

# SCRUM

**Le guide pratique  
de la méthode agile la plus populaire**

**Claude Aubry**

*Consultant indépendant,  
professeur associé à l'université Paul Sabatier de Toulouse*

*Préface de  
François Beauregard*

DUNOD

Toutes les marques citées dans cet ouvrage sont des marques déposées par leurs propriétaires respectifs.

Illustration de couverture :  
© Anatoliy Zavodskov - Fotolia.com

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocollage.

Le Code de la propriété intellectuelle du 1<sup>er</sup> juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements

d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée. Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).



© Dunod, Paris, 2010  
ISBN 978-2-10-054833-0

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2<sup>e</sup> et 3<sup>e</sup> alinéa, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

# Préface

J'ai rencontré Claude une première fois lors d'une formation ScrumMaster que j'ai donnée à Paris en 2005. J'ai immédiatement remarqué Claude dans le groupe par son enthousiasme et sa volonté de comprendre les valeurs et principes qui sont les fondements de Scrum. Depuis Claude ne cesse de me surprendre par son engagement à défier l'ordre établi et par sa générosité dans son travail.

Je suis personnellement fortement engagé dans la communauté Agile et plus spécifiquement la communauté Scrum depuis 2001 car j'ai la ferme conviction que c'est à travers des gens qui ont intégré les valeurs et principes fondamentaux de Scrum et qui les portent chaque jour dans leur travail que nous arriverons à créer des organisations de développement logiciel où les résultats, la qualité de vie et le plaisir pourront coexister de façon durable.

Je fais particulièrement attention à distinguer les gens de l'approche proprement dite car en ces temps où le rythme d'adoption des approches agiles et en particulier Scrum est ultra-accéléré, et où de plus en plus de gens voient Scrum comme un outil qui va magiquement régler beaucoup de leurs difficultés, il est fondamental de communiquer sur les principes fondamentaux et les enjeux culturels liés à son adoption. Si nous pouvions observer toutes les organisations qui ont du succès avec Scrum nous trouverions invariablement des individus qui osent défier l'ordre établi avec ténacité, qui savent se mettre au service de l'autre, se doter d'une grande capacité d'écoute et qui savent guider un groupe vers sa mission. De vrais ScrumMasters ! Claude est l'un d'entre eux !

Vous aurez deviné que j'ai été ravi lorsque Claude m'a demandé d'écrire la préface de son livre sur Scrum. Pourquoi ? Tout simplement parce que c'est Claude ! Aussi parce que je me suis dit enfin un bouquin sur Scrum en français. Il y a un manque flagrant de titres en français dans le domaine de l'informatique et ça m'a toujours un peu gêné. Pourquoi nous francophones serions-nous moins capables d'écrire ? Pourquoi se contenter de traductions ?

Claude nous offre un ouvrage en français d'une grande qualité. Il nous démontre à travers le texte son talent de vulgarisateur. Dans un style très accessible mais sans compromis, il nous amène à découvrir Scrum et à comprendre comment nous pouvons l'appliquer dans nos organisations.

Merci Claude et bonne lecture.

22 septembre 2009 dans un vol Montréal-Paris

François Beauregard

Fondateur de Pyxis Technologies ([www.pyxis-tech.com](http://www.pyxis-tech.com))  
et de Agile Montréal, formateur Scrum certifié depuis 2004.

# Table des matières

Préface .....	III
Avant-propos .....	XVII
Chapitre 1 – Scrum sous la bannière de l'agilité .....	1
1.1 Le mouvement agile .....	1
1.1.1 Méthode agile .....	1
1.1.2 Manifeste agile .....	2
1.1.3 L'agilité .....	3
1.1.4 Pratiques agiles .....	4
1.1.5 Des méthodes agiles à Scrum .....	5
1.2 Survol de Scrum .....	5
1.2.1 Théorie .....	6
1.2.2 Éléments .....	7
Chapitre 2 – Des sprints pour une release .....	9
2.1 L'approche itérative et incrémentale .....	11
2.1.1 Incrémentation et itération .....	11
2.1.2 Bloc de temps .....	13
2.1.3 Durée du sprint .....	14
2.2 Cycle de développement Scrum .....	15
2.2.1 L'aspect temporel .....	15

2.2.2	<i>Activités et cycle de développement</i>	16
2.2.3	<i>Le résultat d'un sprint</i>	18
2.2.4	<i>Le résultat d'une release</i>	19
2.3	Guides pour les sprints et releases	19
2.3.1	<i>Démarrer le premier sprint au bon moment</i>	20
2.3.2	<i>Produire des micro-incréments</i>	21
2.3.3	<i>Enchaîner les sprints</i>	21
2.3.4	<i>Utiliser le produit partiel</i>	22
2.3.5	<i>Savoir finir la release</i>	23
<b>Chapitre 3 – Le Product Owner</b>		25
3.1	Responsabilités	28
3.1.1	<i>Fournir une vision partagée du produit</i>	28
3.1.2	<i>Définir le contenu du produit</i>	29
3.1.3	<i>Planifier la vie du produit</i>	29
3.2	Compétences souhaitées	29
3.2.1	<i>Bonne connaissance du domaine métier</i>	30
3.2.2	<i>Maîtrise des techniques de définition de produit</i>	30
3.2.3	<i>Capacité à prendre des décisions rapidement</i>	31
3.2.4	<i>Capacité à détailler au bon moment</i>	31
3.2.5	<i>Esprit ouvert au changement</i>	31
3.2.6	<i>Aptitude à la négociation</i>	32
3.3	Choisir le Product Owner d'une équipe	32
3.3.1	<i>Une personne disponible</i>	33
3.3.2	<i>Une seule personne</i>	34
3.3.3	<i>Où le trouver dans l'organisation actuelle ?</i>	36
3.3.4	<i>Une personne motivée pour le rôle</i>	36
3.4	Conseils pour progresser dans le rôle	36
<b>Chapitre 4 – Le ScrumMaster et l'équipe</b>		41
4.1	Responsabilités	42
4.1.1	<i>Responsabilités du ScrumMaster</i>	42
4.1.2	<i>Responsabilités de l'équipe</i>	44

4.2 Compétences souhaitées du ScrumMaster .....	44
4.2.1 Bonne connaissance de Scrum .....	45
4.2.2 Aptitude à comprendre les aspects techniques .....	45
4.2.3 Facilité à communiquer .....	45
4.2.4 Capacité à guider .....	46
4.2.5 Talent de médiateur .....	46
4.2.6 Ténacité .....	46
4.2.7 Inclination à la transparence .....	47
4.2.8 Goût à être au service de l'équipe .....	47
4.3 Choisir le ScrumMaster d'une équipe .....	47
4.3.1 Affectation au rôle .....	47
4.3.2 Où trouver la bonne personne ? .....	48
4.3.3 Quelqu'un qui incarne le changement .....	49
4.3.4 ScrumMaster, un état d'esprit .....	49
4.3.5 Rotation dans le rôle .....	50
4.4 Conseils pour progresser dans le rôle .....	50
<b>Chapitre 5 – Le backlog de produit .....</b>	<b>55</b>
5.1 Le backlog, la liste unique des stories .....	56
5.1.1 Utilisateurs du backlog .....	58
5.1.2 Vie du backlog .....	58
5.1.3 Options de représentation du backlog .....	60
5.2 La notion de priorité dans le backlog .....	60
5.2.1 Le sens de la priorité .....	60
5.2.2 Les critères pour définir la priorité .....	60
5.2.3 La gestion des priorités dans le backlog .....	62
5.3 Un élément du backlog .....	62
5.3.1 Attributs .....	62
5.3.2 Types .....	63
5.3.3 Cycle de vie d'un élément .....	63
5.3.4 Taille des éléments .....	64
5.4 Guides d'utilisation du backlog .....	65
5.4.1 Partager le backlog avec toute l'équipe .....	65
5.4.2 Bichonner le backlog .....	65

5.4.3 Surveiller la taille du backlog .....	66
5.4.4 Éviter d'avoir plusieurs backlogs pour une seule équipe .....	67
<b>Chapitre 6 – La planification de la release .....</b>	<b>69</b>
6.1 Planifier la release .....	70
6.1.1 Planifier pour prévoir .....	70
6.1.2 Réunion ou processus ? .....	70
6.1.3 La participation de l'équipe est requise .....	71
6.1.4 La release est planifiée à partir du backlog .....	71
6.1.5 Place dans le cycle de vie .....	72
6.2 Étapes .....	73
6.2.1 Définir le critère de fin de la release .....	73
6.2.2 Estimer les stories du backlog .....	75
6.2.3 Définir la durée des sprints .....	78
6.2.4 Estimer la capacité de l'équipe .....	80
6.2.5 Produire le plan de release .....	81
6.3 Résultats .....	82
6.3.1 Le plan de release .....	82
6.3.2 Burndown chart de release .....	83
6.4 Guides pour la planification de release .....	86
6.4.1 S'adapter au calendrier .....	86
6.4.2 Ne pas confondre valeur et coût, ni vitesse et productivité .....	87
6.4.3 Garder du mou pour les incertitudes .....	87
6.4.4 Provisionner pour le feedback ultérieur .....	88
<b>Chapitre 7 – La réunion de planification de sprint .....</b>	<b>89</b>
7.1 Planifier le sprint .....	90
7.1.1 C'est l'équipe qui planifie .....	90
7.1.2 Espace de travail ouvert .....	91
7.1.3 Durée de la réunion .....	92
7.2 Étapes .....	93
7.2.1 Rappeler le contexte du sprint .....	93
7.2.2 Évaluer le périmètre potentiel .....	94
7.2.3 Définir le but du sprint .....	95

7.2.4	<i>Identifier les tâches</i> .....	95
7.2.5	<i>Estimer les tâches</i> .....	96
7.2.6	<i>Prendre des tâches</i> .....	97
7.2.7	<i>S'engager collectivement</i> .....	98
7.3	Résultats .....	98
7.3.1	<i>Plan de sprint initial</i> .....	98
7.3.2	<i>Backlog et burndown charts actualisés</i> .....	99
7.4	Guides pour la planification de sprint .....	100
7.4.1	<i>Préparer le backlog de produit en anticipation</i> .....	100
7.4.2	<i>Laisser l'équipe décider du périmètre</i> .....	101
7.4.3	<i>Laisser l'équipe identifier les tâches</i> .....	101
7.4.4	<i>Décomposer en tâches courtes</i> .....	101
7.4.5	<i>Prendre un engagement raisonnable</i> .....	102
7.4.6	<i>Garder du mou dans le plan de sprint</i> .....	102
7.4.7	<i>Faire de la conception</i> .....	103
<b>Chapitre 8 – Le scrum quotidien</b> .....		105
8.1	Une réunion quotidienne .....	106
8.1.1	<i>Le sprint appartient à l'équipe</i> .....	106
8.1.2	<i>Un cérémonial balisé</i> .....	107
8.2	Étapes .....	108
8.2.1	<i>Se réunir</i> .....	108
8.2.2	<i>Répondre aux trois questions</i> .....	109
8.2.3	<i>Statuer sur l'atteinte des objectifs</i> .....	110
8.3	Résultats .....	111
8.3.1	<i>Le plan de sprint actualisé</i> .....	111
8.3.2	<i>Le burndown chart de sprint actualisé</i> .....	111
8.3.3	<i>La liste des obstacles</i> .....	112
8.4	Guides pour le scrum .....	114
8.4.1	<i>S'en tenir à un quart d'heure</i> .....	114
8.4.2	<i>Ne s'intéresser qu'au reste à faire, pas au temps passé</i> .....	114
8.4.3	<i>Faire le suivi des tâches avec les états plutôt que les heures</i> .....	115
8.4.4	<i>Veiller à finir les stories</i> .....	116
8.4.5	<i>Organiser des variations dans le déroulement du scrum</i> .....	117

<b>Chapitre 9 – La revue de sprint .....</b>	119
9.1 La revue est basée sur une démonstration .....	119
9.1.1 <i>La revue accueille de nombreux invités .....</i>	120
9.1.2 <i>Durée de la réunion .....</i>	120
9.1.3 <i>La revue montre le produit.....</i>	121
9.2 Étapes.....	121
9.2.1 <i>Préparer la démonstration .....</i>	121
9.2.2 <i>Rappeler les objectifs du sprint .....</i>	122
9.2.3 <i>Effectuer la démonstration.....</i>	122
9.2.4 <i>Calculer la vitesse .....</i>	123
9.2.5 <i>Ajuster le plan de release .....</i>	123
9.3 Résultats .....	124
9.3.1 <i>Backlog de produit actualisé.....</i>	124
9.3.2 <i>Plan de release et burndown chart de release mis à jour .....</i>	124
9.4 Guides pour la revue.....	125
9.4.1 <i>Dérouler un scénario .....</i>	125
9.4.2 <i>Inviter largement, mais expliquer que c'est un produit partiel .....</i>	126
9.4.3 <i>Éviter de modifier le produit partiel au dernier moment .....</i>	126
9.4.4 <i>Parler des stories, pas des tâches .....</i>	127
9.4.5 <i>Solliciter le feedback .....</i>	127
9.4.6 <i>En faire la réunion essentielle sur le produit .....</i>	127
<b>Chapitre 10 – La rétrospective de sprint.....</b>	129
10.1 Une pratique d'amélioration continue .....	130
10.1.1 <i>Un moment de réflexion collective à la fin de chaque sprint .....</i>	131
10.1.2 <i>C'est l'équipe qui refait le match .....</i>	131
10.2 Étapes.....	132
10.2.1 <i>Créer un environnement propice à l'expression .....</i>	132
10.2.2 <i>Collecter les informations sur le sprint passé .....</i>	133
10.2.3 <i>Identifier des idées d'amélioration .....</i>	133
10.2.4 <i>Regrouper les idées .....</i>	133
10.2.5 <i>Définir l'amélioration prioritaire .....</i>	134
10.2.6 <i>Adapter Scrum pour le prochain sprint .....</i>	134
10.3 Résultat .....	135

10.4 Guides .....	135
10.4.1 <i>Ne pas en faire une séance de règlement de comptes</i> .....	135
10.4.2 <i>Parler de ce qui va bien</i> .....	136
10.4.3 <i>Faire aboutir les actions des rétrospectives précédentes</i> .....	136
10.4.4 <i>Se concentrer sur une amélioration</i> .....	137
10.4.5 <i>Mener des rétrospectives plus poussées en fin de release</i> .....	137
10.4.6 <i>Utiliser un facilitateur externe</i> .....	137
<b>Chapitre 11 – La signification de fini</b> .....	139
11.1 Fini, une pratique à part entière .....	139
11.1.1 <i>Impact du mal fini</i> .....	140
11.1.2 <i>Intérêt d'avoir une signification de fini partagée</i> .....	141
11.2 Étapes .....	142
11.2.1 <i>Définir la signification de fini</i> .....	142
11.2.2 <i>Publier la liste des contrôles</i> .....	143
11.2.3 <i>Utiliser la définition de fini</i> .....	144
11.3 Contenu de fini .....	144
11.3.1 <i>Fini pour une story</i> .....	145
11.3.2 <i>Fini pour un sprint</i> .....	146
11.3.3 <i>Fini pour une release</i> .....	147
11.4 Guides pour une bonne pratique de fini .....	147
11.4.1 <i>Faire des tranches verticales</i> .....	147
11.4.2 <i>Adapter à partir d'une définition générique de fini</i> .....	148
11.4.3 <i>Faire évoluer la signification de fini</i> .....	149
11.4.4 <i>Appliquer la pratique avec beaucoup de volonté</i> .....	149
<b>Chapitre 12 – Adapter Scrum au contexte</b> .....	151
12.1 Pratiques agiles .....	152
12.1.1 <i>Pratiques Scrum</i> .....	152
12.1.2 <i>Pratiques complémentaires</i> .....	153
12.2 Risques dans la mise en œuvre de Scrum .....	154
12.3 Définir le contexte .....	155
12.3.1 <i>Influence de l'organisation</i> .....	156
12.3.2 <i>Contexte du projet</i> .....	157

12.4 Impact du contexte sur les pratiques .....	158
12.4.1 <i>Impact par attribut</i> .....	158
12.4.2 <i>Situation d'un projet par rapport à l'agilité</i> .....	162
<b>Chapitre 13 – De la vision aux stories.....</b>	<b>165</b>
13.1 Le produit a une vie avant les sprints .....	165
13.2 Construire une bonne vision .....	166
13.2.1 <i>Techniques pour la vision</i> .....	166
13.2.2 <i>Qualités d'une bonne vision</i> .....	167
13.2.3 <i>Une vision par release</i> .....	167
13.3 Features .....	168
13.3.1 <i>Justification du terme feature</i> .....	168
13.3.2 <i>Identification des features</i> .....	169
13.3.3 <i>Attributs d'une feature</i> .....	170
13.3.4 <i>Features et backlog</i> .....	170
13.3.5 <i>Features et priorité</i> .....	170
13.4 Rôles d'utilisateurs .....	171
13.4.1 <i>Intérêt des rôles</i> .....	171
13.4.2 <i>Identification des rôles</i> .....	172
13.4.3 <i>Attributs d'un rôle</i> .....	172
13.4.4 <i>Personas</i> .....	172
13.5 User Stories .....	173
13.5.1 <i>Définition</i> .....	173
13.5.2 <i>Identifier les stories</i> .....	174
13.5.3 <i>Attributs d'une story</i> .....	174
13.5.4 <i>Techniques de décomposition des stories</i> .....	175
13.5.5 <i>Différence avec les use-cases</i> .....	176
13.6 Améliorer l'ingénierie des exigences .....	176
13.6.1 <i>Exigences et spécifications</i> .....	176
13.6.2 <i>Traçabilité</i> .....	176
13.6.3 <i>Exigences non fonctionnelles</i> .....	177
13.6.4 <i>Avec quelle équipe ?</i> .....	179

<b>Chapitre 14 – De la story aux tests d’acceptation .....</b>	181
14.1 Test d’acceptation.....	182
14.2 Étapes.....	183
14.2.1 Définir les conditions de satisfaction.....	183
14.2.2 Écrire les storytests .....	184
14.2.3 Développer la story .....	186
14.2.4 Passer les storytests .....	186
14.3 Guides pour le test d’acceptation .....	188
14.3.1 Se servir des tests pour communiquer .....	188
14.3.2 Tester une story dans le sprint où elle est développée .....	188
14.3.3 Ne pas stocker les bugs .....	189
14.3.4 Connecter les tests d’acceptation.....	189
14.3.5 Planifier le travail de test .....	190
<b>Chapitre 15 – Estimations, mesures et indicateurs .....</b>	191
15.1 Estimer la taille et l’utilité.....	192
15.1.1 Estimation de la taille des stories en points.....	192
15.1.2 Estimation de la valeur ou de l’utilité .....	193
15.2 Collecter les mesures .....	196
15.2.1 Mesures quotidiennes .....	196
15.2.2 Mesures à chaque sprint.....	196
15.2.3 Mesures à chaque release .....	197
15.2.4 Autres mesures possibles .....	197
15.3 Utiliser les indicateurs .....	197
15.3.1 Indicateurs pour le suivi du sprint .....	197
15.3.2 Indicateurs pour le suivi du produit .....	198
15.3.3 Indicateurs pour le suivi de la release .....	205
15.4 Guides pour estimer, mesurer et publier les indicateurs .....	206
15.4.1 Une estimation n’est pas un engagement .....	206
15.4.2 Pas de mesure du temps consommé .....	207
15.4.3 Collecter les mesures dès le début d’un développement .....	208
15.4.4 Considérer un outil pour la collecte .....	208
15.4.5 Expliquer les indicateurs.....	208

<b>Chapitre 16 – Scrum et l'ingénierie du logiciel .....</b>	211
16.1 Pratiques autour du code .....	212
16.1.1 <i>Intégration continue</i> .....	212
16.1.2 <i>Pilotage par les tests</i> .....	213
16.1.3 <i>Programmation en binôme</i> .....	214
16.2 Pratiques de conception .....	215
16.2.1 <i>Architecture évolutive</i> .....	216
16.2.2 <i>Conception émergente</i> .....	216
16.3 Maintenance .....	217
16.3.1 <i>Il n'y a pas de phase de maintenance</i> .....	217
16.3.2 <i>Gestion des bugs</i> .....	217
<b>Chapitre 17 – Scrum avec un outil .....</b>	221
17.1 Les outils Scrum .....	221
17.1.1 <i>Les outils non informatiques</i> .....	221
17.1.2 <i>Les tableurs ou assimilés</i> .....	222
17.1.3 <i>Les outils spécifiques</i> .....	223
17.2 Un exemple avec IceScrum .....	224
17.2.1 <i>Les rôles Scrum</i> .....	224
17.2.2 <i>Démarrage d'une release</i> .....	225
17.2.3 <i>Déroulement des sprints</i> .....	235
17.2.4 <i>Les tests d'acceptation</i> .....	237
17.2.5 <i>Mesures et indicateurs</i> .....	238
<b>Chapitre 18 – La transition à Scrum .....</b>	241
18.1 Le processus de transition .....	241
18.1.1 <i>Avec qui faire la transition ?</i> .....	242
18.1.2 <i>Cycle de transition</i> .....	242
18.1.3 <i>Backlog d'amélioration des pratiques</i> .....	243
18.1.4 <i>Obstacles d'organisation</i> .....	243
18.2 Étapes du processus de transition .....	243
18.2.1 <i>Évaluer le contexte</i> .....	243
18.2.2 <i>Préparer l'application de Scrum</i> .....	244
18.2.3 <i>Exécuter Scrum sur un projet pilote</i> .....	246

18.2.4 Diffuser dans l'organisation .....	247
18.2.5 Évaluer le niveau atteint .....	249
18.3 Impacts sur l'organisation .....	252
18.3.1 L'évaluation individuelle est contre-productive .....	252
18.3.2 Pas de multitâches .....	252
18.3.3 Spécialistes vs généralistes .....	253
18.3.4 Cohabitation avec d'autres processus .....	253
<b>Chapitre 19 – Scrum en France .....</b>	<b>255</b>
19.1 Scrum à la française .....	255
19.1.1 Utilisateurs de Scrum .....	255
19.1.2 Retours d'expérience .....	256
19.1.3 Domaines .....	256
19.1.4 Des particularités locales ? .....	258
19.2 Des freins à la diffusion ? .....	258
19.2.1 MOA et MOE ne sont pas agiles .....	258
19.2.2 Contrats au forfait, le mythe du périmètre fixé .....	260
19.3 Le french flair pour Scrum .....	262
<b>Références bibliographiques .....</b>	<b>264</b>
<b>Index .....</b>	<b>265</b>



# Avant-propos

Depuis plus d'une dizaine d'années, je conseille des entreprises et je forme des étudiants sur les méthodes itératives et agiles. Depuis cinq ans, cet effort porte presque exclusivement sur Scrum ; cela m'a permis de participer à une cinquantaine de projets menés avec Scrum et de m'impliquer fortement dans le développement du logiciel libre IceScrum (un outil pour Scrum<sup>1</sup>).

Sur le terrain, j'ai constaté ce qui fonctionnait bien et ce qui fonctionnait moins bien. À travers ce livre, je souhaite vous faire partager mon expérience et les leçons apprises.

Vous y apprendrez à appliquer les pratiques de Scrum en les adaptant aux contraintes de votre environnement.

Même si ce livre ne remplace pas une formation et encore moins une application concrète, il présente des conseils, des exemples, des retours d'expérience et des guides qui vous permettront d'optimiser votre mise en œuvre de Scrum.

Ce livre est destiné à tous ceux qui s'intéressent à Scrum. Les novices y trouveront une présentation détaillée des pratiques, ceux qui en ont déjà une connaissance trouveront des conseils utiles.

Il intéressera tous les membres des équipes (pas seulement les ScrumMasters) ayant adopté Scrum ou étant sur le point de le faire, y compris les managers et les clients qui souhaitent se familiariser avec cette technique et son jargon.

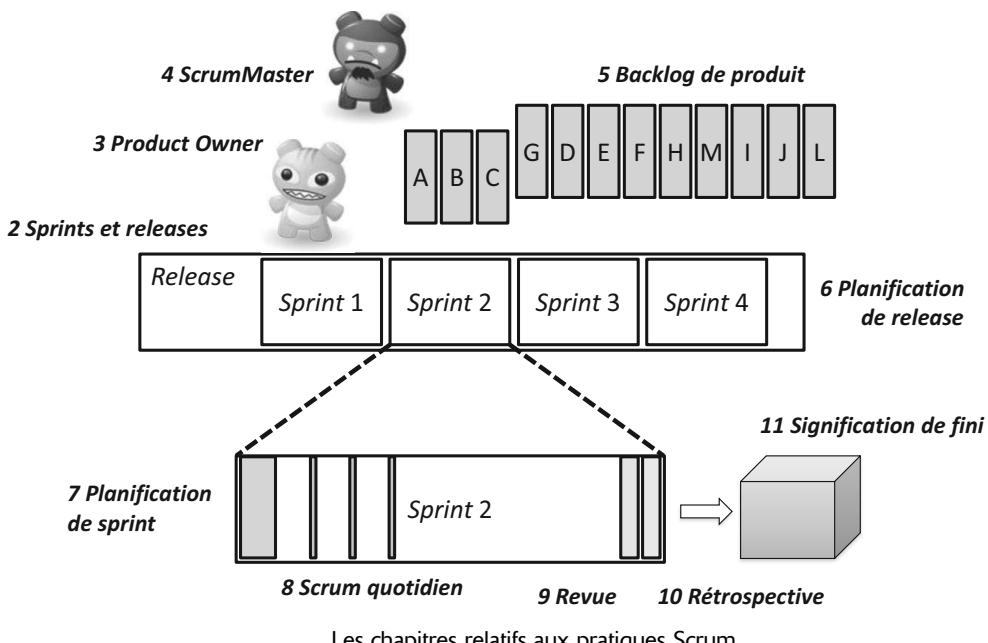
## *Parcours de lecture : combien de sprints vous faut-il ?*

Si vous cherchez une introduction brève aux méthodes agiles et à Scrum, lisez les chapitres 1 et 2.

Si vous voulez connaître Scrum en détail, lisez les chapitres 1 à 11. Les personnes jouant le rôle de Product Owner liront en particulier le chapitre 3 et le chapitre 5, et les ScrumMasters le chapitre 4. Tous les membres de l'équipe appliquant Scrum seront intéressés par les chapitres 4 à 11.

---

1. Voir chapitre 17.



À partir du chapitre 12 jusqu'au chapitre 16, l'accent est mis sur les compléments à Scrum pour le développement d'un produit : le chapitre 12 présente une approche pour utiliser Scrum en tenant compte des contraintes des projets, les chapitres 13 et 14 sont plutôt destinés à ceux qui définissent le produit, les chapitres 15 et 16 aux développeurs.

Le chapitre 17 présente l'outillage pour Scrum, le chapitre 18 propose des pistes pour effectuer la transition dans de grandes organisations et le chapitre 19 aborde la diffusion de Scrum en France.

### Compléments en ligne

Sur le site [www.aubryconseil.com](http://www.aubryconseil.com), vous trouverez les dernières informations relatives au livre, des articles complémentaires, des précisions, des mises à jour, ainsi que les formations et les conférences de l'auteur.

### Remerciements

Mes relecteurs m'ont fourni un *feedback* précieux, en m'obligeant à repenser certaines parties et en m'aidant à les rendre plus accessibles. Je les remercie chaleureusement pour leur contribution ; il s'agit d'Alexandre BOUTIN, Thierry CROS, et Antoine VERNOIS.

Je remercie également Laure AUBRY, Jean-Pierre ODILE et Julien AUBRY qui se sont investis dans la révision du manuscrit et m'ont été très précieux par leurs commentaires.

Jean-Claude GROSJEAN et Philippe KRUCHTEN ont participé chacun à la rédaction d'un chapitre et à sa relecture, je leur en suis très reconnaissant.

Je suis également reconnaissant à François BEAUREGARD d'avoir relu ce livre et d'y avoir contribué en écrivant la préface.

Un grand merci à Patrice COURTIADE, l'auteur des dessins qui apportent une touche de légèreté à un sujet forcément sérieux.

Je remercie mon éditeur Jean-Luc BLANC de m'avoir fait confiance.

Je remercie également Véronique MESSAGER ROTA et Pascal ROQUES, deux auteurs, pour les conseils qu'ils m'ont donnés sur la rédaction d'un livre, ainsi que toutes les personnes que j'ai rencontrées lors de mes formations et interventions sur les projets.

Merci enfin à Ruth pour son soutien sans faille au cours des nombreuses journées, soirées et week-ends que j'ai passés à écrire et réécrire ce livre.



# 1

# Scrum sous la bannière de l'agilité

## 1.1 LE MOUVEMENT AGILE

### 1.1.1 Méthode agile

Les **méthodes agiles** représentent un mouvement novateur qui vise à apporter plus de valeur aux clients et aux utilisateurs, ainsi qu'une plus grande satisfaction dans leur travail aux membres de l'équipe.

Le but affiché d'une méthode agile est de maximiser la valeur ajoutée : le développement s'effectuant par itérations successives, il est possible, à la fin de chaque itération, de changer les priorités en faisant en sorte que les éléments apportant le plus de valeur soient réalisés en premier.

Un meilleur accomplissement des personnes impliquées dans un développement est rendu possible par la notion d'**équipe auto-organisée**.

Une tentative de définition, adaptée de Scott Ambler<sup>1</sup>, pourrait être : « *Une méthode agile est une approche itérative et incrémentale pour le développement de logiciel, réalisé de manière très collaborative par des équipes responsabilisées, appliquant un cérémonial minimal, qui produisent un logiciel de grande qualité dans un délai contraint, répondant aux besoins changeants des utilisateurs.* »

Le cérémonial c'est ce qui définit les règles sociales et conventionnelles régissant la vie d'une équipe ; s'il est vrai que, pour une méthode agile, il est minimal pour

---

1. <http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm>

la documentation, il existe bien pour le côté social (on parle de *cérémonial Scrum* à propos des réunions), mais avec des règles nouvelles.

### 1.1.2 Manifeste agile

Le terme agile est apparu dans le domaine du logiciel en 2001 avec le *Manifeste agile*<sup>1</sup>. Le mouvement a pris de l'ampleur depuis quelques années, il est maintenant diffusé, au-delà des pionniers, dans de très nombreuses organisations impliquées dans le développement de logiciel.

Le *Manifeste* fédère le mouvement agile avec un ensemble de valeurs et de principes.

- **Valeurs du Manifeste agile** – Publié au début des années 2000, le *Manifeste agile* définit une attitude de réaction par rapport à des processus lourds et bureaucratiques, en vogue à l'époque (et parfois encore aujourd'hui). La position prise par rapport à ces processus ne définit pas les valeurs intrinsèques de l'agilité, mais des valeurs relatives :

- Les personnes et leurs interactions sont plus importantes que les processus et les outils.
- Un logiciel qui fonctionne prime sur de la documentation.
- La collaboration est plus importante que le suivi d'un contrat.
- La réponse au changement passe avant le suivi d'un plan.

Avec ces préceptes, pleins de bon sens, le *Manifeste agile* représente un coup de balancier, comme on en voit régulièrement dans l'industrie du logiciel, pour promouvoir des processus plus légers.

- **Les principes du Manifeste agile** – Le *Manifeste* énonce douze principes :
  - Satisfaire le client en livrant tôt et régulièrement des logiciels utiles, qui offrent une véritable valeur ajoutée.
  - Accepter les changements, même tard dans le développement.
  - Livrer fréquemment une application qui fonctionne.
  - Collaborer quotidiennement entre clients et développeurs.
  - Bâtir le projet autour de personnes motivées en leur fournissant environnement et support, et en leur faisant confiance.
  - Communiquer par des conversations en face à face.
  - Mesurer la progression avec le logiciel qui fonctionne.
  - Garder un rythme de travail durable.
  - Rechercher l'excellence technique et la qualité de la conception.
  - Laisser l'équipe s'auto-organiser.
  - Rechercher la simplicité.
  - À intervalles réguliers, réfléchir aux moyens de devenir plus efficace.

---

1. <http://www.agilemanifesto.org>

Cette liste, pas plus que le *Manifeste*, ne définit une méthode agile. Il n'y a d'ailleurs pas une seule méthode, ni un emploi qu'on pourrait qualifier d'orthodoxe. Si les valeurs et les principes sont universels, la façon de les mettre en œuvre sur des projets varie. Cette application se fait par l'intermédiaire de ce qu'on appelle les *pratiques*.

Les pratiques agiles, qui ne sont pas évoquées dans le *Manifeste*, constituent la partie essentielle de ce livre.

### 1.1.3 L'agilité

En fédérant les méthodes agiles, le *Manifeste agile* constitue l'acte de naissance d'un nouveau mouvement, l'agilité.

Jim Highsmith<sup>1</sup> définit l'agilité par rapport au changement :

- « *L'agilité est la capacité à favoriser le changement et à y répondre en vue de s'adapter au mieux à un environnement turbulent.* »
- *Finalement, l'agilité permet d'embrasser le changement plutôt que de lui résister.*
- *Dans notre époque de l'information, l'avantage compétitif vient de la vitesse et de la flexibilité.* »

L'agilité permet donc de s'adapter plus vite au changement. Cependant, tous les environnements des organisations ne sont pas « turbulents » : en tout cas, il y en a qui sont moins soumis aux changements que d'autres, qui ne sont pas dans un milieu concurrentiel. Cela ne signifie pas que l'agilité n'est pas nécessaire à ces projets et ces organisations, mais que la façon de l'appliquer doit être adaptée à leur contexte (voir le chapitre 12).

#### Une nouvelle culture

Avec ses valeurs et ses principes, on peut considérer l'agilité comme une nouvelle culture du développement.

Les valeurs de l'agilité peuvent présenter un caractère indéniablement subversif pour certaines organisations, mais on sait que les valeurs sont assez vite récupérées. Au-delà des idées, l'agilité, en particulier avec Scrum, véhicule un vocabulaire nouveau. En quelque sorte, le vocabulaire contribue à renforcer l'idée du changement de culture. Il importe de tenir compte des aspects culturels dans la formation et la transition à l'agilité : on ne change pas facilement de culture.

#### Place de l'agilité

Pour illustrer la position de l'agilité dans le développement de logiciel, je reprends une phrase de Tom de Marco<sup>2</sup>, qu'on ne peut pas suspecter d'être un zélateur de l'agilité :

1. <http://www.jimhighsmith.com/>

2. De Marco, cité par Highsmith, est un expert du génie logiciel connu depuis plus de 25 ans : [http://en.wikipedia.org/wiki/Tom\\_DeMarco](http://en.wikipedia.org/wiki/Tom_DeMarco).

« La formule du succès : agilité 1, n'importe quoi d'autre 0. »

Ce n'est pas une définition, c'est plutôt un constat, édifiant sur la place de l'agilité dans l'ingénierie du logiciel.

### 1.1.4 Pratiques agiles

Si la culture agile est nouvelle, des **pratiques** maintenant qualifiées **d'agiles** existaient avant, pour certaines avant le *Manifeste agile* et même avant les premières méthodes agiles.

Un certain nombre de pratiques sont reconnues depuis longtemps par la communauté des spécialistes du génie logiciel, par exemple :

- livrer fréquemment et régulièrement le logiciel,
- faire des cycles de développement courts et limités dans le temps,
- constituer une équipe complète pour un développement,
- gérer les membres de l'équipe en les responsabilisant,
- avoir le représentant des utilisateurs sur le même site que le reste de l'équipe,
- produire des plans à plusieurs niveaux : détaillés uniquement pour le court terme, et plus généraux pour le moyen terme,
- développer en intégrant le code de façon continue,
- faire des bilans de projet dans le but d'améliorer la façon de travailler.

D'autres sont apparues avec les méthodes agiles et sont devenues indiscutables, après avoir été éprouvées sur de nombreux projets :

- avoir un *backlog* de produit tenant compte des priorités,
- suivre l'avancement des projets par la tenue d'une réunion quotidienne,
- écrire les tests avant d'écrire le code,
- pratiquer, de temps en temps, le travail en binôme, technique qui consiste à avoir deux personnes derrière un seul écran pour partager les connaissances.

Prises individuellement, ces pratiques sont déjà efficaces. Insérées dans le cadre cohérent d'une approche agile, elles se renforcent mutuellement, et contribuent à la qualité du produit et à son utilité.

#### L'agilité oui, la pagaille non

Des utilisateurs, brimés depuis longtemps par leur direction des systèmes d'information (DSI), découvrent que l'agilité peut accueillir et même favoriser les changements, ce qui les amène à penser qu'ils peuvent tout changer tout le temps.

Des managers se disent qu'avec l'agilité, il leur sera plus facile de demander à leurs équipes de traiter une urgence par du travail supplémentaire non prévu.

Non ! L'agilité favorise le changement, mais ne le rend pas gratuit ni permanent. Si la demande de changement venue d'un utilisateur est la bienvenue, sa prise en compte passe par une gestion des priorités et elle est toujours différée : une équipe qui travaille ne doit pas être perturbée n'importe quand.

### 1.1.5 Des méthodes agiles à Scrum

Les méthodes agiles existaient avant le *Manifeste* : Scrum et *Extreme Programming* datent des années 1990. Le *Lean Software* repose sur des bases encore plus anciennes : le système de production dans les usines Toyota dans les années 1950.

Il y a eu de nombreuses autres méthodes qualifiées d'agiles ou de semi-agiles. Le *Manifeste* en énonçant les valeurs et les principes communs a contribué à les fédérer toutes derrière la même bannière de l'agilité.

Plus récemment, l'engouement pour Scrum (la *ScrumMania !*) a mis fin à une hypothétique rivalité entre les méthodes agiles. Les études d'opinion et les tendances des recherches sur le Web le montrent : Scrum est de loin la plus populaire dans la famille des méthodes agiles.

Maintenant que Scrum a gagné<sup>1</sup>, la difficulté majeure est d'amener ses utilisateurs à en faire un usage correct, sous la bannière de l'agilité.

## 1.2 SURVOL DE SCRUM

### Le nom vient du rugby

On prononce « screum » pas « scrume », ni « scroum ».

**Scrum** signifie mêlée au rugby. Scrum utilise les valeurs et l'esprit du rugby et les adapte aux projets de développement. Comme le *pack* lors d'un ballon porté au rugby, l'équipe chargée du développement travaille de façon collective, soudée vers un objectif précis. Comme un demi de mêlée, le ScrumMaster aiguillonne les membres de l'équipe, les repositionne dans la bonne direction et donne le tempo pour assurer la réussite du projet.

Au-delà de cet accent mis sur la puissance du collectif, Scrum est un processus agile qui attaque la complexité par une approche empirique.

### Scrum, un truc qui marche

On est naturellement tenté de parler de méthode agile ou de processus agile pour Scrum. En fait, la définition officielle, celle donnée par la Scrum Alliance<sup>2</sup> et son fondateur Ken Schwaber est légèrement différente. Scrum n'est présenté ni comme un processus ni comme une méthode.

Le plus souvent, Ken Schwaber décrit Scrum comme un cadre (*framework*) ; à d'autres occasions il en parle comme d'une voie à suivre (*path*) ou d'un outil et il revient à processus... Un spécialiste des processus parlerait, pour Scrum, de *pattern* de processus, orienté gestion de projet, qui peut incorporer différentes méthodes ou pratiques d'ingénierie.

---

1. À l'heure où j'écris ces lignes (septembre 2009), mais ça peut changer.

2. <http://www.scrumalliance.org>

Qu'on le désigne comme un cadre, un *pattern* de processus, une méthode, voire un truc, Scrum définit des éléments qui feront partie du processus appliqué pour développer un produit. Ces éléments sont en petit nombre, le cadre imposé par Scrum étant très léger : guère plus que des itérations, des réunions au début et à la fin de chacune, un *backlog* de produit et trois rôles.

Ce côté minimaliste, plus les succès sur le terrain, donnent à croire que Scrum est un truc qui marche.

### Scrum en bref

Si la vraie nature de Scrum est difficile à définir, il est beaucoup plus simple d'expliquer la mécanique de mise en œuvre :

- **Scrum** sert à développer des produits, généralement en quelques mois. Les fonctionnalités souhaitées sont collectées dans le **backlog de produit** et classées par priorité. C'est le **Product Owner** qui est responsable de la gestion de ce *backlog*.
- Une **version (release)** est produite par une série d'itérations d'un mois<sup>1</sup> appelées des **sprints**. Le contenu d'un *sprint* est défini par l'équipe, avec le Product Owner, en tenant compte des priorités et de la capacité de l'équipe. À partir de ce contenu, l'équipe identifie les tâches nécessaires et s'engage pour réaliser les fonctionnalités sélectionnées pour le *sprint*.
- Pendant un *sprint*, des points de contrôle sur le déroulement des tâches sont effectués lors des mêlées quotidiennes (*scrums*). Cela permet au **ScrumMaster**, l'animateur chargé de faire appliquer Scrum, de déterminer l'avancement par rapport aux engagements et d'appliquer, avec l'équipe, des ajustements pour assurer le succès du *sprint*.
- À la fin de chaque *sprint*, l'équipe obtient un **produit partiel** (un incrément) qui fonctionne. Cet incrément du produit est potentiellement livrable et son évaluation permet d'ajuster le *backlog* pour le *sprint* suivant.

#### 1.2.1 Théorie

Les premières expérimentations de Scrum datent de 1993 et le premier article<sup>2</sup> est paru en 1995, pour la conférence OOPSLA<sup>3</sup> ; signé de Ken Schwaber, il présente Scrum comme un processus empirique adapté aux développements de produits complexes.

Scrum a son origine dans la théorie de contrôle empirique des processus. Les trois piliers de la théorie sont la transparence, l'inspection et l'adaptation du processus dont Scrum fournit le cadre :

---

1. On peut remarquer que l'usage de Scrum évolue : par exemple, une pratique courante aujourd'hui est d'avoir des *sprints* de deux semaines alors que la durée initiale était un mois.

2. <http://jeffsutherland.com/oopsla/schwapub.pdf>

3. *Object-Oriented Programming, Systems, Languages & Applications*.

- **Transparence** – La transparence garantit que tous les indicateurs relatifs à l'état du développement sont visibles par tous ceux qui sont intéressés par le résultat du produit. Non seulement la transparence pousse à la visibilité mais ce qui est rendu visible doit être bien compris ; cela signifie que ce qui est vu est bien le reflet de la réalité. Par exemple, si un indicateur annonce que le produit est fini (ou une partie seulement du produit), cela doit être strictement équivalent à la signification de fini définie par l'équipe.
- **Inspection** – Les différentes facettes du développement doivent être inspectées suffisamment souvent pour que des variations excessives dans les indicateurs puissent être détectées à temps.
- **Adaptation** – Si l'inspection met en évidence que certains indicateurs sont en dehors des limites acceptables, il est probable que le produit résultant sera également inacceptable si on ne réagit pas ; le processus doit donc être ajusté rapidement pour minimiser les futures déviations.

Il y a trois points d'inspection et d'adaptation dans Scrum :

- **Le scrum quotidien** permet d'inspecter la progression par rapport au but du *sprint* et de faire des adaptations qui optimisent la valeur du travail du jour suivant.
- **La planification et la revue de sprint** sont utilisées pour inspecter l'avancement du développement par rapport au but de la *release* et faire des adaptations sur le contenu du produit pour le prochain *sprint*.
- **La rétrospective** inspecte la façon de travailler dans le *sprint* pour déterminer quelles améliorations du processus peuvent être faites dans le prochain *sprint*.

