepts utilisables pour prévoir le changement en utilisant la méthode linguistique sont : l'abstraction (information cachée), avoir une documentation solidement commentée, avoir un ensemble de méthodes complètes et suffisantes, utiliser une méthode objet, utiliser des fichiers de configuration (gestion de version), favoriser la réutilisation et faire des logiciels auto descriptifs (*plug and play*). Encore une fois, nous ne pouvons enseigner tous ces concepts dans les premiers cours de programmation, mais notre méthode prévoit décrire l'abstraction, la réutilisation et la documentation, toujours dans le but de préparer les étudiants à leur passage au génie logiciel.

Il faut aussi penser en fonction de la validation. Les outils décrits dans le guide au corpus de connaissances (Abran et al., 2001, chap. 4; 3.1.3) que nous utiliserons pour structurer en fonction de la validation en utilisant le style linguistique sont : la modularité, la programmation structurée, le raffinement successifs (par échafaudage).

PRINCIPES FONDAMENTAUX

- Les sous-programmes, les modules, les types abstraits, etc. seront conçus dans le but d'être réutilisés.
- Faire le code, un diagramme de la solution et un diagramme des modules offre la possibilité de voir sous plusieurs perspectives.
- Les artéfacts à produire sont définis dans la méthode.
- L'ajout et la modification de sous-programmes sont prévus et facilement applicables.
- L'approche descendante permettra de produire le logiciel par étapes.

- L'anticipation des changements est une considération du code dans la méthode.

LA CONSTRUCTION

Objectifs

Certains des objectifs suivants ne seront réalisables que lorsque vous aurez suivi un cours de programmation plus avancé. Cependant, rien ne vous empêche d'en avoir quelques-uns en tête lorsque vous coderez. Voici donc une liste d'objectifs à atteindre lors du codage d'un logiciel :

- Trouver une solution exécutable à un problème.
- Créer des fonctionnalités qui permettront d'automatiser les actions répétitives.

Pour y parvenir il faut :

- Réduire la complexité du logiciel.
- Anticiper les changements.
- Structurer pour la validation.

RÈGLES DE DÉVELOPPEMENT

Règles à suivre pour réduire la complexité :

Prendre les problèmes un à un et les découper en plus petits problèmes. Il est plus simple de résoudre plusieurs petits problèmes que de résoudre un grand problème. Exemple : pour valider une date entre 1900 et 2100, il faut valider l'année, valider le mois et ensuite valider le jour. Nous venons de diviser un problème en trois plus petits problèmes. Chacun de ces problèmes pourra être résolu comme le sont les exigences et les spécifications lors de la conception (algorithme global, entrées/sorties, algorithmes détaillés). L'important est que les sous problèmes de dernier niveau ne résolvent qu'un seul et unique problème. Si ce n'est pas le cas, ils devraient être découpés encore plus.

Utiliser l'approche descendante pour réduire la complexité de vos problèmes. Cette approche part des exigences vers la solution en découpant en plus petits problèmes. Par la suite, chaque

sous-problème est pris un à la fois et il est séparé en plus petits problèmes, jusqu'à ce que le problème soit simple à résoudre en quelques lignes de code.

Utiliser l'approche ascendante pour éliminer le code répétitif dans les cas semblables. Par exemple, si vous avez du code qui se répète avec seulement quelques différences, il est préférable d'écrire une fois le code dans un sous-programme et de passer des paramètres. Cette étape se fait habituellement après avoir écrit les algorithmes.

Exemple : Si nous reprenons la validation de la date, nous pouvons constater que valider l'année, valider le mois et valider le jour reviennent au même, au point de vue de l'algorithme.

Algorithme pour valider l'année

Il s'agit de lire une année et de s'assurer qu'elle est entre 1900 et 2100

Entrées : Une année de type entier

Sorties : Une année validée entre 1900 et 2100

Début

Saisir une année

Tant que l'année n'est pas entre 1900 et 2100

Afficher un message d'erreur

Saisir une autre année

Fin boucle

Fin

Algorithme pour valider le mois

Il s'agit de lire un mois et de s'assurer qu'il est entre 1 et 12

Entrées : Un mois de type entier

Sorties : Un mois validé entre 1 et 12

Début

Saisir un mois

Tant que le mois n'est pas entre 1 et 12

Afficher un message d'erreur

Saisir un autre mois

Fin boucle

Fin

Algorithme pour valider le jour

Il s'agit de lire un jour et de s'assurer qu'il est entre 1 et le nombre de jours maximum permis pour le mois actuel.

Entrées : Un mois et une année valide

Sorties : Un jour validé selon le mois

Début

Saisir un jour

Tant que le jour n'est pas entre 1 et le nombre de jours maximum

permis pour un mois.

Afficher un message d'erreur

Saisir un autre jour

Fin boucle

Fin

Nous pouvons voir que ces algorithmes se ressemblent beaucoup. Il faut généraliser et tenter de trouver un algorithme qui règle les trois cas. Il faut garder les parties d'algorithme qui sont identiques et les parties différentes deviendront des paramètres.

Algorithme pour valider un entier

Il s'agit de saisir un entier et de valider s'il est entre deux bornes reçues en paramètres

Entrées : Une borne minimale et une borne maximale de type entier et un message d'erreur

Sorties : Un entier validé entre les 2 bornes reçues

Début

Saisir un entier

Tant que l'entier n'est pas entre la borne minimale et la borne maximale

Afficher le message d'erreur reçu

Saisir un autre entier

Fin boucle

Fin

Cette approche est ascendante, puisque c'est la création d'un outil plus général qui aidera pour plusieurs parties qui ont été dessinées par l'approche descendante. De plus, cet utilitaire pourra resservir dans d'autres applications qui exigent le même genre de validation.

- Séparer le code en plusieurs modules pour regrouper les fonctionnalités qui s'occupent des mêmes choses. Par exemple, mettez tous les sous-programmes qui s'occupent de saisie de données au clavier dans le même module, tout ce qui fait référence à une structure de données dans un autre (ex : véhicule, propriétaire, ...), tout ce qui manipule un fichier en particulier, etc.

