CASSANDRA CROSSINGS
Bureau indépendant d'analyses et d'études

Vous reprendrez un peu de supervision ?

Une étude technico-sociale

Cassandra Crossings, environ décembre 2025

Notre expérience professionnelle nous a donné l'occasion d'observer attentivement les circonstances de la production d'une application informatique, des prémisses de la conception jusqu'aux problèmes de pérennité.

Avant de proposer des solutions, nous allons passer en revue les processus de développement informatique.

++++

Quand le sage montre la lune, le fou regarde le doigt...

Dans cette histoire, le sage, c'est le client !

Il émet un vœu : il demande au fou d'aller lui chercher la lune. Et la lune qu'il désigne avec insistance, c'est un logiciel correspondant à ses attentes et besoins.

Le client a besoin d'une application capable d'effectuer certaines tâches. Il dresse un état des lieux et, sur la base de ces constats, rédige un cahier des charges qu'il remet au fournisseur.

Tous deux conviennent de leurs attentes et obligations respectives. Le client s'engage à fournir au développeur tous les éléments que celui-ci demandera, et l'informaticien s'engage à construire une application qui tienne compte des informations fournies.

À la fin du travail, le fournisseur mettra à disposition du client le produit ainsi créé et le client paiera au fournisseur la somme convenue.

Le fou, c'est le monde informatique.

Face à la demande du client, le monde informatique s'engage à fournir un travail de qualité respectant les attentes du client. Pour ce faire, il va suivre une procédure, un canevas de travail qui a été validé par l'expérience de son corps de métier. La structure choisie est censée agir comme un garde-fou pour prévenir les dérives éventuelles.

Le monde informatique n'est pas différent des autres. Lorsque la nécessité d'un encadrement du travail s'est faite sentir, le monde informatique a observé les solutions des autres domaines professionnels et a bâti sa propre solution en se basant sur les analogies qui semblaient les plus pertinentes au moment où ces réflexions avaient lieu.

Nous allons passer en revue ces solutions que le monde informatique a mises en œuvre ces dernières décennies et nous allons examiner si elles ont tenu leurs promesses, c'est-à-dire si elles ont aidé le fou à fournir au sage ce que ce dernier lui avait demandé…

Situation durant la décennie 1960

Vers le milieu de la décennie, se produit une bascule : le prix du logiciel dépasse celui du matériel et l'écart entre eux ne cessera jamais de se creuser.

En 1968, la première Conférence de l'OTAN sur l'Ingénierie Logicielle dresse un état des lieux peu brillant :

Les programmes sont livrés en retard et les budgets ne sont pas tenus.

Mais ces désagréments auraient pu être acceptés si les logiciels n'avaient pas présenté des tares plus lourdes de conséquences : les programmes ne correspondent pas à ce qui a été demandé. Leur utilisation est trop complexe et les bugs sont fréquents.

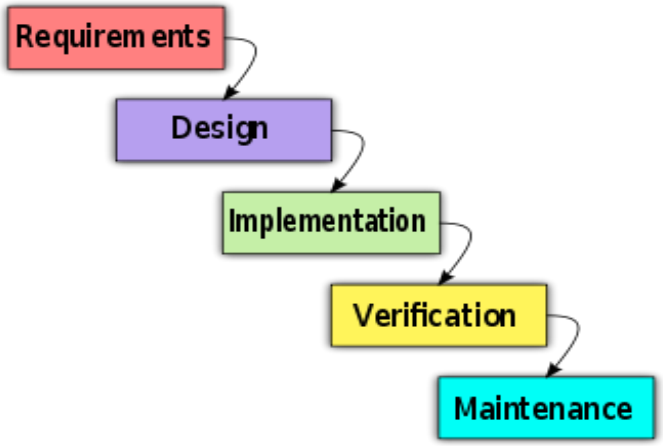
Cerise sur le gâteau (si l'on ose s'exprimer ainsi) : la structure interne des programmes est anarchique, la maintenance est très difficile, le débogage ardu et, par conséquent, l'évolutivité ne peut être garantie à moindre coût.

Dès ce moment, la diminution de la tolérance à l'égard des erreurs de conception et de programmation va entrainer une prise de conscience de leurs conséquences financières et organisationnelles. La pression croissante pour une meilleure qualité des logiciels va entrainer l'émergence de nombreuses réponses plus ou moins efficaces.

Situation durant la décennie 1970

La réponse à ces critiques justifiées a été la création d'un cadre de travail destiné à structurer le développement effectué par le fournisseur.

La première structure mise en œuvre a été la méthode Cascade (Waterfall), inspirée des méthodes en usage dans le génie civil.



Chaque étape doit procéder d'un ordre logique : on demande d'abord au client de préciser ses attentes; puis on dessine les plans; ensuite, on construit les fondations, puis les murs, puis le toit. Et lorsqu'une étape a été franchie, elle est considérée comme définitivement achevée. D'où le nom de "Cascade" : après avoir franchi la chute, on ne revient plus en arrière.

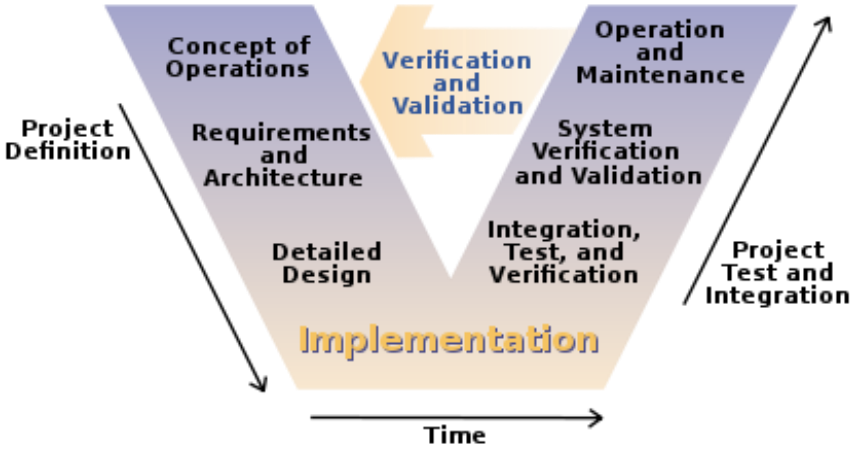
À côté de ces travaux destinés à répondre plus efficacement aux contraintes du fournisseur, des recommandations ont vu le jour au sein de la communauté informatique : l'ajout de commentaires dans le code pour en faciliter la compréhension, la versionalisation des travaux pour faciliter le retour à une situation antérieure stable, la définition de procédures de test explicites et bien documentées afin d'assurer une supervision reproductible de l'avancement des travaux. Ces recommandations sont toujours d'actualité.

Situation durant la décennie 1980

Le modèle en cascade présentait une très grande rigidité temporelle et organisationnelle. Chaque étape devait être menée à terme avant le démarrage de la suivante, du moins en théorie. En conséquence, toute remise en cause ultérieure d'une décision prise dans une étape précédente était très ardue.