## 1.2.2 Éléments

Le cadre Scrum consiste en une équipe avec des rôles bien définis, des blocs de temps (*timeboxes*) et des artefacts (figure 1.1) :

Rôles	<i>Timeboxes</i>	Artefacts
<ul style="list-style-type: none"> <li>• Product Owner</li> <li>• ScrumMaster</li> <li>• Équipe</li> </ul>	<ul style="list-style-type: none"> <li>• Planification de <i>release</i></li> <li>• Planification de <i>sprint</i></li> <li>• Scrum quotidien</li> <li>• Revue de <i>sprint</i></li> <li>• Rétrospective</li> </ul>	<ul style="list-style-type: none"> <li>• <i>Backlog</i> de produit</li> <li>• Plan de <i>release</i></li> <li>• Plan de <i>sprint</i></li> <li>• <i>Burndown</i> de <i>sprint</i></li> <li>• <i>Burndown</i> de <i>release</i></li> </ul>

**Figure 1.1** – Éléments de Scrum

- **Équipe et rôles** – L'équipe a un rôle capital dans Scrum : elle est constituée avec le but d'optimiser la flexibilité et la productivité ; pour cela, elle s'organise elle-même et doit avoir toutes les compétences nécessaires au développement du produit. Elle est investie avec le pouvoir et l'autorité pour faire ce qu'elle a à faire.
- **Timeboxes** – Scrum utilise des blocs de temps pour créer de la régularité. Le cœur du rythme de Scrum est le *sprint*, une itération d'un mois ou moins. Dans chaque *sprint*, le cadre est donné par un cérémonial léger mais précis basé des réunions.
- **Artefacts** –Scrum exige peu d'artefacts lors du développement : le plus remarquable est le *backlog* de produit, pivot des différentes activités.

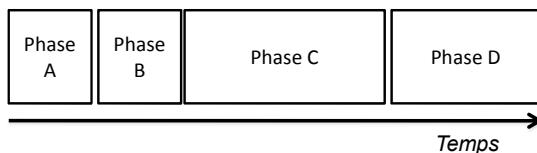
Quelques règles liant les éléments complètent ce cadre simple. Toutefois, derrière l'apparente simplicité de Scrum se cache une grande puissance pour mettre en évidence le degré d'efficacité des pratiques de développement utilisées.

# 2

## Des sprints pour une release

J'ai pris part à des dizaines de projets, soit en tant que développeur, soit en tant que consultant et il n'y en a pas deux qui se soient déroulés de la même façon, bien que certains aient suivi le même processus. Il y a une grande variation dans le déroulement temporel d'un projet, appelé le **cycle de développement** (ou cycle de vie).

Un cycle est défini par des **phases** et des **jalons**. Les phases se succèdent et un jalon permet de contrôler le passage à la phase suivante : une phase a des objectifs et le jalon est là pour vérifier qu'il n'y a pas de déviation par rapport à ces objectifs.



**Figure 2.1** — Dans un cycle traditionnel, chaque phase est différente

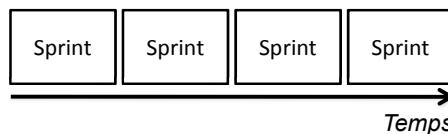
Évidemment le cycle est influencé par le **modèle de cycle** (ou processus) qu'on utilise. Un modèle ancien, encore répandu en France, est le **cycle en V**, mais le plus souvent une entreprise, surtout si elle grande, a défini son propre modèle de cycle.

Dans certaines entreprises, l'application du modèle est fortement recommandée et dans d'autres l'équipe a plus de latitude. Bien souvent, et quel que soit le degré de recommandation, j'ai constaté qu'il y avait un grand écart entre le modèle et sa mise en œuvre sur les projets.

Il y a plusieurs raisons pour l'expliquer :

- Le modèle est bien souvent trop théorique, élaboré par des méthodologues éloignés des réalités, et inapplicable sur le terrain.
- Les contrôles sont difficiles à faire lors des jalons, parce qu'ils portent souvent sur une multitude de documents.
- Les jalons étant franchis sans difficulté, l'équipe accumule les travaux non faits des phases précédentes.

Scrum fait partie des approches itératives et incrémentales, dont le modèle de cycle de développement est basé sur une phase qui se répète plusieurs fois successivement. C'est la notion d'itération, appelée **sprint** avec Scrum. Tous les sprints se déroulent selon le même schéma et on y fait à chaque fois les mêmes types de travaux.



**Figure 2.2** — Avec Scrum, le processus se répète à chaque *sprint*

C'est une différence majeure avec les méthodes traditionnelles pour lesquelles les travaux sont de nature différente pour chaque phase. Cela a un effet sur les objectifs de chaque *sprint* : ils ne sont pas définis par le cadre Scrum, c'est l'équipe qui s'en charge.

C'est de ce cycle de développement et de ses implications dont il est question dans ce chapitre.

## Définitions

**Sprint** est le terme utilisé dans Scrum pour *itération*. Dans le langage Scrum, un *sprint* est un bloc de temps fixé aboutissant à créer un incrément du produit potentiellement livrable.

**Release** peut avoir plusieurs sens :

– **Release comme version** – Le dictionnaire du jargon informatique français définit une *release* comme suit : « *version d'un logiciel effectivement diffusée, donc lâchée dans la nature. Synonyme de « Mise sur le marché »* ». Exemple : *Unix system V release 4* ». Cette définition énonce clairement qu'il y a des versions qui ne constituent pas des *releases*.

– **Release comme période de temps** – Par extension, on appelle également *release* la période de temps qui permet de la produire. On devrait dire le *cycle de production d'une release*, mais l'usage en anglais, et maintenant en français, est d'utiliser *release* comme période de temps, notamment dans l'expression *plan de release*. C'est ce sens, période de temps composée de *sprints*, qui est utilisé pour *release* dans ce livre.

## 2.1 L'APPROCHE ITÉRATIVE ET INCRÉMENTALE

### 2.1.1 Incrémentation et itération

Scrum utilise une approche itérative et incrémentale pour le développement d'un produit.

#### Incrémental

Incrémental est utilisé pour mettre en évidence l'accroissement du produit obtenu à la fin de chaque *sprint*. Un processus incrémental permet de construire un produit morceau par morceau, chaque nouvelle partie venant s'ajouter à l'existant.

Pour l'écriture de ce livre, j'ai utilisé une approche incrémentale : j'ai fait un plan initial et j'ai rédigé chapitre par chapitre, sans respecter l'ordre du plan, d'ailleurs.

La pratique d'un cycle incrémental est répandue dans les développements de logiciel ; on parle souvent de lots pour les incréments qui font l'objet d'un contrat.

#### Itératif

Itérer est l'action de répéter. Dans le calcul scientifique, l'itération est un procédé de calcul répétitif qui permet, par exemple, de trouver la racine d'un nombre, d'une équation, par approximations successives.

Dans le développement de logiciel, le terme *itération* est utilisé pour désigner une période de temps dans laquelle sont effectuées des activités, qui seront répétées dans les prochaines itérations. Le terme est souvent associé à *processus*.

**Exemple** : un article du *Nouvel Observateur* (grand public, donc) de juillet 2008 met en exergue les itérations du processus d'Amazon, qui expliquent, selon l'auteur, les succès de l'entreprise.

Un processus itératif permet de revenir sur ce qui a été fait, dans le but de l'améliorer ou de le compléter. Cela part de l'idée qu'il est difficile, voire impossible, de bien faire la première fois. Le *feedback* collecté sur le résultat d'une itération permet de faire des améliorations dans la suivante. On cesse d'itérer quand la qualité obtenue est jugée satisfaisante.

Pour l'écriture de ce livre, j'ai utilisé une approche itérative : j'ai diffusé le premier jet à des lecteurs et grâce à leur *feedback*, j'ai produit une nouvelle version.

### Itératif et incrémental

Scrum combine les deux approches avec la notion de *sprint* :

- à l'issue du *sprint*, il y a un incrément de produit qui est réalisé,
- le *feedback* sollicité sur cet incrément permet de le perfectionner dans un prochain *sprint*.

En résumé, un *sprint* est une itération qui produit un nouvel incrément (incrémental) et peut aussi enrichir un incrément d'un *sprint* précédent (itératif).

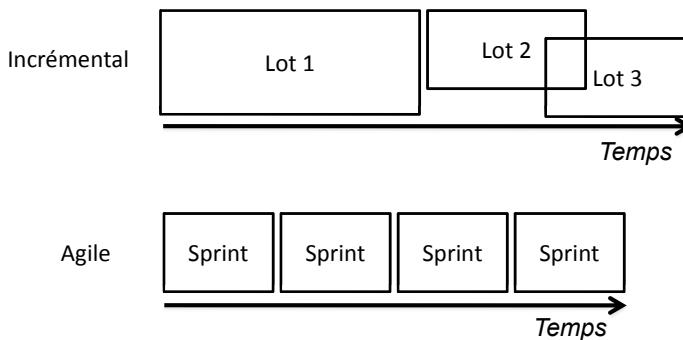
Pour l'écriture de ce livre, j'ai utilisé une approche itérative et incrémentale : en fait, je n'ai pas diffusé le premier jet de tout le livre à mes lecteurs, mais celui d'un chapitre. Comme j'ai suivi Scrum pour ce projet de rédaction, dans un *sprint* je travaillais sur la première version d'un nouveau chapitre et aussi sur la révision d'un chapitre suite au retour d'un ou plusieurs lecteurs.

Les organisations qui font des développements de logiciel en passant par la production d'une maquette, celles qui procèdent par lots, celles qui produisent une ou plusieurs versions intermédiaires disent volontiers qu'elles appliquent un processus *itératif et incrémental*. Elles ne sont pas agiles pour autant.

### Cycle agile

Quelles sont les caractéristiques du cycle de vie Scrum, en plus d'être itératif et incrémental, qui justifient le qualificatif d'*agile* ?

- Des itérations plus courtes : les *sprints* durent au maximum un mois.
- Une séquence plus stricte : les *sprints* ne se chevauchent pas.
- Un rythme régulier : les *sprints* ont toujours la même durée.



**Figure 2.3** – Différences entre incrémental et agile

## 2.1.2 Bloc de temps

Les *sprints* ont tous la même durée : Scrum s'appuie sur la notion de bloc de temps limité (*timebox*).

Il n'y a pas que les *sprints* qui sont *timeboxed* avec Scrum : de nombreuses activités d'un développement sont basées sur cette notion. Cela se manifeste en particulier dans les réunions qui constituent le cérémonial.

### Pas de sprint extensible

Pour le *sprint*, la notion de *timebox* s'applique de la façon suivante : on ne change pas la date de fin une fois que le *sprint* a commencé. Même si on n'a pas fini tout ce qu'on voulait faire, on garde la date de fin prévue.

Pourquoi ? Cela permet d'éviter le syndrome du presque fini (ou fini à 90 %), où, finalement, la date de fin est repoussée plusieurs fois.

J'ai accompagné de nombreux projets, agiles ou pas. J'ai eu très souvent des demandes d'équipes venant négocier la date d'un jalon. Ces équipes me disent qu'elles ont un tout petit peu de retard et demandent à repousser la date d'une revue de deux ou trois jours. Juré, avec ce délai, ce serait mieux et on aurait tout ce qui était prévu. Évidemment la plupart du temps, après ce laps de temps supplémentaire, il en aurait fallu encore un peu... Les développeurs ont tendance à être optimistes.

La notion de bloc de temps évite les dérives : à la date prévue, on fait une inspection objective de l'avancement et on ajuste en conséquence la planification des prochains *sprints*.

### Rythme régulier

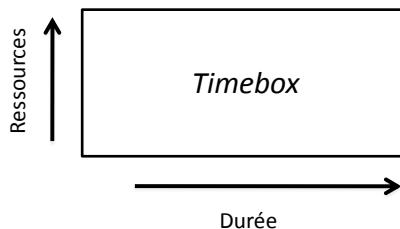
La durée des *sprints* est toujours la même, dans la mesure du possible. L'intérêt est de donner un rythme à l'équipe, qui va apprendre à produire régulièrement.

L'objectif est d'éviter des situations de sur-régime que l'équipe ne pourra pas tenir bien longtemps. Pousser une équipe à travailler au-delà de son régime de croisière a des effets de bord négatifs sur la qualité de son travail : le nombre de défauts augmente, la motivation diminue, les pratiques d'ingénierie sont négligées...

Au contraire, un rythme régulier peut être conservé longtemps, voire indéfiniment. Il présente d'autres avantages : comme on connaît les dates de début et de *sprint* à l'avance, les revues sont plus faciles à organiser et les intervenants peuvent planifier leur participation.

### Ressources régulières

Le bloc de temps délimite la quantité de travail faite pendant le *sprint*. Il est le même pour tous les *sprints* : il dépend de la durée du *sprint* et de la taille de l'équipe qui sont fixes toutes les deux, au moins sur une certaine période.



**Figure 2.4** — Le bloc de temps

Comme pour le développement de logiciel, le coût est directement corrélé aux ressources, le budget d'un *sprint* peut être déduit de la *timebox*.

Il est préférable que la composition de l'équipe reste stable pendant un *sprint*. Il est aussi souhaitable de garder une équipe stable pour une *release*, ce qui permet d'avoir des *timeboxes* équivalentes pour tous les *sprints*.

### 2.1.3 Durée du sprint

La durée d'un mois ou moins correspond à l'horizon de prédictibilité : à la fin du *sprint*, les prévisions sont ajustées en fonction du contrôle effectué sur les résultats obtenus. Pour les développements complexes qui sont la cible privilégiée de Scrum, on considère que ne pas réguler le processus pendant plus d'un mois, ce serait prendre trop de risques.

Aux débuts de Scrum, la règle était de faire des *sprints* d'un mois, sans variation. Par exemple, si le développement commençait avec une année calendaire :

- premier *sprint* du premier au 31 janvier,
- deuxième *sprint* du premier au 28 (ou 29) février,
- troisième *sprint* du premier au 31 mars...

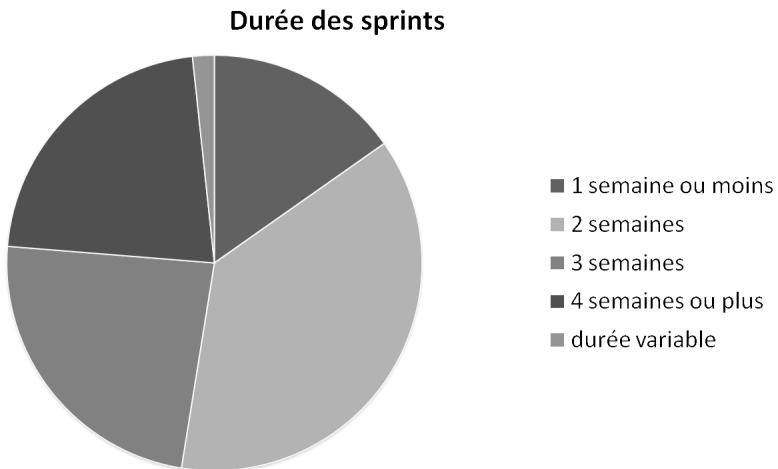
Aujourd'hui, on constate une tendance à faire des *sprints* plus courts : en effet, pour le développement de logiciel, les pratiques d'ingénierie, comme l'intégration continue et les outils associés, permettent de produire des versions partielles plus fréquemment.

#### Quelques infos pratiques

Une enquête faite en avril 2009 sur mon blog ([www.aubryconseil.com](http://www.aubryconseil.com)) auprès d'une

centaine de personnes donnait les résultats présentés figure 2.5.

Les *sprints* de deux ou trois semaines représentent la majorité des réponses.



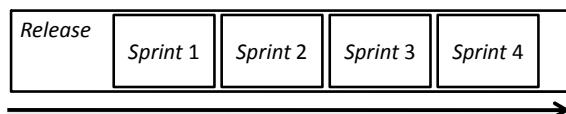
**Figure 2.5** — Sondage sur la durée des *sprints*

## 2.2 CYCLE DE DÉVELOPPEMENT SCRUM

### 2.2.1 L'aspect temporel

#### Phases et jalons

Dans chaque processus de développement, il existe des jalons majeurs et des jalons mineurs. Avec Scrum le jalon min✉ est la fin du *sprint* et le jalon majeur est la production de la *release*.



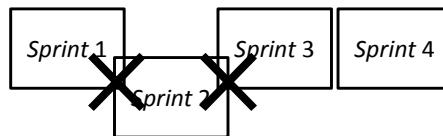
**Figure 2.6** — Une *release* et ses *sprints*

Une *release* est une série de *sprints* qui se termine quand les incrément successifs constituent un produit qui présente suffisamment de valeur à ses utilisateurs.

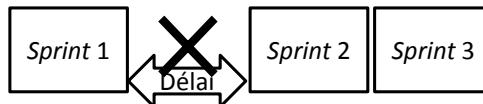
La durée des *releases* est définie par l'équipe et le Product Owner. La tendance est à raccourcir ces durées : pour de nombreuses équipes, une *release* dure environ trois mois, avec des *sprints* de deux ou trois semaines. Cela permet de dérouler de quatre à six *sprints* dans une *release*.

Il n'y a pas de chevauchements : on ne commence pas un *sprint* tant que le précédent n'est pas terminé et, en principe, le nouveau démarre immédiatement après le précédent.

Les *sprints* s'enchaînent sans délai : le nouveau démarre immédiatement après le précédent.



**Figure 2.7** — Les *sprints* sont séquentiels



**Figure 2.8** — Les *sprints* s'enchaînent

### Arrêter le *sprint* plutôt que l'étendre

La date de fin du *sprint* est fixée au début du *sprint* (elle est même définie avant). Elle ne change pas, même si l'équipe ne réalise pas tout ce qu'elle imaginait faire. L'évaluation de fin de *sprint* permettra d'inspecter ce qui a été fait et d'en tirer des conséquences pour la suite.

Il peut arriver que l'équipe n'arrive à pas produire un incrément potentiellement livrable à la fin du *sprint* et n'ait rien de montrable. Heureusement, c'est très rare. Cela peut être dû à des difficultés techniques ou à des perturbations importantes qui changent le but du *sprint*.

Dans ce cas, lorsque l'équipe se rend compte qu'elle ne pourra pas présenter quelque chose à la fin du *sprint*, une possibilité est d'arrêter immédiatement le *sprint* en cours. L'équipe repart aussitôt dans un nouveau *sprint*, avec un périmètre adapté qui tient compte des difficultés rencontrées.

Ce conseil peut être suivi pour une équipe qui fait des *sprints* d'un mois et se rend compte de son impasse au milieu du *sprint*. Pour des durées de *sprint* plus courtes, il est plus simple d'attendre la fin normale du *sprint* et de tirer les enseignements lors de la rétrospective.

### 2.2.2 Activités et cycle de développement

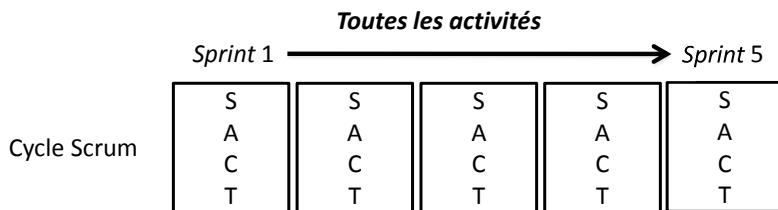
Un cycle de développement se présente comme un enchaînement de phases dans lesquelles on effectue des activités. Pour un développement de logiciel, les activités sont généralement :

- Spécification fonctionnelle (*requirements*)
- Architecture (conception)
- Codage (et test unitaire)
- Test (d'intégration et de recette)

Pour simplifier, je vais utiliser les lettres S A C T pour désigner ces activités dans les schémas.

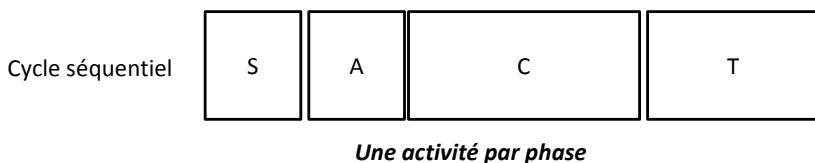
Avec Scrum, chacun des *sprints* a un objectif qui porte sur un produit qui fonctionne (et pas seulement sur des documents), ce qui nécessite du travail dans toutes les activités de développement pendant un *sprint*.

Les activités se déroulent en parallèle pendant le *sprint* (figure 2.9) :



**Figure 2.9** — Des *sprints* et leurs activités en parallèle

À l'autre extrême, un processus dit séquentiel déroule les activités en séquence, avec une activité par phase (figure 2.10) :

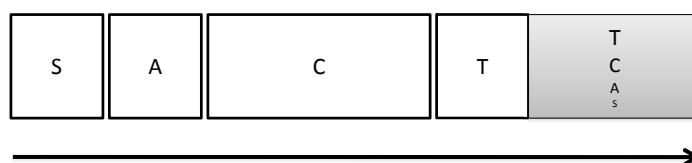


**Figure 2.10** — Phases avec des activités séquentielles

Avec un processus à activités séquentielles, une **phase** est définie avec un objectif exprimé par une liste de documents à produire ; elle dure assez longtemps – quelques mois – et sa durée est variable (l'équipe s'arrête lorsque les objectifs sont atteints) ; le résultat porte uniquement sur des documents au début, et même assez tard dans le développement : le logiciel testé n'est obtenu qu'à la fin. Suivre au pied de la lettre cette approche revient à avoir :

- 100 % de la spécification fonctionnelle détaillée avant de commencer le code,
- 100 % de la conception avant de commencer le code,
- 100 % du code pour commencer les tests.

C'est un modèle idéal mais utopique : dans la réalité on revient toujours sur les activités des phases précédentes, et en particulier dans la phase de test, on revient sur les spécifications, la conception et le code (figure 2.11).



**Figure 2.11** — Retour sur les activités précédentes pendant le test

Ce décalage entre le modèle et la réalité est une des raisons du retard des projets et de leur qualité médiocre.

Avec une méthode agile comme Scrum, les activités de spécification et de conception sont continues. On part du principe que l'architecture va évoluer, dans une certaine limite, pendant la vie du projet. L'approche est plus réaliste et pragmatique. L'autre principe important est que les tests sont pratiqués dès le premier *sprint*.

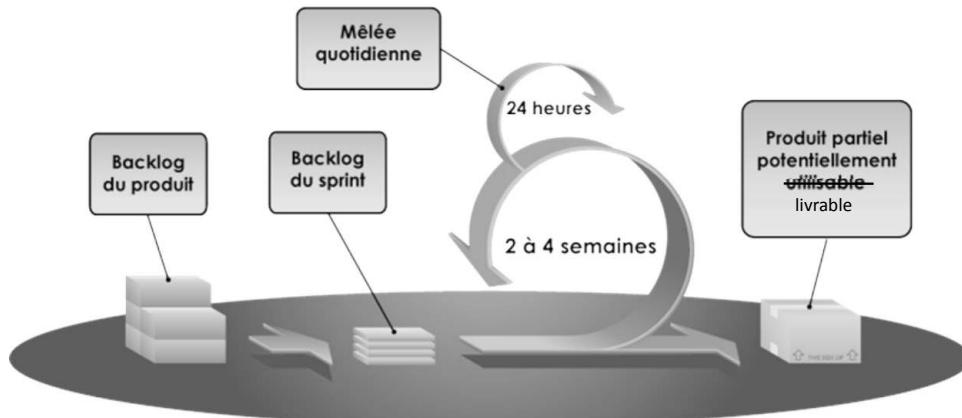
Il existe aussi de nombreux développements qui sont menés de façon chaotique, sans suivre de processus ou en le suivant « à l'arrache »<sup>1</sup>. Le plus souvent, il n'y a même pas de spécification ni de conception. L'équipe se lance directement dans le codage, qui est suivi d'une longue période de test et de corrections de bugs.

Avec Scrum, la qualité n'est pas négligée, et la conception fait partie des activités de chaque *sprint*.

### 2.2.3 Le résultat d'un sprint

L'inspection<sup>2</sup>, afin de faire des adaptations, est à la base de la théorie de Scrum.

À la fin d'un *sprint*, le résultat attendu est un incrément du produit final, qui est potentiellement livrable. C'est ce que montre le célèbre schéma de Mike Cohn que j'avais traduit en français en 2005 (figure 2.12).



**Figure 2.12** — Cycle de vie Scrum simplifié

À l'époque du schéma, en 2005, j'avais (mal !) traduit « *potentially shippable* » par potentiellement utilisable. C'est potentiellement **livrable**.

1. La méthode à l'arrache est bien connue, voir <http://www.la-rache.com>

2. L'inspection du produit est décrite dans le chapitre 9 *La revue de sprint*. L'inspection du processus est décrite dans le chapitre 10 *La rétrospective de sprint*.

Le produit ne doit pas être seulement « potentiellement » utilisable à la fin d'un *sprint*, il doit simplement être utilisable. Certes, il n'est pas complet par rapport à ce qui est prévu dans la *release*, mais il est livrable, au moins à des utilisateurs privilégiés.

Dans le cas d'un développement de logiciel, le minimum est d'avoir déployé, à la fin d'un *sprint*, le produit potentiellement livrable, avec si nécessaire la documentation permettant de l'utiliser.

#### 2.2.4 Le résultat d'une release

Le résultat de la *release* est le produit livrable, fourni à ses utilisateurs. La façon dont il est fourni dépend de la nature du déploiement de ce produit.

La distinction avec le résultat du *sprint* se fait sur le **potentiellement**. Le but est de faire en sorte que cette distinction soit la plus ténue possible voire qu'elle disparaîsse. C'est très variable selon le contexte du projet : des déploiements très fréquents sont possibles pour des applications web hébergées...

**Exemple :** eBay re-déploie ses applications tous les quinze jours et pour Amazon c'est du déploiement continu.

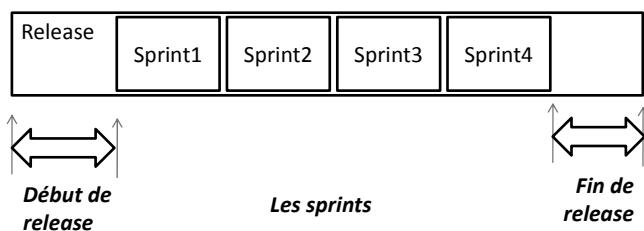
...mais pas pour des applications internes ajoutées à un gros système d'information, ni pour des systèmes embarqués.

Souvent, le jalon majeur que représente la *release* correspond à une annonce marketing : à l'occasion de la « sortie » du produit, les équipes marketing préparent un matériel pour sa promotion.

### 2.3 GUIDES POUR LES SPRINTS ET RELEASES

Avec Scrum, la notion de cycle de vie est peu mise en évidence. Le premier article de Ken Schwaber évoquait trois phases : une première *Planning & Architecture*, la deuxième constituée des *sprints* et une dernière phase appelée *Closure*. Ces notions sont aujourd'hui abandonnées. Néanmoins, il y a bien trois périodes distinctes dans une *release* (figure 2.13) :

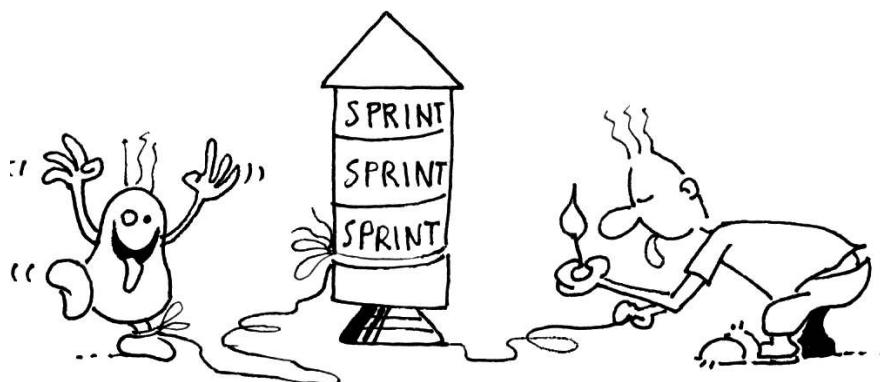
- La période centrale, celle des *sprints*.
- La période avant le premier *sprint*.
- Une période après le dernier *sprint* et avant la fin de la *release*.



**Figure 2.13** – Les trois périodes d'une *release*

### 2.3.1 Démarrer le premier sprint au bon moment

Le développement d'une *release* commence par des travaux particuliers à faire avant de lancer les *sprints* successifs, comme constituer l'équipe, définir la vision, produire un *backlog* initial et une première planification de la *release*. Si la *release* en question est la première dans la vie du produit, il conviendra également de faire des travaux de définition de produit<sup>1</sup> et d'architecture avant de lancer les *sprints*.



**Figure 2.14** – Le lancement des *sprints* se prépare

La période de temps en début de *release* est parfois appelée le *sprint zéro*. *Sprint zéro* est trompeur parce que cette période n'est pas un *sprint* comme les autres : sa durée est variable, les tâches qu'on y fait sont spécifiques de cette phase, il n'y a pas le cérémonial habituel des *sprints*, on ne produit pas une version potentiellement utilisable à la fin, on ne mesure pas de vitesse...

À l'usage je trouve que c'est une très mauvaise appellation, dangereuse. Au-delà du vocabulaire, ce qu'il faut bien comprendre c'est qu'il s'agit d'une phase différente de la série des *sprints* qui va suivre. Elle est prédictive alors que la phase des *sprints* est empirique, le but n'est pas de produire un incrément de produit comme pour

1. Voir aussi, le chapitre 13 *De la vision aux stories*.

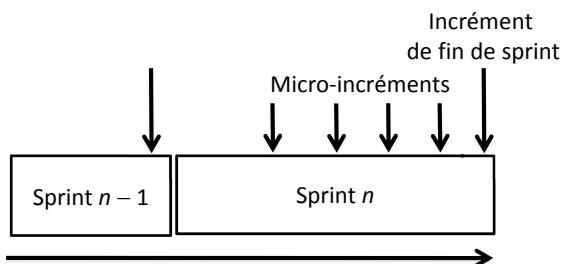
chaque *sprint*. Le risque avec *sprint* zéro est que cette distinction ne soit pas perçue et que cette phase préparatoire soit négligée.

Le premier *sprint* ne doit pas démarrer trop longtemps après le début de la *release* (il ne s'agit pas d'y caser les activités de spécification et de conception détaillées), mais cela dépend du contexte : il faut évidemment plus de temps dans le cas d'une première *release* d'un nouveau projet s'appuyant sur une nouvelle technologie que pour sa énième *release*.

Ce n'est pas non plus une bonne chose de démarrer trop vite : avant que la planification de *release* ne soit élaborée, c'est prématué de commencer le premier *sprint*.

### 2.3.2 Produire des micro-incréments

Un *sprint* ne se déroule pas en travaillant successivement sur les activités (spécification puis conception puis codage puis test) : un *sprint* n'est pas un « mini-cycle en V ». L'équipe travaille pour développer des fonctionnalités dès le début du *sprint*, ce qui permet de produire pendant le *sprint* ce qu'on peut appeler des micro-incréments (figure 2.15).



**Figure 2.15** — Production de micro-incréments

Dans le cadre d'un développement logiciel, ces micro-incréments sont des versions intermédiaires produites pendant le *sprint*. Elles sont utilisées par l'équipe de développement et le Product Owner pour passer les tests fonctionnels.

### 2.3.3 Enchaîner les sprints

Si on se réfère à l'athlétisme, objet de la métaphore, on ne peut pas *sprinter* pendant toute la durée de la course de fond que constitue une *release*, il faut des phases de récupérations.

Ce n'est pas moi qui vais dire le contraire : j'ai couru en longue distance. J'ai encore le souvenir des entraînements en fractionné : par exemple une série de dix fois 200 m avec 30 secondes de récupération entre chaque.



**Figure 2.16** — La fatigue de l'équipe après un *sprint* à fond

Certains membres de l'équipe le font savoir lors des rétrospectives : ils souhaitent des jours de récupération entre les *sprints*. Parmi toutes les équipes que j'ai suivies depuis cinq ans, la plupart se contentent d'un week-end avant de repartir sur un nouveau *sprint* : la revue et la rétrospective se font le vendredi (mais ce n'est pas obligatoire de caler les *sprints* sur les semaines) et le prochain *sprint* démarre le lundi qui suit par la réunion de planification.

D'autres consacrent un jour entre chaque *sprint* à des activités hors projet ; une équipe a instauré un *bug day* intercalaire. J'ai aussi connu des équipes qui choisissaient de s'arrêter une semaine tous les trois ou quatre *sprints*.

Et puis certaines équipes enchaînent directement : par exemple, fin de *sprint* le mardi, début du *sprint* suivant le mercredi matin, par exemple. C'est le mode de fonctionnement optimal. Les situations où les équipes éprouvent le besoin de s'arrêter entre les *sprints* sont souvent le reflet d'un dysfonctionnement. Le terme *sprint*, qu'on court à fond et après lequel on récupère, est trompeur. Un développement avec Scrum s'apparente plus à une course à un rythme régulier, sans pause à chaque étape.

### 2.3.4 Utiliser le produit partiel

En plus de sa présentation à la revue de *sprint*, on peut identifier trois usages de la version produite en fin de *sprint*, pour un développement de logiciel.

- **Utilisation interne** – La version n'est pas utilisée en dehors de l'équipe de développement. Elle a été produite pour chercher à minimiser les risques liés à la technologie et à la capacité de l'équipe à intégrer différents composants.

Elle n'est pas livrée à l'extérieur de l'équipe. Cela est fréquent au début d'un nouveau développement.

- **Utilisation pour feedback par des utilisateurs sélectionnés** – La version est utilisée par un client ou des utilisateurs privilégiés. Cela leur donne la possibilité de la prendre en main, ce qui permet de réduire les risques portant sur l'interface. Ces utilisateurs pourront évaluer la facilité d'utilisation des fonctionnalités et en proposer de nouvelles. Les retours faits iront alimenter le *backlog* pour prise en compte ultérieure.
- **Mise en production** – La version est mise en production ou en exploitation et utilisée par ses utilisateurs finaux. C'est évidemment ce qu'il faut viser puisque chaque nouvelle version apporte de la valeur. Autant l'apporter le plus tôt possible, dès qu'elle est disponible. Mais ce n'est généralement pas faisable de mettre en production à la fin de chaque *sprint* : trop de temps serait pris, par rapport à la durée du sprint, pour passer les tests de recette sur tout le système, pour déployer sur l'environnement de production, pour écrire les manuels utilisateurs, pour préparer et donner la formation aux utilisateurs...

C'est pourquoi, dans la plupart des cas, l'état du produit à la fin<sup>1</sup> de chaque *sprint* est distingué de celui souhaité à la fin d'une *release*.

Mais si on réussit à automatiser le déploiement et à limiter le temps pour le faire, on peut alors mettre en production plus souvent qu'à la fin des *releases*. Dans ce cas-là, le produit n'est plus simplement potentiellement livrable à la fin de chaque *sprint*, il est livré. Le terme de *release* garde son sens de période de temps pour la planification.

### 2.3.5 Savoir finir la release

Faire le plus possible pendant les *sprints* évite de devoir poursuivre le travail après le dernier *sprint*. Si l'état du produit partiel en fin de *sprint* est équivalent à l'état attendu en fin de *release*, la fin du dernier *sprint* coïncidera avec la fin de la *release*.

Ce n'est pas possible dans tous les contextes et dans certains cas, il faut une période de temps entre la fin du dernier *sprint* et la fin de la *release* pour faire des travaux nécessaires avant la mise en production. Ces travaux varient selon le type de déploiement : mise en production à chaud, *packaging* du produit, mise à disposition par téléchargement en ligne...

Cette période était auparavant appelée *sprint* de stabilisation. Ce n'était pas une bonne idée, cela laissait entendre que le logiciel n'était pas stable avant. Les termes de *sprint* de *release* ou de *sprint* de durcissement parfois évoqués sont tout aussi discutables :

- il vaut mieux rendre le produit plus robuste à chaque *sprint* plutôt que de le faire à la fin,
- si malgré tout, il y a des travaux de durcissement à faire avant de livrer la *release*, cela ne rentre pas dans le cadre d'un *sprint*.

---

1. La signification de fini fait l'objet du chapitre 11 .

## En résumé

Avec Scrum, un produit est développé selon une approche itérative et incrémentale. Le *sprint* donne le rythme auquel sont produites des versions partielles potentiellement livrables du produit. La *release* est la séquence de *sprints* à l'issue de laquelle le produit est livré aux utilisateurs.

# 3

## Le Product Owner

Il y a quelques années, je travaillais chez un éditeur de logiciel. Plus exactement un éditeur de génie logiciel qui réalisait et commercialisait des outils de modélisation.

Le poste que j'occupais s'appelait *Marketing Produit*. Dans d'autres entreprises, on parle de **chef de produit**. À la direction marketing, nous faisions des études de marché, nous animions des groupes d'utilisateurs, nous définissions notre vision des nouveaux produits et nous en faisions la promotion. Je m'occupais d'un produit, aujourd'hui disparu, comprenant un éditeur graphique, un simulateur et un générateur de code. La cible principale était le secteur des télécoms.

J'avais été moi-même développeur et chef de projet dans le domaine des télécoms, et donc je connaissais bien les utilisateurs potentiels du produit. Avec le rôle que j'avais, je les représentais en quelque sorte. J'essayais de faire prendre en compte leurs besoins dans le produit.

Le produit était développé par une équipe de la direction technique. Je ne la voyais pas très souvent, je ne les connaissais même pas tous. Je voyais un peu plus souvent le chef de projet. Lui et le directeur technique avaient aussi leurs idées sur le produit.

Comme c'était une société dirigée par la Technique, c'était souvent eux qui avaient le dernier mot pour mettre ce qu'ils avaient décidé dans le produit. Il y avait aussi les commerciaux qui, pour essayer de vendre à leurs clients, allaient voir directement le chef de projet, ou le directeur technique, voire le PDG. Je n'étais mis au courant qu'après, sans pouvoir revenir sur les décisions prises.

Le résultat de ces apports différents à la définition du produit était un manque d'homogénéité. Le produit comportait des fonctions très puissantes. Par exemple de la simulation exhaustive de modèles basés sur des machines à états. De ce fait, il n'était pas simple à utiliser. Les fonctions proposées ne tenaient pas réellement compte des besoins des utilisateurs. Le produit était finalement bancal, même si, au

marketing, nous essayions de le masquer, lors des présentations aux commerciaux et aux utilisateurs.

C'est pour éviter ce manque d'unité dans le contenu d'un produit que le **Product Owner** existe dans Scrum. Sa raison d'être c'est de s'assurer que le travail fait apporte de la valeur aux utilisateurs. Il est responsable de la définition du contenu du produit et de la gestion des priorités pour son développement.

L'existence du Product Owner permet aussi d'éviter, comme dans mon histoire, la prédominance de la Technique dans un développement de produit. Son boulot, c'est de dire à l'équipe quel produit réaliser. Il est le seul à avoir cette responsabilité. Cela évite les conflits d'intérêts survenant lorsqu'elle est partagée entre plusieurs personnes.

À l'époque, si j'avais connu le rôle, j'aurais bien aimé être Product Owner. Le temps a passé et depuis que je pratique Scrum, c'est le rôle que j'ai le plus exercé sur des projets. En particulier, je suis, depuis plusieurs années, le Product Owner du produit IceScrum, un outil *open source* de gestion de projet avec Scrum.

C'est de ce rôle et de mes expériences dont il est question dans ce chapitre.

### **Product Owner vs directeur de produit**

J'ai d'abord appris Scrum par la lecture de livres et d'articles, tous en anglais. J'ai naturellement traduit Product Owner en propriétaire de produit quand je m'exprimais à propos de ce rôle.

Le terme propriétaire peut être satisfaisant pour celui qui joue le rôle : il peut dire « c'est mon produit ! ». Ayant fait, il y a quelques années, des travaux sur les processus métier (*Business Process Management* ou BPM), je me souviens du rôle de Process Owner, le « propriétaire de processus ». Mon expérience avec des propriétaires de processus m'incline à penser que finalement cette idée de propriétaire n'est pas bonne. Le terme ne reflète pas la vraie nature du rôle : le Product Owner aime son produit certes, mais il ne le possède pas plus que le reste de l'équipe, et moins que ses utilisateurs.

En 2005, j'ai suivi une formation Scrum en français donnée par François<sup>1</sup>, un Québécois. Le support de cours était en français aussi. Product Owner y était traduit en « Administrateur du produit ». Cet intitulé n'a jamais pris en France et je ne crois pas que nos cousins du Canada l'aient conservé non plus.

Un peu plus tard, suite à la lecture d'un article de Brian Marick (*How to be a Product Director*<sup>2</sup>), j'ai adopté le terme **directeur de produit**. J'en ai parlé dans plusieurs billets de mon blog<sup>3</sup>, pendant quelques mois.

1. François Beauregard, préfacier de cet ouvrage.

2. <http://www.testing.com/cgi-bin/blog/2006/04/21# how-to-be-a-product-diretor>

3. [www.aubryconseil.com](http://www.aubryconseil.com)



**Figure 3.1** — Un propriétaire de produit un peu possessif

L’article en français de Wikipedia sur Scrum<sup>1</sup> a repris le terme. À l’heure où j’écris ces lignes, directeur de produit apparaît toujours dans cet article, d’ailleurs. Ce qui fait que cette traduction de Product Owner a eu un certain succès et que je rencontre toujours des personnes qui l’utilisent. La version française de « *Scrum et XP depuis les tranchées*<sup>2</sup> » l’a reprise également.

Pourtant, j’ai arrêté d’utiliser directeur de produit pour revenir à l’anglais Product Owner (mais en le prononçant à la française). Le mot directeur ne passait pas dans des organisations : celui qui tenait le rôle n’était pas un directeur au sens hiérarchique. Le Product Owner donne bien la direction, mais il n’a pas de responsabilité hiérarchique sur des personnes.

En 2007, j’ai fait un petit sondage auprès des utilisateurs de Scrum et finalement c’est Product Owner qui est apparu convenir le mieux. Même dans les administrations françaises !

Depuis je reste fidèle à l’appellation **Product Owner**, PO en abrégé, et j’ai l’impression que c’est le cas de la majorité des personnes de la communauté française de Scrum.

1. <http://fr.wikipedia.org/wiki/Scrum>

2. <http://henrik-kniberg.developpez.com/livre/scrum-xp/>

## 3.1 RESPONSABILITÉS

Le Product Owner a des **responsabilités** importantes, portant à la fois sur la stratégie et la tactique dans le cadre du développement d'un produit.

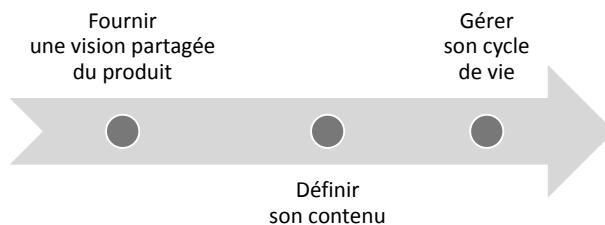
Il prend des décisions de niveau stratégique qui sont d'habitude du ressort de la direction du projet ou des comités de pilotage. Le Product Owner décide dans des domaines qui sont ceux d'un chef de projet traditionnel, par exemple la date de livraison du produit.

Le Product Owner prend aussi de nombreuses décisions de niveau tactique qui sont habituellement prises par une équipe de développement, faute d'implication des « clients ». En effet, en leur absence, les développeurs ont tendance à faire des choix qui ne sont pas, en principe, de leur responsabilité.

Un Product Owner présent et impliqué fera ces choix tactiques, comme par exemple l'apparence et le contenu d'un formulaire pour un site marchand.

**Attention :** même si le Product Owner a des responsabilités, il ne faut pas le considérer comme quelqu'un qui impose ses choix de façon autoritaire ; beaucoup de travaux qu'il mène se font en équipe et ses décisions sont prises, chaque fois que c'est important, en accord avec elle.

Le rôle de Product Owner varie beaucoup avec le contexte de l'organisation, cependant il est possible d'identifier ses responsabilités majeures :



**Figure 3.2** – Responsabilités du Product Owner

### 3.1.1 Fournir une vision partagée du produit

Le Product Owner est responsable de définir l'objectif du produit et de le partager avec l'équipe qui le développe.

