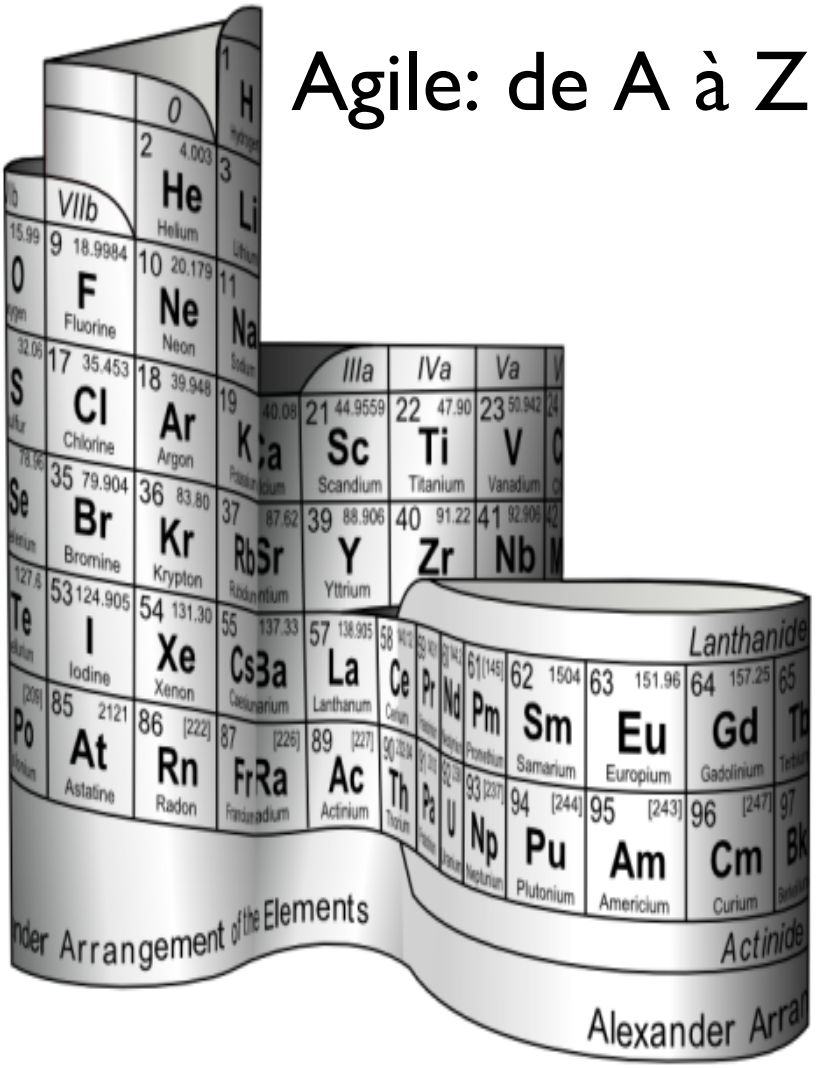
Agile: de A à Z



Référentiel des Pratiques Agiles: édition eBook

Une production de l'Institut Agile

Préface de Pierre Pezziardi, Directeur Informatique, Bred Banques Populaires

Agile: de A à Z

(...ou presque...)

RÉFÉRENTIEL DES PRATIQUES AGILES

EDITION EBOOK

Chapitre 1

Préface

AGILE ?

En ce début d'année 2011, nous sommes à un tournant important du monde de l'informatique d'entreprise. Les méthodes classiques de développement logiciel - le fameux cycle en V où phase après phase un logiciel émerge de spécifications réputées "justes" - ont échoué à adresser la complexité toujours grandissante de nos Systèmes d'Information. La productivité, s'y est dégradée, c'est à dire que le coût marginal et le délai de mise à disposition d'une nouvelle fonctionnalité dans le SI ne fait qu'augmenter. "Trop long, trop cher" sont les principaux griefs adressés aux DSI, dont la durée moyenne du mandat a considérablement diminué, passant en dessous des deux ans en France (source Cigref)...

En marge de ce phénomène généralisé, des éditeurs professionnels ou Open Source obtiennent des résultats à l'extrême opposé : leur productivité est stable malgré la masse croissante de code qu'ils doivent gérer : Google, Eclipse, Firefox, iPhone OS ... Mais ces éditeurs ne fonctionnent plus selon des méthodes classiques, mais au sein d'équipes intégrées produisant en cycles courts des logiciels qualifiés d'utile par leurs utilisateurs.

Mais au fond qui pourrait ne pas le souhaiter ? Qui peut préférer livrer dans des délais longs des logiciels laissant les utilisateurs insatisfaits ? Alors en matière d'informatique, les modes se succèdent, reflets de cette perpétuelle insatisfaction : UML, SOA, RAD, ERP, EAI, MDA, XML, Cloud (was ASP)... et récemment l'Agile et (bientôt) le Lean.

Le marché l'a bien compris. tout le monde doit être agile ! Pourtant ces nouvelles méthodes ne réclament pas qu'un simple coup de peinture, mais l'abandon de règles et d'habitudes profondément ancrées. Le dirigeant qui souhaitera obtenir les résultats des pionniers ne pourra pas se payer uniquement de mots avec "la suite de modélisation XYZ, votre meilleur outil pour agiliser vos équipes", "notre chef de projet MOA agile à 800€ jour", "10 leçons pour devenir agile en 15 jours !". Ce serait reproduire la tragédie de la diffusion du Lean au-delà des frontières de son inventeur Japonais Toyota : en ne reprenant que les pratiques superficielles, sans changer les habitudes managériales qui laissent peu d'autonomie aux équipes, nos industriels occidentaux n'ont tiré que de maigres bénéfices de cette méthode. Pendant ce temps, Toyota est devenu leader mondial, car pour lui le Lean n'est pas un corpus figé d'outils, mais une dynamique quotidienne d'amélioration continue emmenée par la base avec le soutien des cadres.

Si votre objectif est de rechercher la satisfaction de vos clients, en livrant régulièrement des fonctionnalités, sans perturber celles qui fonctionnent déjà, ce recueil de pratiques est fait pour vous. Il ne se lit pas comme un vulgaire manuel d'agilité en 15 leçons, mais comme un jeu de cartes vous permettant de toucher du doigt des principes qui se sont avérés utiles dans de nombreuses équipes, et que vous pourrez progressivement apprendre, puis utiliser - ou pas - dans les vôtres. J'espère que vous prendrez autant de plaisir que moi à le parcourir.

Pierre Pezziardi, 13 avril 2011

Pierre Pezziardi est Directeur Informatique de la Bred Banques Populaires, et auteur de "Lean Management, Mieux, plus vite, avec les mêmes personnes" publié aux éditions Eyrolles.

° ° ° ° °

Table des matières

DEUXIÈME PARTIE: LES PRATIQUES

Chapitre 2

Remerciements

Fédérant un réseau de compétences individuelles reconnues, mandaté par des entreprises développant un savoir-faire et une culture distincte basées sur ces compétences, mais agissant en toute indépendance et neutralité, l'Institut Agile a pour missions de développer en France le marché du "produit" Agile, de présenter une définition claire et cohérente de ce que recouvre le terme Agile, de recenser les attentes du marché, de mettre au point des programmes de collecte d'informations et de mesures sur les effets constatés des pratiques dites Agiles.
Pour plus d'informations: http://institut-agile.fr/

Merci à tous ceux qui ont soutenu le projet de l'Institut Agile, à commencer par les partenaires et soutiens financiers:

axones

EXAKIS

AGILiDEE
Catalyseur d'Agilité

Remerciements personnels de la part de Laurent Bossavit à tous ceux, qui ont à un moment ou un autre donné un coup de pouce crucial: bien qu'il soit impossible faute de place de rendre à chacun l'hommage qui convient, je tiens à saluer Godefroy Beauvallet pour ses conseils et

Xavier Warzee pour y avoir cru le premier. Merci également pour leurs idées et encouragements à Karim, Vincent, Romain, Benoit, Alex, Manu et Manu, Gilles, Jérôme, Arthur, Colin et Alain. Merci pour leur amitié à Bernard Notarianni et Jonathan Perret. Pour leur patience, à Sophie, Axel, Erwan et Mika.

Image de couverture: représentation en volume dite "Alexander", extraite de Wikimedia Commons, par l'utilisateur Bastianow d'après J. Scholten.

Chapitre 3

Guide de lecture

La **première partie** de cet ouvrage en présente les motivations: pourquoi vouloir publier, en 2011, soit dix ans après la parution du Manifeste Agile, et alors que ne nombreux livres déjà parus sur le sujet, un parcours encyclopédique à travers les nombreuses pratiques dites "agiles"?

Si vous souhaitez aller rapidement à l'essentiel, vous pouvez commencer directement par le chapitre 5, présentant le canevas des fiches consacrées aux pratiques Agiles. Si vous préférez prendre le temps de connaître les questions de fond qui sous-tendent le travail réalisé, suivez l'ordre chronologique.

Le premier chapitre aborde donc le positionnement de ces approches Agiles sous l'angle historique, en remontant aux origines de la discipline dont la plupart des personnes concernées se réclament encore aujourd'hui, le Génie Logiciel.

Le chapitre suivant est consacré notamment à décrire le mouvement Agile comme établissant une rupture avec cet héritage; il s'attache à mettre en lumière les différences les plus structurantes entre les deux courants de pensée.

Afin d'éclairer à la fois les enjeux du présent livre et plus largement les perspectives d'avenir qui s'ouvrent, dans l'analyse de l'Institut Agile, à ces nouvelles pratiques, le chapitre 3 établit un bilan des années 2001-2011 et cite quelques défis restant à relever.

Au chapitre 4, nous verrons plus en détail pourquoi les "pratiques Agiles" constituent la matière la plus intéressante produite par ce mouvement au cours de cette décennie.

Le chapitre 5 aborde pour finir le canevas de description détaillée de ces différentes pratiques.

La deuxième partie est consacrée aux pratiques Agiles à proprement parler. Abordées par ordre alphabétique, chacune y est décrite selon le canevas précédemment décrit, de la façon la plus concise mais également la plus claire possible.

Chapitre 4

Au commencement était... le Génie Logiciel

De même qu'il existe une Histoire de France ou une Histoire des sciences, il existe une Histoire, fût-elle brève, de l'activité qui consiste à créer des logiciels. Mais contrairement au deux premières, on a parfois l'impression que ceux qu'elle concernent - nous, les programmeurs, analystes, développeurs, ingénieurs; les testeurs, les chefs de projets, etc. - ne s'y intéressent quasiment pas. Notre profession semble tourner le dos à son passé.

Ainsi des idées techniques nous sont-elles présentées comme "neuves" et "révolutionnaires" alors que, pour qui s'est penché sur l'historique de la discipline, il ne s'agit que de réchauffé: des idées présentes dans tel ou tel dialecte de Lisp ou de Smalltalk depuis trente ou quarante ans. (Si vous ne me croyez pas, penchez-vous sur les comparaisons entre XML et les "s-expressions" ou entre la Programmation Orientée Aspects et les MOP ou Meta-Object Protocols.) On pourrait aussi se demander combien, parmi ceux qui se réclament aujourd'hui du mouvement Agile, ont conscience de ses prédécesseurs, tels le RAD ou le Processus en Spirale dû à Barry Boehm.

De façon générale, notre profession tend à considérer toute idée âge de plus de cinq ans comme obsolète et digne de peu d'intérêt, et c'est sans doute en partie pour cela qu'on a parfois l'impression que l'industrie du logiciel est le jouet de cycles qui relèvent plus de la mode que du progrès.

RETOUR AUX SOURCES

Pourtant, il semble impossible de comprendre les enjeux que recouvrent le terme "Agile", et les raisons qui ont poussé un nombre considérable de professionnels à se mobiliser pour le faire connaître, sans en comprendre les racines lointaine, et son positionnement par rapport à ces origines.

Parmi les idées qui ont façonné les métiers du logiciel, une l'a fait de façon particulièrement forte et durable, l'idée du Génie Logiciel. Le logiciel relève-t-il vraiment d'une activité d'ingénieur? La question, lancinante, revient comme un serpent de mer dans les conférences consacrées au sujet, mais elle est rarement traitée sous un angle historique.

Rares sont ceux qui semblent conscients que le "logiciel" ayant lui-même une histoire d'à peine plus d'un demi-siècle, il a bien fallu inventer l'expression Génie Logiciel; que celle-ci ne va pas de soi, et que l'application des techniques de l'ingénieur au domaine du logiciel relève, non pas des lois de la nature, mais d'une décision à laquelle il a fallu rallier divers groupes: des militaires, des scientifiques, des groupes industriels.

Nous allons donc brièvement rappeler la discrète histoire du Génie Logiciel. C'est une excursion qui a de quoi satisfaire les esprits curieux, même s'il devaient en ressortir plus convaincus que jamais de la pertinence de ce mariage. Mais nous verrons que cette histoire a de quoi alimenter bien des doutes...

DE GARMISCH À ROME

On fait en général remonter l'histoire de la discipline à l'année 1968, et en particulier à une conférence organisée sous l'égide de l'OTAN dans le cadre enchanteur de Garmisch-Partenkirchen, une station de sports d'hiver, aux pieds des montagnes de Bavière.

Il y eut non pas une mais deux conférences de l'OTAN sur le Génie Logiciel, à un an d'intervalle; même les rares auteurs qui font référence à la conférence de 1968 semblent ignorer celle qui se tient en 1969 à Rome.

Les actes des deux conférences sont disponibles sur le Web, dans une version PDF d'une qualité remarquable alors que la plupart des documents de cette époque sont en général simplement scannés, donc non indexés ni "cherchables". Ils méritent d'être lus avec attention. (On y

trouve par exemple ce conseil de Peter Naur qui recommande de s'intéresser aux idées d'un jeune architecte, Christopher Alexander. Le même Alexander qui sera redécouvert vingt anx plus tard par un certain Kent Beck, donnant naissance au mouvement des Design Patterns.)

Structurés d'une façon très systématique, ils couvrent la quasi-totalité des préoccupations encore d'actualité aujourd'hui quant à la façon de mener des projets dans le domaine du logiciel. Environ cinquante experts venus de onze pays sont présents. Parmi les participants on compte des sommités comme Edsger Dijkstra (connu pour sa campagne contre le "goto" et en faveur de la programmation structurée), Alan Perlis (créateur d'Algol et auteur de proverbes qui sont au logiciel ce qu'une certaine tradition japonaise est au jeu de Go) ou Peter Naur (co-crédité de l'invention de la notation BNF pour décrire les langages de programmation).

Ces documents sont éloquents quant au degré de controverse que suscite la question du génie logiciel. Voici une citation d'un participant: "La chose la plus dangereuse dans le domaine du logiciel est l'idée, apparemment presque universelle, que vous allez spécifier ce qu'il y a à réaliser, puis le réaliser. Voilà d'où viennent la plupart de nos ennuis. On appelle réussis les projets qui sont conformes à leurs spécifications. Mais ces spécifications s'appuient sur l'ignorance dans laquelle étaient les concepteurs avant de démarrer le boulot!"

Les titres de la conférence de 1968 reflètent un certain degré d'incertitude: "Réflexions sur le séquencement de l'écriture d'un logiciel", "Vers une méthodologie de la conception", "Quelques réflexions sur la production de systèmes de grande taille". Certes la plupart des participants utilisent l'expression "Génie Logiciel" comme si elle allait de soi, et des lacunes sont déjà apparentes (on parle notamment assez peu du facteur humain), mais on peut deviner une véritable controverse sur les grandes lignes de ce qui préoccupera cette discipline.

En 1969 les titres des articles publiés ont gagné en assurance. "Critères pour un langage de description de systèmes", "La conception de systèmes très fiables en exploitation continue", etc. Mais c'est surtout en lisant entre les lignes qu'on décèle un changement, et notamment en lisant "The Writing of the NATO reports" de Brian Randell, une sorte de "making of" datant de 1996. Une drôle d'ambiance règne apparemment à la conférence de 1969, mais on ne peut que la deviner dans la description à demi-mot qu'en fait Randell:

> Contrairement à la première conférence, ou il était tout à fait clair que le terme de Génie Logiciel reflétait l'expression d'un besoin plutôt qu'une réalité, à Rome on avait déjà tendance à en parler comme si le sujet existait déjà. Et, pendant la conférence, l'intention cachée des organisateurs se précisa, à savoir: persuader l'OTAN de financer la mise en place d'un Institut International du Génie Logiciel. Cependant les choses ne se passèrent pas comme ils l'avaient prévu. Les sessions qui étaient censées fournir les preuves d'un large et ferme soutien à cette initiative furent en fait dominées par le plus grand scepticisme, au point qu'un des participants, Tom Simpson de chez IBM, écrivit une superbe et courte satire intitulée "Masterpiece Engineering" (Ingénierie du Chef-d'Oeuvre).

Un article qui parlait, par exemple, de mesurer la productivité des peintres en nombre de coups de pinceau par journée. Et Randell d'ajouter que les organisateurs réussirent à le "persuader" d'omettre cet article satirique de Tom Simpson des actes officiels!

L'acte de naissance définitif du Génie Logiciel ayant ainsi été associé à un acte de censure, Randell ajoute qu'il s'interdit pendant la décennie qui suivit d'utiliser le terme, le jugeant injustifié. Il n'acceptera de revenir sur cette décision que pour une conférence marquant en 1979 le dixième anniversaire de Rome, où il profita de l'occasion pour adresser à Barry Boehm, alors la "nouvelle star" de la discipline, une série de piques, que Boehm "ignora soigneusement, je suis navré de le rapporter, à moins qu'il n'ait pas été en mesure de les reconnaitre comme telles".

QUARANTE ANS DE CRISE

Comment expliquer, malgré ces débuts au mieux hésitants, le succès qu'allait connaître par la suite l'idée même de Génie Logiciel?

Il est parfois difficile de faire le tri, s'agissant de ce sujet, entre réalité et mythologie. L'époque des pionniers étant relativement récente, l'essentiel de ce qu'il en reste est constitué de ce que les historiens professionnels appellent la littérature *primaire* - des écrits que nous devons à ceux-là même qui sont été les acteurs de ces événements, qui ont participé aux projets ou assisté aux conférences de cette époque. Le regard que portent des auteurs tel que Frederick Brooks ou Edsger Dijkstra sur le sujet est nécessairement partial: c'est sur leur

recommandation que la discipline se met en place. Plus récemment, des historiens ont commencé à proposer un regard plus distancié sur ces jeunes années du Génie Logiciel.

L'un des facteurs qui contribuera pendant cette première décennie à convaincre une large population de sy'intéresser sera, selon ces historiens, l'utilisation permanente d'une "rhétorique de crise": on se réfère à chaque occasion à la "Crise du Logiciel".

Cette soi-disant Crise se manifestait par divers aspects. Les projets de développement dépassaient les délais et budgets impartis, les logiciels produits étaient de mauvaise qualité et leurs performances insuffisantes, ils ne répondaient pas aux exigences exprimées, ils étaient difficiles à faire évoluer. La situation était catastrophique et il y avait urgence à y mettre bon ordre!

La réalité, telle que la rapportent les historiens, est plus nuancée. D'une part, les seuls à agiter le chiffon rouge de la Crise à l'époque furent sans doutes les mêmes personnes qui militèrent pour la création d'un Institut du Génie Logiciel. Les actes des deux conférences, relèvent les historiens, ne font quasiment pas référence à une "crise" et certains des intervenants cherchent même explicitement à relativiser certains constats parmi les plus alarmistes.

SUR-PLACE

Plus de quarante ans après, les "symptômes" de la "crise" semblent persister, inchangés. D'importantes dérives continuent à frapper des projets informatiques de grande ampleur: ainsi Chorus), le nouveau système d'information destiné à toutes les administrations françaises, défraie la chronique en 2010 par ses dérapages considérables.

Le projet prévu sur quatre ans démarre en Mars 2006, son coût annoncé aux parlementaires à l'origine est déjà important: 600M€. Dès le début 2008 apparaissent les premières dérives et l'on apprend que des retards sont à envisager, que le budget serait dépassé. Fin 2008, une "mise au point" du ministère permet d'apprendre qu'en fait le budget communiqué ne tenait pas compte des coûts de fonctionnement, chiffrés à 100M€ annuels sur cinq ans: la facture serait donc de 1,1Md€. Un petit oubli, en somme. (Un rapport parlementaire de juillet 2010, un peu inquiétant, recommande "d'actualiser l'évaluation du coût complet de Chorus" - ce qui laisse supposer que ce coût réel est encore inconnu à l'heure actuelle...) Les délais s'accumulent, et dès 2009 Chorus qui devait

être déployé entièrement à partir de 2010 se voit repoussé à janvier 2011, et ce malgré une révision à la baisse de ses objectifs fonctionnels.

Il y a pire: Chorus... ne fonctionne pas. Plus exactement son installation perturbe le paiement des factures aux fournisseurs des administrations. Alors que l'Etat gronde le secteur privé sur les délais de paiement, son propre système d'information met en difficulté de très nombreuses PME qui se retrouvent dans l'incapacité d'encaisser leur dû: un comble! Est-ce une difficulté transitoire? C'est en tout cas un transitoire qui dure... et la Cour des Comptes émet fin 2010 des réserves sur l'éventualité que Chorus puisse un jour assumer pleinement la comptabilité de l'Etat français.

Ce n'est là que l'exemple le plus récent et le plus visible, mais chacun au sein de la profession peut rapporter des "histoires de guerre" du même type, concernant des projets parmi les plus petits et les plus simples aussi bien que les plus importants.

Combien de livres, d'articles ou de présentations dans la profession ne commencent-il pas par l'évocation du fameux (et controversé) rapport CHAOS, édité par le Standish Group, et qui est la source quasi systématique des "statistiques" sur le taux d'échec des projets informatiques: vers la fin des années 2000, moins d'un tiers des projets menés par les entreprises interrogées pouvaient selon le rapport être considéré comme réussis; un taux cependant en amélioration par rapport aux années 1990 où un projet sur six seulement entrait dans cette catégorie.

Cependant les chiffres et la méthodologie du Standish Group ont été vivement critiquées. En effet, s'agissant de projets de développement logiciel, ou même plus généralement dans les technologies de l'information en général, la mesure précise de différentes quantités pertinentes qui servent d'indicateurs objectifs (productivité, délais, ROI) est rendue difficile par la complexité même du sujet.

On peut donc difficilement se prononcer sur la *réalité* de la crise elle-même, mais il est évident que le *discours de crise* a joué, et continue de jouer, un rôle important de justification.

MONOPOLE

Malgré ce peu de succès à apporter des solutions à la "crise", la discipline devient une institution et continue à bien se porter pendant les années 70, 80, 90. Il faut dire (et, à regret car ils sont passionnants, sans rentrer dans le détail) que bien des bouleversements caractérisent ces décennies,

qui rendent difficile l'évaluation des progrès réellement accomplis. L'informatique des années 1960 concerne encore surtout des systèmes militaires ou scientifiques; elle va dans un premier temps se diffuser vers les milieux d'affaires, subir de profondes mutations lorsque s'imposent successivement l'ordinateur personnel, les interfaces graphiques, les réseaux d'entreprise et enfin l'Internet. De façon moins visible extérieurement, des innovations telles que la programmation objet ou UML seront tour à tour accueillies comme "LA solution" à tous les maux qui constituent la "crise".

A chacune de ces révolutions, à chaque mode, succède une désillusion, et la prise de conscience qu'une seule innovation ne saurait suffire: selon l'expression consacrée "il n'y a pas de balle d'argent". (Ceux qui se déclarent sceptiques quant à la réalité de la crise sont enclins à dire: "tant pis, de toutes façons il n'y a jamais eu de loup-garou non plus!".)

Voilà le contexte dans lequel, discrètement d'abord avant d'éclater au grand jour au début 2001, se développe un nouveau courant (une nouvelle mode?) qui prendra le nom de "méthodes Agiles" à l'issue d'une rencontre entre quelques auteurs, "gourous" ou consultants qui cherchent, bien que chacun d'une "chapelle" différente, une synthèse entre leurs différentes approches. Cela se passe aux Etats-Unis, à Snowbird, une... station de ski. L'histoire ne se répète pas, dit-on, elle bégaie.

Chapitre 5

La rupture Agile

Citons l'histoire officielle (dans la traduction qu'en propose Fabrice Aimetti):

> Du 11 au 13 Février 2001, dans l'hôtel The Lodge de la station de ski Snowbird située dans les montagnes Wasatch de l'Utah, dix-sept personnes se sont réunies pour parler, skier, se détendre, essayer de trouver un terrain d'entente et bien sûr, manger. Ce qui en est sorti s'appelle le Manifeste du Développement Agile de Logiciels. Y participèrent des représentants de l'Extreme Programming, de SCRUM, de DSDM, de l'Adaptive Software Development, de Crystal, du Feature-Driven Development, du Pragmatic Programming, et d'autres personnes sensibles à la nécessité de trouver une alternative aux pesants processus de développement logiciel pilotés par la documentation.

Le ton est donné: le mouvement Agile se positionne d'emblée en réaction à une situation jugée insatisfaisante, et qui trouve pourtant son inspiration aux racines de la discipline du Génie Logiciel: les "processus de développement" sont excessivement "pilotés par la documentation".

Mais pourquoi cette approche "Agile" devient-elle une alternative, pourquoi à ce moment? Et surtout, comment la caractériser?

THÉORIE DE LA RUPTURE

La théorie de la rupture est due à Clayton Christensen qui la présente en 1997 dans The Innovator's Dilemma. Le sous-titre, éloquent, avertit

du risque que présentent certaines innovations ou "ruptures technologiques": "lorsque de nouvelles technologies entraînent la chute de grandes entreprises".

Christensen distingue deux types d'innovation, l'innovation de continuité et l'innovation de rupture. La première est caractérisée par des produits qui renforcent, dans le contexte concurrentiel, la position dominante des acteurs déjà bien placés sur ce marché: ces produits sont jugés sur des critères de performance déjà reconnus. Imaginez par exemple la dernière génération des appareils photos numériques, qui passent de 8 à 12 megapixels. (Une innovation "révolutionnaire" n'est pas nécessairement une rupture en ce sens, ainsi l'article Wikipedia consacré à la théorie précise que l'automobile fut une révolution mais pas, dans ses effets sur les acteurs de l'économie industrielle, une véritable rupture.)

L'innovation de rupture se caractérise par l'ouverture de nouveaux marchés, et la mise en danger des acteurs dominants dans un marché donné; elle propose quelque chose d'inattendu en décalage avec les critères de valorisation dominants.

L'histoire de la photo numérique à ses débuts est à ce titre éclairante, sa victime emblématique étant Polaroid, l'un de ces noms de marque passés dans l'usage courant, tant son emprise sur un important segment du marché était complète. Pourtant, en 2001 Polaroid se mettait en faillite, pour ne renaître de ses cendres qu'à la suite du rachat de la marque et presque uniquement de cela par des investisseurs cherchant à profiter d'une image encore favorable.

Pourtant, seulement dix ans plus tôt, bien malin qui aurait pu prédire à la photographie le bel avenir qui est devenu notre présent: disparition inéluctable non seulement des modèles argentiques mais également de tout un secteur d'économie qui en dépendait, comme les petites boutiques qui assuraient le développement de nos "péloches".

Ainsi en 1991 le modèle Fotoman, est selon une critique pourtant lucide "incapable de produire des images de qualité professionnelle". En noir et blanc et d'une résolution maximum de 320x240, il coûte la bagatelle de 1000$. Ce n'est en fait qu'un gadget, conseillé aux personnes qui souhaitent s'amuser sur leur PC à triturer et retoucher leurs photos.

Comment un tel produit arrive-t-il, petit à petit, à s'imposer jusqu'à vingt ans plus tard avoir totalement détrôné la génération précédente? Pour avoir un début de réponse, il faut comprendre que les acteurs dominants du marché le sont en vertu de l'excellence de leurs produits selon des critères de performance appréciés par la clientèle: en

l'occurrence, la finesse de l'image, le bon rendu des couleurs, etc. De ce point de vue la photo numérique présente des avantages (facilité de stockage, moins d'usure mécanique, manipulation numérique) mais ceux-ci ne sont pas suffisant pour l'emporter face aux concurrents dominants.

Oui, mais certains marchés n'ont que faire de ces critères de performance majeurs. Parmi les premiers clients auxquels s'adresse le numérique on trouve des fabricants d'avions, qui doivent prendre leurs appareils sous toutes les coutures pendant leur construction pour des raisons règlementaires, et vont donc faire des économies considérables sur le développement chimique, mais ne se soucient guère d'une résolution très fine. Ou bien encore des journalistes qui vont privilégier la transmission rapide d'une information brûlante mais ne sont que peu handicapés, compte tenu des qualités d'impression, par une résolution limitée ou par le noir et blanc.

L'innovation de rupture est donc portée par des produits initialement totalement inadéquats quand on les juge à l'aune des critères de performance établis, mais qui vont conquérir un marché de niche puis se développer. L'innovation plus classique, "de continuité", permet progressivement à ces produits de rattraper tout ou partie de leur retard sur ces critères dominants, et c'est ainsi qu'ils peuvent finir par mettre en danger les leaders du marché.