- Écrire et utiliser des types de données abstraits (piles, listes, files, arbres, ...) dans des modules séparés du code de l'application que vous êtes en train de construire. Ceci permet de séparer les structures de données utilisées du problème à résoudre.

- Séparer les spécifications des modules de leur implantation (encapsulation), lorsque le langage le permet (ads, adb en Ada, .h et .cpp en C++).

- Écrire des modules génériques (TEMPLATE en C++, GENERIC en Ada). Les modules génériques permettent également de généraliser en faisant de l'abstraction, mais cette fois-ci avec des types plutôt que des valeurs. Par exemple, les opérations (recherche, tri, insertion, suppression,...) d'un tableau d'entiers et celles d'un tableau de réels sont identiques. Dans un langage qui ne permet pas la généricité, il faut écrire deux fois le même code. Ce qui change, c'est seulement le type de données. Avec la généricité, le code est écrit une seule fois, c'est le type de données qui est passé en paramètre. Lorsque le langage le permet, il est préférable d'utiliser la généricité. Si le langage ne permet pas la généricité, il est parfois possible de la simuler (classe OBJECT en Java). Si c'est le cas, il est valable de le faire.

- Utiliser les modules déjà existants qui sont fournis avec le langage utilisé. Plus vous connaissez bien ces modules, moins vous avez à écrire de code.

Règles à suivre pour anticiper les changements :

- Généraliser (ou abstraire) plutôt que d'être spécifique. L'algèbre est un bon exemple de généralisation. Nous pouvons dire que $3 + 6 = 9$, que $4 + 8 = 12$ et que $5 + 10 = 15$. Nous pouvons, de façon plus générale, dire que $x + 2x = 3x$. Généraliser, c'est décrire de façon générale tous les scénarios possibles. Pour y arriver, utiliser les constantes, les paramètres et les variables dans la construction de vos sous-programmes, plutôt que de mettre des valeurs littérales liées à l'application que vous êtres en train de construire.

- Réutiliser le code lorsque cela est possible. Préférablement, le code est écrit, de prime abord, dans le but d'être réutilisé. Les sous-programmes, les modules, la généricité et les types de données abstraits servent à la réutilisation.

- Documenter le code.

    - Pour chaque variable, expliquer brièvement à quoi servira cette variable que vous êtes en train de définir.

    Exemple :

    'Sert à calculer le nombre de jours maximum permis pour un mois

Dim Nbr_jour_max As Integer;

***Le code est écrit en Visual Basic.

- Pour chaque sous-programme, il faut faire un descriptif du problème à résoudre, répéter la stratégie globale employée, décrire les entrées/sorties, décrire les critères qui doivent êtres respectés avant l'exécution du sous-programme (antécédent) et décrire les changements qu'apporte le sous-programme sur le système après son exécution (conséquent). De plus, nous mettrons une référence aux exigences et aux spécifications (E1, S9, ...), si cela s'applique. Si c'est un sous-programme utilitaire, nous l'indiquerons.

Exemple :

Procédure d'insertion d'une donnée dans un tableau

Description : Il s'agit d'insérer en ordre une donnée reçue en paramètre dans un tableau également reçu en paramètre.

Algorithme : Nous utilisons la fouille binaire pour trouver l'endroit où insérer. Nous décalons les données d'une case vers la droite et nous insérons la donnée à sa place.

Entrées : La donnée et le tableau

Sorties : Le tableau

Antécédent : Le tableau n'est pas plein

Conséquent : La donnée est dans le tableau, à sa place, en ordre croissant

Référence : Sous-programme utilitaire

- À l'intérieur même des instructions qui implantent la stratégie globale, insérer des commentaires pour guider le lecteur de votre code. Indiquer à quelle étape de votre algorithme vous êtes rendu.

Exemple :

```
'Nous recherchons l'emplacement où insérer la donnée

Endroit = Recherche_Binaire(LaDonnee, LeTableau)

'Nous décalons les cases vers la droite

For I = (N – 1) to Endroit step -1

    Tableau(I+1) = Tableau(I)

Next I

'Nous insérons la donnée

Tableau(Endroit) = LaDonnee

'Nous avons un élément de plus

N = N + 1
```

- Établisser des normes de programmation. Comment écrivez-vous le nom de vos constantes (majuscules ou minuscules)? Comment écrivez-vous le nom de vos variables : première lettre de chaque mot en majuscule, un petit souligné entre chaque mot ? Comment écrivez-vous les mots réservés ? Comment indentez-vous les blocs de code (boucle, sous-programme, ...). Où mettez-vous les commentaires dans le code, avant le bloc, après le bloc, à coté du bloc ? Le plus important est d'être constant dans vos choix et de suivre les normes imposées par vos enseignants.

Remarque

Le code d'un sous-programme n'est écrit qu'une seule fois. Le reste du temps, ce code est lu et modifié. Pour donner un aperçu, nous pouvons dire que le code est lu 1000 fois plus souvent qu'il n'est écrit[27]. Il est donc essentiel, pour faciliter le changement, qu'il soit bien écrit dès la première fois. L'erreur la plus fréquente est d'écrire les commentaires, d'indenter et d'aérer après que le code soit terminé. Si vous commentez au fur et à mesure, les commentaires vous guideront tout au long du développement de vos programmes, ce qui ne peut pas être le cas si vous ajoutez les commentaires à la fin.

Résolution de problèmes

L'acte de construction est fondamental. Il faut maintenant suivre le plan qui a été fait lors de la conception. Nous avons pris la peine de décrire ce qu'il y avait à faire, maintenant il faut le faire. Il faut voir ici que chacun des sous-programmes qui seront écrits peut s'entreprendre de la même façon que l'ont été les exigences et les spécifications. En fait, un sous-programme est un programme à écrire qui contient des exigences, possiblement des entrées/sorties, qui doit être fait dans un ordre précis, et qui doit fournir quelque chose d'exécutable. Le cycle de vie logiciel s'applique en entier pour chacun de ces sous-programmes. Cette étape se termine lorsque les algorithmes sont traduits dans un langage de programmation et qu'il existe une stratégie de vérification pour chaque exigence et spécification. En résumé, il y a aussi un cycle de vie pour un sous-programme.