Sans être obligatoirement un visionnaire comme Steve Jobs, le Product Owner doit avoir une bonne vision du produit. La vision se construit au début du développement d'un nouveau produit et se consolide ensuite. Elle consiste typiquement à définir :

- l'énoncé du problème que le produit veut résoudre,
- une position du produit qui soit claire pour tout le monde,
- une liste des fonctionnalités essentielles.

Chaque membre de l'équipe et toutes les parties prenantes du projet doivent partager la même vision<sup>1</sup>, et c'est au Product Owner de s'en assurer.

### 3.1.2 Définir le contenu du produit

Le Product Owner définit le contenu du produit. Pour cela, il identifie les fonctionnalités requises et les collecte dans une liste, appelée le **backlog de produit**<sup>2</sup>.

Concrètement, le Product Owner est responsable du *backlog* de produit et y contribue de façon régulière. En plus de son travail pour le *sprint* courant, il passe une bonne partie de son temps sur les éléments du *backlog* prévus pour les *sprints* suivants.

Comme il est moteur pour établir ce que doit faire le produit, il est logique qu'il fournit aussi ses conditions de satisfaction, qui permettront de s'assurer que ce qu'il demande est bien réalisé : il est donc impliqué dans les tests d'acceptation<sup>3</sup>.

### 3.1.3 Planifier la vie du produit

C'est le Product Owner qui définit l'ordre dans lequel les parties du produit seront développées. Il doit alimenter l'équipe avec les fonctionnalités à développer, selon ses priorités définies en fonction de la valeur qu'elles apportent.

L'ordre de réalisation définit le cycle de vie du produit. Cette vie est constituée d'une succession de versions (les *releases*). Le Product Owner définit l'objectif d'une *release* et prend les décisions sur son contenu et sa date de mise à disposition du produit.

En résumé, s'il n'a pas d'autorité formelle sur des personnes, le Product Owner a une grande influence sur le produit réalisé.

## 3.2 COMPÉTENCES SOUHAITÉES

La personne idéale pour jouer ce rôle devrait posséder les compétences présentées figure 3.3.

On imagine qu'une personne avec ce profil idéal est un oiseau rare dans la plupart des organisations.

---

1. Pour plus d'informations, voir le chapitre 13 *De la vison aux stories*.

2. Le *backlog* fait l'objet du chapitre 5.

3. Les tests d'acceptation font l'objet du chapitre 14 *De la story aux tests*.

- Bonne connaissance du domaine métier
- Maîtrise des techniques de définition de produit
- Capacité à prendre des décisions rapidement
- Capacité à détailler au bon moment
- Esprit ouvert au changement
- Aptitude à la négociation

**Figure 3.3** — Les compétences souhaitées d'un Product Owner

### 3.2.1 Bonne connaissance du domaine métier

Ce qu'on appelle le métier (*business*), et qu'on retrouve en français dans l'expression cœur de métier, constitue un domaine de connaissances en relation avec le quoi et le pourquoi d'un produit. Le quoi (*what*), c'est ce que doit faire le produit. Le pourquoi (*why*), c'est la justification de l'existence du produit et de son contenu, en termes de fonctionnalités.

Une bonne connaissance du domaine métier est fondamentale pour le Product Owner, puisqu'il est le représentant dans l'équipe de toutes les personnes qui utilisent ou font utiliser le produit ; celui étant développé pour rendre des services à ces personnes, par exemple en automatisant des parties d'un processus.

Le Product Owner peut avoir acquis cette connaissance parce qu'il est un des utilisateurs potentiels et c'est souvent le cas dans les entreprises qui développent des produits à usage interne ; dans celles produisant pour des clients externes, le Product Owner vient souvent des équipes marketing ou produit.

On ne demande pas à un Product Owner de tout connaître du domaine fonctionnel : sur des produits de grande taille, cela peut être une gageure ou demander beaucoup de temps pour tout connaître du métier. Il s'appuiera, quand cela s'avérera nécessaire, sur les bonnes personnes pour assumer pleinement son rôle.

### 3.2.2 Maîtrise des techniques de définition de produit

Le Product Owner définit ce que fait le produit. Il a besoin pour cela d'avoir la maîtrise des techniques de collecte des besoins et de leur transformation en éléments du *backlog* de produit. Traditionnellement, dans le développement de logiciel, la discipline d'ingénierie des exigences (*requirements engineering*) est celle qui définit le processus de collecte de ce que doit faire le produit.

Scrum ne prescrit pas de technique particulière pour identifier les éléments du *backlog*, Cependant le constat fait sur le terrain est que la pratique la plus fréquemment associée à Scrum est celle des « *user stories* »<sup>1</sup>.

La connaissance, et si possible la pratique des techniques de définition de produit, est requise pour le Product Owner.

### 3.2.3 Capacité à prendre des décisions rapidement

Choisir entre plusieurs alternatives sur des sujets d'importance variée, cela arrive quotidiennement sur un projet : pour des raisons d'efficacité, le Product Owner doit pouvoir le faire seul, sans en référer à une hiérarchie ou une autorité supérieure. Pour cela, il est essentiel qu'il ait la confiance des différents intervenants : il doit les voir régulièrement, les consulter sur les éléments du *backlog* qu'il connaît mal et sur les priorités pour la *release* en cours.

Si les avis de ces intervenants sont contradictoires, ce qui ne manquera pas d'arriver, le Product Owner doit avoir l'autorité pour décider et fédérer les points de vue en une seule proposition.

Il ne suffit pas qu'une personne décide, il faut aussi que ses décisions soient respectées et appliquées. Le Product Owner entraîne l'équipe en faisant en sorte que tout le monde adhère pleinement aux choix sur le contenu du produit.

### 3.2.4 Capacité à détailler au bon moment

Avec une approche agile, la spécification des exigences n'est pas détaillée dès le début du développement. Un gros travail pour tout spécifier au commencement du projet (BRUF, *Big Requirements Up Front*) n'est pas recommandé.

Comme tout n'est pas précisé au départ, le Product Owner doit savoir, en fonction de l'avancement, à quel moment détailler des éléments du *backlog* de produit.

Pour cela, il doit faire preuve d'anticipation. La notion de priorité entre les éléments du *backlog* va l'y aider : ce qui est le plus prioritaire doit être prêt en premier. Concrètement, cela signifie qu'il va passer du temps à s'impliquer sur les travaux courants, comme les autres membres de l'équipe, mais qu'une bonne partie de son temps porte aussi sur le futur du produit dans les semaines ou les mois qui viennent.

Le Product Owner décompose et détaille, au bon moment, le contenu du produit.

### 3.2.5 Esprit ouvert au changement

L'outil principal du Product Owner, le *backlog* de produit, est vivant et évolue pendant toute la vie du projet. Cela constitue une différence fondamentale avec la pratique souvent ancrée dans la culture des organisations qui consiste à essayer de figer les spécifications au début.

---

1. Cette pratique est détaillée dans le chapitre 13 *De la vision aux stories*.

Un bon Product Owner s'adapte et prend en compte le *feedback* qui remonte suite aux livraisons régulières des versions du produit. Ajuster la vision et mettre à jour le *backlog* sont la preuve de son ouverture au changement. Il va même jusqu'à solliciter des idées de changement auprès des utilisateurs et de l'équipe.

**Attention :** il ne faut pas comprendre la capacité au changement comme le droit de changer d'avis à tout moment. Un Product Owner garde le cap : sa vision ne varie pas fondamentalement pendant le développement.

Ouvert au changement, le Product Owner ne doit cependant pas être une girouette.

### 3.2.6 Aptitude à la négociation

Le Product Owner décide des priorités en fonction de la valeur ajoutée, cependant ce n'est pas le seul paramètre à prendre en compte. En particulier au début du projet, il doit tenir compte des risques techniques. Un dialogue avec l'équipe est nécessaire pour pondérer les critères de choix lors de l'établissement des priorités.

Au début d'un développement, si l'équipe n'est pas encore constituée, la discussion se fera avec la personne en charge de l'architecture. Cette concertation préalable permet au Product Owner d'être mieux armé pour définir les priorités.

Lors des séances d'estimation et de planification qu'il effectue en compagnie de l'équipe, le Product Owner doit faire preuve de pédagogie pour expliquer ce qu'il propose d'ajouter au produit et convaincre l'équipe que les priorités qu'il propose sont les bonnes.

Pendant le développement, il est souhaitable que le Product Owner rencontre souvent le reste de l'équipe. Il vient normalement au *scrum* quotidien et c'est une bonne habitude qu'il reste discuter avec les développeurs à la suite de cette réunion. C'est l'occasion d'écouter leurs propositions sur le produit et ils ont souvent de très bonnes idées.

Il peut se mettre en binôme avec un développeur, pour que celui-ci lui montre sur son environnement de développement le fonctionnement d'une story.

Le Product Owner négocie fréquemment avec l'équipe et doit faire preuve de force de conviction auprès des développeurs pour justifier ses prises de position.

## 3.3 CHOISIR LE PRODUCT OWNER D'UNE ÉQUIPE

Au moment où se met en place la première expérimentation de Scrum, il n'existe pas de Product Owner dans l'organisation. Il faut donc trouver la bonne personne pour tenir le rôle. Les organisations étant extrêmement diverses, les possibilités de choix sont très variables.

On peut se baser sur les compétences souhaitées du Product Owner présentées plus haut. Seulement, il n'est pas évident de trouver quelqu'un qui possède toutes ces qualités, et même si on le trouve, il n'est pas forcément disponible pour tenir le rôle sur un projet.

### 3.3.1 Une personne disponible

En effet, sa disponibilité pour le rôle est une condition *sine qua non* pour qu'une personne devienne Product Owner. On considère généralement que le travail, pour une équipe de sept à dix personnes, nécessite une affectation à plein-temps.

#### Disponibilité continue

Le Product Owner fait partie de l'équipe et participe aux réunions du cérémonial. Par opposition à ce qui se passe pour un projet classique, où l'intervention du client (ou de son représentant) est importante au début et à la fin, l'implication du Product Owner est continue sur tous les *sprints*.

On peut même estimer que, plus la fin de la *release* se rapproche, plus le temps que le Product Owner devrait consacrer au projet augmente : en effet, les priorités deviennent plus difficiles à décider, les validations plus nombreuses à devoir être faites...

#### Participation au cérémonial Scrum

Le Product Owner est un membre de l'équipe à part entière, il est tenu de participer aux réunions Scrum.

Le constat fait sur le terrain est que cette règle, pourtant essentielle, n'est pas toujours respectée. Parfois au moment de choisir le Product Owner, il arrive qu'on me demande quelle est son implication minimale : comme la personne susceptible de tenir le rôle a d'autres activités (par exemple, il est également utilisateur de l'ancienne version du produit), on cherche, à tort, à limiter le temps qu'il passera sur le projet, en particulier dans les réunions avec l'équipe.

Pour un projet avec des *sprints* de deux semaines, voilà un exemple de la participation d'un Product Owner aux réunions Scrum que j'ai constatée :

- réunion de planification de *sprint* : environ deux heures ;
- scrums quotidiens, deux fois par semaine (c'est un minimum) : 60 minutes pour quatre séances ;
- revue de *sprint* : 60 minutes ;
- rétrospective : 30 minutes.

Dans ce cas optimiste, cela fait quatre heures et demie, soit un peu plus de 5 % de son temps. Cela peut varier entre 5 et 10 %.

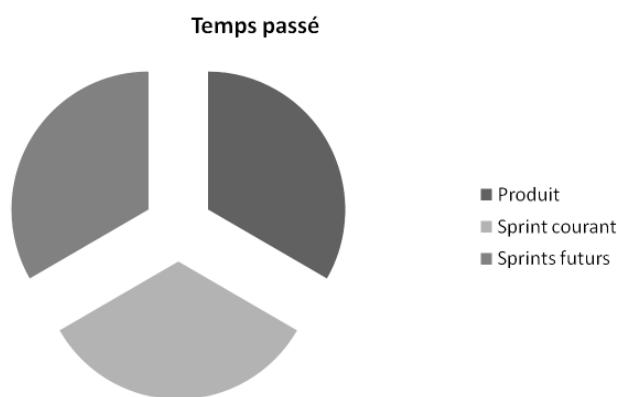
#### Implication au jour le jour

En plus de participer à ces réunions, le Product Owner se doit aussi de :

- mettre à jour le *backlog* de produit, ajuster les priorités,
- répondre aux questions sur le produit,

- définir les tests d'acceptation,
- passer ces tests ou s'assurer qu'ils sont bien passés.

À titre d'exemple, voici quelques indications chiffrées sur le temps que passe un Product Owner sur un produit, chez un éditeur de logiciel : il est environ le tiers de son temps avec les utilisateurs ou le management produit, à collecter des informations ou donner des explications, et le reste sur le projet, à participer aux travaux de l'équipe, dont la moitié pour le travail sur le *sprint* courant et l'autre moitié pour anticiper les *sprints* suivants.



**Figure 3.4** — Répartition typique des activités d'un Product Owner

### 3.3.2 Une seule personne

#### Parler d'une seule voix

L'histoire qui suit illustre l'intérêt d'avoir un seul interlocuteur face à l'équipe de développement pour les questions qui relèvent de ce que doit faire le produit.

Une équipe Scrum avait demandé à me rencontrer pour me poser des questions sur le produit, après une réunion qu'ils avaient eue avec leur Product Owner, qui les avait envoyés vers moi. Il se trouve que je connaissais bien le domaine pour avoir été le Product Owner sur un développement précédent du produit. J'ai donc rencontré l'équipe et répondu à leurs questions qui étaient nombreuses et portaient sur des *user stories* à propos desquelles ils m'ont demandé des précisions. Les réponses que j'ai données étaient orientées selon mes idées sur le produit. Seulement leur Product Owner leur avait demandé de me voir pour une seule question, bien précise. Je ne l'ai su qu'après, l'équipe ne me l'a pas dit et m'a posé plein d'autres questions. Résultat : j'ai orienté l'équipe dans une voie qui n'était pas celle de son Product Owner et celui-ci a eu ensuite beaucoup de mal à s'affirmer et à mettre en avant sa vision.

Il faut parler d'une seule voix à une équipe et cette voix, c'est celle du Product Owner de l'équipe. S'il décide de faire intervenir d'autres personnes sur des domaines complémentaires aux siens, il vaut mieux qu'il soit présent aussi, sinon il lui sera difficile de s'approprier le produit.

### ***Ne pas déléguer trop de responsabilités***

Comme il est difficile de trouver toutes les qualités attendues dans une seule personne, peut-on les partager entre plusieurs ? Si le Product Owner n'est pas suffisamment disponible pour assumer ses responsabilités, est-il alors envisageable d'avoir dans l'équipe un Product Owner délégué ?

**Exemple :** dans le cas d'une société de services qui réalise un produit pour un client, le Product Owner est normalement dans l'organisation du client. Si l'équipe considère qu'il n'est pas suffisamment présent, elle sera tentée de désigner un Product Owner délégué (proxy) en son sein.

Cela peut fonctionner un temps, mais le risque dans ce cas est de soumettre l'équipe à deux sons de cloche et de la perturber quand le vrai Product Owner s'exprime après son délégué. Et surtout c'est contraire à une pratique essentielle des méthodes agiles : une équipe complète inclut le représentant des utilisateurs ou clients.

L'idée d'un Product Owner délégué n'est qu'un pis-aller qu'il est préférable d'éviter. Si le Product Owner désigné ne s'implique pas assez, mieux vaut avoir une approche radicale : arrêter le développement en attendant qu'une bonne solution soit trouvée, comme trouver un véritable remplaçant.

### ***Un groupe de PO doit avoir un responsable***

Sur de gros projets, peut-on envisager d'avoir plusieurs Product Owners, voire une équipe de PO ?

Dans de nombreuses organisations, il semble difficile de trouver une personne suffisamment disponible pour être Product Owner. Apparemment il serait plus facile d'en trouver plusieurs à temps partiel pour se partager le rôle. Il y a aussi des cas où la connaissance est disséminée entre plusieurs personnes. C'est particulièrement vrai dans les grands projets ou lorsque le produit fait partie d'une ligne de produits. Il est alors tentant de constituer un groupe, qui pourrait être appelé **groupe de Product Owners**.

Le problème avec un groupe de Product Owners, c'est que la responsabilité ne réside plus dans une seule personne. Or c'est justement pour éviter les inconvénients dus aux conflits à cause d'intérêts contradictoires, aux décisions ralenties, aux difficultés à prioriser, que Scrum recommande une seule personne pour le rôle de Product Owner. Comme le dit Ken Schwaber : « *Le Product Owner est une personne, pas un comité.* »

Comment avoir une équipe de Product Owners tout en suivant ce conseil ? En ayant dans le groupe une personne étant le **Super Product Owner**, qui est là pour

donner la vision globale et pour arbitrer. Une équipe Scrum possède **un et un seul Product Owner**.

### 3.3.3 Où le trouver dans l'organisation actuelle ?

Le Product Owner est à chercher dans un service qui est en contact avec les utilisateurs ou qui les représente, donc généralement une entité autre que celle des membres de l'équipe de développement.

Par exemple, chez un éditeur de logiciel, il est probable que le Product Owner soit issu du service marketing ou produit. Venant d'une entité où l'idée de produit est déjà très forte, il y a de bonnes chances que cette personne possède déjà quelques compétences requises pour être un bon Product Owner.

C'est dans ce cadre que j'ai rencontré les meilleurs Product Owners, avec une bonne culture produit.

Dans les grandes entreprises qui ont des utilisateurs internes, il paraît logique qu'un Product Owner soit choisi dans le service où sont les utilisateurs et leurs représentants. Quelqu'un qui a été analyste métier (*Business Analyst*), en assistance à ce qu'on appelle la maîtrise d'ouvrage (AMOA), ou chef de projet utilisateurs est un bon candidat pour ce rôle.

Dans ce type d'organisation j'ai rencontré des Product Owners qui avaient plus de difficultés à bien jouer leur rôle, notamment par manque de disponibilité.

### 3.3.4 Une personne motivée pour le rôle

Dans une équipe Scrum, le Product Owner est souvent choisi après le ScrumMaster, avec une culture de Scrum et de l'agilité parfois superficielle. Il pourra recevoir des compléments de formation plus tard, mais ce qui importe lors du choix, c'est son adhésion aux valeurs et principes véhiculés par le mouvement agile.

On ne peut pas être un bon Product Owner si on n'est pas motivé pour le rôle.

## 3.4 CONSEILS POUR PROGRESSER DANS LE RÔLE

Le rôle de Product Owner est probablement celui qui est le plus difficile à tenir dans une équipe. Les projets Scrum qui sont en échec le sont souvent à cause du manque de savoir-faire du Product Owner. Une des raisons (à mon avis la principale) pour expliquer pour ces échecs est le manque d'anticipation sur les sprints suivants.

Voici quelques pistes pour progresser dans le rôle de Product Owner et éviter des échecs.

### **Se former au rôle de Product Owner**

Devenir un bon Product Owner nécessite des compétences particulières qu'une formation spécifique au rôle peut aider à acquérir. En particulier ceux qui ont appris Scrum sur le tas peuvent avoir besoin d'être recadrés sur les fondamentaux.

Une technique très utile au Product Owner pour décrire les éléments du *backlog* est celle des *user stories*. Elle facilite la gestion de projet et permet d'avoir des tests d'acceptation tracés par rapport aux éléments du *backlog* de produit. En complément, l'acquisition des pratiques de définition de produit, incluant la vision, lui permettra d'appréhender une approche descendante, utile pour le développement d'un nouveau produit. Une bonne formation doit inclure l'apprentissage de ces techniques, de préférence avec des ateliers de travail en groupe.

Une fois formé, le Product Owner comprendra pourquoi son rôle demande une grande disponibilité. Il aura plus d'arguments pour obtenir cette disponibilité.

Pour apprendre concrètement le rôle, en particulier dans les organisations où le clivage entre utilisateurs et informaticiens est fort (de type MOA/MOE), il est encore plus efficace d'être accompagné dans sa mise en œuvre sur un projet. Un coach externe apporte sa connaissance et son expérience du rôle ce qui permet de progresser rapidement grâce à un travail en binôme.

### **S'impliquer dans les tests d'acceptation**

C'est la responsabilité du Product Owner de s'assurer que le travail fait par l'équipe est fini à la fin d'un *sprint*.

C'est en exposant ses conditions de satisfaction que le Product Owner formule à l'équipe le comportement qu'il attend d'une fonctionnalité. C'est en vérifiant que ces conditions sont bien remplies qu'il peut s'assurer de leur bonne fin. Ces conditions peuvent être exprimées de manière informelle, ce qui est déjà un moyen pour améliorer la communication ; une approche plus aboutie est de les formaliser en tests d'acceptation : l'intérêt est que ces tests pourront être automatisés et passés régulièrement.

Cette transformation de conditions métier dans le langage des utilisateurs en tests exécutables constitue l'approche appelée ATDD (*Acceptance Test Driven Development*, pilotage par les tests d'acceptation<sup>1</sup>), qui est porteuse de gains de productivité et de qualité. Cette approche repose sur des langages et des outils dans lesquels un Product Owner doit s'impliquer.

### **Collaborer avec l'équipe**

Le Product Owner n'est pas simplement le représentant des clients et utilisateurs. Il fait partie de l'équipe et participe activement aux travaux pendant un *sprint*. Sa présence physique régulière avec l'équipe a un impact sur son rôle :

---

1. Le test d'acceptation est présenté dans le chapitre 14 *De la story aux tests*.

- S'il est installé avec l'équipe, dans le même espace de travail, c'est la situation idéale. Cependant il suffit qu'il soit assez proche pour que l'équipe puisse aller le voir et lui demander des précisions, et dans l'autre sens, qu'il puisse rencontrer facilement l'équipe. Il peut s'asseoir de temps en temps à côté d'un membre de l'équipe pour rentrer dans les détails d'une story.
- Si le Product Owner est à distance et ne voit pas physiquement l'équipe, la mise en œuvre du rôle sera plus délicate, mais il existe des solutions, notamment avec des outils de travail collaboratif en ligne, pour pallier son manque de présence physique.

En collaborant avec l'équipe, le Product Owner apprendra à écouter ce qu'ont à dire les développeurs. Il comprendra mieux leurs préoccupations de délai et de qualité. Ainsi il sera moins sujet à leur imposer une pression négative en les poussant à livrer toujours plus de fonctionnalités. De plus, le travail en commun permet à la créativité des développeurs de se manifester.

### *Utiliser le produit*

Un bon Product Owner aime son produit. S'il l'utilise tous les jours, il y a des chances qu'il soit incité à faire en sorte qu'il soit de qualité et qu'il discerne bien la valeur ajoutée par les fonctions dans le *backlog*. Bien connaître son produit lui donnera une position respectée par tous et facilitera ses prises de décision.

C'est aussi la fonction du Product Owner de faire des démonstrations à l'extérieur et de présenter le produit aux utilisateurs.

### *Impliquer les parties prenantes*

Le Product Owner est un représentant : pour bien exercer son rôle, il doit communiquer avec ceux qu'il représente.

Il représente non seulement les utilisateurs du produit final, mais aussi le client ou sponsor et les autres personnes impliquées dans le produit. Il les invite à la revue de chaque fin de *sprint*. Il doit vraiment les inciter à y venir car cette revue est un moment d'inspection collectif de l'état du produit.

En complément, le Product Owner peut organiser des séances de travail pour préparer la feuille de route (*roadmap*) du produit portant sur les futures versions.

### *Planifier à moyen terme*

Une des pratiques de Scrum les moins utilisées, d'après les sondages (et mon expérience), est la production d'un plan de *release*. C'est pourtant un moyen de donner une direction à l'équipe et de la communiquer aux parties prenantes. Le Product Owner est moteur dans cette quête d'avoir un horizon qui va au-delà du *sprint* courant.

### Utiliser un outil pour gérer le backlog

L'outil du Product Owner est le *backlog* de produit. Un *backlog* est une liste qui peut être gérée de multiples façons. Cependant, au bout d'un peu de pratique Scrum, il devient nécessaire de disposer d'un outil<sup>1</sup> pour gérer la traçabilité, être assisté dans le suivi des éléments du *backlog*, faire des plans et produire des graphiques.

### En résumé

Dans une équipe Scrum qui développe un produit, le Product Owner est la personne qui représente le « métier ». Il apporte sa vision à l'équipe et définit les priorités de façon à obtenir un produit ayant le maximum de valeur.

L'implication d'un bon Product Owner est capitale pour assurer le succès du projet. En définissant sa vision sur le produit, il donne l'impulsion à l'équipe. En faisant la promotion du résultat de chaque *sprint* auprès des utilisateurs, il fournit à l'équipe une reconnaissance qui la motive. En collaborant avec l'équipe, il fait converger les énergies pour maximiser la valeur ajoutée au produit.

---

1. Les outils sont présentés dans le chapitre 17.



# 4

## Le ScrumMaster et l'équipe

Lorsqu'on évoque un projet développé par un groupe de personnes, une pensée très répandue est de considérer qu'une personne identifiée doit être responsable de l'équipe. Traditionnellement, ce rôle est appelé **chef de projet**. En France, le rôle de chef de projet est solidement ancré dans la culture du développement. En voici deux exemples :

- Certains étudiants en informatique, passant un entretien pour rentrer dans une école, mettent un point d'honneur à dire que leur objectif est de devenir chef de projet dès leur entrée dans la vie professionnelle. Probablement parce que des enseignants croyant bien faire leur ont inculqué cette notion de l'ambition. En tant que membre du jury devant lequel ils passent, je considère pourtant qu'une meilleure ambition serait de mettre en avant leur capacité à travailler en équipe.
- Récemment, au cours d'une présentation de Scrum dans une grande entreprise publique, tous les participants sauf un se sont présentés, lors du tour de table, comme chefs de projet. Souvent dans les entreprises qui font appel à la délégation de personnel, il ne reste que des chefs de projet dans l'organisation. J'ai jeté un froid quand, en décrivant les rôles dans une équipe Scrum, j'ai annoncé l'absence de chef de projet.

**Pas de chef de projet dans Scrum !** Le rôle est éliminé.

Le travail et les responsabilités d'un chef de projet ne disparaissent pas pour autant dans les projets Scrum. Une grande partie est dévolue au Product Owner, une autre partie est laissée à l'équipe. Un des principes de Scrum est l'auto-organisation : il signifie que les membres de l'équipe s'organisent eux-mêmes et n'ont pas besoin d'un

chef qui leur assigne le travail à faire. **ScrumMaster** n'est donc pas un nouveau nom pour chef de projet.

Le **ScrumMaster** a pour responsabilité essentielle d'aider l'équipe à appliquer Scrum et à l'adapter au contexte. Il a une grande influence sur la façon de travailler, sur le **processus**, comme le Product Owner en a une sur le produit. Ce qui pourrait conduire à qualifier le ScrumMaster de Process Owner par équivalence.

C'est de ce rôle emblématique de Scrum, dont il est question dans ce chapitre. Nous y aborderons aussi le rôle de l'équipe Scrum, dont le comportement est fortement influencé par ce qu'est un ScrumMaster.

### Définition

**ScrumMaster**, c'est un nom original, voire décalé. Même pour un anglophone. Scrum signifie mêlée au rugby, donc littéralement ScrumMaster signifie le maître de la mêlée. C'est effectivement très original, mais cela n'aide pas beaucoup à comprendre le rôle. Je crois que personne n'a essayé d'adopter une traduction française pour ScrumMaster. L'usage est de conserver le terme dans les pays francophones. Cela a l'avantage de bien montrer que c'est un rôle nouveau qui implique du changement par rapport aux pratiques habituelles de gestion d'équipe.

On trouve de nombreuses analogies pour expliquer le rôle de ScrumMaster, en voici deux :

- Pour rester dans le rugby, Scrum se réfère aux huit membres du pack dans le rugby à quinze. Le poste de demi de mêlée est celui qui se rapproche le plus de l'idée de ScrumMaster. C'est lui qui fait avancer son pack, le guide dans la progression, demande le ballon au bon moment.
- Ken Schwaber, un des auteurs de Scrum, compare le ScrumMaster à un chien de berger qui veille sur son troupeau de moutons. Certains considèrent que le berger constituerait une meilleure analogie pour le rôle de ScrumMaster.

## 4.1 RESPONSABILITÉS

### 4.1.1 Responsabilités du ScrumMaster

Ses responsabilités générales sont les suivantes :

- veiller à la mise en application de Scrum, par exemple en faisant en sorte que les différentes réunions aient lieu et qu'elles se fassent dans le respect des règles ;
- encourager l'équipe à apprendre, et à progresser, en fonction du contexte du projet, dans les pratiques d'ingénierie ;
- faire en sorte d'éliminer les obstacles qui pourraient freiner l'avancement, par exemple en protégeant l'équipe des interférences extérieures pendant le déroulement d'un *sprint* ;
- inciter l'équipe à devenir autonome.

Le développement avec Scrum se fait par une succession de *sprints*. Les responsabilités du ScrumMaster s'appliquent pendant les *sprints* mais commencent à s'exercer même avant le lancement du premier.

### *Avant le premier sprint*

Le ScrumMaster est souvent la première personne désignée dans l'équipe. Il peut donc être impliqué dans la constitution de l'équipe. Il arrive qu'il participe au choix, toujours délicat, du Product Owner.

C'est également à lui qu'échoira le plus souvent la responsabilité de s'assurer que la logistique, en particulier les bureaux et leur agencement, est adaptée aux pratiques de travail en équipe.

### *Tâches périodiques pendant les sprints*

Le ScrumMaster met Scrum en application. Il organise et anime les réunions qui constituent le cérémonial d'un *sprint*. Il fait en sorte que ces réunions aient lieu et qu'elles soient efficaces. Il y joue un rôle de facilitateur, littéralement « celui qui facilite les choses ».

### *Élimination des obstacles*

Scrum est un processus empirique et à part le rituel des réunions, l'enchaînement des tâches n'est pas défini à l'avance dans un *sprint* ; mais c'est l'équipe, et non pas le ScrumMaster qui le décidera de façon opportuniste, en fonction de l'avancement.

En plus de cet indéterminisme sur le déroulement des tâches, il se produit toujours des événements pendant un développement. Parmi ces événements, certains sont susceptibles de ralentir ou de bloquer le travail de l'équipe. Dans le jargon Scrum, ils sont appelés des **obstacles** (*impediments*). Les obstacles peuvent être de nature et d'importance très variables.

**Exemple :** un développeur se casse le bras au ski, le serveur SVN est en panne, le composant attendu d'une autre équipe n'est pas prêt...

C'est au ScrumMaster d'identifier les obstacles mis en évidence par l'application de Scrum et de s'assurer de leur élimination.

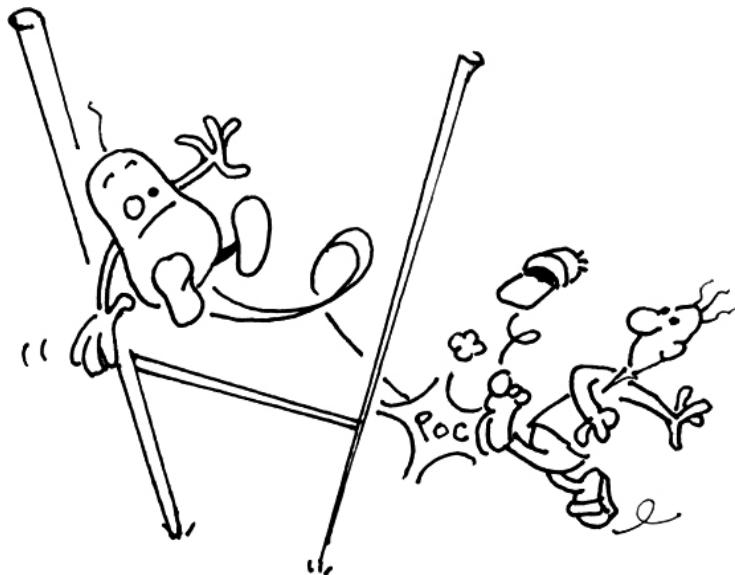
C'est une activité curative : le ScrumMaster fait tout pour éviter qu'ils ralentissent durablement l'équipe. Il s'appuie sur des compétences internes à l'équipe ou va chercher des compétences externes si c'est nécessaire pour solutionner un problème.

Le ScrumMaster s'occupe également du préventif : certains événements ne constituent pas des obstacles mais pourraient le devenir. Il doit s'efforcer de protéger l'équipe, en différant les impacts négatifs de futurs événements sur sa capacité à produire.

À travers la description de ses responsabilités, on voit que le rôle tranche avec l'idée qu'on se fait d'un chef omnipotent. Une différence marquante est que ce n'est pas lui qui organise le travail des membres de l'équipe.

### 4.1.2 Responsabilités de l'équipe

Une des missions du ScrumMaster est de motiver l'équipe pour qu'elle devienne autonome et responsable.



**Figure 4.1** — Un ScrumMaster énergique donne l'impulsion à un membre de l'équipe

Le rôle de l'équipe est essentiel : c'est elle qui va réaliser le produit, en développant un incrément à chaque *sprint*. Elle est investie avec le pouvoir et l'autorité pour le faire de la façon la plus efficace. Pour cela, elle s'organise elle-même et doit avoir toutes les compétences nécessaires au développement du produit. Il n'y a pas de rôles spécialisés dans une équipe, elle doit posséder les compétences nécessaires pour accomplir le travail à faire : on dit qu'elle est multifonctionnelle.

C'est l'équipe qui définit elle-même la façon dont elle organise ses travaux, ce n'est pas le ScrumMaster (ni le Product Owner). Chaque membre de l'équipe apporte son expertise, la synergie améliorant l'efficacité globale. Les responsabilités de l'équipe sont donc très importantes. L'origine du nom Scrum, le rugby, rappelle d'ailleurs le rôle central de l'équipe.

## 4.2 COMPÉTENCES SOUHAITÉES DU SCRUMMASTER

La façon dont le rôle de ScrumMaster est joué dépend de la taille de l'équipe, de la complexité technique du produit à réaliser, ainsi que de l'importance du changement induit par le passage à Scrum dans l'organisation. Cependant, on peut dégager les compétences attendues d'un ScrumMaster (figure 4.2).

Bonne connaissance de Scrum

Aptitude à comprendre les aspects techniques

Facilité à communiquer

Capacité à guider

Talent de médiateur

Ténacité

Inclination à la transparence

Goût à être au service de l'équipe

**Figure 4.2** — Les compétences souhaitées d'un ScrumMaster

### 4.2.1 Bonne connaissance de Scrum

S'il y a une personne qui doit bien maîtriser Scrum dans une équipe, c'est le ScrumMaster. Au-delà de la simple connaissance « livresque » de Scrum, il est préférable qu'il ait déjà une expérience de sa mise en œuvre, pour éviter d'appliquer des règles sans discernement, car il est toujours nécessaire d'adapter Scrum au contexte.

Sa connaissance ne doit pas s'arrêter à son rôle, mais englober l'ensemble du cadre Scrum. En particulier, il devra s'intéresser aux valeurs et aux principes de Scrum pour être capable de les promouvoir auprès de l'équipe.

### 4.2.2 Aptitude à comprendre les aspects techniques

Il n'est pas nécessaire pour un ScrumMaster de bien connaître le domaine de l'application à développer. Une expérience dans le « métier » peut cependant faciliter la communication avec le Product Owner et mieux impliquer l'équipe dans la recherche de la valeur pour le produit.

On ne demande pas non plus à un ScrumMaster d'être un cador en technique. Le ScrumMaster s'appuie sur des experts pour les aspects techniques pointus. Cependant des connaissances dans les technologies utilisées permettent de mieux appréhender les problèmes rencontrés par son équipe. Cela facilite la communication, en particulier avec les geeks qu'on rencontre dans certaines équipes, et rend plus aisée l'identification des obstacles qu'ils rencontrent.

### 4.2.3 Facilité à communiquer

Des talents de communication sont nécessaires. Le ScrumMaster est amené à discuter fréquemment avec l'équipe et aussi avec le management.

Ces discussions ont lieu dans différents contextes, ce qui nécessite de sa part d'adapter le style de communication :

- il doit savoir obtenir la confiance quand il est en face à face avec un membre de l'équipe ;
- il fait en sorte que les réunions du cérémonial, en présence de nombreuses personnes, se déroulent efficacement ;
- il est tenace et ferme dans ses demandes au management, sans pour autant être intransigeant.

#### 4.2.4 Capacité à guider

Il influence l'équipe et c'est un meneur d'hommes qui sait motiver une équipe, pour qu'elle arrive au résultat. Mais il doit arriver à ses fins par la persuasion, sans imposer ses décisions : un ScrumMaster ne dispose d'aucune autorité hiérarchique sur l'équipe.

Pendant un *sprint*, il guide l'équipe vers le respect de l'objectif essentiel, qui est de livrer un produit qui apporte de la valeur avec une utilisation optimale des ressources.

#### 4.2.5 Talent de médiateur

Le travail le plus important du ScrumMaster est d'éliminer les obstacles. Parmi ceux-ci, un certain nombre est dû à des conflits entre personnes.

Lors d'un différend entre des membres de l'équipe, il joue le rôle de médiateur pour aider les personnes concernées à trouver une solution acceptable. Il pousse au consensus. En cas de désaccord persistant, il peut proposer une mesure plus radicale, comme changer une personne d'équipe.

En cas de conflit d'une personne avec le Product Owner, il devra s'efforcer de ne pas prendre systématiquement parti contre le Product Owner : il ne faut pas créer une opposition entre les développeurs et les utilisateurs (un des principes de l'agilité est l'équipe complète, dont le but est d'éviter cette opposition).

J'ai connu un ScrumMaster qui avait mal compris son rôle. Sous prétexte de considérations techniques, il s'opposait au Product Owner, essayant d'empêcher une mise en production. S'il est normal qu'il existe une tension entre les deux rôles, ce n'est pas le ScrumMaster qui est responsable de la vie du produit. Il devrait simplement se limiter à exposer le point de vue de l'équipe.

#### 4.2.6 Ténacité

Le ScrumMaster fait son possible pour éviter que des obstacles aient un impact sur la progression de l'équipe. Parfois, des obstacles ne peuvent être éliminés que par l'intervention de personnes faisant partie d'autres équipes ou par le management. Ces personnes sont souvent difficiles à rencontrer et encore plus à convaincre d'agir rapidement. Un ScrumMaster n'abandonne pas à la première adversité. Il doit se montrer opiniâtre et poursuivre sa quête sans relâche, jusqu'à l'élimination des problèmes qui freinent l'équipe.

### 4.2.7 Inclination à la transparence

Par rapport au rôle de chef de projet, celui de ScrumMaster est davantage sur l'accompagnement de l'équipe que sur le suivi individuel.

Les mesures faites avec Scrum sont collectives. Elles sont nécessaires pour suivre la progression de l'équipe et produire des graphiques, comme les rapports d'avancement. Le ScrumMaster s'assure de leur communication, notamment auprès du management.

Scrum propose un cadre méthodologique qui contribue à une grande transparence. Le ScrumMaster est garant de cette transparence dans l'avancement de l'équipe.

D'ailleurs, un apport fondamental de Scrum est de révéler les dysfonctionnements au plus tôt. La responsabilité du ScrumMaster n'est pas de les masquer, mais au contraire de les mettre en évidence.

### 4.2.8 Goût à être au service de l'équipe

La différence radicale entre le ScrumMaster et le chef de projet est que le ScrumMaster n'est pas un chef : il ne commande pas, il n'impose pas, il ne constraint pas. Au-delà de la différence de nom, il s'agit d'une différence de style.

Il est au service de l'équipe, qu'il soutient dans la résolution des conflits et dont il encourage la prise d'autonomie vers une auto-organisation. Il doit faire preuve d'humilité et ne pas se mettre en avant :

- ce n'est pas lui qui réussit si le projet avance bien : c'est l'équipe,
- ce n'est pas la faute des autres membres de l'équipe si le projet a des difficultés : c'est aussi sa responsabilité.

## 4.3 CHOISIR LE SCRUMMASTER D'UNE ÉQUIPE

Le rôle de ScrumMaster n'existe pas dans les organisations actuelles. Il faut donc trouver la personne qui va le mieux convenir pour jouer ce rôle. De préférence, elle sera choisie à l'aune des compétences souhaitées. Il faut aussi savoir quelle sera son implication pour identifier la bonne ressource.

### 4.3.1 Affectation au rôle

Pour une équipe Scrum typique (de six à dix personnes), une seule personne joue ce rôle sur un projet.

Le ScrumMaster fait partie de l'équipe : il s'engage avec les autres. Il doit régulièrement rencontrer – physiquement – les membres de l'équipe, il ne reste pas dans son bureau.

Dans de petites équipes, le ScrumMaster peut aussi participer aux travaux de développement. Il prend alors des tâches du *sprint* comme les autres membres de

l'équipe, mais cela doit rester limité : le rôle de ScrumMaster prend du temps et il est prioritaire sur ses autres tâches.

En revanche, il vaut mieux éviter qu'une personne soit :

- le ScrumMaster de plusieurs équipes,
- en même temps ScrumMaster et Product Owner de l'équipe.

### 4.3.2 Où trouver la bonne personne ?

Cela dépend de la structure des organisations.

Pour certaines équipes, c'est un développeur expérimenté qui devient le ScrumMaster. Mais dans la majorité de celles pour lesquelles j'ai travaillé, c'est un ancien chef de projet qui a pris le rôle. Par exemple, dans les grandes organisations, un chef de projet informatique (CPI) devient naturellement ScrumMaster.

Dans le cas d'organisation à culture hiérarchique forte, le rôle de ScrumMaster impacte les fondements de la gouvernance : Scrum représente un changement radical.

#### Théorie X et Y<sup>1</sup>

La théorie X induit un cercle vicieux dans lequel l'organisation est construite sur des règles strictes et des contrôles sévères : les employés s'adaptent en choisissant de travailler au minimum, et en adoptant une attitude passive. Ils fuient alors les responsabilités puisque le système est répressif, et donc non sécurisant pour les prises de risque.

Ceci conforte les dirigeants dans leurs convictions, ce qui les incite à renforcer les règles et les contrôles.

Au contraire, la théorie Y introduit un système vertueux dans lequel l'organisation est construite autour de principes de confiance, de délégation et d'autocontrôle : les employés utilisent cette liberté supplémentaire pour mieux s'impliquer dans le travail. Ils prennent alors des initiatives, acceptent les responsabilités et vont même jusqu'à les rechercher.

Ceci conforte les dirigeants dans leurs convictions, ce qui les incite à maintenir la confiance, la délégation et l'autocontrôle.

Le rôle de ScrumMaster s'inscrit dans la ligne Y. On comprend que les organisations où la gouvernance a développé les présupposés de la théorie X auront un peu plus de mal à devenir agiles. Cela ne veut pas dire que ce n'est pas possible, ce sera plus long et plus difficile puisqu'il faudra faire évoluer les cultures. Parce que vouloir être agile en gardant une gouvernance proche de la théorie X, c'est contradictoire.

1. [http://fr.wikipedia.org/wiki/Th%C3%A9orie\\_X\\_et\\_th%C3%A9orie\\_Y](http://fr.wikipedia.org/wiki/Th%C3%A9orie_X_et_th%C3%A9orie_Y)

Dans une organisation avec une gouvernance forte, l'existence simultanée des deux rôles (ScrumMaster et chef de projet) est temporairement possible. De nombreuses activités habituelles de gestion de projet comme le *reporting*, la participation au comité de pilotage, le budget et la gestion des ressources humaines sont faites par chef de projet tandis que le ScrumMaster se consacre uniquement à l'animation de l'équipe. Cependant cette cohabitation ne peut être que provisoire et uniquement pour faciliter la transition d'une organisation à Scrum.