Alors, en quoi le débat entre "Génie Logiciel" et "Agile" peut-il être considéré comme une instantiation de ce modèle? Certes, il faut faire preuve d'un peu de prudence: considérer l'un et l'autre comme des "produits" en concurrence sur un "marché" relève, sinon de la métaphore, d'une moins d'une interprétation assez libre de ces termes. Je trouve pourtant que le modèle s'applique plutôt bien, et permet d'expliquer non seulement le succès initial de l'approche Agile mais également ses évolutions au cours des dernières années. (Evolutions qui sont souvent mal connues des personnes qui pratiquent ces approches et même de certains experts: toujours ce blocage sur l'histoire...)

L'idée est donc d'analyser le Génie Logiciel comme le produit dominant, et d'expliquer cette domination par le fait qu'il répond aux critères exigés par le marché. L'histoire du projet Chorus nous suggère que "le succès du projet" ne fait pas nécessairement partie de ces critères... alors quels sont-ils? Selon quel critère de performance le discours classique du Génie Logiciel, issu des conférences de 1968 et 1969, est-il, malgré ces échecs, jugé satisfaisant, à tel point qu'on enseigne encore le cycle en V dans les universités?

Pour obtenir une réponse, il faut se repencher sur les origines du Génie Logiciel dans la "crise du logiciel". Celle-ci se manifeste initialement par des difficultés à fournir des prévisions fiables quant aux projets, souvent liées à des allers-retours autour des exigences. Le critère de performance exigé de tout ce qui se présente comme une solution à ladite crise est, en un mot, la possibilité de contrôler tout aléa. On va privilégier la traçabilité, la documentation, et le respect des contraintes que ces outils permettent de suivre le mieux: délais (et donc coûts) maîtrisés, spécifications respectées.

"ÇA NE PEUT PAS MARCHER"

Cette phrase qui constitue le leitmotiv d'un des chapitres du livre de Kent Beck illustre bien en quoi l'approche Agile joue le rôle du produit "gadget", totalement insuffisant, dans l'analogie avec la photo numérique autour des années 1990.

Planifier en permanence, itérativement, ça ne peut pas marcher, on ferait peur au client (pas assez de contrôle). Mettre une nouvelle version en exploitation tous les mois ou deux, ça ne peut pas marcher, on ferait peur aux utilisateurs (pas assez de contrôle). Démarrer le développement avec une vision architecturale résumée en une métaphore, ça ne peut pas marcher, et si on s'était trompés? (Pas assez de contrôle.) Ne pas anticiper sur les besoins futurs lors de la conception, ça ne peut pas marcher, on se retrouvera coincés (pas assez de contrôle). Faire écrire les tests par les programmeurs, ça ne peut pas marcher, tout le monde sait que les programmeurs ne peuvent pas tester leur propre travail et que le test est une profession à part entière (pas assez de spécialisation, donc de contrôle). Travailler en binômes, ça ne peut pas marcher, ça va prendre deux fois plus longtemps, et ils risquent de ne pas s'entendre (pas assez de contrôle, mais aussi une vision du développement comme activité mécanique, réduite à la saisie de code au clavier).

Pour la plupart ces arguments sont légitimes, de même qu'il est légitime de condamner en 1990 la nouvelle technologie numérique comme tout à fait inadéquate.

VALEURS CLASSIQUES CONTRE VALEURS AGILES

L'approche Agile quand à elle est initialement destinée à des projets qui ne se soucient que très peu de contrôler et de tracer. On privilégie la communication directe et orale avec le client, lui permettant d'ajuster sa demande en temps réel; la qualité du produit et sa pertinence pour répondre aux besoins réels (plus qu'à ceux exprimés), la maîtrise des priorités fonctionnelles sous contrainte de délais.

Ne pas chercher à faire concurrence aux "méthodologies" dominantes et aux approches de management orientées sur les production documentaires de ces dernières permet aux "agilistes" comme on ne les appellera que plus tard de s'affranchir de nombreuses contraintes, et d'inventer un mode de développement très souple, basé sur des itérations courtes régies par la règle du "timebox" (on cherche à en faire le maximum en temps imparti plutôt qu'à terminer un objectif fixe quitte à jouer les prolongations), décomposé par incréments fonctionnels. Les phases amont et le travail documentaire sont réduits à l'essentiel, c'est-à-dire à ce qui permet à l'équipe de partager une même compréhension de l'objectif. Le chef de projet perd ses prérogatives: il n'attribue plus les tâches et ne joue plus les intermédiaires, le client étant présent sur place. Des dispositifs techniques (automatisation des tests, intégration continue) limitent au minimum les coûts d'intégration et de refonte.

Une métaphore souvent employée compare l'équipe Agile à des musiciens de jazz qui improvisent, par rapport à leurs collègues dans une formation classique: l'absence de contrôle n'implique pas moins de technicité, et de satisfaction procurée par le résultat final.

Une recherche textuelle dans les actes des conférences de l'OTAN (NATO1968), comparée à la même recherche (grâce à Google Books) dans les actes de la conférence européenne sur XP et les processus agiles (XP2008) permet d'objectiver quelque peu ces différences de valeurs:

- le mot-clé "control" apparaît plus de 100 fois dans NATO1968, 27 fois dans XP2008 (et souvent dans le contexte "groupe contrôle vs groupe expérimental", pour les papiers scientifiques
- le mot-clé "team" apparaît plus de 100 fois dans XP2008, seulement 17 fois dans NATO1968

- le mot-clé "skill" apparaît seulement 5 fois dans NATO1968, 15 fois dans XP2008
- le mot-clé "motivation" apparaît seulement une fois dans NATO1968 ("la motivation de cette conférence..."), 12 fois dans XP2008 et pour la plupart au sens "motivation de l'équipe"

Le critère de performance privilégié par l'approche Agile, ce n'est plus le contrôle, mais bien l'exploitation du talent individuel et collectif. Lorsque les approches Agiles se font connaître sous ce nom, en 2001, ces valeurs sont codifiées sous la forme du fameux Manifeste Agile.

AGILE, DIX ANS APRÈS

Mais depuis 2001, les pratiques ont évolué: le challenger Agile a progressé sur plusieurs critères de performance y compris ceux reconnus par le modèle dominant. C'est exactement ce que prédit la théorie de la rupture, pourtant cette évolution est en grande partie invisible, très peu reconnue par les professionnels qui ont rejoint la communauté récemment et ignorée même d'un certain nombre de vétérans. (Ou bien, elle se traduit pour certains par un inconfort, un sentiment que "Agile c'est devenu n'importe quoi".)

Plusieurs pratiques considérées désormais comme au coeur de Scrum sont ainsi apparues depuis 2001, postérieurement au terme “Agile” lui-même: par exemple les Rétrospectives et le Task Board, deux pratiques très proche du "noyau" mais pourtant d'importation relativement récente. Quant à Extreme Programming son statut de "work in progress" est acquis depuis la deuxième édition du livre de Kent Beck.

Des pratiques sont apparues pour combler par exemple des lacunes dans la prise en compte de l'ergonomie par les approches Agiles, ou encore pour donner des directives plus précises sur la façon d'organiser les "phases amont", le recueil des exigences: on parle de Personas, de Charte Projet, de Story Mapping.

Initialement vécue comme une obligation déraisonnable, la pratique Extreme Programming désignée à l'origine par le conseil "faites écrire vos tests de recette par le client" a beaucoup évolué et donné naissance à des écoles et appellations diverses: ATDD ou pilotage par les tests client, BDD ou Spécifications Exécutables. Bien que ce sujet soit actuellement en pleine fermentation, une tendance se dégage clairement: ces évolutions commencent à répondre de façon satisfaisante aux

exigences de formalisation et de traçabilité qui sont l'apanage des organisations les plus exigeantes en termes de contrôle.

Des pans entiers de pratiques ont enfin servi à occuper de nouveaux terrains. En 2001, Scrum et XP se focalisent sur de petites équipes colocalisées dont la préoccupation presque exclusive est le développement dans un contexte projet: la "date de fin" est une des préoccupations majeures, on conçoit le projet comme un effort unique au-delà duquel l'équipe passe à autre chose.

Ce n'est qu'au fil du temps que la recherche d'adaptation des approches agiles à d'autres contextes conduit à explorer des pratiques alternatives. Ainsi l'approche Lean et Kanban peut-elle trouver un terrain d'application fertile parmi les équipes chargées de maintenance ou de TMA. La communauté Agile voit aussi se développer des mouvements visant à repenser d'autres métiers, à l'image de DevOps qui réunit des admin systèmes sensibles aux évolutions du développement.

AGILE DANS DIX ANS?

En 2010, la photo numérique a entièrement remplacé la technologie précédente (et une nouvelle mutation s'amorce, celle du cinéma, mais c'est une autre histoire). Peut-on imaginer un avenir comparable pour les approches Agiles? Il convient de rester prudent. D'une part, on apprend vite quand on s'intéresse à l'histoire et à la sociologie des sciences et des techniques que le progrès n'est jamais gagné d'avance. Rejeté par toute la communauté médicale de son temps, Semmelweiss ne fut reconnu comme un pionnier de l'hygiène médicale qu'après avoir été poussé à une dépression nerveuse qui lui fut fatale. Il faut du temps, beaucoup de travail et sans doute un peu de chance pour qu'une idée s'impose de manière irréversible; le plus simple est sans doute encore de considérer que tout est réversible.

Il existe une autre raison d'être prudent: le statut de "challenger" ne confère pas automatiquement l'infallibilité de ceux qui se rangent sous sa bannière. Comme le disait Carl Sagan: "They laughed at Columbus, they laughed at Fulton, they laughed at the Wright brothers. But they also laughed at Bozo the Clown." On s'est moqués de Christophe Colomb, de Robert Fulton - l'inventeur du bateau à vapeur - des frères Wright, et de Semmelweiss.

Mais on s'est aussi moqués de tout un tas de gugusses que l'histoire a eu bien raison d'ignorer.

Notamment, certaines lacunes persistantes continuent à plomber la communauté Agile, et risquent de mettre en péril l'avenir du sujet.

Chapitre 6

Quel avenir pour les approches Agiles?

En dix ans, de 2001 à 2011, les approches dites agiles pour le développement de logiciels (Extreme Programming, Scrum) ont changé de statut: initialement considérées comme marginales, elles sont désormais perçues comme l'une des alternatives majeures à l'interprétation traditionnelle d'une discipline, le génie logiciel, qui s'avère incapable de remplir les promesses sans doutes excessives avancées lors de la conférence fondatrice de 1968.

En France notamment, de nombreux cabinets de conseil remportent un succès croissant en répondant à l'appétence d'entreprises de tous profils pour une expertise sur ces stratégies, vues comme mieux capables de répondre au contexte de crise et d'instablité actuel. Une rupture, plus qu'une simple évolution, semble se dessiner.

Pour autant, ces approches, explorées essentiellement sur le terrain par des intervenants de projets de toutes sortes, sont encore mal comprises par le public qu'elles visent, souvent mal expliquées, leurs bénéfices attendus spécifiés de façon trop vague, de sorte que beaucoup d'équipes qui s'y essayent font fausse route dans leur application; avec pour conséquences que le marché qu'elles représentent tarde à se développer, et la promesse d'une amélioration globale dans l'industrie du logiciel semble toujours aussi lointaine.

AGILE: DES VALEURS OU DES PRATIQUES?

Un débat agite régulièrement les listes et forums de diffusions destinées aux spécialistes du sujet: "Qu'appelle-t-on Agile, finalement; s'agit-il d'une philosophie, de valeurs, ou de pratiques concrètes?"

Les agilistes forment ce qu'il convient d'appeler une "communauté", le terme consacré dès lors qu'un nombre important de personnes, communiquant via Internet, s'intéressent à un même sujet. Le terme peut sembler intrinsèquement flatteur: il évoque respect mutuel, solidarité. Pourtant, il n'est pas anodin, car il sous-tend parfois des connotations moins agréables: fermeture sur soi-même, rejet des "autres". En dehors de cette communauté justement, des voix s'élèvent parfois pour poser la question plus brutalement: "Est-ce que ça veut dire quelque chose, Agile-avec-un-grand-A?"

A cette question, et justement pour éviter de trancher d'une façon trop dogmatique, il convient de répondre par une autre question:

"Dans une bonne bouteille, vous buvez l'étiquette?"

Ce qui est important dans l'ensemble certes un peu hétéroclite de notions associées au terme "Agile", ce n'est évidemment pas le mot "Agile", de la même façon que ce qui nous procure du plaisir dans une bonne bouteille de vin, c'est (en principe) son contenu.

En principe? Oui, une des vertus de cette analogie, c'est de nous rappeler que trop souvent c'est bien de l'étiquette qu'on se délecte. On appelle "effet de halo" le biais cognitif qui nous fait trouver plus plaisant un produit lorsqu'on sait qu'il provient d'une "bonne" marque, alors même que si nous faisions l'essai à l'aveugle nous ne saurions pas distinguer une piquette d'un cru à la mode.

Sans jeter la pierre à personne, le fait est que nous courons un risque à trop parler des étiquettes: Agile, Lean, Scrum, XP... On entend trop souvent des choses parfaitement absurdes, par exemple "Lean est le successeur d'Agile".

C'est un contre-sens pour qui connaît le contenu qu'on désigne par le terme Agile: un ensemble de pratiques, certes pas toutes de la même origine, certes pas toutes aussi largement connues et pratiquées, mais dont statistiquement, en interrogeant un assez grand nombre de personnes faisant partie de la communauté depuis longtemps (cette appartenance pourrait se déterminer par un critère concret tel que la participation à des conférences), le recouvrement nous donnerait une idée assez précise.

Citons en vrac, et pour l'exercice: le développement par les tests (TDD), le refactoring, l'automatisation du build, l'intégration continue, la

conception incrémentale, les User Stories, les tests de recette, les critères "Done", les Personas, le Story Mapping, le Planning Poker, les itérations timeboxées, les rétrospectives, le tableau des tâches, le libre choix des tâches, la réunion quotidienne ou "mélée", la programmation en binômes, les demandes d'aide explicites...

Parmi les approches dites Agiles on peut effectivement recenser certaines dont les préconisations sont des sous-ensembles de cette longue liste: Scrum et XP notamment.

Si on se pose la question de savoir de quel mode de pensée sont issues les pratiques Agiles, alors oui, on retombe sur quelque chose qui a de nombreuses affinités avec le discours Lean. Pour autant, le recouvrement en termes de pratiques est relativement faible.

Or, soyons clair là-dessus, ce qui fait le succès ou non d'un projet, ce n'est pas le nom du discours dont on se revendique, l'attachement identitaire des membres de l'équipe: "nous sommes Agiles" ou "nous sommes Lean". **Ce qui fait le succès d'un projet, c'est ce que font les gens!**

De ce point de vue, force est de constater que beaucoup d'équipes se proclament "Agiles" alors même qu'elles peinent à appliquer de façon compétente des pratiques aussi élémentaires que le refactoring, ou qu'elles révèlent lorsqu'on interroge les ingénieurs des contresens sur des notions de base comme la vélocité.

Une partie de ce déficit est à mettre sur le compte de la communauté Agile elle-même. Pendant de longues années, elle ne s'est que peu préoccupée de donner une description systématique, détaillée, cohérente et mise à jour des pratiques désignées par le terme "Agile".

C'est justement ce constat qui a motivé l'ouvrage que vous avez en mains.

LE PROJET DE L'INSTITUT AGILE: UN AVENIR POSSIBLE

A quoi pourrait ressembler un marché des prestations informatiques dans lequel les approches Agiles auraient pleinement trouvé leur place ?

- les entreprises consommatrices qui s'y intéressent trouveraient une information claire, cohérente, neutre et indépendante de tout discours "commercial" sur ce que recouvre réellement le terme "Agile" - ce Référentiel étant un début de réponse, sans doute à compléter

- les entreprises prestataires compétentes dans le domaine Agile bénéficieraient d'une reconnaissance de ces compétences et d'un accès facilité aux contrat les exigeant
- cette reconnaissance proviendrait notamment, au travers de partenariats entre les grandes SSII et les acteurs aux compétences les plus "pointues", du succès reconnu de projets de référence notables par leur échelle ou leur importance
- les bénéfices réels de ces pratiques feraient l'objet d'études empiriques, également réalisées de façon neutre et indépendante, agrégeant des échantillons statistiquement représentatifs
- ces éléments empiriques permettraient d'une part de justifier l'utilisation des approches Agiles, mais aussi d'infléchir leur évolution au fil du temps, vers plus d'efficacité
- les parcours de formation permettant d'obtenir les compétences nécessaires à la mise en oeuvre de projets Agiles seraient identifiés, afin de pouvoir recruter les compétences appropriées pour ces projets
- le milieu de la recherche et de l'enseignement, en lien avec l'industrie, contribuerait pleinement à la consolidation et à l'évolution des connaissances et compétences dans ce domaine
- pour autant, l'ensemble de cet écosystème resterait fidèle à ce qui est à la fois la particularité et la valeur de la dynamique Agile, à savoir d'être avant tout un mouvement "de terrain" initié par des intervenants opérationnels (développeurs, chefs de projet) et qui s'appuie sur leur expérience au quotidien.

LES OBSTACLES À SURMONTER

L'analyse faite plus haut du discours Agile, replacé dans le contexte historique plus global qui est celui du Génie Logiciel, et vu sous l'angle de la théorie de la rupture, éclaire un certain nombre d'obstacles qui peuvent encore empêcher cette situation de se réaliser.

L'obession persistante de la communauté pour le sujet de la **certification** continue de mener bataille sur un terrain, le contrôle des qualifications, où elle représente précisément ce qu'il ne faut pas faire, une concurrence frontale au modèle dominant. Le mouvement Agile doit trouver des solutions à cette question lancinante, mais des solutions qui soient compatibles avec sa propre philosophie. Des modèles ont été proposés, inspirés notamment par le succès des "réseaux sociaux": ils

ont pour nom WeVouchFor (défunt), Entaggle (naissant). La version Web du présent Référentiel des pratiques servira de base à l'Institut Agile pour proposer, sans prétentions, des outils permettant

Il faut répondre enfin à une autre question lancinante, celle du **modèle contractuel**. Nous continuons à dire que le contrat forfaitaire est un frein à l'adoption des pratiques Agiles en France, mais quelle solution proposons-nous?

La prolifération de jargon rend difficile la conversation au-delà du cercle des initiés; la communication sur les pratiques Agile est brouillonne et trop centrée sur les étiquettes: Scrum vs Kanban vs Lean vs XP. Le présent travail est un début de réponse, mais l'Institut Agile veut encourager ceux qui présentent la réalité du terrain, les **éléments concrets et visibles** qui au sein d'une équipe permettent d'identifier **les pratiques Agiles et les bénéfices qu'elles apportent**.

A ce sujet, il est regrettable que l'une des critiques les plus courantes du mouvement Agile soit désormais la suivante: "Depuis 10 ans, il n'y a toujours pas de chiffres probants". La communauté Agile a désormais suffisamment d'ampleur pour être en principe capable de **fournir ces chiffres**.

Certes, il y a dix ans, la demande "montrez-nous vos statistiques" pouvait à raison être rejetée comme un discours de délégitimation, une façon de remettre à leur place ces "hippies du logiciel". Mais nous ne sommes plus en 2001, nous sommes en 2010; les éléments empiriques sont nécessaires, non seulement pour étayer ce que nous avançons, mais aussi pour faire le tri entre celles des pratiques qu'il faut conserver et celles qu'il faut modifier ou éliminer de notre discours. Non seulement nécessaires, mais disponibles, pour peu que nous nous donnions la peine d'aller les chercher.

Dans cette logique, il est critique que la communauté Agile se rapproche de la communauté **de la recherche et de l'enseignement**. Pour l'instant, rares sont les chercheurs qui s'intéressent à l'Agilité, tout simplement parce que les travaux dans ce domaine ne trouveraient pas à être publiés. Ceci pourrait être réparé en entamant une démarche explicite pour faire de ce sujet une discipline soeur, voire concurrente, du Génie Logiciel. Après tout, cette dernière a pu persister pendant 40 ans sans solutionner les difficultés qui ont motivé sa création... il est temps qu'elle subisse la concurrence.

Cette liste n'est sans doute pas exhaustive mais suffirait probablement à tracer une feuille de route pour les approches Agiles au cours des 10 ans à venir.

Chapitre 7

Un capital: les pratiques Agiles

La question "sommes-nous Agiles" n'a que peu d'intérêt. Bien plus importante est la question "qu'avons-nous appris, qui nous permet dans la pratique de mieux réussir nos projets?"

C'est la réponse à cette question qui semble le mieux cerner la contribution du mouvement Agile, un capital progressivement enrichi au cours des dix années écoulées. On peut tenter de le structurer un peu en ajoutant: ce capital se compose de **principes, concepts, pratiques et compétences**.

Principes, concepts, pratiques et compétences : voici résumés, par ordre croissant d'importance, les éléments qui font le contenu de l'approche Agile. Les principes, c'est ce qui nous préoccupe quand on s'arrête pour réfléchir; les concepts, ce sont les définitions auxquelles on revient pour éviter de se tromper; les pratiques, c'est ce qui nous distingue visiblement d'autres équipes; mais le plus important reste les compétences, c'est-à-dire que nous nous attachons à améliorer.

Lors de la réunion où fut écrit le fameux Manifeste, c'est cette question qui animait les participants. Leur objectif était de décrire leurs points de convergence les plus forts possibles: c'est une démarche qui les conduisait nécessairement à une formulation un peu abstraite, un peu éloignée de la réalité quotidienne des projets qu'ils vivaient alors. Ce sont donc les "douze principes" réputés sous-tendre la pratique Agile.

Les principes fournissent un garde-fou utile. Si je m'aperçois que la mise en place de pratiques censément Agiles a eu pour résultat de démoraliser toute l'équipe, le cinquième principe, "bâtissez le projet autour de personnes motivées", m'avertit qu'il y a probablement une

contradiction quelque part. A tout le moins, je dois le prendre comme un sérieux avertissement, envisager que j'ai pu faire fausse route.

Mais on ne peut pas construire un projet sur la seule base de quelques principes. Admettons, nous somme tous d'accord pour "satisfaire le client en livrant tôt et régulièrement des logiciels utiles". Cela ne constitue pas un conseil opérant, ou comme on le dit en anglais avec beaucoup de pertinence, "actionable". C'est comme si on vous conseillait, pour gagner en bourse: "achetez des actions lorsque les prix sont bas et revendez-les lorsque les prix sont hauts". Parfaitement juste et sensé, mais aussi parfaitement inutile: on demande immédiatement *comment*?

Les concepts doivent être maîtrisés sous peine de contresens fatals. Scène vécue: un "Scrum Master" nouvellement certifié débarque sur une liste de diffusion et demande, "est-il raisonnable de fixer pour le prochain Sprint une vélocité de 80% parce qu'un membre de l'équipe est absent?". Des questions de ce type mettent mal à l'aise quant à l'efficacité et à la qualité des formations!

La mise en place d'une approche Agile exige de connaitre un peu de théorie. Pour acquérir un concept théorique il suffit le plus souvent d'en recevoir la définition, par exemple "la vélocité est obtenue en faisant le total des points (estimations) associés aux User Stories qui ont été terminées dans l'itération qui vient de s'achever". Cette définition suffit à identifier les erreurs de notre jeune Scrum Master; la vélocité est quelque chose qui se mesure, non quelque chose qui se décrete à l'avance; c'est une mesure qui s'exprime dans la même unité que les estimations, donc des points ou des hommes-jours mais en aucun cas un pourcentage. Le réflexe qui joue est le même que celui du physicien à qui on annoncerait une vitesse exprimée en mètres carrés: ce n'est pas la peine de vérifier le calcul, il faut d'abord rectifier une incompréhension théorique.

Nuançons quand même le propos: certains des termes utilisés pour décrire Scrum ou XP exigent plus qu'une définition brute, il faut comprendre comment les mettre en relation. La communauté Agile affectione les simulations de projet comme outil pédagogique pour garantir que des termes comme vélocité ou itération sont bien compris.

Les pratiques, c'est ce dont on peut constater la mise en place sur le terrain, de façon visible. Par exemple, une heuristique souvent utile pour juger si une équipe prend au sérieux l'approche Agile: les murs de la salle de travail sont-ils recouverts de tableaux blancs remplis, de grandes feuilles de papier chargés de Post-It, de documents divers relatifs

au projet et mis à jours récemment? La présence de ces indices n'est évidemment pas une garantie de succès, mais elle met en évidence certaines pratiques communes parmi les équipes Agiles: utilisation d'un tableau de tâches par exemple.

Ou bien encore, passer un peu de temps avec un développeur, et constater quelques différences visibles entre son travail et ce qu'on a vu faire ailleurs. Lorsqu'il programme, on le voit systématiquement se préoccuper de tests unitaires automatisés, qui se matérialisent par une "barre verte" ou "barre rouge" dans l'outil qui gère ces tests. Ou bien, on remarque qu'il ne travaille que rarement seul, mais le plus souvent avec un autre développeur assis au même bureau; il ne travaille pas en silence mais explique (sans trop lever la voix pour ne pas perturber d'autres binômes) le raisonnement derrière le code qu'il propose à son collègue.

Les compétences, c'est ce qu'on peut voir quand on examine de plus près certaines pratiques (mais pas toutes), ce sont celles parmi les pratiques qui amènent à distinguer des niveaux de performance. Telle équipe est meilleure que l'équipe voisine dans le domaine de la conception évolutive. Tel développeur est plus doué qu'un autre pour le refactoring, il l'applique de façon plus fluide, à une plus grande échelle, avec plus de rapidité.

Certaines pratiques ne sont pas nécessairement des compétences: d'un certain point de vue une réunion de type "stand-up" ou "daily Scrum", c'est simplement une réunion. On constate, ou non, que l'équipe se réunit une fois par jour à heure fixe pour échanger sur les progrès accomplis. Par contre on peut considérer que l'animation de cette réunion est une compétence. Si cette compétence est présente cela se traduit par une réunion courte et le sentiment (subjectif, certes) qu'elle a été utile. Si cette compétence fait défaut, la réunion peut s'éterniser et sembler contre-productive.

Attention, "animer un stand-up court" ne décrit pas une compétence! On préfèrera quelque chose comme: "telle personne est capable d'équilibrer les temps de parole lors d'une réunion, en encourageant les plus timides à contribuer et les plus extravertis à se limiter". Cette compétence s'appuie sur de l'observation (il faut avoir noté qui a parlé ou pas parlé) et sur de l'autorité (adopter le ton juste lorsqu'on demande à quelqu'un de rendre la parole: ni minauderie ni éclat de voix).

Conclusion: pour pouvoir obtenir les bénéfices d'une approche Agile, il est important d'avoir accès à une description approfondie et détaillée du contenu de cette approche, avec une attention particulière aux pratiques et aux compétences qui, sans doute parce qu'elles sont plus

difficile à décrire finement, ont jusqu'à présent surtout fait l'objet d'une "tradition orale", d'un apprentissage par la transmission individuelle. Même si cette dernière garde son importance dans la culture Agile, elle ne peut que bénéficier d'un corpus écrit, si celui-ci est soigneusement réalisé et entretenu.

PRATIQUES AGILES: PRÉCAUTIONS D'UTILISATION

Attention, terrain miné. Nous venons d'exposer que pour tirer parti des approches Agiles, il était plus efficace de s'intéresser en priorité aux pratiques: à ce que font les équipes et les individus; les principes servant plutôt de règles générales pour vérifier la cohérence de ces actions.

Pour autant, il y a un risque majeur à trop se focaliser sur ces pratiques: celui de tomber dans l'imitation, ce que l'on connait sous le nom de Culte du Cargo, en référence à ces tribus mélanésiennes qui, durant la seconde guerre mondiale, voyaient les militaires japonais et américains construire des pistes d'atterrissage et des tours de contrôle. Activités systématiquement suivies de l'arrivée d'avions chargées de cargaisons de vivres et autres objets sophistiqués. Lorsque la guerre cessa, et faisant le lien logique entre l'activité et son résultat, certains chefs tribaux promirent à leurs ouailles la reprise des livraisons: il suffisait pour cela... de construire des répliques en bambou des tours et des pistes!

On aurait tort de se moquer, tant on retrouve souvent la même attitude au sein d'équipes et d'entreprises dans le domaine du logiciel. "Chez Goopple les équipes utilisent des pratiques Agiles: du TDD, des rétrospectives, des itérations. Et regardez leur valorisation en bourse! Chez nous aussi, faisons du TDD, des rétrospectives et des itérations. On devrait avoir les mêmes résultats." (De même pour l'approche "classique" des projets de développement: les boîtes qui marchent bien font un contrat, puis un cahier des charges, puis une conception technique détaillée, puis implémentent et testent, alors on va faire la même chose. Ainsi se perpétue une "méthode" qui, en réalité, ne fonctionne pas.)

L'utilisation judicieuse des pratiques et des compétences proposées par la communauté Agile exige d'abord de bien connaître les mécanismes par lesquels on produit du logiciel, puis de comprendre en quoi les pratiques qu'on souhaite utiliser modifient ces mécanismes.