En particulier, les vérifications effectuées tout à la fin du processus pouvaient faire apparaître des problèmes dont la correction était très difficile et très coûteuse. Chaque correction pouvait également impliquer qu'une partie plus ou moins importante du travail était perdue.

Le modèle appelé "V", né de l'évolution du modèle Cascade, a permis de mieux prendre en compte ces contraintes.



La branche descendante du V représente la conception et la programmation qui part de l'abstraction des concepts généraux pour descendre dans la concrétisation des éléments distincts.

La branche ascendante représente les tests dont le niveau de généralisation peut être mis en parallèle avec l'élément correspondant de la branche descendante. Aux éléments distincts qui sont programmés alors qu'on atteint le pied de la branche descendante, correspondent les tests unitaires de ces mêmes éléments. Si une erreur apparaît à ce moment, la boucle de rétroaction en vue de corriger l'élément en question est très courte, c'est-à-dire très économique.

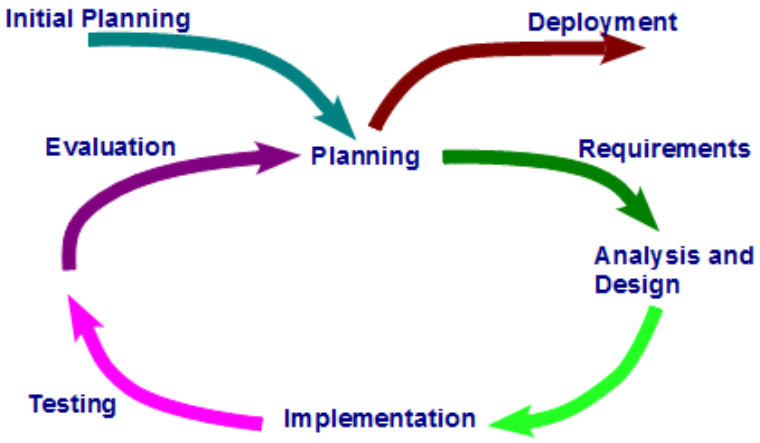
L'étape suivante de la branche ascendante correspond à la validation des modules. Si un module est défaillant, la boucle de rétroaction sera plus longue car elle va impliquer la correction d'un ou plusieurs éléments présents dans le module concerné. Dans la pratique, plus l'avancement du travail remonte le long de la branche ascendante et plus la probabilité d'une erreur nécessitant une rétroaction correctrice est faible. Mais plus cette éventuelle correction sera lourde de conséquences.

Situation durant la décennie 1990

La réalisation de logiciels toujours plus complexes et intégrant de nombreux modules a mis au jour une tare importante du modèle V : la modification d'un module est certes coûteuse, mais le problème est nettement aggravé lorsque la correction d'un module entraîne une cascade de corrections dans d'autres modules de l'application.

Dans une telle situation, l'augmentation des coûts peut devenir littéralement explosive. De plus, ce n'est qu'en dernière étape que le client peut enfin examiner si le travail correspond à ses attentes.

Une évolution du modèle V a amené la création du modèle incrémental en spirale. La famille des processus itératifs actuels tels que Rapid Application Development, Scrum ou Extreme Programing est issue de ce modèle incrémental. Ils sont pour la plupart regroupés sous l'égide du Manifeste Agile.



Le Manifeste Agile

Le courant de pensée Agile regroupe plusieurs méthodes qui reconnaissent des valeurs et des pratiques communes. Ces méthodes revendiquent une approche pragmatique impliquant fortement le client tout au long du processus de développement.

Les valeurs Agile

Les 4 valeurs fondamentales sont :

- **L'équipe** : selon Agile, les individus qui composent le groupe de projet ont une importance supérieure à celle des outils mis en œuvre et l'esprit d'équipe est fortement encouragé.
- **L'application** : Le fonctionnement correct de l'application est vital. Il vaut mieux une application qui fonctionne qu'une documentation complète.
- **La collaboration** : Le client est impliqué dans le développement. Il doit collaborer avec l'équipe et fournir un feed-back explicite sur l'adéquation du produit à ses besoins.
- **L'acceptation du changement** : La planification et la structure du produit doivent être flexibles afin de permettre l'adaptation du produit à l'évolution de la demande du client.

Les principes Agile

Ces 4 valeurs sont déclinées en 12 principes communs à toutes les méthodes Agile :

I. La satisfaction du client, via la livraison régulière de fonctionnalités à grande valeur ajoutée.

II. L'ouverture aux changements, même tardifs, pour donner un avantage compétitif au client.

III. La livraison fréquente d'un logiciel opérationnel selon des cycles les plus courts possible.

IV. La collaboration entre les utilisateurs ainsi que leurs représentants et les développeurs tout au long du projet.

V. La motivation des personnes impliquées.

VI. Le dialogue, si possible en face à face.

VII. Un logiciel opérationnel est la principale mesure d'avancement.

VIII. Un rythme de développement soutenable.

IX. Une attention continue à l'excellence technique et à une bonne conception.

X. La simplicité – c'est-à-dire l'art de minimiser la quantité de travail inutile – est essentielle.

XI. Les meilleures architectures, spécifications et conceptions émergent d'équipes auto-organisées.

XII. À intervalles réguliers, l'équipe réfléchit aux moyens de devenir plus efficace; puis règle et modifie son comportement en conséquence.

Toutes ces valeurs, tous ces principes sont censés garantir la fourniture d'un produit correspondant aux vœux du client puisque ce dernier a été impliqué du début à la fin du processus et que les livraisons partielles et les contrôles à chaque étape devaient assurer l'adéquation fine de l'application par rapport aux attentes et besoins du client.

Dans ces conditions, comment expliquer des échecs aussi retentissants que le fiasco de l'environnement informatique de l'avion militaire F-35 ?

Comment la Confédération Suisse explique-t-elle l'abandon du développement de programmes ayant englouti des centaines de millions CHF ? (AFC INSIEME, 150 mio CHF ou DDPS FIS, 700 mio CHF)

Ou le naufrage en cours de la nouvelle application de gestion des chômeurs du SECO (Suisse, Secrétariat d'Etat à l'économie) application dédiée à la gestion des chômeurs nécessitant plus de 6 mois de correctifs?

Notre connaissance de ce monde nous a appris que les lois du développement informatique ont cessé d'évoluer après la mise en œuvre des environnements itératifs Agile.

Dans son œuvre vieille de près de 300 ans, Charles Louis de Secondat, baron de Montesquieu, nous enseignait déjà que "Si une loi est mauvaise, il faut la changer."

Qu'attend le monde informatique pour changer ses lois ?

Nous allons répondre à cette question très légitime dans les prochains paragraphes et notre réponse sera aussi explosive que l'augmentation des coûts des programmes informatiques incriminés.