### 4.3.3 Quelqu'un qui incarne le changement

Le terme ScrumMaster est sujet à caution, dans sa partie *master* (maître). Le langage influence le comportement : même si l'appellation ScrumMaster est nouvelle, le terme *master* n'aide pas toujours les organisations à changer de paradigme durablement.

Dans certaines organisations à culture hiérarchique, le rôle de ScrumMaster, *maître de Scrum* peut être perçu comme un rôle de responsable, dirigeant des personnes.

J'ai aussi vu des ScrumMasters retombant dans les habitudes du chef de projet traditionnel, avec des équipes acceptant cet état de fait, parce que c'est leur façon habituelle de travailler.

C'est pourquoi la personne devenant ScrumMaster doit avoir bien compris l'essence du rôle pour être l'incarnation du changement qu'il représente.

### 4.3.4 ScrumMaster, un état d'esprit

La bonne personne pour jouer le rôle a un état d'esprit approprié.

Il m'est arrivé de rencontrer des ScrumMasters naturels. Ceux dont on se dit, comme pour Obélix : ils sont tombés dedans quand ils étaient petits.

Quelques traits permettent de déceler ces ScrumMasters en puissance :

- la capacité à percevoir les émotions dans l'équipe,
- la curiosité et l'envie d'apprendre,
- l'inclination à penser que les gens font de leur mieux dans leur travail,
- l'envie de changer les choses qui ne vont pas bien,
- l'orientation vers le collectif,
- la prise de risques.

### 4.3.5 Rotation dans le rôle

Dans une équipe, la personne qui joue le rôle de ScrumMaster peut tourner : à chaque *sprint* ou au bout de quelques *sprints*, c'est une autre personne qui devient ScrumMaster.

Cela se produit dans les projets que j'encadre avec des étudiants. Évidemment tous les membres d'une équipe d'étudiants sont dans la même classe et ont, a priori, la même expérience. Aucun d'entre eux n'a jamais été ScrumMaster auparavant, ni chef de projet d'ailleurs. Le choix du ScrumMaster est fait par l'équipe, les enseignants n'interviennent pas. Lorsque le projet avance, je propose, si l'équipe ne le demande pas elle-même, que ce rôle soit affecté à tour de rôle. Le choix est laissé à l'appréciation de l'équipe. Bon an mal an, il y a environ la moitié des équipes qui pratiquent la rotation de ScrumMaster. Dans les autres équipes, on préfère garder le ScrumMaster actuel, probablement parce qu'il a dégagé une autorité naturelle et des aptitudes pour ce rôle.

ScrumMaster est un rôle dynamique, dans la mesure où la personne affectée à ce rôle peut varier dans le temps. Cela permet de réagir si la personne qui tient le rôle a des difficultés.

## 4.4 CONSEILS POUR PROGRESSER DANS LE RÔLE

Le rôle de ScrumMaster est difficile à tenir quand on débute dans Scrum. Des difficultés peuvent apparaître à cause du ScrumMaster qui remplit mal son rôle, par exemple s'il ne fait pas confiance aux membres de l'équipe et décide à sa place.

Pour éviter de connaître des échecs dans vos projets, voici quelques pistes pour progresser dans le rôle de ScrumMaster.

#### *Parfaire sa connaissance de Scrum*

Être un bon ScrumMaster nécessite une culture agile et une maîtrise de Scrum. Cela s'apprend d'abord en appliquant bien sûr, mais aussi en lisant des livres ou des articles. La participation à des conférences où sont présentés des retours d'expérience est particulièrement enrichissante. Il existe aussi des groupes d'utilisateurs, comme le *Scrum User Group français*<sup>1</sup>.

#### *Se former au rôle de ScrumMaster*

Au-delà de la simple connaissance de Scrum, devenir un bon ScrumMaster nécessite des compétences particulières qu'une formation spécifique peut aider à acquérir. En particulier ceux qui n'ont pas suivi une formation au départ et ont appris leur rôle sur le tas peuvent avoir besoin d'être recadrés, après quelques mois de pratique.

1. Pour en savoir plus : [www.frenchsug.org](http://www.frenchsug.org).

Dans certaines sociétés, généralement des petites, la personne qui devient ScrumMaster était située, dans la hiérarchie, sous l'autorité de celle qui est le Product Owner. Une bonne connaissance du rôle lui permettra de s'affirmer, ce qui aura pour effet de limiter un pouvoir excessif du Product Owner.

### ***Se faire assister par un coach***

Pour apprendre le rôle, la meilleure solution est d'être accompagné dans sa mise en œuvre sur le projet par un expert. C'est particulièrement important pour de grandes organisations dans lesquelles la culture traditionnelle des projets est fortement marquée. Certaines de ces organisations semblent résister de façon coriace au changement et le *coaching* des ScrumMasters y est indispensable dans les premières expériences de Scrum.

### ***Favoriser l'analyse des causes des problèmes***

Lorsqu'un obstacle est détecté, l'identification de ses causes profondes permet d'éviter qu'il ne se reproduise. En favorisant l'analyse causale, un ScrumMaster permettra à l'équipe de progresser dans ses pratiques, grâce à la capitalisation faite de façon collective.

### ***Pratiquer l'art du possible***

Le ScrumMaster a pour mission de faire appliquer Scrum, mais une posture trop radicale peut conduire au rejet de Scrum. Il doit tenir compte du contexte de l'organisation. En particulier, dans sa responsabilité de protéger l'équipe des perturbations, le ScrumMaster doit savoir jusqu'où il est possible d'aller, face à une organisation qui n'arrive pas à changer ses habitudes rapidement. Comme le dit Ken Schwaber : « *un ScrumMaster mort<sup>1</sup> n'est plus utile* ».

### ***Favoriser l'auto-organisation de l'équipe***

Le ScrumMaster fait tout pour que l'équipe prenne de l'autonomie, il l'aide à apprendre Scrum et à mettre en place les meilleures pratiques d'ingénierie. Cela demande beaucoup d'investissement, en particulier au début, et surtout si l'équipe connaît mal Scrum. Mais si cela réussit l'équipe s'auto-organise et a moins besoin de lui.

À l'extrême, alors qu'il est impossible de se passer d'un Product Owner, il est envisageable d'avoir une équipe Scrum sans ScrumMaster, dans certains cas bien particuliers.

**Attention :** il faut se rappeler que ScrumMaster est un rôle à plein-temps pour une équipe standard qui démarre avec Scrum. Les situations présentées ci-après représentent des cas particuliers.

- **Très petite équipe** – Dans le cas d'une toute petite équipe, jusqu'à trois personnes, on peut se passer d'un ScrumMaster.

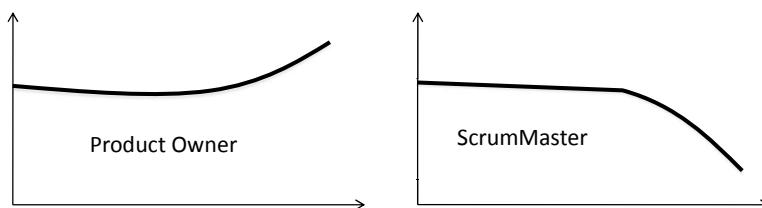
---

1. En fait dans *Agile Project Management with Scrum*, il dit (page 33) qu'un chien de berger mort est un chien de berger inutile.

C'est ce qui s'est passé pendant certaines périodes sur le projet IceScrum pour lequel je suis le Product Owner. Il est arrivé que l'équipe ne compte, pendant quelques sprints, que deux ou trois personnes, en plus de moi. Il s'est avéré inutile de désigner un ScrumMaster.

C'est vrai que, de plus, le fait de développer un outil dédié Scrum leur procure une bonne connaissance de la méthode, ce qui nous amène à la deuxième situation où une équipe peut se passer de ScrumMaster.

- **Équipe mature avec Scrum** – Une équipe expérimentée qui a acquis un niveau d'auto-organisation très élevé peut fonctionner sans ScrumMaster. Si l'équipe s'organise elle-même, que tout le monde adhère aux principes agiles, le rôle de ScrumMaster devient superflu. La quantité de temps disponible nécessaire d'un ScrumMaster diminue dans le temps, en fait, à l'opposé de ce qui se passe avec le Product Owner, dont l'implication augmente au fur et à mesure de l'avancement du projet.



**Figure 4.3** – Implication comparée du Product Owner et du ScrumMaster

Lorsqu'un ScrumMaster s'aperçoit qu'il n'est plus indispensable, c'est probablement qu'il a réussi.

Comme le dit Charles Piaget dans le film Les LIP : « *un leader sait qu'il a réussi quand on n'a plus besoin de lui ou en tout cas quand sa voix ne compte que pour un, comme celle de tout le monde dans le groupe*<sup>1</sup>. » C'est sûrement plus facile à mettre en place dans le développement de logiciel que dans la production de montres.

C'est paradoxal : un ScrumMaster qui a réussi devient inutile dans son équipe... il a réussi à obtenir une équipe qui n'a plus besoin de lui et doit se saborder. Il peut alors :

- aller coacher une autre équipe,
- redevenir simple développeur s'il était à la fois ScrumMaster et membre actif de l'équipe.

Dans le cas d'une équipe où le ScrumMaster est en même temps développeur, une phase intermédiaire de maturité avant la disparition du rôle est de pratiquer la rotation : le rôle de ScrumMaster tourne dans l'équipe.

1. Dans l'expérience des LIP, cela peut être efficace : en quelques semaines d'imagination au pouvoir, les équipes en autogestion de chez LIP ont produit autant de montres qu'en une année normale de production.

### *Maîtriser le reporting*

La tendance des chefs de projet traditionnels est de faire beaucoup de *reporting*. Avec Scrum, la façon de produire des indicateurs<sup>1</sup> est différente et cela est fait rapidement : pas besoin de passer beaucoup de temps à faire des consolidations. Il y a une forte orientation résultat : l'avancement est fait sur ce qui est réellement fini et visible.

### **En résumé**

Le ScrumMaster ne gère pas des ressources interchangeables, mais les hommes et les femmes de l'équipe. Son rôle essentiel est de les faire progresser collectivement pour la réussite du projet.

Les méthodes agiles reprennent l'idée d'organisation sans hiérarchie autoritaire : on y parle d'équipe investie avec le pouvoir et l'autorité pour faire ce qu'elle a à faire ou qui s'organise par elle-même. C'est une des différences majeures avec les méthodes traditionnelles. Elle est mise en pratique avec le ScrumMaster qui n'est pas un chef mais un facilitateur.

---

1. Voir le chapitre 15 *Estimations, mesures et indicateurs*.



# 5

## Le backlog de produit

Après avoir décidé de lancer le développement d'un produit, la difficulté fondamentale est de transformer l'idée de départ en quelque chose d'utilisable par l'équipe de développement.

Dans les projets traditionnels, cette transformation se fait entièrement au début du projet et se concrétise dans un document, qui décrit ce que va faire le produit, quelles sont ses fonctions et quel est son comportement.

Dans ma carrière, j'ai passé du temps à rédiger de gros documents de spécification<sup>1</sup>. Notamment pour un projet dans les télécoms, j'ai élaboré le document de spécification, appelé à l'époque spécification externe du système. Je faisais de mon mieux pour imaginer le comportement du produit. Comme j'étais un utilisateur potentiel du système à produire, puisqu'il s'agissait de téléphonie, cela facilitait mes réflexions. Je discutais fréquemment avec le chef de produit et le service marketing. Je faisais des versions fréquentes de ce document en essayant d'avoir du *feedback* du marketing. Même s'il était plutôt rare, j'avancais bien dans la rédaction du document.

Cela m'a apporté beaucoup sur la connaissance du produit et j'ai pu transmettre cette connaissance aux développeurs de l'équipe, plus oralement, par des discussions régulières, que par leur lecture du document. Cela a fonctionné : le produit a été développé et, après quelques mois, commençait à fonctionner.

Et puis un jour, alors que nous avions commencé les tests d'intégration, la direction a décidé de constituer une équipe pour les tests fonctionnels du produit. Le nouveau responsable des tests a évidemment voulu avoir accès aux spécifications du produit.

---

1. Il existe différentes appellations de *spécification* dans le cas d'un développement de logiciel : spécification fonctionnelle, spécification externe du logiciel, spécification technique du besoin du logiciel (STBL), en anglais *Software Requirement Specification* (SRS).

Le gros document produit quelques mois auparavant et dont j'étais plutôt fier ne lui a pas vraiment convenu.

C'est vrai qu'il n'était pas facile de rentrer dans les 200 pages pour quelqu'un qui ne connaissait pas bien le domaine. Il fallait aussi faire le tri entre ce qui était dans le document et ce qui était dans le produit à tester, car le développement se faisait en deux incrément et le document portait sur le tout. Et bien sûr, le document n'était plus tout à fait à jour.

Le constat courant dans le domaine du logiciel est qu'un gros document de spécification élaboré au début est difficile à maintenir, qu'il n'est pas très utile pour le développement et encore moins pour les tests.

La démarche est fondamentalement différente avec Scrum.

Une équipe Scrum ne produit pas une documentation faite au début du projet, qui décrit en détail toutes les spécifications fonctionnelles. Elle collecte les fonctions essentielles et les raffine progressivement. L'outil de collecte s'appelle le **backlog** de produit.

#### **Backlog, n'y aurait-il pas un mot pour le dire en français ?**

Avant de connaître le terme **backlog**, j'utilisais la notion de référentiel des exigences. D'ailleurs sur le premier projet que j'ai accompagné dans la transition à Scrum, c'est ce terme qui a été utilisé plutôt que *backlog*. Mais le sens n'est pas le même : littéralement le *backlog* est la liste des choses en attente.

Une tentative de traduction de *backlog* en français a été de l'appeler carnet de produit. Cela n'a pas vraiment percé, même pas au Québec où nos cousins sont pourtant de fervents adeptes de la francisation des mots issus de l'anglais.

L'usage courant de la communauté francophone Scrum est de ne pas traduire *backlog*.

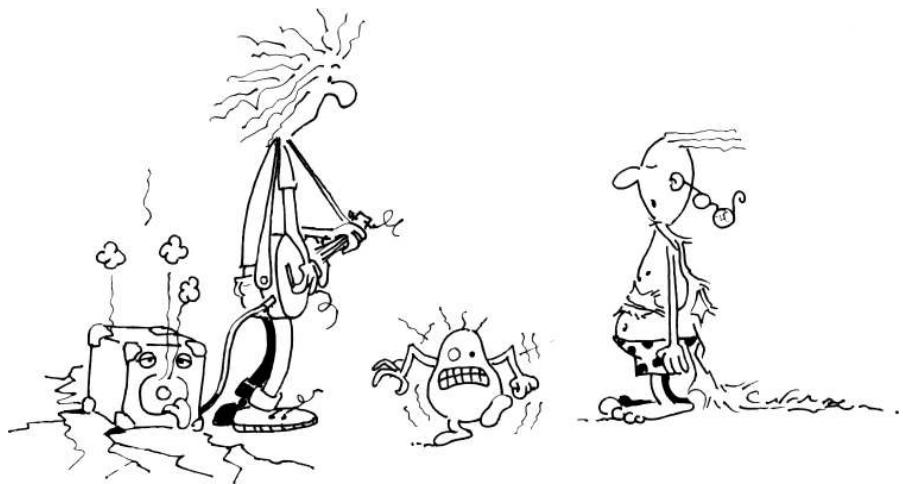
**Attention :** les présentations et articles sur Scrum en anglais présentent deux *backlogs*, le *backlog* de produit et le *backlog* de sprint.

Dans ce chapitre, il n'est question que du *backlog* de produit. Plus généralement dans ce livre, quand j'utilise *backlog*, c'est du *backlog* de produit qu'il s'agit.

D'ailleurs pour éviter les confusions, je ne parle pas de *backlog* de sprint, mais de plan de sprint. Le risque de confusion est moindre en anglais, avec *Product Backlog* et *Sprint Backlog*, le premier mot faisant la différence.

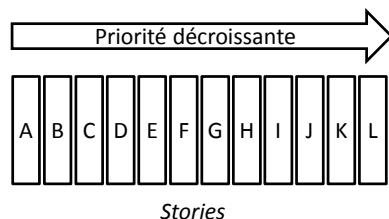
## **5.1 LE BACKLOG, LA LISTE UNIQUE DES STORIES**

La notion de *backlog* n'est pas très difficile à comprendre : c'est une liste dont les éléments sont rangés par priorité.



**Figure 5.1** — « J'ai dit de travailler sur le backlog, pas black dog ! »  
(Black dog est un morceau célèbre du groupe Led Zeppelin.)

Scrum n'impose pas de pratique pour identifier et nommer les éléments du *backlog*. L'usage le plus courant est de définir un élément comme étant une *story*.



**Figure 5.2** — Un *backlog* de produit

### Définition

La technique des *user stories* est fréquemment associée à Scrum. Seulement, dans un *backlog*, toutes les histoires (*stories*) ne sont pas relatives à des utilisateurs (*user*), certaines sont à caractère technique, par exemple. C'est pourquoi j'utiliserais le terme général de **story** pour parler d'un élément du *backlog* de produit.

Dans un *backlog* de produit, les *stories* sont rangées selon l'ordre envisagé pour leur réalisation. Cette notion de priorité, qui n'est pas usuelle dans les documents de spécification, prend une grande importance dans le développement itératif.

Pour un produit, il existe un et un seul *backlog* (c'est pour cela qu'on dit *backlog* de produit). On y trouve rassemblé ce qui est d'ordinaire dispersé dans plusieurs documents ou dans plusieurs outils.

### 5.1.1 Utilisateurs du backlog

Le *backlog* sert à communiquer : il est utilisé largement, car au carrefour de plusieurs activités :

- la gestion de projet, car le *backlog* est la base pour la planification ;
- l'ingénierie des exigences, puisqu'on y collecte ce que doit faire le produit ;
- la conception et le codage des *stories* sélectionnées pour un *sprint* ;
- le test, qui permettra de s'assurer que les *stories* sont bien finies dans un *sprint*.

Même si c'est le Product Owner qui en est responsable et qui définit les priorités, le *backlog* est partagé entre toutes les personnes de l'équipe. Le *backlog* est également visible des personnes extérieures à l'équipe qui sont intéressées par le développement du produit : clients, utilisateurs, managers, marketing, opérations...

Tout ce monde y a accès, ce qui favorise la transparence et facilite le *feedback*, qui se concrétise par l'ajout de nouvelles *stories*.

### 5.1.2 Vie du backlog

Le *backlog* suit la vie du produit qu'il décrit, il évolue avec lui ; il peut donc avoir une durée de vie très longue, courant sur de nombreuses *releases*.

#### Émergence progressive

Plutôt que d'essayer de tout figer dans le détail au début, le contenu du *backlog* est décomposé progressivement.

J'ai participé à de nombreuses définitions et spécifications d'exigences dans différents domaines du développement de logiciel. Le constat a toujours été le même : il est impossible de connaître toutes les exigences dès le début d'un projet. En réfléchissant longtemps et en essayant d'imaginer les situations dans lesquelles se trouveront les utilisateurs, on peut bien sûr découvrir un bon nombre d'exigences significatives, mais il en existera toujours qui émergeront plus tard : même en se mettant dans la peau des utilisateurs, on ne pourra pas les spécifier ni même les identifier à l'avance. Seul le *feedback* sur une version partielle permettra de savoir ce que les utilisateurs attendent réellement.

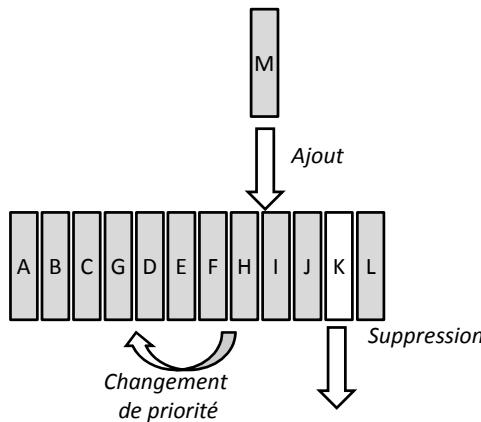
Plus récemment, j'étais le Product Owner d'une équipe Scrum et j'aurais été bien incapable de définir précisément au début ce qu'est finalement le produit aujourd'hui, même si j'avais des idées et que j'avais rédigé un document donnant la vision que j'en avais. Non seulement il n'aurait pas été possible de spécifier en détail les exigences, mais cela aurait été du gaspillage, car les changements ont été nombreux et fréquents.

Dans un développement agile, le changement est possible et même encouragé. Il ne coûte presque rien, tant qu'il porte sur un élément pour lequel on a fait peu d'effort.

Le contenu du *backlog* émerge progressivement, juste à temps. Ce concept est à la base de Scrum et des méthodes agiles. Pour un nouveau produit, cette émergence découle d'une approche descendante où les *stories* sont décomposées progressivement.

### Changements continuels

Le *backlog* n'est pas un document figé, il n'est jamais complet ou fini tant que vit le produit, il évolue continuellement : des éléments sont ajoutés, des éléments sont supprimés, des éléments sont décomposés ou les priorités ajustées.



**Figure 5.3** – Les changements dans le *backlog* : ajout, suppression et changement de priorité

Le Product Owner peut le modifier souvent, voire quotidiennement, en fonction du *feedback*. Cependant, les modifications les plus importantes surviennent au cours des réunions de fin de *sprint* et de début du suivant.

Par exemple, lors de la revue de *sprint*, il est ajusté en fonction du produit obtenu à la fin du *sprint* : les *stories* finies sont enlevées.

**Attention** : ce changement continu n'a pas d'impact sur l'équipe qui développe ; pendant un *sprint*, la partie du *backlog* sur laquelle l'équipe travaille est gelée.

### Utilisation continue

Le *backlog* est élaboré dans une forme initiale avant le lancement du premier *sprint* : il permet de planifier la *release*<sup>1</sup>.

Il sert pendant les *sprints* pour connaître les *stories* en cours de développement. Par exemple, lors de la réunion de planification en début de *sprint*, il est utilisé pour décider du sous-ensemble qui sera réalisé pendant le *sprint*.

1. Pour en savoir plus, lire les chapitres 6 *La planification de la release* et 13 *De la vision aux stories*.

### 5.1.3 Options de représentation du backlog

Un *backlog* est une liste qui peut être gérée de multiples façons.

Aux premiers temps des méthodes agiles, les *stories* étaient identifiées par des cartes (des fiches bristol) et la priorité définie par l'ordre dans lequel les cartes étaient rangées. Cependant, le besoin d'un outil informatique<sup>1</sup> vient assez vite pour gérer le *backlog*, être assisté dans le suivi, faire des plans et produire des graphiques.

Les outils permettent de produire plus facilement les artefacts dérivés du *backlog* :

- Le plan de *release* est une projection du contenu du *backlog* sur les *sprints* de la *release*.
- Le *burndown chart* de *release* est un graphe, mis à jour à la fin de chaque *sprint*, basé sur la taille courante du *backlog* de produit.

## 5.2 LA NOTION DE PRIORITÉ DANS LE BACKLOG

Le *backlog* est la liste unique de tout ce qui est à faire, ce qui donne beaucoup d'importance à la notion de priorité.

### 5.2.1 Le sens de la priorité

#### Que signifie la priorité ?

Dire que la *story* A est plus prioritaire que la *story* B signifie que A sera réalisée avant B. Les priorités sont utilisées pour définir l'ordre de réalisation.

À n'importe quel moment de la vie du produit, on sait que les *stories* les plus prioritaires du *backlog* sont candidates à être développées dans le prochain *sprint*. Les *stories* moins prioritaires seront développées plus tard, dans des *sprints* futurs. Elles peuvent donc être moins détaillées et être dans un état différent, avec des attributs moins précis.

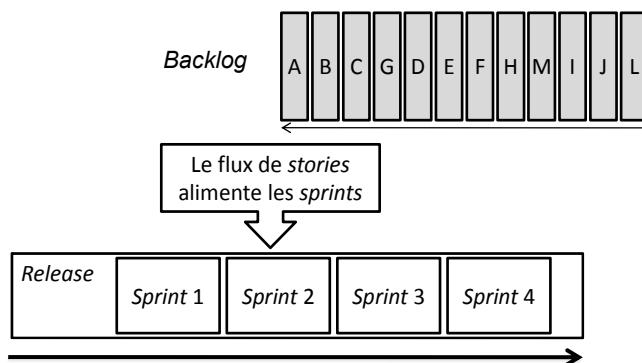
En résumé, la priorité permet de constituer le flux de *stories* qui va alimenter l'équipe. L'ordre peut changer tant que l'équipe n'a pas commencé à développer la *story*.

### 5.2.2 Les critères pour définir la priorité

La priorité a un impact sur le contenu du produit partiel qui est réalisé à la fin de chaque *sprint*. Pourquoi considérer qu'il vaut mieux avoir à la fin d'un *sprint* un produit avec la *story* A plutôt qu'avec la *story* B ?

---

1. La présentation des outils est faite au chapitre 17 *Scrum avec un outil*

**Figure 5.4** — Priorité et *sprints*

La priorité dépend de différents critères qui varient avec le temps. On a vu que les méthodes agiles avaient pour but de maximiser la valeur, ce qui en fait un critère majeur pour définir la priorité. Cependant cette notion de valeur n'est réellement applicable que sur des grandes fonctions en nombre limité (*features*) et sert essentiellement à guider dans l'ordre de décomposition. En rythme de croisière un *backlog* contient fréquemment 50 éléments dont certains détaillés, cela demande une approche plus large pour établir les priorités.

Les critères suivants poussent à donner une grande priorité à une *story* :

- **Le risque qu'elle permet de réduire** – L'objectif est de réduire l'exposition au risque le plus rapidement possible. Des *stories* permettant de valider des choix techniques structurants sont monnaie courante dans les premiers *sprints* d'un nouveau développement innovant.
- **L'incertitude sur des besoins des utilisateurs qu'elle permettra de diminuer** – Quand un utilisateur désire une fonctionnalité mais ne sait pas de quelle façon le service doit être rendu, la solution est de lui montrer rapidement une version pour obtenir du *feedback*. La connaissance apportée par le développement des *stories* relatives à cette fonctionnalité est une occasion d'améliorer le produit.
- **La qualité à laquelle elle contribue** – Les travaux visant à garantir la qualité du produit devraient être prioritaires. Par exemple, si une rétrospective a mis en évidence la nécessité d'automatiser les tests, la *story* correspondant aura une priorité élevée.
- **Les dépendances entre stories** – Si une *story* A ne peut être développée que si une autre *story* B existe, la *story* B doit être plus prioritaire que A.

C'est seulement une fois que ces critères sont pris en compte que la complétude fonctionnelle (et donc la valeur pour le client) devient prépondérante.

### 5.2.3 La gestion des priorités dans le backlog

Toute l'équipe participe à la définition des priorités, mais c'est en dernier lieu le Product Owner qui en est responsable.

#### Techniques de gestion des priorités dans le backlog

L'utilisation d'éléments physiques, comme des cartes ou des Post-it, facilite le rangement par priorité. Avec un outil comme un tableur, il est possible d'ajouter une colonne Priorité, et de donner une valeur pour chaque élément du *backlog*. Il est alors facile d'ordonner la liste sur cette colonne priorité. Mais l'utilisation d'un nombre fini et limité de valeurs pour la priorité, ne résout pas tout car au moment de trier il y a des éléments *ex æquo*. Il y a aussi un risque de confusion entre la priorité et la valeur ajoutée, qui, comme nous l'avons vu, n'est pas le seul facteur pour définir la priorité. C'est pourquoi la pratique la plus efficace est, plutôt que donner une valeur, d'ordonner tout simplement la liste des éléments du plus prioritaire au moins prioritaire. Les tableurs le permettent généralement mais les outils dédiés à Scrum offrent plus de facilités pour cela.

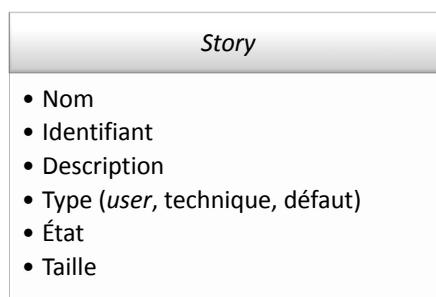
Dans ce cas, la priorité n'est pas un attribut explicite. Elle est implicite, donnée par l'ordre de rangement dans le *backlog*. On ne peut donc pas avoir deux éléments avec la même priorité.

## 5.3 UN ÉLÉMENT DU BACKLOG

### 5.3.1 Attributs

Scrum ne définit pas strictement les attributs d'un élément de *backlog*. Le choix est laissé aux équipes, en fonction du contexte.

Je conseille aux équipes avec lesquelles je travaille d'utiliser les attributs suivants :



**Figure 5.5** – Une *story* avec ses principaux attributs

Je vais détailler les trois derniers attributs un peu plus loin. D'autres attributs peuvent être utiles : l'estimation de sa valeur ajoutée, le créateur et la date de création, une estimation de sa fréquence d'exécution sur le produit déployé (une

*user story* peut se dérouler selon le cas plusieurs fois par jour ou une fois par semaine), le rôle associé (l'utilisateur qui est à l'initiative de la *story*).

Il faut ajouter les conditions permettant de considérer que la *story* est finie. Ces conditions sont essentielles pour l'acceptation de la *story* à la fin du *sprint*. Chacune correspond en fait à un cas de test qui peut être décrit de façon plus ou moins formelle. Un haut niveau de formalisation facilite l'automatisation des tests d'acceptation<sup>1</sup>.

Lors de la vie de la *story*, on y associera : le *sprint* dans lequel la *story* est planifiée et les tâches permettant de la réaliser.

### 5.3.2 Types

Il est courant d'identifier plusieurs types de *story*, ma recommandation est d'en utiliser trois :

- **User story** – Elle décrit un comportement du produit visible pour les utilisateurs et qui leur apporte de la valeur ou de l'utilité.
- **Story technique** – Elle est invisible pour les utilisateurs mais visible par l'équipe de développement. Elle est nécessaire pour pouvoir développer certaines *user stories*, son utilité est donc indirecte.
- **Défaut** – Il est relatif à un comportement visible des utilisateurs mais qui enlève de la valeur au produit. En développement de logiciel, on parle couramment de bug. Avec Scrum, on ne se préoccupe pas de savoir si une anomalie correspond à un bug ou à une demande d'évolution du produit. Peu importe, c'est un défaut qui demande du travail. Le défaut va dans le *backlog* et, par facilité de langage, nous allons considérer que c'est aussi une *story*, de type défaut.

Pour un nouveau produit, la majorité des éléments du *backlog* sont des *user stories*. La proportion est variable selon le contexte technique. Pour un produit qui est déjà utilisé, le nombre de défauts peut être significatif.

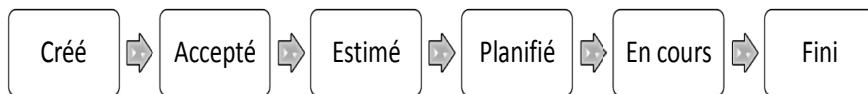
### 5.3.3 Cycle de vie d'un élément

La vie d'un élément du *backlog* est définie par les états présentés figure 5.6.

La vie d'une *story* est soumise à quelques règles : une *story* n'est estimée que si elle a été acceptée ; une *story* ne peut être planifiée que si elle est estimée ; une *story* ne peut devenir « en cours » que si elle est planifiée ; à la fin d'un *sprint* la *story* « en cours » devient finie.

En principe, une *story* finie ne fait plus partie du *backlog*, mais elle est souvent conservée pour garder une trace de ce qui a été fait.

1. Les tests d'acceptation sont présentés en détail dans le chapitre 15.



**Figure 5.6** – Cycle de vie simplifié d'un élément :

Créé (par n'importe qui) ; Accepté (par le Product Owner) ; Estimé (par l'équipe dans une séance de *planning poker*) ; Planifié (associé à un *sprint* futur lors de la planification de *release*) ; En cours (développé dans le *sprint* courant) ; Fini (terminé, selon la signification de fini).

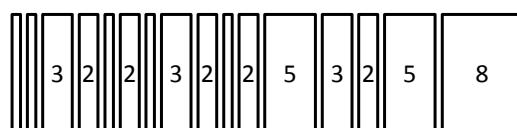
À partir de ces états, il est possible de filtrer les éléments du *backlog*, par exemple pour obtenir celles en cours. Selon l'état des *stories*, on peut identifier quatre grandes parties dans un *backlog*, selon les accès :

- une personne qui veut savoir ce qui est en cours de développement dans le *sprint* en cours filtrera le *backlog* uniquement sur les *stories en cours* ;
- quelqu'un qui veut connaître les prévisions à moyen terme est intéressé par les *stories planifiées*, qui constituent le plan de *release* ;
- le Product Owner dans son travail d'anticipation sur les futurs *sprint* va consulter la partie du *backlog* qu'il faut peut-être décomposer, compléter ou estimer ;
- tout le monde peut avoir envie de connaître les *stories finies* dans les *sprints* passés.

### 5.3.4 Taille des éléments

Un *backlog* contient des éléments de taille différente, ce qui est reflété par la valeur de l'attribut taille :

- Un élément prioritaire, placé en tête de la liste, va être bientôt développé. Il convient qu'il soit suffisamment détaillé pour être compris et réalisé par l'équipe. À ce niveau on a une *story* de petite taille, qui est prête pour le prochain *sprint*.
- Un élément au milieu du *backlog* ne sera développé que dans quelques *sprints*. À cette place les *stories* sont de taille moyenne et peuvent encore être décomposées.
- Un élément placé en fin de la liste sera développé plus tard. On y trouve des *stories* de plus grande taille qui seront décomposées ultérieurement.



**Figure 5.7** – Taille typique des éléments du *backlog* (les éléments les plus fins ont une taille de un) : on voit que la taille est plus grande pour les éléments les moins prioritaires

## 5.4 GUIDES D'UTILISATION DU BACKLOG

### 5.4.1 Partager le backlog avec toute l'équipe

Constituer et faire vivre le *backlog* favorise la collaboration : le Product Owner est moteur, mais toute l'équipe contribue à la découverte des stories et utilise le *backlog*.

La facilité avec laquelle le *backlog* est partagé dépend dans une large mesure de l'outil utilisé pour le gérer : plus il sera facile d'y accéder plus il y aura des chances que les personnes l'utilisent fréquemment. Si l'équipe se l'approprie, cela permet de renforcer sa connaissance du produit et la motive à obtenir ce qui est identifié.

En plus des réunions balisées, d'autres points de rencontre de toute l'équipe ont lieu, qui portent sur le *backlog* :

- **Ateliers** – La première fois, le *backlog* de produit est élaboré par des ateliers permettant d'identifier les éléments, les ordonner par priorité et les estimer.
- **Communication en face à face** – Tout n'est pas écrit dans le *backlog* : une story est l'objet de discussions pendant le *sprint* où elle est développée. Cela permettra de la faire passer de l'esprit du Product Owner à celui du développeur, tout en laissant à ce dernier une marge de manœuvre pour sa réalisation.

### 5.4.2 Bichonner le backlog

Bichonner signifie « *entourer de soins délicats et attentifs* » et c'est exactement ce que doit faire le Product Owner pour son *backlog*. Il le triture, il le nettoie, il le range et fait en sorte que ce soit un outil toujours opérationnel.

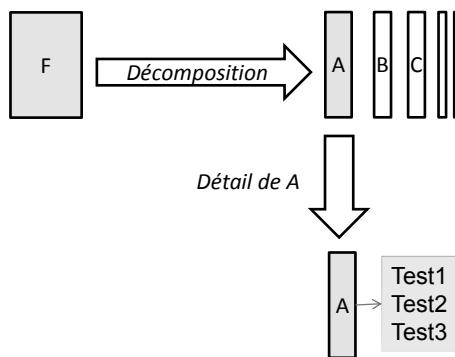
Un *backlog* à jour, cela implique notamment que des stories soient insérées (ou retirées) quand c'est nécessaire et qu'elles soient ordonnées par priorité, à la bonne granularité, estimées et détaillées au bon niveau, avec l'équipe.

#### Ajouter et retirer

Puisque les méthodes agiles favorisent le changement, le *backlog* est le réceptacle des nouvelles stories qui en sont le reflet. De nouvelles stories sont identifiées mais d'autres peuvent disparaître, parce qu'elles ne sont plus considérées utiles par le Product Owner.

#### Décomposer et détailler

Une story insérée dans le *backlog* peut être de grande taille. Elle est d'abord décomposée en stories de plus petite taille. Cette décomposition progressive précède le raffinement d'une story, lorsqu'on lui apporte plus de détails, généralement sous forme de cas de tests d'acceptation.



**Figure 5.8** — Décomposition et détail d'une story

### Changer les priorités

Un Product Owner est amené à changer l'ordre des priorités pour plusieurs raisons :

- chaque fois qu'une story est ajoutée dans le backlog, il faut lui donner une priorité, par rapport aux autres éléments : elle ne se place pas forcément en dernier ;
- une meilleure connaissance d'une story ;
- une découverte d'un bug ou le besoin d'une étude ;
- l'estimation de la taille d'une story ;
- son utilité potentielle peut changer au fil du temps ;
- la planification d'un sprint peut changer les priorités pour ajuster le périmètre à l'engagement de l'équipe.

### 5.4.3 Surveiller la taille du backlog

Le backlog ne doit pas comporter trop d'éléments. Sur sa partie active, celle qui porte sur les éléments qui sont à réaliser, donc sans considérer celles qui sont finies ni celles en cours, il est conseillé de rester à moins d'une cinquantaine d'éléments.

Cela est possible parce que les éléments se décomposent progressivement si on utilise une approche descendante. Cette approche de décomposition progressive est naturelle au début du développement d'un produit.

Si le backlog est constitué sur un produit existant ou commencé sans définir une vision, il y a des risques qu'il contienne trop d'éléments.

### Ne pas copier les gros documents de spécification

Quand une équipe démarre avec Scrum, il peut exister des documents de spécification traditionnels déjà rédigés, avec peut-être des centaines d'exigences identifiées et numérotées. Il est tentant d'en faire des entrées du backlog. Les mettre dans le backlog sans discernement rendrait le backlog ingérable. D'une part, parce qu'il y aurait trop d'éléments, d'autre part, parce qu'il faut structurer les éléments introduits dans le backlog.

La solution est alors d'analyser le document pour vérifier sa compatibilité avec une approche agile. Si ce n'est pas le cas, il vaut mieux s'en servir comme point de départ pour partir sur une identification des stories puis s'en débarrasser.

### ***Ne pas insérer tout le stock de bugs***

Sur des projets, on trouve parfois des stocks de bugs résiduels énormes que l'équipe a du mal à éliminer. C'est d'ailleurs souvent pour essayer de régler le problème que l'équipe passe à Scrum.

Seulement, transférer des centaines de bugs d'un « bugtracker » dans le *backlog* n'est sûrement pas une bonne idée, pour les mêmes raisons de taille : un tri préalable s'impose avant de les inclure.

### ***Nettoyer en profondeur***

À côté des changements au jour le jour, il peut y avoir besoin d'un gros ménage dans le *backlog*. Cette mise à jour, à faire de préférence à la fin d'une *release*, permet de supprimer, par exemple :

- des stories qui restent toujours au fond du *backlog*, probablement parce qu'elles ne sont finalement pas utiles,
- des doublons.

### **5.4.4 Éviter d'avoir plusieurs backlogs pour une seule équipe**

La règle est d'avoir un *backlog* par produit, une autre règle est d'avoir une équipe pour développer un produit, avec la recommandation que l'équipe est à temps plein sur un seul projet. De nombreuses organisations passant à Scrum sont dans un schéma où une seule équipe travaille sur plusieurs projets en même temps.

Bien souvent elles abordent la transition à Scrum en faisant un *backlog* par projet. Si la taille de l'équipe ne dépasse pas la taille standard d'une équipe Scrum, ce n'est pas la bonne approche : les problèmes de priorité ne seront pas résolus (sans oublier les autres inconvénients du multiprojets pour une personne). Une meilleure solution de transition est de rester dans le cadre : une équipe – un *backlog*.

## En résumé

Le *backlog de produit* est la liste des futures réalisations de l'équipe. C'est l'élément pivot d'un développement avec Scrum en ce qui concerne le contenu du produit et la gestion de projet.

Cette liste unique des choses à faire, rangées par priorité, est au cœur de la mécanique de mise en œuvre de Scrum lors des *sprints* : elle permet de définir le produit partiel visé et de faire la planification.

Le *backlog de produit* est pour beaucoup dans l'élégance et la simplicité du cadre de développement que constitue Scrum.

# 6

## La planification de la release

Ceux qui ne connaissent pas bien les méthodes agiles pensent parfois, à tort, qu'elles ne permettent pas de planifier, parce que « *ça change tout le temps* ». Ils ont bien compris que le client pouvait faire des changements et en déduisent trop rapidement : « *À quoi sert de faire des plans s'ils sont remis en question sans arrêt ?* »

D'autres qui appliquent Scrum depuis peu ont bien compris qu'il y avait de la gestion de projet pendant le *sprint*, mais ne voient pas encore l'intérêt de la planifier la *release*. Pourtant, agile ou pas, il est toujours nécessaire d'avoir des prévisions sur le moyen terme.

Imaginons un projet de développement d'une application pour une société qui organise des événements. Des questions ne manqueront pas de se poser sur le futur, comme par exemple :

- *Pourrons-nous utiliser la gestion des inscriptions en ligne pour la conférence de mars ?*
- *Dans combien de temps aurons-nous la possibilité de diffuser des conférences en streaming ?*
- *Quel est le budget nécessaire pour développer la version de mars de l'application ?*

Ce chapitre montre comment la planification de *release* permet de répondre à ces questions.

## Définitions

Une **release** est la période de temps constituée de *sprints* utilisée pour planifier à moyen terme.

La **vélocité** est la mesure de la partie du *backlog* de produit qui est réalisée par l'équipe pendant un *sprint*. Les mesures de vélocité sont utilisées pour planifier.

Un **burndown chart** est une représentation graphique du reste à faire dans une période, actualisé aussi souvent que possible et permettant de montrer la tendance. Dans le cas d'un *burndown chart de sprint*, la mise à jour est quotidienne, et le *burndown chart de release*, qui nous intéresse dans ce chapitre, est actualisé à chaque *sprint*.

# 6.1 PLANIFIER LA RELEASE

## 6.1.1 Planifier pour prévoir

Dans tous les projets, on fait des plans, pour essayer de prévoir ce que va contenir un produit ou à quelle date il sortira sur le marché. Avec Scrum, la planification de *release* a les mêmes objectifs : fournir à l'équipe et aux parties prenantes des informations sur le contenu des *sprints* constituant la *release*.

Il y a cependant deux différences fondamentales avec les approches classiques :

- il y a deux niveaux de plan<sup>1</sup> et le plan de *release* est *gros grain*,
- le plan n'est pas fait une fois pour toutes au début du projet, il évolue.

Le plan de *release* est certes élaboré une première fois au début du développement d'un produit, mais de manière plus légère qu'avec une approche traditionnelle, et il est mis à jour à chaque *sprint*.

## 6.1.2 Réunion ou processus ?

Dans le livre *Agile Project Management with Scrum* de Ken Schwaber (2004), la planification de *release* était à peine évoquée. Le *Guide Scrum* de la Scrum Alliance daté de mai 2009<sup>2</sup>, toujours écrit par Ken Schwaber, la mentionne explicitement comme faisant partie du cérémonial (les *timeboxes*). Mais il ne la détaille pas beaucoup ; à la différence des autres réunions du cérémonial, la planification de *release* est peu balisée. D'ailleurs dans les enquêtes sur les usages de Scrum ou de l'agilité, on constate que la planification de *release* est une des pratiques les moins répandues.