Si l'on ne se préoccupe pas du tout de la question sous-jacente, "qu'est-ce que c'est, finalement, que cette activité qui consiste à produire du logiciel", on se retrouve dans une situation analogue à celle des tribus Mélanésiennes, qui voient arriver "magiquement" du cargo mais ignorent tout de la formidable complexité du système industriel qui, à des milliers de kilomètres, est responsable de la production de ces richesses.

Voici donc, en version courte, la notice d'utilisation des pratiques Agiles:

On déploie une pratique dans le but d'en obtenir des bénéfices bien identifiés; l'hypothèse selon laquelle nous obtiendrons ces bénéfices doit être justifiée par un mécanisme supposé (on pourrait aussi dire une modélisation de l'activité de développement) qui nous permet de penser que cette pratique aura les bénéfices attendus. L'utilisation sur la durée de cette pratique ou compétence doit être soumise à une vérification empirique. Si nous n'obtenons pas, dans un délai préalablement établi, les bénéfices attendus d'une façon que nous pouvons un tant soit peu objectiver, alors il nous faudra, d'une part abandonner ou modifier cette pratique, d'autre part remettre en question notre compréhension des mécanismes.

La question "qu'est-ce que c'est que la production de logiciel" est évidemment très vaste, mais on ne peut pas en faire l'économie. Pour être utile, la description des pratiques Agiles doit faire le lien entre la nature de cette activité d'une part, les "lois" et les contraintes qui la régissent, et d'autre part les bénéfices attendus et le raisonnement qui nous laisse penser que ces pratiques apporteront ces bénéfices.

Agilité en kit: les outils de montage

Une autre raison pousse à vouloir donner un coup de projecteur sur les pratiques Agiles, plutôt que sur les étiquettes Scrum, XP, etc. Chacune de ces différentes approches est présentée comme un tout: un ensemble de pratiques qui vont donner un bénéfice maximal si on les met en place ensemble.

Oui, mais vouloir passer d'un seul coup de "pas Agile du tout" à "tout Scrum" ou "tout XP" est, du point de vue humain et managérial, une situation très rare. Elle peut se produire lorsque les équipes sont en crise et sont prêtes à tout pour en sortir; ou encore lorsque des gens se retrouvent pour un nouveau projet qui ont déjà utilisé Scrum ou XP précédemment. Mais la situation encore la plus courante est celle

d'équipes et d'entreprises qui veulent bien mettre un pied dans l'eau, mais pas plonger la tête la première.

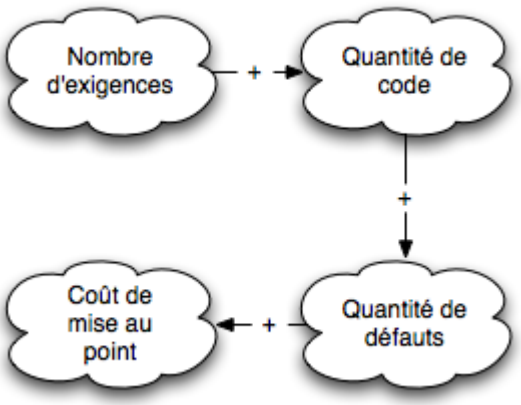
Une première idée est d'appliquer la logique Agile à la transition elle-même: "Adoptez la pratique qui vous semble la plus importante, puis itérez". Ce type de conseil a un intérêt limité, d'une part parce qu'il n'est justement pas évident de déterminer quel outil "agile" est pertinent dans telle ou telle situation, d'autre part parce qu'une seule pratique utilisée en isolation peut avoir des bénéfices marginaux. Enfin, il existe un risque que le discours consistant à dire "cette pratique ne s'applique pas dans mon contexte" serve de prétexte à ne pas se remettre en question.

Pour éviter les écueils du type "Culte du Cargo", nous voudrions une démarche qui permette de déployer, dans un contexte donné, et à diverses étapes d'un projet, les pratiques agiles les plus pertinentes. Avec au départ un "kit" de pratiques, et des outils permettant un assemblage cohérent, on va fabriquer une méthodologie sur mesure.

Le premier de ces outils est la modélisation du processus de développement.

Un exemple. Un des principes Agiles consiste à tester tout au long du projet: on a remis en cause l'approche classique consistant à laisser le test pour la fin. La base de cette critique est une réflexion sur les modèles qui conduisent à prendre cette décision. Dans un modèle trop simplifié, on part du principe que la production de code aboutit à une certaine production de défauts, et qu'il suffit une fois le code produit de procéder à l'élimination de ces défauts:

Figure 1: Diagramme "naïf"



Si on croit cela, on va légitimement remettre les tests à la fin du projet. Le diagramme ci-dessus, dans une notation appelée "diagramme d'effets",

permet de rendre explicite cette supposition, et on peut le comparer avec le diagramme suivant:

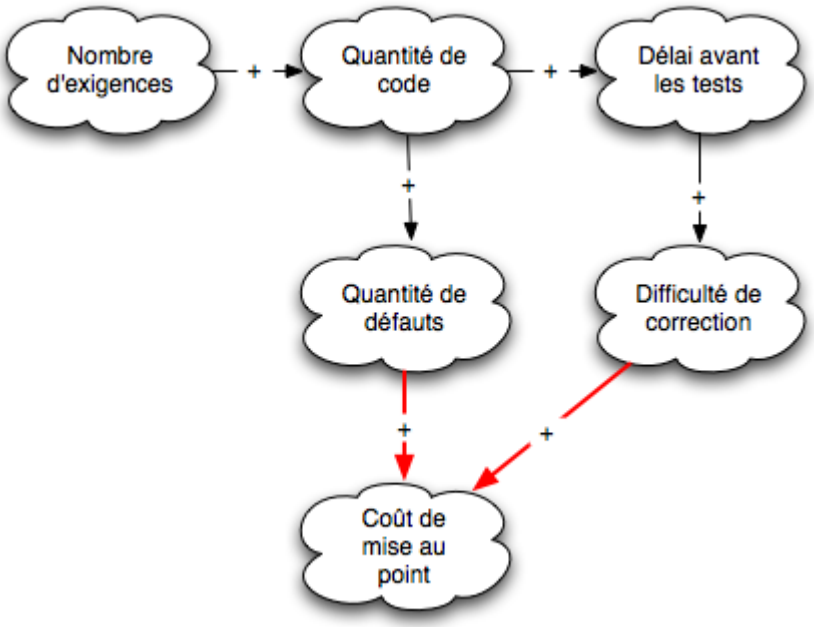


Figure 2: Diagramme "corrigé"

Dans ces diagrammes on s'intéresse aux caractéristiques quantitatives des situations: une bulle est une quantité susceptible, si on le voulait, d'être mesurée. Une flèche est un lien de cause à effet. Lorsqu'elle est ornée d'un "+", l'effet et la cause vont dans le même sens (si l'un augmente, l'autre augmente, et vice-versa); ornée d'un "-", cause et effet vont dans le sens opposé. On doit ce type de diagramme à Peter Senge et Jerry Weinberg.

Je dis "susceptibles d'être mesurée" parce que ces diagrammes donnent surtout des résultats qualitatifs. Il existe des outils permettant de transformer des diagrammes de ce type en simulations numériques. Mais dès que l'on fait intervenir tous les facteurs qui rendent l'analyse plus réaliste, la complexité des modèles rend cette analyse laborieuse, alors qu'on peut obtenir de très bons résultats sur la base d'une analyse quantitative. Ainsi on voit sur le second diagramme que le coût du test est fonction de plusieurs variables, notamment la taille du projet et le délai entre réalisation et test: on rend compte du fait que plus il s'écoule de temps entre l'introduction d'un défaut et sa mise en évidence par le

test, plus il est difficile à corriger. Comme les effets de ces deux variables se renforcent, on peut s'attendre à ce que le coût du test augmente plus que linéairement avec la taille des projets.

Les paramètres qui nous préoccupent sont souvent les mêmes d'un projet à un autre: taille du projet, taille de l'équipe, délai, productivité, qualité, etc. Par contre les influences qui s'exercent sur ces paramètres peuvent être très différentes, selon le type d'entreprises (SSII, éditeur, industriel), le secteur d'activité (finance, scientifique, commerce) et autres caractéristiques du contexte. Et, toujours en fonction du contexte, la priorité accordée à ces différents paramètres peut être très différente. Chez un éditeur on pourra privilégier le respect des délais par rapport à la qualité, dans un contexte industriel l'inverse peut se produire; dans la finance de marchés les performance peuvent être le critère dominant, etc.

Par conséquent, il faut s'attendre à ce que chaque projet soit régi par un modèle différent. L'idéal est de réunir plusieurs personnes concernées par la situation et chercher à explorer, dans une session de type "brainstorm", les liens de causalité entre les différentes variables. Ensuite, on cherche à trouver des interventions: des modifications dans la façon habituelle d'aborder le projet qui, en modifiant le sens d'une relation existante, en supprimant ou créant de nouvelles relations, aient une influence favorable sur les paramètres qui nous préoccupent sans pour autant avoir d'influence néfastes. (C'est souvent là qu'est l'os...)

Souvent, on ne trouve une réponse efficace qu'après avoir obtenu un "déclic" qui permet de réduire le modèle à un nombre moins importants de variables. Les pratiques agiles sont autant d'interventions permettant de modifier la structure des influences mutuelles entre les paramètres du projet. Ainsi la pratique du développement par les tests modifie profondément l'interaction modélisée ci-dessus: pour une bonne partie l'activité de test a lieu avant le développement, la valeur moyenne du paramètre "délai entre l'introduction d'un défaut et le test permettant de le détecter" est réduit de plusieurs ordres de grandeur.

Ces diagrammes d'effets font partie des outils utilisés par les meilleurs consultants étiquetés "Agiles" pour s'assurer de l'efficacité de leurs interventions. Pour être efficace avec une approche Agile il faut non seulement très bien connaître les mécanismes "typiques" qui régissent les projets de développement, mais encore être capable d'analyser finement les variations de ces mécanismes propres à un contexte donné. La simple connaissance des pratiques ne suffit pas: il faut savoir pourquoi et

comment elles fonctionnent, mais aussi quand elles sont susceptibles de ne pas fonctionner.

LE MONDE EST UNE SCÈNE

Les chapitres qui viennent n'aborderont pas de façon détaillée les "rôles", qu'ils soient traditionnels et connus - testeur, développeur, analyste - ou nouveaux et propres à tel ou tel approche Agile: Scrum Master, Product Owner, Coach... Pourquoi? Il convient de rappeler le principe suivant:

Ce qui compte surtout pour le succès d'un projet, c'est ce que les gens font.

Il existe bien sûr des contraintes: les journées de travail ne faisant (en principe) que 8 heures, une même personne ne peut pas tout faire. On doit bien se spécialiser afin de mener certaines activités de façon compétente; le temps que j'investis à devenir un meilleur développeur est autant de temps que je ne passerai pas à acquérir des rudiments de graphisme.

Le principe de Ricardo ou principe de l'avantage comparatif me pousse à m'associer à une personne compétente dans le domaine graphique, et montre que je peux y trouver un avantage même dans le cas où je suis un peu doué en graphisme. (Ce qui est contre-intuitif dans le principe de Ricardo, c'est l'idée que même si je suis strictement meilleur en développement ET en graphisme qu'une autre personne, m'associer avec elle en nous spécialisant chacun dans un domaine peut quand même être intéressant pour chacun des deux.)

Mais ce qui vaut en analyse économique ne vaut pas nécessairement dans un projet. Par exemple les acteurs d'un projet ne travaillent pas chacun dans leur coin: ils ont besoin, chacun, de comprendre ce que font les autres acteurs. Trop de spécialisation peut engendrer le "syndrome du bébé", d'après une conversation entre parents, par exemple au supermarché: "Dis donc, le bébé n'est pas avec toi?" - "Ah non, je croyais qu'il était avec toi..." - "Alors où est-il?" Transposé au monde du projet, cela donne des tâches importantes qui ne sont pas réalisées parce que chacun pense qu'elles relèvent de la responsabilité de quelqu'un d'autre.

Prenons donc l'exemple d'un des nouveaux "rôles" Agiles les plus emblématiques, celui du Scrum Master. Le Scrum Master est chargé:

- de lever les obstacles signalés par l'équipe lors des réunions quotidiennes ("mélées")
- d'être un facilitateur lors de ces réunions
- de rappeler à l'équipe les fondamentaux théoriques de Scrum

- de protéger l'équipe des interruptions

Sur le papier, ce rôle est cohérent. Mais imaginez la situation suivante: dans votre équipe, une personne est très douée pour agir et communiquer vers l'extérieur; une autre est très à l'aise pour animer des réunions. A votre avis, vaut-il mieux:

- donner à l'une de ces personnes les quatre responsabilités ci-dessus, au risque que deux d'entre elles soient moins bien assurées, ou
- jeter aux orties la définition stricte du rôle du Scrum Master, et répartir ces quatre responsabilités entre les deux équipiers en fonction de leurs talents?

La seconde solution a plus de chances d'être efficace.

Un rôle formellement défini n'est qu'une "checklist", le rappel utile d'un certain "lot" d'activités ou responsabilités dont il est important de garantir que chacune est assurée par au moins un membre de l'équipe, afin d'éviter le "syndrome du bébé". La spécialisation a du sens au niveau des responsabilités isolées, pas au niveau de rôles agrégeant plusieurs responsabilités.

Il existe, bien sûr, des corrélations. Un développeur compétent est probablement une personne plus apte à se charger d'un travail de rédaction technique qu'un graphiste compétent. Mais c'est seulement une tendance (peut-être même pas une tendance très marquée). Lorsqu'il s'agit de se répartir des tâches précises, identifiées, au sein d'un projet particulier, entre des personnes réelles, dans toute leur singularité, la notion de rôles doit passer au second plan.

Chapitre 8

Comment utiliser le référentiel

Nous voici donc prêts à aborder le contenu de la démarche Agile, les pratiques elles-mêmes. Comme indiqué plus haut, les chapitres sont destinés à être lus en "accès direct" plutôt que séquentiellement: vous lirez ainsi la description d'une pratique vous intéressant en particulier. Suivant votre situation, vous pourrez vous intéresser à des aspects différents d'une même pratique; ci-dessous, nous abordons le schéma auxquel se conforme chacun des chapitres restants.

CANEVAS DE DESCRIPTION D'UNE COMPÉTENCE

Voici les éléments qui semblent pertinents pour décrire une pratique agile. Nous allons prendre pour exemple une pratique Scrum relativement récente, la Définition de 'fini'.

Parlons d'abord, même si ce n'est pas la première chose qu'on lira (ce sera le nom, abordé plus bas), du **descriptif** succint d'une pratique. Le référentiel n'a pas vocation à se substituer aux livres, articles et contenus de formation qui donnent une définition plus précise et plus détaillée de chaque pratique, compétence et outil abordés. Il s'agit donc ici de se borner à un paragraphe court qui reprend l'essentiel.

Un autre exercice délicat consiste à choisir un nom canonique pour cette pratique. La plupart de nos collègues francophones utilisent un demi-anglicisme, et parlent de "Definition du Done", quand ce n'est pas le terme anglais qu'ils utilisent directement. Quand c'est possible, il est

préférable de vraiment traduire, c'est pourquoi nous préférons "fini" à "done".

Le but n'est pas de faire évoluer le lexique, même s'il est tentant de chercher jouer de son influence... Notre préférence irait peut-être au terme "Liste Sashimi", le terme "sashimi" bénéficie actuellement d'un petit effet de mode, et comme l'illustre le site Web d'un collègue coach, J.B. Rainsberger il est parlant et plaisant. Mais cette appellation pour désigner une pratique spécifique reste tout à fait marginale.

En tout cas, pour toutes ces raisons, il semble important de recenser les **synonymes** connus qui désignent la même pratique ou des pratiques tellement voisines que nous les considérerons comme une seule (par exemple le Sprint de Scrum ne diffère pas assez de l'Itération d'XP pour y voir deux pratiques distinctes).

Une pratique s'accompagne généralement d'un certain nombre d'**erreurs classiques**, de contre-sens et d'abus. En rencensant les plus courantes, on donne de la profondeur à la description brève (mais toujours sans chercher à se substituer à un contenu plus pédagogique), et on encourage à se méfier des imitations et contrefaçons.

Conformément à la division "principes, concepts, pratiques, compétences" on fera ressortir de cette pratique le côté visible: on sait qu'une équipe utilise une pratique quand on peut le constater par des signes observables dans l'espace de travail, par exemple une feuille de paperboard sur laquelle on peut lire la (ou les) définition(s) de "fini".

(C'est là un arbitrage, certaines personnes diront qu'il suffit qu'une équipe connaisse sa définition de "fini" et que tout le monde soit d'accord. Pourquoi pas: dans ce cas, le signe observable consiste à s'asseoir avec plusieurs personnes dans l'équipe et leur demander de réciter cette définition. Il est presque certain que dans ces conditions chacun donnera une définition de "fini" légérement différente; ce qui nous renvoie donc à la rubrique "erreurs classiques".)

Comme indiqué précédemment, une pratique ne vaut que par les **bénéfices** qu'elle confère au projet. Décider, en conscience, d'adopter une pratique, c'est s'engager à vérifier quelques temps après si ces bénéfices ont bien été obtenus, et remettre en question sa vision du fonctionnement du projet si ce n'est pas le cas. L'objectif n'est pas d'utiliser telle et telle pratique pour le plaisir de s'identifier comme "Agile" mais bel et bien d'obtenir ces bénéfices. Notamment, chaque pratique se caractérisera aussi par un coût d'utilisation plus ou moins grand et une efficacité plus ou moins nette: notre objectif, idéalement, est d'obtenir avec le minimum de pratiques, exigeant le minimum d'effort

pour les adopter, le plus grand nombre de bénéfices possibles sur les aspects du projet qui nous intéressent. Peu importe d'où viennent ces pratiques - de Scrum, d'XP, de Lean...

Il est cependant important, pour plusieurs raisons approfondies plus loin, de positionner les pratiques dans un contexte historique, de retracer leurs **origines** et leur évolution. La "définition de 'fini'" ne fait partie de Scrum que depuis quelques années, et son intégration définitive dans les pratiques considérées comme indispensables ne remonte qu'à 2006-2007 environ d'après les différentes sources disponibles: listes de diffusion, blogs, livres sur le sujet. Ou bien il est possible de pratiquer Scrum sans utiliser une définition explicite de "fini" - auquel cas on doit se demander s'il est nécessaire dans un contexte particulier d'utiliser cette pratique - ou bien cet évolution s'explique par le fait que seules les équipes qui l'utilisaient réussissaient dans leur application de Scrum, et qu'on a finalement mis le discours en conformité avec la pratique. La différence entre ces deux scénarios n'est pas anodine.

BIBLIOGRAPHIE SCIENTIFIQUE

Un dernier élément, mais qui a son importance, consiste à relever les travaux scientifiques pertinents. Ceux-ci sont de deux types: théoriques ou conceptuels, qui vont donner un éclairage plus précis et plus rigoureux sur les mécanismes par lesquels une pratique produit ses effets; et, les plus importants, empiriques, qui vont constater dans des conditions contrôlées la réalité et l'importance quantitative de ces prétendus effets.

Cette validation scientifique n'est pas un prérequis à l'utilisation de pratiques qui ont montré leur efficacité. J'aurai l'occasion de revenir sur ce sujet, mais il faut bien constater que le dialogue entre les chercheurs qui s'intéressent à la dynamique des projets agiles d'une part, et les praticiens d'autre part, n'est pas encore d'une grande qualité. On ne s'étonnera pas, par conséquent, que la recherche tarde à s'intéresser à ce que les praticiens trouvent évident depuis longtemps, ou que ces derniers ignorent des résultats que les chercheurs considèrent comme acquis. Mais la convergence au fil du temps entre ces deux communautés me semble indispensable.

Si les pratiques sont réellement utiles et produisent leurs effets de façon fiable, alors on doit être en mesure de le prouver; sinon c'est qu'elles ne sont qu'un placebo, et leur utilisation est nuisible puisqu'elles

mobilisent une partie de l'énergie des intervenants d'un projet, qu'ils pourraient consacrer à d'autres pratiques qui elles sont réellement utiles.

Les travaux les plus importants sont mentionnés dans le chapitre correspondant aux quelques pratiques qui on fait l'objet de telles recherches; une bibliographie reprend l'ensemble de ces références, en annexe.

CANEVAS DE DESCRIPTION D'UNE COMPÉTENCE

Que dire de plus pour éclairer précisément une compétence, par rapport à ce qu'on peut dire concernant une pratique? Notre exemple sera le Refactoring, compétence centrale dans Extreme Programming.

Tout d'abord, une compétence est aussi une pratique: elle se constate à certains signes visibles, elle apporte un bénéfice présumé, qui peut être démontré par des études empiriques. Le canevas de description d'une compétence est donc en règle générale un sur-ensemble de celui qui vaut pour les pratiques.

La principale différence est qu'une compétence nous amènes à distinguer des **niveaux de performance**. Cela n'a pas tellement de sens de dire qu'une personne ou même une équipe est "douée" pour établir sa "definition of Done". Soit elle a une définition, et la trouve satisfaisante; soit elle n'en a pas. Il n'y a pas grand chose à dire à une équipe qui lui permettra d'améliorer sa pratique de la définition. (Certes, une équipe peut avoir une mauvaise définition de "fini", si elle est incomplète ou au contraire trop rigoureuse. Pour autant on ne parlera pas d'améliorer une compétence, mais simplement d'améliorer... le document!)

Au contraire, une personne peut maîtriser plus ou moins bien les différentes techniques du refactoring, et s'améliorer dans ce domaine, soit par la recherche et la pratique personnelle, soit par des mécanismes d'apprentissage en travaillant auprès d'un mentor qui lui enseignera des éléments plus avancés que ceux déjà connus, soit en suivant une formation qui fixera des objectifs pédagogiques adaptés.

N'étant pas lui-même un support pédagogique, le référentiel n'a pas vocation à recenser l'intégralité des savoirs-faires qui composent une compétence donnée, mais simplement de fournir des pointeurs vers les ressources documentaires, lorsqu'elles existent, qui font autorité. Par exemple, le livre de Martin Fowler qui définit plusieurs dizaines de

refactorings élémentaires. La connaissance encyclopédique de tous ces refactorings n'est certes pas une exigence pour pouvoir prétendre maîtriser le sujet, mais d'évidence l'une des choses qui distinguera un expert d'un débutant sera le nombre de ces refactorings qu'il est capable d'utiliser à bon escient.

PLACE AUX PRATIQUES!

Vous voilà maintenant équipés pour tirer le meilleur parti de cet outil. Bonne lecture! N'oubliez pas son pendant sur le Web, constamment actualisé:

http://referentiel.institut-agile.fr/

Chapitre 9

BDD (Behaviour-Driven Development)

Pratique

DE QUOI S'AGIT-IL?

BDD est une élaboration des pratiques TDD (développement par les tests) et ATDD (développement par les tests client).

Au lieu de parler de "tests", une personne utilisant BDD préfèrera le terme "spécifications". Il s'agit en fait de réunir dans un même document des exigences (User Stories) exprimés selon le formalisme rôle-fonction-bénéfice et des scénarios ou exemples exprimés selon le canevas given-when-then, ces deux notations étant les plus lisibles.

En mettant l'accent sur le mot "spécifications", BDD cherche à fournir une réponse unique à ce que nombre d'équipe Agiles considèrent comme deux activités séparées: l'élaboration de tests unitaires et de code "technique" d'une part, l'élaboration de tests fonctionnels (servant à formaliser les exigences) et de "fonctionnalités" d'autre part.

Plutôt que parler de "test unitaire d'une classe", une personne ou une équipe utilisant BDD préfère dire qu'elle fournit les "spécifications du comportement (behaviour) de la classe". Cela se traduit par une plus grande attention portée au rôle documentaire de ces spécifications: leur nom doit être parlant et, complété par leur description structurée par le canevas given-when-then, doit pouvoir servir de documentation technique.

Plutôt que parler de "test fonctionnel" on préfèrera "spécifications du comportement du produit"; par ailleurs le volet technique de BDD est complété par un ensemble de techniques favorisant la conversation avec les interlocuteurs responsables de la définition du produit.

En supplément des techniques de refactoring utilisées dans TDD, l'approche BDD prête, en matière de conception, une attention particulière à la *répartition des responsabilités* ce qui conduit à favoriser la technique dite de "mocking".

En synthèse, BDD consiste à augmenter TDD et ATDD d'un certain nombre de principes supplémentaires:

- appliquer le principe des "cinq pourquoi" à chaque User Story proposée pour en comprendre l'objectif
- raisonner "de l'extérieur vers l'intérieur", c'est-à-dire toujours implémenter le comportement qui contribue le plus directement à cet objectif
- décrire ces comportements dans des notations accessibles à tous: experts fonctionnels, développeurs, testeurs
- appliquer ces techniques jusqu'aux plus bas niveaux de description du logiciel, en étant attentif à la répartition des comportements entre classes

ON L'APPELLE ÉGALEMENT...

On parle également de "behaviour driven design" pour des raisons similaires à celles invoquées dans le cas de TDD.

Le terme n'a pas été francisé.

ERREURS COURANTES

- bien que son créateur, Dan North, explique avoir conçu BDD pour répondre à des difficultés récurrentes lors de l'enseignement de TDD, il faut bien constater que BDD mobilise un plus grand nombre de concepts que TDD, et il semble difficile d'envisager qu'un programmeur novice soit formé *d'abord* à BDD sans s'être préalablement familiarisé avec TDD
- il est parfaitement possible d'appliquer BDD sans outils particuliers, et indifféremment du langage: l'erreur serait d'y

voir un sujet purement technique ou de réduire la pratique à l'utilisation d'un outil

COMMENT RECONNAITRE SON UTILISATION?

- au sein d'une équipe utilisant BDD, une partie significative de la "documentation fonctionnelle" du produit devrait être disponible sous la forme de User Stories agrémentées de scénarios exécutables

QUELS BÉNÉFICES EN ATTENDRE?

Une équipe utilisant déjà TDD ou ATDD peut souhaiter évoluer vers BDD pour les raisons suivantes:

- BDD propose un cadre plus précis pour le dialogue avec les experts fonctionnels
- les notations issues de l'approche BDD (notamment le canevas given-when-then) sont plus proches du langage courant et moins contraignantes comparées à des outils du type Fitnesse
- l'approche BDD permet de générer automatiquement une documentation technique à partir des "spécifications"

ORIGINES

- l'ancêtre de BDD est un outil, agiledox, qui permet de générer automatiquement une documentation technique à partir de tests unitaires JUnit, réalisé par Chris Stevenson en 2003
- visant à éliminer le mot "test" et à le remplacer par "comportement", Dan North réalise l'outil JBehave et le diffuse à partir de mi-2004
- en collaboration avec Chris Matts, North formule le canevas given-when-then pour intégrer l'activité d'analyse à l'approche BDD, qu'il décrit dans un article "Introducing BDD" qui paraît en 2006
- sont apparus depuis de nombreux outils confirmant l'engouement pour l'approche BDD, tels RSpec ou, plus récemment, Cucumber ou GivWenZen

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- "Introducing BDD", de Dan North (2006)
- "Translating TDD to BDD", de Liz Keogh (2009)

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Le peu de publications à ce sujet se concentre sur des questions d'implémentation, par exemple cet article.

Chapitre 10

Backlog

Concept

DE QUOI S'AGIT-IL?

Un "backlog" est une liste de fonctionnalités ou de tâches, jugées *nécessaires* et *suffisantes* pour la réalisation satisfaisante du projet:

- si une tâche contenue dans le backlog ne contribue pas aux objectifs du projet, il faut l'en retirer;
- a contrario, dès qu'on a connaissance d'une tâche qu'on pense nécessaire pour le projet, on doit l'inclure dans le backlog

Ces propriétés s'apprécient relativement à l'état des connaissances de l'équipe à un instant donné: l'élaboration du backlog peut se poursuivre tout au long du projet.

Il est le principal référentiel de l'équipe en matière d'exigences

ERREURS COURANTES

Le backlog n'est pas un cahier des charges.

Dans l'absolu, sa forme n'est pas imposée: on peut le représenter par un document Excel, un fichier texte, une base de données ou encore par un ensemble de Post-It ou fiches cartonnées, ces dernières formes étant les plus courantes parmi les équipes Agiles.

L'important est l'aspect "atomique" des éléments d'un backlog, par opposition à un document narratif dans lequel, par exemple, une seule

phrase peut contenir plusieurs exigences distinctes, ou au contraire décrire sur plusieurs paragraphes une exigence unique.

Tous les éléments du backlog ne sont pas décrits au même niveau de détail à chaque moment du projet: les éléments dont la réalisation est prévue à une date lointaine peuvent être des pans entiers de fonctionnalités décrits en une seule phrase, alors que ceux dont la réalisation est imminente peuvent être très détaillés et accompagnés de nombreux éléments de détails tels que tests, dessins d'interface, etc.

Chapitre 11

Boîte de temps

Concept

DE QUOI S'AGIT-IL?

Une boîte de temps ou "timebox" est une période fixe pendant laquelle on cherche à exécuter le plus efficacement possible une ou plusieurs tâches.

On peut utiliser les "timebox" à différentes échelles de temps. La "technique du pomodoro" s'appuie sur des boîtes de 25 minutes. Dans un autre domaine que le logiciel, le "speed dating" est connu pour ses boîtes de 7 minutes. D'autres domaines peuvent utiliser des durées allant de la journée à plusieurs semaines.

