VADE-MECUM DU CHEF DE PROJET

☐ 1 - Le chef de projet est à un carrefour de conflits humains et d'intérêts économiques contradictoires :

- Le client maître d'ouvrage (MOA) souhaite « en avoir » le plus possible pour son argent.

  ➘ *Le client est naturellement inflationniste (c'est normal, il est dans son rôle !).*

- Le maître d'œuvre (MOE) industriel du projet (ou sa direction générale) souhaite investir le minimum en personne, en équipement, en simulation, etc. de façon à maximiser la marge du projet.

  ➘ *C'est une donnée économique fondamentale que tout projet doit dégager du profit.*

- L'équipe de développement a une tendance naturelle à se faire plaisir et souhaite :
  - plus de fonctionnalités à développer,
  - plus de complexité (gratification technique) et d'innovations (défi personnel face aux incertitudes et aux aléas),
  - plus de moyens pour développer (équipements, outils, personnel),

  ➘ *Si elle est jeune et inexpérimentée, elle pêchera naturellement par excès d'optimisme et de naïveté, ce qui se traduira, le plus souvent, par une sous-estimation chronique de l'effort requis pour les tests et les tâches transverses.*

Le chef de projet doit assumer ces contradictions et agir en conséquence :

- C'est un homme de défis qui doit faire preuve de caractère, de réalisme et de courage. Il ne doit pas fuir les conflits, mais toujours s'efforcer de les résoudre positivement en recherchant le compromis juste.

- Dans tout projet, il y a les risques prévisibles et les impondérables. Le chef de projet avisé éliminera tous les risques prévisibles qui, comme chacun sait, ne manqueront pas de se produire[1], et il réservera son énergie vitale pour réagir aux incertitudes et à l'imprévisible.

- L'excès d'optimisme (ou l'irréalisme), la fuite en avant sont en général cruellement, et **toujours**, sanctionnés par les faits.

[1] C'est ce que dit la loi de Murphy: "If anything can go wrong, it will".

☐ 2 - Le chef de projet doit avoir en tête quelques données simples :

- La règle 40 - 20 - 40[2] stipule qu'en moyenne :

  - 40% de l'effort est requis pour spécifier et concevoir,
  - 20% de l'effort va à la programmation,
  - 40% de l'effort va aux tests de vérification et de validation.

  ➘ *Se focaliser uniquement sur la programmation, c'est se préparer à une catastrophe qui ne manquera certainement pas d'arriver.*

- Il est normal de consacrer 15 à 20% des ressources du projet à des tâches dites transversales (gestion de l'équipe, assurance qualité, formation, documentation, configuration matériel et logiciel, etc.).

  ➘ *C'est le ratio classique du « chief programmer team » connu depuis le début des années 70s.*

- Pour une ressource allouée de 100, la dépense constatée est couramment 130. De plus, si ***x*** est alloué, alors que ***x-a*** aurait suffit, de toute façon ***x*** sera consommé et ***a*** aura été dilapidé (Loi de Parkinson[3]). Enfin, il faudra réserver des forces pour la dernière ligne droite où l'expérience montre qu'il y a toujours concentration des contingences et des incertitudes (d'où l'effet bénéfique de définir des dates butées intermédiaires avant l'échéance finale à titre préventif).

  ➘ *Le chef de projet avisé se gardera d'allouer toutes ses ressources d'entrée de jeu ; il en réservera au moins 30% comme marge de management.*

- L'erreur étant humaine, une bonne partie de l'effort de développement est consacré à corriger et à refaire. La règle 40 - 20 - 40 est à compléter par les ratios 30 - 50 - 70[4] qui précisent l'effort consacré à ce qui sera probablement refait (50% selon ce modèle).