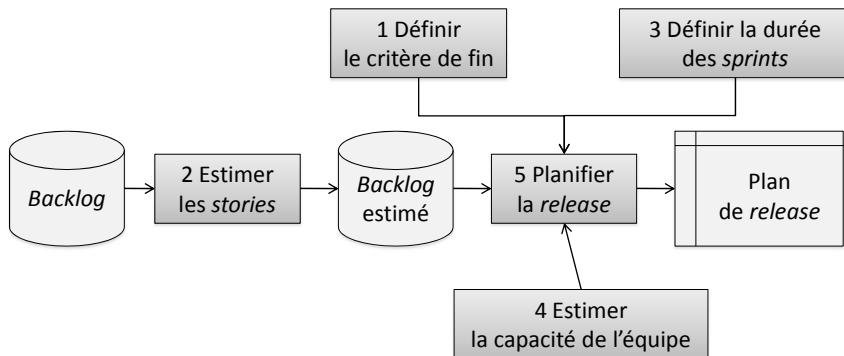
En fait, la planification de *release* correspond plus à un processus et elle ne se réduit pas en une seule réunion.

Une grande partie de l'effort nécessaire pour produire le plan de *release* est consacrée à l'estimation : pour planifier, il faut d'abord estimer. Avec Scrum et les

1. L'autre est le plan de *sprint* pour le court terme.

2. *Guide Scrum* : <http://www.scrumalliance.org/resources>

méthodes agiles, l'estimation est collective, elle s'élabore lors de réunions où toute l'équipe participe. Ce sont ces réunions qui rythment le processus de planification de la *release*.



**Figure 6.1** — Le processus de planification de *release*

La planification de *release* repose sur l'estimation et demande de définir deux variables, la durée du *sprint* et la capacité de l'équipe, avant de planifier.

### 6.1.3 La participation de l'équipe est requise

Toute l'équipe Scrum participe à la planification de la *release*. L'équipe considérée est l'équipe complète, avec le ScrumMaster et le Product Owner.

L'habitude prise par des managers, avant de passer à Scrum, de faire des plans seuls ou en comité restreint en début de projet peut les pousser à planifier la *release* sans faire participer l'équipe. En effet dans la gestion de projet traditionnelle, ce type de plan est élaboré par un ou plusieurs experts de l'estimation et de la planification. Avec Scrum, c'est différent, les estimations sont faites par l'équipe et aussi la planification de *release* qui en découle.

C'est un point fondamental de l'estimation agile : c'est l'équipe qui la fait, car ceux qui exécutent les tâches de réalisation sont les mieux placés pour en connaître les difficultés.

### 6.1.4 La *release* est planifiée à partir du backlog

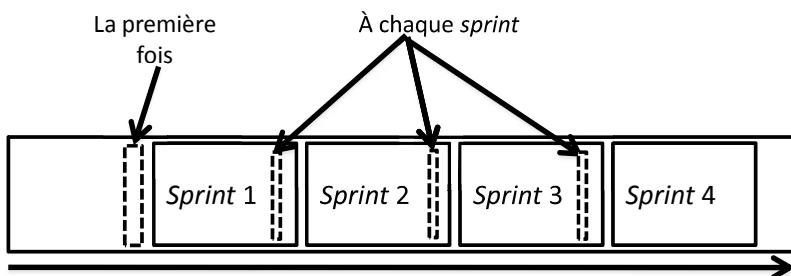
La planification de *release* utilise le *backlog* de produit que le Product Owner a préparé en ordonnant les *stories* par priorité.

Un *backlog* peut contenir 50 éléments, voire plus, mais la subtilité est que tout n'est pas décomposé au même niveau : on a besoin d'avoir une décomposition très fine, en petites *stories*, uniquement pour ce qui sera fait dans les prochains *sprints*. Pour le reste, la décomposition s'arrête lorsque l'estimation de l'élément est possible. Cela donne un *backlog* avec les petites *stories* devant et les grandes qui attendent leur tour derrière.

### 6.1.5 Place dans le cycle de vie

La planification de *release* commence pour la première fois avant le début du premier *sprint*, et ensuite elle a lieu au cours de chaque *sprint*.

La pratique la plus efficace est de tenir une réunion quelques jours avant la fin du *sprint* (et le début du prochain).



**Figure 6.2** — Quand faire la planification de *release*

La réunion de planification de *release* ne se déroule pas de façon aussi uniforme que les autres réunions Scrum. On peut distinguer la première, avant le début du premier *sprint*, de celles faites au cours de chaque *sprint*.

#### *La première fois, avant le premier sprint*

La première planification de *release* demande plus d'efforts que les suivantes. Élaborer pour la première fois un plan de *release* est difficile : il faut déjà disposer du *backlog* initial et il est souvent nécessaire pour cela d'organiser plusieurs ateliers.

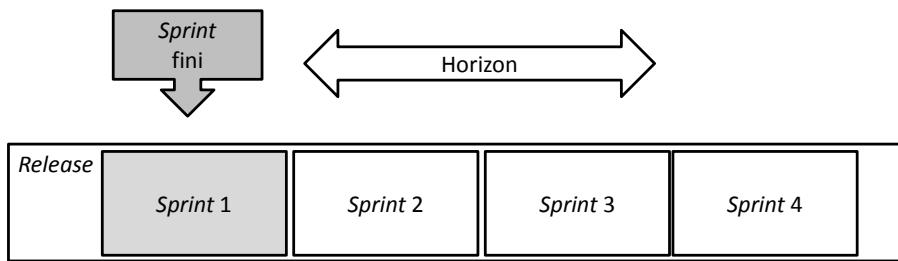
Une fois le *backlog* prêt, après une première passe sur les priorités et sur les estimations, la première réunion de planification de *release* peut se tenir. C'est une sorte de réunion de lancement de *release* (*release kickoff meeting*) permettant à l'équipe de dérouler les étapes du processus de planification de *release*.

**Attention :** si la *release* comporte de nombreux *sprints*, il n'est pas utile de faire une planification en détaillant les *stories* pour tous les *sprints*. Avoir un horizon précis à deux ou trois *sprints* est suffisant sachant que la planification de *release* est ajustée à chaque *sprint* : les *stories* à décomposer le seront au moment adéquat.

#### *À chaque sprint*

Dans les *sprints* successifs de la *release*, le travail à faire pour ajuster le plan de *release* dépend de la quantité de changements survenus.

Les changements dont il est question sont ceux qui nécessitent d'estimer ou ré-estimer : une nouvelle *story*, des *stories* qui résultent d'une décomposition, une modification substantielle d'une *story* existante. Tout cela fait varier la taille du *backlog*.



**Figure 6.3** — Horizon du plan de *release*

et a un impact sur le plan de *release*. Une modification dans les priorités du *backlog* est aussi un changement qui modifie le plan de *release*.

### Planification de *release* pendant un *sprint*

Pour mettre à jour la planification de *release*, je recommande de faire une réunion, un peu avant la fin du *sprint*. Lors de cette réunion, les étapes de la planification sont déroulées, mais uniquement sur ce qui a changé depuis le début du *sprint*.

Elle a un intérêt supplémentaire : quand le Product Owner présente les *stories* qu'il pense inclure dans le prochain *sprint*, l'équipe peut se prononcer sur sa perception de la *story*. Soit l'équipe confirme qu'elle est prête à être incluse dans le prochain *sprint*, soit elle considère que la présentation faite par le Product Owner est insuffisante et ne permet pas l'inclusion de l'élément dans le *sprint* à venir. Dans ce cas, il restera quelques jours au Product Owner pour compléter la connaissance relative à cette *story*. S'il ne peut pas le faire avant le début du *sprint*, l'élément ne sera pas inclus dans ce *sprint*.

**Durée** : une heure maximum. Ce temps passé en réunion est largement compensé par la diminution de la durée de la planification du *sprint* et par le temps gagné sur la compréhension du travail à faire.

**Quand** : il faut tenir cette réunion un peu avant la fin du *sprint*. Pas trop tard pour laisser un peu de temps au Product Owner s'il lui faut approfondir des *stories*. Pas trop tôt pour avoir une idée de ce qui sera effectivement fini à la fin du *sprint*. Pour des *sprints* qui durent trois semaines, je conseille de la tenir trois jours avant la revue de fin.

Lors de la revue de *sprint* et de la planification du *sprint* suivant, le plan de *release* peut être légèrement ajusté, en fonction des résultats et du nouvel engagement de l'équipe.

## 6.2 ÉTAPES

### 6.2.1 Définir le critère de fin de la *release*

Une *release* est une séquence de *sprints*, mais quand finit-elle ? Pour décider de l'arrêt des *sprints* et de la fin d'une *release*, il existe plusieurs possibilités.

## Finir quand le backlog est vide

Ceux qui sont habitués à développer à partir d'une spécification contenant tout ce qu'il y a à faire seront tentés de dire que la *release* s'arrêtera quand le *backlog* de produit sera vide. Mais le *backlog* est vivant et, finalement, il se rapproche d'un flot continu toujours alimenté. Ce n'est donc pas une bonne idée que de vouloir vider le *backlog*.

J'ai connu une équipe qui a essayé pendant 18 *sprints*, sans succès !

Une variante est de sélectionner un sous-ensemble du *backlog* pour la *release*. Comme les éléments sont rangés par priorité, il suffit de fixer une limite en disant : « *on s'arrêtera là pour la release courante, les stories suivantes seront faites dans la release suivante* ».

C'est ce qu'on appelle une *release* à périmètre fixé. La planification de la *release* a alors pour objectif d'estimer la date de fin.

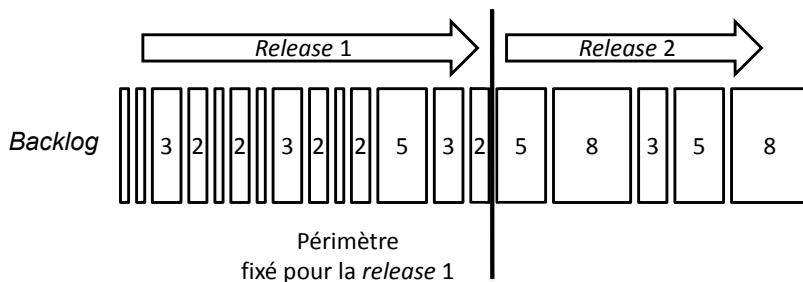


Figure 6.4 — La *release* à périmètre fixé

**Attention :** même restreint de cette façon, le périmètre d'une *release* peut toujours évoluer, il serait stupide de figer le *backlog* en refusant un changement qui apporte de la valeur.

## Fixer la date à l'avance

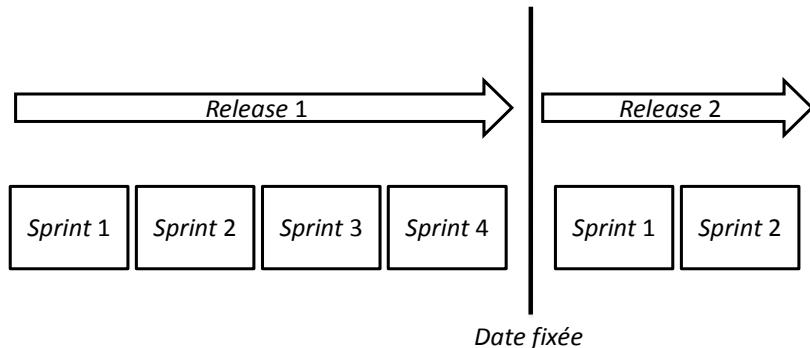
Une meilleure façon de procéder est de définir une date de fin et de s'y tenir, en reprenant l'idée de la *timebox*. L'objectif de la planification d'une *release* à date fixée est alors d'estimer quel contenu sera fourni à la date prévue.

La *release* à date fixée présente de nombreux avantages :

- elle donne un objectif précis et généralement pas trop lointain, ce qui motive plus l'équipe ;
- elle demande obligatoirement une réflexion poussée sur les priorités des éléments du *backlog* par le Product Owner ;
- des éléments du *backlog* présentant finalement peu d'intérêt ne seront pas intégrés dans la *release* ;

- on passe généralement moins de temps à estimer et planifier, puisque la date de livraison est connue.

Un autre avantage est le rythme donné par des *releases* régulières : une organisation s'habituerà à cette fréquence, qui cadence le travail de l'équipe mais aussi celui des utilisateurs et de leurs représentants.



**Figure 6.5** — La *release* à date fixée

Il y a des variantes possibles pour définir la fin d'une *release* :

- Attendre quelques *sprints* pour décider. Une fois la décision prise, on se retrouve dans une des deux situations précédentes.
- Arrêter la *release* quand le produit partiel a suffisamment de valeur.
- Arrêter la *release* quand le budget est consommé.

La *release* à date fixée et à durée uniforme, par exemple une *release* tous les trois mois, est la formule la plus facile à mettre en œuvre.

Pour prendre une métaphore sportive, la *release* à date fixée s'apparente au test de Cooper. Ce test était pratiqué autrefois à l'armée : il s'agit de parcourir la plus grande distance possible en douze minutes. La *release* à périmètre fixé se rapprocherait d'une course sur une distance définie, par exemple un 5 000 mètres.

## 6.2.2 Estimer les stories du backlog

Chaque *story* du *backlog* doit être estimée si on veut en tenir compte dans la planification. L'estimation dont il est question ici est celle relative à l'effort à fournir pour la développer.

En effet, les *stories* dans le *backlog* ne sont pas toutes de même taille et on ne peut pas simplement se baser sur le nombre de *stories* à faire pour planifier.

La technique utilisée pour estimer n'est pas imposée dans Scrum. L'usage le plus fréquent est de faire une estimation collective au cours d'une séance appelée *planning poker* et d'estimer la taille plutôt que la durée.

La technique du *planning poker*<sup>1</sup> (<http://www.planningpoker.com/detail.html>) connaît un succès grandissant auprès des équipes Scrum (en fait, il ne s'agit pas de *poker* ni de *planning*, un nom plus approprié serait estimation de *backlog*).

C'est une séance d'estimation en groupe, avec des cartes, qui combine le jugement d'expert et l'estimation par analogie.

### Déroulement du *planning poker*

Chaque participant reçoit un jeu de cartes.

Sur chaque carte il y a une valeur possible pour l'estimation d'une *story*.

Le Product Owner présente la *story*.

Les membres de l'équipe posent des questions pour bien la comprendre et débattent brièvement.

Tous les participants présentent en même temps la carte choisie pour l'estimation.

Le groupe discute des différences éventuelles.

On recommence jusqu'à arriver à une convergence des estimations pour la *story*, puis on passe à la suivante.

Comme il est plus facile de faire des estimations sur une échelle prédéfinie plutôt que d'avoir à sa disposition tous les entiers, la suite de Fibonacci est généralement utilisée : 1, 2, 3, 5, 8, 13.

De nombreux jeux de cartes ont fait leur apparition et sont vendus sur Internet ou fournis par des sponsors lors de conférences. La suite de Fibonacci est complétée avec 0 et  $\frac{1}{2}$  pour les petites *stories* et 20, 40 et 100 pour les grandes.



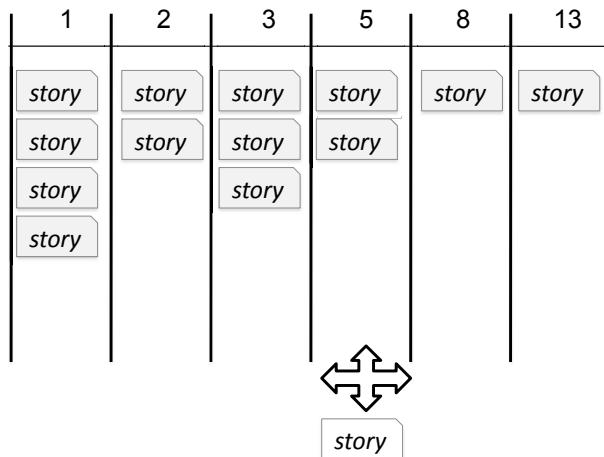
**Figure 6.6** – Des cartes de *planning poker*  
(<http://www.tekool.net/blog/2009/07/21/printable-agile-planning-poker/>)

1. Pour plus d'infos, lire : <http://www.planningpoker.com/detail.html>

Pour commencer la séance, il faut définir un étalon. C'est simplement une *story* connue de tous, pour laquelle l'équipe décide en commun d'une valeur arbitraire.

Il est préférable de choisir une *story* de taille moyenne et de lui donner une valeur de 2, 3 ou 5, pour laisser le spectre ouvert vers le haut et vers le bas.

L'estimation par comparaison est facilitée par l'usage de Post-it pour chacune des *stories* du *backlog* : on fait des rangées pour chaque valeur possible d'une estimation.



**Figure 6.7** – *Stories rangées par niveau d'estimation au cours du planning poker*

Comme toutes les *stories* déjà estimées sont visibles, cela facilite les estimations des suivantes et permet de raccourcir le temps consacré au *planning poker* en cas de divergence. Il arrive également que l'équipe ajuste *a posteriori* une estimation lorsqu'elle la compare à d'autres ayant obtenu la même estimation lors du *planning poker*.

Pour que cette technique soit efficace, il faut avoir déjà une bonne définition du produit, avoir constitué un *backlog* et avoir ordonné les éléments par priorité.

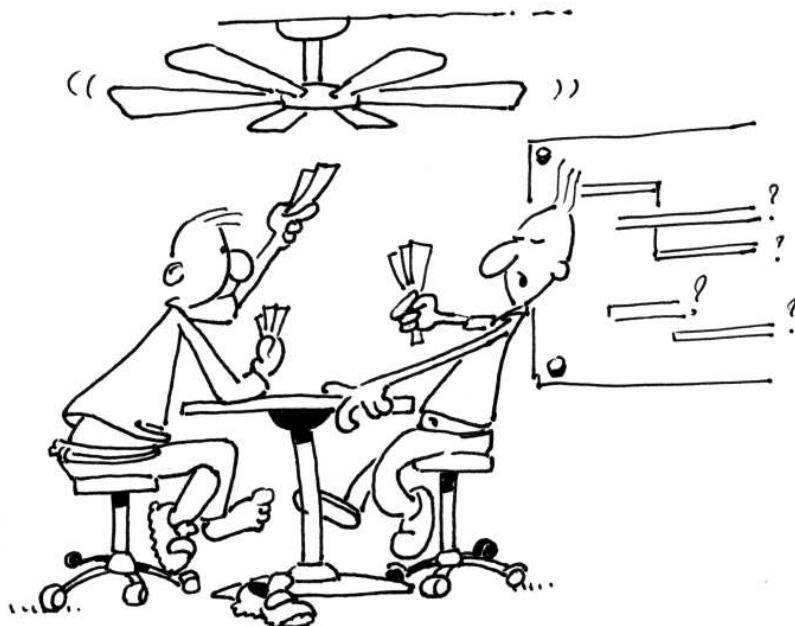
C'est alors la meilleure technique d'estimation que j'aie pratiquée. En plus de l'aspect estimation, elle favorise une discussion riche entre l'équipe et le Product Owner.

Quelques leçons tirées des nombreuses sessions de *planning poker* que j'ai animées<sup>1</sup> :

- c'est facile à organiser, il suffit de disposer d'un jeu de cartes par personne ;
- il y a rarement besoin de revoter une seconde fois, les discussions suite au premier vote suffisent généralement pour converger ;
- la réflexion par analogie est souvent utilisée dans les discussions après le premier vote ;

1. Même dans de grandes administrations ou des banques, nous avons sorti les cartes de poker !

- la technique de décomposition des *stories* (elle peut être appliquée pendant la réunion) améliore la qualité des estimations ;
- quand l'équipe est perplexe avant d'estimer, c'est le signe que la *story* est vraiment trop vague et que le Product Owner devrait l'approfondir.



**Figure 6.8** — Des cartes au travail, oui mais pour planifier !

### 6.2.3 Définir la durée des sprints

Lorsqu'on se lance dans le développement agile, une des premières questions à laquelle il faut répondre concerne la durée des itérations.

Il n'y a pas de réponse universelle, chaque projet est différent et doit définir sa façon de travailler. S'il existe un consensus de la communauté agile pour préconiser que toutes les itérations doivent être de même durée et qu'une itération est un bloc de temps fixé, il y a des variations sur la durée d'une itération.

Scrum recommandait, jusqu'à il y a quelques années, des *sprints* d'un mois. La pratique actuelle dans les développements avec Scrum, c'est moins : la plupart des projets ont des *sprints* de deux ou trois semaines.

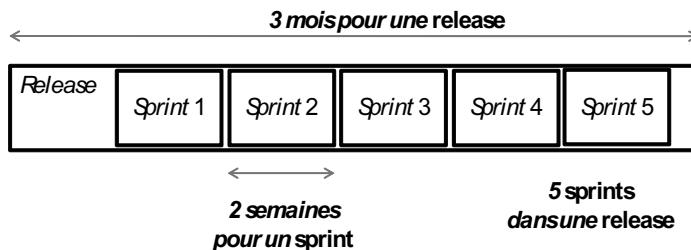
Les critères à retenir pour définir la bonne durée sont :

- **L'implication des clients et utilisateurs** – Il faut tenir compte de leur disponibilité à utiliser les versions partielles produites à la fin de chaque *sprint*.

- **Le coût supplémentaire engendré par le *sprint*** – Un *sprint* ajoute du travail supplémentaire pour préparer le produit partiel, faire les tests de non-régression, préparer la démonstration et pour les revues de fin et de début de *sprint*.
- **La taille de l'équipe** – Plus il y a de personnes dans l'équipe, plus il faudra de temps pour se synchroniser.
- **La durée maximum pour prendre en compte un changement** – Il faut tenir compte du fait que cette durée peut aller jusqu'à deux fois la durée d'un *sprint* (le changement est demandé pendant l'itération  $n$  et développé au plus tôt dans l'itération  $n+1$ ).
- **La date de fin de la *release*** – La *release* devrait comporter au moins quatre *sprints* pour que l'équipe commence à bénéficier des avantages de l'itératif. Donc si la date de fin est dans deux mois, il est préférable d'avoir des *sprints* de deux semaines ou moins.
- **Le maintien de la motivation de l'équipe** – Un *sprint* avec une durée trop longue est sujet à ne pas avoir une distribution uniforme du travail pendant l'itération ce qui conduit à travailler dans l'urgence à la fin.
- **La stabilité de l'architecture** – Ce sera difficile d'obtenir un produit qui fonctionne dans une durée courte si l'architecture n'est pas stable.

Comme le dit Thierry Cros<sup>1</sup> : « *une durée d'une semaine pour les sprints, c'est le mieux, dans les contextes où c'est possible. Quand ce n'est pas possible, alors on envisage deux semaines. Si une durée de deux semaines semble difficile dans le contexte, on passe à trois.* » En général, la durée d'un *sprint* est un multiple de nombre de semaines, on ne fait pas des *sprints* de 13 jours ou 17 jours.

#### Exemple de durées fréquemment utilisées dans les équipes :



**Figure 6.9** — Durées de *releases* et de *sprints* usuelles

1. Durée du *sprint* : <http://etreagile.thierrycros.net>

## 6.2.4 Estimer la capacité de l'équipe

### Vélocité et capacité

#### Définition

La **vélocité**, mesure de la partie de *backlog* réalisée par l'équipe pendant un *sprint*, se calcule juste après la démonstration lors de la revue de *sprint*.

La **capacité** de l'équipe est une prévision de ce que l'équipe est capable de faire pendant un *sprint*. Elle se base sur la vélocité, selon le principe de la « météo de la veille » : l'équipe devrait faire dans un *sprint* à peu près autant qu'elle a fait dans le précédent.

La vélocité est une mesure : il faut que l'équipe ait vécu une expérience commune pour l'avoir collectée. Quand un développement commence et qu'une nouvelle équipe vient d'être constituée, elle n'a pas de passé, elle n'a donc pas de vélocité connue. Si on veut quand même faire un plan de *release*, sans attendre de dérouler un ou deux *sprints*, il faut estimer sa capacité à produire. Une façon de faire est de simuler la réunion de planification du premier *sprint* : l'équipe étudie quelques *stories* parmi les plus prioritaires du *backlog*, les décompose en tâches et estime la durée de ces tâches. En les rapportant à la taille des *stories* en points, il est possible d'obtenir une valeur pour la capacité de l'équipe.

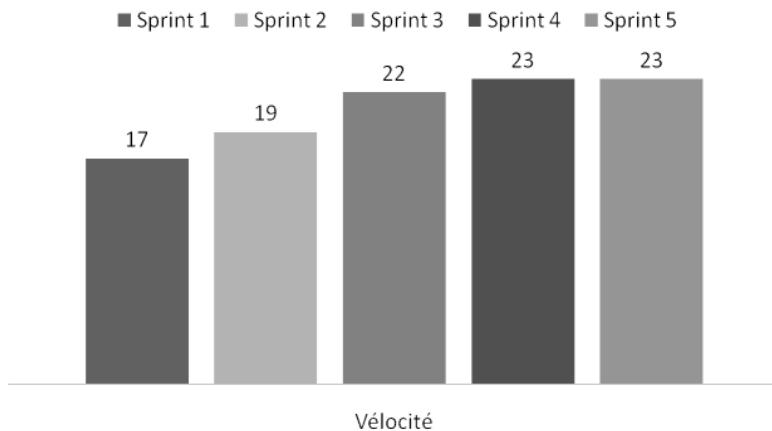
**Exemple :** trois *stories* sont étudiées, qui avaient été estimées à respectivement 3, 2 et 5 points. Les tâches identifiées pour ces *stories* sont estimées à 30 heures. L'équipe dispose de 300 heures pour le *sprint*. La capacité estimée est de  $300/30 \times 10$  soit 100 points.

Le chiffre obtenu contient une grande part d'incertitude. Il ne doit absolument pas être pris pour un engagement. C'est seulement une première valeur qui permet de construire le plan de *release* initial.

Il ne faut pas confondre estimation et engagement. Un plan de *release* se base sur des estimations et il peut, éventuellement, permettre de prendre des engagements.

Une fois que la série des *sprints* a commencé, la vélocité est mesurée à la fin de chaque *sprint* lors de la revue. La vélocité est volatile et, surtout au début d'une *release* avec une nouvelle équipe, elle peut varier sensiblement entre des *sprints* consécutifs. Il faut plusieurs *sprints* pour avoir des mesures pertinentes.

Pour le plan de *release*, on définit la capacité en faisant la moyenne des vélocités mesurées sur les *sprints* depuis le début, ou seulement sur celle des trois derniers *sprints*, selon la confiance qu'on a dans les premières valeurs.



**Figure 6.10** — Graphe de vélocité : la capacité utilisée pour la planification de *release* est de 22 pour le schéma.

#### *Ne pas surestimer la fiabilité de la mesure*

La vélocité est bien une mesure qui se collecte pour chaque *sprint*, cependant il ne faut pas perdre de vue qu'elle est basée sur des estimations, celles faites sur la taille des *stories*. Si la vélocité augmente, il n'y a pas moyen de savoir si c'est dû à une amélioration de la productivité ou à des estimations imprécises.

Il est aussi possible d'augmenter artificiellement la vélocité, j'ai connu un chef de projet, qui n'avait pas encore l'esprit de ScrumMaster, pousser à des ré-estimations à la hausse avant chaque *sprint* parce qu'il croyait habile d'afficher une vélocité qui augmente, en rejetant sur le Product Owner l'augmentation du nombre de points à faire au total.

La vélocité varie, alors que les ressources restent fixes : cela illustre que la capacité de l'équipe évolue et met en évidence l'intérêt d'estimer la taille plutôt que l'effort.

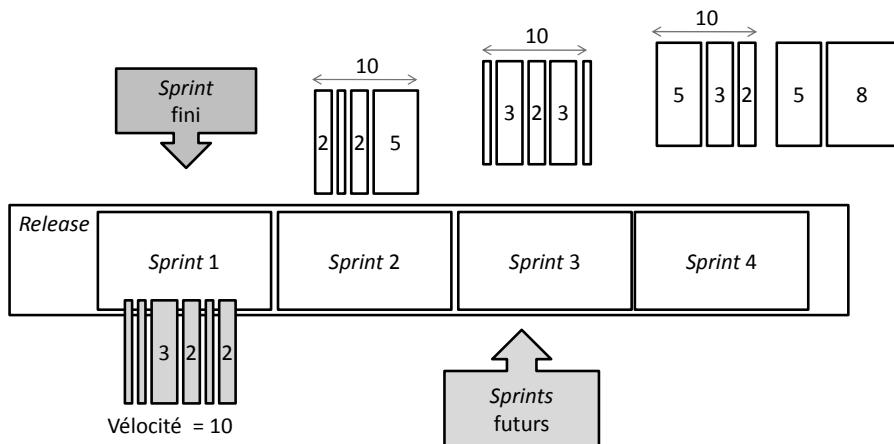
**Attention** : la vélocité est une mesure de l'équipe, pas de personnes individuelles.

#### **6.2.5 Produire le plan de release**

Une fois qu'on a déroulé les trois premières étapes du processus, la production du plan de *release* est un jeu d'enfant (s'il aime les mathématiques).

On prend le *backlog* priorisé et estimé. On commence par le premier *sprint* de la *release*. On y associe les *stories* en commençant par la plus prioritaire. On continue dans ce *sprint* en additionnant le taille en points des *stories* jusqu'à arriver à la capacité de l'équipe. Quand on y arrive, on passe au *sprint* suivant.

**Exemple :** la vitesse moyenne est de dix, ce qui conduit à prendre dix comme capacité de l'équipe. Pour les *sprints* à planifier, dix points de *backlog* sont affectés en suivant les priorités.



**Figure 6.11** — Planification de *release*

C'est simple et d'ailleurs certains outils permettent de faire la planification automatique de la *release*.

Une intervention manuelle s'avère quand même utile :

- si on ne tombe pas juste sur la capacité en ajoutant une *story* au *sprint* : il faut décider si on se place plutôt en dessous, plutôt en dessus ou si on prend une *story* moins prioritaire mais qui permet d'être plus proche de la capacité ;
- parce que la vue du plan de *release* pousse à faire des ajustements : l'art de la priorité est délicat et voir le contenu des *sprints* amène parfois à l'équipe à faire de nouveaux arbitrages dans le plan de *release*.

## 6.3 RÉSULTATS

### 6.3.1 Le plan de *release*

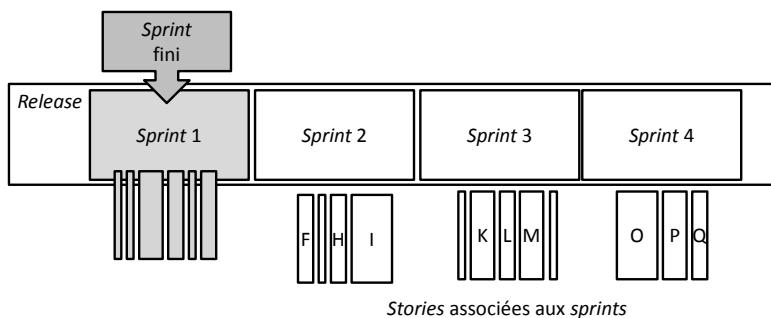
Un plan de *release* présente les *sprints* à venir et le contenu prévu de ces *sprints*, défini par les *stories* associées.

Un plan de *release* a deux caractéristiques nouvelles par rapport aux plans qu'on fait habituellement dans les projets :

- Il est orienté vers les clients et les utilisateurs pour que ceux-ci comprennent l'impact des changements proposés : plutôt que de voir des tâches qui ne leur parlent pas, ils y trouvent des *stories* qui devraient les intéresser davantage.

- Il est mis à jour régulièrement : plutôt qu'un plan détaillé fait à l'avance qui devient la référence intouchable, le plan de *release* évolue pour tenir compte des changements.

Les informations contenues dans le plan de *release* servent à anticiper les interdépendances. Dans un développement il arrive que les travaux d'une équipe dépendent de ceux faits par une autre équipe. C'est fréquent quand plusieurs équipes travaillent sur le même produit ou quand le logiciel d'un système embarqué dépend du matériel. Le plan de *release* permet d'identifier les points de synchronisation nécessaires et d'anticiper en adaptant les priorités.



**Figure 6.12** — Un plan de *release*

Présenté sous forme de tableau, un plan de *release* est facile à comprendre. Il constitue un outil de communication important avec tous les intervenants du projet. Le plan de *release* est visible, soit en étant affiché dans l'espace collaboratif de l'équipe, soit, si l'équipe est dispersée, en étant facilement accessible en ligne (voir fig. 6.13 exemple d'un plan réalisé avec IceScrum).

La *release* apparaît dans le bandeau supérieur, avec sa date de fin. En dessous, les *sprints* sont présentés de façon séquentielle de gauche à droite, avec pour chacun le but, la capacité prévue et les dates de début et fin. Les éléments du *backlog* planifiés (associés aux *sprints*) sont estimés en points. Le type d'élément (*user story*, *story technique* ou défaut) est montré par les icônes en haut à gauche des Post-it.

### 6.3.2 Burndown chart de *release*

Un *burndown* de *release* est un indicateur graphique basé sur la mesure de ce qui reste à faire. Un point dans le graphe est ajouté pour chaque *sprint* : la valeur de l'abscisse correspond à la taille de la partie du *backlog* qui reste à faire d'ici la fin de la *release*.

Pour l'obtenir il faut donc une liste de ce qui reste à faire et une mesure de la taille de chaque élément, ce qu'on trouve dans le *backlog* de produit.

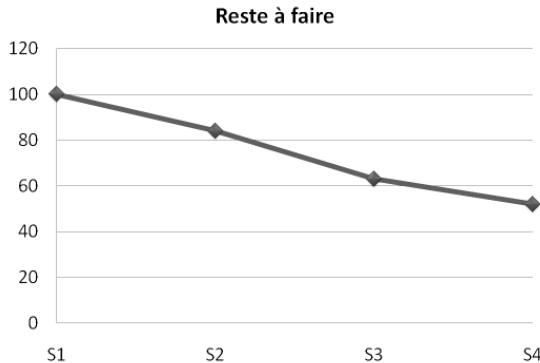
À quoi sert le *burndown chart* de *release* ?

- à montrer l'avancement réel, en tout cas le meilleur qu'on ait, puisqu'il est basé sur la distinction entre ce qui est complètement fini et ce qui reste.

- [ 26-07-2009 > 10-10-2009 ] - Release activée		
Sprint 1 Annonces 24pts 02/08/2009 → 23/08/2009	Sprint 2 Inscriptions 23pts 23/08/2009 → 13/09/2009	Sprint 3 Generated Sprint 18pts 13/09/2009 → 04/10/2009
Framework Symphony Planifié 5	Suivi inscriptions Planifié 5	Lecture compte-rendu Planifié 5
Nouvelle conférence Planifié 2	Inscription Planifié 3	Dépôt de photos Planifié 13
Accès conférence Planifié 1	Ouverture des inscriptions Planifié 2	
Programme Planifié 3	Annulation inscription Planifié 2	
Relance Planifié 5	Compte-rendus de conférence Planifié 8	
Lecture programme Planifié 8	Dépôt de présentation Planifié 3	

**Figure 6.13** — Un plan de *release* avec IceScrum

- à montrer la tendance et par là à se poser des questions sur la façon de continuer.



**Figure 6.14** — Un *burndown chart* de *release*

Si la *release* est à périmètre fixé, le *burndown* permet d'estimer la date de fin (figure 6.15).

Si la *release* est à date fixée, le *burndown* permet d'estimer le contenu qui sera fini à cette date.

Le schéma de la figure 6.16 illustre, pour une *release* à date fixée, le nombre de points résiduels obtenus en prolongeant la tendance des premières itérations.

En fonction de ce que présente le *burndown*, des décisions peuvent être prises plus facilement pour ajuster l'objectif de la *release*.

Le *burndown* est limité<sup>1</sup> dans les informations qu'il apporte : il ne fait pas apparaître les variations dues aux modifications de périmètre.

**Exemple** : si lors de l'itération 3 le graphe montre qu'on est passé de 140 points au début à 134 à la fin, on ne sait pas si la vitesse est de 6 ou si elle est en fait plus élevée et combinée à des ajouts de nouvelles *stories* à faire pour la *release*.

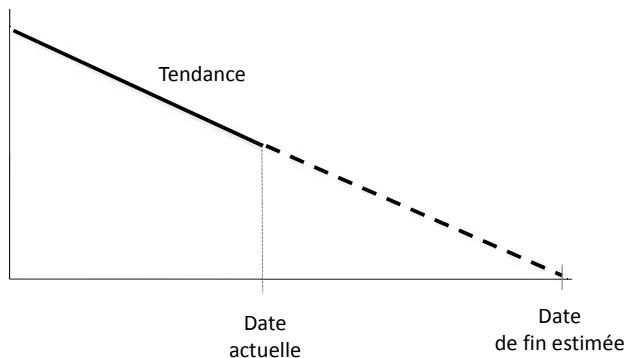


Figure 6.15 – Date déduite du *burndown*

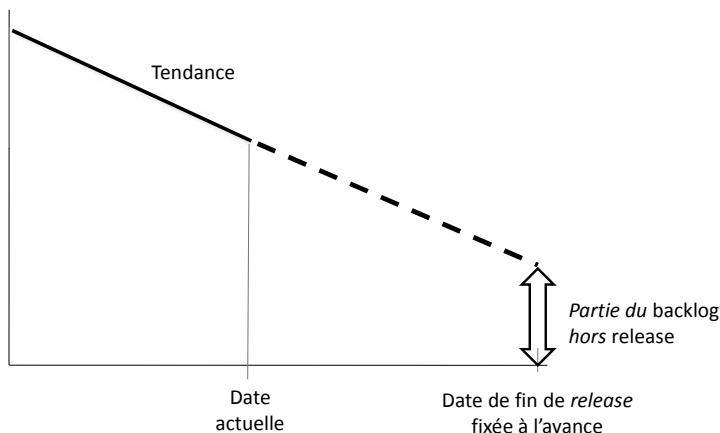


Figure 6.16 – Contenu déduit du *burndown*

1. Pour en savoir plus, voir les autres graphiques, chapitre 15.

## 6.4 GUIDES POUR LA PLANIFICATION DE RELEASE

À essayer	À éviter
S'adapter au calendrier	Confondre valeur et coût, vitesse et productivité
Provisionner pour du <i>feedback</i> ultérieur	Ne pas garder de mou pour les incertitudes

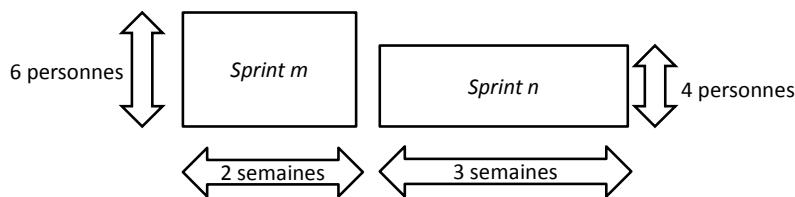
### 6.4.1 S'adapter au calendrier

La pertinence de la vitesse repose sur une durée fixe des *sprints* associée à une composition de l'équipe qui ne change pas : des ressources identiques d'un *sprint* à l'autre.

Agile ou pas, la planification agile doit tenir compte des événements connus à l'avance, comme les ponts en mai ou les vacances de membres de l'équipe qui peuvent influencer la quantité de ressources disponibles.

**Exemple :** une équipe de cinq personnes qui fait habituellement des *sprints* de trois semaines dispose de cinq fois cinq fois trois soit 75 jours de ressources. Elle commence un nouveau *sprint* le 28 avril et les membres de l'équipe font les ponts de mai. Pour avoir à peu près les mêmes ressources que pour les autres *sprints*, il est logique de passer la durée de ce *sprint* à quatre semaines. Cela devrait être anticipé dans le planning de la *release*.

La durée peut exceptionnellement varier pour garder les ressources à peu près stables pour tous les *sprints*.



**Figure 6.17** — Ressources stables pendant les vacances de deux personnes au *sprint n*

Si la taille de l'équipe évolue pendant la *release*, la mesure de la vitesse est évidemment moins fiable : une personne en plus ou en moins, cela a un impact sur sa capacité, en particulier pour les petites équipes. Un autre inconvénient est d'ordre collectif : un nouvel arrivant doit apprendre à s'intégrer dans l'équipe, cela lui demande du temps et l'équipe y consacre aussi du temps.

## 6.4.2 Ne pas confondre valeur et coût, ni vitesse et productivité

Une story a deux attributs différents : un porte sur la valeur qu'elle apporte, un autre sur sa taille. Ce sont deux notions distinctes qui sont parfois confondues. Il n'y a pas toujours une corrélation entre les deux : deux stories de même taille peuvent avoir des valeurs ajoutées bien différentes.

De la même façon, la vitesse ne doit pas être confondue avec la productivité. Ce sont deux mesures différentes, contrairement à des idées répandues, et il arrive même qu'une augmentation de la vitesse aille de pair avec une diminution de la productivité.

La définition classique de la productivité, c'est le quotient de résultat/temps passé à produire. La notion est surtout utilisée en économie pour montrer que l'utilisation de machines permet de réduire le temps de production.

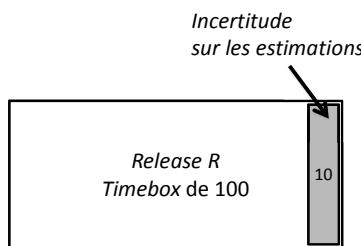
L'estimation en points porte sur le coût de développement, et donc la vitesse aussi. La définition de la productivité parle de résultat. À mon avis, ce n'est pas le coût qu'il faudrait utiliser mais la valeur ajoutée. La mesure de la valeur apportée par chaque sprint, faite de cette façon, se rapprocherait plus de la productivité au sens utilisé en économie.

## 6.4.3 Garder du mou pour les incertitudes

Dans le plan de release, tout est basé sur l'estimation des stories du backlog. Même si l'estimation est collective et faite par ceux qui réalisent, il y a une part d'incertitude, en particulier au début d'une release.

Pour en tenir compte, il est indispensable de garder du mou (un *buffer*) dans les plans :

- pour une release à périmètre fixé, le mou consiste en du temps ;
- pour une release à date fixée, le mou porte sur des stories.



**Figure 6.18** — Garder du mou pour les incertitudes dans une release à date fixée

Dans l'exemple de la figure 6.18, la vitesse moyenne est de 20 et il reste cinq sprints avant la fin de la release. Sans mou, le plan prévoit que 100 points du backlog seront inclus dans la release. Avec un mou de 10 %, la prévision de 90 points prend en compte le risque lié aux estimations. Le pourcentage de mou se réduit à l'approche de la fin de release.

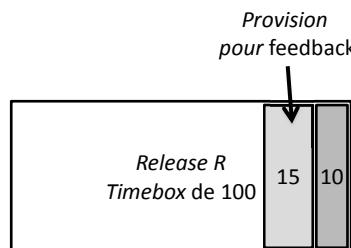
#### 6.4.4 Provisionner pour le feedback ultérieur

Une erreur classique lorsqu'on fait ses premières armes en planification de *release* est d'oublier le *feedback*.

Le *feedback* est une pratique essentielle du développement agile : il permet d'améliorer le produit en prenant en compte les retours des utilisateurs. Mais cela a un coût qu'il ne faut pas oublier dans le plan de *release*.

La prise en compte concrète du *feedback*, ce sont de nouvelles stories ajoutées dans le *backlog*, qu'il convient d'estimer et de prioriser. Si le Product Owner considère que cela est prioritaire, et c'est souvent le cas, l'impact sera de repousser d'autres stories plus loin dans le *backlog*.

Le *feedback* se décline en des demandes d'évolution sur des *stories* existantes et, éventuellement, en de nouvelles *stories*. Cela peut consister aussi en des défauts sur des *stories*, ce qu'on appelle communément des bugs dans le développement de logiciel.



**Figure 6.19** – Garder du mou pour le *feedback*

Dans la figure 6.19, le mou cumule l'incertitude sur les estimations et la provision pour le *feedback*. Avec un mou total de 25 %, la prévision devient 75 points de stories d'ici la fin de la *release*.