La source : La conception est la principale source pour l'implantation. Il faut suivre l'ordre de priorité et les algorithmes qui ont été choisis. Si, lors de la construction, les stratégies d'implantation changent par rapport à la conception, il faudra mettre à jour la documentation.

La documentation : Le code est documenté le plus souvent sous forme de commentaires qui sont incorporés au code. Nous avons également un plan de l'organisation de la solution et des modules, que nous retrouverons à l'aide des diagrammes.

L'étape suivante : La prochaine étape est celle des tests. Si vous avez adopté l'approche descendante itérative, vous faites les tests au fur et à mesure que vous développez une fonctionnalité. Sinon, vous attendez à la fin, dans le modèle en cascade. Plus le projet est gros, moins le cycle de vie en cascade est utilisable. Naturellement, si vous détectez des lacunes dans votre conception, vous devez faire les ajustements avant de poursuivre.

---

[27] Ces valeurs ne sont basées sur une recherche scientifique, mais exprime la relation entre le code écrit et le code lu.

Mot sur le génie logiciel

Les objectifs de la construction en génie logiciel sont plus larges que ceux présentés ici. Il faut prévoir, dans cette phase, améliorer la productivité et la qualité du système, évaluer le logiciel, sélectionner les standards de développement du logiciel, choisir les langages de programmation, de configuration et les techniques de construction (manuelles ou automatisées). Tout ceci peut se faire en utilisant des outils existants ou en produisant soi-même ses outils. Il faut penser à réduire la complexité du système, anticiper le changement, structurer en fonction de la validation et utiliser des standards externes pour permettre de partager des données avec le monde extérieur. Les concepts comme l'abstraction, l'encapsulation, la réutilisation et les autres doivent être présents constamment dans votre esprit lors de la construction. Ceci vous dirigera dans vos choix à toutes les étapes de la conception et de la construction.

Description de la documentation

La documentation du code se résume au listage du programme documenté et d'un plan visuel du code. Pour le plan, nous construirons un diagramme hiérarchique que nous appellerons ***diagramme de la solution***, qui doit être une représentation des étapes qui ont servi à répondre aux exigences. Le diagramme de la solution est une image des fonctionnalités (procédures et fonctions) qui ont été réalisées. Pour ce diagramme, chaque élément représente une exigence, une sous-exigence ou une fonctionnalité implantée. Si une fonctionnalité se retrouve à plusieurs endroits, elle contiendra une étoile. Ce diagramme est fait après le code. C'est, en d'autres mots, une image du code.

Nous construirons également un diagramme de modules. Ce diagramme se fait lorsque vous découpez votre solution en plusieurs modules. Chaque module contient plusieurs fonctionnalités qui interagissent avec d'autres modules. Il suffit ici de montrer visuellement le découpage en modules et les interactions entre les modules.

Note : Nous n'incluons pas les modules standards du langage de programmation utilisé, seulement ceux construits par le programmeur.

Exemple

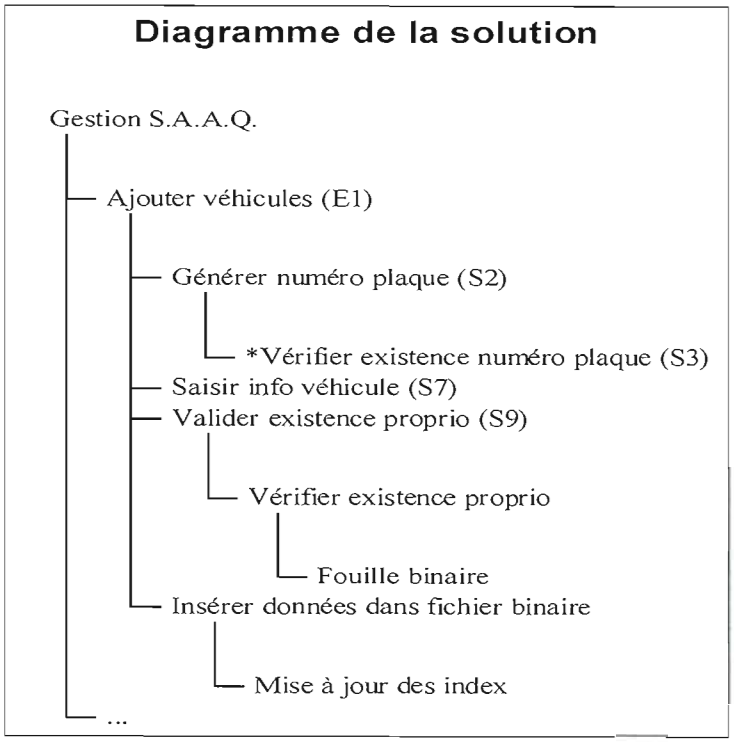
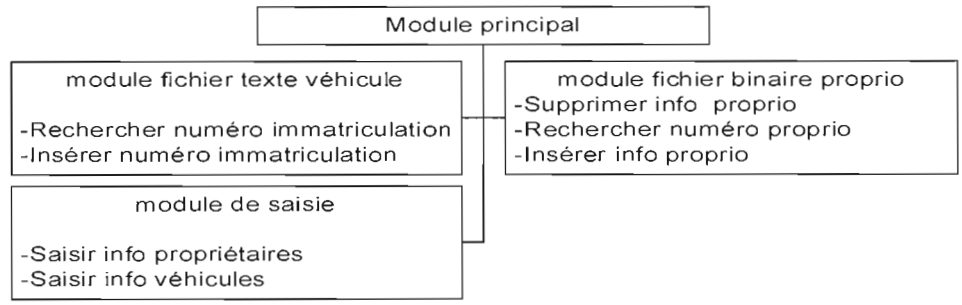


Diagramme de la solution
Gestion S.A.A.Q.
Ajouter véhicules (E1)
Générer numéro plaque (S2)
*Vérifier existence numéro plaque (S3)
Saisir info véhicule (S7)
Valider existence proprio (S9)
Vérifier existence proprio
Fouille binaire
Insérer données dans fichier binaire
Mise à jour des index
...