Nous mettrons en évidence les raisons qui interdisent l'amélioration de la qualité des applications informatiques, nous expliquerons pourquoi les budgets ne peuvent être que dépassés à multiples reprises. Nous démontrerons pourquoi nous retombons dans une crise informatique d'une ampleur plus vaste que celle de la crise originelle de 1968 mais nous donnerons aussi les clés des remèdes à appliquer.

Crise ? Quelle Crise ?

Comprendre la crise

(NDLA : cette analyse date de 2007, nous la reprenons sans changement puisque la situation qui avait justifié sa rédaction n'a pas fondamentalement changé.)

Plus de 40 ans après la conférence de l'OTAN, les techniques utilisées pour développer les logiciels n'ont pas permis de répondre à ces questions et la complexité croissante des grands systèmes a aggravé le problème.

Comparaison des modèles

Nous avons analysé ces modèles et déterminé comment ils réagissaient aux failles connues sous le nom de «crise du logiciel».

	Gestion globale	Test	RETEX (retours d'expérience)	Exigences	Budget
Waterfall	OK	-	-	-	Échec
V-Model	OK	+/-	-	-	Échec
Iterative / Agile	OK	OK	+/-	+/-	Échec

Le modèle **Waterfall** ajoute un contrôle global sur le cycle de vie du projet mais ne résout pas les autres problèmes, notamment le fait que les tests à la fin du processus ne permettent de détecter les erreurs que lorsque le budget n'est plus suffisant pour les corriger.

Le **modèle V** répond également au contrôle global et met en place une procédure de test efficace pour les petites parties du logiciel (tests unitaires) mais ne permet pas de résoudre le problème des défauts dans un module entier. Avec le modèle V, il est possible de détecter les erreurs et de les corriger dans les limites du budget, à condition que ces erreurs soient de petite taille et d'impact réduit, c'est pourquoi le test est noté (+/-). Lorsque les erreurs sont plus importantes, le contrôle du budget échoue.

La famille des **modèles itératifs** a un contrôle global plutôt efficace et teste précisément le code écrit. Les retours des utilisateurs sont injectés dans les itérations suivantes pour corriger les défauts. Ces retours, atout majeur des familles itératives, autorisent un respect plus ou moins bon des exigences de l'utilisateur, c'est-à-dire que pour répondre précisément aux exigences, il peut être nécessaire de faire plus d'itérations que prévu, ce qui peut entraîner un dépassement de budget. C'est pourquoi les exigences sont notées (+/-) et le budget échoue.

Voilà pour la théorie.

Dans la pratique, les modèles Waterfall et V ont des limitations si criantes qu'ils ont été abandonnés en faveur des modèles itératifs. Mais même ces modèles censés corriger tous les défauts échouent lamentablement, entrainant parfois l'abandon total du projet.

L'échec des modèles itératifs expliqué

L'échec des modèles itératifs est dû à 2 causes sans lien entre elles.

Le premier facteur est un effet d'échelle : tous les membres du groupe qui a rédigé le Manifeste Agile étaient des experts de très haut niveau, chacun dans son domaine, ils avaient l'habitude de travailler avec des groupes réduits de suiveurs eux-mêmes de compétence très élevée.

Cette double sélection (groupe de taille réduite et groupe de personnes de haut niveau) a entrainé un effet particulièrement pervers : toutes les recommandations émises par les membres du Manifeste ne pouvaient fonctionner qu'à l'intérieur de groupes de travail possédant les mêmes caractéristiques, c'est-à-dire de petits groupes d'experts.

Malheureusement pour les clients qui ont acheté des applications nées sous l'égide du Manifeste Agile, leurs produits étaient construits par des gens ordinaires travaillant à l'intérieur de structures de grandes dimensions. Le changement d'échelle ne pouvait mener qu'à l'échec le plus total parce qu'aucune des conditions de création du modèle n'était présente.

Pour être applicable aux grandes équipes formées de gens normalement compétents, le Manifeste Agile aurait dû être écrit par des gens ordinaires membres de grandes équipes. Ils auraient aligné des banalités et personne ne leur aurait accordé la moindre attention.

Les rédacteurs du Manifeste ont été écoutés parce qu'ils étaient des gourous reconnus et respectés et c'est précisément ce statut de gourous d'entre les gourous qui a entrainé l'échec du modèle.

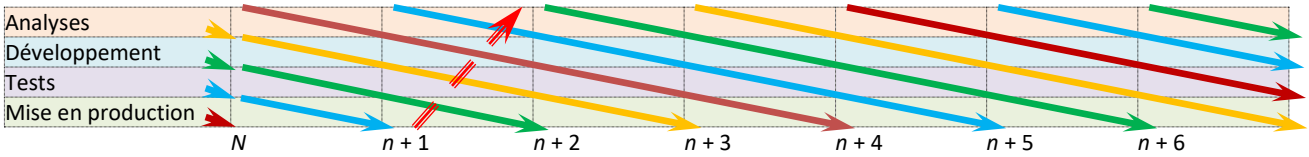
L'autre raison de l'échec des modèles itératifs à fournir des applications de qualité suffisante est le changement de paradigme financier.

Avec les modèles Waterfall et V, le client recevait toujours une application complètement fonctionnelle, développée et testée en tant que .produit fini. Cela laissait aux clients le choix d'acquérir ou de renoncer à une nouvelle version en fonction des nouveautés intégrées dans l'application.

Pour l'acheteur, cela signifiait une grande liberté d'action et de substantielles économies. Pour le fabricant, chaque nouvelle version était une prise de risque. Les nouvelles fonctionnalités allaient-elles rencontrer un public désireux de les acquérir ? À moins que le fabricant n'ait manqué le coche d'une nouvelle niche apparue trop soudainement pour avoir pu être anticipée et qui rendrait le produit obsolète d'un jour à l'autre ?

Le développement itératif a complètement bouleversé le monde informatique.

Désormais, le produit serait en perpétuel développement : la notion de version stable a disparu, remplacée par le sprint censé être la panacée puisqu'il permettait de rapidement intégrer les demandes issues de la clientèle.



Les itérations voient se dérouler les 4 étapes majeures du développement d'un logiciel se dérouler en parallèle. Dès qu'une équipe a terminé les travaux prévus dans une étape donnée, elle transmet son ébauche à l'équipe suivante pour la prochaine étape et débute les travaux prévus pour l'étape ***n* + 1** dans l'ébauche que lui aura transmise l'équipe précédente. On peut littéralement parler d'un développement sur un tapis roulant.
Lorsque survient un problème ; par exemple, si l'utilisateur trouve une erreur dans l'application en production, cette erreur est transmise à l'équipe d'analyse qui disposera d'une itération pour examiner l'erreur et établir une correction, laquelle correction sera transmise aux développeurs pour correction lors de la prochaine étape, laquelle correction sera testée une itération plus tard, etc. jusqu'à ce que l'application soit remise au client pour production. Si on suppose une itération par mois, il va s'écouler 5 à 6 mois avant que l'erreur soit corrigée.