#### En résumé

Une caractéristique importante des méthodes agiles est leur capacité à prendre en compte les changements. Cela implique que les plans sont remis à jour régulièrement. C'est particulièrement vrai pour le plan de *release*, qui est actualisé à chaque *sprint*. Cette adaptation au changement s'accompagne d'anticipation : le plan de *release* permet de prendre des décisions sur le produit.

# 7

## La réunion de planification de sprint

Dans le domaine de l'estimation et de la planification, de nombreuses anecdotes illustrent les difficultés rencontrées, en voici deux :

- Certains programmeurs, investis dans le travail sur leur code, n'aiment pas trop qu'on leur demande quand ils auront fini. Parfois en insistant plusieurs jours, on peut obtenir, après un ronchonnement, « *je vais essayer de terminer demain* » et le lendemain on croise les doigts.
- Certains chefs de projet font la planification tout seuls, ils identifient des grandes tâches, les « chiffrent » et les affectent aux personnes de leur équipe. Si les tâches n'avancent pas comme prévu, le chef de projet aura beau râler, les équipiers diront que les estimations n'étaient pas现实istes.

C'est pour prévenir ce genre de situations que la réunion de planification de *sprint* existe. La planification de *sprint* est basée sur l'idée qu'on ne peut pas prévoir de façon précise au-delà d'un certain horizon. L'horizon pour la planification détaillée correspond au *sprint*.

Cette réunion met en évidence, peut-être encore plus que pour la planification de *release*, le rôle essentiel de l'équipe dans l'élaboration des plans. Le travail du *sprint* appartient à l'équipe : ce n'est pas un chef qui définit ce qu'il y a à faire, c'est l'équipe qui s'organise elle-même. Au-delà de sa fonction première de planification, la réunion est un rituel qui prépare l'équipe à travailler de façon collective pendant le *sprint*, comme les préparatifs dans les vestiaires amènent une équipe de rugby à rentrer dans son match.

## 7.1 PLANIFIER LE SPRINT

Le dogme est de considérer qu'il y a deux parties distinctes dans la réunion :

- la première pour avoir une bonne idée du périmètre et définir le but du *sprint*,
- la seconde consacrée à l'identification des tâches nécessaires pour l'atteindre et à leur estimation.

Cette distinction ne correspond pas toujours à l'usage sur le terrain et, à mon avis, contribue à la confondre avec la planification de la *release*.

Le *backlog* de produit est indispensable pour faire la planification de *sprint*. Pour une réunion efficace, il doit être prêt, c'est-à-dire :

- Les *stories* rangées par priorité.
- Les plus prioritaires estimées.
- Celles associées au *sprint* qui commence suffisamment détaillées et connues.

Si la planification de *release*<sup>1</sup> a été pratiquée correctement, le *backlog* sera prêt.

La réunion a lieu juste après la fin du *sprint* précédent et utilise aussi les résultats de la revue : la vélocité du dernier *sprint* permet de calibrer celui qui commence.

Le résultat tangible de cette réunion est un plan, contenant la liste des tâches du *sprint*. Mais le plan n'est pas l'essentiel, ce qui compte c'est la **planification** : les réflexions collectives faites au cours de ce rituel soudent l'équipe vers l'objectif du *sprint*.

### 7.1.1 C'est l'équipe qui planifie

L'équipe complète, y compris le Product Owner, participe à toute la réunion.

La présence du Product Owner est difficile à conserver sur toute la durée : comme la réunion est longue et comporte des discussions techniques, la tentation est forte, pour certains Product Owners, de n'assister qu'au début de la réunion et de s'éclipser quand commence l'identification des tâches. Même dans la seconde partie, il est indispensable qu'il soit présent pour répondre aux questions que l'équipe va inévitablement se poser. Il est obligatoire qu'il soit présent à la fin de la réunion, lors de l'engagement de l'équipe sur un périmètre.

Des experts peuvent être invités à intervenir, ponctuellement, pour apporter des éclaircissements sur des aspects fonctionnels ou techniques. Il n'y a pas d'autres personnes invitées.

1. La planification de *release* fait l'objet du chapitre 6.

## 7.1.2 Espace de travail ouvert

L'idéal est que l'équipe dispose d'un espace de travail ouvert (on parle aussi de plateau projet). Du point de vue logistique, cela signifie une salle, avec les postes de travail disposés de façon à favoriser la communication, et une zone d'affichage. Un tableau accroché au mur répond à ce besoin, à condition qu'il soit suffisamment grand, visible de tous et dispose d'un espace libre permettant d'y accéder facilement.

### Tableau des tâches mural

Un tableau de tâches sert à montrer l'avancement des travaux pendant le *sprint*, c'est une représentation physique du plan de *sprint*. Il est élaboré lors de la réunion de planification du *sprint*. Pour chaque *story* sélectionnée, l'équipe identifie les tâches correspondantes. Sur le tableau, les *stories* et les tâches sont placées avec des Post-it. L'état des tâches est reconnu selon la place de la tâche dans des zones représentant chaque état : à faire, en cours et finie.

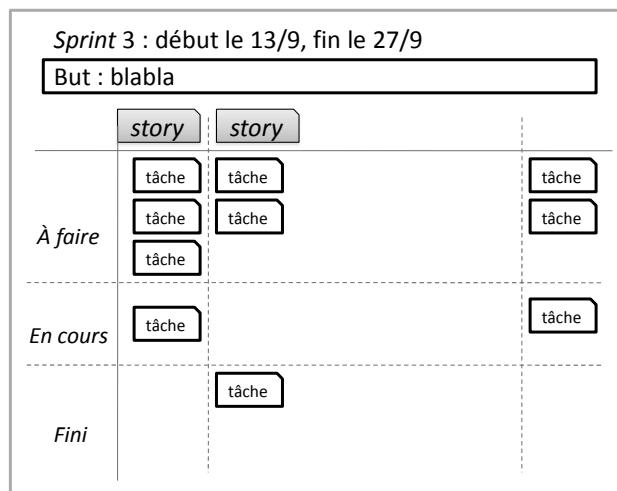
Les tâches peuvent être disposées de deux façons :

- **Tableau à disposition verticale (figure 7.1)** : les tableaux de tâches sont disposés sur un mur, avec en lignes les *stories* et leurs tâches associées et en colonnes les états des tâches.

Sprint 3 : début le 13/9, fin le 27/9			
But : blabla			
	À faire	En cours	Fini
story	tâche tâche		
story	tâche tâche tâche tâche		
	tâche tâche		

**Figure 7.1** – Tableau des tâches vertical

- **Tableau à disposition horizontale (figure 7.2)** : en haut, les *stories* sélectionnées pour le *sprint*, en dessous sont disposées les tâches qui y correspondent. Elles sont rangées dans les grandes zones horizontales selon leur état : les tâches à faire, puis les tâches en cours et en bas les tâches finies. Les tâches qui figurent à droite représentent les tâches dites *storyless*, c'est-à-dire qui ne sont pas en relation avec une *user story*.



**Figure 7.2** – Tableau des tâches horizontal

L'intérêt de se tenir devant ce tableau est de le remplir avec les informations constituant le plan de *sprint* pendant la réunion.

**Et si toute l'équipe n'est pas dans le même bureau ?**

Dans ce cas, on utilisera un outil informatique pour collecter les tâches. Voir le chapitre 17 Scrum avec un outil.

### 7.1.3 Durée de la réunion

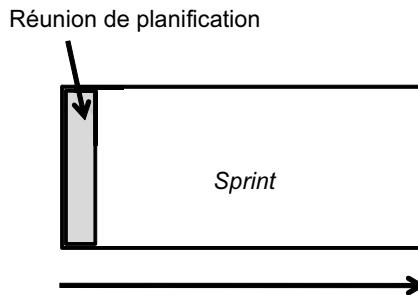
La planification de *sprint* est une séance de travail collectif, limitée dans le temps, comme toutes les réunions du cérémonial Scrum.

Ken Schwaber donne une limite de huit heures pour cette réunion, pour des *sprints* d'un mois : quatre heures pour la première partie et quatre heures pour la seconde.

Ces chiffres sont à ajuster en fonction de la durée du *sprint* : limiter à  $2*n$  heures,  $n$  étant le nombre de semaines dans le *sprint*. Pour un *sprint* de deux semaines, la réunion a une limitation à quatre heures. Ou dit autrement, la réunion ne doit pas dépasser 5 % de la durée du *sprint*. Il s'agit de durée maximum et la durée moyenne est inférieure si le *backlog* est bien préparé avant la réunion.

D'après mon expérience, il vaut mieux éviter de dépasser une demi-journée pour cette réunion, sinon la motivation de quelques personnes risque de baisser. À condition de faire une planification de *release* correcte, on y arrive, même pour des *sprints* de trois semaines.

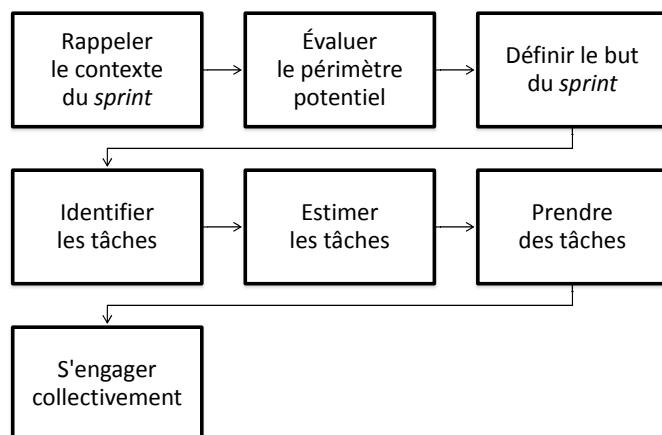
La réunion est la première activité du *sprint* qui commence.



**Figure 7.3** — La place de la réunion dans le *sprint*

## 7.2 ÉTAPES

La première partie est consacrée au quoi : le périmètre et le but, la seconde au comment, avec notamment l'identification des tâches.



**Figure 7.4** — Les étapes de la planification de *sprint*

### 7.2.1 Rappeler le contexte du sprint

Le Product Owner rappelle la place de ce *sprint* dans la *release* en cours (chaque *sprint* a un numéro séquentiel qui lui est affecté) ; il annonce la date de fin, en fonction de la durée usuelle.

Si la durée n'est pas exactement celle définie pour les *sprints*, toute l'équipe doit en connaître les raisons et y adhérer. Cela peut être dû à des vacances, des absences ponctuelles, des jours fériés...

Sprint 3
<i>Début 2 septembre - Fin le 15 septembre</i>
Disponibilité de l'équipe :
CJ : 10 j
DB : 9j
HG : 10j
AS : 10j
TF : 5j
CB : 10j

**Figure 7.5** – Contexte d'un *sprint*

### 7.2.2 Évaluer le périmètre potentiel

Il s'agit de préciser le périmètre envisagé pour ce *sprint*, c'est-à-dire les éléments du *backlog* de produit qui vont être réalisés.

Dans le cas où les membres de l'équipe ne sont pas à plein temps sur le projet, il est utile de noter leur disponibilité prévue pour ce *sprint*. Cela permet de mettre en évidence les ressources dont dispose l'équipe.

Le périmètre est défini en sélectionnant la première *story* en haut de la liste ordonnée constituant le *backlog* de produit, puis la suivante, ainsi de suite jusqu'à ce que le total corresponde à la capacité de l'équipe. Les *stories* en question sont présentées par le Product Owner à l'équipe et le dialogue qui s'installe permet à toute l'équipe d'en acquérir une bonne connaissance.

Si la planification de *release* a été bien effectuée, cette étape consiste essentiellement à valider collectivement le sous-ensemble du *backlog* prévu pour ce *sprint*. Dans ce cas, la première partie de la réunion sera rapide et nous serons bien en deçà de la limite des quatre heures.

Il s'agit d'une première évaluation de la capacité, permettant de poursuivre la réunion. Le périmètre pourra encore être ajusté avant la fin de la réunion.

**Règle** – C'est le Product Owner qui définit les priorités et donc l'ordre des *stories* candidates à être dans le *sprint*. C'est l'équipe qui est la seule à décider du périmètre, c'est-à-dire à arrêter la liste des *stories* candidates.

Le périmètre consiste en une liste de *stories* extraites du *backlog* de produit (en général, cinq à dix *stories*). La mesure du périmètre, obtenue en faisant la somme de la taille des *stories*, correspond à la capacité pour ce *sprint*. Bien entendu, la définition du périmètre tient compte de la vitesse des *sprints* précédents, mais plutôt pour avoir un ordre de grandeur que de façon précise.

### 7.2.3 Définir le but du sprint

Le but est énoncé en une phrase qui montre l'objectif principal du *sprint*. Le but d'un *sprint* est élaboré par l'équipe, à partir d'une proposition du Product Owner.

Il porte le plus souvent sur un domaine fonctionnel (au début du projet, lors des premiers *sprints* de la première *release* d'un nouveau produit, le but peut être orienté sur des considérations techniques).

**Exemples** : but du *sprint* 2 – authentification des utilisateurs, but du *sprint* 5 – mettre en place le connecteur Mylyn, but du *sprint* 93 – réaliser la sortie PDF des informations du projet...

### 7.2.4 Identifier les tâches

La suite de la réunion a pour objectif de définir comment l'équipe s'organise pour réaliser les *stories* sélectionnées.

Pour cela, on part de la liste élaborée lors de l'étape précédente ; chaque *story* est présentée par le Product Owner et étudiée par l'équipe qui identifie les tâches nécessaires pour sa réalisation. Cela force toute l'équipe à discuter pour éclaircir des points de solution par rapport à cette *story*, en demandant si nécessaire au Product Owner des précisions sur le comportement attendu.

La liste des tâches se construit progressivement avec l'examen des *stories* sélectionnées, puis en complétant avec des tâches indépendantes des *stories*. Dans les deux cas, l'identification des tâches s'appuie sur la signification de fini<sup>1</sup> pour l'équipe.

#### Tâches déduites des stories

L'ensemble des activités de développement seront déroulées lors d'un *sprint* ce qui conduit à identifier les tâches pour les réaliser.

Toutes les activités liées au développement d'une *story* doivent être prises en compte, y compris les lectures de documents ou de code si cela fait partie de la définition de fini de l'équipe.

Ce travail fait en commun pour identifier les tâches permet à toute l'équipe d'être impliquée. Elle acquiert ainsi une bonne connaissance des *stories* étudiées et surtout de la façon de les réaliser dans le produit : pendant cette réunion, l'équipe fait de la conception.

#### Tâches indépendantes des stories

En plus de celles qui découlent des *stories*, il convient également d'identifier les tâches transverses qui ne sont pas associées à une *story* spécifique (on parle aussi de tâches *storyless*).

---

1. La signification de fini fait l'objet du chapitre 11.

On peut identifier plusieurs types pour ces tâches :

- Pour un événement ponctuel et exceptionnel survenant durant le *sprint*. **Exemple** : une conférence pour laquelle il faut préparer une démo spécifique.
- Récurrentes, qui proviennent généralement de la signification de fini. **Exemple** : le travail à faire pour déployer le logiciel, à chaque fin de *sprint*, sur un serveur de test.
- Pour éliminer des obstacles identifiés dans le *sprint* précédent et pas encore éliminés.
- Pour une action décidée lors de la rétrospective.
- Pour améliorer la qualité du produit.

Les tâches indépendantes des *stories* peuvent varier à chaque *sprint*, mais il est préférable que la quantité de travail pour les réaliser reste à peu près stable : elle a un impact indirect sur la vitesse.

#### Doit-on mettre les réunions dans la liste des tâches ?

Les réunions Scrum ne sont pas incluses dans la liste des tâches, mais les autres éventuelles réunions de travail sur un sujet technique ou fonctionnel le sont.

Pour une équipe de cinq personnes et des *sprints* de deux semaines, on peut s'attendre à identifier une quarantaine de tâches pour un *sprint*.

### 7.2.5 Estimer les tâches

L'estimation du temps à passer sur une tâche est faite collectivement, par l'équipe. Il n'est pas nécessaire de passer beaucoup de temps à discuter d'une estimation : l'objectif principal est de finir une tâche (et finalement une *story*) pas de l'estimer.

Les tâches sont estimées en heures. Il est conseillé d'avoir des tâches suffisamment petites pour qu'elles soient finies en une journée de travail (si une tâche obtient une estimation supérieure à deux jours de travail, il convient de la décomposer en tâches plus petites).

La liste des tâches constituée lors de la réunion de planification n'est pas figée : des tâches peuvent être ajoutées, d'autres supprimées et d'autres décomposées pendant le *sprint*.

**Variante** – Les équipes expérimentées peuvent se passer de l'estimation en heures des tâches : elles gèrent les tâches de façon binaire : pas finie ou finie, ou avec trois états (à faire, en cours et finie).

## 7.2.6 Prendre des tâches

Ce sont les membres de l'équipe qui prennent eux-mêmes les tâches. Il n'est pas utile d'aboutir à l'attribution de toutes les tâches : il suffit que chacun ait du travail pour les premiers jours du *sprint* ; l'affectation des autres tâches est différée, elles seront prises pendant le *sprint* en fonction des disponibilités des membres de l'équipe.

### Et si une tâche pénible mais importante n'est prise par personne ?

Dans les développements traditionnels, il y a des tâches considérées comme ingrates (par exemple faire des tests unitaires, du *reporting* et autres documentations) pour lesquelles il semble difficile de trouver un volontaire. C'est en général le chef de projet qui les affecte de façon autoritaire.

En fait, j'ai côtoyé de nombreuses équipes qui passaient à Scrum et je n'ai jamais constaté le phénomène des tâches pénibles que personne ne voudrait prendre.

Il y a plusieurs raisons pour que ça ne se produise pas lors de l'application de Scrum :

- Il n'y a pas de tâches pénibles. En effet, le découpage en tâches est fait de façon radicalement différente. Lors de la réunion de planification, les tâches sont identifiées à partir des *stories* sélectionnées et servent clairement à finir quelque chose. Il n'y a pas de tâches non corrélées aux résultats concrets, comme par exemple lorsqu'on demande à quelqu'un d'écrire à la fin du projet des jeux de tests unitaires simplement parce que c'est demandé dans le processus.
- Si une tâche est malgré tout considérée comme pénible, elle sera courte. En effet, une tâche dans un *sprint* représente une journée de travail pour une personne. Cela sera perçu comme beaucoup moins pénible que, par exemple, la tâche d'écriture d'une documentation de conception pendant dix jours dans une approche traditionnelle.
- Le passage d'un mode directif à un mode autonome responsabilise chacun. Le fait que les tâches soient identifiées en commun pousse à les trouver moins ingrates que si elles sont imposées.
- L'intérêt d'une tâche pour l'avancement du projet est plus explicite. Cela évite les tâches considérées comme embêtantes et qui, en plus, ne semblent pas utiles du point de vue de celui qui la fait.
- L'estimation de l'effort est faite en commun et, de plus, il n'y a pas de relevé du temps passé sur une tâche : Scrum ne se préoccupe que du reste à faire pour finir la tâche, qui peut être actualisé. Cela met moins de pression au réalisateur de la tâche que si le délai lui est imposé.
- L'affectation des tâches aux membres de l'équipe n'est pas faite à l'avance. Les tâches sont prises de façon opportuniste en fonction de l'avancement des travaux. La question de qui va prendre une tâche a une réponse quasi naturelle en fonction de la disponibilité et de la compétence, ce qui évite les tergiversations.

Il est fréquent qu'il faille revoir le périmètre après la décomposition en tâches. Avec une idée plus précise du travail à faire, l'équipe peut décider d'en faire plus ou moins que le périmètre évalué en début de réunion. C'est une des raisons pour laquelle le Product Owner doit rester dans la deuxième partie de la réunion.

## 7.2.7 S'engager collectivement

Pour finir la réunion, l'équipe s'engage à réaliser les *stories* sélectionnées. L'engagement collectif est important pour motiver l'équipe. Il peut permettre de déceler des réticences de certains, qu'il est préférable de prendre en compte avant de finir la réunion.

Avant de demander l'engagement, le ScrumMaster annonce la capacité prévue pour ce *sprint* en la calculant à partir des *stories* dans le périmètre. Même s'il peut y avoir de légères variations, il est important que cette capacité reste dans une fourchette raisonnable par rapport à la vitesse moyenne des derniers *sprints*.

## 7.3 RÉSULTATS

Le résultat principal est le plan de *sprint*, qui contient la liste des tâches avec leurs attributs, sous une forme facilement visible ou accessible.

### 7.3.1 Plan de sprint initial

Le plan s'appuie sur la liste des tâches. Une tâche est du travail à faire pendant le *sprint*. Pendant la réunion, cette liste est produite et chaque tâche peut avoir les attributs suivants :

- **Un nom et la description du travail à faire** – Il suffit généralement d'avoir un nom permettant d'identifier la tâche et, éventuellement, du texte collecté lors de la réunion permettant de comprendre le travail à faire.
- **La story associée** – À une story sont en général associées plusieurs tâches. À part les tâches dites *storyless*, toutes les tâches sont associées à une *story*.
- **Le reste à faire estimé pour la tâche, en heures** – L'estimation de l'effort nécessaire pour réaliser la tâche est faite pendant la réunion. Cela donne une première valeur, sachant que le reste à faire peut être actualisé pendant le *sprint*.
- **La personne qui prend la tâche** pour la réaliser – Une tâche peut être réalisée par une ou plusieurs personnes. Toutes les tâches ne sont pas prises à la fin de la réunion, seules le sont un petit sous-ensemble permettant à chacun d'avoir du travail pour le jour qui vient.

Pour chaque *story*, on retrouve des tâches similaires : concevoir, coder l'IHM, coder la couche métier, faire les tests unitaires...

Pour les équipes qui sont regroupées dans le même espace, le plan est affiché sur un tableau mural. Sur ce tableau seront également notés le but du *sprint* et les dates de début et de fin.

Selon la taille de l'équipe, une ou plusieurs *stories* peuvent être commencées dès le début du *sprint*, en sortant de la réunion. Ce sont les tâches associées à ces *stories* qui sont commencées en premier.

En tout cas, il ne faut pas commencer toutes les tâches dès le début du *sprint*. Pendant le *sprint*, les tâches seront prises de façon opportuniste : quand une personne finit une tâche, elle consulte la liste des tâches qui restent libres et en prend une, en tenant compte des priorités du *sprint*.

### 7.3.2 Backlog et burndown charts actualisés

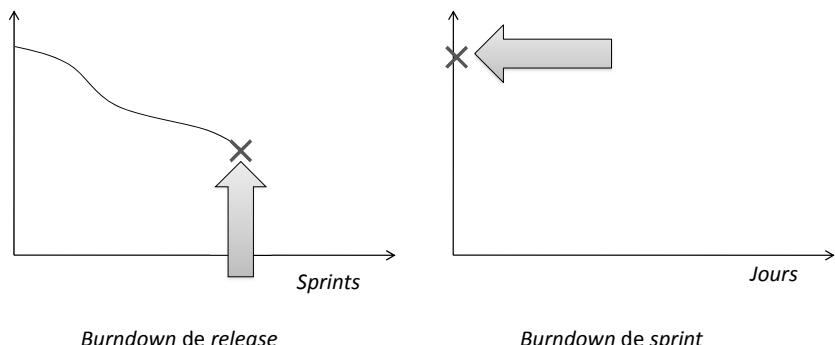
À l'issue de la réunion, le *backlog* de produit est actualisé, toutes les *stories* associées au *sprint* qui démarre changent d'état : elles passent en cours.

La mise à jour du *backlog* est l'occasion de prendre un cliché de l'état du développement. Dans ce cliché de mesures figure en première place la capacité que l'équipe pense assurer pendant le *sprint* qui commence. D'autres mesures permettent de collecter des informations sur le *backlog*, permettant d'ajuster le plan de *release*, et sur le *sprint* :

- le reste à faire dans le *backlog* de produit pour la fin de la *release*,
- le nombre d'heures de travail à faire, en faisant le total des estimations de chaque tâche.

À partir de ces mesures, des rapports graphiques peuvent être commencés ou mis à jour :

- On ajoute un nouveau point dans le *burndown chart* de *release*, avec le reste à faire dans le *backlog* de produit.
- On met le premier point du *burndown chart* de *sprint* avec le reste à faire (en heures) dans la liste des tâches.



**Figure 7.6** — Mise à jour des *burndowns*

## 7.4 GUIDES POUR LA PLANIFICATION DE SPRINT

À essayer	À éviter
Préparer le <i>backlog</i> en anticipation	Décider du périmètre à la place de l'équipe
Décomposer en tâches courtes	Ne pas laisser l'équipe identifier les tâches
Garder du mou	Prendre un engagement déraisonnable
Faire de la conception	

### 7.4.1 Préparer le backlog de produit en anticipation

Une réunion de planification de *sprint* ne peut se dérouler dans de bonnes conditions et aboutir au résultat souhaité dans la boîte de temps (*timebox*) allouée que si le *backlog* de produit est dans un état le permettant.

La préparation du *backlog*, c'est du travail qui est fait pendant le *sprint* précédent, pour arriver à la réunion avec une liste dans laquelle les *stories* sont priorisées et estimées. Ce travail, à l'initiative du Product Owner, implique aussi toute l'équipe ; on considère raisonnable qu'elle passe 10 % de son temps en anticipation sur le *sprint* suivant.

Si l'équipe considère qu'une *story* du *backlog* n'est pas compréhensible et qu'elle n'est pas en mesure d'identifier les tâches, le Product Owner est invité à revoir sa copie pour le prochain *sprint*, c'est mieux que de passer la moitié de la réunion dessus. Une bonne pratique de la planification de *release* permet de ne pas en arriver à cette extrémité.

#### Décomposer en histoires courtes

La mécanique de Scrum repose sur la réalisation de *stories* pendant un *sprint* : à la fin du *sprint*, les *stories* doivent être finies. Il convient donc que les *stories* soient suffisamment petites pour être finies pendant la durée du *sprint* choisi.

Quelques chiffres pour donner une idée de ce que signifie petit, avec l'hypothèse qu'une équipe de cinq personnes qui déroule des *sprints* de deux semaines réalise dix *stories* dans un *sprint*. Une *story* moyenne demande environ quatre jours d'effort pour son développement, dans le cas où l'équipe travaille à 80 % sur leur développement.

#### Penser aux stories techniques et aux défauts

Le travail d'identification des tâches se fait à partir de *stories* sélectionnées pour ce *sprint*. La liste comporte une majorité de tâches associées à une et une seule *story*, qui sont le reflet du travail à faire pour réaliser cette *story*.

Si le *backlog* est fait correctement, il contiendra, en plus des *user stories*, des *stories* techniques et des défauts. Il y a, bien sûr, des tâches à identifier pour tous les types de *stories*. En général, selon la définition de fini, une équipe trouve entre quatre et dix

tâches par *user story*. Pour les *stories* techniques et les défauts, le nombre de tâches associées est plus faible. Parfois il n'y aura qu'une seule tâche par *story*.

### 7.4.2 Laisser l'équipe décider du périmètre

Ce n'est pas le ScrumMaster qui décide de ce qui doit être fait. Ce n'est pas non plus le Product Owner qui impose à l'équipe le périmètre du *sprint* en lui donnant la liste des *stories* à réaliser.

Lors de la réunion, le Product Owner présente les *stories* par priorité décroissante et l'équipe dit « *stop !* » quand elle pense que sa besace est assez chargée pour le *sprint*.

Un Product Owner peut avoir tendance à demander un périmètre plus large, en insistant pour ajouter une *story* ou deux. Ce n'est pas une bonne pratique car cela risque d'avoir un effet négatif sur la motivation de l'équipe à moyen terme, si elle ne tient pas ses objectifs.

En revanche, les discussions peuvent amener l'équipe à proposer des changements dans les priorités pour faire passer une *story* non prévue à la place d'une autre dans le *sprint*. C'est au Product Owner d'accepter ou pas.

### 7.4.3 Laisser l'équipe identifier les tâches

Des ScrumMasters soucieux d'efficacité après une réunion de planification de *sprint* trop longue peuvent décider, la prochaine fois, d'arriver avec une liste des tâches déjà prête. Il est possible que la réunion soit effectivement raccourcie, mais cela a l'inconvénient énorme de moins impliquer l'équipe. En effet, si les tâches sont déjà identifiées, voire affectées, l'équipe va se sentir moins responsabilisée. Ce serait un retour à un schéma de gestion de projet avec un chef.

Cela se passait à peu près comme ça dans un projet itératif, avant Scrum : quelques jours avant la fin de l'itération  $n$ , le chef de projet prépare le plan de l'itération  $n + 1$ , tout seul (au mieux il le montre aux membres de l'équipe) ; lors de la revue de l'itération  $n$ , le plan de l'itération  $n + 1$  fait partie des livrables présentés au management et compte tenu des retours faits lors de la revue, le chef de projet ajuste le plan qui s'applique pour l'itération.

La pratique Scrum d'une équipe autonome et responsabilisée rend caducs ces allers-retours entre le chef de projet, les membres de l'équipe et le management.

### 7.4.4 Décomposer en tâches courtes

Dans la plupart des organisations, avant de passer à Scrum, la gestion des projets est basée sur une décomposition en tâches plus longues que celle conseillée dans les *sprints* : l'habitude est de définir des tâches de plusieurs jours voire plusieurs semaines.

Avec Scrum, une tâche prend en moyenne un jour. Un bon principe est d'avoir une tâche finie le lendemain du jour où a été commencée. C'est un constat qui peut être fait lors du *scrum quotidien*.

Cela veut dire que non seulement l'effort pour réaliser la tâche est limité, mais que cet effort est condensé sur un jour au lieu d'être morcelé en petites périodes sur plusieurs jours.

#### 7.4.5 Prendre un engagement raisonnable

Même sans la pression du Product Owner, une équipe qui débute avec Scrum a tendance, par excès d'optimisme, à s'engager sur plus de *stories* que ce qu'elle peut raisonnablement réaliser en un *sprint*.

Le périmètre sur lequel s'engage l'équipe est la capacité prévue.

C'est à la fin du *sprint*, lors de la revue, qu'est mesurée la vélocité réelle de l'équipe.

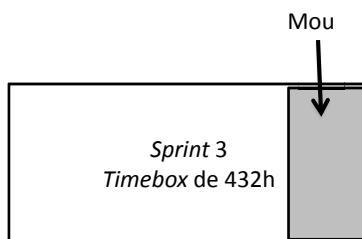
S'il peut arriver que la vélocité mesurée soit inférieure à la capacité estimée, cela ne doit pas être systématiquement le cas, surtout après plusieurs *sprints*. Une équipe doit apprendre à tenir ses engagements. Cet apprentissage passe par la mise en place de pratiques correctives lors de la rétrospective et par un engagement raisonnable.

#### 7.4.6 Garder du mou dans le plan de sprint

Même si on a mis en place une stratégie de réduction des risques, des événements inattendus viennent toujours freiner l'avancement du *sprint*, en bloquant ou ralentissant une ou plusieurs tâches en cours.

Pour empêcher ces événements de remettre en cause les engagements, il faut garder du mou lorsqu'on planifie le *sprint*.

Dans la planification d'un *sprint*, le mou, c'est du temps non affecté, qui reste disponible pour pallier les impondérables. Concrètement le mou est la différence entre les ressources de l'équipe et le total des heures associées aux tâches du *sprint* lors de la réunion de planification.



**Figure 7.7** — Garder du mou dans le plan de *sprint*

**Exemple :** si l'équipe a 432 heures disponibles pour le *sprint* et que le total des estimations sur les tâches atteint 400, il n'y a pas assez de mou et les objectifs du *sprint* ne pourront pas sûrement être tenus.

On peut différencier plusieurs types de mou dans le plan de *sprint* :

- Pour les incertitudes dans les estimations sur les tâches.
- Pour le travail en anticipation sur le *sprint* suivant.
- Pour les impondérables susceptibles de se produire.

Le pourcentage de mou varie selon le contexte, la moyenne que j'ai constatée s'établit à environ 30 %.

Dans le cas où l'équipe assure, en plus du développement, le support d'une version en production, incluant la gestion des incidents, le pourcentage est plus élevé.

Une équipe expérimentée prend du mou mais n'a pas besoin de connaître sa taille ni de suivre son évolution pendant le *sprint*.



**Figure 7.8** — Trop de mou dans le plan !

#### 7.4.7 Faire de la conception

Une erreur classique des équipes novices en Scrum est de se lancer à corps perdu dans la réalisation des *stories*. Le développement de logiciel nécessite de la réflexion. Même si Scrum n'évoque pas explicitement de pratiques d'ingénierie, il est nécessaire de faire de la conception.

Avec les méthodes agiles, la conception n'est pas faite une fois pour toutes au début du projet, elle est faite tout le temps. Chaque *sprint* comporte des activités de conception.

La réunion de planification de *sprint* est le moment idéal pour faire de la conception collective. En effet, l'identification des tâches nécessite de réfléchir à la façon dont une *story* va être conçue. Les discussions qui s'engagent permettent de partager la connaissance de cette conception entre tous les membres de l'équipe.

L'élaboration d'un modèle graphique, comme un diagramme de séquence UML, permet de mieux partager cette connaissance. La modélisation agile se fait avec un outillage léger : par exemple, un diagramme dessiné à la main, élaboré en groupe et qui restera collé au mur pendant le *sprint* pour rester visible de tous.

## En résumé

La planification du *sprint* est la première réunion du *sprint* et c'est aussi la plus délicate : son bon déroulement conditionne le succès du *sprint*.

Pour la réussir, il convient de ne pas la voir uniquement comme une séance de planification, mais aussi comme un exercice collectif pendant lequel l'équipe apprend à s'auto-organiser et à partager la connaissance sur le produit.

# 8

## Le scrum quotidien

Il y a des grands projets pour lesquels tout allait bien et on apprend un jour, en écoutant la radio le matin, qu'ils ont subitement des mois de retard.

Toutes les personnes ayant suivi des projets ont vécu ce genre de choses, même sur des projets plus petits : tout se passe bien, pas de problème, les indicateurs sont au vert et puis tout d'un coup, on apprend qu'il va y avoir un gros retard. Ce n'est pas le retard en soi qui est gênant, c'est de ne le savoir qu'au moment où on ne peut plus le cacher et qu'il est trop tard pour réagir.

Frederic Brooks, l'auteur de *Mythical Man Month*, à qui on demandait comment fait un projet pour avoir un an de retard, répondait il y a plus de 20 ans « *En prenant un jour de retard, puis un autre... »<sup>1</sup>.*

C'est pour éviter ce genre de surprise que le scrum « quotidien » a lieu tous les jours. Ce n'est pas l'unique raison d'être du scrum quotidien : c'est aussi pour développer l'esprit collectif et garder l'équipe concentrée sur l'objectif du *sprint*. Scrum, c'est la mêlée au rugby, symbole de la poussée collective.

C'est l'objectif de ce chapitre de présenter cette pratique très représentative de Scrum. Dans son sillage nous aborderons également plusieurs pratiques connexes comme l'espace de travail collaboratif, l'élimination des obstacles et le *burndown chart* de *sprint*.

---

1. <http://courses.cs.vt.edu/~cs1104/HLL/Brooks.html>, voir [page 153].

## Définitions

La réunion quotidienne se nomme officiellement le « *Daily Scrum meeting* » en anglais. La Scrum Alliance utilise simplement *daily scrum* et je vais reprendre cette appellation en parlant de **scrum quotidien** en français.

Le terme scrum est utilisé au Québec par les journalistes. Un exemple trouvé par Google : « *Je viens d'assister à mon premier scrum. Une quinzaine de journalistes entourent François Legault...* ».

Toujours au Québec, certains réagissent contre ce néologisme. Dans un article « *pourquoi pas en français ?* », on suggère de s'en passer et d'utiliser mélée de presse. Car la traduction de scrum en français, c'est mélée.

J'ai essayé un temps d'utiliser la mélée quotidienne comme traduction de *daily scrum*, mais ça n'a pas marché : au nord de Montauban, appeler une réunion mélée a un côté peu rassurant pour les participants. Dit-on un scrum ou une scrum ? Dans une équipe, on me demande à quelle heure est la scrum aujourd'hui. Dans une autre équipe, on dit le scrum est à 10 heures. Mon choix est pour **le** scrum. En revanche, la méthode éponyme s'écrit Scrum avec une majuscule et n'a pas de genre.

Cette réunion s'appelle **standup meeting** dans *Extreme Programming*, ce qui fait bien faire comprendre qu'on reste debout.

Un **obstacle** est une situation ou un événement qui peut empêcher ou retarder la progression du travail prévu au cours du *sprint*. Dans la littérature Scrum en anglais, le mot pour obstacle est *impediment*.

**Exemples d'obstacle** : un développeur n'arrive pas à *commiter* ses modifications sur le serveur SVN, la licence de l'outil de test xyz a expiré, un membre de l'équipe s'est cassé le bras au ski pendant le week-end.

## 8.1 UNE RÉUNION QUOTIDIENNE

Le scrum est plutôt facile à décrire : un point de rencontre où tous les membres de l'équipe répondent à trois questions simples et actualisent le plan de *sprint*.

Son but principal est d'optimiser la probabilité que l'équipe atteigne les objectifs du *sprint*. Les moyens pour atteindre ce but consistent en :

- Éliminer les obstacles nuisant à la progression de l'équipe.
- Garder l'équipe concentrée sur l'objectif du *sprint*.
- Évaluer l'avancement du travail pour le *sprint* en cours.
- Communiquer objectivement sur l'avancement.

### 8.1.1 Le *sprint* appartient à l'équipe

Toute l'équipe participe à la réunion. Le ScrumMaster s'assure que la réunion a lieu et que les règles sont respectées. Pendant la réunion, il n'a pas d'autre responsabilité particulière : le scrum est un des moyens permettant d'aller vers plus d'auto-organisation,

ce n'est pas une réunion pour faire un compte-rendu au ScrumMaster. La présence du Product Owner est fortement souhaitée.

Les personnes extérieures à l'équipe peuvent assister, mais n'ont pas le droit de parler.

La distinction entre l'équipe et les autres personnes a pour objectif de montrer que seuls les membres de l'équipe réellement engagés dans les travaux du *sprint* sont à même de savoir quelle attitude avoir devant une situation donnée. Cela est illustré dans l'histoire « *Pigs and chickens* », le genre de fable dont raffolent les Américains, à tel point qu'on la retrouve dans un *cartoon*<sup>1</sup>. Les Français amateurs de rugby pourront préférer cette vérité pour illustrer la distinction : « *pendant le match, le jeu appartient aux joueurs* ».

## 8.1.2 Un cérémonial balisé

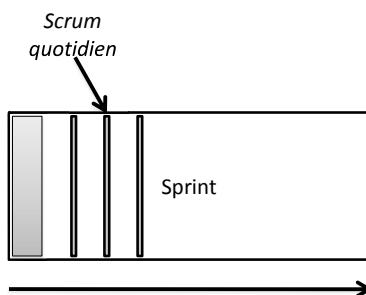
**Lieu** : si l'équipe dispose d'un espace de travail ouvert avec un tableau mural où est affichée la liste des tâches, c'est l'endroit idéal pour y tenir le scrum quotidien.

Le plan de *sprint* avec sa liste des tâches est actualisé pendant le *sprint* : quand un membre de l'équipe finit une tâche, il met à jour la liste (sans forcément attendre le prochain scrum), quand il commence une nouvelle tâche aussi. Ceux qui ont des tâches en cours au moment du scrum doivent mettre à jour le reste à faire sur ces tâches ou au moins y avoir réfléchi pour la réunion.

Si toutes les personnes de l'équipe ne sont pas dans le même espace physique, le scrum se fait avec des outils de vidéoconférence ou audioconférence.

**Durée** : le scrum dure moins d'un quart d'heure.

**Fréquence** : le scrum est quotidien.



**Figure 8.1** — Le scrum a lieu tous les jours du *sprint*

1. On trouve même une traduction en français de ce cartoon : <http://www.implementingscrum.com/translations/>

### Est-il obligatoire de faire le scrum tous les jours ?

Il est préférable de faire la réunion tous les jours.

Le succès de l'application de Scrum sur un projet passe par le respect des principes de base. Un de ceux-ci peut se résumer en *no scrum no win*, comme pour le rugby : une équipe qui ne fait pas tous les jours le scrum aura bien des difficultés à réussir son projet.

Lorsqu'une équipe ne pratique pas les réunions quotidiennes, c'est souvent le signe qu'elle n'est pas vraiment constituée comme devrait l'être une équipe Scrum. La réunion quotidienne permet de rappeler l'engagement pour le *sprint*, d'évaluer l'avancement, de définir ce qui devrait être fait pour optimiser les chances de succès. Et surtout, sans le scrum quotidien et notamment sa troisième question, l'équipe ne gère pas les risques et les obstacles, qui arrivent forcément, jour après jour.

Le premier et le dernier jour du *sprint* sont particuliers : il y a d'autres réunions du cérémonial et le scrum n'est pas fait ces jours-là.

Dans le cas de projet à plusieurs équipes, chaque équipe organise son scrum qui sera suivi du scrum de scrums<sup>1</sup> avec le ScrumMaster de chaque équipe.

## 8.2 ÉTAPES

### 8.2.1 Se réunir

La réunion s'effectue avec toutes les personnes de l'équipe, debout et en cercle, de préférence à un endroit où les tâches sont visibles de tout le monde. Il est préférable que le scrum ait lieu le matin en arrivant, tous les jours au même endroit et à la même heure. La réunion commence à l'heure prévue, même si des personnes sont en retard.

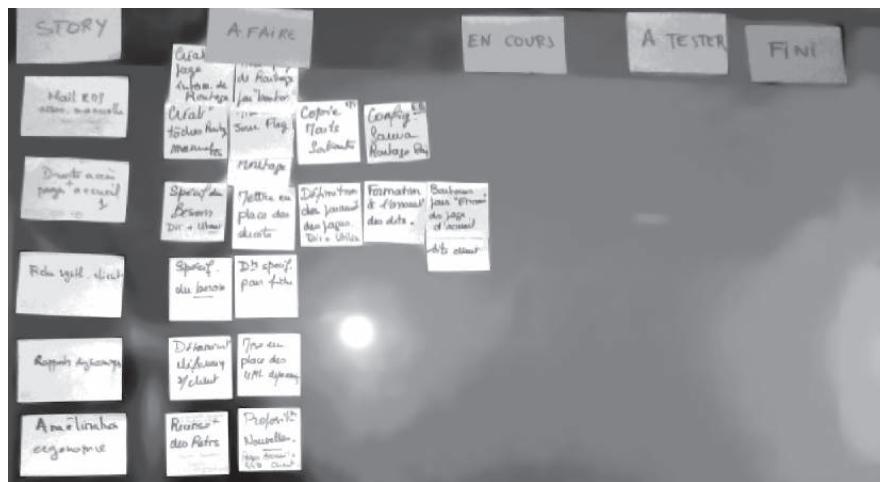
Si la réunion a lieu dans un endroit où les postes de travail sont accessibles, le ScrumMaster doit veiller à ce que les occupants du bureau enlèvent les mains de leur clavier et ne gardent pas un œil sur l'écran pendant la réunion. De même l'usage des messageries instantanées, de Facebook ou Twitter est suspendue pendant un quart d'heure...

Pour tenir les objectifs du scrum dans un délai aussi court, il est souhaitable de disposer d'une aide visuelle. Un tableau des tâches fournit cette assistance (figure 8.2).

Si on utilise un outil, on peut souhaiter s'en servir lors de ces réunions pour mettre à jour l'état des tâches. Avec un vidéo-projecteur, on projette la liste des tâches sur un écran et il est possible, éventuellement, de mettre à jour le reste à faire en direct. Cependant, il vaut mieux réserver l'outil aux équipes dispersées : cette pratique complique la logistique et l'inconvénient principal est qu'une seule personne a les

1. Pour en savoir plus sur les *scrums de scrums*, voir le chapitre 18, *La transition à Scrum*.

mains sur le clavier. Cela ne contribue pas à renforcer l'autonomie et la responsabilité de l'équipe.