Diagramme des modules



Note au lecteur

La décision de faire un autre diagramme pour la solution devrait permettre aux étudiants de voir une réelle différence entre les exigences et la solution. Les diagrammes se ressembleront, mais les étudiants devraient mieux voir ainsi que s'il fallait changer les décisions d'implantation, cela ne touche pas aux exigences. Par contre, s'ils changent le diagramme des exigences, cela aura un effet sur le diagramme de la solution.

Note au lecteur concernant les tests

Définition selon le guide au corpus de connaissances en génie logiciel (Abran et al., 2001)

Tester un logiciel consiste à effectuer une vérification *dynamique* du comportement du programme sur un ensemble *fini* de cas tests, convenablement *sélectionnés* à partir du domaine d'exécution habituellement infini, relativement au comportement spécifié *désiré*. Les termes en italiques dans cette définition sont importants pour la direction que doivent prendre les tests.

- *Dynamique* : parce que les tests se font durant l'exécution du logiciel. Il faut donc faire des tests sur le produit final.
- *Fini et sélectionné* : parce qu'il y a une infinité de tests possibles et qu'il est impossible de les faire tous. Il faudra donc en choisir.
- *Désirés* : parce qu'il faudra choisir aussi le genre de comportement que l'on veut tester.

C'est donc sur le choix des tests et sur les techniques permettant d'y parvenir que les décisions doivent se prendre, dans le but de satisfaire des objectifs précis.

Objectifs (Abran et al., 2001, chap. 5; 2.1)

1. Tester à tous les niveaux du processus de développement.
2. Exposer les erreurs.
3. Déterminer les tests cibles et les tests objectifs à différents niveaux.
4. Assurer la qualité du logiciel.
5. Mesurer la fiabilité.
6. Évaluer l'utilisabilité.
7. Obtenir l'acceptation du contracteur (client).

Les objectifs que nous pouvons retenir dans le cadre des cours qui nous intéressent sont 1, 2, 3, 4, la mesure, l'évaluation et l'acceptation étant peu réalisables. Pour atteindre les objectifs retenus, notre méthode prévoit commencer la période de test dès l'étape de la conception en déterminant des scénarios possibles qui viseront à vérifier les algorithmes. C'est ce que nous appellerons la stratégie de vérification. De cette façon, plusieurs erreurs pourront être évitées

avant de commencer le processus de test. Par la suite, c'est la mise en application de cette stratégie, sur la version exécutable du logiciel, qui consistuera les tests.

Toujours selon le guide au corpus de connaissances (Abran et al., 2001), la littérature fournit un certain nombre de fondements théoriques aux tests. Voici la liste des fondements qui ont été retenus avec une courte définition pour chacun :

- La sélection des critères de test : consiste à décider quels critères permettront de sélectionner les ensembles de tests que doit passer le logiciel (Abran et al., 2001; Pfleeger, 1998; Zhu, 1997; Weyuker, 1983; Weyuker et al., 1991).
- La sélection des modèles de test : peut être guidée par différents objectifs (Abran et al., 2001; Beizer, 1990; Perry, 1995; Frankl et al., 1998).
- L'identification des défectuosités : il est préférable de penser à des tests pour trouver des erreurs plutôt qu'à des tests qui prouvent le bon fonctionnement du logiciel (Abran et al., 2001; Beizer, 1990; Kaner et al., 1999).
- La prédiction des résultats attendus pour l'acceptation ou le refus d'un test (*Oracle problem*) (Abran et al., 2001; Beizer, 1990; Weyuker, 1982).

Le guide au corpus de connaissances (Abran et al., 2001) décrit trois étapes pour les tests d'un logiciel :

- Test des unités (*Unit testing*) : Vérifie un logiciel en morceaux, unité par unité. Cette étape est faite lorsque le code a atteint une complète maturité. Donc, lorsqu'il compile correctement (Abran et al., 2001; Beizer, 1990; Perry, 1995; Pfleeger, 1998; IEEE 1008-1987).
- Test d'intégration : Vérifie l'intégration des unités dans le système. Cette étape est faite après le test des unités. On procède traditionnellement selon la hiérarchie du logiciel, soit du bas vers le haut, soit l'inverse. Dans les approches plus modernes on cherche plutôt à tester l'intégration des unités fonctionnelles (Abran et al., 2001; Jorgensen, 1995; Pfleeger, 1998).
- Test du système : Vérifie le comportement global du logiciel dans son environnement. Vérifie aussi les exigences non-fonctionnelles (sécurité, performance,...) (Abran et al., 2001; Jorgensen, 1995; Pfleeger, 1998).

Pour ces trois étapes, les tests dépendront des buts à atteindre. La liste des buts reconnus par le génie logiciel est assez longue et exhaustive. Elle requiert de vérifier l'installation, la fiabilité, la performance, la configuration, de faire des tests alpha/beta, etc. Pour les cours de programmation de base, les buts sont plus restreints et voici la liste de ceux que nous avons retenus :

- Acceptation/qualification : Le but est de vérifier si le logiciel respecte les exigences du client (Abran et al., 2001; Perry, 1995; Pfleeger, 1998; IEEE 12207-1996).
- Test de conformité et test fonctionnel : Le but est de vérifier si le comportement obtenu est conforme au comportement attendu (Kaner et al., 1999; Perry, 1995; Wakid et al., 1999).
- Performance : Vérifie que le logiciel respecte les exigences non-fonctionnelles de performance. Dans le cas des cours de programmation de base, il se peut que ce soit des contraintes d'efficacité sur les types de données abstraits, par exemple (Abran et al., 2001; Perry, 1995; Pfleeger, 1998; Wakid et al., 1999).