Lorsqu'une nouvelle itération est mise sur le marché, elle est déjà suivie de trois successeresses déjà en cours de développement, chacune à un stade différent. Le retour d'expérience d'une version *n* sera donc intégré dans la version *n* + 4 ou 5. De même, les corrections d'éventuels bugs devront attendre une durée similaire.

Comme nous l'avons déjà décrit dans notre étude sur la **Comparaison des Cultures et des Langages**, nous nous trouvons également face à une discrépance marquée entre le comportement revendiqué dans le discours commercial et le comportement réellement exprimé dans les faits.

Le discours commercial est que le fournisseur travaille pour le bien de son client. Dans les faits, non seulement rien n'incite le fournisseur à chercher à produire de la qualité mais, en fait, tout concours à lui faire produire des applications buggées.

Il n'y a pas de magie noire derrière cette affirmation. Il y a le simple constat que, pour retenir le client prisonnier de son abonnement, il faut lui donner soif pour qu'il continue à payer.

À supposer que (pour son plus grand malheur) un fournisseur sorte une itération parfaitement débuggée et contenant toutes les fonctionnalités désirées par le client, il prend le risque qu'une partie plus ou moins grande de ses clients lui annoncent leur désir de résilier l'abonnement pour demander une proposition de vente de la version en question.

Une telle situation serait dramatique pour ce fournisseur : au lieu d'une vache à traire avec une régularité d'horloge, il se retrouverait face à une soudaine rentrée d'argent très importante mais sans lendemain.

Et si ce fournisseur ne trouve pas rapidement de nouveaux clients pour remplacer ceux qui viennent de s'en aller, il risque la cessation d'activité. En conclusion, une application de qualité est un vrai cauchemar !

Par conséquent, les modèles itératifs sont la réponse idéale à ce cauchemar économique. Ils sont la promesse d'un lien indestructible entre le fournisseur et son client. Hélas la médaille sans revers reste encore à inventer ; pour son malheur, les modèles itératifs ont transformé le client en vache à lait pour le bonheur du fournisseur qui n' a même plus besoin de se fatiguer à produire de la qualité.

Cette situation ne manque pas de nous interpeler car elle ne correspond pas à la réalité du terrain : les besoins en applications informatiques sont si vastes et leur croissance si rapide que les entreprises actives dans le domaine informatique ne vont jamais manquer de clients. Il serait donc tout à fait possible de produire des applications de grande qualité sans avoir à redouter la famine économique.

En fait, tout indique que ce serait plutôt l'inverse qui pourrait se produire, c'est-à-dire qu'un fournisseur qui se distinguerait par une qualité optimale serait très certainement submergé de demandes de développement de la part de clients désireux de s'équiper en applications sans défaut.

En conclusion, le focus placé sur les modèles de développement depuis la Conférence de l'OTAN de 1968 l'a été en pure perte car cette question des modèles de développement était nettement plus liée au contexte économique du développement informatique qu'à l'obtention d'une meilleure qualité.

Mise à jour 2025

Nous avons récemment été invités à participer à la revue périodique des équipes d'un département informatique et nous avons pu apprécier les changements de gestion intervenus depuis nos débuts dans ce monde au millénaire passé.

L'élément qui nous a le plus surpris est que tout n'est que métriques, tableaux de progression et pourcentages de succès. Où se trouve la place de l'humain, de l'utilisateur-client dans ce fatras d'autocongratulation ??

Bien sûr, cette interminable litanie de témoignages d'atteinte des objectifs flatte ce petit monde mais ce nombrilisme hyperactif masque un détachement de tout ce qui concerne l'expérience de l'utilisateur.

De fait, toutes ces présentations n'avaient que pour but de montrer comment chaque team était aligné avec les étapes présentées lors de la séance du trimestre passé et comment ces résultats du jour servaient de marchepied pour les objectifs du prochain trimestre.

Toutes ces méthodes managériales portent des noms hyperboliques transpirant leur origine transatlantique et toutes les communications sont truffées de jargon qui sert avant tout à séparer celui qui fait partie du sérail du béotien qui n'est pas du milieu…

Mais le plus angoissant reste la prédominance de la pensée étatsunienne à 2 étages dont nous avons démontré les limitations et les dangers dans notre **Étude en rétroingénierie sociale**.

Un responsable a décrit avec fierté la réactivité de son équipe en louant sa capacité à corriger dans un délai très bref les erreurs remontées de la Production.

Un tableau vaut mille mots :

#	**Approche Agile**	**Approche "à la française[1]"**
1 – 1	Une erreur est détectée en Production	L'utilisateur final n'est pas impliqué dans la recherche ergonomique de solutions (en infraction avec les préceptes de l'ergonomie à la française qui sont très différents des préceptes étatsuniens et qui exigent cette participation)
2 – 2	L'erreur est corrigée en l'espace de quelques jours	L'ergonomie de l'activité ne peut déployer ses effets positifs.
3 – 3	Champagne !! 	Les caractéristiques du travail réel ne sont pas prises en compte.
| - 4		Les procédures de test ne peuvent refléter la réalité du travail de l'utilisateur.
| - 5		Le produit fini ne correspond pas à ce dont l'utilisateur a besoin.
| - 6		Même s'il a été développé avec un grand professionnalisme, le produit ne peut dépasser les lacunes de la méthode d'encadrement et faillit à la tâche.
| - 7		N'aurait-il pas été préférable de développer un outil qui soit plus proche des besoins réels des utilisateur ? Le produit n'aurait pas couté plus cher à construire et il n'aurait montré aucun décalage avec les attentes des utilisateurs ni failli à la tâche.
| - 8		Le produit aurait couté moins cher – *puisqu'immédiatement développé tel qu'attendu par les utilisateurs* – et il n'aurait pas entrainé de perte de productivité – *première conséquence de l'erreur en Production* – ni péjoré le Retour sur Investissement.

Il est patent que les outils existent, il suffit de les utiliser. Le premier piège n'est donc pas l'absence de solution mais de volonté de les appliquer.

Il est évident qu'affronter un problème vieux de 60 ans et qui a, par son âge, réussi à convaincre tout un chacun de l'impossibilité de son éradication, est chose peu aisée mais les seuls problèmes insolubles sont ceux qu'on n'affronte pas.

Ce qui est également à relever est que le monde informatique a brillamment réussi, non pas à résoudre le problème, mais à changer la manière de le regarder pour faire croire qu'il était résolu.

[1] L'approche à la française est l'exercice théorique résultat d'une éventuelle mise en œuvre de l'ergonomie de l'activité basée sur l'ergonomie française telle qu'enseignée il y a deux décennies dans quelques universités techniques françaises ainsi que de l'utilisation de logiciels développés à l'aide des outils décrits dans la littérature spécialisée à la fin du millénaire passé.
Nous précisons que ces outils n'ont jamais été mis en production à l'échelle industrielle, ils sont restés des curiosités de laboratoire.