  ➘ *Le taux de reprise est maximal dans la phase de tests (ce qui ne surprendra personne) ; en conséquence le chef de projet, imposera dès la conception, une architecture et une programmation permettant d'automatiser le plus possible les rejeux en phase de tests, ce qui requiert une grande force de caractère dans l'euphorie des débuts ; c'est un critère fondamental de choix du langage et, surtout, de son environnement de tests : « debugger » symbolique, couvertures, « profiler », etc. ; la « beauté » et la modernité d'un langage le sont souvent au détriment de ces caractéristiques trop souvent jugées « ringardes ». L'utilisation des traits de langage dynamiques (allocation*

[2] F.Brooks, *Mythical man-month*, donne une règle équivalente : 33% planning (il y inclut les spécifications fonctionnelles), 17% codage, 25% test unitaire, 25% tests d'intégration. Hewlett-Packard a communiqué : 18% expression de besoin et spécification, 19% design, 34% codage, 29% VV&T.

[3] Cette loi dit que : « Work expands to fill the available volume ».

[4] Cf. les statistiques communément admises de répartition des défauts logiciel ci-dessous.

mémoire, exceptions, etc.) se fait toujours au détriment de la testabilité du programme et augmente inexorablement le non déterminisme, ce qui se traduira par un nombre important d'erreurs non reproductibles[5] ! Ils ne doivent être utilisés que lorsqu'ils sont prouvés indispensables.

- Le temps qui s'écoule entre le *T0* du projet et la fin de programmation donne une bonne approximation du temps nécessaire pour effectivement terminer le projet :

  ➘ *En fin de programmation on est généralement à ≈ 50% du délai.*

- Les revues de conception et de programmation (relecture du code source par des pairs) bien conduites (i.e. sans complaisance !) permettent généralement de détecter plus de 50% des erreurs.

- 350 instructions source sans commentaire est une bonne productivité moyenne *par personne et par mois* pour un projet de complexité moyenne. La durée moyenne est $\approx 0.5 \times \sqrt[3]{Effort(EnHA)}$.

  ➘ *Pour des applications à caractère technique et à dominante algorithmique (volume entre 500 et 1.000 KLS) on peut prendre en moyenne :*

  - *Programme simple de complexité linéaire : 4.000 LS/HA,*
  - *Programme à forte combinatoire : 2.000 LS/HA,*
  - *Programme dépendant de l'environnement (Cf. contrôle et/ou temps réel) : 1.000 LS/HA,*
  - *Programme à tolérance de pannes et très fortes contraintes[6] : 250 LS/HA .*

  Pour un programme composite il faut calculer une moyenne pondérée.

- Une stratégie de tests avisée exige d'exécuter tous les tests disponibles à chaque modification du programme, soit un nombre de passage égal à $\frac{n \times (n+1)}{2}$ *n* étant le nombre de tests disponibles. En l'absence d'automatisme pour la non régression, cela conduit à un nombre de tests qui croît comme $\approx k \times \sqrt[2]{Effort\,de\,Test(EnHA)}$. Les tests d'un logiciel font partie intégrante de ce logiciel : **il faut les gérer comme tel**.

- Seule la **courbe de maturité du logiciel** observée renseigne véritablement sur le niveau de qualité atteint suite aux efforts de test et de validation (Nombre d'erreurs découvertes par unité d'effort). Un taux résiduel de 1 à 2 erreurs par milliers de LS est considéré comme une

[5] Cf. mon ouvrage, *Puissance et limites des systèmes informatisés*, chez Hermès.
[6] Pour ces deux derniers types de logiciel, cf. DO-178B / ED-12B et/ou les ITSEC.

bonne performance sur de gros logiciels[7]. Pour faire mieux, il faut mettre en œuvre des techniques de tests sophistiquées, associées à une excellente stratégie de test pour cibler les efforts sur des objectifs précis et éviter les doublons très fréquents.

☐ 3 - La relation économique Coût Qualité Fonctionnalité Délai *CQFD* se stabilise dans l'ordre suivant :

Coût
- Fixé par le client dès le début.

Qualité
- Dépend des actions du chef de projet, et en particulier de l'effort de tests; en théorie, elle est fixée dès que le plan qualité est approuvé, généralement en début de projet. Il est particulièrement malvenu et maladroit de réviser le plan qualité à la baisse !