Voici la liste des différentes techniques présentées dans le guide au corpus de connaissances (Abran et al., 2001) qui permettent de tester un logiciel :

- Basées sur l'intuition et l'expérience du testeur
- Basées sur les spécifications
- Basées sur le code
- Basées sur les erreurs
- Basées sur l'utilisation
- Basées sur la nature de l'application

Pour cette section, nous voulions que notre méthode utilise une combinaison de certaines de ces techniques. Les techniques que nous croyons qui seront les plus applicables dans le cadre de cours de programmation de base seront basées sur les spécifications, le code et les erreurs. Nous laisserons les tests basés sur l'intuition et l'expérience du testeur puisque les étudiants visés par la méthode sont supposés être des débutants. Nous n'utiliserons pas non plus les tests basés sur l'utilisation, puisqu'il n'y aura pas de transition, ni sur la nature de l'application, parce que ce ne seront pas des applications très complexes.

Pour la plupart de ces techniques, il existe une ou plusieurs façons de les faire. Par exemple, lorsque les tests sont basés sur les spécifications, ils peuvent l'être par la technique des partitions équivalentes, par une analyse des valeurs limites, à l'aide d'une table de décision, par un automate fini, à l'aide de spécifications formelles ou par des tests générés aléatoirement. Le but de ces tests est de s'assurer que le logiciel répond aux exigences et aux spécifications. Si les tests sont basés sur le code, ils le seront à l'aide des diagrammes de flots de contrôles, de décisions et de flots de données. Le but étant de vérifier l'interaction entre les unités, les définitions et l'utilisation des variables du programme. Finalement, si les tests sont basés sur les erreurs, les tests seront faits grâce à la technique de prévisions d'erreurs (*error guessing*) ou grâce à la technique des mutations. Il est possible de sélectionner et de combiner quelques-unes de ces techniques. C'est ce que nous ferons pour fournir aux étudiants plusieurs façons de vérifier leur travail selon des objectifs prédéterminés. Comme nous ne pourrons pas enseigner dans un seul cours toutes ces techniques et toutes ces sortes de diagramme, nous avons opté pour un style plus linguistique pour la description des tests à effectuer. La stratégie de vérification devra permettre l'analyse des valeurs limites, la vérification des interactions entre les unités du programme et la prévision d'erreurs.

Même s'il faut tester à tous les niveaux du processus de développement (Abran et al., 2001, chap. 5 ; 1), la plupart des tests se préparent au moment de la conception, mais se réalisent au moment de la phase de transition. Pour nous, cette phase se résume à tester la version exécutable du logiciel que les étudiants auront à construire. Notre méthode prévoit faire une stratégie de vérification et pour prévoir les résultats attendus, durant l'étape de construction. Les tests seront d'effectuer la stratégie de vérification sur la version exécutable et de comparer les résultats obtenus avec les résultats attendus.

PRINCIPES FONDAMENTAUX

- Les tests seront faits du point de vue des spécifications, du code et des erreurs.
- Les artéfacts à produire sont définis dans la méthode.
- La stratégie de vérification fait partie d'un processus rigoureux.
- Les tests font partie des étapes à franchir pour la production du logiciel.

LES TESTS

Objectifs

Les objectifs à atteindre sont importants pour la sélection des bons tests. Il faut avoir en tête quel est l'objectif principal lorsqu'on détermine quel est le test que l'on veut faire. Voici la liste des objectifs possibles :

- Tester tous les niveaux du processus de développement.
- Exposer les erreurs.
- Déterminer les tests cibles et les tests objectifs à différents niveaux.
- Assurer la qualité du logiciel.

Il faut donc faire une stratégie de vérification. Cette stratégie doit permettre de vérifier que les exigences et les spécifications sont respectées. Elle doit aussi vérifier que les interactions entre les différents sous-programmes se font correctement. Elle doit vérifier que les données se transmettent correctement entre chaque sous-programme et elle doit finalement vérifier les erreurs prévisibles.

Règles de développement

- Sélectionner des critères de tests (fiabilité, conformité, validité,...). C'est ce qui consiste à décider quels critères permettront de sélectionner les ensembles de tests que doit passer le logiciel.

- Parcourir les fonctionnalités qui ont été implantées et vous assurer que les entrées/sorties sont valides et que les interactions entre les fonctionnalités respectent le diagramme hiérarchique.

- Identifier les défectuosités possibles et prévisibles (ajout d'un client existant, retrait d'un client inexistant, ouverture d'un fichier inexistant, etc). Il est préférable de penser à des tests pour trouver des erreurs qu'à des tests qui prouvent le bon fonctionnement du logiciel.

- Prévoir des résultats attendus concrets pour l'acceptation ou le refus d'un test. Il faudra vérifier toutes les contraintes des spécifications du document sur les exigences et les spécifications. Vous aurez à faire référence au numéro des exigences et des spécifications testées.

Résolution de problèmes

Les tests sont comme le logiciel et il faut les planifier. Idéalement, il faut prévoir l'objectif et le résultat attendu d'un test. Il faut tester le logiciel du point de vue des spécifications, du code et des erreurs.

La source : Les exigences et les spécifications, le diagramme de la solution et le code sont les principales sources des tests. Il faut s'assurer que les exigences et les spécifications sont respectées et que la solution respecte les relations du diagramme hiérarchique de la solution.

La documentation : La description de la stratégie de vérification servira à documenter les tests à effectuer. Il faudra s'assurer d'avoir une autre section qui décrit les tests lorsqu'ils auront été effectués, afin de pouvoir comparer les résultats attendus à ceux qui ont été obtenus lors du test réel.

L'étape suivante : Si une erreur est détectée lors des tests, il faut détecter la source du problème. Cela peut être une exigence mal énoncée, un oubli dans le diagramme hiérarchique de la solution, une erreur de logique dans le code ou une erreur dans la stratégie de vérification ou dans le test final. Peu importe, il faudra réparer l'erreur à la source et refaire le cycle de vie à partir de l'endroit où le problème a été détecté. **L'important est de maintenir la documentation à jour.**

Mot sur le génie logiciel

En génie logiciel, des tests sont prévus pour différents types d'application et de programmation. Il existe plusieurs techniques de tests pour vérifier les spécifications, plusieurs sortes de diagrammes, plusieurs méthodes et plusieurs outils automatisés pour trouver les erreurs, etc. Des tests sont également prévus pour vérifier les composants un par un et ensuite leur comportement dans le système. Il est prévu de faire tester le logiciel par des utilisateurs internes et externes, dans différents environnements. En résumé, les tests ne sont pas une mince tâche et il ne faut pas les prendre à la légère.