Un espoir à la crise du logiciel : l'application de haute qualité

Il n'y a rien à attendre des modèles de développement, aucun n'est capable, à lui seul, d'apporter une solution à la crise de la qualité.

La réponse ne viendra, s'il se trouve quelqu'un susceptible d'y prêter l'oreille, que du côté de l'encadrement du projet informatique.

Que demande le client en se plaignant de la crise du logiciel ?

La plainte du client est limpide :

- Les logiciels sont inadaptés, ils ne correspondent pas aux besoins.
- Ils comportent trop d'erreurs, ils sont difficiles à maintenir.
- Ils sont trop chers, les délais et les devis ne sont pas tenus

Et qu'a compris le monde informatique de cette demande ?

Le monde informatique a certainement dû être étonné d'être ainsi pris à partie car, à moins de travailler sciemment mal, chacun est persuadé de donner le meilleur de lui-même. Par conséquent, chacun tient pour évident que son travail est bon.

La première réponse, le modèle Waterfall/Cascade, a été une bonne réponse. La validation du travail effectuée à la fin de chaque étape a permis une progression de la qualité moyenne.

Mais la question majeure du contrôle des coûts subsistait puisqu'une erreur détectée lors des tests finaux pouvait mettre en péril la viabilité du produit. Le problème de l'inaptitude restait entier : le client ne pouvait contrôler l'adéquation du produit à ses besoins que lorsque celui-ci était fini, c'est-à-dire lorsque le budget relatif à son développement était épuisé.

Le modèle V qui a été adopté par la suite corrigeait en partie le problème posé par la découverte tardive des erreurs durant la procédure de test final. Les tests unitaires, puis les tests modulaires permettaient de garantir une application débarrassée de ses erreurs les plus flagrantes et plus facile à maintenir ainsi qu'une certaine maîtrise des coûts; mais la question de l'inaptitude restait irrésolue, car ce n'était que lors de la livraison du produit fini que le client avait la possibilité de découvrir les manques.

L'adoption des modèles incrémentaux et itératifs a permis la correction d'une partie de ces problèmes. En effet, dans ces modèles, un module est livré au client sitôt terminé et testé. Le client reçoit donc son application à valider par briques et morceaux et si l'un d'eux présente un problème, la correction peut être entreprise dans un délai très bref.

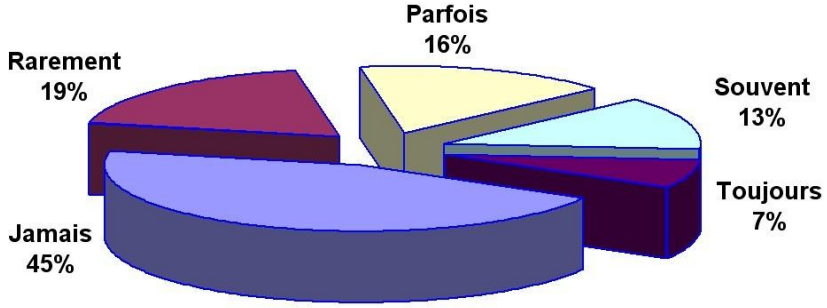
Le problème de l'inaptitude recevait enfin un début de réponse : si un manque était constaté, sa correction pouvait être entreprise dans le cadre d'un financement encore disponible.

Situation actuelle

Malgré toutes les promesses des promoteurs de ces diverses méthodes, la situation dans le monde du logiciel ne s'est guère améliorée depuis la conférence de l'OTAN en 1968. Les projets sont toujours aussi mal gérés, les budgets et les délais régulièrement dépassés, la satisfaction des utilisateurs toujours aussi mauvaise et la qualité toujours aussi médiocre.

Comment un constat aussi sévère peut-il être établi alors que les méthodes mises en œuvre sont censées garantir un développement informatique libre de telles dérives ?

Le CHAOS Report de 2002, édité par le Standish Group, a publié une statistique très intéressante :



Le camembert montre le taux d'utilisation des fonctionnalités par les utilisateurs d'une application.

Comme on le voit, près de la moitié des fonctionnalités développées (et payées par le client) le sont en vain puisqu'elles ne sont jamais utilisées. Le bloc des fonctionnalités rarement ou jamais utilisées représente les deux tiers alors que le bloc des fonctionnalités souvent ou toujours utilisées représente 20% du total.

La question des fonctionnalités jamais utilisées est un problème très important. L'étude du Standish Group porte sur des logiciels *ad hoc* développés pour un usage professionnel. C'est-à-dire que toute fonctionnalité présente dans un tel logiciel a été explicitement demandée et définie dans le cahier des charges qui a servi de base au développement du produit final.

Il y a par conséquent une contradiction flagrante dans le fait qu'une certaine fonctionnalité soit requise, analysée, développée, payée mais pas utilisée.

Fréquence d'utilisation et rejet

Un début de réponse se trouve dans le fait que les fonctionnalités qui sont rarement utilisées sont mal maîtrisées et demandent une attention soutenue lors de leur utilisation. Par conséquent, la charge de travail élevée liée à la mise en œuvre de telles fonctionnalités péjore la productivité.

Le corollaire de ce constat est que l'utilisateur va développer une certaine aversion envers les fonctionnalités qui présentent une certaine difficulté d'utilisation ainsi qu'un taux d'erreur élevé.

Par conséquent, l'utilisateur va développer des stratégies d'évitement pour ne pas avoir à les utiliser. Et comme il ne les utilise pas, il n'a aucune possibilité d'apprendre à les maîtriser.

Nous sommes en présence d'un cercle vicieux :

- Si la fonctionnalité est d'un apprentissage difficile, l'utilisateur va éviter de s'en servir.
- S'il ne s'en sert pas, il n'a aucun moyen d'apprendre à en maîtriser les subtilités.
- Par conséquent, la fonctionnalité a été développée en vain et l'argent investi dans cette fonctionnalité l'a été en pure perte. Un logiciel qui contient de telles fonctionnalités est trop coûteux par rapport à l'usage qui en est fait, le retour sur investissement sera mauvais et tant la satisfaction de l'utilisateur que celle du pourvoyeur de financement seront faibles.

Ce dernier constat est très exactement celui qui a été fait en 1968 lorsque la crise du logiciel a été dénoncée pour la première fois et toutes les réponses fournies depuis ont toutes manqué la cible, quels que soient les mérites réels ou supposés dont les aient affublées leurs promoteurs.

Origine de l'échec des méthodes de développement

Les méthodes utilisées jusqu'ici ont toutes failli à des degrés divers car elles n'ont pas pris en compte la caractéristique fondamentale des produits informatiques : ils sont tous uniques.

Contrairement à ce qui se passe dans l'industrie, il n'y a pas de prototype en informatique. L'objet programmé est exactement identique à celui qui sera utilisé par la suite chez le client. Ce n'est pas une copie au sens industriel du terme qui voit deux objets distincts sortir d'un même moule.