Le plus important est qu'on s'astreint, à la fin de la période, à s'arrêter de travailler pour effectuer un bilan: l'objectif est-il atteint, ou partiellement atteint si on s'est fixé plusieurs tâches? Dans le cas où une seule tâche était prévue on ne tient compte que de deux états possibles de complétion: 0% ou 100%.

ORIGINES

Le "timeboxing" est une variante de la pratique ancienne de la "deadline", mais son utilisation explicite comme stratégie de gestion de projets logiciels remonte sans doute à James Martin, *Rapid Application Development*, en 1991. Il s'inspire des travaux de Scott Shultz chez

l'industriel DuPont, qui met au point une stratégie de prototypage de nouveaux produits en moins de 90 jours.

Chapitre 12

Build automatisé

Pratique

DE QUOI S'AGIT-IL?

Le terme "build" désigne la production, à partir de l'ensemble des fichiers qui sont sous la responsabilité d'une équipe de développement, du produit sous sa forme définitive.

Cela inclut non seulement la compilation des fichiers sources, éventuellement leur regroupement sous la forme de fichiers compressés (Jar, zip, etc.) mais également la production de fichiers d'installation, de mise à jour ou création de bases de données, et ainsi de suite.

On parle de "build automatisé" dès lors que toutes ces étapes peuvent être réalisées de manière totalement répétable et sans intervention humaine, uniquement à partir des fichiers sources présents dans l'outil de gestion des versions.

ERREURS COURANTES

- ne pas confondre build automatisé et intégration continue: l'intégration continue consiste à *déclencher* le plus fréquemment possible le processus de construction (idéalement, automatiquement à chaque publication de code dans l'outil de gestion des versions) et à *vérifier* l'intégrité du résultat produit, notamment par l'exécution de tests automatisés

- en particulier, les outils d'intégration continue (CruiseControl, Hudson, etc.) sont distincts des outils d'automatisation du build (make, Ant, Maven, rake, etc.)
- la prise en charge par un environnement de développement (IDE) de certaines opérations d'assemblage n'est pas suffisante: il doit être possible de réaliser le "build" en dehors de l'IDE
- on conseille en général s'assurer que le "build" dure moins de 10 minutes, tests automatisés compris; au-delà, on compromet la capacité de l'équipe à pratiquer l'intégration continue

ORIGINES

- le principe d'automatiser une mécanique complexe d'assemblage de composants logiciels ne date pas d'hier: l'outil "make" remonte à... 1977
- bien que la pratique ne soit pas nouvelle, ni limitée aux approches Agiles, ce sont ces dernières qui relancent l'intérêt pour une automatisation complète; la vogue des environnements intégrés pendant les années 90 a putôt eu pour effet de marginaliser les outils du type "make", on assiste ensuite à un revirement dans les années 2000

COMMENT RECONNAITRE SON UTILISATION?

Pour s'assurer de la mise en place d'un build automatisé, le plus simple est d'effectuer un test à l'improviste: demander par exemple à une équipe de fournir une version installable de son produit.

Utilisez un chronomètre pour mesurer le temps nécessaire à l'obtention d'une version, puis tentez l'installation sur une machine qui n'a pas été préparée par l'équipe de développement. Toute "surprise" pendant ce processus peut être considérée comme une piste d'amélioration du build automatisé.

QUELS BÉNÉFICES EN ATTENDRE?

L'automatisation du build est un prérequis à l'utilisation efficace de l'intégration continue. Elle apporte cependant des bénéfices propres:

- elle élimine une source de variation et par conséquent de défauts; une procédure manuelle qui contient de nombreuses étapes, toutes nécessaires, offre autant d'opportunités de se tromper
- elle impose de documenter l'ensemble des hypothèses et suppositions qui caractérisent l'environnement cible, ainsi que les dépendances envers des produits externes

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Pragmatic Project Automation, de Mike Clark (2004)

Chapitre 13

Cartes CRC (Classe, Responsabilité, Collaborateurs)

Pratique

DE QUOI S'AGIT-IL?

Les cartes CRC sont un exercice mélant "jeux de rôles" et conception objet.

Afin de décrire rapidement plusieurs manières d'aborder la conception d'une même partie du système, on écrit sur des fiches cartonnées les noms des principales classes concernées, et, afin de cerner la façon dont elles interagissent, les responsabilités de chacune et celles avec lesquelles elles doivent collaborer.

On cherche ensuite à valider le modèle de conception en simulant un scénario d'exécution, chaque développeur participant à la réunion jouant le rôle d'une des classes concernées.

Ainsi un dialogue typique pourrait être: "Bonjour Contrôleur d'Authentification, je suis une Requête Web" - "Très bien, je vais noter vos informations d'identité et l'opération que vous souhaitez effectuer et les transmettre au Gestionnaire d'Accès, s'il vous accepte je vous redirigerai vers l'Afficheur de Page", etc.

ORIGINES

- La technique des cartes CRC est inventée par Ward Cunningham en reprenant, sur des fiches cartonnées, la maquette d'une application Hypercard destinée à documenter la conception d'un système; elle fait l'objet d'un article coécrit avec Kent Beck (1989)
- (Pour l'anecdote, c'est à partir de l'idée originale de cette même application que Ward Cunningham invente en 1995 le concept du Wiki, qui devient quelques années plus tard l'ancêtre de Wikipedia et l'une des innovations les plus marquantes de ce qu'il convient d'appeler le Web 2.0)

Chapitre 14

Carton, Conversation, Confirmation

Concept

DE QUOI S'AGIT-IL?

Formule proposée par Ron Jeffries selon laquelle une User Story est la conjonction de trois éléments:

- le *carton* ou Post-It, qui permet de donner une forme tangible et durable à ce qui n'est sans cela qu'une abstraction, à savoir:
- une *conversation* qui peut avoir lieu en plusieurs temps au fil d'un projet, entre toutes les personnes concernées par un aspect fonctionnel d'un logiciel: clients, utilisateurs, développeurs, testeurs; cette conversation est en grande partie orale mais très souvent complétée par des documents;
- la *confirmation*, enfin, la plus formalisée possible, que l'objectif dont il a été question au cours de cette conversation est atteint.

Chapitre 15

Charte projet

Pratique

DE QUOI S'AGIT-IL?

L'équipe résume, dans un document très court, idéalement susceptible d'être affiché aux murs de l'espace de travail au format d'une feuille de "paperboard", les enjeux principaux du projet, son périmètre et les accords réciproques passés avec les commanditaires du projet.

La communauté Agile s'est appropriée divers techniques ou formalismes intéressants pour leur capacité à condenser ces informations: par exemple le "dessin riche" issu de l'approche SSM, le "diagramme de contexte" hérité de l'approche Yourdon pour l'analyse structurée, ou le A3 du Lean (qui tire précisément son nom du format de papier).

ON L'APPELLE ÉGALEMENT...

Le terme anglais est "project charter".

ERREURS COURANTES

- évitez d'utiliser un "modèle de document" pour une charte de projet; l'intérêt de l'exercice consiste à cerner l'information

qui, dans le contexte unique de votre projet, permettra à l'équipe de prendre des décisions avisées
- une charte de projet ne doit pas dépasser une page en longueur, faute de quoi elle deviendra un nouveau document rébarbatif et rarement lu

QUELS BÉNÉFICES EN ATTENDRE?

On constate souvent, quand on interroge les membres d'une même équipe, des divergences surprenantes entre leurs réponses à des questions du type: "Quel est l'objectif de ce projet? Quels en sont les aspects les plus importants? Qui cherchez-vous à satisfaire? De quels moyens disposez-vous?"

L'établissement d'une charte mais surtout le fait de s'assurer que son contenu soit *connu et approuvé par l'ensemble de l'équipe* donne au projet une unité d'intention qui est un facteur de succès souvent critique.

ORIGINES

- le contenu idéal d'une charte projet est défini dans un article longtemps resté confidentiel, voire obscur: "Charters and Charterings, Immunizing your Project Against Foreseeable Failure" (2001)
- l'adoption de cette pratique par la communauté Agile a été lente et graduelle, et sa popularité plus récente s'explique probablement par la volonté de répondre à la critique souvent formulée qu'une approche Agile porte trop peu d'intérêt à la "vue d'ensemble" du projet

Chapitre 16

Conception au tableau blanc

Pratique

DE QUOI S'AGIT-IL?

Afin de prendre les décisions de conception "juste à temps", les développeurs au sein de l'équipe guettent certains moments critiques où un choix se pose parmi plusieurs alternatives qui peut avoir une influence durable sur la suite du projet.

Lorsqu'un choix de ce type est identifié, une réunion impromptue autour d'un tableau blanc ou d'une session de cartes CRC s'organise, mobilisant une partie de l'équipe (généralement deux ou trois développeurs). Deux facteurs d'efficacité de ce type de session sont à noter:

- on envisage *plusieurs alternatives*, idéalement un minimum de trois; le choix définitif se fait parmi des alternatives acceptables sur la base de critères telles que simplicité ou cohérence avec l'existant;
- on éprouves chacune des alternatives proposées en considérant un scénario concret; par exemple en imaginant comment le test de recette associé à une user story serait traité dans une conception donnée

ON L'APPELLE ÉGALEMENT...

En anglais le terme consacré, provenant de Ron Jeffries dans sa description d'Extreme Programming, est "quick design session".

QUELS BÉNÉFICES EN ATTENDRE?

Dans une approche Agile l'activité de conception est réputée être "lissée" tout au long du projet, plutôt que concentrée dans une phase explicite de conception en amont de l'implémentation.

Cependant, il n'est pas suffisant de supprimer cette phase explicite de conception pour *s'assurer* que les activités de conception, qui restent nécessaires, sont menées de façon satisfaisante.

Les sessions de conception au tableau blanc sont l'une des deux pratiques Agiles qui jouent ce rôle, l'autre étant le refactoring.

Chapitre 17

Conception simple

Compétence

DE QUOI S'AGIT-IL?

L'adhésion à cette pratique implique trois règles de conduite, sur lesquelles l'équipe appuie sa stratégie de conception logicielle:

- la conception est évaluée selon les 4 critères de simplicité énoncés par Kent Beck, ce qui suppose notamment la pratique du refactoring mais également l'adoption de l'heuristique YAGNI (voir définition ci-dessous), fort controversée;
- les éléments de conception tels que patrons de conception ou "design patterns", architecture à base de "plugins", etc. sont conçus comme ayant un coût et pas uniquement une valeur;
- l'équipe cherche à différer aussi longtemps qu'il est possible de le faire de façon responsable les décisions importantes de conception, afin d'obtenir le plus d'information possible sur la pertinence de ces décisions avant d'en payer le coût

Elle s'appuie souvent sur ces pratiques annexes:

- sessions improvisées de conception
- sessions CRC, moins répandues
- remanier vers un pattern
- rétrospectives ou revues de conception

ON L'APPELLE ÉGALEMENT...

- la littérature anglophone désigne souvent cette pratique par l'expression YAGNI, un acronyme qui signifie "You Aren't Gonna Need It", c'est-à-dire "Tu n'en auras pas besoin"; allusion à l'argumentation utilisée par certains programmeurs pour justifier une décision de conception ("Plus tard nous aurons besoin de telle ou telle capacité technique, alors pourquoi pas la réaliser maintenant")
- on emploie également le terme de "Conception Emergente", pour insister sur le fait que la conception n'est pas considérée comme une activité qui a lieu antérieurement à la programmation et impose un cadre à cette dernière; mais qu'au contraire une bonne conception ou une bonne architecture résultent d'une attention portée tout au long du projet aux qualités structurelles du code, et "émergent" donc des interactions entre les détails de bas niveau et les préoccupations d'ensemble

ERREURS COURANTES

- la première erreur, fatale, serait de négliger, par exemple lors du recrutement, l'importance des compétences en conception au sein de l'équipe, au motif que la conception est "émergente" ou "au fil de l'eau": ces termes ne signifient pas qu'elle se fera toute seule!
- il s'agit exclusivement de conception **logicielle** et ce serait un abus d'invoquer ces règles pour argumenter par exemple une décision relevant des exigences du client ou d'un arbitrage d'ergonomie
- la pratique doit être modérée, voire est contre-indiquée, lorsque:
    - le coût du déploiement de nouvelles versions du logiciel est important
    - le projet exige ou doit s'accomoder d'une équipe pléthorique et dispersée

ORIGINES

- l'expression YAGNI est associée à Extreme Programming dès les premiers jours (1998)
- la formulation des critères de simplicité est à peine plus tardive (avant 2000)
- l'application délibérée du remaniement en vue d'obtenir certains patrons de conception fait l'objet d'une première publication par Joshua Kerievsky, "Refactoring to Patterns" (2004)
- les pratiques agiles ayant trait à la conception sont relativement stables dans la période 2000-2010, avec peu d'innovations par rapport à d'autres domaines comme les tests automatisés

COMMENT S'AMÉLIORER?

Sur le plan individuel:

- Débutant
  - je suis capable d'identifier des éléments de conception redondants et de proposer des *simplifications* à du code existant
- Intermédiaire
  - je suis capable de *différer* une décision de conception liée à une exigence future, et de déterminer les critères qui permettront d'arbitrer cette décision
- Avancé
  - je suis capable d'identifier le moment pertinent pour introduire une décision de conception très structurante, par exemple une architecture à base de "plugins"

A titre collectif, une étape majeure est à franchir par toute équipe abordant la Conception Simple: **partager** les décisions de conception. Celles-ci ne sont pas uniquement le fait des architectes ou développeurs senior, elles sont comprises et mise en oeuvre par l'ensemble de l'équipe qui sait se les approprier.

COMMENT RECONNAITRE SON UTILISATION?

- l'équipe dispose d'un "backlog" de tâches spécifiquement liées à la conception:
    - défauts identifiés nécessitant un refactoring explicite
    - opportunitiés de simplification
    - décisions potentielles, différées en attendant plus d'informations
- ce "backlog" ne stagne pas, et ne sert pas de cahier de doléances jamais confrontées; une partie du temps productif de l'équipe est effectivement consacrée à ces évolutions de conception
- l'équipe utilise une ou plusieurs pratiques annexes (sessions improvisées, CRC, revues de conception) offrant une opportunité d'aborder le sujet
- on doit considérer comme un **signal d'alarme**, indiquant que la pratique n'est pas correctement mise en oeuvre, toute impression que des évolutions relativement simples prennent de plus en plus de temps à mesure que le projet progresse

QUELS BÉNÉFICES EN ATTENDRE?

- une réduction du coût total de développement
- une réduction des risques liés à la surconception ("gold plating")

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Is Design Dead?, article de Martin Fowler publié en 2000 et mis à jour en 2004, synthèse du point de vue Agile sur la conception logicielle

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

- Empiriques

- Les études empiriques font état de façon constante d'un taux important d'évolution des exigences au cours d'un projet
- Capers Jones estime qu'en moyenne 35% des exigences (volume calculé en points de fonction) d'un projet sont modifiées au cours de sa durée ("Assessment and Control of Software Risks", 1994)
- Les recherches se sont jusqu'à présent focalisées sur la *prévention* de ces évolutions, il n'existe pour l'instant pas de travaux notables axés sur l'approche Agile consistant à tenir la volatilité des exigences comme un constat auquel s'adapter, non une difficulté à surmonter...

Chapitre 18

Critères de simplicité

Concept

DE QUOI S'AGIT-IL?

Grille d'évaluation proposée par Kent Beck pour juger qu'un code source est "simple":

- le code est doté de tests unitaires et fonctionnels et tous ces tests passent
- le code ne fait apparaître aucune duplication
- le code fait apparaître séparément chaque responsabilité distincte
- le code contient le nombre minimum d'élément (classes, méthodes, lignes) compatible avec les trois premiers critères

Le premier critère est aisé à juger, bien qu'il sous-entende quelque chose de moins trivial, à savoir que le logiciel en question est *correct*, ne présente pas de défauts. Les tests unitaires ne sont au mieux qu'une indication favorable de cet état; le discours Agile tient cependant pour acquis qu'ils sont la solution la plus pragmatique connue à ce jour.

Les second et troisième critères présentent une plus grande part de subjectivité. Ainsi le second peut être interprété littéralement: malheureusement, la "programmation par copier-coller" est une pratique encore répandue dans l'industrie et plombe de nombreux projets, le refactoring est un antidote efficace. Mais les programmeurs plus chevronnés sont capables d'identifier comme "duplication" des

ressemblances plus subtiles entre éléments de code. De même le troisième s'appuie sur les notions non triviales de couplage et cohésion.

Le quatrième critère, sous réserve d'avoir pu se mettre d'accord sur les trois premiers, peut être plus facilement objectivé.

Chapitre 19

Découpage d'une user story

Concept

DE QUOI S'AGIT-IL?

On considère en général que l'estimation associée à une user story doit être suffisamment petite pour qu'il soit envisageable de réaliser cette story en une seule itération.

Lorsque ce n'est pas le cas, il est nécessaire de découper une story en plusieurs, plus petites mais de telle sorte que chacune garde un réel intérêt pour le responsable produit ou l'utilisateur final.

Chapitre 20

Définition de "prêt"

Pratique

DE QUOI S'AGIT-IL?

Par analogie à la "définition de 'fini'", l'équipe explicite et rend visible les critères (généralement une déclinaison de la grille INVEST) faute desquels une fonctionnalité ne saurait faire l'objet d'un travail au cours de l'itération qui commence.

ON L'APPELLE ÉGALEMENT...

Traduction directe de "definition of ready".

QUELS BÉNÉFICES EN ATTENDRE?

- évite de commencer à travailler alors que les critères de satisfaction ne sont pas clairs, ce qui risquerait d'entraîner de coûteux allers-retours avant de se mettre d'accord

ORIGINES

- appellation très récente, formulée par Jeff Sutherland en 2008

Chapitre 21

Définition de 'fini'

Pratique

DE QUOI S'AGIT-IL?

L'équipe affiche de façon visible une liste de critères génériques qui conditionnent le fait de pouvoir considérer un incrément comme "fini". Faute de remplir ces critères en fin de Sprint ou d'itération le travail réalisé n'est pas comptabilisé dans la vélocité.

ON L'APPELLE ÉGALEMENT...

En anglais "done" signifie "terminé, fini". L'anecdote veut que quand on demande à un développeur si quelque chose est "fini" cela n'a qu'un sens technique: il a fini de coder. Si le sens de la question est de savoir s'il a fini de coder, de tester, de mettre à jour les tests et la documentation, il faut le regarder droit dans les yeux en insistant: "est-ce que c'est fini-fini?"

On parle de "Definition of Done" ou "Done List", en français on pourra entendre "Définition de fini" ou "Définition de terminé".

Le terme "Sashimi" plus imagé gagne du terrain pour désigner une "tranche" fonctionnelle à laquelle rien ne saurait manquer.

ERREURS COURANTES

- Accorder trop d'importance à la liste peut être contre-productif: elle doit définir le *minimum* à faire pour qu'on puisse considérer un incrément comme terminé
- Si la liste n'est qu'implicite, au lieu d'être affichée au mur, elle perd beaucoup de son intérêt, ce qui fait sa valeur étant précisément que chacun est au courant de tous les critères
- Constater qu'un incrément n'est pas réellement fini mais se dire "on reviendra dessus plus tard" réduit considérablement l'intérêt de cette pratique.

COMMENT RECONNAITRE SON UTILISATION?

- l'équipe est capable de montrer, sur demande, sa définition de "fini"
- ces critères sont évoqués en fin de Sprint ou d'itération et justifient la décision de comptabiliser un incrément dans la vélocité, ou non

QUELS BÉNÉFICES EN ATTENDRE?

- en amont, fonctionne comme une "checklist" guidant la réflexion des développeurs pendant l'estimation et la réalisation
- en aval, moins de temps perdu en travaux de "réfection" une fois qu'une fonctionnalité a été acceptée
- réduit les risques de brouille entre l'équipe et ses commanditaires en instaurant un contrat clair

ORIGINES

- Proposée par Dan Rawsthorne en 2002-2004
- Répandue sous ce terme à partir de 2007 environ
- Considérée comme un élément majeur de Scrum

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Aucun travail de recherche théorique ou empirique connu ne concerne cette pratique. On peut cependant considérer comme pertinents les travaux de Flores et Winograd, Macomber:

- Understanding Computers and Cognition, Flores F., Winograd T., 1987, présentant le modèle "Conversation for Action"
- Securing Reliable Promises on Projects, Macomber H., 2001

Chapitre 22

Déploiement continu

Pratique

DE QUOI S'AGIT-IL?

Le déploiement continu, prolongement de l'intégration continue, est une pratique visant à réduire le plus possible le temps de cycle, le temps écoulé entre l'écriture d'une nouvelle ligne de code et l'utilisation réelle de ce même code par des utilisateurs finaux.

L'équipe s'appuie sur une infrastructure qui automatise l'ensemble des étapes de déploiement (ou "mise en production"), de sorte qu'après chaque intégration qui se solde par des tests passant avec succès, l'application en production est mise à jour.

QUELS BÉNÉFICES EN ATTENDRE?

Dans le contexte des entreprises qui l'ont mise en place initialement (startups du Web 2.0), le principal bénéfice du déploiement continu est la réduction du temps de cycle, avec deux effets principaux:

- le retour sur investissement pour chaque nouveau développement commence bien plus tôt, ce qui limite les besoins en capitaux
- il est possible d'obtenir très rapidement des retours des utilisateurs sur une nouvelle fonctionnalité, ce qui permet par exemple de tester plusieurs hypothèses et de retenir celle qui représente une amélioration du produit

QUELS COÛTS OU INVESTISSEMENTS FAUT-IL CONSENTIR?

L'infrastructure permettant le déploiement continu tient compte non seulement du test automatisé, mais également (entre autres) de la surveillance du comportement de l'application en production pour détecter rapidement des anomalies qui n'auraient pas été révélée par ces tests, et de la possibilité de retour en arrière très rapide dans le cas d'une telle anomalie. L'investissement nécessaire est donc sensiblement plus important.

ORIGINES

- cette pratique relativement récente (2008) est représentative de deux tendances dans la communauté Agile:
    - adaptation des pratiques Agiles pour les besoins des nouvelles générations de startups, ou Lean Startup
    - extension de l'approche Agile à des métiers connexes au développement, en l'occurrence les métiers de l'exploitation; c'est le mouvement Devops

Chapitre 23

Développement incrémental

Concept

DE QUOI S'AGIT-IL?

Le développement incrémental consiste à réaliser successivement des éléments fonctionnels utilisables, plutôt que des composants techniques.

Un découpage en incréments est dit "vertical", en référence à l'imagerie habituelle qui présente les composants techniques d'une architecture logicielle comme les couches empilées d'un gâteau. (Cet article en est une illustration typique.) Un incrément est une fonctionnalité complète, métaphoriquement une tranche verticale du gâteau.

Une approche incrémentale implique nécessairement d'adopter également, au moins dans une certaine mesure, une approche itérative, mais les deux concepts ne sont pas identiques.

Chapitre 24

Développement itératif

Concept

DE QUOI S'AGIT-IL?

Le développement itératif implique de découper un projet en un certain nombre de cycles, ou itérations, au cours desquelles on prévoit de répéter les *mêmes* activités.

Ainsi, on considère comme itératif un cycle dans lequel on prévoirait, à l'issue d'une phase de spécifications et d'analyse, de répéter 3 fois une itération au cours de laquelle on réaliserait successivement la conception, le développement et le test.

Un tel cycle serait itératif mais ne serait pas "Agile" au sens strict: l'approche Agile privilégie des itérations nombreuses et très courtes, et considère que chaque itération doit regrouper l'ensemble des activités du développement, de la spécification jusqu'au test.

A distinguer du développement incrémental qui consiste à planifier par éléments fonctionnels ayant de l'intérêt séparément.

Chapitre 25

Développement par les tests

Compétence

DE QUOI S'AGIT-IL?

Ce terme désigne une technique de développement qui entremèle la programmation, l'écriture de tests unitaires et l'activité de remaniement. Elle propose les règles suivantes:

- créer **un seul** test unitaire décrivant un aspect du programme
- s'assurer, en l'exécutant, que ce test échoue pour les bonnes raisons
- écrire **juste assez** de code, le plus simple possible, pour que ce test passe
- **remanier** le code autant que nécessaire pour se conformer aux critères de simplicité
- recommencer, en **accumulant** les tests au fur et à mesure

La pratique est indissociable de la famille d'outils de tests xUnit, à qui elle doit son vocabulaire: "barre verte" signifie que l'ensemble des tests unitaires accumulés passent avec succès, "barre rouge" signifie qu'au moins un test est en échec. (L'échec d'un unique test suffit à déclencher l'affichage rouge, avec sa connotation d'alerte: cette tolérance zéro reflète la philosophie de l'outil et de la pratique.) L'expression "dérouler les tests" désigne le fait de lancer ou d'exécuter tous les tests accumulés ("suite" ou "batterie" de tests).

L'une des notions les plus importantes pour un débutant est celle de **granularité** d'un test. Supposons par exemple qu'on écrive un correcteur orthographique. Un test "gros grain" consisterait à vérifier que lorsqu'il est appelé avec un mot mal orthographié, par exemple "importent", le correcteur renvoie la suggestion "important". Un test "grain fin" à l'inverse vise à vérifier la bonne implémentation d'un aspect plus précis de l'algorithme de correction: par exemple qu'une fonction calculant la "distance" entre ces deux mots renvoie bien 1 (une seule lettre a changé).

ON L'APPELLE ÉGALEMENT...

Le nom anglais est "Test Driven Development", l'acronyme TDD étant très souvent utilisé.

On parle également, moins souvent, de développement piloté par les tests.

A l'origine on parle de "Test First Coding", programmation en commençant par les tests, souvent abrégé en "Test First". A mesure qu'une communauté de programmeurs s'empare de cette pratique on cherche à lui donner un nom plus valorisant. La variante "Test Driven" insiste sur le rôle déterminant des tests. Une construction inverse à partir de l'acronyme TDD le réinterprète en "Test Driven Design", ou *conception* par les tests.

ERREURS COURANTES

Quelques erreurs courantes des programmeurs novices en développement par les tests:

- oublier de dérouler les tests fréquemment
- écrire de nombreux tests à la fois
- écrire des tests d'une granularité inadaptée
- écrire des tests non probants, par exemple dépourvus d'assertions
- écrire des tests pour du code trivial, par exemple des accesseurs

Quant à l'organisation de l'équipe autour de cette pratique, les écueils suivants sont courants:

- adoption partielle: seuls certains développeurs plus motivés ou mieux formés utilisent TDD; on ne peut pas attendre de bénéfices collectifs dans ce cas
- mauvais entretien de la batterie de tests: en particulier, une batterie de tests qui prend trop longtemps à dérouler
- délaissement de la batterie de tests: découlant parfois du mauvais entretien, parfois d'autres facteurs tel un trop brusque renouvellement de l'équipe

ORIGINES

L'idée d'une séquence chronologique dans laquelle l'élaboration de tests précède celle des programmes eux-mêmes n'est pas nouvelle; c'est en fait dans la mesure où cette tâche incombe **aux programmeurs eux-mêmes** qu'il existe une rupture. Depuis 1976 date la publication du livre de Glenford Myers *Software Reliability*, et jusqu'à l'apparition d'Extreme Programming, il sera communément admis **"qu'un programmeur ne doit jamais tester son propre code"**.

Cette affirmation présentée comme un axiome fournit une justification à l'établissement du test comme une discipline séparée de la programmation, le "test indépendant". Jusqu'aux années 1990, la tendance se confirme avec la vogue de l'approche "black box testing" et la domination du marché par des outils qui enregistrent et rejouent des séquences de clics (ce qui suppose évidemment que le code soit déjà écrit!).

On peut donc faire remonter l'historique de cette pratique aux premiers outils encourageant les programmeurs à tester leur propre code:

- l'article "A Brief History of Test Frameworks" présente l'histoire apparemment parallèle de deux outils remontant à 1992 environ
- l'événement le plus déterminant est sans conteste la création par Kent Beck de l'outil SUnit pour Smalltalk, dont toute la famille xUnit va s'inspirer (The Smalltalk Report, octobre 1994)
- l'idée d'écrire les tests en premier est décrite comme un des piliers de l'approche Extreme Programming dès 1996
- l'élaboration du "Test First" en "Test Driven" fait partie d'une période d'intense élaboration sur le Wiki C2.com entre 1996 et 2002

- des techniques spécifiques avec leurs propres outils apparaissent pendant cette période, l'une des plus connues est sans doute l'approche "Mock Objects" de Freeman et McKinnon en 2000
- le livre de Kent Beck *Test Driven Development: By Example* achève de codifier la pratique en 2003
- on assiste ensuite à la naissance de plusieurs pratiques inspirées par TDD mais qui s'en écartent suffisamment pour être considérées comme des innovations à part entière: "ATDD" ou "développement piloté par les tests fonctionnels" et "BDD)" sont les plus notables (2006-2010)

COMMENT S'AMÉLIORER?