Description de la documentation

En plus de la modularité, du raffinement successif et de la programmation structurée, nous pouvons structurer en fonction de la validation, en prévoyant des scénarios de tests que devront passer les différents sous-programmes, dans le but de satisfaire les exigences et les spécifications. Préférablement, vous devez décrire les tests en fournissant leur description, leur but, quelles sont les exigences et les spécifications qui seront satisfaites, les valeurs concrètes que vous utiliserez pour effectuer le test ainsi que les résultats attendus.

Nous diviserons la stratégie de vérification en trois sections :

- Tests sur les spécifications.
- Tests sur le comportement.
- Tests sur la détection d'erreurs.

Pour les tests, la description se fera de la façon suivante :

- Test (numéroter le test):
- But :
- Démarche :
- Comportement ou résultats attendus :
- Références :

***Il est important de décrire les valeurs exactes qui devront être entrées pour les tests sur les spécifications de validation.

Exemple de documentation :

Tests pour les spécifications

Test 1 : vérifier que l'année acceptée est seulement entre 1900 et 2100 inclusivement.

But : s'assurer que le programme redemande une année si l'entrée est invalide.

Démarche : entrer les valeurs 1899, 1900, 2000, 2100, 2101.

Comportement ou résultats attendus : 1900, 2000 et 2100 sont valides et 1899 et 2101 sont invalides.

Références : E8, E10, S12 (hypothétique ici).

Truc : Dans le cas d'une vérification de valeurs **consécutives** (numériques ou caractères), cinq valeurs suffisent. Les deux valeurs limites (1900 et 2100 dans notre exemple), les deux premières valeurs hors bornes (1899 et 2101) et une valeur entre les bornes (2000). Naturellement, ceci ne tient pas compte des erreurs d'incompatibilité de type. Ce genre d'erreur se gère différemment selon le langage de programmation utilisée.

Exemple d'une stratégie de vérification inutilisable.

Test : entrer différents choix au menu principal.

But : s'assurer que la validation des choix s'effectue correctement.

Démarche : **entrer des valeurs valides et invalides pour tester.**

***Lors du test il n'y aura aucune comparaison possible entre la stratégie de vérification et le test puisqu'il n'y a pas de valeurs concrètes. Lorsque vous faites la stratégie de vérification, imaginez que ce sera une autre personne qui fera les tests et écrivez quelles valeurs vous aimeriez qu'elle entre et quels sont les résultats attendus.

Tests pour les comportements

Test 8 : quitter au début de l'application.

But : s'assurer que le programme se termine correctement.

Démarche : entrer 'Q' au menu principal.

Comportement ou résultats attendus : le programme se termine.

Références : utilitaire.

Test 18 : entrer différents choix au menu principal.

But : s'assurer que la validation des choix s'effectue correctement.

Démarche : entrer 0, 3, 8, 1, 7.

Comportement ou résultats attendus : 1,7 et 8 sont valides et 0,3 sont invalides.

Références : E9, S10 (hypothétique ici).

Test 25 : entrer un nom de fichier inexistant lors de l'ouverture d'un fichier.

But : s'assurer que le programme avise et redemande un nom de fichier existant.

Démarche : entrer abcdefghijklmnopqrst.2334579.

Comportement ou résultats attendus : un message d'erreur apparaît et redemande un nom de fichier existant.

Références : utilitaire.

- Pensez toujours à vérifier les cas extrêmes. Par exemple, un fichier vide pour la manipulation de fichier, une structure pleine (pile, file, tableau,...) lors de l'insertion de donnée, une structure vide lors de suppression ou une recherche de données inexistantes, etc.

Tests pour la détection d'erreurs.

Test 36 : entrer le même numéro de plaque à deux propriétaires différents.

But : s'assurer que le programme rejète deux fois le même numéro de plaque.

Démarche : ajouter un véhicule et laisser le système attribuer un numéro de plaque.

prendre ce numéro en note et ajouter deux propriétaires avec ce numéro de plaque en référence.

Comportement ou résultats attendus : un message avise que le numéro de plaque a déjà été attribué.

Références : S3

***Les tests peuvent être moins précis lorsqu'il y a des générations aléatoires faites par le logiciel.

CONCLUSION

Cette méthode illustre les étapes de développement de logiciel ainsi que les liens qu'il y a entre ces étapes. Nous présentons comment documenter les différentes étapes et nous illustrons le tout avec des exemples. Chaque affirmation de cette méthode provient d'une étude rigoureuse du corpus de connaissances en génie logiciel nommé *Guide to the Software Engineering Body Of Knowledge* (Abran et al., 2001) et la méthode utilise des principes fondamentaux du génie logiciel identifiés par des experts dans le domaine (Dupuis et al., 1999).

RÉFÉRENCES

(Abran et al., 2001) *Guide to the Software Engineering Body Of Knowledge*, Abran, A., P. Bourque, R. Dupuis, Iron Man version 1.0, 2001.

(Bagert et al., 1999) Bagert, Donald J., T. B. Hilburn, G. Hislop, M. Lutz, M. McCracken, S. Mengel, *Guidelines for Software Engineering Education Version 1.0*, Octobre 1999, Technical Report, CMU/SEI-99-TR-032, ESC-TR-99-002.

(Bass et al., 1998) Bass, L., P. Clements, R. Kazman., *Software Architecture in Practice*, SEI Series in Software Engineering, Addison-Wesley, 1998

(Beizer, 1990) Beizer, B. *Software Testing Techniques, 2nd Edition.*V. N. Reinhold, 1990. Chapters 1, 2, 3, 5, 7s4, 10s3, 11, 13].

(Bell, 1987) Bell, D., I. Morrey, J. Pugh, *Software Engineering : A Practionner's Approach*, Prentice-Hall, 1987.

(Beaubouef et al., 2001) Beaubouef, T., R. Lucas, J. Howatt, *The UNLOCK System: Enhancing Problem Solving Skill in CS-1 Students*, SIGCSE Bulletin, Volume 33 No 2, p.43-46, June 2001.