Délai
- Fixé par le client qui en général synchronise le travail avec d'autres projets ; peut varier en cours de projet.

Fonctionnalités
- Service rendu (i.e. fonctions offertes) proposé par le maître d'œuvre à son client ; les fonctionnalités peuvent souvent être négociées en contre partie du coût et du délai.

- Le chef de projet avisé, évitera la dérive fonctionnelle en contrôlant soigneusement les fonctionnalités proposées ; certaines fonctionnalités (surtout celles qui sont globales et diffuses comme la haute disponibilité, la sécurité, les performances, etc.) ne requièrent que quelques phrases dans la spécification des exigences, mais des milliers de lignes de programmation. Il fera une chasse sans merci aux embellissements inutiles pour l'utilisateur, mais qui complique singulièrement l'effort de tests (comme les IHMs, qui, s'y l'on n'y prend pas garde, deviennent très rapidement complètement subjectives et sujet à changements au gré de l'humeur de qui les regarde).

- En cas de dérapage, il choisira toujours de négocier sur les fonctionnalités (on peut toujours incriminer le client si le **C**ahier **d**es **C**harges (CdC) n'est pas parfait; il l'est rarement !) plutôt que sur la qualité qui ne dépend que de lui, et qu'il faudra payer de toute façon.

- L'estimation du coût est un exercice particulièrement difficile et risqué. A coté des modèles d'estimation analytique comme ***COCOMO*** ou les ***Points de Fonctions*** il faut surtout savoir user de son bon sens et de son expérience : comparaison avec des projets analogues, consultation de vrais experts[8] (se méfier particulièrement des « gurus » et des prophètes ; ne jamais oublier que « les conseilleurs ne sont pas les

[7] Cf. Bug statistics and taxonomy, in B.Beizer, *Software testing techniques* ; la moyenne est ≈ 3 à 5 par KLS.

[8] C'est à dire ayant une expérience de terrain incontestable. Se méfier tout particulièrement des connaissances livresques mal digérées.

payeurs »), pratiquer systématiquement l'analyse *Top-down* (si l'on connaît l'architecture du système).

➘ *Dans tous les cas de figure, le plus grand risque est l'absence de méthode.*

- Il y a une grande différence entre le coût d'un développement à exemplaire *unique* (le forfait classique) et le coût de développement d'un produit (exemplaires *multiples*) qui implique une AQ totale.

  - un coefficient de dilatation de 2 est un minimum ; si le produit est un produit système (par exemple des Télécoms), il faut encore prendre un ratio 2, soit ≈ 4 par rapport au nominal ! .

- Dans la hiérarchie des coûts, l'**innovation** est désormais un luxe que très peu de projets peuvent supporter :

  ➘ *La sagesse consiste principalement à réduire l'innovation au strict minimum car le manque de culture ou d'information (i.e. faible maturité selon l'échelle CMM/ISO SPICE) conduit souvent à réinventer la roue.*

  La conception par **imitation** et le **portage** de programmes, lorsque c'est possible, sont d'excellents moyens d'éviter les déboires de l'innovation :

  ➘ *Utiliser systématiquement les collections d'algorithmes comme le « Art of computer programming » de D.E.Knuth ainsi que les bibliothèques d'API standards.*

  La **duplication** pure et simple de programmes (réutilisation) minimise l'ensemble des coûts, mais n'évite pas l'intégration.

  ➘ *L'achat de progiciels « sur étagères » largement diffusés et bien homologués est généralement la solution de loin la plus économique en dépit des incertitudes inhérentes à cette approche.*

☐ 4 - La productivité du développement dépend beaucoup plus de la compétence des personnes et de l'équipe que des outils.