Niveaux de performance individuels:

- Débutant
    - je suis capable d'écrire un test unitaire avant le code correspondant
    - je suis capable d'écrire le code permettant de faire passer un test
- Intermédiaire
    - je pratique la "correction de défauts pilotée par les tests", lorsqu'un défaut est détecté j'écris le test le mettant en évidence avant de le corriger
    - je suis capable de décomposer une fonctionnalité à coder en un certain nombre de tests à écrire
    - je connais plusieurs "recettes" pour guider l'écriture de mes tests (par exemple: "pour un algorithme récursif, écrire d'abord le test pour le cas terminal")
    - je suis capable d'extraire des éléments réutilisables de mes tests unitaires afin d'obtenir un outil de test adapté à mon projet
- Avancé
    - je suis capable d'élaborer une "feuille de route" pour une fonctionnalité complexe sous forme de tests envisagés, et de la remettre en question si nécessaire
    - je suis capable de piloter par les tests différents "paradigmes" de conception: objet, fonctionnel, par événements...

    - je suis capable de piloter par les tests différents types de domaines techniques: calcul, interface graphique, accès aux données...

COMMENT RECONNAITRE SON UTILISATION?

- la couverture de code est un des moyens courants de constater l'utilisation du développement par les tests; une couverture élevée ne signifie pas nécessairement une bonne utilisation, mais une couverture inférieure à environ 80% est à contrario un signe d'utilisation probablement déficiente
- les historiques de la gestion de versions (logs CVS ou git par exemple) font apparaître des modifications équilibrées: chaque publication de code principal s'accompagne d'une publication de nouveaux tests (hors refactoring)

QUELS BÉNÉFICES EN ATTENDRE?

- de façon informelle, de nombreux retours d'expérience d'équipes utilisant TDD font état d'une réduction très significative du nombre de défauts, en contrepartie d'un surcoût modéré du développement *initial*
- ces mêmes retours suggèrent que ce surcoût est compensé par l'élimination d'une partie importante de l'effort de mise au point en fin de projet
- en imposant l'écriture du test avant celle du code, cette pratique annule le "biais de confirmation" qui est la justification avancée par Myers pour affirmer qu'un développeur "ne doit jamais tester son propre code"
- bien que les travaux scientifiques à ce sujet restent circonspects, de nombreux vétérans de cette pratique y voient un facteur d'amélioration de la conception objet de leur code, et plus généralement de la qualité technique: on entend ainsi que la pratique du TDD a pour résultat du code possédant une meilleure cohésion et un plus faible couplage

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Test Driven Development: By Example, de Kent Beck

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Cette pratique considérée par des auteurs tels que Steve McConnell (Code Complete, 2nd Edition) comme l'une des contributions les plus importantes du mouvement Agile a fait l'objet de plusieurs études.

De même que pour le binômage, les évaluations empiriques sont souvent sont menées sur des populations d'étudiants plutôt que sur des professionnels dans des conditions réelles d'exercice, ce qui limite la portée de leurs conclusions.

- Empiriques
    - "Test Driven Development: Empirical Body of Evidence" recense les principales études connues en 2006, portant majoritairement sur les effets constatés informellement de réduction des défauts (globalement favorable) et d'augmentation de l'effort (globalement neutre)
    - "Realizing quality improvement through test driven development" est une étude plus récente dans un contexte industriel, confirmant ces résultats (2008)
    - "Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study " est l'une des premières à s'intéresser à l'effet de TDD sur les métriques usuelles considérées comme corrélées à la qualité du code, et "s'alarme" d'un effet défavorable du TDD sur ces dernières (2008)

Chapitre 26

Développement par les tests client

Pratique

DE QUOI S'AGIT-IL?

Sur le modèle du développement par les tests, cette pratique ajoute au développement de tests de recette automatisés une contrainte supplémentaire: ceux-ci doivent être réalisés en amont des développements correspondants.

L'idéal recherché (mais atteint relativement rarement) consiste à ce que le responsable du produit, client ou expert fonctionnel soit en mesure de définir de nouvelles fonctionnalités sous la forme de tests de recette, sans que l'intervention des développeurs soit nécessaire.

ON L'APPELLE ÉGALEMENT...

Couramment désignée par l'acronyme ATDD ("acceptance test driven development"), plus rarement STDD ("storytest driven development").

ERREURS COURANTES

Plus encore que la simple utilisation de tests de recette automatisés, cette pratique est liée à des outils tels que Fit/FitNesse, Cucumber, etc.

Il convient par conséquent d'être d'autant plus vigilant à ce que le choix des outils ne se fasse pas au détriment de la facilité, pour les responsables produit, à dialoguer avec les développeurs à propos de la définition des exigences.

ORIGINES

- Kent Beck aborde brièvement cette pratique dans son livre "Test Driven Development: By Example" mais la juge peu praticable (2003)
- c'est à partir de 2003-2004 et sous l'impulsion de Fit/FitNesse que cette pratique fait son chemin malgré les objections soulevées par Beck

QUELS BÉNÉFICES EN ATTENDRE?

- de même que le développement par les tests oblige à concevoir les applications pour être facilement testable au niveau unitaire, le développement par les tests client oblige à créer des interfaces spécifiques pour le test fonctionnel; il est généralement conseillé de ne pas tester directement à travers l'IHM mais de fournir une couche dédiée au test applicatif par exemple

Chapitre 27

Entretien du backlog

Pratique

DE QUOI S'AGIT-IL?

Le responsable client et tout ou partie de l'équipe se réunissent régulièrement pour un "toilettage" du backlog, à l'occasion duquel on peut:

- identifier des user stories qui n'auraient plus de sens
- créer de nouvelles stories si des besoins nouveaux ont été signalés
- réévaluer la priorité des stories entre elles
- attribuer des estimations à des stories qui n'en ont pas encore
- corriger des estimations en fonction de nouvelles informations
- découper des user stories prioritaires mais encore trop lourdes pour être planifiées dans une prochaine itération.

ON L'APPELLE ÉGALEMENT...

En anglais "backlog grooming", littérallement "toilettage" ou "épouillage" du backlog.

QUELS BÉNÉFICES EN ATTENDRE?

L'intérêt de cette réunion est de s'assurer de la pertinence du backlog. Contrairement à un cahier des charges celui-ci est un "document" par nature évolutif: il n'est pas nécessaire par exemple que toutes les user stories aient été finement découpées et estimées en début de projet, par contre il est important *qu'à tout moment* une quantité suffisante de stories soient finement découpées et estimées.

Il arrive fréquemment qu'un backlog "enfle" excessivement au cours d'un projet, se chargeant de fonctionnalités qui ne sont pas vraiment prioritaires mais qu'on a considéré comme potentiellement de bonnes idées à ne pas oublier: en l'absence d'une effort dans le sens inverse, cette inflation peut s'avérer néfaste au respect du calendrier de livraison souhaité.

ORIGINES

On doit apparemment l'expression "backlog grooming" à Mike Cohn et elle remonte au moins à 2005.

Chapitre 28

Equipe

Concept

DE QUOI S'AGIT-IL?

Une "équipe" au sens Agile est un petit groupe de personnes affectées à un seul projet, pour la plupart à temps plein (un membre d'une équipe peut avoir par ailleurs des responsabilités opérationnelles, c'est-à-dire qui ne constituent pas un "projet" à proprement parler).

La notion d'équipe implique le partage des responsabilités: bon ou mauvais, le résultat obtenu est le fait de l'équipe plutôt que de tel ou tel individu.

L'équipe réunit l'ensemble des compétences nécessaires: fonctionnelles et techniques. L'accent est mis sur les résultats à obtenir plus que sur les rôles et responsabilités de chacun: un développeur peut tester, analyser ou découvrir des exigences, même si celles-ci restent soumises à la validation du client; un testeur peut développer si nécessaire, etc.

ON L'APPELLE ÉGALEMENT...

En anglais "Whole Team" ou "One Team".

ERREURS COURANTES

- l'erreur la plus élémentaire est de penser qu'il suffit de réunir un groupe de personnes pour constituer une équipe
- une équipe est constituée d'au minimum 3 personnes (à deux, on forme un binôme), et dépasse rarement une dizaine
- une même personne peut intervenir sur plusieurs *projets* pendant une même période, mais ne fait généralement pas partie de plus d'une *équipe* à la fois

ORIGINES

Kent Beck propose l'expression "Whole Team" fin 2004 pendant la rédaction de la deuxième édition de son livre "Extreme Programming Explained", en remplacement de sa précédente idée du "Client sur site", jugée trop restrictive.

Chapitre 29

Estimation

Concept

DE QUOI S'AGIT-IL?

Une "estimation" au sens usuel en développement logiciel consiste à évaluer l'effort nécessaire à la réalisation d'une tâche de développement.

On cherche ensuite a "agréger" ces estimations individuelles de façon à établir une planification prévisionnelle d'un projet.

ERREURS COURANTES

Il existe de nombreuses écoles de pensée au sein de la communauté Agile concernant les estimations et leur utilisation. Cependant un consensus se dégage sur un certain nombre d'erreurs élémentaires à ne pas commettre:

- une "estimation" n'est pas la même chose qu'un "engagement"; il n'est pas judicieux de reprocher à un programmeur de mettre 3 jours à terminer ce qu'il avait prévu de réaliser en 2 - la notion d'estimation implique une incertitude; confondre "estimation" et "engagement" conduit les personnes concernées à gonfler artificiellement leurs estimations, ce qui est contre-productif
- une "estimation" n'est pas définitive; elle est le reflet l'information dont on disposait au moment de l'émettre: il est par conséquent toujours admissible de réviser une estimation,

à la hausse ou à la baisse, lorsqu'on a acquis de nouvelles informations plus pertinentes

Chapitre 30

Estimation relative

Pratique

DE QUOI S'AGIT-IL?

Une des nombreuses pratiques d'estimation utilisées par des équipes Agiles. Celle-ci préconise d'attribuer des estimations aux tâches ou user stories, non pas considérées isolément, mais en regroupant ensemble des éléments considérés comme de difficulté équivalente.

Une mise en oeuvre possible consiste à tracer sur un tableau blanc plusieurs colonnes, autant qu'on souhaite avoir de valeurs distinctes à l'issue de l'activité. Plus le nombre de colonnes est important, plus le résultat sera précis; mais plus il prendra de temps. Certaines équipes utilisent pour cet exercice les "tailles tee-shirt", de XS à XXL.

Cette technique peut également être utilisée pour ajouter des user stories à un backlog déjà estimé: plutôt que de les estimer isolément, on cherchera à les rapprocher d'une user story déjà connue dont la difficulté serait comparable.

ON L'APPELLE ÉGALEMENT...

On parle également d'estimation par affinité ("affinity estimation") ou d'estimation par similitude d'effort.

ORIGINES

- le nom "Affinity Estimating" semble dû à Lowell Lindstrom qui met au point cette technique lors d'un Scrum Gathering en 2008
- l'idée d'estimation en "tailles tee-shirt" remonte au moins à 2005

QUELS BÉNÉFICES EN ATTENDRE?

- exprimer les estimations de façon relative, surtout lorsqu'on utilise une unité telle que les points, est une façon d'éviter les erreurs courantes liées à l'activité d'estimation: exiger trop de précision, ou confondre estimation et engagement

Chapitre 31

Facilitation

Concept

DE QUOI S'AGIT-IL?

Un facilitateur est une personne désignée pour animer une réunion. Il ou elle s'interdit généralement de participer au contenu de la réunion, mais se cantonne à créer les conditions dans lesquelles un groupe en réunion pourra parvenir à l'objectif qu'il s'est préalablement fixé.

La facilitation est une compétence à part entière, dont les tenants et les aboutissants dépassent largement le cadre des approches Agiles; plutôt que d'en faire ici une description complète, on se réfèrera avec profit aux publications de l'IAF, International Association of Facilitators.

Chapitre 32

Gestion de versions

Concept

DE QUOI S'AGIT-IL?

La gestion de versions n'est pas à proprement parler une "pratique" Agile, puisqu'elle caractérise (heureusement) de très nombreux projets de développement menés selon d'autres principes.

Il est cependant pertinent de la citer ici pour deux raisons:

- bien qu'elles soient devenues rares, certaines équipes restent encore réfractaire à l'utilisation d'outils de gestion de versions, or il s'agit là non seulement d'une "bonne pratique" mais d'un prérequis à nombre de pratiques Agiles (intégration continue par exemple)
- il est judicieux pour une équipe Agile d'expliciter sa politique de gestion de versions, et de s'assurer que celle-ci est adaptée aux pratiques qu'elle utilise

Chapitre 33

Given - When - Then

Concept

DE QUOI S'AGIT-IL?

La matrice Given-When-Then est un format recommandé pour le test fonctionnel d'une User Story:

- (Given) (Etant donné) un contexte,
- (When) (Lorsque) l'utilisateur effectue certaines actions,
- (Then) (Alors) on doit pouvoir constater telles conséquences

Par exemple:

- Etant donné un solde positif de mon compte, et aucun retrait cette semaine,
- Lorsque je retire un montant inférieur à la limite de retrait,
- Alors mon retrait doit se dérouler sans erreur ou avertissement

Des outils tels que JBehave, RSpec ou Cucumber encouragent l'utilisation de cette formule.

Chapitre 34

Graphe burn-down

Pratique

DE QUOI S'AGIT-IL?

L'équipe affiche en grand format, sur un des murs de son local, un graphe représentant la quantité de travail restant à effectuer (sur l'axe vertical) rapportée au temps (sur l'axe horizontal). C'est un "radiateur d'information".

Ce graphe peut concerner l'itération en cours ("iteration burndown") ou plus couramment l'ensemble du projet ("product burndown").

ON L'APPELLE ÉGALEMENT...

Claude Aubry, expert Scrum, a proposé la francisation "beurdone"; elle est restée quelque peu marginale.

QUELS BÉNÉFICES EN ATTENDRE?

En rendant non seulement visible mais indéniable la situation réelle de l'itération ou du projet par rapport au planning initialement prévu, cette pratique oblige l'équipe à confronter les difficultés plus rapidement.

(Corollairement, pour être efficace ce graphe doit être affiché dans un format assez grand et dans un endroit où il suscite des discussions; on évitera de le placer à l'écart dans un couloir, au format A4...)

ORIGINES

- Cette pratique est décrite pour la première fois par Ken Schwaber sur son site "Control Chaos" (2000)
- Elle devient populaire dans la communauté Scrum à partir de 2002
- Diverses alternatives sont ensuite proposées, sans toutefois détrôner le "burn down" comme choix le plus simple, telles que le "burnup" (on inverse l'axe des Y) ou le plus sophistiqué "Cumulative Flow Diagram", à partir de 2003

Chapitre 35

Grille INVEST

Concept

DE QUOI S'AGIT-IL?

La grille des critères INVEST permet de juger de la qualité d'une User Story; elle conduira éventuellement à reformuler son énoncé, voire à modifier en profondeur la Story (ce qui se traduit souvent physiquement: on déchire la fiche ou le Post-It correspondant et on en écrit une autre). Une bonne User Story est:

- **I**ndépendante des autres
- **N**égociable initialement, plutôt qu'un engagement ferme
- **V**erticale, ou ayant de la *v*aleur en soit
- **E**valuée en termes de complexité relative
- **S**uffisamment petite (en anglais *S*mall)
- **T**estable en principe, ce qu'on vérifie en écrivant un test

Plus en détail, une bonne User Story:

- Pour répondre au critère **I**, doit pouvoir être implémentée avant ou après n'importe quelle autre; une erreur classique étant par exemple d'argumenter que "la Story sur la prise de commande implique d'avoir ouvert un compte, donc il faut réaliser en premier celle concernant l'identification (login) de l'acheteur". C'est un peu comme supposer qu'on ne peut écrire le chapitre 2 d'un roman qu'après avoir achevé le chapitre 1: plus facile, mais avec un peu d'imagination on arrive très bien à inverser cette séquence. Dans notre exemple l'équipe de

développement mettra en place les "bouchons" nécessaires pour simuler un utilisateur identifié.

- Pour répondre au critère **N**, ne formuler dans un premier temps que l'essentiel, à savoir l'objectif fonctionnel recherché; on évitera par exemple de spécifier dans une User Story des éléments techniques, par exemple "En tant qu'acheteur, lorsque j'écris dans le champ texte puis que je clique sur le bouton Recherche, la liste à gauche du champ de recherche est renseignée avec les articles correspondants". Ces détails d'implémentation feront l'objet d'une discussion permettant d'identifier la meilleure solution; initialement, une formulation du type "L'acheteur peut chercher des articles par mot-clé" est suffisante pour l'estimation et la planification.
- Pour répondre au critère **V**, représenter un incrément réellement utile pour l'utilisateur final ou du point de vue du client. Par exemple, "réaliser le schéma de la base de données pour la facturation" n'est pas un incrément ayant de la valeur en soi, mais une tâche technique. A contrario, "émettre une facture pour les achats d'articles en France" en laissant pour plus tard une seconde Story dont l'énoncé serait "émettre une facture pour des achats livrés depuis l'étranger" représente un meilleur découpage: chaque incrément permet de réaliser une partie distincte du chiffre d'affaires.
- Pour répondre au critère **E**, être suffisamment comprise, mais également suffisamment précise. Il arrive parfois qu'on formule des User Stories qui représentent presque un projet à part entière, par exemple "Optimiser le calendrier de livraison des achats". Les conditions de satisfaction doivent être suffisamment précises et restreintes pour que l'équipe de développement puisse quantifier l'effort d'implémentation, sinon dans l'absolu du moins en termes de complexité relative. (L'équipe estime par exemple que "Livrer en deux fois lorsque des écarts supérieurs à une semaine séparent les dates de livraison de deux articles du panier" représente deux fois l'effort requis pour "Emettre la facture", cette dernière servant en quelque sorte d'étalon.)
- Pour répondre au critère **S**, ne pas dépasser quelques jours-hommes. La granularité exacte est fonction du nombre de personnes dans l'équipe de développement et de la durée de l'itération, le critère déterminant étant la possibilité de

terminer au minimum une, et idéalement cinq ou six au minimum, User Stories dans une seule itération.

- Pour répondre au critère **T**, être suffisamment bien comprise pour qu'il soit possible de fournir un exemple détaillé: "Lorsque j'achète l'article X au prix Y, sachant que la TVA sur la catégorie Livres est de Z, la facture doit indiquer le montant total suivant:..." La fonctionnalité envisagée doit entraîner de la part du produit des conséquences ou des comportements observables. Ainsi "Améliorer la performance" n'est pas une bonne User Story, il est préférable de préciser: "La page contenant les résultats de recherche doit s'afficher en moins de 2 secondes".

Chapitre 36

Intégration continue

Pratique

DE QUOI S'AGIT-IL?

On appelle "intégration" tout ce qu'il reste à faire à une équipe projet, quand le travail de développement à proprement parler est terminé, pour obtenir un produit exploitable, "prêt à l'emploi".

Par exemple, si deux développeurs ont travaillé en parallèle sur deux composants A et B, et considèrent leur travail comme terminé, vérifier que A et B sont cohérents et corriger d'éventuelles incohérences relève de l'intégration. Ou encore, si le produit final est fourni sous la forme d'un fichier compressé, regroupant un exécutable, des données, une documentation, les étapes consistant à produire ces fichiers et à les rassembler relèvent aussi de l'intégration.

Une équipe qui pratique l'intégration continue vise deux objectifs:

- réduire *à presque zéro* la durée et l'effort nécessaire à *chaque* épisode d'intégration
- pouvoir *à tout moment* fournir un produit exploitable

Dans la pratique, cet objectif exige de faire de l'intégration une procédure **reproductible** et dans la plus large mesure **automatisée**.

Pour la plupart des équipes utilisant actuellement une approche Agile, cela se traduit ainsi:

- l'équipe utilise un référentiel de gestion de versions du code source (CVS, SVN, Git, etc.)

- l'équipe a automatisé entièrement le processus de compilation et génération du produit
- ce processus inclut l'exécution d'une batterie de tests unitaires et fonctionnels automatisés *à chaque publication dans le référentiel des sources*
- en cas d'échec ne serait-ce que de l'un de ces tests, l'équipe cherche prioritairement à rétablir la stabilité du produit

ERREURS COURANTES

- Attention à ne pas confondre les outils (serveurs d'intégration continue du type Cruise Control, Hudson, etc.) avec la pratique. L'intégration continue n'est pas en premier lieu une question d'outil mais d'attitude, et s'appuie sur un outillage diversifié: outils d'automatisation de la compilation et génération, outils de test, et bien sûr outils de gestion des versions.
- L'opposé d'une intégration continue serait une équipe prévoyant un unique épisode d'intégration, à la toute fin du projet. Généralement, cette intégration s'avère alors longue et pénible, ce qui conduit la plupart des développeurs expérimentés à préférer plusieurs épisodes d'intégration au fil du projet, afin d'anticiper les difficultés et d'améliorer la procédure d'intégration. L'intégration continue consiste à pousser ce raisonnement à sa limite. Par conséquent, *toute* activité qui n'apparaît qu'au moment d'une livraison intermédiaire et que l'équipe vit comme longue et pénible est candidate pour être prise en compte au titre de l'intégration continue... même si l'équipe estime déjà utiliser cette pratique.

ORIGINES

- l'expression "intégration continue" est relativement ancienne, on la trouve par exemple dans cet article de 1993, qui ne l'emploie que pour recommander au contraire une intégration assez fréquente mais "programmée" dans le cadre d'un processus incrémental
- la technique du "Daily Build and Smoke Test" est un précurseur, appliquée dans les années 90 par Microsoft pour

Windows NT 3.0 et décrite entre autres par Steve McConnell en 1996
    - on ne parle alors pas d'automatisation, ni des tests, ni du"build", c'est-à-dire les étapes de compilation, édition de liens et production de l'exécutable et des fichiers annexes; l'accent est mis sur la fréquence, le cycle quotidien étant considéré comme "extrême"
    - l'expression "Smoke Test" est héritée de l'électronique: le test le plus basique pour un circuit complexe consiste à le mettre sous tension; si on voit de la fumée s'échapper d'un des composants, c'est qu'il y avait une erreur... il s'agit d'un test rudimentaire pour s'assurer du bon fonctionnement *général* du logiciel, sans préjuger de l'existence de défauts moins évidents
- l'idée d'intégration continue comme un objectif de l'organisation du projet, caractérisant un processus de développement "moderne" et par contraste aux intégrations infréquentes qui ont été la règle jusqu'alors, apparaît dans "Software project management: a unified framework" de Walker Royce, en 1998
- l'idée d'intégration continue, avec les ingrédients suivants, est proposée par Extreme Programming en 1997 et décrite par Martin Fowler en 2000:
    - utilisation d'un référentiel de gestion de versions du code source
    - automatisation du processus de "build"
    - présence de tests unitaires et fonctionnels automatisés (en lieu et place d'un "smoke test" manuel)
    - exécution au minimum quotidienne de l'ensemble (build+test)
    - gestion manuelle du processus d'intégration
- le premier "serveur d'intégration continue", Cruise Control, est publié sous une license Open Source en 2001
    - par rapport à la pratique précédente, il automatise la surveillance du référentiel des versions, le lancement de l'ensemble (build+test), la notification des résultats et la publication de rapports

- la période 2001-2007 est surtout marquée par l'apparition de nombreux outils de ce type, la concurrence entre outils attirant l'attention de façon sans doute disproportionnée sur ces aspects un peu annexes
- un progrès important, cependant, est l'idée de rendre visible l'état de l'intégration la plus récente par un signal visuel, l'idée remonte environ à 2004
    - dans les premiers temps elle a plutôt l'aspect d'un gadget, utilisant une lampe d'ambiance
    - on voit ensuite apparaître de véritable tableaux de bord de l'intégration continue, semblable à des tableaux de bord industriels
- à partir de 2007 on commence à parler de "déploiement continu", tout d'abord comme un idéal théorique; c'est un article d'un ingénieur chez l'éditeur IMVU qui fera sensation et marque l'avènement concret de cette pratique, au début 2009

COMMENT RECONNAITRE SON UTILISATION?

Le signe le plus courant de la pratique au sein d'une équipe est la présence d'un moniteur dédié ou d'un indicateur visuel (lampe, écran LED ou LCD, etc.) dédié à l'affichage du résultat de l'intégration la plus récente.

Plus subtilement, il convient d'observer comment l'équipe réagit à une intégration "cassée" signalant un défaut dans le produit. Dès lors que l'équipe est consciente d'un défaut, mais tolère et laisse persister une situation où elle n'est pas en mesure de livrer un produit exploitable, il n'est plus possible de parler d'intégration continue!

Il est, de ce fait, presque tentant de parler de l'intégration continue comme d'une compétence (d'équipe) plutôt que comme d'une pratique.

QUELS BÉNÉFICES EN ATTENDRE?

- le principal intérêt de l'intégration continue est de réduire la durée, l'effort et la douleur provoquée par *chaque* intégration, l'expérience suggérant qu'il existe un "cercle vicieux" dans le sens inverse: plus les intégrations sont espacées, plus elles sont

difficiles, et plus (en réaction à la douleur provoquée) on a tendance à les espacer
- l'intégration continue démultiplie le bénéfice d'une batterie étendue de tests unitaires: elle permet de détecter au plus tôt les défauts n'apparaissant qu'à l'intégration et par conséquent de minimiser leurs conséquences et les risques associés aux défauts d'intégration
- l'intégration continue permet de tirer le meilleur parti du développement incrémental: des questions comme l'installation et le déploiment du produit ne sont pas laissées de côté jusqu'à la fin du projet mais résolues dès le départ dans le cadre de la pratique

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Continuous Integration, Martin Fowler (2006)
- Continuous Integration: Improving Software Quality And Reducing Risk, de Paul Duvall (2007)

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Sur le plan de l'évaluation empirique, l'une des questions les plus importantes semblerait être la suivante: si l'on compare le coût total, c'est-à-dire le coût unitaire d'une intégration multiplié par le nombre d'intégrations au cours d'un projet, existe-t-il une fréquence optimale d'intégration?

Cette question semble pour l'instant sans réponse, et aucune étude ne semble avoir été directement consacrée à l'évaluation de l'intégration continue.

Chapitre 37

Itération

Concept

DE QUOI S'AGIT-IL?

Une itération au sens Agile est une "boite de temps" ou "timebox" dont la durée:

- varie d'un projet à l'autre, de 1 semaine à 4 semaines, rarement plus
- est en principe fixe sur la durée du projet

Dans la plupart des cas les itérations sont alignées avec les semaines civiles, débutant un lundi et se terminant un vendredi.

La durée fixe des itérations permet d'appliquer une simple formule de proportionnalité pour déduire, compte tenu de la vélocité observée et de l'effort estimé restant à fournir, la durée prévisionnelle du projet.

ON L'APPELLE ÉGALEMENT...

Le terme "Sprint" est également utilisé et vient de Scrum. Les deux termes "Sprint" et "itération" sont utilisés indifféremment sans connotation particulière.

ORIGINES

L'usage courant du terme "itération" suggère bien l'idée de base de répéter plus d'une fois une même démarche, mais son association avec la stratégie consistant à délibérément "tronçonner" la durée d'un projet en sous-périodes semble remonter à Objectory, précurseur d'Unified Process, vers 1993.

Chapitre 38

Langage omniprésent (ubiquitous language)

Pratique

DE QUOI S'AGIT-IL?

Pratique de conception issue du livre de David Evans "Domain Driven Design" (2003), elle consiste notamment à s'attacher à utiliser le vocabulaire et les notions des clients et experts métiers, non seulement dans les discussions autour des exigences, mais également *dans le code source*.

(D'autres techniques complémentaires sont détaillées dans le livre d'Evans, mais le nom de "langage omniprésent" reflète l'intention principale.)

QUELS BÉNÉFICES EN ATTENDRE?

L'un des obstacles couramment rencontré par les projets de développement concerne la difficulté de la *traduction* entre deux langages techniques, celui des développeurs d'une part, celui des experts métier d'autre part.

Pour une part cette traduction est inévitable: les développeurs doivent s'exprimer en termes d'algorithmes, par exemple, qui n'ont pas

nécessairement un équivalent dans le vocabulaire de l'entreprise pour laquelle ils travaillent.

Cependant on constate souvent que ce vocabulaire du logiciel "déborde" du cadre dans lequel il est justifié, au point que les interlocuteurs des développeurs se sentent démunis, voire aliénés, lors des conversations avec ces derniers.

L'adoption explicite et délibérée d'un "langage omniprésent" permet de mitiger ces difficultés et représente donc un facteur de succès d'une approche Agile.

ORIGINES

- dans les premières années, Kent Beck tente de populariser une pratique d'Extreme Programming appelée "Métaphore"; celle-ci est rarement utilisée car mal comprise (2000 environ)
- le terme "langage omniprésent" apparaît avec le livre de David Evans (2003)
- un consensus s'établit ensuite pour ne plus parler de "Métaphore", et remplacer cette pratique par les recommandations, jugées plus précises, de David Evans concernant ce "langage omniprésent" (2004)

Chapitre 39

Livraisons fréquentes

Pratique

DE QUOI S'AGIT-IL?

Une équipe Agile met fréquemment son produit entre les mains d'utilisateurs finaux, aptes à l'évaluer et à formuler des critiques ou des appréciations.