(Bertolino, 2001) Bertolino, A., *Software test* In *Guide to the Software Engineering Body Of Knowledge*, Abran, A., P. Bourque, R. Dupuis, Iron Man version 1.0, 2001. Chap. 5.

(Bollinger et al., 2001) Bollinger, T., P. Gabrini, L. Martin, *Software construction* In *Guide to the Software Engineering Body Of Knowledge*, Abran, A., P. Bourque, R. Dupuis, Iron Man version 1.0, 2001. Chap. 4.

(Booch, et al., 1999) Booch, G., J. Rumbauch, I. Jacobson, *The Unified Modeling Language User Guide,* Addison-Wesley, *1999*

Bouthat, Chantal. 1993. *Guide de présentation des mémoires et thèses*, Université du Québec à Montréal.

(Budgen, 1994) Budgen , D. *Software Design.* Addison-Wesley, 1994.

(Bushman et al., 1996) Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-oriented Software Architecture – A System Patterns*, J. Wiley and Sons, 1996.

(Clements, 1986) Clements, D.H., *Effects of Logo and CAI Environnements On Cognition And Creativity*, Journal of Educational Psychology, 78, p. 309-318, 1986.

(Clements, 1990) Clements, D.H., *Logo: Search and research. Turtle soup: A beginning look at logo research.* Logo Exchange, 78, p. 309-318, 1990.

(Clements et al., 1984) Clements, D.H., D.F. Gullo, "Effects of computer programming on young children's cognition", *Journal of Educational Psychology*, 76, 1051-1058, 1984

(Davis, 1993) Davis, A.M., *Software Requirements: Objects, Functions and States.* Prentice-Hall, 1993.

(Dupuis et al., 1999) Dupuis, R., P. Bourque, A. Abran, J. W. Moore, L. Tripp, S. Wolff. 1999. *Fundamental Principles Of Software Engineering-A Journey (*23 décembre). Université du Québec à Montréal.

(Fowler and Scott, 1997) Fowler, M., K., Scott, 1997. *UML distilled.* Addison-Wesley, 1997

(Frank et al., 1998) Frank, P., D. Hamlet, , B. Littlewood, and L Strigini, Evaluating testing methods by delivered reliability, *IEEE Transactions on Software Engineering,* 24, 8, August 1998, p.586-601.

(Gantenbein, 1989) Gantenbein, E. R., *Programming as Process: A "Novel"* Approach to Teaching Programming", SIGCSE Bulletin, Volume 21 No 1, p. 22-26, February 1989.

(Ghezzi et al., 2003) Ghezzi, C., M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering,* (2e éd.), New-Jersey : Prentice Hall., 2003

(Gries, 1974) Gries, D., *What should we teach in an Introductory Programming Course?*, SIGCSE Bulletin, Volume 6 No 1, p. 81-89, February 1974.

(Henderson, 1986) Henderson P., *Anatomy of An Introductory Computer Science Course*, SIGCSE Bulletin, p. 257-263, February 1986.

(Henderson, 1987) Henderson, P., *Modern Introductory Computer Science*, SIGCSE Bulletin, Volume 19, No 1, p. 183-189, February 1987.

(Hyde et al., 1979) Hyde, D. C., B. D. Gay, D. Utter, Jr.*, *The integration of a problem solving process in the first course*, SIGCSE Bulletin, Volume 11, No 1, p. 54-59, February 1979.

(IEEE 610.12-1990) Institute of Electrical and Electronics Engineers. IEEE *Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12-1990, IEEE, New York, 1990.

(IEEE 830-1983) Institute of Electrical and Electronics Engineers. IEEE *Standard Glossary of Software Engineering Terminology.* ANSI/IEEE Standard 830-1984. IEEE, New York, 1984.

(IEEE 830-1998) Institute of Electrical and Electronics Engineers. IEEE *Recommended Practice for Software Requirements Specifications.* ANSI/IEEE Standard 830-1998. IEEE, New York, 1998.

(IEEE 1008-1987) Institute of Electrical and Electronics Engineers. IEEE Std 1008-1987, *Standard for Software Unit Testing*, IEEE, New York, 1993

(IEEE 1016-1998) Institute of Electrical and Electronics Engineers. IEEE *Recommended Practice for Software Conception Description.* IEEE Std 1016-1998, IEEE New York, 1998.

(IEEE 12207-1996) 12207 IEEE/EIA 12207.0-1996, Industry Implementation of Int. Std. ISO/IEC 12207:1995, *Standard for Information Technology-Software Life cycle processes*, IEEE, New York, 1996

(Jorgensen, 1995) Jorgensen, P.C., *Software Testing A Craftsman's Approach*, CRC Press, 1995. [Chapters 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15].

(Kaner, et al., 1999) Kaner, C., J. Falk , and H. Q. Nguyen, , *Testing Computer Software*, 2nd Edition, Wiley, 1999. [Chapters 1, 2, 5, 6, 7, 8, 11, 12, 13, 15]

(Kruchten, 2000) Kruchten, P. *The Rational Unified Process : An Introduction.*, Addison-Wesley, 2000

(Kushan, 1994) Kushan B., *Preparing Programming Teaching*, SISCSE Bulletin Volume 26 No 1, p. 248-252, Mars 1994.

(Kontoya and Sommerville, 2000) Kontoya, G., and I. Sommerville, *Requirements Engineering: Processes and Techniques*. John Wiley and Sons, 2000.

(Laforest, 1994) Laforest, L. 1994. *Introduction au génie logiciel et à la programmation* (notes de cours).

(Larousse, 1999*) Le petit Larousse illustré* 1999

(Liskov and Guttag, 2001) Liskov, B. and J. Guttag. *Abstraction and Specification and Object-Oriented Design.* Addison-Wesley, 2001.

(Lyu, 1996) Lyu, M.R. (Ed.), *Handbook of Software Reliability Engineering*, Mc-Graw-Hill/IEEE, 1996. [Chapters 2s2.2, 5, 6, 7].