➘ *C'est ce que B.W.Boehm, et beaucoup d'autres experts, enseignent depuis le début des années 80s et que confirme toutes les statistiques sérieuses[9].*

- Le chef de projet avisé ne se laissera pas séduire par le discours outils, mais cherchera quels sont les vrais talents de son équipe et affectera les tâches en conséquence. Il sera attentif au « moral » de ses troupes. Il sait que 80% de la contribution viendront de 20% des contributeurs, mais ne négligera pas pour autant les 20% restant, faute de quoi le projet ne sera jamais terminé.

[9] Cf. le rapport du Standish Group : *Chaos, charter the seas of information technologies ; à http://standishgroup.com*

- Il faut se méfier tout particulièrement des outils « sophistiqués » qui ne font souvent que rajouter de la complexité (et donc des causes d'erreurs) là où il y en a généralement trop. Un bon outil est un outil simple (mais non simpliste !) qui ne nécessite pas un effort d'apprentissage excessif. Un outil largement répandu, disponible sur de nombreuses plates-formes doit être préféré à tout autre.

☐ 5 - Le plan de développement requerra toute l'attention du chef de projet

- C'est un instrument de dialogue, et donc de négociation avec tous les acteurs du projet ; il doit être aussi exhaustif et précis que possible.

- C'est un moyen de communication irremplaçable avec les membres de l'équipe de développement : chacun y voit le rôle qu'il joue et les dépendances qu'il a avec ses collègues.

- Le plan est un instrument de mobilisation permettant de canaliser les efforts de chacun ; il désigne clairement l'objectif à atteindre et les difficultés du chemin critique.

  ➘ *le chef de projet avisé s'assurera de la bonne compréhension du plan auprès de chaque membre de l'équipe.*

- Un bon plan[10] est un instrument de suivi permettant de vérifier la conformité des prévisions par rapport au réel. Le chef de projet essayera de quantifier le plus possible les jalons intermédiaires (nombre de pages de documents, nombre d'écrans, lignes de code, volume de tests, couvertures, etc.).

- Parmi les différentes mesures que lui permettra son plan, il s'attachera tout particulièrement à la **détection des dérives** :

  ➘ *un retard peut se combler par un surcroît d'effort ; une dérive est **toujours** révélatrice d'un **vice profond** ; c'est le syndrome du projet ou de la tâche qui reste stationnaire à 90% où chaque semaine qui s'écoule entraîne une semaine de retard[11] ! **La cause d'une dérive doit toujours être analysée et éradiquée**.*

- Le plan doit exhiber quelques jalons externes, dit *jalons de confiance*, permettant de montrer au client que le développement avance, faute de quoi le client aura tendance à s'inquiéter et à générer du bruit de fond et des perturbations (i.e. une perte de temps du point de vue du projet).

  ➘ *La tenue de ces jalons est fondamentale pour la crédibilité du projet : ils doivent être précédés de jalons internes permettant de vérifier préventivement que tout est bien en place.*

[10] Le général Eisenhower, au jour J, disait : « The plan is nothing ! Planning is everything ».
[11] Loi de Brooks: «How does a project get to be a year late?… One day at a time».

- Le plan doit être tenu **à jour**, faute de quoi ce n'est plus un plan ! Le processus de planification doit être conduit en y intégrant soigneusement les contraintes organisationnelle et humaine. Trop de planification tue le plan et démotive l'équipe de développement. Une mise à jour du plan une fois par mois est une moyenne raisonnable car l'effort consacré à remplir des formulaires l'est au détriment des tâches directement productives. Dans un projet, il n'y a que des compromis, et des vases communiquants.

☐ 6 - Le chef de projet doit toujours se préoccuper et, au minimum, être informé du contexte organisationnel de son client.

- Il est fréquent qu'en période difficile (Il y en a toujours dans les projets réels !) le client ait des arrières pensées ou essaye de tirer partie d'un rapport de force qui lui est favorable.

- Pour le client (MOA), le chef de projet est le garant technique que la réalisation sera menée à bon port avec le niveau de qualité voulue : le chef de projet doit donc fortement contribuer à sécuriser le client (MOA).