Ce qu'on entend par "fréquemment" varie selon le contexte technique, mais on estime en général qu'une telle livraison doit avoir lieu toutes les quatre à six itérations au grand maximum.

(Dans certains contextes, tels que le développement Web, on peut souvent envisager des fréquences plus élevées, par exemple une livraison par itération; la limite étant le déploiement continu.)

ON L'APPELLE ÉGALEMENT...

Le terme anglais est "small release" ou "frequent releases", on entend parfois le proverbe "release early, release often".

ERREURS COURANTES

- livrer le produit à un responsable marketing ou un chef de projet pour qu'il "teste" la dernière version n'est pas suffisant, pas plus qu'à une équipe d'assurance qualité; au minimum une

livraison doit être une "version beta" évaluée par des utilisateurs représentatifs
- dans certains contextes (logiciels embarqués, par exemple) il n'est pas possible de prévoir une livraison fréquente à *tous* les utilisateurs; cela ne doit pas servir de prétexte à ne pas organiser une livraison fréquente à *des* utilisateurs (sites pilotes, clients volontaires, etc.)

QUELS BÉNÉFICES EN ATTENDRE?

Savoir organiser une mise en production fréquente *dès les débuts du projet* est l'une des pierres angulaires de la réduction du risque par l'approche Agile:

- on diminue l'effet tunnel pour la planification du projet
- on s'assure plus rapidement de l'adéquation du produit aux besoins réels
- on obtient un retour plus rapide sur la qualité et la stabilité du produit

Chapitre 40

Niko-niko

Pratique

DE QUOI S'AGIT-IL?

L'équipe affiche, sur un mur visible de tous, un calendrier de type journalier. A la fin de chaque journée de travail les membres de l'équipe indiquent par un dessin ou en collant des gommettes de couleur leur évaluation subjective de cette journée.

ON L'APPELLE ÉGALEMENT...

Le terme "feeling board" est également utilisé. C'est un radiateur d'information.

Le terme est d'origine japonaise, le redoublement indiquant un effet d'onomatopée, "niko" signifie "sourire".

ORIGINES

- proposé par Akinori Sakata dans cet article début 2006

QUELS BÉNÉFICES EN ATTENDRE?

L'intérêt de la pratique est d'objectiver un élément, la motivation ou le bien-être de l'équipe, habituellement subjectif et difficile à mesurer.

C'est une excellente illustration du "Principe de Gilb": "Si vous avez besoin de quantifier quelque chose, il existe toujours une manière de le faire qui soit préférable à ne pas le quantifier du tout." En d'autres termes, il n'est pas nécessaire qu'une mesure soit parfaite ou très précise, dès lors que votre objectif est de rendre quantifiable quelque chose qui ne l'est pas encore: l'essentiel est de commencer.

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Niko-niko, d'Emmanuel Chenu (2009)

Chapitre 41

Objets fantaisie (Mock Objects)

Concept

DE QUOI S'AGIT-IL?

Technique couramment utilisée dans la mise au point de tests unitaires automatisés.

Il s'agit de créer pour les besoins d'un test une version spécifique d'un composant logiciel (typiquement une classe, qui sera par conséquent instanciée par un objet), qui au lieu de reproduire le fonctionnement réel de ce composant en fournit une version "pré-enregistrée": par exemple, le "mock" d'une base de données est un objet qui simule la base de données, en renvoyant des données prédéterminées quelle que soit la requêtre qu'on lui transmet.

ON L'APPELLE ÉGALEMENT...

On parle également de "simulacres".

Le terme "stub" (bouchon) n'est pas à proprement parler un synonyme, mais fait partie du même vocabulaire créé pour parler avec plus de précision de certaines techniques liées aux tests unitaires.

Le terme générique pour l'ensemble des objets de ce type est "doublure" (en anglais "test double" par analogie à "stunt double").

ORIGINES

- le terme est dû à Steve Freeman, Tim McKinnon et Philip Craig, c'est une allusion à un passage *d'Alice au Pays des Merveilles* de Lewis Carroll sur la Tortue Fantaisie (Mock Turtle), leur article "Endo-Testing: Unit Testing with Mock Objects" est publié en 2000

QUELS BÉNÉFICES EN ATTENDRE?

Le principal intérêt de cette technique est qu'elle permet de *découpler* des parties du code de façon à les tester de façon réellement "unitaire", c'est-à-dire sans que le résultat du test soit dépendant d'un composant qui n'a rien à voir avec celui que l'on teste, mais dont ce dernier dépend pour son fonctionnement.

ERREURS COURANTES

Cette technique un peu controversée a de nombreux adeptes mais également des détracteurs, qui estiment que l'utilisation de ces objets fantaisie ou "mocks" complique, souvent inutilement, le code des tests unitaires, au détriment de leur rôle de documentation technique.

Chapitre 42

Personas

Pratique

DE QUOI S'AGIT-IL?

Lorsque le projet l'exige (par exemple si l'ergonomie joue un rôle déterminant dans le succès du projet), l'équipe rédige la fiche biographique détaillée d'un utilisateur fictif de son produit: c'est ce qu'on appelle une "persona".

Le plus souvent cela prend la forme d'une page A4 avec la photo de ce personnage (découpée par exemple dans un magazine), son nom et des détails de sa situation sociale, personnelle et professionnelle: "Amanda Jones, 34 ans, attachée de presse d'un grand groupe alimentaire, etc."

Un produit logiciel étant le plus souvant destiné à plusieurs catégories de personnes, avec potentiellement des préférences différentes qui peuvent les amener à avoir des attentes différentes quant au produit, on réalise autant de "personas" qu'on a identifié de catégories importantes.

Ces fiches biographiques sont affichées dans le local de l'équipe.

QUELS BÉNÉFICES EN ATTENDRE?

Le principe des Personas se résume à l'observation suivante: un concepteur qui cherche à faire plaisir à tous les utilisateurs concevable est contraint à tellement de compromis que sa création ne convient finalement à personne.

Par conséquent, il est préférable de décrire de façon précise et personnalisée un utilisateur néammoins représentatif, et justifier ses choix (de fonctionnalité, de conception graphique, d'ergonomie) en fonction de cet utilisateur.

Partager ces informations au sein de l'équipe permet également de responsabiliser chacun sur les conséquences de ses choix. Un développeur soucieux d'optimiser son produit pour "Jeanine, retraitée" évitera de lui-même de proposer un dialogue de préférences contenant 50 réglages; ce ne serait pas nécessairement le cas s'il avait en tête la notion plus abstraite de "l'utilisateur".

ERREURS COURANTES

Ne pas confondre les Personas avec différents autres outils utilisés pour la gestion des exigences ou dans le marketing:

- ce ne sont pas des "rôles d'utilisateurs" (par exemple administrateur, agent de saisie, etc.) caractérisés principalement par leur fonction professionnelle
- ce ne sont pas non plus des "segments de marché" (l'exemple canonique étant la ménagère de moins de cinquante ans) définis par des critères principalement démographiques

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Deux études empiriques notables concernent cette pratique:

- "An Empirical Study Demonstrating How Different Design Constraints, Project Organization and Contexts Limited the Utility of Personas" de Ronkko et al. (2005) est une étude de cas qui s'appuie sur trois projets pour montrer que les enjeux politiques présents au sein de toute entreprise peuvent limiter voire annuler l'efficacité de cette pratique
- "Real or Imaginary: The effectiveness of using personas in product design" de Long (2009) présente une étude expérimentale, utilisant des étudiants, qui tend à confirmer l'intérêt de la pratique

Chapitre 43

Planning poker

Pratique

DE QUOI S'AGIT-IL?

Une méthode ludique d'estimation, utilisées par de nombreuses équipes Agiles.

L'équipe se réunit avec son client ou Product Owner. Autour de la table, chacun dispose d'un jeu de cartes représentant des valeurs typiques pour l'estimation en points d'une user story.

Le client présente rapidement l'objectif d'une story. Chacun choisit ensuite une estimation, en silence, et prépare la carte correspondante face cachée. Lorsque tout le monde est prêt, on retourne les cartes simultanément et on donne lecture des estimations.

Les membres de l'équipe ayant donné les estimations la plus basse et la plus haute sont invités à expliquer leur raisonnement; après une brève discussion, on cherche à émettre une estimation faisant consensus, éventuellement en répétant le jeu.

ERREURS COURANTES

Un danger potentiel du Planning Poker réside dans l'obligation de "converger" vers une estimation produisant un consensus. En procédant de la sorte, on risque d'ignorer une information importante, à savoir *le degré d'incertitude* que représentait une divergence importante entre les estimations initiales.

ORIGINES

- le Planning Poker est une adaptation du "Wideband Delphi" proposé par Barry Boehm dans les années 1970
- sa forme actuelle est proposée par James Grenning dans un article paru en 2002
- rendue populaire notamment dans la communauté Scrum, et comme nombre d'autres techniques liées à la planification, par le livre de Mike Cohn *Agile Estimating and Planning paru en 2005

QUELS BÉNÉFICES EN ATTENDRE?

- le format de la réunion permet d'exploiter la connaissance de tous les membres de l'équipe, alors que dans une réunion "à bâtons rompus" il arrive souvent que certaines personnes ne s'expriment jamais
- la discussion qui a lieu suite à la révélation des estimations initiales permet de mettre en commun des connaissances sur la user story en question et les difficultés potentielles de son estimation

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Plusieurs études initiales menées par Nils Christian Haugen semblent confirmer l'intérêt de la pratique, qui produirait des estimations légèrement plus fiables que l'estimation par un seul "expert".

Chapitre 44

Points (estimation en)

Pratique

DE QUOI S'AGIT-IL?

La plupart des équipes Agiles préfèrent émettre des estimations dans une unité autre que les classiques hommes-jours.

L'une des principales raisons en est que l'usage de la vélocité pour la planification permet de s'affranchir de toute unité: quelle que soit l'unité utilisée, on peut calculer la durée prévisible du projet en termes du nombre d'itérations, chaque itération produisant une certaine quantité de fonctionnalités.

Une raison plus socio-psychologique tient au fait que l'estimation dans une unité arbitraire tels que les "points" fait peser sur les développeurs une moindre pression.

ERREURS COURANTES

La principale erreur consiste à investir trop de temps ou d'énergie dans le débat sur l'unité d'estimation, dans la mesure où la planification basée sur la vélocité rend ce choix largement sans conséquence.

ON L'APPELLE ÉGALEMENT...

L'expression anglophone consacrée est "story points".

Plusieurs synonymes existent exprimant avec plus ou moins de fantaisie l'intention de se démarquer des estimations "réalistes" en hommes-jours: l'un des plus courants est "NUTS", acronyme anglophone pour "Nebulous Units Of Time".

On peut également croiser des "cahouettes" (francisation de "nuts"), des "nounours" (francisation de "gummi bears", c'est à dire des bonbons en forme d'ourson), etc.

Chapitre 45

Programmation en binômes

Compétence

DE QUOI S'AGIT-IL?

Deux programmeurs partagent un seul poste de travail (écran, clavier, souris), se répartissant les rôles entre un "conducteur" (aux commandes) et un "copilote" (surveillant l'écran et intervenant en cas de besoin), intervertissant les rôles fréquemment.

ON L'APPELLE ÉGALEMENT...

De l'anglais "pair programming", ou programmation (ou développement) par paires, on parle aussi de binômage.

Donne les verbes: binômer, "pairer" (fréquent au Québec).

ERREURS COURANTES

- les deux programmeurs doivent être actifs et impliqués tout au long de la session, sinon, aucun bénéfice ne peut en être attendu
- un calcul simpliste assimile le binômage à un doublement des coûts; les études mettent en évidence qu'il n'en est rien,

lorsqu'il est pratiqué correctement, mais dans le cas limite où un seul programmeur travaille réellement, c'est bien le risque encouru
- on doit entendre parler le conducteur; binômer c'est aussi "programmer à haute voix", s'il est silencieux le copilote doit intervenir
- imposer le binômage ne peut en aucun cas s'avérer fructueux dans une situation où les aspects relationnels, y compris les plus concrets (hygiène personnelle, etc.) présentent un obstacle

ORIGINES

- longue tradition en milieux universitaires, militaires...
- suggérée par Larry Constantine (Constantine on Peopleware) sous le nom "Dynamic Duo" (1995)
- simultanément décrite dans les "Organizational Patterns" de Jim Coplien (1995)
- pratique Extreme Programming dès ses débuts (1996)
- la variante "ping pong" (cf. ci-dessous) est décrite en 2003

COMMENT S'AMÉLIORER?

L'un des principaux écueils à la pratique du binômage est la passivité. Lorsqu'on l'utilise conjointement avec le développement par les tests, une variante appelée "ping pong programming" favorise l'échange de rôles: l'un des deux programmeurs écrit un test unitaire qui échoue, puis passe le clavier à son collègue qui cherche alors à faire passer ce test, et peut alors à son tour écrire un test. Cette variante peut être utilisée soit pour des raisons pédagogiques, soit dans un esprit plus ludique par des programmeurs déjà confirmés.

- Débutant
    - je suis capable de participer en tant que copilote et intervenir à bon escient
    - je suis capable de participer en tant que conducteur, en particulier d'expliquer à tout moment le code que j'écris
- Intermédiaire

        - je reconnais le bon moment pour abandonner le clavier et changer de rôle
        - je reconnais le bon moment pour "voler" le clavier et changer de rôle
- Avancé
    - je suis capable de m'insérer dans le rôle de copilote en cours de développement d'une tâche commencée par un autre

COMMENT RECONNAITRE SON UTILISATION?

- l'agencement du mobilier dans la pièce et des postes de travail sont adaptés au binômage (on peut reconnaître une équipe novice à tout aménagement qui rend le binômage inconfortable, par exemple trop peu de place pour deux sièges)
- l'ambiance sonore est maîtrisée: les conversations de N binômes simultanées crééent un murmure perceptible mais qui ne perturbe pas le travail
- si vous pouvez voir des programmeurs portant un casque audio en entrant dans la pièce, c'est un signe *négatif* suggérant non seulement l'absence de binômage mais également de conditions défavorables à son adoption

QUELS BÉNÉFICES EN ATTENDRE?

- une plus grande qualité des développements; "programmer à haute voix" conduit à mieux appréhender les complexités et les détails pernicieux, limitant ainsi le risque d'erreurs ou de se fourvoyer
- une meilleure diffusion des connaissances dans l'équipe, notamment lorsqu'un développeur peu familier d'un module travaille avec un autre qui le connait mieux
- une amélioration plus rapide des compétences des développeurs juniors, au contact de leurs aînés
- une rédution de l'effort de coordination, puisqu'au lieu de N développeurs on est amené à coordonner N/2 binômes

- une meilleure capacité à rester concentrer et résister aux interruptions: lorsqu'un des deux membres du binôme doit s'interrompre, l'autre peut rester focalisé sur la tâche et aider son collègue à se reconcentrer ensuite

QUELS COÛTS OU INVESTISSEMENTS FAUT-IL CONSENTIR?

Sans que ce soit confirmé de façon probante par des études empiriques, on estime que dans le cas idéal, la programmation en binômes se traduit par un surcoût de 15% par rapport à la situation où chaque développeur travaille séparément, surcoût qui serait facilement compensé par l'amélioration de la qualité, puisqu'une mauvaise qualité se traduit par une pénalité de maintenance tout au long du projet.

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Pair Programming Illuminated, de Laurie Williams

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

- Théoriques
    - Les travaux théoriques les plus intéressants sont ceux qui poursuivent l'approche ethnographique initiée entre autres par Sallyann Freudenberg (née Bryant), en examinant "à la loupe" des interactions entre programmeurs dans leur milieu habituel
    - How Pair Programming Really Works cite certains de ces travaux assez représentatifs de la tendance récente à remettre en question le modèle "conducteur/copilote"
- Empiriques
    - The Collaborative Software Process, thèse doctorale de Laurie Williams, la plus connue des études à ce sujet, rapportant les conclusions initiales suivantes:

amélioration de la qualité, pas de surcoût constaté de façon statistiquement significative
- The effectiveness of pair programming: A meta-analysis, une méta-analyse recensant et regroupant les 18 principales études empiriques connues à ce jour, conclusions: le binômage améliore la qualité et permet de réduire les délais de développement, mais conduit à une augmentation du coût (i.e. en termes de charge); la réduction du délai concerne surtout les tâches les plus simples confiées à des juniors et peut s'accompagner d'une réduction de la qualité.
- La plupart des études empiriques (14 sur les 18 analysées dans la référence ci-dessus) souffrent d'un défaut qui entâche leurs conclusions; elles sont menées sur des populations d'étudiants plutôt que sur des professionnels dans des conditions réelles d'exercice.

Chapitre 46

Radiateurs d'information

Pratique

DE QUOI S'AGIT-IL?

"Radiateur d'information" est le terme générique désignant un affichage mural (passif, par exemple une feuille de papier, ou actif, par exemple un écran affichant des résultats d'intégration continue) réalisé par une équipe pour diffuser publiquement une information jugée pertinente: nombre total de tests, vélocité, nombre d'incidents en exploitation, etc.

ON L'APPELLE ÉGALEMENT...

- un terme apparenté est "Big Visible Chart" (un grand graphe bien visible).
- pour généraliser on utilise également le terme "informative workspace"
- on peut enfin rapprocher ces termes de la notion de "Visual management" de la pensée dite Lean

QUELS BÉNÉFICES EN ATTENDRE?

L'usage significatif de radiateurs d'information véhicule deux messages importants:

- l'équipe n'a rien à cacher (notamment à ses clients)
- l'équipe n'a rien à *se* cacher: elle admet et confronte ses difficultés

Le principal intérêt de cette pratique est donc de responsabiliser les différents intervenants. Un avantage annexe consiste à susciter des conversations lorsque des personnes extérieures à l'équipe visitent les locaux.

ORIGINES

- le terme "information radiator" est dû à Alistair Cockburn, entre 2000 et 2002
- le terme "big visible chart" remonte à la même époque, il est en tout cas antérieur à 2001

Chapitre 47

Refactoring

Compétence

DE QUOI S'AGIT-IL?

Refactorer consiste à modifier un code source de façon à en améliorer la structure, sans que cela modifie son comportement fonctionnel.

ON L'APPELLE ÉGALEMENT...

L'anglicisme est le plus souvent employé: l'activité est le refactoring, on utiliser également le verbe refactorer. Par le substantif, un refactoring, on désigne une modification générique qu'il est possible d'appliquer à un code pour le transformer.

On peut également traduire par remanier, remaniement. Le terme de refactorisation est également rencontré, mais nettement moins courant.

ERREURS COURANTES

Attention, refactorer n'est pas:

- réécrire du code
- corriger des bugs
- améliorer un aspect extérieurement visible du logiciel, comme l'IHM

ORIGINES

- Connue sous le nom de "factoring" chez les programmeurs Forth depuis 1984
- Formellement décrite par Bill Opdyke en 1992
- Intégrée par Kent Beck dans Extreme Programming en 1997
- Popularisée par Martin Fowler en 1999

COMMENT S'AMÉLIORER?

Niveaux de performance individuels:

- Débutant
    - je connais la définition
    - j'utilise quelques refactorings de mon environnement de développement
    - j'utilise quelques refactorings que je sais appliquer manuellement
    - je connais les risques de régression associés au refactorings manuels et automatiques
    - je reconnais la duplication et sais l'éliminer par refactoring
- Intermédiaire
    - je reconnais et j'élimine une gamme plus étendue de "mauvaises odeurs" dans le code
    - je peux enchaîner plusieurs refactorings pour mener à bien une intention de conception, en maîtrisant leurs dépendances (méthode dite "Mikado")
    - j'applique le refactoring en continu, en ayant rarement besoin d'une longue session de refactoring
- Avancé
    - j'ai un sens aigu de la duplication et des différentes formes de couplage
    - je maîtrise des refactorings concernant d'autres éléments que le code: schémas de bases de données, de documents...
    - je suis en mesure de faire évoluer mon code vers des structures de mon choix issues de différentes origines: paradigme objet, paradigme fonctionnel, "patterns" connus

COMMENT RECONNAITRE SON UTILISATION?

- les historiques de la gestion de versions (logs CVS ou git par exemple) contiennent des entrées libellées "Refactoring"
- les modifications correspondantes sont réellement isofonctionnelles (pas d'effet sur les tests unitaires, pas de code ajouté)

QUELS BÉNÉFICES EN ATTENDRE?

- les aspects objectifs de la qualité du code (longueur, duplication, couplage, cohésion, complexité cyclomatique) s'améliorent au fil du temps
- en lien avec la [propriété collective], la connaissance des décisions de conception est mieux partagée dans l'équipe
- des schémas de conception, ou des modules génériques, émergent et peuvent être réutilisés par la suite

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Refactoring, de Martin Fowler
- Refactoring (SourceMaking)
- Le Refactoring, Régis Medina
- La méthode Mikado

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Les travaux empiriques concernant la pratique du remaniement sont équivoques, et peinent à mettre en évidence les bénéfices prétendus par les équipes de développement, qui y voient pourtant une pratique indispensable. C'est un sujet de recherche prioritaire compte tenu de l'importance accordée à cette pratique au sein des mouvements Agile, Extreme Programming et Software Craftsmanship.

- Théoriques

        - Refactoring Object-Oriented Frameworks, thèse de doctorat de Bill Opdyke
- Empiriques
    - An Empirical Evaluation of Refactoring, étude préliminaire, bilan: pas d'effets constatés
    - Refactoring--Does It Improve Software Quality?, étude préliminaire, bilan: effets néfastes selon certaines métriques sur des projets open source

Chapitre 48

Responsabilité collective du code

Pratique

DE QUOI S'AGIT-IL?

Une équipe établit généralement des conventions, tacites, orales ou implciites, précisant quel développeur peut intervenir sur quelle partie du code source dont l'équipe a la charge. Différents modes existent: ainsi un module ou un fichier peut être la responsabilité exclusive d'un développeur, un autre membre de l'équipe ne pouvant le modifier qu'avec sa permission.

Lorsque cette responsabilité est collective, *tous* les développeurs sont autorisés et encouragés à modifier *toute partie* du code lorsque c'est nécessaire, soit pour réaliser une tâche en cours, soit pour corriger un défaut ou améliorer la structure de l'ensemble.

ON L'APPELLE ÉGALEMENT...

Le terme anglais est "collective code ownership".

ERREURS COURANTES

Il est fréquent que des règles tacites existent qui sont en contradiction avec les règles explicite de l'équipe: par exemple, une équipe peut avoir ostensiblement adopté un modèle de responsabilité collective, mais un des développeurs manifeste des mouvements d'humeur lorsqu'on modifie certains fichiers, conduisant l'équipe à les considérer comme *de facto* sous sa responsabilité exclusive. Il est important de vérifier qu'une responsabilité collective revendiquée se traduit effectivement dans les faits.

QUELS BÉNÉFICES EN ATTENDRE?

Un modèle de responsabilité collective

- diminue les risques de blocage en cas d'absence d'un des développeurs
- favorise la transmission des connaissances techniques
- rend chacun des développeurs responsable de la qualité de l'ensemble
- favorise une conception du code dictée par des décisions techniques et non par la structure sociale de l'équipe

QUELS COÛTS OU INVESTISSEMENTS FAUT-IL CONSENTIR?

Si la notion de responsabilité collective est bien acceptée dans le discours Agile car conforme à la philosophie générale de responsabilité partagée, elle a ses détracteurs qui y voient plusieurs risques, auxquels il convient d'être attentif:

- rendre chacun responsable de la qualité peut amener à ce que personne ne s'en préoccupe ou à ce que chacun se renvoie la balle
- la possiblité d'intervenir sur toutes les parties du code peut être défavorable à la définition d'interfaces claires et explicites entre ces différentes parties, alors que ces interfaces sont garantes d'une bonne conception

ORIGINES

Dès ses débuts, Extreme Programming fait une pratique à part entière de ce qui n'est à l'origine qu'un à priori favorable, issu des pratiques des développeurs Smalltalk - pour qui la notion de "fichier" est nettement moins structurante, compte tenu des outils de développement utilisés.

Chapitre 49

Rythme soutenable

Pratique

DE QUOI S'AGIT-IL?

L'équipe vise un rythme de travail tel qu'il pourrait être soutenu indéfiniment.

Ceci implique en général de refuser ce qui est parfois considéré comme un "mal nécessaire": heures supplémentaires, travail le week-end.

ON L'APPELLE ÉGALEMENT...

Le terme anglais originel était "40 hour week" soit "semaine de 40 heures". On lui a préféré "sustainable pace", terme plus général.

QUELS BÉNÉFICES EN ATTENDRE?

L'approche Agile considère que le recours autre qu'exceptionnel à des mesures telles que les heures supplémentaires est rarement productif, et contribue en fait à *masquer* des défauts de planification, de management ou de qualité interne. Il est préférable de mettre au jour ces défauts et d'en traiter la cause profonde plutôt que de leur appliquer un traitement symptômatique.

Chapitre 50

Rétrospective jalon

Pratique

DE QUOI S'AGIT-IL?

A mi-chemin de la durée prévue du projet, ou bien en fin de projet (et surtout lorsque l'équipe prévoit d'être ensuite à nouveau réunie pour un nouveau projet), l'ensemble des membres permanents de l'équipe (au sens large, et pas seulement les développeurs) consacre une voire deux journées entières à un bilan approfondi.

Plus encore que lors d'une rétrospective d'itération, cette réunion doit être facilitée, suivre un format qui peut varier selon les objectifs mais qui sera décidé à l'avance.

ON L'APPELLE ÉGALEMENT...

On parle en anglais de "project retrospective" ou "interim retrospective" pour la distinguer de la rétrospective d'itération.

ERREURS COURANTES

- Il est souvent conseillé de faire appel à un facilitateur extérieur plutôt que de confier ce rôle à un membre de l'équipe: on considère que le facilitateur ne peut pas jouer ce rôle s'il prend également part aux discussions.

- Les enjeux d'une rétrospective jalon sont différents et par certains aspects plus importants que ceux d'une rétrospective d'itération; l'accent est mis sur des questions stratégiques, sur le long terme, sur la santé relationnelle de l'équipe, alors qu'une rétrospective d'itération s'intéresse souvent à des questions concrètes et tactiques.

ORIGINES

- le terme "Rétrospectives" est dû au livre de Norm Kerth (cf. ci-dessous) en 2001
- la pratique de la rétrospective jalon reste marginale au sein des équipes agiles, qui privilégient le plus souvent les rétrospectives d'itération, populaires à partir de 2006

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Project Retrospectives, de Norm Kerth (2001)

Chapitre 51

Rétrospectives d'itération

Pratique

DE QUOI S'AGIT-IL?

L'équipe se réunit périodiquement, le plus souvent à un rythme calé sur celui des itérations, pour réfléchir explicitement sur les événements saillants depuis la précédente réunion de ce type, et décider collectivement des actions d'amélioration ou de remédiation suggérées par la discussion.

Le plus souvent il s'agit d'une réunion facilitée et utilisant un format prédéterminé, l'intention étant de s'assurer que tous les membres de l'équipe ont l'occasion de s'exprimer.

ON L'APPELLE ÉGALEMENT...

Le terme "retrospective" est emprunté directement à l'anglais, on le préfère à "debriefing" ou "post-mortem" pour ses connotations plus neutres. On parle également de "heartbeat retrospectives" pour insister sur le fait que leur régularité contribue à donner au projet une cadence (le sens littéral est "battement de coeur").

Le terme "reflection workshop" (atelier de réflexion) utilisé par Alistair Cockburn est moins utilisé mais connu de quelques experts.

ERREURS COURANTES

- Une rétrospective a pour objet de mettre à jour des éléments soit factuels, soit d'ordre relationnel, qui ont des effets tangibles sur le travail de l'équipe, et d'en tirer des pistes d'amélioration; elle ne doit pas se transformer en discussion "de comptoir" ou en défouloir.
- A contrario, une rétrospective ne peut fonctionner que si chacun se sent libre de s'exprimer, la responsabilité de l'animateur est d'établir les conditions de la confiance entre participants; cela peut exiger la prise en compte des relations hiérarchiques; la présence d'un responsable peut entraîner des tensions.
- Comme toute réunion impliquant l'ensemble de l'équipe, elle représente un investissement en temps conséquent; si la réunion est inefficace, que ce soit pour des raisons usuelles (manque de préparation, participants en retard ou dissipés) ou spécifiques à ce format (réticences des participants, non-dits), cette pratique pourtant très utile risque d'être discréditée au sein de l'équipe.
- Une rétrospective d'itération doit aboutir à des décisions; ni trop peu (il y a toujours quelque chose à améliorer), ni en trop grand nombre (il ne sera pas possible de régler toutes les difficultés en une seule itération). Une ou deux pistes d'amélioration par rétrospective suffisent.
- Si les mêmes constats reviennent systématiquement d'une rétrospective sur l'autre, sans que les décisions prises soient suivies d'effets, c'est le signe d'une ritualisation des rétrospectives.

COMMENT RECONNAITRE SON UTILISATION?

- assister en observateur à une réunion de ce type est la meilleure façon de constater la mise en place de cette pratique
- à défaut, on sera attentif aux pistes et décisions d'amélioration prises lors de la rétrospective: elles peuvent prendre la forme d'un document écrit, ou simplement de Post-It matérialisant les décisions prises et affichés sur un tableau mural spécifique

QUELS BÉNÉFICES EN ATTENDRE?