En fonction des aléas de la matière injectée, la durée de vie et la résistance aux sollicitations d'un produit industriel ne sont pas identiques. Le premier peut tomber en panne après 2 jours alors que le suivant offrira plusieurs années de service à l'utilisateur.

Les copies d'un même logiciel installé chez deux clients différents sont, en revanche, un seul et même objet. Si les mêmes manipulations sont effectuées en parallèle, les deux clones failliront de manière identique et simultanée.

À ce titre, la création d'un logiciel est beaucoup plus proche d'une activité artistique, telle que la sculpture, que d'une production industrielle. L'absence de prise en compte de cette caractéristique est à l'origine de la plupart des problèmes connus.

Le malentendu est renforcé par les usages imitant l'industrie : lorsque l'industrie parle de prototype, il s'agit d'un objet qui sort des bureaux d'étude et qui n'a aucun point commun avec le produit fini. Par exemple, le prototype a été usiné pièce par pièce avec des machines-outils ou réalisé avec une imprimante 3D à résine polymère alors que le produit fini sera injecté dans un moule.

Chacune de ces méthodes de fabrication crée des objets dont le comportement peut être complètement différent. Par exemple, la réponse aux contraintes ne peut pas être comparée : l'usinage a tendance à créer des microfissures qui peuvent servir d'amorces de rupture; l'impression 3D a un comportement plutôt isotrope alors que le moulage induit des tensions à l'intérieur de la pièce en fonction de la vitesse de coulée et des circonstances de solidification.

Dans le monde informatique, un prototype devrait s'appeler une ébauche, car tout ce qu'il contient se retrouvera dans le produit fini; de la même manière que la statue terminée contiendra tous les défauts présents sur l'esquisse, le développement *quick and dirty* ne peut en aucun cas donner autre chose qu'un produit final *quick and dirty*.

En effet, tout composant créé dans l'ébauche va rester intégré tout au long de l'évolution du produit jusqu'au stade de la distribution; et chaque imperfection qui y est laissée sera une faille potentielle.

Cette méthode est d'autant plus dangereuse qu'elle est basée sur l'illusion que le code est revu en permanence de manière à y traquer les erreurs. Dans la réalité, un code sale qui résiste aux tests va être intégré tel quel dans le produit fini car le critère d'introduction d'un code dans le produit fini n'est pas la propreté de sa programmation, mais sa capacité à franchir les tests.

Le Manifeste Agile

Ce talon d'Achille démontre que toutes les méthodes qui se réclament du Manifeste Agile ne peuvent produire que des applications de qualité médiocre. Cette faille des méthodes à réalisation itérative ou incrémentale est renforcée par les contingences financières de l'environnement économique. Si l'informatique copiait réellement l'industrie, elle réaliserait un prototype du produit et en réécrirait l'entier du code au moment de rédiger la version finale à commercialiser.

Ne serait-ce que pour des questions financières, une telle méthode est irréaliste, bien qu'en théorie, elle soit capable de garantir un code débarrassé de la quasi-totalité de ses erreurs.

L'explication de l'échec des méthodes Agile réside dans le fait que ces méthodes partent du principe que tout code sera révisé au moins une fois au cours du processus du développement. En conséquence, aucun intervenant n'est soumis à une pression de qualité puisque chacun est en droit de croire que la charge de détecter les éventuelles erreurs repose sur un contrôle ultérieur implicitement prévu par la mise en œuvre du processus Agile.

La définition du cadre de travail est primordiale car seul un cadre explicite basé sur des procédures définies avec clarté est à même de garantir une qualité satisfaisante. Si la qualité repose sur des procédures implicites, elle devient dépendante des humains qui interprètent ces procédures en fonction de leurs idiosyncrasies.

La fiabilité du système ne peut dès lors plus être garantie, puisque la moindre défaillance humaine entraîne automatiquement celle de tout le processus de développement.

Une redéfinition du système

Par conséquent, le système doit être rendu indépendant des personnes et ne reposer que sur les procédures qui seront mises en œuvre par les individus. L'individu s'efface derrière la procédure qui devient le support de la qualité.

Contrairement aux apparences, un tel système est très favorable aux individus qui voient leur niveau de stress fortement réduit. La stabilité du système est renforcée par le fait que les employés peuvent se consacrer à leur tâche en toute sérénité.

Toutes les méthodes actuelles ayant échoué à atteindre une qualité satisfaisante à cause de leur échec à prendre en compte le facteur humain, les chapitres suivants vont s'attacher à examiner quels outils sont les plus adaptés à répondre à ces défis.

Rousseau et les managers

Il semble régner un *a priori* définissant le personnel comme travaillant systématiquement correctement; les erreurs étant des événements malheureux indépendants de la volonté du management comme des exécutants.

L'employé est intuitivement perçu par sa hiérarchie comme une personne qui commet des fautes à l'insu de son plein gré, mais qui est éducable, car elle fait preuve de bonne volonté.

Le corollaire logique de cette situation est que rien n'est entrepris pour tenter de prévenir l'apparition d'erreurs dans le développement du produit.

Ce rousseauisme industriel peut se traduire par : "*L'employé naît bon mais les contraintes du travail le pervertissent. Une éducation appropriée sauvera son âme et améliorera sa productivité*."

A l'opposé de cette mélasse de bons sentiments, le comportementalisme a une position qu'on pourrait qualifier de cynique ou de désabusée : "*L'employé travaille toujours de son mieux, mais il est incapable d'atteindre la perfection. Il n'existe aucune méthode pour sublimer son imperfection originelle. Seule une modification de son environnement peut transcender ses défauts pour permettre l'émergence des qualités désirées*."

Certes, les spécialistes en ressources humaines et la hiérarchie se défendront en arguant qu'ils disposent de nombreuses stratégies très efficaces pour améliorer la qualité. Mais l'analyse des résultats montre que ces pratiques tiennent autant des consignes d'entraînement sportif que du drill militaire et sont peu ou prou redevables à la Méthode Coué.

Leurs succès très relatifs et ponctuels, doublés d'une incapacité flagrante à stabiliser à long terme les leçons de ces réussites, témoignent de la faiblesse de ces approches.

Si les informaticiens étaient des moutons

Nous proposons d'expliciter les problèmes liés à la structure hiérarchique du développement informatique en comparant une entreprise à un troupeau de moutons.