**Figure 8.2** — Un tableau des tâches simple avec des Post-it

## 8.2.2 Répondre aux trois questions

### Présenter ce qui a été fait

Chaque participant répond à la première question :

*Qu'as-tu<sup>1</sup> fait depuis le dernier scrum ?*

Il s'agit, pour chaque membre de l'équipe, de parler des tâches sur lesquelles il a travaillé. Pour chacune d'entre elles, il indique si la tâche est en cours ou si elle est finie.

Pour les équipes qui utilisent un tableau des tâches mural, réalisé avec des cartes (ou des Post-it), lorsque c'est le tour d'une personne de répondre à cette question, elle déplace physiquement la carte correspondant à la tâche dans la colonne « en cours » ou « fini ».

### Prévoir ce qui va être fait

Chaque participant répond à la deuxième question :

*Que prévois-tu de faire jusqu'au prochain scrum ?*

Il s'agit de parler des tâches sur lesquelles il prévoit de travailler. Pour chacune d'entre elles, il indique en quoi elle consiste et s'il pense la finir dans la journée.

1. On se tutoie quand on fait partie de la même équipe.

### Identifier les obstacles

Chaque participant répond à la troisième question :

*Quels sont les obstacles qui te freinent dans ton travail ?*

Un obstacle empêche une tâche (ou plusieurs) de se dérouler normalement.

La troisième question est un peu plus délicate que les deux autres : dans la pratique, les réponses ne viennent pas aussi facilement qu'avec les deux premières questions. Le plus souvent les obstacles mentionnés ont déjà été rencontrés : la personne pense à ce qui l'a gêné dans les tâches qu'elle a faites. C'est au ScrumMaster de rechercher les vrais obstacles derrière les formulations vagues comme « *difficulté à communiquer avec le Product Owner* ». Un problème identifié avec cette troisième question sera formulé plus précisément comme « *j'ai proposé deux façons de procéder pour l'arrangement des boutons sur la fenêtre de validation de la story 22 et j'attends toujours la réponse du Product Owner* ». Ce qui rend l'identification des obstacles et leur élimination plus faciles...

Quand un obstacle est identifié en séance, il est possible qu'un des membres de l'équipe connaisse déjà la solution pour l'éliminer. Il est bien évident qu'il la donne immédiatement.

En revanche, si des personnes ont seulement des pistes de solution, une discussion peut s'engager, mais ce n'est pas le lieu adéquat pour la prolonger. Le ScrumMaster arrête la discussion et propose aux personnes intéressées de la repousser à plus tard, en dehors du scrum.

### 8.2.3 Statuer sur l'atteinte des objectifs

Comme la mêlée au rugby vient après une phase de désordre et permet au jeu de repartir sur de nouvelles bases, le scrum quotidien permet à l'équipe de se synchroniser par rapport aux objectifs du *sprint*.

Avec les informations collectées sur les tâches et les obstacles, l'équipe acquiert une connaissance objective de ce qu'il lui reste à faire jusqu'à la fin du *sprint*. Elle peut prendre une décision sur l'ajustement du but du *sprint*.

Lors du scrum, chacun s'engage devant ses pairs. Lorsqu'une personne de l'équipe dit ce qu'elle va faire d'ici le prochain scrum, elle l'annonce devant toute l'équipe et cela constitue un engagement fort. Comme tout le monde connaît la situation et les obstacles rencontrés, c'est plus facile de s'engager.

Quant à l'engagement de l'équipe fait au début du *sprint*, il doit rester l'objectif de tous. Certains proposent d'ailleurs une quatrième question dans le scrum : « *est-ce que, à ton avis, compte tenu de l'avancement actuel, le but du sprint sera atteint ?* »

Un processus empirique comme Scrum demande de s'appuyer sur l'expérience du jour passé pour adapter le planning des jours restant dans le *sprint*. Il faut garder à l'esprit qu'un *sprint* est limité dans le temps (le principe Scrum du *timebox*) et qu'il est toujours possible de supprimer du contenu prévu initialement (mais pas de reculer la date de fin).

L'adaptation, si elle est jugée nécessaire, peut se manifester de plusieurs façons :

- si l'équipe considère qu'elle n'est pas en mesure de montrer quoi que ce soit à la revue de *sprint*, le *sprint* est arrêté, de façon exceptionnelle ;
- si l'équipe est dans une situation où toutes les *stories* prévues au début ne pourront pas être réalisées, elle négocie avec le Product Owner pour savoir quelle *story* peut être enlevée du *sprint* ;
- si l'équipe estime pouvoir finir toutes les *stories* avant la fin du *sprint*, elle demande au Product Owner une nouvelle *story* qui pourrait être ajoutée au *sprint*.

Dans le cas où l'équipe a un peu de temps, mais pas suffisamment pour prendre une nouvelle *story*, elle se consacre à l'amélioration de la qualité ; il y a toujours à faire, par exemple du remaniement de code.

Cette inspection facilitée par la transparence, suivie d'adaptation, fait du scrum un maillon de l'approche empirique de Scrum.

## 8.3 RÉSULTATS

### 8.3.1 Le plan de sprint actualisé

Le plan évolue jour après jour : la liste des tâches peut changer et surtout l'état de quelques tâches est modifié.

Il arrive que des tâches soient découvertes pendant le *sprint* : en travaillant sur une *story*, on s'aperçoit qu'un travail important a été oublié lors de la planification du *sprint*. Il est aussi possible de supprimer des tâches devenues inutiles.

Les attributs qui reflètent le changement d'une tâche sont :

- La quantité de travail restant à faire pour la finir.
- Son état, qui peut prendre trois valeurs : à faire (quand la tâche n'est pas commencée), en cours, finie.

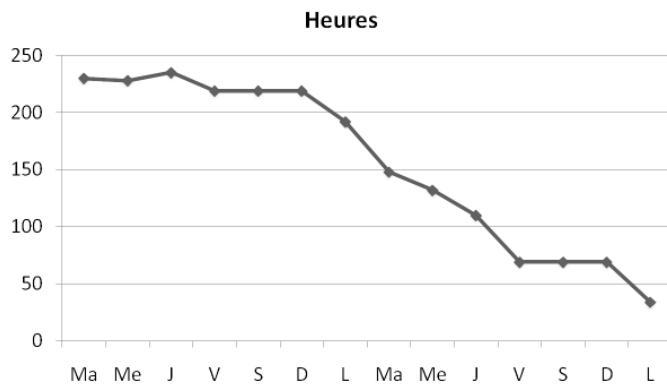
Les deux attributs sont corrélés : une tâche finie a un reste à faire de 0 et une tâche en cours a un reste à faire actualisé.

### 8.3.2 Le burndown chart de sprint actualisé

Le *burndown chart* de *sprint* est un graphe, actualisé tous les jours, montrant la tendance de l'avancement dans la réalisation des tâches du *backlog* de *sprint*.

Le *burndown chart* de *sprint* a souvent l'allure de la figure 8.3 : une descente d'abord modérée, voire pas de descente du tout (à cause de découverte de nouvelles tâches ou de travaux commencés et plus compliqués que prévus), suivie d'une décroissance plus rapide. S'il est un bon indicateur de la quantité de travail qui reste à faire, il ne

montre pas l'avancement par rapport aux objectifs : on ne voit pas si les tâches sont finies (et encore moins si les stories sont finies).



**Figure 8.3** — Un burndown chart de sprint

À l'issue du scrum, un nouveau point, calculé en faisant la somme du reste à faire des tâches, est ajouté dans le graphe. Dans ce schéma, on vient d'ajouter le point du lundi de la dernière semaine du *sprint* : il reste 34 heures.

Le burndown chart de sprint a trois variantes à considérer :

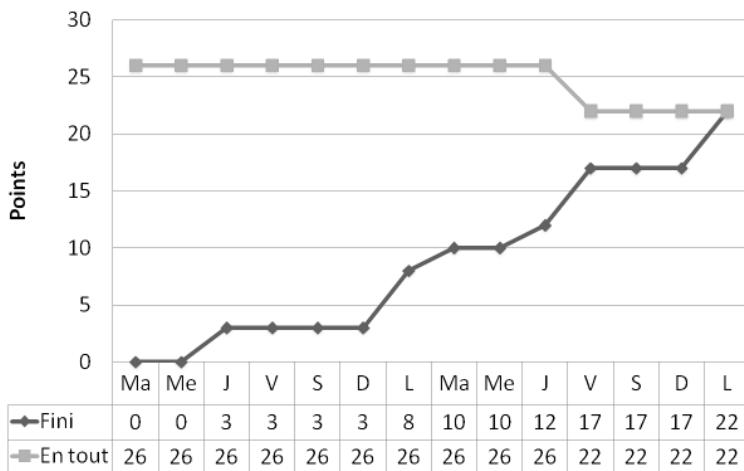
- Le burndown en tâches, basé sur le nombre de tâches restant à faire.
- Le burndown en stories, basé sur le nombre de stories restant à faire pour ce *sprint*.
- Le burndown en points, basé sur le total des points de stories restant à faire.

Ceux qui préfèrent les courbes qui montent à celles qui descendent choisiront la représentation avec un *burnup chart*. Celui qui a ma prédilection est le *burnup* en points à deux courbes : une pour ce qui est fini, l'autre pour le total des stories associées au *sprint* (figure 8.4).

### 8.3.3 La liste des obstacles

La liste des obstacles ne fait pas partie des artefacts premiers de Scrum, c'est à chaque équipe de se définir quant à son utilisation. Jeff Sutherland<sup>1</sup> évoque la liste des obstacles (*impediment backlog*) dans un billet sur les trois questions du Scrum meeting. Il dit, à propos de la troisième question : « *L'effet le plus important de cette question est de créer une liste des obstacles dont le travail d'élimination est assigné à l'équipe ou aux managers. Une responsabilité majeure du ScrumMaster est de gérer cette liste, de la prioriser et de s'assurer que sa taille diminue.* »

1. Un des fondateurs de Scrum : <http://www.jeffsutherland.com/>



**Figure 8.4** — Un *burnup* de *sprint* en points à la fin d'un sprint de deux semaines. Dans cet exemple, on voit que le périmètre du sprint d'abord à 26 points a été ramené à 22 points.

### Comment gérer cette liste des obstacles ?

Elle peut être gérée de façon très informelle. Le plus souvent, ce sera fait en listant les obstacles sur le tableau des tâches. Parfois il y a de nombreux obstacles, dont certains ne sont pas faciles à éliminer et si le ScrumMaster n'est pas à plein temps sur le projet, alors il devient nécessaire de les enregistrer.

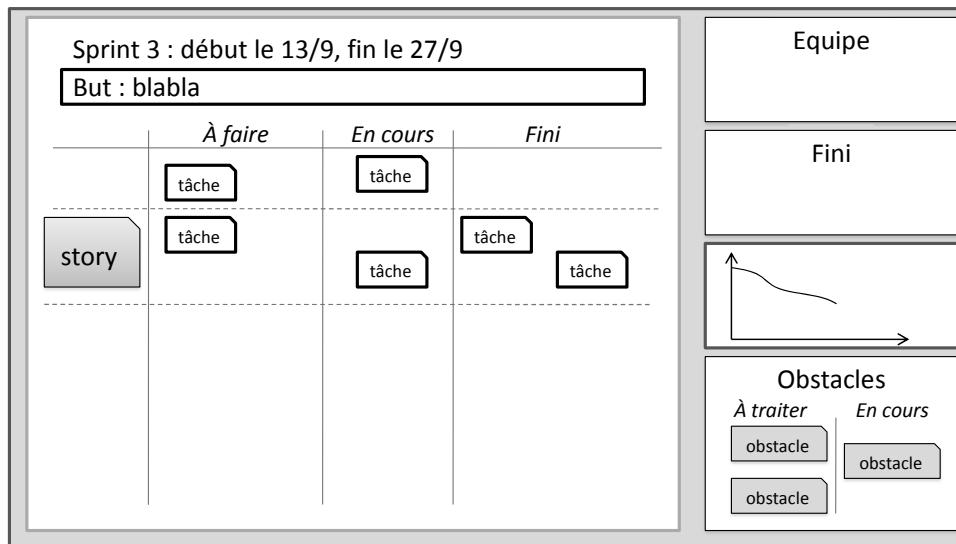
C'est la responsabilité du ScrumMaster de les classer par priorité et de faire en sorte qu'ils soient éliminés au plus vite. Certains peuvent être du ressort de l'équipe, d'autres ne peuvent avoir une solution qu'avec l'intervention de personnes extérieures, dans d'autres équipes (par exemple, si une équipe support s'occupe de l'environnement de développement) ou au niveau de la direction du projet.

### Quel est le lien de cette liste avec les tâches ?

Un obstacle bloque ou ralentit le bon déroulement d'une tâche ou de plusieurs (et pour éliminer l'obstacle, il peut être utile de créer une nouvelle tâche).

Un obstacle possède les attributs suivants :

- son nom,
- son état (identifié, en cours de résolution, résolu),
- son impact (défini par les tâches qui sont bloquées ou freinées),
- la date d'identification.



**Figure 8.5** — Tableau des tâches avec la liste des obstacles

Dans la partie droite du tableau, on trouve les obstacles avec les Post-it rangés en deux lignes en fonction de leur état : les obstacles à traiter et les obstacles en cours de résolution. Les obstacles déjà réglés sont éliminés du tableau. Au-dessus figurent l'état prévisionnel des ressources, la définition de fini et le *burndown chart* de sprint.

## 8.4 GUIDES POUR LE SCRUM

À essayer	À éviter
Faire le suivi des tâches avec les états	Dépasser un quart d'heure
Organiser des variations dans le déroulement du scrum	S'intéresser au temps passé
Finir vite les stories	Prendre un engagement déraisonnable

### 8.4.1 S'en tenir à un quart d'heure

Pour que le scrum ne dure pas trop longtemps, il faut appliquer les règles suivantes :

- Il a lieu tous les jours de travail.
- Les solutions pour éliminer les obstacles ne sont pas discutées en réunion.

### 8.4.2 Ne s'intéresser qu'au reste à faire, pas au temps passé

L'objectif du *sprint* est de finir des *stories* et pour y arriver il faut finir les tâches qui permettent la réalisation d'une story. Même si c'est utile de bien estimer, ce n'est pas le plus important. C'est pourquoi Scrum s'intéresse uniquement au reste à faire sur une tâche, pas au temps passé dessus. Pour certains, c'est un changement radical.

Chez un de mes clients, au cours du scrum quotidien, le premier du *sprint*, chacun prit la parole à tour de rôle. Quand vint le tour d'Ibrahima, il présenta ce qu'il avait fait depuis la veille, et sur quelles tâches il avait travaillé. La liste des tâches, élaborée au cours de la réunion de planification du *sprint* était affichée sur le tableau autour duquel nous étions debout, comme il se doit. Comme je lui demandais combien il restait à faire sur la tâche qu'il venait d'évoquer, il consulta la liste et y lit le nombre d'heures inscrit la veille pour cette tâche, soit 14 heures. Il réfléchit brièvement et dit : j'ai travaillé deux heures sur cette tâche alors il reste 14 moins 2 soit 12 heures à faire.

Il n'avait pas été formé à la gestion de projet agile et donc sa manière de procéder est celle de la gestion de projet classique. Elle met en évidence plusieurs aspects venant des habitudes prises :

- la première estimation est souvent considérée comme la référence,
- quand on a du mal à savoir combien il reste à faire, on trouve plus facile de prendre le chiffre de l'estimation précédente et de soustraire le temps passé.

Le reste à faire sur une tâche est obtenu en actualisant l'estimation, pas par une soustraction entre la première estimation et le temps passé. La mesure du temps passé sur une tâche n'est pas utile avec Scrum.

### 8.4.3 Faire le suivi des tâches avec les états plutôt que les heures

La réactualisation en heures du reste à faire pendant un *sprint* est difficile à obtenir des équipes. J'ai observé, sur certains projets, que :

- même si on pense qu'il faut plus de temps que prévu pour finir une tâche, on hésite à l'annoncer (en espérant se rattraper ?) ;
- même si on pense qu'on aura fini avant, on hésite à diminuer le reste à faire (de peur de se tromper ?).

Une première solution est souvent de décomposer plus finement les tâches. Si on ne sait pas dire combien il reste à faire, c'est que la tâche est trop importante, mal cernée. La décomposer en tâches plus petites permet effectivement d'y voir plus clair.

De façon générale, il est souhaitable que les tâches soient suffisamment petites pour être finies en une journée de travail.

Une solution complémentaire est de suivre les tâches uniquement par les états : lors de la réunion de planification du *sprint*, après que les tâches aient été identifiées, l'estimation des heures n'est pas faite.

Le découpage doit être suffisamment fin pour qu'une tâche puisse être finie en une journée de travail.

### Suivi avec deux états

Lors des scrums quotidiens, plutôt que d'actualiser le reste à faire en heures, on note simplement les tâches qui sont finies. Si une tâche n'est pas finie, c'est le signe qu'il y a un problème.

Le *burndown chart* montre normalement l'évolution du reste à faire pendant le *sprint*. Plutôt que de le baser sur les heures, on va simplement noter les tâches restant à faire, c'est-à-dire celles qui ne sont pas finies.

**Exemple :** il y avait 49 tâches identifiées au début du *sprint*. Le premier jour 3 sont finies, le deuxième 5... cela donne un *burndown chart* avec 49, 46, 41.

Cette pratique simplifiée me paraît adaptée aux équipes peu expérimentées dans les estimations. De plus la mesure binaire est de nature à motiver davantage à finir une tâche. Le *burndown chart* de *sprint* basé sur les heures peut être remplacé par un *burndown* portant sur le nombre de tâches qui restent à réaliser.

### Suivi avec trois états

Une tâche du *sprint* est dans un de ces trois états : à faire, en cours, finie.

Un bon suivi sur le tableau des tâches avec ces trois états permet de se passer de l'estimation du reste à faire.

#### **Pour un développeur est-il plus facile d'arriver à dire qu'il reste six heures sur une tâche ou bien de dire qu'elle est en cours et qu'elle sera finie demain ?**

Mon expérience avec de nombreuses équipes me pousse à privilégier le suivi par les états plutôt que par les valeurs en heures. J'ai constaté que pour les membres de l'équipe c'est beaucoup plus facile à vivre. Certains sont tourmentés quand on leur demande avec insistance le reste à faire. Ce n'est pas le cas quand il s'agit simplement de donner l'état de la tâche.

En se passant du reste à faire, on évite de perdre du temps à y réfléchir. Mais peut-on assurer un bon suivi ? Cela dépend des équipes et de l'expérience du ScrumMaster.

Un bon ScrumMaster saura déceler un problème quand une tâche reste en cours plusieurs jours sans se terminer, par exemple.

### 8.4.4 Veiller à finir les stories

Le scrum permet de gérer les priorités et les dépendances entre les tâches. L'objectif primaire est de finir les tâches, mais l'objectif essentiel est de finir les *stories*.

Pour qu'une *story* soit finie, il faut évidemment que toutes les tâches associées soient terminées. Cela conduit à avoir des priorités pour la prise des tâches libres.

### 8.4.5 Organiser des variations dans le déroulement du scrum

Il est fondamental que le scrum reste un moment où l'équipe reprend de l'énergie pour la journée.

Pour éviter de tomber dans la routine, le ScrumMaster peut proposer des variations dans le déroulement des scrums :

- la prise de parole se fait un coup de gauche à droite, un coup de droite à gauche ;
- la réponse aux trois questions se fait une fois à la suite pour chaque personne, une autre fois par trois tours avec une question à chaque fois.

Des accessoires peuvent renforcer la cohésion de l'équipe, par exemple :

- de la musique pour annoncer le scrum, qui change à chaque *sprint* ;
- un ballon de rugby passé de main en main pour montrer qui a la parole.



**Figure 8.6** — Un ballon, pas une boule de pétanque !

#### En résumé

Le scrum quotidien est une réunion qui se passe tous les jours, avec toute l'équipe debout qui fait le point sur le travail effectué et celui à faire.

C'est une pratique qui a prouvé son efficacité et qui ne coûte pas cher à mettre en place.



# 9

## La revue de sprint

Avant celles de Scrum, j'ai participé à de nombreuses revues. Elles portaient sur des documents, qui devaient être envoyés plusieurs jours avant la réunion pour que les participants aient eu le temps de les lire. La réunion elle-même pouvait durer longtemps. Certaines revues de fin de phase de systèmes spatiaux se déroulent pendant plusieurs jours, chaque document étant épluché et chaque remarque commentée.

L'objectif de ces revues était de connaître l'état d'un projet pour prendre des décisions sur sa poursuite. La revue de *sprint* Scrum a la même raison d'être, mais la façon de les mener est radicalement différente.

Scrum se situe dans le giron des méthodes agiles et du *Manifeste agile* qui dit que « *pour connaître l'avancement d'un développement, il vaut mieux se baser sur du logiciel fonctionnel plutôt que sur de la documentation* ». Dans un développement de logiciel, on va montrer du code qui marche.

C'est ce qui est mis en œuvre lors de la revue, avec la démonstration de l'incrément de produit réalisé pendant le *sprint*. C'est un moment essentiel de la mise en œuvre de la théorie empirique de Scrum, basée sur les notions de visibilité, inspection et adaptation.

### 9.1 LA REVUE EST BASÉE SUR UNE DÉMONSTRATION

Le but de la revue est de montrer ce qui a été réalisé pendant le *sprint* afin d'en tirer des enseignements pour la suite du projet.

### 9.1.1 La revue accueille de nombreux invités

Toute l'équipe Scrum participe à la réunion. L'équipe considérée est l'équipe étendue, avec le ScrumMaster et le Product Owner.

Toutes les personnes qui sont parties prenantes (*stakeholders*) du projet y sont invitées et leur présence vivement encouragée. On peut même y accueillir des invités qui ne sont pas directement concernés par le projet, mais intéressés à voir l'application de Scrum.

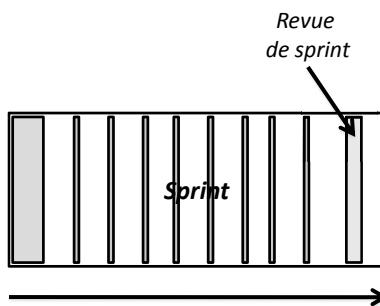
C'est la réunion Scrum à laquelle assistent le plus grand nombre de personnes. C'est l'occasion de les faire collaborer sur l'avenir du produit.

### 9.1.2 Durée de la réunion

La revue de *sprint* a lieu le dernier jour du *sprint*, elle sera suivie de la rétrospective. Pour des *sprints* d'un mois, la durée maximum est de 4 heures. Pour des *sprints* plus courts, la durée de la revue s'ajuste en fonction de celle du *sprint*.

| **Exemple :** pour un *sprint* de deux semaines, cela fait une limite de deux heures.

C'est ce que mentionne le Guide Scrum de la Scrum Alliance, mais je préconise une durée plus courte : une heure pour un *sprint* de deux semaines. Dans la pratique, cette limite est assez facile à tenir si on se tient à une démonstration qui ne porte que sur les nouveautés du *sprint*.



**Figure 9.1** — Place de la revue de *sprint*

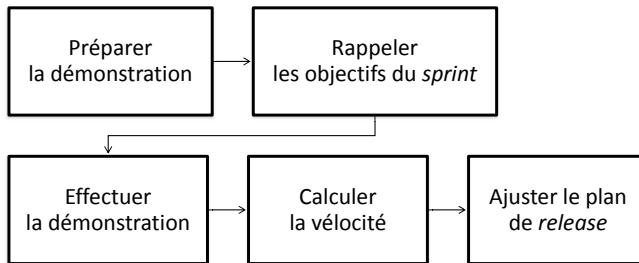
La revue nécessite une préparation, au moins pour accueillir le public et être en capacité de leur faire la démonstration. Le temps de préparation global ne devrait pas excéder une heure, car il n'y a pas de présentation formelle à faire : pas de slides à préparer.

### 9.1.3 La revue montre le produit

La revue de *sprint* porte sur le résultat du travail de l'équipe pendant le *sprint* : le produit partiel potentiellement livrable. C'est cette version opérationnelle qui est présentée.

Dans le cas d'un développement de logiciel, il a la forme d'un *build* incluant les *stories* finies, accompagné de ce qui est nécessaire pour le faire fonctionner (tests, documentation, scripts...) et déployé sur un environnement de test.

## 9.2 ÉTAPES



**Figure 9.2** — Les étapes de la revue de *sprint*

### 9.2.1 Préparer la démonstration

#### Logistique

L'équipe doit s'assurer que le matériel nécessaire pour la revue fonctionne bien. La démonstration est généralement faite dans une salle pouvant accueillir du monde et disposant d'un vidéoprojecteur.

Une erreur classique est de ne pas vérifier les connexions réseau de la salle si l'application en a besoin. J'ai vu une revue de *sprint* perdre une heure à cause de ce détail.

Si des participants sont à distance, la revue nécessite l'emploi d'un système de vidéoconférence.

#### Environnement de démonstration

La préparation de la réunion consiste aussi à imaginer un scénario d'enchaînement des différentes *stories* qui seront montrées, avec éventuellement des jeux de données associés. La démonstration se fait de préférence sur un environnement de test, le plus proche possible de l'environnement de production, et surtout pas sur l'environnement de développement.

L'équipe se met d'accord sur le déroulement de la démonstration et sur quelles personnes vont la présenter, afin qu'il n'y ait pas de temps perdu à organiser cela pendant la réunion.

### 9.2.2 Rappeler les objectifs du sprint

Le Product Owner rappelle le but du *sprint* défini lors de la réunion de planification, en début de *sprint*. Il donne la liste des *stories* qui étaient dans le périmètre prévu et annonce celles qui vont effectivement être montrées lors de la revue.

S'il y a des différences, l'équipe intervient pour en donner brièvement les raisons, sachant que les causes seront examinées lors de la rétrospective.

### 9.2.3 Effectuer la démonstration

L'équipe présente le produit partiel, résultat de ses travaux, en faisant une démonstration des *stories* réalisées.

Seules les *stories* complètement finies (selon la définition de fini) sont présentées. Cela permet d'avoir une mesure objective de l'avancement. La démonstration est faite en indiquant une à une les *stories* présentées.

#### *Qui fait la démonstration ?*

La démonstration peut-être faite par l'équipe, avec chaque personne de l'équipe montrant la *story* sur laquelle elle s'est particulièrement impliquée. Le risque est le manque d'homogénéité. De plus il arrive que certains développeurs aient tendance à avoir le clic trop rapide pour un public composé d'utilisateurs.

Il est préférable que ce soit le Product Owner qui fasse la démonstration, cela a l'avantage de l'obliger à s'impliquer dans le test du produit avant la revue.

Les meilleures démonstrations que j'ai vues étaient faites par un binôme constitué du Product Owner et d'un membre de l'équipe : le Product Owner parle et l'équipier manipule.

#### *Impliquer les participants*

Les participants à la réunion sont invités à poser des questions à l'équipe et à donner leur impression sur le produit montré. Leur *feedback* se concrétise en propositions et demandes de changement. Le *backlog* de produit est enrichi avec les éventuels défauts découverts et les demandes d'évolution suggérées.

Si c'est possible, les personnes présentes à la réunion sont invitées à manipuler le produit à la fin de la démonstration.

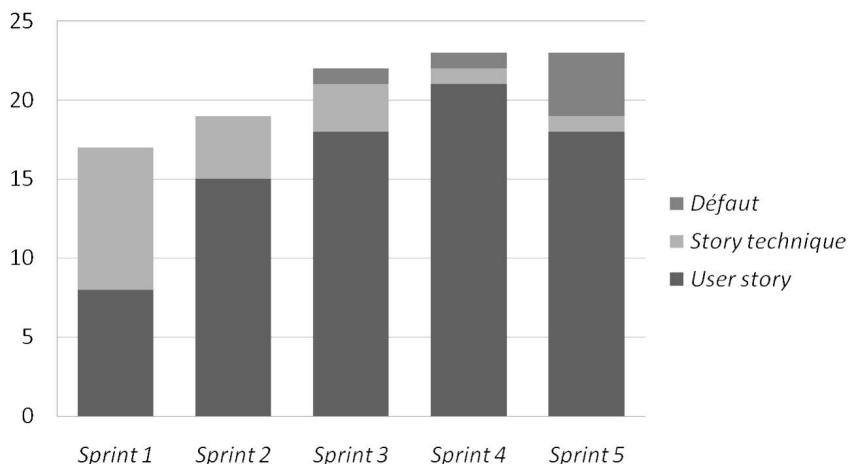
### 9.2.4 Calculer la vélocité

La liste des éléments du *backlog* considérés comme finis est établie en commun. En principe, si l'équipe n'a présenté que les *stories* vraiment finies, la liste est connue à l'avance, mais il est préférable d'impliquer tout le monde dans la décision. La vélocité du *sprint* est obtenue en faisant la somme de tous les points attribués à ces éléments.

#### Que faire si une *story* est presque finie ?

Il n'y a pas de *stories* moitié finies ou à 90 % finies lorsqu'il s'agit de mesurer la vélocité : seules les *stories* qui passent les tests d'acceptation et respectent la signification de fini comptent. Toutes les autres ne contribuent pas à la vélocité.

La vélocité obtenue est comparée à celles des *sprints* précédents.



**Figure 9.3** — Le graphe de vélocité

La figure 9.3 montre la vélocité avec une classification des *stories* selon leur type.

La vélocité moyenne est calculée en tenant compte de celle ajoutée pour le *sprint*. Cette nouvelle vélocité moyenne sera utilisée pour ajuster le plan de *release*.

### 9.2.5 Ajuster le plan de *release*

Les conditions ont changé depuis la dernière planification de la *release* : des éléments du *backlog* ont pu être ajoutés ou supprimés, les estimations modifiées, l'ordre dans lesquels les éléments sont classés (la priorité) a pu être modifié, la vélocité moyenne a pu évoluer.

Le Product Owner présente le plan de *release* ajusté (s'il y a eu un *feedback* important, cet ajustement ne pourra pas se faire en séance et sera communiqué après la réunion).

C'est l'occasion de discuter avec les intervenants sur le contenu de la *release* et sur sa date de fin. Des projections peuvent être faites en tenant compte d'hypothèses sur la capacité de l'équipe et sur les variations de périmètre.

Il est possible de prendre une décision sur la date de la mise en production de la *release*, ou sur son contenu, si cela n'a pas été fait avant.

## 9.3 RÉSULTATS

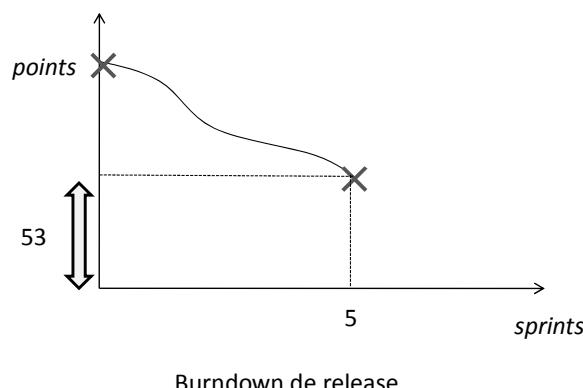
### 9.3.1 Backlog de produit actualisé

Le *backlog* de produit est mis à jour :

- les *stories* du *sprint* déclarées terminées changent d'état et passent dans la zone du *backlog* réservée aux *stories* finies,
- le *feedback* des participants à la démonstration peut entraîner la création de nouvelles *stories*.

### 9.3.2 Plan de release et burndown chart de release mis à jour

Le plan de *release* est impacté par les modifications sur le *backlog* et par la mesure de la vélocité. Le *burndown chart* de *release* est complété avec un nouveau point pour le *sprint* qui s'achève, calculé à partir du *backlog*.



**Figure 9.4** — Mise à jour du *burndown chart* de *release*

Dans la figure 9.4, il reste 53 points dans le *backlog* à la fin du *sprint* 5.

## 9.4 GUIDES POUR LA REVUE

À essayer	À éviter
Dérouler un scénario	Modifier le produit au dernier moment
Inviter largement mais expliquer que c'est un produit partiel	Parler processus
Parler des <i>stories</i>	Parler des tâches
Solliciter le <i>feedback</i>	Ne pas finir les <i>stories</i>
En faire la réunion essentielle sur le produit	Faire un compte-rendu

### 9.4.1 Dérouler un scénario

Une démonstration de fin de *sprint* n'est pas une présentation marketing avec des paillettes, ce n'est pas non plus une recette fonctionnelle avec le passage roboratif des tests.



**Figure 9.5** — Le Product Owner fait sa revue en tutu (et gilet pare-balles au cas où)

Pour que l'auditoire comprenne bien, il faut que la démonstration soit fluide. Il peut être utile de préparer un scénario. Ce n'est pas bien difficile, puisqu'on dispose de la liste des *stories* ; il suffit d'imaginer un enchaînement de ces *stories* et de réfléchir, pour chacune, à la façon de les présenter. Ce scénario peut être écrit, ce n'est pas du travail inutile, il pourra être laissé aux utilisateurs du produit partiel après le *sprint*.

**Attention :** la revue n'est pas l'endroit où l'on passe les tests d'acceptation : ils sont passés avant et ne sont montrées que les *stories* les ayant passés avec succès.

#### 9.4.2 Inviter largement, mais expliquer que c'est un produit partiel

La revue de *sprint* est l'occasion de présenter le produit et il faut rechercher l'audience la plus large possible.

Des invitations sont envoyées à toutes les personnes impliquées directement ou indirectement dans le futur produit.

Dans certaines organisations, on rechigne à inviter des personnes extérieures tant que le produit n'est pas suffisamment complet. Il existe la crainte que les personnes soient frustrées parce que le produit ne contient pas tout ce qu'elles attendent. Ce serait une erreur de se priver des bénéfices d'une revue avec des personnes extérieures. Le mieux est de les former à Scrum si c'est nécessaire et de les inviter à la revue.

Au début de la réunion, il est bon de leur rappeler qu'il s'agit d'un produit partiel et que les autres fonctions du produit seront réalisées ultérieurement. Une bonne façon de procéder est de présenter le plan de *release* actuel.

#### 9.4.3 Éviter de modifier le produit partiel au dernier moment

Le logiciel, c'est souple, des modifications peuvent être faites rapidement. À quelques minutes de la revue, si l'équipe découvre un défaut, il est tentant de le corriger. Il faut résister à cette envie, c'est trop tard. On pourrait se dire que si ça ne marche pas, ce n'est pas grave. En fait, cela peut l'être : une modification peut entraîner des régressions. Si le temps ne permet pas de passer des tests qui permettent de s'assurer qu'il n'y a pas de régressions, alors il ne faut pas modifier le code.

J'ai entendu de nombreuses équipes, et pas seulement des équipes d'étudiants, qui, voyant leur démo *planter*, se sont exclamées, de bonne foi : « *mais ça marchait tout à l'heure* ».

Pour ne pas risquer le fameux « effet démo », il est conseillé de ne pas toucher au code le dernier jour.

##### Montrer uniquement ce qui est fini

Quand c'est l'équipe qui présente, et en particulier quand c'est un développeur qui fait la démonstration, la tentation est forte de montrer tout le travail réalisé, même si ce n'est pas fini.

Il faut considérer la démonstration comme un résumé du travail du *sprint* où on ne montre que les actions abouties, de la même façon qu'un résumé de match de rugby ne porte que sur les essais marqués.

#### 9.4.4 Parler des *stories*, pas des tâches

La revue de *sprint* est un moment privilégié pour la transparence : l'équipe montre au public ce qui fonctionne. Il s'agit aussi d'inspecter le résultat, en particulier par rapport aux objectifs de début de *sprint*. Le Product Owner présente les *stories* prévues, et ce qui a été effectivement réalisé.

Certaines équipes donnent trop d'importance au *burndown chart* de *sprint* et le présentent lors de la revue. C'est une erreur : il est à usage de l'équipe pendant son *sprint*, seules les *stories* finies intéressent les invités de la revue. Ce qu'on peut leur montrer, c'est le *burndown chart* de *release*, mais celui de *sprint*, non.

Ce qui compte pour les invités de la revue, c'est le produit partiel avec les *stories* qu'il contient. Le déroulement du *sprint* et en particulier les tâches qui sont finies ou pas, cela ne concerne que l'équipe et ce n'est pas l'objet de la revue.

#### 9.4.5 Solliciter le feedback

Un processus itératif fonctionne avec le *feedback*. La revue de *sprint* est l'endroit idéal pour le solliciter. Les personnes présentes peuvent intervenir pendant la revue et poser des questions lors de la démonstration. L'équipe y répond, soit en précisant un point mal compris, soit en expliquant que cela est déjà dans le *backlog* et sera fait plus tard, soit en ajoutant une entrée au *backlog*.

Le *feedback* pendant la revue reste limité : les personnes ne manipulent pas elles-mêmes en général. C'est pourquoi il faut pousser à utiliser le résultat du *sprint* après la revue : pour cela, un environnement spécifique peut être mis à disposition des personnes qui souhaitent essayer le produit et faire ainsi du *feedback* plus complet.

#### 9.4.6 En faire la réunion essentielle sur le produit

La présence des personnes impliquées dans le produit rend inutiles la tenue d'autres réunions.

La revue permet d'éviter des démonstrations spécifiques pour des personnes importantes qui n'ont pas daigné se déplacer. C'est de la perte de temps alors que la revue est prévue pour ça. En revanche, il peut toujours être nécessaire d'organiser d'autres démonstrations pour des clients.

Si quelqu'un d'important ne peut pas assister à la revue, ce n'est pas dramatique, la prochaine reviendra vite, au rythme des *sprints*.

Les organisations qui fonctionnent d'habitude avec des comités (comité de pilotage, comité de suivi) devraient les supprimer pour ne conserver que les revues de *sprint*.

### **Ne pas faire de compte-rendu de réunion et ne pas parler processus**

Les personnes qui assistent n'ont pas besoin de compte-rendu, elles ont vu le produit dans son état actuel. Les autres n'avaient qu'à venir ! Sinon, elles ont toujours accès au *backlog* de produit, au plan de *release* et au *burndown chart* de *release* qui sont publiés et diffusés largement. Dans la mesure du possible, ces artefacts sont mis à jour pendant la réunion.

Le processus, c'est l'objet de la rétrospective, qui a lieu juste après. La revue est consacrée au produit.

## **En résumé**

La revue de *sprint* est l'occasion de faire partager les réalisations de l'équipe avec le reste de l'organisation. La visibilité apportée et le *feedback* reçu permettent d'augmenter les chances que le produit soit un succès.

# 10

## La rétrospective de sprint

Dans de grandes entreprises, quand un projet touche à sa fin, le service méthodes ou son équivalent rappelle au chef de projet qu'il faudrait faire un bilan avant que l'équipe ne se disloque. Le bilan de projet alors mené est l'occasion de revenir sur la façon dont s'est déroulé le projet et de faire une liste des faits marquants.

Malheureusement le bilan de projet arrive trop tard : le projet est fini. On peut toujours se dire que d'autres projets bénéficieront des résultats du bilan. Encore faudrait-il que la capitalisation s'opère bien dans l'entreprise. Dans la plupart des entreprises, ce genre d'initiative n'existe même pas.

C'est pour permettre à une équipe de s'améliorer régulièrement que la rétrospective de *sprint* existe. Avec Scrum, on n'attend pas la toute fin du projet, elle fait partie du cérémonial de chaque *sprint*.

En prenant l'analogie d'un match de rugby pour le sprint, on peut comparer la rétrospective à la discussion sur « *on refait le match* », mais à laquelle participeraient uniquement les joueurs.

Le *Manifeste agile* rappelle que les personnes et les interactions entre elles sont plus importantes que les processus et les outils. Pourtant la rétrospective, c'est de l'amélioration de processus, ne serait-ce pas contradictoire ?

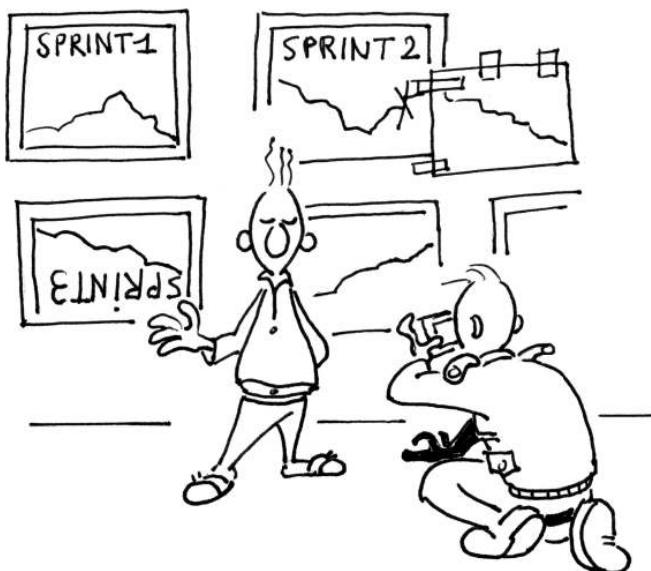
Avec Scrum, le processus n'est pas imposé à l'équipe c'est elle qui le construit régulièrement à partir du cadre proposé par Scrum, et en particulier lors des rétrospectives. À partir des expériences tirées du développement sur le *sprint* courant, l'équipe identifie ce qui marche bien pour elle, ce qui marche moins bien et trouve collectivement ce qu'il faut modifier au processus qu'elle utilise.

Un des principes du *Manifeste* l'énonce : « *À intervalles réguliers, l'équipe réfléchit à comment devenir plus efficace, puis adapte et ajuste son comportement en conséquence.* »

En résumé, le but de cette réunion est l'évaluation par l'équipe de sa façon de travailler pour trouver un moyen de l'améliorer dans le prochain *sprint*.

### Définition

Le terme est surtout utilisé dans le domaine artistique pour évoquer la présentation de l'œuvre d'un créateur de façon chronologique : on entend que France 3 propose une rétrospective Hitchcock pendant l'été et on lit dans *La Dépêche* que Les Abattoirs exposent du Saura pour une rétrospective.



**Figure 10.1** — Un ScrumMaster mégalo pose pour la rétrospective de ses *sprints*

C'est Norm Kerth<sup>1</sup> qui est à l'origine de l'utilisation de rétrospective pour le développement, il la définit comme : « *un rituel à la fin d'une étape pour acquérir de la connaissance et décider d'améliorations à mettre en œuvre lors de la prochaine étape* ».

La rétrospective « agile » est maintenant passée dans le langage courant.

## 10.1 UNE PRATIQUE D'AMÉLIORATION CONTINUE

Une équipe a été sensibilisée à Scrum et essaie de l'appliquer. À la fin d'un *sprint*, la rétrospective est le moment pour se poser des questions sur la façon dont elle a travaillé. Il est normal de procéder à des adaptations du processus, en fonction du vécu sur le *sprint* qui se termine.

1. *Project Retrospectives : A Handbook for Team Reviews*- Dorset House, 2001

Ces adaptations se poursuivent pendant toute la vie du produit : en effet, il y a toujours des améliorations à faire, il n'existe pas de processus ultime.

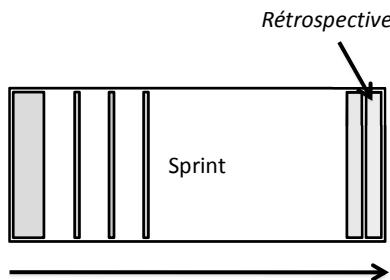
### 10.1.1 Un moment de réflexion collective à la fin de chaque sprint

La rétrospective constitue un moment particulier où l'équipe s'arrête de produire, prend le temps de réfléchir et parle de ses expériences, avec l'objectif de :

- capitaliser sur les pratiques qui ont marché,
- éviter de refaire les mêmes erreurs,
- partager différents points de vue,
- permettre au processus de s'adapter aux nouvelles avancées dans la technologie utilisée pour développer.