- Dans la phase de démarrage du projet (rédaction du cahier des charges, spécification, conception générale) le chef de projet doit travailler en étroite relation avec le responsable commercial de l'affaire (Il est toujours recommandé de ne pas mélanger les genres : la technique est une chose qui nécessite savoir-faire et expertise, le commerce en est une autre ; l'un peut être le recours de l'autre), et ils doivent se répartir les rôles en cas de négociation (retard de livraison, compléments de travaux, avenants, etc.). Négocier des retraits de fonctionnalités, c'est relativement facile en début de projet ; c'est toujours très difficile, et catastrophique pour l'image, en fin de projet.

☐ 7 - Le chef de projet s'investira totalement (et surtout prioritairement) sur les phases critiques du cycle de développement là ou l'incertitude est maximale.

En particulier:

- La transition *Expression du besoin/Exigences système → Expression fonctionnelle* qui est généralement la source d'ambiguïtés majeures entre le maître d'ouvrage (MOA) et le maître d'œuvre (MOE).

  ➘ *Si le CdC n'est vraiment pas bon (ce qui est relativement fréquent, car la rédaction d'un bon CdC n'est pas facile !), il est recommandé d'écrire un* ***CdC de réalisation*** *que le chef de projet présentera à son client le moment venu comme le vrai contrat.*

- L'architecture du projet (i.e. le résultat de l'activité de conception générale de tout le projet et les interfaces, la conception détaillée des

modules critiques) car il est convaincu que la « représentation » des entités système sous une forme ad hoc (SADT, SART, IDEF, UML, etc.) est l'essence même de la programmation.

➘ *Par rapport à la règle 40-20-40, l'effort correspondant peut ne représenter que 10 à 15%, voire moins, mais cet effort, très localisé dans l'espace et dans le temps, conditionne totalement la tenue globale du projet. On retrouve ici un exemple de la distribution statistique de Pareto (dite 80-20): 80% des problèmes ou de la difficulté viennent de 20% des modules ; le chef de projet devra organiser son équipe en connaissance de cause.*

- Le chef de projet veillera tout particulièrement aux aspects transversaux et diffus (Cf. les caractéristiques non fonctionnelles selon ISO 9126) : facilité d'emploi, performance, fiabilité, disponibilité, maintenabilité, portabilité ; ainsi que les aspects système, (cf. NASA *System engineering handbook* et IEEE 1220 Std[12], *Standard for Application and management of the system engineering process* ainsi que ANSI/EIA 632, *Processes for engineering a system*.) que l'on a tendance à repousser en fin de projet en vertu du (très mauvais !) principe que cela n'est pas immédiatement palpable (on ne « démontre » pas la maintenabilité !) et que de toute façon, on aura bien le temps en fin de projet ou pendant la recette.

  ☞ *La méthode n'est **jamais** un substitut à l'intelligence et à la créativité, mais c'est un garde-fou irremplaçable.*

THE METHOD WON'T SAVE YOU
(but it can help !)

Quelques références utiles :

- IEEE SOFTWARE ENGINNEERING standards collection.
- Normes ISO/AFNOR : ISO 12207/AFNOR Z67-150 *Cycle de vie logiciel* ; ISO 9126 *Caractéristiques qualité des produits logiciels* ; ISO 15504 SPICE *Software Process Improvement Capability Evaluation*.
- B.Boehm, *Software engineering economics*, Prentice Hall.
- W.S.Humphrey, *Managing the software process* et *A discipline for software engineering*, SEI series in software engineering, Addison Wesley.
- J.Printz, Cours de gestion de projet du CNAM/CMSL.
- J.Printz, *Le génie logiciel*, Que sais-je N°2956, PUF
- J.Printz, *Puissance et limites des systèmes informatisés*, chez Hermès.
- J.Printz, *La productivité des programmeurs* et *Coûts et durée des projets informatiques*, chez Hermès.

[12] Cf. IEEE standards collection, *Software engineering*, dernière édition 2004 (existe en CD-ROM).