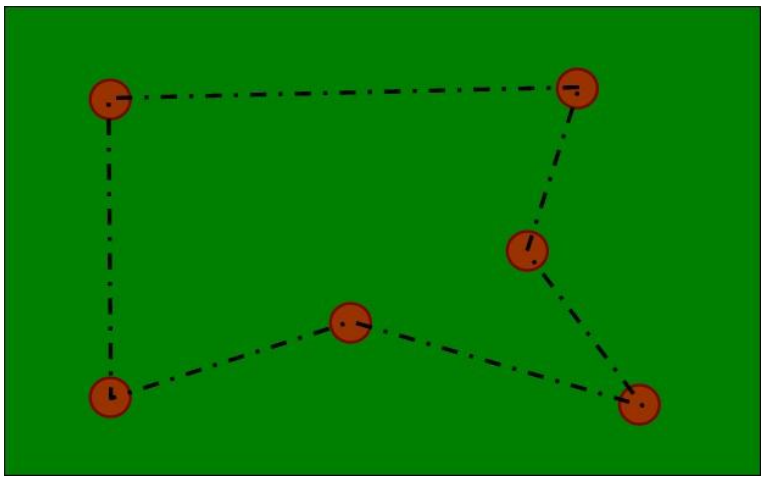
Dans le cadre de cette démonstration, le berger est le miroir de la direction de l'entreprise. L'encadrement hiérarchique, du top management jusqu'au low level management, est symbolisé par les chiens du berger, et les forces productives sont représentées par les moutons qui constituent le troupeau.

Il serait futile de chercher une quelconque correspondance entre la réputation faite usuellement à ces avatars et le comportement réel des individus ainsi décrits. Cette étude n'est pas une satire sociale, au sens où l'ont été les fables d'Ésope ou de Jean de la Fontaine.

L'objet de l'étude n'est pas l'animal ou l'humain mais les relations existant entre les acteurs de cette démonstration.

Le berger et ses moutons

Ainsi qu'expliqué dans le préambule, le fournisseur informatique est comme un berger devant faire transiter ses moutons sur les terres de son client. Il y a des zones où les moutons pourront paître; et des zones interdites ou dangereuses. Le dessin ci-dessous représente un tel territoire.



Le client a ordonné au berger de ne laisser brouter que l'herbe entre les arbres.

Le bon fournisseur, le bon berger devra amener ses employés, ses moutons, à fournir au client le produit qui lui convient, respectivement, à brouter les zones autorisées sans toucher au reste. Pour ce faire, le berger est aidé par ses chiens, tout comme l'entrepreneur dispose d'un encadrement, d'une hiérarchie censément efficace.

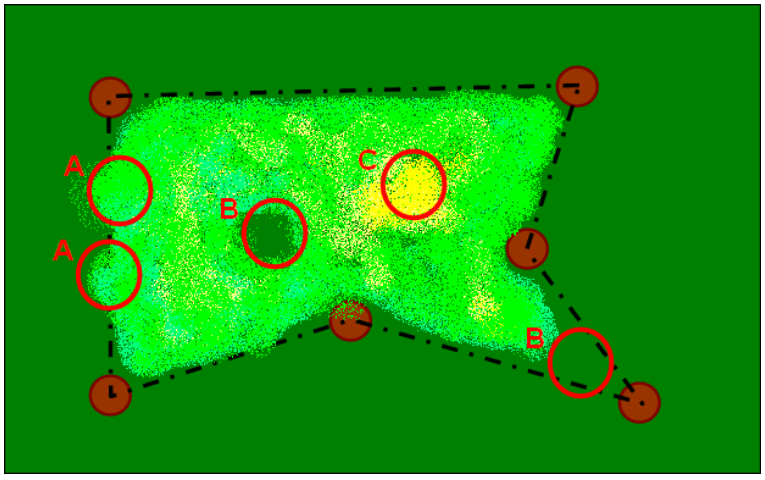
C'est à l'aide de longs sifflements modulés que le berger obtiendra de ses chiens qu'ils guident le troupeau à rester sur le terrain convenu; tout comme la direction édite des consignes et des ordres de service pour que les employés travaillent correctement.

Mais, derrière cette vision idyllique, se cache un stress énorme : l'encadrement déploie beaucoup d'énergie pour contenir le projet dans son cadre, de la même manière que les chiens courent en tous sens pour garder les moutons dans le périmètre autorisé.

Les moutons sont quant à eux stressés par les sifflements et les aboiements qui sont autant de rappels à l'ordre; situation qui correspond également à celle des personnes impliquées dans le développement d'un produit et qui doivent justifier l'avancement de leur travail à chaque séance d'équipe.

Lorsque le travail est accompli

Après le passage des moutons, voici à quoi pourrait ressembler le territoire en question :

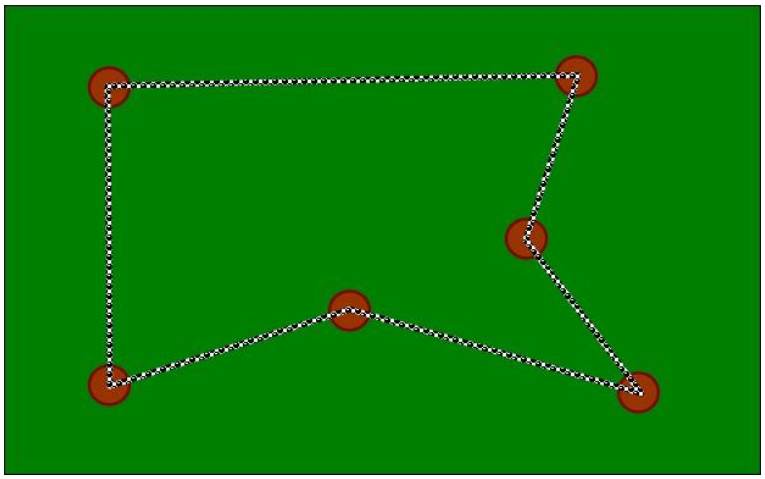


En examinant attentivement cette image, on constate plusieurs problèmes :

- En **A**, les moutons ont dépassé la limite autorisée. Dans le cas d'un logiciel, cela peut se traduire par un défaut de protection de l'intégrité des données, ou une backdoor qui ouvre une faille de sécurité.
- En **B**, les moutons n'ont pas brouté une partie qui aurait dû l'être. Dans le cas d'un logiciel, cela peut se traduire par une fonction manquante, bien que correctement spécifiée dans le cahier des charges remis au fournisseur.
- En **C**, les moutons ont excessivement brouté l'herbe, ce qui va entraîner des problèmes de repousse; en d'autres termes, il y a diminution de la valeur du terrain. Dans le cas d'un logiciel, cela équivaut à un excès de spécification de la part des analystes ou une redondance de fonctionnalité. La conséquence de cet excès est que le client devra payer un prix exagérément élevé pour la fonction en question. Par conséquent, le produit final verra ses coûts de développement dépasser l'optimal, péjorant d'autant le retour sur investissement.