- en premier lieu, les rétrospectives permettent de tirer le meilleur parti des cycles itératifs: elles sont l'occasion d'adapter et d'améliorer graduellement le fonctionnement de l'équipe
- en mettant entre les mains des acteurs du projet les décisions sur le fonctionnement du projet, les rétrospectives responsabilisent les participants et favorisent l'appropriation des décisions

ORIGINES

- Alistair Cockburn revendique l'usage du terme "reflection workshop" dès 1995, bien que la première description par écrit apparaisse dans son livre de 2001
- la notion de "reflection" et d'ajustement est inscrite au Manifeste Agile dès 2001, très probablement inspirée par Cockburn
- le terme "Rétrospectives" est dû au livre de Norm Kerth (cf. ci-dessous) en 2001
- la pratique se diffuse notamment par le biais des conférences XP Day en Europe à partir de 2003
- le livre de Diana Larsen et Esther Derby ciblant spécifiquement les équipes Agiles finit de codifier la pratique de la rétrospective d'itération en 2006

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Project Retrospectives, de Norm Kerth (2001)
- Agile Software Development, de Alistair Cockburn (2001, réédité 2006)
- Agile Retrospectives, de Esther Derby et Diana Larsen (2006)

Chapitre 52

Réunion quotidienne

Pratique

DE QUOI S'AGIT-IL?

L'équipe se réunit une fois par jour, à heure fixe, pour mettre en commun les apports de chacun au produit ou à l'infrastructure de développement, et signaler les obstacles rencontrés. On utilise souvent les trois questions suggérées par Scrum pour structurer la réunion.

Cette réunion est généralement timeboxée pour une durée de 15 minutes. Tout sujet risquant de déborder est noté et abordé en dehors de la réunion en plus petit comité.

ON L'APPELLE ÉGALEMENT...

Plusieurs synonymes existent:

- "Stand-up" ou "réunion debout" : appellation Extreme Programming, qui recommande que les participants restent debout pour encourager à raccourcir la durée de la réunion
- "Daily Scrum" ou "mélée quotidienne": bien que ce soit historiquement incorrect (cf. ci-dessous) cette "mélée" (en anglais rugbystique, "scrum") quotidienne est la pratique éponyme de l'approche Scrum

ERREURS COURANTES

- la réunion quotidienne doit être une "place de marché" où les échanges se font d'un membre de l'équipe à un autre; l'erreur la plus courante consiste à en faire un "compte-rendu", chaque équipier s'adressant à une seule personne (son responsable, ou bien le "Scrum Master" désigné de l'équipe)
- la deuxième difficulté la plus couramment rencontrée: une réunion qui s'éternise; des compétences en facilitation en viendront rapidement à bout
- troisième difficulté courante: l'équipe trouve peu de valeur dans la réunion, au point qu'elle "oublie" de la tenir si le Scrum Master ou chef de projet n'en prend pas l'initiative; c'est souvent le symptôme d'une adhésion tiède aux principes de l'approche Agile
- dernier syndrome couramment rencontré: le "tout va bien", aucun membre de l'équipe ne profite de la réunion quotidienne pour faire état d'obstacles ou de difficultés rencontrée, alors même que le fonctionnement de l'équipe est objectivement améliorable - une telle réticence à admettre des difficultés doit amener à réfléchir sur le style d'encadrement de l'entreprise.

COMMENT RECONNAITRE SON UTILISATION?

- assister en observateur à une réunion de ce type est la meilleure façon de constater la mise en place de cette pratique

QUELS BÉNÉFICES EN ATTENDRE?

- la réunion quotidienne favorise la circulation d'informations importantes; elle apporte une occasion explicite permettant à l'ensemble de l'équipe de se synchroniser
- le partage d'informations à l'occasion d'une réunion courte et énergique contribue également à la cohésion de l'équipe

ORIGINES

- Jim Coplien observe chez l'équipe Quattro Pro Windows de Borland, "hyperproductive", l'habitude de tenir une réunion presque tous les jours, mais sans lui donner un nom distinct, dans un article de 1994
- bien que la pratique "Daily Scrum" soit décrite dans les articles concernant Scrum à partir de 2000 elle n'apparaît pas dans les tous premiers écrits (1996)
- (le nom "SCRUM" lui-même n'est pas initialement associé à la "mélée" strictement, mais fait allusion à l'expression "moving the scrum downfield" utilisée dans l'article de Takeuchi et Nonaka "The New New Product Development Game" et exprime l'idée d'auto-organisation)
- on retrouve le "Daily Stand Up" dans les premiers articles sur Extreme Programming (1998)
- les "trois questions" de Scrum sont adoptées par les équipes Extreme Programming dès le début des années 2000
- La pratique trouve sa forme définitive avec la généralisation du tableau des tâches, il devient alors important de tenir la réunion devant le tableau (2004-2006)

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- It's not Just Standing Up, de Jason Yip (2004)

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Peu de travaux de recherche concernent directement cette pratique.

Chapitre 53

Rôle - fonctionnalité - bénéfice

Concept

DE QUOI S'AGIT-IL?

La matrice Rôle-Fonctionnalité-Bénéfice est un format recommandé pour verbaliser ou écrire une User Story:

- En tant que <rôle>
- Je veux que <fonctionnalité>
- Afin de <bénéfice>

Par exemple:

- En tant que client de la banque
- Je veux retirer de l'argent à un guichet automatique
- Afin de ne me soucier ni des horaires d'ouverture ni de l'affluence en agence

Cette formulation permet d'équilibrer les préoccupations: non pas seulement ce que le logiciel doit faire, mais également *pour qui* et *afin de satisfaire quel objectif* de la personne en question.

Il en existe de nombreuses variantes.

Chapitre 54

Salle dédiée

Pratique

DE QUOI S'AGIT-IL?

L'équipe (idéalement au complet, c'est à dire incluant le "client" ou Product Owner) dispose d'un local dédié à plein temps pour le projet, cloisonné des autres activités de l'entreprise.

Ce local est équipé de façon à répondre aux besoins logistiques du projet: stations de travail (si nécessaires adaptées au binômage), tableaux effaçables et chevalets, espaces aux murs pour l'affichage de Post-Its, etc.

ERREURS COURANTES

- le critère de cloisonnement est important: un "open space" ou bureau paysager **n'est pas** une mise en oeuvre adéquate de cette pratique; le bruit occasioné par des personnes étrangères au projet constitue une distraction néfaste à la concentration requise pour un projet de développement

QUELS BÉNÉFICES EN ATTENDRE?

- une salle dédiée favorise ce que Alistair Cockburn nomme la "communication osmotique": la diffusion d'information (jusqu'à l'équilibre) entre ceux qui en disposent et ceux qui en ont

besoin; au lieu de dépendre de mécanismes *explicites* de communication (téléphone, email, réunion) cette diffusion se fait de façon "ambiante", par exemple en entendant (sans les écouter activement) les conversations des autres membres de l'équipe, ou en lisant des Post-It affichés au mur

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Les conditions "environnementales" favorisant la performance des développeurs ont fait l'objet d'un certain nombre d'études. Bien qu'il soit généralement difficile d'établir de façon fiable quoi que ce soit qui ait trait à la productivité, en raison du caractère flou et ambigu de la notion même de "productivité" pour un programmeur, il semble en ressortir que les deux "extrêmes" que sont le bureau individuel **ou** le local dédié soient les plus favorables.

Le bureau paysager, malgré son intérêt économique évident, et l'attrait (plus subtil) qu'il peut exercer sur des managers soucieux de "surveiller" leurs troupes, s'avère être la pire des solutions, retenant le plus mauvais des deux mondes.

- How office space affects programming productivity examine quelques études empiriques assez anciennes et évoque l'intérêt de bureaux privatifs (1995)
- Rapid software development through team collocation, une étude plus récente, semble confirmer qu'un local dédié favorise de meilleures performances (du moins mesurées par des indices subjectifs de satisfaction, du client comme des développeurs) (2002)

Chapitre 55

Scrum de Scrums

Pratique

DE QUOI S'AGIT-IL?

Approche utilisée lorsqu'un projet mobilise plus d'une dizaine de personnes, pour "monter en échelle".

On divise le groupe en petites équipes (de 4 à 9), chacune utilisant Scrum de façon habituelle. A l'issue de chaque réunion quotidienne, un membre de chaque équipe est désigné comme "ambassadeur" pour participer à une meta-réunion, formant ainsi une équipe "virtuelle" dont la mission est de résoudre les difficultés posées par la coordination inter-équipes: création d'interfaces, d'accords de fonctionnement..

Ce "Scrum de Scrums" dispose de son propre backlog qui regroupe toutes les tâches identifiées qui contribuent à une meilleure coordination entre les équipes opérationnelles.

ON L'APPELLE ÉGALEMENT...

Traduction directe de l'anglais "Scrum of Scrums", on parle aussi de "meta Scrum".

ORIGINES

- Décrite pour la première fois dans l'article *Cross-continent development using scrum and XP* de Bent Jensen (2003)

Chapitre 56

Story mapping

Pratique

DE QUOI S'AGIT-IL?

Pratique émergente visant à structurer la planification des livraisons, le "story mapping" consiste en une organisation en deux dimensions des user stories: l'axe horizontal matérialise la succession (chronologique) des usages de l'utilisateur, l'axe vertical matérialise la priorité des fonctionnalités: en haut celles qui sont indispensables à l'usage, en bas celles qui sont annexes ou ne concernent qu'une fraction des flux d'informations que traite le produit.

QUELS BÉNÉFICES EN ATTENDRE?

L'intention de cette pratique est de modérer le risque de livrer un produit qui comporterait des fonctionnalités avec une forte "valeur métier", mais que l'utilisateur ne pourrait pas exploiter parce qu'elles s'avèrent dépendantes de fonctionnalités à plus faible "valeur métier", donc de plus basse priorité et qui n'ont pas encore été réalisées.

ORIGINES

- formulée, sans la nommer, par Jeff Patton dans un article intitulé "It's All in How You Slice" en 2005

- nommée "Story Mapping" dans un article, très illustré, du même Jeff Patton: "The new user story backlog is a map", en 2008

Chapitre 57

Tableau des tâches

Pratique

DE QUOI S'AGIT-IL?

On répartit sur un tableau mural divisé en trois colonnes, "à faire", "en cours", "terminé" des Post-It ou fiches bristol représentant les tâches à réaliser au cours de l'itération.

De nombreuses variantes existent; soit dans la disposition qui peut également être horizontale, ou plus sophistiquée; soit dans le nombre et l'intitulé des colonnes, qui matérialisent en général des activités - par exemple une colonne "en test".

En général au cours de la réunion quotidienne, l'équipe met à jour le tableau au fil de l'itération pour visualiser sa progression. Le tableau est "remis à zéro" en début d'itération avec de nouvelles tâches.

ON L'APPELLE ÉGALEMENT...

En anglais, "task board"; on parle aussi de "project wall".

On entend plus rarement "tableau de Sprint" ou "tableau d'itération".

ERREURS COURANTES

- une erreur classique consiste à préférer d'emblée un support informatisé ("tableau des tâches virtuel"); c'est se priver des

nombreux bénéfices de la réalisation plus "artisanale", et seules de fortes contraintes (par exemple une équipe dispersée géographiquement) doivent justifier cette solution de dernier recours
- ne pas confondre le tableau des tâches "canonique" avec le tableau Kanban dont le principe est assez différent (notamment, il n'est pas "remis à zéro")

QUELS BÉNÉFICES EN ATTENDRE?

- le tableau des tâches est un "radiateur d'information" - il favorise le partage instantané d'informations importantes pour toute l'équipe
- le tableau des tâches sert de point focal pour la réunion quotidienne

ORIGINES

- dès les origines de Scrum et d'Extreme Programming on manipule des fiches bristol ou des Post-It mais le format précis du tableau des tâches ne se généralise qu'assez tardivement
- La galerie de photos collectée par Bill Wake montre que même si de nombreuses équipes Agiles utilisent un affichage mural, le tableau des tâches "standardisé" le plus largement adopté sera celui de Mike Cohn, comportant cinq colonnes (fin 2004)
- En partie en réaction aux tableaux plus sophistiqués produits par les équipes qui commencent à s'intéresser à la transposition Agile des tableaux Kanban, on viendra un peu plus tard à considérer comme "canonique" la version à trois colonnes (fin 2007)

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- Task Boards, de Mike Cohn (2004-2010)
- Task Boards, de Tom Perry

- Elements of Task Board Design, et plus généralement le blog "Visual Management", de Xavier Quesada

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

Peu de travaux de recherche concernent directement cette pratique.

Des investissements considérables sont consacrés par de nombreux éditeurs ou équipes Open Source à la réalisation d'équivalents virtuels du tableau de tâches, alors même que la communauté des praticiens aguerris reste majoritairement persuadée de l'intérêt, voire de la nécessité, du tableau de tâches physique; mettre en évidence le bénéfice réel de cette deuxième solution serait donc une piste de recherche prioritaire.

Chapitre 58

Tableau kanban

Pratique

DE QUOI S'AGIT-IL?

Pratique émergente, portée par une minorité très active, l'approche dite "kanban" regroupe un ensemble de modifications au dispositif Agile le plus couramment utilisé pour le pilotage du projet:

- les itérations, les estimations et la mesure de la vélocité sont abolies;
- la mesure du temps de cycle se substitue à celle de la vélocité
- enfin, la modification la plus visible remplace le tableau des tâches par un tableau dit "tableau kanban":
    - on y retrouve les colonnes correspondant aux différents états par lesquels peut transiter une "unité de valeur" (qui est souvent une user story, mais pas nécessairement)
    - par contre il comporte, outre ces colonnes, des **limites de stock**: si une activité, par exemple le test, se voit attribuer une limite de stock égale à 2, cela signifie qu'il est "interdit" de démarrer l'activité de test concernant une troisième user story, si deux d'entre elles sont déjà en test;
    - si cette situation se présente, il faut au contraire demander à d'autres membres de l'équipe qui sont éventuellement disponibles à ce moment de venir

prêter main-forte, de façon à ce que l'une des deux user stories puisse sortir de l'état "en test" le plus rapidement possible;
- contrairement au tableau des tâches, le tableau kanban n'est pas "réinitialisé" en début de chaque itération, c'est un tableau perpétuel.

ON L'APPELLE ÉGALEMENT...

Le terme est japonais, et signifie simplement "étiquette" ou "carte".

Il désigne un système utilisé dans les usines Toyota; sans entrer dans les détails, il est analogue à ce qu'on peut observer dans certains supermarchés, où derrière un stock, par exemple de boîtes de céréales, on place une carte portant la mention "veuillez réapprovisionner ce produit".

Ce système permet de tenir une comptabilité précise des "stocks" ou "en-cours" (en anglais "Work In Process" qui donne l'acronyme WIP). Le système de production Toyota cherche à réduire la quantité de ces en-cours.

ERREURS COURANTES

Les tableaux kanban sont souvent plus sophistiqués qu'un simple tableau des tâches. Ce n'est pas en soi une erreur, mais il convient d'éviter la réintroduction, par le biais d'un tableau kanban structuré en fonction d'une séquence d'activités, d'un modèle "en cascade" avec tout ce que cela peut impliquer: création de "silos" ou de spécialistes au sein de l'équipe.

On se méfiera en particulier de l'utilisation d'un tableau kanban **sans** l'assortir de "limites de stock" qui soient véritablement opposables aux clients et commanditaires de l'équipe. Ce sont ces limites de stock qui sont l'élément le plus structurant de l'approche dite "kanban".

QUELS BÉNÉFICES EN ATTENDRE?

Il existe des contextes dans lesquels la mesure du temps de cycle et l'utilisation de tableaux kanban (perpétuel) a plus de sens que la mesure de la vélocité à chaque itération: par exemple lorsque le respect d'une

date de livraison précise n'est pas une priorité, ou lorsque l'équipe assure simultanément l'évolution ou la maintenance de produits multiples.

De façon très schématique, on peut considérer que l'approche "kanban" se prête plus à un mode de maintenance ou d'évolution continue, alors que l'approche par itérations se prête plus à un mode projet.

ORIGINES

- le rapprochement entre la philosophie d'organisation du travail en vigueur chez Toyota (sous l'étiquette "Lean") remonte à la publication du livre de Mary et Tom Poppendieck, *Lean Software Development* (2003)
- cependant ce rapprochement reste relativement abstrait dans le début des années 2000; l'un des premiers adeptes de cette approche, David Anderson, expérimente certaines idées (suppression des estimations et des itérations notamment) chez Microsoft, dès 2004
- d'après David Anderson, la première mise en oeuvre complète, y compris l'utilisation d'un tableau kanban avec des limites de stock, a lieu alors qu'il travaille pour Corbis à partir de septembre 2006
- suite à une présentation lors de la conférence Agile aux Etats-Unis, un groupe de discussion se forme pour échanger sur cette approche (mi-2007)
- une communauté plus visible, organisant des conférences dédiées, sous le titre "The Limited WIP Society" finit par se former en 2009

Chapitre 59

Temps de cycle

Concept

DE QUOI S'AGIT-IL?

Le "temps de cycle" est un terme emprunté à l'approche manufacturière connue sous le nom de Lean ou Toyota Production System, où il désigne le délai entre la réception de la commande d'un client, et la livraison au client du produit fini.

Transposé au domaine du logiciel, le temps de cycle est le délai écoulé entre le moment où apparaît un besoin, et la satisfaction de ce besoin. Cette définition plus abstraite permet d'analyser plusieurs types de situation: par exemple on peut mesurer le "temps de cycle" entre la formalisation d'une exigence sous la forme d'une user story et le début de l'utilisation ("en production", en conditions réelles) de la fonctionnalité correspondante.

Les équipes ayant adopté une approche à base de kanban privilégient cette mesure, de préférence à celle de la vélocité. Au lieu d'avoir pour effet d'augmenter la vélocité, les améliorations du fonctionnement de l'équipe ont (en principe) pour effet de diminuer le temps de cycle.

ON L'APPELLE ÉGALEMENT...

Le terme anglais est "lead time".

Chapitre 60

Test d'utilisabilité

Pratique

DE QUOI S'AGIT-IL?

Le test d'utilisabilité est une technique empirique et exploratoire permettant de répondre à la question "comment un utilisateur appréhendra notre produit dans les conditions réelles d'utilisation ?".

On place un utilisateur final devant le produit, en lui fixant une tâche typique de celles que le produit permet de réaliser (par exemple "utilisez notre site de banque en ligne pour régler par virement votre facture de téléphone").

Les développeurs et/ou ergonomes observent l'utilisateur *sans intervenir* et enregistrent (soit informellement en prenant des notes, soit de façon plus complète par des moyens tels que la vidéo, la capture d'écran ou l'eye-tracking) les difficultés éventuelles rencontrées par l'utilisateur, afin d'améliorer la prise en main du produit dans des versions ultérieures.

ORIGINES

- l'invention de cette technique est attribué aux ingénieurs du célèbre Xerox PARC à Palo Alto en 1981
- l'assimilation progressive (et encore balbutiante) de cette pratique est représentative de l'ouverture récente de la communauté Agile aux techniques issues des disciplines

"orientées utilisateurs" que sont l'ergonomie ou l'Interaction Design (à partir de 2008)

Chapitre 61

Test exploratoire

Pratique

DE QUOI S'AGIT-IL?

Au sein d'une équipe Agile, la répartition usuelle des rôles entre les spécialistes du développement et les spécialistes du test se trouve fortement modifiée suite à l'utilisation de tests unitaires et fonctionnels automatisés. Une grande partie des activités de vérification consistant à dérouler des scripts, scénarios ou plans de test est désormais sous la responsabilité des développeurs.

Cependant, il s'avère que ces tests automatisés ne suffisent pas à évaluer les risques de défauts de qualité dont peut souffrir le produit: les spécialistes du test continuent à jouer un rôle important dans les projets Agiles.

L'approche exploratoire de l'activité de test s'avère très complémentaire de l'approche Agile pour plusieurs raisons:

- elle insiste sur l'autonomie, la compétence et la créativité du testeur, tout comme l'approche Agile met en avant ces qualités du développeur;
- elle préconise de mener de front toutes les activités liées au test: découverte d'informations sur le produit, conception de tests susceptibles de révéler des défauts et exécution de ces tests;
- elle insiste sur l'importance d'une pluralité de techniques, qui ne peut en aucun cas se réduire à un "plan de test" formel, et

par conséquent rejoint la philosophie Agile en faisant peu de cas des référentiels documentaires pour le test

ON L'APPELLE ÉGALEMENT...

Le terme anglais est "exploratory testing". Il est utilisé par une communauté de testeurs qui revendiquent leur appartenance intellectuelle à une "école de pensée" qu'ils appellent "the Context-Driven School" et qu'ils distinguent des autres "écoles" ayant des approches différentes de l'activité de test: Analytique, Standardisée, Orientée Qualité, et Agile.

QUELS BÉNÉFICES EN ATTENDRE?

Le rôle des testeurs est souvent marginal au sein des projets de développement en France. Cependant, dans les équipes travaillant avec des testeurs professionnels, l'approche Agile suscite souvent des interrogations profondes sur la pertinence et l'organisation de leur activité. L'approche "exploratoire" offre de nombreuses pistes pour intégrer efficacement ces spécialistes dans une équipe Agile.

ORIGINES

La pratique du test exploratoire, bien que restée relativement marginale dans la communauté Agile, est évangélisée dès 2001 par Brian Marick, l'un des participants au séminaire de Snowbird qui donne naissance au Manifeste Agile, et qui se décrit lui-même comme "le testeur de service" au sein de ce groupe.

Il est indéniable que l'essor du mouvement Agile a contribué à renforcer l'intérêt de nombreux développeurs pour les activités de test, jusqu'alors peu valorisées dans la profession. Cependant, il a également eu l'effet paradoxal de marginaliser un peu plus le testeur, puisque c'est le développeur qui s'appropriait certaines activités de test.

Chapitre 62

Tests fonctionnels automatisés

Pratique

DE QUOI S'AGIT-IL?

Un test fonctionnel, au sens Agile, est une description formelle du comportement d'un produit, généralement exprimée sous la forme d'un exemple ou d'un scénario.

Des formalismes très divers existent pour l'expression de ces exemples ou scénarios, mais on cherche le plus souvent à ce qu'il soit possible d'en automatiser le déroulement à l'aide d'un outil (interne à l'équipe de développement, ou disponible "sur étagère").

Comme pour un test unitaire son résultat est binaire, un "échec" (comportement différend de celui attendu) laissant présumer la présence d'un bug.

De tels tests tiennent lieu, pour une équipe Agile, de cahier des charges principal: il arrive qu'ils complètent et précisent un cahier des charges rédigé selon une technique non spécifiquement Agile ("use cases", ou encore document narratif); et dans certains cas qu'ils constituent l'unique expression formelle des exigences.

ON L'APPELLE ÉGALEMENT...

On parle également de "tests de recette", plus rarement de "tests client" (par opposition à "tests développeur"). La traduction littérale "tests d'acceptation" est aussi utilisée (de l'anglais "acceptance test").

Le terme anglais "Storytest" (le plus souvent en un seul mot) est parfois utilisé, dans l'expression "storytest driven development" par exemple.

ERREURS COURANTES

- un travers courant consiste à exprimer les tests fonctionnels dans un formalisme trop proche de l'implémentation, ce qui les rend inintelligibles par les clients, utilisateurs ou experts métiers qui en sont l'audience privilégiée; on rencontre souvent ce défaut lorsque ce sont les développeurs qui prennent l'initiative de mettre en place cette pratique, il est donc **impératif de faire participer client, utilisateur ou expert métier** à cette activité
- une autre conséquence de ce type de dérive est de rendre les tests fonctionnels "fragiles", c'est à dire qu'ils échouent suite à des modifications de comportement qui sont sans rapport avec la fonctionnalité testée (par exemple la modification de l'étiquette d'un champ de texte)

ORIGINES

- la pratique est intégrée dans Extreme Programming dès ses débuts, mais sans être liée à un outil ou un formalisme particulier; sa description recouvre sous une même étiquette tests unitaires et de recette (1996)
- Ward Cunningham, l'un des créateurs d'Extreme Programming, publie l'outil Fit pour formaliser les tests de recette (2002)
- l'adoption de Fit reste marginale pendant quelque temps, mais explose lorsque Bob Martin fusionne Fit avec une autre création de Cunningham, le Wiki, donnant naissance à FitNesse (2003)

- pendant quelque temps, le duo Fit/FitNesse éclipse les autres outils et s'impose en modèle pour la mise en oeuvre de tests fonctionnels Agiles
- plus récemment de nouveaux outils liés à l'approche BDD ont contribué à réouvrir le débat

QUELS BÉNÉFICES EN ATTENDRE?

L'utilisation de tests fonctionnels automatisés apporte plusieurs bénéfices, complémentaires de ceux liés aux tests unitaires:

- ils encouragent à une collaboration étroite entre les développeurs d'une part et les clients, utilisateurs ou experts métier d'autre part, puisqu'ils imposent de préciser suffisamment les exigences pour être en mesure de les exprimer dans un formalisme exécutable
- ils fournissent par la même occasion un "contrat" clair et précis pour l'acceptation par le client du travail fourni par les développeurs; si le produit passe les tests fonctionnels, il est considéré adéquat (mais les clients ou utilisateurs ont la possibilité d'affiner les tests ou d'en proposer de nouveaux si nécessaire)
- ils contribuent également à limiter le nombre de défauts et notamment de régressions (défauts apparus sur des fonctionnalités précédemment validées)

QUELS COÛTS OU INVESTISSEMENTS FAUT-IL CONSENTIR?

Les tests fonctionnels, contrairement aux tests unitaires, font l'objet d'une controverse au sein de la communauté Agile, certains experts tels que Jim Shore ou Brian Marick jugeant que les bénéfices obtenus ne justifiaient pas les coûts à consentir:

- beaucoup d'équipes Agiles constatent que l'écriture de tests fonctionnels automatisé représente un effort important
- pour des raisons liées aux erreurs de mise en oeuvre évoquées ci-dessus, la maintenance de ces tests peut également s'avérer contraignante et coûteuse
- la première génération d'outils, inspirée de Fit/Fitnesse, a souvent conduit à des tests fonctionnels qui n'étaient pas

acceptés comme intelligibles par les clients, utilisateurs ou experts métier

Cependant, les évolutions récentes notamment celles dûes au BDD peuvent être de nature à améliorer ce rapport coût-bénéfice.

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

- Automated Acceptance Testing: A Literature Review and an Industrial Case Study, passe en revue les articles sur le sujet et rapporte une étude de cas globalement positive

Chapitre 63

Tests unitaires automatisés

Pratique

DE QUOI S'AGIT-IL?

Un test unitaire, au sens Agile, est un court programme, écrit et maintenu par les développeurs, servant à vérifier de manière très étroite le bon fonctionnement d'une partie restreinte du programme principal.

Son résultat est binaire: il "passe" si le comportement du programme est celui attendu et "échoue" dans le cas contraire. Une convention héritée de l'outil JUnit et de ses nombreux dérivés veut qu'on représente par une "barre rouge" l'échec d'un ou plusieurs tests; une "barre verte" symbolise l'exécution avec succès de la **totalité** des tests unitaires associés à un produit.

ON L'APPELLE ÉGALEMENT...

On parle également de "tests développeur" (par opposition à "tests client") pour insister sur la fonction qui en est responsable, plutôt que sur le niveau d'abstraction technique.

ERREURS COURANTES

La terminologie entourant les différents types de test prête à confusion. En effet, la mouvance Agile a repris à son compte des termes tels que "tests unitaires" déjà utilisés auparavant en leur donnant un sens différent (dans un environnement Agile ce terme ne saurait désigner que des tests automatisés, par exemple).

Il est également important de différencier la simple utilisation de tests unitaires automatisés, réalisés par des programmeurs, de la pratique plus contraignante qu'est le développement par les tests.

ORIGINES

- plusieurs articles du même auteur, David J. Panzl, attestent de l'invention d'outils très proches de JUnit dès 1976
- l'émergence des outils du type "enregistrer et rejouer" a sans doute contribué à marginaliser les tests unitaires plus proches du code, de la fin des années 80 jusqu'au milieu des années 90
- la tendance s'inversera grâce à la popularité du développement par les tests et de l'outil JUnit dans la fin des années 90
- la pratique est intégrée dans Extreme Programming dès ses débuts sous la forme du développement par les tests (1996)

QUELS BÉNÉFICES EN ATTENDRE?

- une équipe utilisant des tests unitaires automatisés récoltera une partie des bénéfices attribués au développement par les tests: réduction du nombre de défauts, principalement

Chapitre 64

Trois questions

Pratique

DE QUOI S'AGIT-IL?

Les réunions quotidiennes sont structurées autour des trois questions suivantes:

- Qu'as-tu terminé depuis la précédente réunion?
- Que penses-tu pouvoir terminer d'ici la prochaine réunion?
- Quels obstacles rencontres-tu en ce moment?