(Mayer, 1987) Mayer, R.E., "The exclusive search for teachable aspects of problem solving", In J.A. Glover & R.R. Ronning (Eds.), *Historical Foundations Of Educational Psychology,* p. 327-347, New-York: Plenum.

(Mayer et al., 1986) Mayer, R.E., J.L. Dick et W. Vilberg, "Learning to program and learning to think : What's the connection?", *Communications of the ACM*, No 29, p. 605-610, 1986.

(McCauley et al., 1995) McCauley, R., C. Archer, N. Dale, R. Mili, J. Roberge, and H. Taylor. "The Effective Integration of the Software Engineering Principles Throughout the Undergraduate Computer Science Curriculum" SIGCSE Bulletin, Volume 27 No 1, p. 364-365, mars 1995.

(McCauley et al., 2000) McCauley, R., N. Dale, T. Hilburn, S. Mengel, B. W. Murrill, "The Assimilation of Software Engineering Into the Undergraduate Computer Science Curriculum", SIGCSE Bulletin, Volume 32 No 1, p. 423-424, Marsh 2000.

(McConnell, 1993) McConnell, S., *Code complete - A practical handbook of software construction*, chap. 4 – 19, Microsoft press, 1993.

(Page-Jones, 1988) Page-Jones M., *The practical guide to structured systems design*, Englewood Cliffs, N.J. : Yourdon Press , c1988

(Perry, 1995) Perry, W. *Effective Methods for Software Testing*, Wiley, 1995. [Chapters 1, 2, 3, 4, 9, 10, 11, 12, 17, 19, 20, 21].

(Pfleeger, 1998) Pfleeger, S.L., *Software Engineering -- Theory and Practice*. Prentice-Hall, 1998.

(Pfleeger, 2001) Pfleeger, S.L., *Software Engineering -- Theory and Practice (second edition)*, Prentice-Hall, 2001.

(Pressman, 1997) Pressman, R.S. *Software Engineering -- A Practitioner's Approach (Fourth Edition).* McGraw-Hill, Inc., 1997.

(Robillard et al., 2002) Robillard P., P. Kruchten, P. d'Astous, *Software Engineering Process--with the UPEDU*, Addison-Wesley, 2002.

(Robins et al., 2001) Robins, A., N. Rountree and J. Rountree, *My program is correct but it doesn't run: A review of novice programming and a study of an introductory programming paper*, Technical Report *OUCS-2001-06, 2001*

(Rombach, 2003) Rombach, H. D., *Teaching how to engineer software*, Conference on the 16th Software Engineering Education and Training (CSEET'03). http://www.ls.fi.upm.es/cseet03/ks_rombach.html.

(Sawyer et al., 2001) Sawyer, P. and G. Kotonya., *Software requirements* In *Guide to the Software Engineering Body Of Knowledge*, Abran, A., P. Bourque, R. Dupuis, Iron Man version 1.0, 2001. Chap. 2.

(Sheil, 1981) Sheil, B. *The psychological study of programming, Computer Surveys,* Vol 13, No 1, Marsh 1981, pp. 101-120.

(Soloway, 1986) Soloway, E., *Learning to program = learning to construct mechanisms and explanations*, Communications of the ACM, No 29, p. 850-858, 1986.

(Soloway et al., 1982) Soloway, E., J. Lochhead, J. Clements, *Computer programming enhance problem solving ability ? Some positive evidence on algebra word problems.* In R. Seidel, B. Hunter, & R. Anderson (Eds) Computer literacy p. 171-215. New-York: Academic Press.

(Sommerville, 1997) Sommerville, I., and P. Sawyer, *Requirements Engineering: A good Practice Guide*. John Wiley and Sons, Chap. 1-2, 1997.

(Sommerville, 2001) Sommerville, I., *Software Engineering (6th edition)*, Addison-Wesley, 2001.

(Tremblay, 2001) Tremblay, G., *Software design* In *Guide to the Software Engineering Body Of Knowledge*, Abran, A., P. Bourque, R. Dupuis, Iron Man version 1.0, 2001. Chap. 3.

(Turner, 2001) Turner, J., *Reflections on Curriculum Development in Computing Programs*, Invited Editorial, Volume 33 No 2, SIGCSE Bulletin, p. 4-6, Juin 2001.

(Thayer and Dorfman, 1997) Thayer, R.H., and M. Dorfman, *Software Requirements Engineering (2nd Ed)*. IEEE Computer Society Press, p. 176-205, 389-404, 1997.

(Wakid et al., 1999) Wakid, S.A., D.R. Kuhn, and D.R. Wallace, *Toward Credible IT Testing and Certification*, August 1999, p. 39-47.

(West, 2002) West, D. *Planning a project with the Rational Unified Project. Rational Software White Paper, TP 151, 08/2002* http://www.rational.com/products/whitepapers/453.jsp

(Weyuker, 1982) Weyuker, E.J. On Testing Non-testable Programs. *The Computer Journal*, 25, 4, p. 465-470, 1982.

(Weyuker, 1983) Weyuker, E.J. *Assessing Test Data Adequacy through Program Inference*. ACM Trans. on Programming Languages and Systems, 5, 4, p. 641-655, October 1983.

(Weyuker et al., 1991) Weyuker, E.J., S. N. Weiss, and D. Hamlet. *Comparison of Program Test Strategies* in *Proc.* Symposium on Testing, Analysis and Verification TAV 4, ACM Press, p. 1-10, Victoria, British Columbia, October 1991.

(Winslow, 1996) Winslow, L. E., *Programming Pedagogy - A Psychological Overview*, SISCSE Bulletin, Volume 28 No 3, p.17-25, September 1996.

(Wolff, 1999) Wolff, S. *La place de la mesure au sein des principes fondamentaux génie logiciel*. Activité de synthèse, département d'informatique de l'Université du Québec à Montréal, juillet 1999.

(Zhu et al., 1997) Zhu, H., P.A.V. Hall, , and J.H.R. May. *Software Unit Test Coverage and Adequacy*. ACM Computing Surveys, Volume 29, No 4, p. 366-427. [Sections 1, 2.2, 3.2, 3.3]. December 1997.