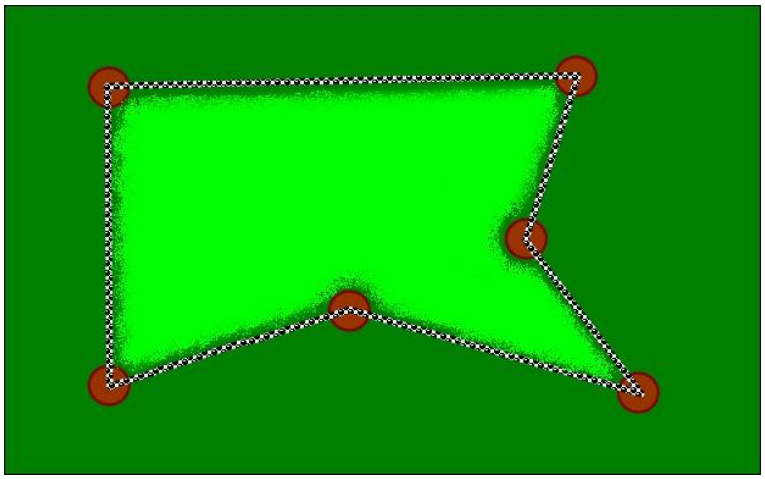
Une solution simple : la clôture !

Le berger n'est pas sans ressources face à ce problème : au lieu d'être stressé de ne pouvoir efficacement maîtriser le comportement des moutons ou de voir le troupeau soumis au stress d'une possible sortie de la zone autorisée, il est beaucoup plus simple de délimiter celle-ci à l'aide d'une clôture.



Alors que le berger rousseauiste espère qu'un jour ses moutons seront suffisamment bien éduqués pour déterminer par eux-mêmes quelles sont les parties à brouter ou non, le berger comportementaliste sait qu'il ne peut accorder aucune confiance à ses moutons, et il n'imagine pas un instant tenter cette possibilité.

Il pose une clôture, car il sait que c'est le seul moyen vraiment efficace pour obtenir de ses moutons qu'ils ne broutent que là où c'est autorisé, avec ou sans l'aide des chiens.



Les forces productives, moutons ou employés, verront leur stress fortement réduit car ils ne seront plus soumis à une pression excessive de la part de l'encadrement.

En revanche, un événement va apparaître dans le monde des employés, qui n'a aucun équivalent dans le monde pastoral : la clôture des moutons sera traduite par un cadre virtuel qui induira une responsabilisation des employés.

Il y a échange d'un stress externe, la pression de l'encadrement, pour un stress interne, le respect du cahier des charges. Mais autant un stress externe peut être vécu comme un facteur négatif qui est subi par l'employé; autant un stress interne peut être vécu comme un facteur positif, car l'employé sait qu'il a la maîtrise de cette contrainte.

Face à cette obligation, l'employé a le choix de la soumission librement consentie. Le cas échéant, la contrainte sera perçue, non plus comme un poids, mais comme une motivation.

Derrière cet exposé, ne se cache rien de nouveau, les études scientifiques sur la soumission librement consentie datent de près d'un demi-siècle et les recommandations qui en sont issues ont déjà démontré leur puissance dans le domaine du marketing.

Cette proposition comportementaliste est semblable à un interféromètre Fabry-Pérot. Même un laser de très grande qualité ne peut émettre un rayonnement parfaitement cohérent. Il est semblable à l'employé qui fait de son mieux mais reste incapable de dépasser un certain plafond de qualité.

Pour augmenter la précision de la longueur d'onde d'un faisceau laser, on adjoint à l'émetteur un interféromètre formé d'une lame mince recouverte de surfaces semi-réfléchissantes.

Cette lame génère des interférences destructrices qui annulent les longueurs d'ondes autres que celles comprises dans la gamme désirée. La gamme de longueur d'onde entrant en résonance avec l'épaisseur de la lame optique sera seule capable de traverser le dispositif optique et le faisceau ressort de l'interféromètre épuré des fréquences non désirées, donc de meilleure qualité.

C'est très précisément ce que nous proposons dans le cadre de cet exposé : réaliser une approche du développement informatique qui s'appuie sur les enseignements de la psychologie comportementaliste pour annihiler les faiblesses intrinsèques de l'humain, avec pour objectif d'obtenir une qualité proche de la perfection.

<u>Une image vaut mille maux</u>

Nous avons démontré qu'un environnement taillé pour la masse et non pas pour d'hypothétiques gourous présente de bien meilleures chances de succès. L'exemple ci-dessus n'est certes qu'une illustration de ce qui est souhaitable mais cet exemple montre surtout qu'il est possible de faire mieux pour autant qu'on aborde correctement la situation et qu'on fixe correctement le but à atteindre.

Nous vous remercions de votre attention.

Addendum mi-avril 2026

Un nouveau-venu plein de promesses ?

Les médias viennent tout récemment de s'extasier sur les prouesses d'un moteur IA qui aurait "découvert" des milliers de failles dans des logiciels existant, pour certains, depuis plusieurs décennies et réputés pour leur solidité.

Il n'y a pas de "rocket science" derrière ces annonces, ce moteur a fait ce pourquoi il est conçu, rechercher des incohérences et, comme tout programme d'ordinateur, il l'a fait, très vite et de manière très fiable. Mais il n'a pu aller au-delà de ses limitations intrinsèques : ordinateur, il est né, ordinateur il reste.

Chercher une incohérence est une chose, la corriger en est une autre et l'ordinateur n'en est pas (encore) capable.

Comme cela a justement révélé par les journalistes qui ont couvert ce sujet, il y a deux conséquences pratiques à cette réussite :

- La première est qu'elle donne aux entreprises qui ont accès à cet outil la possibilité de traquer les failles et les corriger au plus vite. C'est une stratégie "White Hat" (le bon hacker aux nobles motivations.)
- Le corollaire est tout autant valable : l'entreprise propriétaire du moteur en a soigneusement verrouillé l'accès pour éviter qu'il ne soit utilisé par des "Black Hats" (les méchants hackers aux plus sombres desseins.).

Mais ce qui a pu être fait peut être copié et si l'accès à la seule instance actuellement fonctionnelle est actuellement verrouillé, cela ne sera qu'une motivation supplémentaire pour programmer un autre moteur qui répliquera ces fonctionnalités.
Nous devrions donc assister dans un futur proche à un tsunami de cybercrimes.

Y a-t-il un espoir ?

L'espoir est ce qui meurt en dernier, aussi longtemps que nous ne sommes pas morts, il y a toujours un espoir !

Cet espoir est simplement ce que nous venons de décrire ci-dessus : à l'aide d'outils de supervision adaptés, il est possible de créer des logiciels sans défaut simplement en soutenant l'effort des ingénieurs et architectes de manière à prévenir les failles potentielles.
Nous avons déjà créé un démonstrateur qui a pleinement validé cette réalité. Non seulement le travail en a été grandement simplifié mais aussi bien la durée de développement a été fortement raccourcie de moitié que les coûts ont pu être réduits d'un tiers.

Cependant, nous devons souligner un point crucial : autant il est possible pour une IA de passer un code en revue pour y déceler les failles, autant il est conceptuellement impossible de charger une IA de développer un logiciel sans défaut.
La création *ex nihilo* est (et va rester encore longtemps) hors de portée d'un outil automatique, seul un humain peut posséder les capacités d'abstraction nécessaires à la réussite de l'exercice.