ERREURS COURANTES

- il est important d'insister sur le mot "terminé" dans ces questions; une erreur courante consiste à répondre: "hier j'ai travaillé sur le schéma de la base de données... et je continue là-dessus aujourd'hui... sinon tout va bien"; dans ce cas-là il vaut mieux dire "je n'ai rien terminé de nouveau"; cf. la définition de 'terminé'

ORIGINES

Ces trois questions sont à l'origine une spécificité des réunions Scrum, désormais adoptée par presque toutes les équipes Agiles

Chapitre 65

Tâches choisies

Pratique

DE QUOI S'AGIT-IL?

Au sein d'une équipe Agile, il est rare qu'un responsable attribue des tâches de développement aux différents membres de l'équipe. Le plus souvent l'affectation des tâches est basée sur le volontariat et la discussion.

En règle générale ce choix des tâches s'effectue devant le tableau des tâches au cours de la réunion quotidienne. Il est souvent matérialisé par une annotation sur le Post-It représentant la tâche concernée. Certaines équipes utilisent des photos (format identités) ou de plus petits Post-It qui sont collées de façon amovible à celui représentant la tâche qu'ils ont choisi de réaliser.

ERREURS COURANTES

Bien que rarement identifiée comme une pratique à part entière, l'attribution des tâches sur la base du volontariat semble être un élément important, sinon essentiel, constitutif d'une approche Agile.

Certaines équipes "en transition" vers l'Agilité peuvent conserver quelque temps un rôle de "chef de projet" dont l'une des prérogatives est traditionnellement la distribution des tâches au membres de l'équipe. Ce modèle est cependant largement considéré comme incompatible sur le long terme avec les autres pratiques Agiles.

Chapitre 66

User stories

Compétence

DE QUOI S'AGIT-IL?

L'intégralité du travail à réaliser est découpée en incréments fonctionnels et les activités de développement s'organisent autour de ces incréments appelés "User Stories".

Adopter la pratique des User Stories implique de tenir pour généralement vraies un certain nombre d'hypothèses sur ces incréments: on peut les réaliser dans un ordre arbitraire, et chacun indépendamment des autres peut avoir une valeur pour l'utilisateur.

Pour rendre ces hypothèses très concrètes, on privilégie une *représentation* (on peut même parler de réification) de ces User Stories sous une forme bien précise: la fiche cartonnée ou le Post-It, qui en renforcent le caractère atomique et manipulable par tous les intervenants du projet.

ON L'APPELLE ÉGALEMENT...

Littéralement "histoire utilisateur", le terme "scénario client" est également utilisé. Dans les deux cas il s'agit de mettre en avant l'aspect narratif ("voici ce qui se passe") et d'autre part le point de vue adopté, celui, non technique, de l'utilisateur ou du donneur d'ordres ("client").

Au singulier: "une user story".

ERREURS COURANTES

- une "erreur" classique consiste à partir d'un cahier des charges rédigé en amont et le découper en User Stories en s'appuyant sur sa structure textuelle; cela peut être une bonne approche pour une équipe débutante mais ne constitue pas une pratique recommandée
- une User Story n'est pas un document; le terme désigne l'intégralité des activités qui sont nécessaires pour transformer une version V du produit en une version V' qui a plus de valeur; en ce sens une User Story est un *objectif*
- le niveau de détail correspondant à une User Story n'est pas constant, mais évolue au fil du temps, en fonction de l'horizon de planification: une User Story dont la réalisation est prévue dans plusieurs semaines ne sera décrite que d'une brève phrase, alors qu'une User Story dont la réalisation est imminente doit être cadrée par des tests de recette, des exemples, etc.
- une User Story n'est pas un Use Case; bien que la comparaison entre les deux notions soit souvent utile, il n'est pas possible d'établir un lien simple et direct
- une User Story ne correspond en règle générale pas à un quelconque découpage technique: un écran, une boîte de dialogue, un bouton ne sont pas des User Stories

ORIGINES

Les User Stories sont issues d'Extreme Programming. Elles ne sont initialement pas décrites comme une pratique à part entière; elles n'apparaissent dans la première édition du livre de Kent Beck, Extreme Programming Explained, que comme une des "pièces" utilisées dans le "jeu du planning".

Au fil des années, une pression croissante pousse la communauté Agile et sa composante Extreme Programming en particulier à répondre de façon plus précise à la question "comment sont traitées les exigences", il émergera de cette pression une définition considérablement plus sophistiquée de ces User Story:

- la matrice rôle-fonctionnalité est proposée par Rachel Davies et Tim McKinnon en 2001

- le modèle des "3C", ou "Carton, Conversation, Confirmation" est proposé par Ron Jeffries en 2001 ("Card, Conversation, Confirmation")
- la grille INVEST est décrite par Bill Wake en 2003 ("INVEST in Good Stories and SMART tasks")
- la matrice given-when-then pour décrire les tests de recette est décrite par Dan North en 2006 ("Introducing BDD")

COMMENT S'AMÉLIORER?

A titre individuel, les personnes concernées par l'acquisition de *compétences* en matière de User Stories sont en premier lieu celles occupant le rôle et les responsabilités d'analystes. Il est très souhaitable, comme pour la plupart des compétences Agiles, que celles-ci soit largement partagées dans l'équipe.

A titre individuel:

- Débutant
  - je suis capable de partir d'une autre formalisation (cahier des charges, use cases, etc.) et la transposer en User Stories
  - je connais au moins une matrice d'expression des User Stories
  - je suis capable d'illustrer une User Story par un exemple (raconter ce que fait l'utilisateur et le résultat)
- Intermédiaire
  - je suis capable de "morceler" l'ensemble des objectifs fonctionnels d'un projet en User Stories, en m'assurant qu'il n'y a pas d'oubli
  - je connais différentes matrices d'expression des User Stories et suis capable d'utiliser la plus appropriée
  - je suis capable de formaliser les critères de satisfaction d'une User Story sous une forme susceptible d'être transformée en test de recette
  - je connais ou puis identifier les différentes populations d'utilisateurs et m'y référer dans les User Stories
  - je suis capable d'évaluer une User Story selon la grilles INVEST ou un équivalent, et reformuler ou

découper une User Story sans produire un résultat dégradé
- Avancé
  - je suis capable d'interpréter des exigences considérées comme "non fonctionnelles" sous la forme d'une User Story: persistance, performance, portabilité...
  - je suis capable de rattacher des User Stories à des descriptions de plus haut niveau du projet: "charte projet", etc. - et de justifier pour chaque User Story sa pertinence et sa valeur

A titre collectif:
- Débutant
  - l'équipe sait organiser son activité autour des User Stories
  - l'équipe est capable d'obtenir "juste à temps" le détail nécessaire à l'implémentation des User Stories, par exemple par la présence physique du client ou donneur d'ordres
- Intermédiaire
  - l'équipe formalise la totalité de son travail sous la forme de User Stories, tout membre de l'équipe peut répondre à la question "sur quelle User Story travailles-tu?"
- Avancé
  - a tout moment l'équipe est capable de répondre à la question "quelle est la User Story la plus importante restant à faire et pourquoi?"

COMMENT RECONNAITRE SON UTILISATION?

- l'équipe utilise des outils de planification visuels (Release Plan, Story Map, Task Board) et on peut y voir des fiches cartonnées ou Post-It
- les descriptifs de User Stories ne contiennent pas ou peu de langage technique ("base de données", "écran X" ou "dialogue Y") mais font référence à des objectifs d'utilisateurs

QUELS BÉNÉFICES EN ATTENDRE?

La pratique des User Stories conduit à un développement incrémental, et doit permettre d'en réaliser les bénéfices, notamment

- une réduction des risques liés à l'effet tunnel, d'autant plus importante
    - que les incréments sont de petite grandeur
    - que les mises en production ont lieu tôt et fréquemment
- pour le donneur d'ordres, la possibilité de changer d'avis en cours de route sur les détails d'implémentation ou la priorité donnée à une User Story non encore réalisée
- pour les développeurs, l'obtention de critères d'acceptation et de validation clairs et précis, et la validation de leur travail au fil de l'eau
- la possibilité "d'auto-financer" le projet en réalisant très tôt les bénéfices d'une mise en exploitation rendue possible par l'approche incrémentale
- en scindant clairement le "quoi" et le "comment" (les donneurs d'ordre doivent se cantonner à décrire le "quoi" à savoir les objectifs, les développeurs restant force de proposition sur le "comment"), permettre d'exploiter au mieux le talent et la créativité de chacun

QUELS COÛTS OU INVESTISSEMENTS FAUT-IL CONSENTIR?

Le développement incrémental en général nécessite une stratégie de test plus élaborée. En effet, le risque d'effets de bord lors du développement d'une fonctionnalité F2, entraînant la régression d'une fonctionnalité F1, exige que les fonctionnalités existantes soit re-testées. Dans le cas le plus pessimiste, le coût du test évolue donc comme le *carré* du nombre de fonctionnalités: on doit tester successivement F1, F1+F2, F1+F2+F3, et ainsi de suite. Plus les incréments sont de petite grandeur, plus cet effet est sensible.

OÙ SE RENSEIGNER POUR EN SAVOIR PLUS? (LIVRES, ARTICLES)

- User Stories Applied, de Mike Cohn
- Software By Numbers examine les aspects économiques du développement itératif

PUBLICATIONS ACADÉMIQUES ET TRAVAUX DE RECHERCHE

- Théoriques
    - Les travaux de Robert Biddle, Angela Martin, James Noble et leurs collaborateurs sont parmi les premiers à se pencher spécifiquement sur le rôle du Client dans les approches agiles et à en analyser les mécanismes, portés par les User Stories
- Empiriques
    - ...

Chapitre 67

Vélocité

Concept

DE QUOI S'AGIT-IL?

A la fin d'une itération, l'équipe additionne les estimations associées aux user stories qui ont été terminées au cours de cette itération. Ce total est appelé vélocité.

Une fois connue, la vélocité peut être utilisée pour valider ou réviser la planification de l'ensemble du projet, en partant du principe que la vélocité lors de futures itérations sera approximativement égale à la dernière vélocité constatée.

Exemple: une équipe entame une itération, prévoyant de terminer les stories suivantes: A, estimée à 2 points; B, estimée à 2 points; C, estimée à 3 points. A la fin de l'itération, les stories A et B sont terminées à 100% mais C n'est terminée qu'à 80%.

Une équipe Agile ne reconnait que deux niveaux de complétion: 0% ou 100%. Par conséquent C n'est pas comptabilisée, et la vélocité de l'itération est de 4 points. Supposons que la totalité des stories du projet représente 40 points; on prévoit donc une durée totale du projet équivalente à 10 itérations.

L'itération suivante, l'équipe devrait ne prévoir qu'un lot de stories équivalent à 4 points, afin d'éviter de retomber dans le travers que constitue une story terminée à 80%. La vélocité fonctionne ainsi comme une soupape permettant de faire retomber la pression lorsque l'équipe rencontre des difficultés à tenir ses engagements. (Si la story C est

complétée au début de l'itération suivante, les 3 points correspondants pourront être comptabilisés dans la vélocité de cette dernière. La vélocité de l'équipe va donc "naturellement" remonter.)

ERREURS COURANTES

Cette définition a plusieurs conséquences importantes:

- la vélocité est une **constatation**, une mesure à postériori, et non un budget ou une prévision; parler de "fixer une vélocité" est un contresens
- on parle de la vélocité d'une équipe uniquement, en aucun cas de vélocité individuelle; cela n'aurait pas de sens, une équipe étant conçue précisément comme un ensemble plus performant que la somme de ses individus
- on ne peut pas directement comparer la vélocité de deux équipes différentes, puisque ces équipes peuvent avoir émis leurs estimations sur des bases différentes
- pour que la vélocité permette la prévision d'une date de fin du projet, il est impératif que l'ensemble des user stories que comprend le projet soient estimées de façon cohérente; il existe deux manières d'y parvenir:
    - estimer la totalité des user stories très tôt dans le projet (avant ou pendant les premières itérations)
    - utiliser une technique d'estimations relatives pour s'assurer que les estimations tardives sont émises sur la même base que celles émises en début de projet

ORIGINES

- le terme de "vélocité" provient d'Extreme Programming, où il apparaît assez tardivement, remplaçant le terme "load factor" dont la définition est jugée trop complexe (milieu de l'année 2000)
- la définition canonique apparaît dans *Planning Extreme Programming* de Beck et Fowler (2001)
- le terme est ensuite adopté par la communauté Scrum à partir de 2002

Chapitre 68

Bibliographie académique

La centaine de références ci-dessous donne un aperçu des quelques pratiques Agiles qui ont le plus intéressé les chercheurs (et dessinent en creux le travail restant à effectuer dans ce domaine).

La sélection que nous proposons s'intéresse principalement aux études empiriques, laissant de côté la plupart des travaux purement conceptuels ou spéculatifs qui n'ont pas fait l'objet d'une collecte de données.

BDD (Behaviour-Driven Development)

- *Towards an empirical evaluation of behaviour-driven development* (Laplante, C.S., 2011)
- *Behaviour-Driven Development of Foundational UML Components* (Lazr, Ioan and Motogna, Simona and Pirv, Bazil, 2010)

Charte projet

- *Charters and Chartering: Immunizing Against Foreseeable Project Failure* (III, 2001)

Développement par les tests

- *Empirical Studies for the Application of Agile Methods to Embedded Systems* (Wilking, Dirk, 2008)

- *The Effect of Test-Driven Development on Program Code* (Muller, Matthias, 2006)
- *Evaluation of Test Driven Development* (Wasmus, Hans, 2006)
- *A Leveled Examination of Test-Driven Development Acceptance* (Janzen, David S. and Saiedian, Hossein, 2007)
- *On the Sustained Use of a Test-Driven Development Practice at IBM* (Sanchez, Julio Cesar and Williams, Laurie and Maximilien, E. Michael, 2007)
- *Programmer's Expertise during Test-Driven Software Development.* (Shaochun Xu and Zendi Cui and Dapeng Liu and Xuhui Chen, 2007)
- *Realizing quality improvement through test driven development: results and experiences of four industrial teams* (Nagappan, Nachiappan and Maximilien, E. Michael and Bhat, Thirumalesh and Williams, Laurie, 2008)
- *Adding planned design to xp might help novices' productivity (or might not): two controlled experiments* (Noel, René and Valdes, Gonzalo and Visconti, Marcello and Astudillo, Hernan, 2008)
- *Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study* (Siniaalto, Maria and Abrahamsson, Pekka, 2008)
- *The Impact of Test Driven Development on the Evolution of a Reusable Framework of Components - An Industrial Case Study* (Slyngstad, Odd Petter N. and Li, Jingyue and Conradi, Reidar and Ronneberg, Harald and Landre, Einar and Wesenberg, Harald, 2008)
- *The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study* (Madeyski, Lech and Szala, Lukasz, 2007)
- *Effective test driven development for embedded software* (Karlesky, M. and Bereza, W. and Erickson, C., 2006)
- *On the Influence of Test-Driven Development on Software Design* (Janzen, D.S. and Saiedian, H., 2006)
- *Empirical investigation towards the effectiveness of Test First programming* (Liang Huang and Mike Holcombe, 2009)
- *An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development* (Gupta, Atul and Jalote, Pankaj, 2007)
- *A Prototype Empirical Evaluation of Test Driven Development* (Geras, A. and Smith, M. and Miller, J., 2004)

- *A structured experiment of test-driven development* (George, B. and Williams, L., 2004)
- *Lessons learned from an XP Experiment with Students: Test-First needs more teachings* (Flohr, T. and Schneider, T., 2006)
- *Using software testing to move students from trial-and-error to reflection-in-action* (Edwards, Stephen H., 2004)
- *Test driven development of embedded systems using existing software test infrastructure* (Dowty, M., 2004)
- *A survey of evidence for test-driven development in academia* (Desai, Chetan and Janzen, David and Savage, Kyle, 2008)
- *The effect of experience on the test-driven development process* (Muller, Matthias M. and Hofer, Andreas, 2007)
- *Quality Impact of Introducing Component-Level Test Automation and Test-Driven Development* (Damm, Lars-Ola and Lundberg, Lars, 2007)
- *Results from introducing component-level test automation and test-driven development* (Damm, Lars-Ola and Lundberg, Lars, 2006)
- *Evaluating the efficacy of test-driven development: industrial case studies* (Bhat, Thirumalesh and Nagappan, Nachiappan, 2006)
- *Improving Business Agility Through Technical Solutions: A Case Study on Test-Driven Development in Mobile Software Development* (Abrahamsson, Pekka and Hanhineva, Antti and Jaalinoja, Juho, 2005)
- *Test Driven Development and Software Process Improvement in China* (Lui, Kim Man and Chan, Keith C.C., 2004)
- *An experimental evaluation of continuous testing during development* (Saff, David and Ernst, Michael D., 2004)
- *Using test-driven development in the classroom: providing students with automatic, concrete feedback on performance* (Edwards, S., 2003)
- *Improving student performance by evaluating how well students test their own programs* (Edwards, Stephen H., 2003)
- *Evaluating advantages of test driven development: a controlled experiment with professionals* (Canfora, Gerardo and Cimitile, Aniello and Garcia, Felix and Piattini, Mario and Visaggio, Corrado Aaron, 2006)
- *An initial investigation of test driven development in industry* (George, Boby and Williams, Laurie, 2003)
- *Implications of test-driven development: a pilot study* (Kaufmann, Reid and Janzen, David, 2003)

- *Assessing test-driven development at IBM* (Maximilien, E. Michael and Williams, Laurie, 2003)
- *Towards empirical evaluation of test-driven development in a university environment* (Pancur, M. and Ciglaric, M. and Trampus, M. and Vidmar, T., 2003)
- *Web-CAT: A Web-based Center for Automated Testing* (Shah, Anuj, 2003)
- *Test-Driven Development as a Defect-Reduction Practice* (Williams, Laurie and Maximilien, E. Michael and Vouk, Mladen, 2003)
- *Analysis and Quantification of Test Driven Development Approach* (George, B., 2002)
- *Experiment about test-first programming* (Muller, M.M. and Hagner, O., 2002)
- *An Informal Formal Method for Systematic JUnit Test Case Generation* (Stotts, P. David and Lindsey, Mark and Antley, Angus, 2002)
- *The effect of unit tests on entry points, coupling and cohesion in an introductory Java programming course* (Steinberg, Daniel H., 2001)
- *Integrating Unit Testing into a Software Development Team's Process* (R.A. Ynchausti, 2001)
- *Test-Driven Development of Relational Databases* (Ambler, Scott W., 2007)
- *Does Test-Driven Development Really Improve Software Design Quality?* (Janzen, D.S. and Saiedian, H., 2008)
- *On the effectiveness of the test-first approach to programming* (Hakan Erdogmus and Maurizio Morisio and Ieee Computer Society and Marco Torchiano and Ieee Computer Society, 2005)
- *Automated Recognition of Test-Driven Development with {Z}orro* (Johnson, Philip M. and Kou, Hongbing, 2007)
- *The impact of test-driven development on design quality* (Maria Siniaalto, 2006)
- *Test-driven development: empirical body of evidence* (Maria Siniaalto, 2006)

Développement par les tests client

- *Empirical analyses of executable acceptance test driven development* (Melnik, Grigori Igorovych, 2007)

- *Are fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks* (Ricca, Filippo and Di Penta, Massimiliano and Torchiano, Marco and Tonella, Paolo and Ceccato, Mariano and Visaggio, Corrado Aaron, 2008)
- *FitClipse: a fit-based eclipse plug-in for executable acceptance test driven development* (Deng, Chengyao and Wilson, Patrick and Maurer, Frank, 2007)
- *Using acceptance tests as a support for clarifying requirements: A series of experiments* (Ricca, Filippo and Torchiano, Marco and Di Penta, Massimiliano and Ceccato, Mariano and Tonella, Paolo, 2009)

Objets fantaisie (Mock Objects)

- *An empirical study of testing file-system-dependent software with mock objects* (Marri, M.R. and Tao Xie and Tillmann, N. and de Halleux, J. and Schulte, W., 2009)

Personas

- *Personas - a Literature Review*
- *Personas is not applicable: local remedies interpreted in a wider context* (Ronkko, Kari and Hellman, Mats and Kilander, Britta and Dittrich, Yvonne, 2004)
- *Resolving Incommensurable Debates: A Preliminary Identification of Persona Kinds, Attributes, and Characteristics* (Ingbert R. Floyd and M. Cameron Jones and Michael B. Twidale, 2008)
- *Real or Imaginary: The effectiveness of using personas in product design* (Long, Frank, 2009)
- *An Empirical Study Demonstrating How Different Design Constraints, Project Organization and Contexts Limited the Utility of Personas* (Ronkko, K., 2005)

Planning poker

- *Using planning poker for combining expert estimates in software projects* (Molokken-Ostvold, Kjetil and Haugen, Nils Christian and Benestad, Hans Christian, 2008)

- *Combining Estimates with Planning Poker–An Empirical Study* (Molokken-Ostvold, Kjetil and Haugen, Nils Christian, 2007)
- *An Empirical Study of Using Planning Poker for User Story Estimation* (Haugen, Nils C., 2006)

Programmation en binômes

- *Controlled experimentation on adaptations of pair programming* (Domino, Madeline Ann and Collins, Rosann Webb and Hevner, Alan R., 2007)
- *Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality* (Madeyski, Lech, 2005)
- *Pair programming productivity: Novice-novice vs. expert-expert* (Lui, Kim Man and Chan, Keith C. C., 2006)
- *Personality and the nature of collaboration in pair programming* (Walle, Thorbjorn and Hannay, Jo E., 2009)
- *An empirical comparison between pair development and software inspection in Thailand* (Phongpaibul, Monvorath and Boehm, Barry, 2006)
- *The case for collaborative programming* (Nosek, John T., 1998)
- *A multiple case study on the impact of pair programming on product quality* (Hulkko, Hanna and Abrahamsson, Pekka, 2005)
- *Video analysis of pair programming* (Hofer, Andreas, 2008)
- *Effects of Personality on Pair Programming* (Jo E. Hannay and Erik Arisholm and Harald Engvik and Dag I.K. Sjoberg, 2010)
- *The benefits of pairing by ability* (Braught, Grant and MacCormick, John and Wahls, Tim, 2010)
- *An interpretation of the results of the analysis of pair programming during novices integration in a team* (Fronza, Ilenia and Sillitti, Alberto and Succi, Giancarlo, 2009)
- *The Economics of Software Development by Pair Programmers* (Erdogmus, H. and Williams, L., 2003)
- *The Costs and Benefits of Pair Programming* (Cockburn, Alistair and Williams, Laurie, 2000)
- *The effectiveness of pair programming: A meta-analysis* (Hannay, Jo E. and Dybaa, Tore and Arisholm, Erik and Sjoberg, Dag I. K., 2009)

- *Strengthening the Case for Pair Programming* (Laurie Williams and Robert R. Kessler and Ward Cunningham and Ron Jeffries, 2000)
- *The collaborative software process(sm)* (Williams, Laurie Ann, 2000)
- *"Talking the talk": Is intermediate-level conversation the key to the pair programming success story?* (Freudenberg, S. (nee Bryant) and Romero, P. and du Boulay, B., 2007)
- *Are Two Heads Better than One? On the Effectiveness of Pair Programming* (Dybaa, Tore and Arisholm, Erik and Sjoberg, Dag I. K. and Hannay, Jo E. and Shull, Forrest, 2007)
- *The Social Dynamics of Pair Programming* (Chong, Jan and Hurlbutt, Tom, 2007)
- *Critical Personality Traits in Successful Pair Programming* (Chao, Joseph and Atli, Gulgunes, 2006)
- *A replicated experiment of pair-programming in a 2nd-year software development and design computer science course* (Mendes, Emilia and Al-Fakhri, Lubna and Luxton-Reilly, Andrew, 2006)
- *The effects of pair-programming on individual programming skill* (Braught, Grant and Eby, L. Martin and Wahls, Tim, 2008)
- *Pair programming: what's in it for me?* (Begel, Andrew and Nagappan, Nachiappan, 2008)
- *Promiscuous pairing and beginner's mind: embrace inexperience [agile programming]* (Belshee, A., 2005)
- *Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise* (Erik Arisholm and Hans Gallis and Tore Dyba and Dag I.K. Sjoberg, 2007)
- *Pair programming improves student retention, confidence, and program quality* (C McDowell and L Werner and H E Bullock and J Fernald, 2006)
- *The effects of pair-programming on performance in an introductory programming course* (C McDowell and L Werner and H Bullock and J Fernald, 2002)
- *Experimental evaluation of pair programming* (Jerzy Nawrocki and Adam Wojciechowski, 2001)
- *Code Warriors and Code-a-Phobes: A study in attitude and pair programming* (L Thomas and M Ratcliffe and A Robertson, 2003)
- *Female Computer Science Students Who Pair Program Persist* (Linda Werner and Charlie McDowell and Brian Hanks, 2004)

Refactoring

- *Refactoring–Does It Improve Software Quality?* (Stroggylos, Konstantinos and Spinellis, Diomidis, 2007)
- *Refactoring trends across n versions of n java open source systems: an empirical study* (Deepak Advani and Youssef Hassoun and Steve Counsell, 2005)
- *Why don't people use refactoring tools* (Emerson Murphy-Hill and Andrew P. Black, 2007)
- *A comparison of software refactoring tools* (J Simmonds and T Mens, 2002)
- *Digging the Development Dust for Refactorings* (C Schofield and B Tansey and Z Xing and E Stroulia, 2006)
- *Are refactorings less errorprone than other changes* (P Weissgerber and S Diehl, 2006)
- *Roots of refactoring* (J Philipps and B Rumpe, 2001)
- *Program Refactoring, Program Synthesis, and Model-Driven Development* (D Batory, 2007)
- *Work experience versus refactoring to design patterns: a controlled experiment* (T H Ng and S C Cheung and W K Chan and Y T Yu, 2006)
- *A discussion of refactoring in research and practice* (B Du Bois and P Van Gorp and A Amsel and N Van Eetvelde and H Stenten and S Demeyer and T Mens, 2004)
- *Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study* (Zhenchang Xing and Stroulia, E., 2006)
- *A survey of software refactoring* (Tom Mens and Tom Tourwe, 2004)
- *Refactoring: emerging trends and open problems* (T Mens and A V Deursen, 2003)

Rétrospective jalon

- *Project retrospectives, and why they never happen* (Glass, R.L., 2002)

Réunion quotidienne

- *Daily scrums in a distributed environment* (Ganis, Matt and Surdek, Steffan and Woodward, Elizabeth, 2009)

- *The effects of stand-up and sit-down meeting formats on meeting outcomes* (Bluedorn, A C and Turban, D B and Love, M S, 1999)
- *Supporting daily scrum meetings with change structure* (Rubart, Jessica and Freykamp, Frank, 2009)
- *'Today' messages: lightweight group awareness via email* (Brush, A. J. and Borning, Alan, 2003)
- *Introduction to Stand-up Meetings in Agile Methods* (Eisha Hasnain and Tracy Hall, 2009)
- *Stand and Deliver: Why I Hate Stand-Up Meetings* (Laplante, Phillip A, 2003)

Tableau des tâches

- *Drifting Toward Invisibility: The Transition to the Electronic Task Board* (Perry, Thomas, 2008)
- *Design and technology for Collaborage: collaborative collages of information on physical walls* (Moran, Thomas P. and Saund, Eric and Van Melle, William and Gujar, Anuj U. and Fishkin, Kenneth P. and Harrison, Beverly L., 1999)

Tableau kanban

- *Exploring the Sources of Waste in Kanban Software Development Projects* (Ikonen, Marko and Kettunen, Petri and Oza, Nilay and Abrahamsson, Pekka, 2010)

Test exploratoire

- *Defect Detection Efficiency: Test Case Based vs. Exploratory Testing* (Itkonen, Juha and Mantyla, Mika V. and Lassenius, Casper, 2007)

Tests fonctionnels automatisés

- *"Talking tests": a Preliminary Experimental Study on Fit User Acceptance Tests* (Marco Torchiano and Filippo Ricca and Massimiliano Di Penta, 2007)
- *Automated Acceptance Testing: A Literature Review and an Industrial Case Study* (Haugset, B. and Hanssen, G.K., 2008)

Tests unitaires automatisés

- *A survey of software testing practices in Alberta* (Geras, A.M. and Smith, M.R. and Miller, J., 2004)
- *A replicated survey of software testing practices in the Canadian province of Alberta: What has changed from 2004 to 2009?* (Garousi, Vahid and Varma, Tan, 2010)
- *Automatic Software Test Drivers* (Panzl, D. J., 1978)
- *Automatic revision of formal test procedures* (Panzl, David J., 1978)
- *On the effectiveness of unit test automation at Microsoft* (Williams, Laurie and Kudrjavets, Gunnar and Nagappan, Nachiappan, 2009)

User stories

- *Supporting Program Comprehension in Agile with Links to User Stories* (Sukanya Ratanotayanon and Susan Elliott Sim and Rosalva Gallardo-Valencia, 2009)
- *Sizing user stories using paired comparisons* (Miranda, Eduardo and Bourque, Pierre and Abran, Alain, 2009)
- *Generating user stories in groups* (Nguyen, Cuong D. and Gallagher, Erin and Read, Aaron and De Vreede, Gert-Jan, 2009)
- *INVEST in Good Stories, and SMART Tasks* (William Wake, 2003)

Version 1.0 du 04/05/2011