Ce travail est réalisé en groupe au cours d'une réunion, dont la durée est limitée, comme toutes les autres. La règle est de ne pas dépasser trois heures pour un *sprint* d'un mois.

D'après mon expérience, elle dure en moyenne une heure et a lieu dans la même demi-journée que la revue de *sprint*.



**Figure 10.2** — Place de la rétrospective

### 10.1.2 C'est l'équipe qui refait le match

Toute l'équipe Scrum participe à la réunion. L'équipe considérée est l'équipe étendue, avec le ScrumMaster et le Product Owner.

La rétrospective a généralement lieu juste après la revue de *sprint* et les intervenants qui sont venus y assister peuvent rester pour la rétrospective, à titre d'observateurs. Cependant, la confiance est nécessaire pour le succès d'une rétrospective et la présence de certaines personnes peut être un obstacle à son instauration.

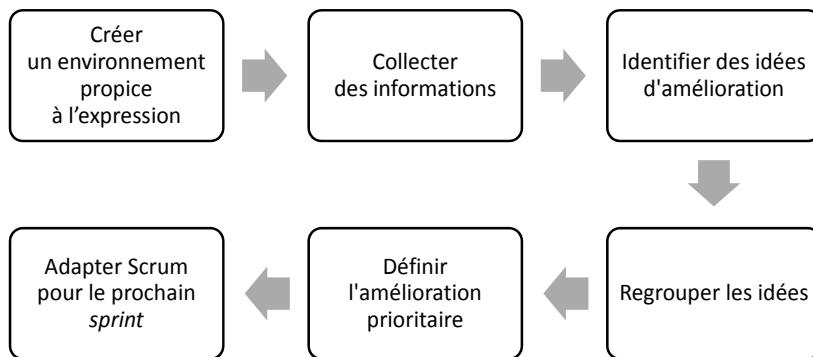
La réunion est animée par le ScrumMaster. Mais, notamment dans les environnements difficiles, il est préférable que ce soit une personne extérieure à l'équipe qui joue le rôle de facilitateur de cette réunion.

Une rétrospective représente pour les participants une possibilité d'apprendre comment s'améliorer. L'idée est que ceux qui réalisent sont les mieux placés pour savoir comment progresser. L'objectif est l'apprentissage, et non la recherche de fautes.

L'équipe est impliquée dans la mesure où chacun doit :

- comprendre le besoin d'amélioration,
- concevoir les améliorations,
- s'approprier les améliorations,
- être motivé pour s'améliorer.

## 10.2 ÉTAPES



**Figure 10.3** — Les étapes de la rétrospective

### 10.2.1 Créer un environnement propice à l'expression

Il s'agit de s'assurer que tout le monde se sent en confiance et pourra s'exprimer librement pendant la réunion. Norm Kerth a défini le principe qui doit guider les participants :

Indépendamment de ce que nous allons découvrir aujourd'hui, nous comprenons et nous croyons vraiment que chacun a fait de son mieux, en fonction de ses connaissances, de ses compétences et de ses capacités, des ressources disponibles et de la situation courante.

Il s'agit de regarder le passé, non pour juger le comportement de certains ni pour faire son autocritique, mais dans le but de s'améliorer en tant qu'équipe.

S'il y a des doutes sur le climat de confiance qui entoure la réunion, il est préférable de demander à chacun, à bulletin secret, s'il se sent en capacité de s'exprimer librement. Si l'animateur constate que ce n'est pas le cas, la rétrospective ne peut pas avoir lieu, il faudra trouver des conditions différentes qui rétablissent la confiance.

### 10.2.2 Collecter les informations sur le sprint passé

Dans une rétrospective, on commence par regarder en arrière : avant de chercher à améliorer les pratiques, il convient de collecter le ressenti des participants sur ce qui s'est passé pendant le *sprint* qui s'achève.

L'approche basique consiste à poser des questions à chaque participant, de façon un peu similaire à ce qui est fait dans le scrum quotidien :

- Qu'est-ce qui a bien fonctionné ?
- Qu'est-ce qui s'est mal passé ?
- Qu'est-ce que nous pouvons améliorer ?

Cependant le risque avec cette démarche un peu brutale est de ne pas collecter beaucoup d'informations, aussi il est préférable d'utiliser une approche plus progressive.

Une technique courante qui facilite la collecte est la présentation chronologique de la vie de l'équipe pendant le *sprint*, appelée *timeline* : la ligne de vie est tracée avec les événements marquants qui ont jalonné la période. On peut s'appuyer sur la liste des obstacles rencontrés pour se remémorer les péripéties du *sprint*. Les événements sont annotés selon leur perception par l'équipe : agréables, frustrants, fâcheux.

### 10.2.3 Identifier des idées d'amélioration

L'objectif d'une rétrospective n'est pas seulement de poser des questions sur le passé, mais surtout celui d'apporter des réponses concrètes qui seront mises en place dans la prochaine itération.

Les informations collectées dans l'étape précédente sont complétées avec les idées d'amélioration proposées par l'équipe.

Il existe de nombreuses techniques, sous forme d'ateliers, permettant d'y arriver. Je conseille de commencer par une réflexion individuelle, pendant laquelle chaque participant note ses idées sur des Post-it (un par idée). Une fois que chacun a identifié au moins quatre idées et que l'inspiration se tarit, les Post-it sont collés sur un mur.

### 10.2.4 Regrouper les idées

Les informations sont regroupées en catégories, de façon à en obtenir entre cinq et dix. Les catégories correspondent en général à des pratiques agiles et aux outils qui les supportent. Chacune est nommée pour que tout le monde l'identifie clairement.



**Figure 10.4** – Les catégories identifiées après regroupement des idées d'amélioration

Ce travail est fait collectivement, ce qui est facilité par l'utilisation de Post-it. Lors des rétrospectives que j'anime, il m'arrive de demander un regroupement en silence, ce qui permet de dérouler cette étape plus rapidement, les discussions ayant eu lieu lors de l'étape précédente.

### 10.2.5 Définir l'amélioration prioritaire

Il s'agit de définir sur quelle pratique va être porté l'effort d'amélioration. Pour un sprint de deux semaines, il est préférable de n'en sélectionner qu'une seule.

La technique de sélection doit permettre la participation de toute l'équipe, avec par exemple un système de votes. Sur la figure 10.4, on voit le résultat d'un vote, pour lequel chaque participant disposait de cinq points à répartir sur la ou les catégories de son choix.

Quelle que soit la technique utilisée pour choisir, l'objectif est de renforcer l'implication de chacun et de le motiver pour mettre en œuvre l'amélioration.

### 10.2.6 Adapter Scrum pour le prochain sprint

Pour la pratique choisie en vue de l'améliorer, il s'agit de définir les actions concrètes qui vont être menées lors du prochain *sprint*. Les actions sont identifiées par l'équipe, en réfléchissant à toutes les tâches nécessaires.

À la fin de la réunion, il est conseillé que l'équipe revoie sa signification de fini<sup>1</sup> : elle étudie chaque élément de la liste, décide de conserver ou pas pour le prochain sprint et en ajoute éventuellement.

1. La signification de fini fait l'objet du chapitre 11.

## 10.3 RÉSULTAT

Le résultat de la rétrospective est la liste des actions qui ont été décidées. On peut classer les actions selon leur orientation :

- celles dirigées vers les personnes et leur façon d'appliquer les pratiques,
- celles orientées outils à installer ou à adapter,
- celles axées sur l'amélioration de la qualité.

Les actions impactant les personnes sont par exemple : limiter le scrum à 15 minutes, faire une heure de travail en binôme tous les jours, modifier la façon de prendre en compte les bugs...

**Exemple :** la limitation des scrums quotidiens à un quart d'heure est l'amélioration choisie par l'équipe. Les actions identifiées sont : apporter un minuteur, annoncer le temps écoulé, afficher la durée tous les jours. Une personne de l'équipe propose d'apporter son minuteur. Les autres actions trouveront des responsables au moment du scrum. L'amélioration choisie et les tâches à faire restent bien visibles pendant le *sprint*, dans l'espace de travail.

Les actions relatives aux outils nécessitent généralement des études d'impact avant généralisation éventuelle. Par exemple, si l'équipe décide de mettre en place un outil de test automatique, elle ne peut pas le faire dès le prochain *sprint*. Il faut au préalable étudier plusieurs outils, en choisir un, former l'équipe... L'étude devient une *story* technique, prioritaire pour le *sprint* à venir.

Les actions relatives à la qualité, comme augmenter le taux de commentaires dans le code ou la couverture de tests vont dans la signification de fini.

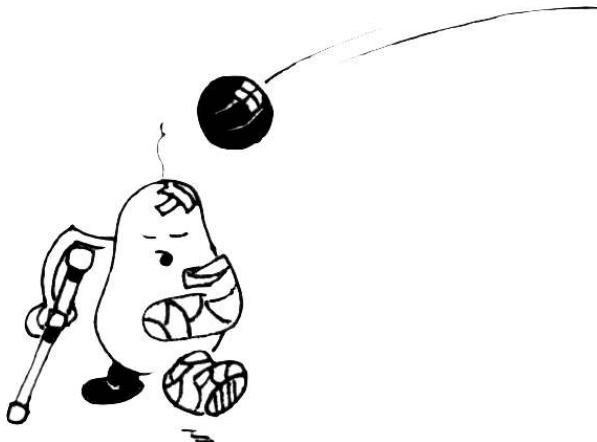
Les idées d'amélioration identifiées et non sélectionnées pour le prochain *sprint* sont placées dans le *backlog* de produit.

## 10.4 GUIDES

À essayer	À éviter
Parler de ce qui va bien	Régler ses comptes
Se concentrer sur une seule amélioration	Vouloir tout améliorer d'un coup
Mener des rétrospectives plus poussées en fin de <i>release</i>	Ne pas faire aboutir les actions décidées
Utiliser un facilitateur externe	Arrêter de faire les rétrospectives

### 10.4.1 Ne pas en faire une séance de règlement de comptes

La rétrospective permet à toute l'équipe de s'exprimer et chacun est invité à participer à l'identification des dysfonctionnements constatés pendant le *sprint*. La mise en évidence d'erreurs ne doit pas conduire à accuser une personne d'en être responsable.



**Figure 10.5** — La rétrospective ne doit pas tourner au règlement de comptes

Le ScrumMaster doit veiller à ce que la rétrospective ne dérive pas vers les attaques personnelles, en faisant de l'équipe l'objet central des discussions.

Il arrive aussi que l'équipe estime que des dysfonctionnements sont provoqués par des entités extérieures. C'est fréquent : l'équipe n'est pas dans sa bulle, elle n'est jamais complètement indépendante.

Il est difficile de définir des actions dont les responsables ne participent pas à la réunion. Le ScrumMaster doit remonter les problèmes à la hiérarchie et tout faire pour qu'ils soient traités. Cependant, l'objectif de la rétrospective est d'abord de se concentrer sur son processus et d'identifier des actions que l'équipe peut mettre en œuvre elle-même.

#### 10.4.2 Parler de ce qui va bien

Comme l'objectif est d'améliorer des façons de faire, la collecte passe essentiellement par la mise en évidence de pratiques qui ne se passent pas bien, en tout cas pas aussi bien qu'elles le pourraient.

Pour équilibrer ce point de vue qui peut être perçu comme négatif, il convient aussi de mettre en exergue ce qui s'est bien passé pendant le *sprint*, les pratiques qui ont apporté de la satisfaction à l'équipe.

Cela permet de construire une communauté à partir d'aspects positifs.

#### 10.4.3 Faire aboutir les actions des rétrospectives précédentes

Il n'y a rien de plus frustrant pour les participants à une rétrospective que de constater, *sprint* après *sprint*, que rien ne bouge vraiment. C'est le rôle du ScrumMaster de s'assurer que les actions décidées sont vraiment mises en œuvre.

Parfois, les équipes se lassent des rétrospectives, estimant qu'il n'y a plus rien à améliorer, ou que « *nous disons toujours la même chose* ». L'erreur la plus grave est d'arrêter de faire des rétrospectives. Ne pas utiliser cette possibilité de *feedback* sur la façon de travailler, c'est passer à côté d'un des bénéfices majeurs des méthodes agiles.

Il y a toujours des choses à améliorer. Par exemple une idée d'amélioration sortant du cadre traditionnel, ce serait une demi-journée par semaine pour travailler sur des projets personnels.

#### 10.4.4 Se concentrer sur une amélioration

La rétrospective de *sprint* revient régulièrement : à chaque fin de *sprint*. Il ne sert à rien d'être trop ambitieux en cherchant à adapter la façon de travailler sur des nombreuses pratiques.

Vouloir faire un trop grand nombre d'actions conduit au risque qu'aucune qui n'aboutisse vraiment. Pour des *sprints* courts, d'une semaine ou deux, il vaut mieux n'essayer d'améliorer qu'une seule chose à la fois.

On peut se dire que les améliorations qui ne sont pas choisies lors d'une rétrospective ont de bonnes chances de l'être dans une prochaine.

#### 10.4.5 Mener des rétrospectives plus poussées en fin de release

La rétrospective de *sprint* revient sur le proche passé du *sprint* qui se finit. À chaque fin de *release*, il est souhaitable de mener une rétrospective qui porte sur ce qui s'est passé pendant les quelques mois de la *release*.

À cette occasion, il vaut mieux prévoir une réunion plus longue que d'habitude et d'utiliser une technique différente.

#### 10.4.6 Utiliser un facilitateur externe

Quand le ScrumMaster n'y arrive pas, quand il y a des blocages qui ne permettent pas à l'équipe de s'exprimer, l'intervention d'un consultant extérieur est souhaitable.

## En résumé

La rétrospective est la pratique Scrum pour améliorer le processus. Placée à la fin d'un *sprint*, la réunion implique toute l'équipe dans un *brainstorming* en vue d'identifier des façons de mieux travailler.

Par rapport aux approches habituelles d'amélioration de processus souvent imposées d'en haut, elle donne la parole à l'équipe pour qu'elle s'approprie la conduite de son processus.

# 11

## La signification de fini

Est-ce que tu as fini ton travail ? Quand aurez-vous fini de coder ?

Ces questions reviennent sans arrêt dans le développement d'un produit. Avec Scrum aussi : il y a même une pratique dédiée à cela. Fini ? Fini fini, c'est le credo de Scrum ! Et quand c'est fini, ça ne recommence pas !

À la fin d'un *sprint* l'équipe a réalisé un produit partiel contenant de nouvelles *stories* et le produit est potentiellement livrable. À la fin de la *release*, le produit est vraiment livré.

Scrum pousse à finir une *story* à la fin d'un *sprint*. Mais que signifie fini ? La réponse varie selon les organisations et les équipes. C'est à chaque équipe de le définir et de l'appliquer.

Le constat fait avec les équipes qui démarrent avec Scrum est que cette pratique est une des plus difficiles à réussir : souvent on pense que c'est fini, mais c'est seulement presque fini.

C'est l'objet de ce chapitre de montrer comment procéder pour que fini ait la même signification pour toute l'équipe.

### 11.1 FINI, UNE PRATIQUE À PART ENTIÈRE

La mécanique de Scrum repose sur la réalisation d'une *story* du *backlog* en un *sprint*. À la fin du *sprint*, la *story* sélectionnée pour ce *sprint* est finie et, comme toutes les autres *stories* faites dans le *sprint*, elle s'ajoute aux incrémentations précédentes. L'ensemble doit fonctionner, ce qui implique qu'il passe avec succès quelques contrôles. C'est la liste de ces contrôles qui constitue la signification de fini.

Toute l'équipe doit définir ce que signifie fini, puis l'assumer. Il est entendu que fini englobe du test, mais cette définition n'est pas suffisante :

- d'une part, il existe de nombreux types de test différents ;
- d'autre part, il ne suffit pas toujours de passer des tests avec succès pour s'assurer qu'une story est finie.

Le travail à faire par l'équipe en rapport avec la signification de fini constitue une pratique Scrum à part entière.



**Figure 11.1** — J'ai fini de ranger ma chambre !  
La signification de fini n'est pas la même pour tout le monde.

### 11.1.1 Impact du mal fini

Ne pas bien mettre en œuvre cette pratique a des impacts néfastes sur les *sprints* à venir. Les deux impacts les plus représentatifs sont la découverte de bugs, qu'il faudra corriger et l'accumulation d'une dette technique, qu'il faudra rembourser en payant les intérêts.

#### Bug sur une story

Si la définition de fini n'est pas suffisamment complète ou mal appliquée, une story qu'on avait cru finie à la fin d'un *sprint* peut finalement receler un ou plusieurs bugs qui seront découverts dans les *sprints* suivants. En particulier des lacunes au niveau des tests permettant d'accepter la story expliquent la découverte ultérieure de bugs.

La pratique « signification de fini » a pour objectif d'éviter de laisser des bugs sur les stories réalisées dans un *sprint*.

### Dette technique

Ward Cunningham est l'inventeur du terme *technical debt*<sup>1</sup>. Il utilise l'analogie avec la dette financière pour mettre en évidence que, comme les intérêts s'accumulent progressivement lorsqu'on contracte une dette, le coût de développement augmente si le logiciel comporte des imperfections techniques. Chaque minute supplémentaire passée sur du code de mauvaise qualité, cela correspond à l'intérêt de la dette.

Les bugs sont visibles des utilisateurs, alors que la dette technique ne l'est pas, seuls les développeurs la perçoivent. Comme son nom l'indique, elle est technique et cela peut avoir de multiples formes : architecture bancale, code mal écrit, standard de codage mal suivi, absence de documentation ou manque d'automatisation des tests.

La pratique « signification de fini » a pour objectif de détecter la présence d'une dette technique et d'en limiter les effets.

### Report à la fin de la release

Tout ce qui n'est pas fini pendant les *sprints*, et qui doit cependant être fait pour que les utilisateurs puissent utiliser le produit, constitue du travail qui devra être fait dans la période de temps comprise entre le dernier *sprint* et la livraison de la *release*.

On comprend bien que la signification de fini a un impact sur ce qu'il y a à faire pendant cette période : plus il y aura de choses finies pendant les *sprints* moins il en restera à faire. L'idéal est de tout faire pendant les *sprints* et de raccourcir au maximum cette période. Cela peut nécessiter d'ajuster la signification de fini au fur et à mesure du déroulement des *sprints*.

### 11.1.2 Intérêt d'avoir une signification de fini partagée

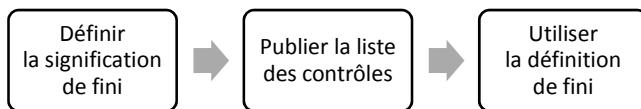
- **Motivation de l'équipe** – Comme c'est l'équipe qui décide elle-même de la signification de fini, cela contribue à la motiver, bien plus que si cela venait du management. L'appropriation collective pousse à une application de la pratique.
- **Moins d'ambiguïté** – En l'absence de définition de fini, c'est normalement le Product Owner qui décide, à la fin du *sprint*, de ce qui est vraiment fini. Il existe des Product Owners qui, manquant d'implication, ne sont pas assez disponibles pour passer des tests d'acceptation et (souvent à cause de leur expérience d'utilisateurs face à des informaticiens) sont méfiants et ont du mal à considérer que quelque chose est vraiment fini. Avoir une définition élaborée par toute l'équipe évite d'avoir une trop forte dépendance de la décision du Product Owner.
- **Plus de qualité, plus de transparence** – La qualité requise est connue par tous, grâce à la liste explicite des contrôles définis dans la signification de fini. La transparence fait que le produit partiel est conforme à cette définition.

---

1. Dans un article publié en 1992 : <http://c2.com/doc/oops92.html>

## 11.2 ÉTAPES

La pratique couvre non seulement la définition de ce que signifie fini mais aussi son application pendant le développement.

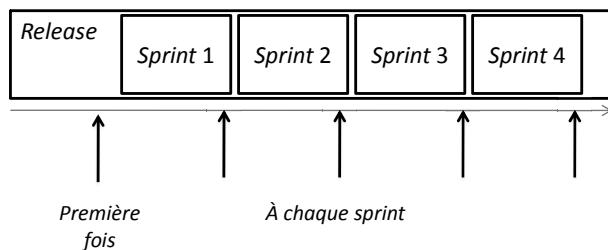


**Figure 11.2** — Les étapes de mise en œuvre de la pratique

### 11.2.1 Définir la signification de fini

Toute l'équipe est impliquée : c'est normal, c'est elle qui appliquera cette définition. La participation du Product Owner au travail collectif est indispensable et celle des managers est fortement souhaitée.

La signification de fini est définie pour la première fois avant de commencer le premier *sprint*. Elle est mise à jour, si nécessaire, à chaque *sprint*, le plus commode étant de le faire à l'occasion de la rétrospective.

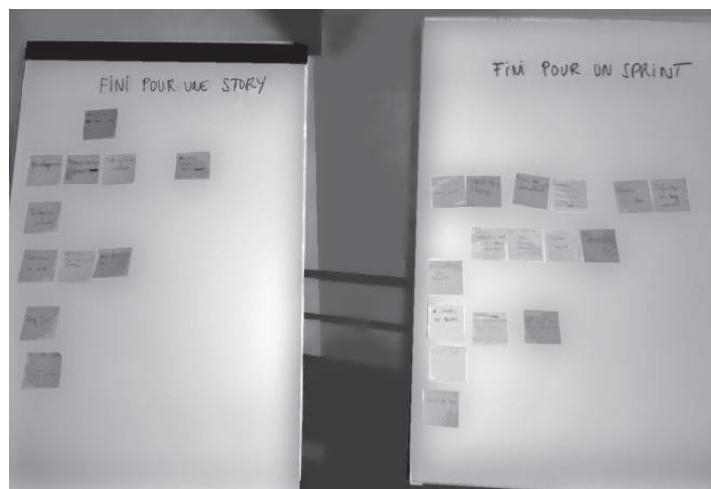


**Figure 11.3** — Moments où l'équipe travaille sur la définition de la signification de fini

Le premier travail sur la définition de fini n'est pas une réunion balisée dans Scrum, la façon de procéder non plus. Le plus important, c'est que ce travail se fasse en équipe : un *brainstorming* est la formule la plus adaptée. Son but est de collecter les idées de chacun pour construire une signification de fini complète et partagée.

La réunion est animée par le ScrumMaster (ou par un consultant externe) qui mène la consolidation à partir des idées collectées. En particulier, il doit s'assurer que la voix des personnes les plus impliquées dans les tests soit entendue : les tests sont au cœur de la finition d'une story.

La durée de la réunion varie selon la technique utilisée. D'après mon expérience, la durée moyenne d'une séance de travail sur la signification de fini est d'une heure et demie.



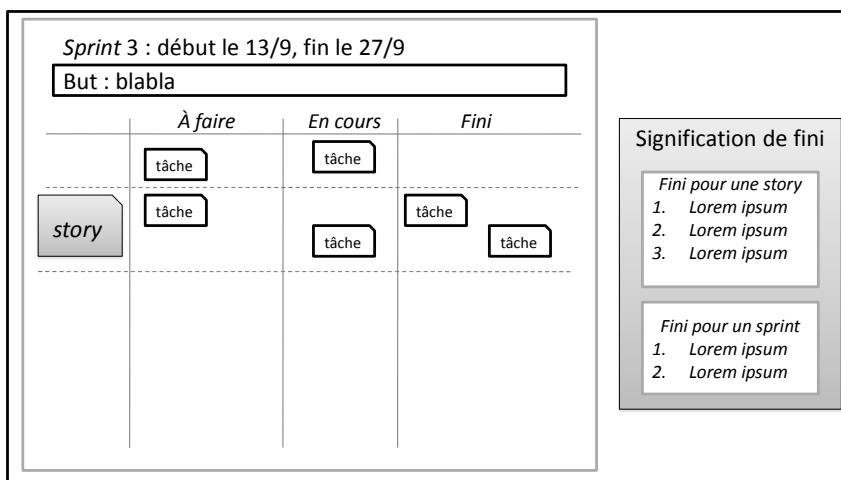
**Figure 11.4** — Collecte des Post-its préparant la consolidation

L'objectif de la première réunion est d'obtenir une liste de contrôles qui s'appliquent aux *user stories*. Cette liste sera remise à jour à chaque sprint.

### 11.2.2 Publier la liste des contrôles

La liste doit être publiée pour être visible de toute l'équipe. C'est un moyen simple de communiquer sur le processus, qui permet à tout le monde d'avoir une compréhension commune de ce que fini signifie.

Si l'équipe dispose d'un espace de travail ouvert, la publication prend la forme d'une affiche collée sur le mur à côté du tableau des tâches.



**Figure 11.5** — La signification de fini sur le tableau mural

La liste a également sa place dans les autres formes de communication utilisées par l'équipe : site, wiki...

### 11.2.3 Utiliser la définition de fini

La pratique est mise en œuvre pendant les travaux du *sprint* et au cours des réunions du cérémonial Scrum :

- **Planification de release** – La planification de *release* repose sur le principe qu'une *story* est finie à la fin d'un *sprint*, c'est pourquoi il est préférable d'avoir défini au préalable ce que cela signifie. En effet, selon la définition de fini, les estimations peuvent varier.

**Exemple :** il arrive que deux *stories* qui nécessitent autant de développement soient bien différentes sur le travail à faire en test. Une estimation faite sans avoir en tête la définition de fini précisant le type de tests à mettre en œuvre peut conduire à leur donner la même taille, à tort.

- **Planification de sprint** – Ce que signifie fini a un impact sur les tâches du *sprint*. En effet les travaux induits doivent explicitement figurer dans le plan de *sprint*.  
**Exemple :** si la définition de fini inclut la rédaction de la documentation utilisateur, il faut une tâche pour cela.

À la fin de la réunion, quand l'équipe s'engage pour le périmètre d'un *sprint*, elle doit le faire à l'aune de sa définition de fini, en étant bien consciente des implications de son engagement.

- **Pendant le sprint** – Pendant le *sprint*, des contrôles sont effectués pour s'assurer que la signification de fini est bien appliquée. En général, un contrôle est associé à une tâche et il est effectué à la fin de la tâche.

**Exemple :** si fini inclut la rédaction d'un document, la tâche pour le rédiger inclut le contrôle du document produit.

- **Revue de sprint** – Lors de la revue de *sprint*, l'équipe ne montre que ce qu'elle a fini. La décision de considérer un élément du *backlog* comme fini dépend – évidemment – de la définition élaborée par l'équipe.

**Exemple :** si elle inclut la disponibilité d'une version en plusieurs langues et qu'à la fin du *sprint* il n'existe que le français, l'élément ne devrait pas être considéré comme fini.

- **Rétrospective** – La rétrospective est le moment où l'équipe s'interroge sur sa façon de travailler. Comme la définition de fini a un impact sur de nombreuses pratiques, elle constitue la réunion idéale pour la revoir et si nécessaire l'améliorer.

## 11.3 CONTENU DE FINI

Dans les processus classiques, les livrables attendus pour un jalon sont organisés selon les disciplines du développement. Pour du logiciel, à un jalon est associée la liste les

documents de conception et de gestion de projet, par exemple, qui doivent être livrés pour la revue.

Avec Scrum, la signification de fini définit ce qui est attendu pour un *sprint*. Mais comme un *sprint* porte sur des *stories*, le découpage n'est pas fait sur des disciplines, il est d'abord relatif à ces *stories*.

Pour qu'un *sprint* soit bien fini, il convient que les *stories* soient finies, c'est le premier niveau de la définition de fini. Le deuxième niveau porte sur ce que doit inclure le *sprint* en plus des *stories* qui le composent. Un troisième niveau décrit ce qu'on entend par fini à la fin d'une *release*.

Fini pour une <i>story</i>	Fini pour un <i>sprint</i>	Fini pour une <i>release</i>
<ul style="list-style-type: none"> <li>• Les tests à passer</li> <li>• Les autres contrôles relatifs à une <i>story</i></li> </ul>	<ul style="list-style-type: none"> <li>• Le produit partiel livré dans un environnement de test</li> <li>• Les autres contrôles à la fin du <i>sprint</i></li> </ul>	<ul style="list-style-type: none"> <li>• Ce qui reste à faire pour la mise en production</li> </ul>

**Figure 11.6** — Les trois niveaux de la signification de fini

### 11.3.1 Fini pour une story

Avec une approche simpliste, on pourrait considérer que c'est simplement : démontré avec succès lors de la revue de *sprint*. Mais ce n'est pas suffisant : les tests doivent être passés, par exemple.

Fini pour une *story* est variable, mais on peut identifier des travaux à faire absolument et d'autres qui sont dépendants du contexte.

#### Obligatoire

Le minimum, pour une *user story*, est que ses tests d'acceptation<sup>1</sup> soient passés avec succès. Sans le passage avec succès de ces tests, une *story* ne peut pas être considérée comme finie.

#### Dépendant du contexte

Fini peut aller beaucoup plus loin. Chez un de mes clients, le *brainstorming* avec l'équipe a abouti à cette définition de fini pour une *user story* :

---

1. Les tests d'acceptation sont présentés dans le chapitre 14 *De la story aux tests*.

- codée en suivant le standard de codage,
- les tests unitaires passés,
- les tests d'acceptation passés avec succès dans un environnement de test,
- la version disponible en français et en anglais,
- la documentation marketing rédigée,
- la documentation utilisateur rédigée.

#### Contrôle

Les contrôles sont effectués pendant le *sprint* avec l'objectif qu'ils soient tous passés avec succès en fin de *sprint*. Si ce n'est pas le cas, la *story* n'est pas finie et sauf dérogation décidée par l'équipe, elle devra être reprise dans le prochain *sprint*.

### 11.3.2 Fini pour un sprint

À la fin du sprint, l'essentiel est que les *stories* soient finies. Mais, en plus du contrôle sur les *stories*, la signification de fini pour le *sprint* porte sur des travaux qui sont transverses, dans la mesure où ils ne sont pas spécifiques d'une *story*. Ces travaux reviennent à chaque *sprint*, c'est pourquoi on peut aussi les qualifier de récurrents.

#### Obligatoire

Le minimum est qu'un *build* contenant le produit partiel soit placé dans un environnement de test.

#### Dépendant du contexte

Selon le contexte, la signification de fini pour un *sprint* peut varier énormément. Pour certains projets, où la criticité est forte, les besoins en documentation seront importants, pour d'autres, comme des projets de recherche, ils seront très faibles.

Pour un développement de logiciel, les rubriques de la signification de fini concourent à la qualité. On peut y retrouver des exigences sur le code :

- un pourcentage de couverture de tests,
- un taux de commentaires dans le code,
- des classes limitées en complexité...

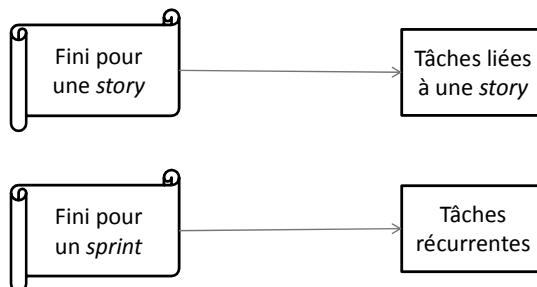
Des outils qui mesurent la qualité du code<sup>1</sup> permettent de s'assurer que ces exigences sont respectées.

On peut y retrouver des exigences sur la conception. C'est aussi dans cette rubrique que viennent des exigences, dites non fonctionnelles, en rapport avec le comportement du produit considéré comme une boîte noire.

---

1. Comme Sonar, outil Open Source : <http://sonar.codehaus.org/>

**Exemple :** fini pour un *sprint* peut inclure le passage de tests visant à vérifier que les performances spécifiées sont bien tenues, par exemple des tests de charge.



**Figure 11.7 – Impact sur les tâches du *sprint***

Si la signification de fini pour une *story* permet d'identifier des tâches liées aux stories du *sprint*, celle de fini pour un *sprint* facilite l'identification des tâches récurrentes.

### Contrôle

Même si les exigences de fini pour un *sprint* n'ont pas été respectées, le *sprint* se termine à la date prévue. Mais il faut en tenir compte pour les *sprints* suivants : c'est un point à aborder lors de la rétrospective et l'équipe devra trouver des solutions concrètes.

Ne pas le faire contribuerait à alimenter la dette technique, qu'il faudra rembourser un jour, avec les intérêts qui augmentent avec le temps.

### 11.3.3 Fini pour une release

Cela dépend de ce qu'on fait du produit à la fin de la période de temps que constitue la *release*. Est-ce qu'il est mis en production ? Ou est-ce qu'il va être mis dans un environnement d'intégration ou de pré-production ? Est-ce que des activités de marketing sont à réaliser ?

Quelle que soit la réponse, il est important de définir l'état du produit pour le jalon que constitue la fin de *release*. Cela permet de savoir, par différence avec fini à la fin d'un *sprint*, ce qui reste à faire après le dernier *sprint*.

## 11.4 GUIDES POUR UNE BONNE PRATIQUE DE FINI

### 11.4.1 Faire des tranches verticales

Pour que fini pour une *story* ait un sens, il convient de s'assurer que les *stories* correspondent à des tranches verticales, en référence aux architectures de logiciel en couches.

Une tranche verticale signifie que la *story* part de l'utilisateur, traverse toutes les couches pour remonter jusqu'à lui.

### Histoire de couches

Un développement commence qui reprend une application existante avec une nouvelle technologie. La première réaction de l'équipe, voyant l'importance des changements à effectuer, est de se consacrer, pour le premier *sprint* de trois semaines, à toute la couche IHM dans sa globalité, c'est-à-dire à une tranche horizontale dans une présentation en couches.

Cependant l'approche agile pousse à développer les *user stories* de bout en bout incluant leur IHM, ce qui constitue des tranches verticales. À la fin d'un *sprint*, on doit disposer d'une version partielle mais fonctionnelle du logiciel. Ne faire que de la conception d'IHM au cours du *sprint* ne rentre pas dans ce schéma.

#### 11.4.2 Adapter à partir d'une définition générique de fini

Toutes les *stories* ne sont pas de même nature : il y en a orientées utilisateur, d'autres techniques. La signification de fini n'est pas la même pour ces deux types. Même pour les *user stories*, il peut y avoir des variations dans l'état dans lequel on veut la voir à la fin d'un *sprint*, en fonction du *feedback* attendu. C'est pourquoi, à partir d'une définition générique, il peut être nécessaire de la spécialiser selon les types de *stories*.

Certains proposent même une spécialisation pour chaque *story* : la signification de fini devient alors une matrice.

**Exemple :** l'équipe a mis en place une matrice, avec en lignes une liste de contrôles possibles sur les *stories* et en colonnes les *stories* sélectionnées pour le *sprint* courant. Pour chaque *story*, une croix dans une ligne signifie que le contrôle correspondant est à faire. La matrice est définie par l'équipe en début de *sprint*.

Cela va plus loin que la pratique habituelle qui consiste à avoir une seule définition de fini, valable pour toutes les *stories*. C'est ce que faisait l'équipe avant et c'est parce que cela ne marchait pas bien que l'équipe a décidé de passer à la matrice.

Dans l'environnement, les choses à faire pour finir une *story* sont nombreuses. La matrice comporte une vingtaine de lignes pour des contrôles comme : scripts JMeter écrits, installation sur le serveur de test, *release note* rédigée, tests de charge, Javadoc écrite, modèle UML mis à jour, pour en citer quelques-uns, en plus des classiques tests unitaires et tests fonctionnels (tests d'acceptation associés aux *stories*) passés avec succès.

L'élaboration de cette matrice lors de la réunion de planification du *sprint* facilite l'identification des tâches associées aux *stories*. On pourrait imaginer un outil qui facilite la saisie des contrôles à appliquer pour chaque *story*, en propose des tâches candidates et permette de cocher les contrôles retenus au fur et à mesure qu'ils sont vérifiés pour rendre visible ce qui reste à contrôler sur les *stories* du *sprint*. Une autre solution est d'identifier les contrôles spécifiques faits sur une *story* comme des conditions de satisfaction de cette *story*.

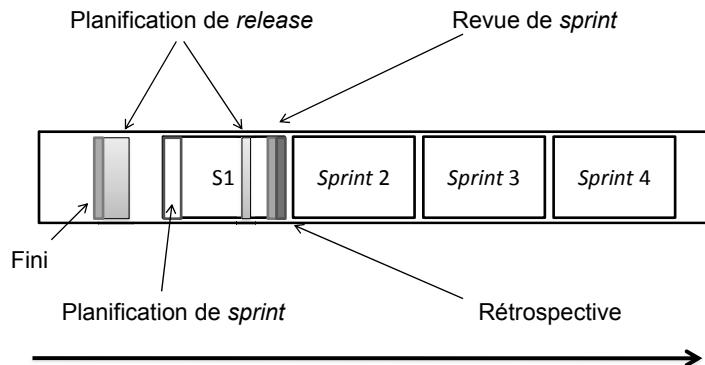
### 11.4.3 Faire évoluer la signification de fini

La signification de fini est susceptible d'évoluer pour deux raisons :

- Des aspects avaient été oubliés et ont été mis en évidence lors de la revue.
- À travers la rétrospective, les pratiques d'une équipe évoluent régulièrement, à chaque *sprint*. La signification de fini doit évoluer également pour prendre en compte ces améliorations.

### 11.4.4 Appliquer la pratique avec beaucoup de volonté

L'équipe doit s'approprier cette pratique : pour commencer, c'est elle qui définit ce que signifie fini. Mais c'est surtout lors des travaux qui se déroulent pendant le *sprint* que son application prend toute son importance. Il ne suffit pas de définir fini, il faut le mettre en œuvre, notamment pendant les réunions du cérémonial.



**Figure 11.8** – La signification de fini et les moments où l'appliquer, montrés sur le *sprint* 1

Comme l'équipe a souvent le nez dans le guidon pendant le *sprint*, le ScrumMaster joue un rôle essentiel dans l'application de la pratique « fini ». C'est à lui de rappeler régulièrement la définition élaborée en commun et de motiver l'équipe pour qu'elle soit respectée.

## En résumé

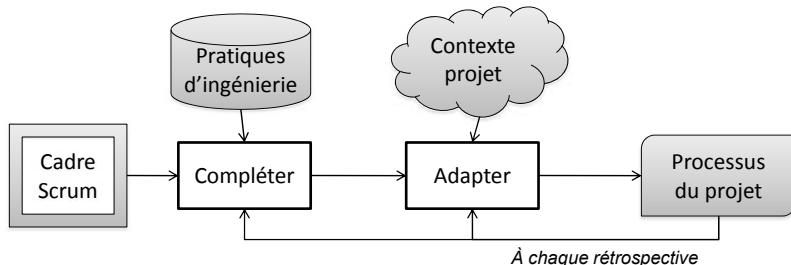
La signification de fini est la pratique qui définit les jalons du processus. Avec Scrum, elle décrit l'état attendu du produit partiel à la fin de chaque *sprint*.

Une bonne mise en œuvre de cette pratique permet de motiver l'équipe, contribue à diminuer les mauvaises surprises et à améliorer la qualité du produit.

# 12

## Adapter Scrum au contexte

Scrum se présente comme un cadre pour développer des produits, ce n'est pas vraiment une méthode ni un processus complet. Quand on utilise Scrum sur un projet il faut ajouter à ce cadre des pratiques complémentaires, variables selon le domaine, et adapter le tout au contexte. Cette adaptation, faite une première fois au démarrage avec Scrum, est réitérée à chaque *sprint*, lors de chaque rétrospective.



**Figure 12.1** — Du cadre Scrum au processus d'un projet

Comme le rappelle Ken Schwaber dans *Flaccid Scrum*<sup>1</sup>, Scrum ne prétend pas tout prendre en compte et il est naturel de le renforcer avec des pratiques issues d'autres domaines, comme la définition de produit, l'ingénierie des exigences, la gestion de projet et l'ingénierie du logiciel.

1. <http://www.scrumalliance.org/resources/745>

**Exemple :** nous avons vu dans les chapitres précédents que Scrum ne préconisait pas de façon de faire spécifique pour identifier les éléments du *backlog*, ni pour faire des estimations. Des pratiques complémentaires, habituellement associées à Scrum, ont été évoquées comme celles des *user stories* pour peupler le *backlog* et le *planning poker* pour estimer sa taille.

C'est l'objet de ce chapitre que de présenter comment Scrum doit être appliqué en tenant compte des contraintes imposées par le contexte.

## 12.1 PRATIQUES AGILES

La notion de **pratique** prend toute son importance avec les méthodes agiles. À côté des valeurs et des principes qui sont universels, les pratiques sont le reflet de la mise en œuvre sur le terrain.

### Définition

Une **pratique** est une approche concrète et éprouvée qui permet de résoudre un ou plusieurs problèmes courants ou d'améliorer la façon de travailler dans un développement. Exemple : la revue de *sprint*.

Une pratique peut être adoptée ou pas sur un projet. Si elle est adoptée, sa mise en œuvre peut varier selon un grand nombre de paramètres dépendant du contexte.

Les valeurs et les principes sont du niveau de la culture et ne changent pas d'un projet à l'autre, tandis que les pratiques sont leur application dans une situation particulière.

### 12.1.1 Pratiques Scrum

Dans le cadre offert par Scrum, on peut identifier 14 pratiques, qui correspondent *grossso modo* aux chapitres du livre :

- Les rôles :
  - Product Owner
  - ScrumMaster
  - Équipe auto-organisée
- Le cérémonial avec les *timeboxes* :
  - *Sprints* et *release*
  - Planification de *release*
  - Planification de *sprint*
  - Scrum quotidien
  - Revue de *sprint*
  - Rétrospective
  - Signification de fini

- Les artefacts :
  - *Backlog* de produit
  - Plan de *sprint*
  - *Burndown chart* de *sprint*
  - *Burndown chart* de *release*

### 12.1.2 Pratiques complémentaires

Il existe de nombreuses pratiques utilisées dans le cadre de projets se réclamant de l'agilité. Leur nombre, la façon de les nommer et leur classification varient selon les auteurs. Il n'y a pas de liste officielle et certains ont tenté de faire une synthèse des pratiques. C'est le cas de Jurgen Appelo qui présente sur son blog<sup>1</sup> « *the big list of Agile practices* », élaborée à partir de huit sources différentes.

La façon d'organiser ces pratiques est très variable et sujette à discussion. La classification peut s'opérer selon différentes caractéristiques :

- la méthode agile d'origine de la pratique (Scrum, XP, Lean...),
- le fait qu'elle soit obligatoire ou optionnelle,

La nature optionnelle d'une pratique est un sujet à polémique qui divise les experts. À propos des pratiques Scrum présentées, mon opinion est que toutes sont obligatoires, sauf le *burndown chart*, et que c'est la façon de les mettre en œuvre qui diffère.

- les activités ou disciplines du développement (codage, gestion de projet, test...).

Dans les chapitres suivants nous allons présenter quelques pratiques qui viennent régulièrement en complément à Scrum.

#### *Pour l'ingénierie des exigences et la définition de produit*

Quand on utilise Scrum pour développer un nouveau produit, il est nécessaire d'y adjoindre des pratiques en amont.

Les pratiques de définition de produit (*Product Management*) et d'ingénierie des exigences (*Requirements Engineering*) sont importantes, en particulier au début d'un développement, et facilitent la constitution du *backlog*.

Dans le chapitre « De la vision aux stories », nous aborderons les pratiques suivantes :

- Construire une vision partagée.
- Élaborer une liste de *features*.
- Identifier les rôles d'utilisateur.
- Décomposer en *user stories*.

1. <http://www.noop.nl/2009/04/the-big-list-of-agile-practices.html>

### *Pour les tests d'acceptation*

La signification de fini met en évidence le besoin de passer des tests pendant un *sprint* pour considérer une *story* comme finie. Dans le chapitre « *De la story aux tests* », nous verrons comment mettre en œuvre la pratique de pilotage par les tests d'acceptation.

### *Pour la gestion de projet*

Scrum est une approche très orientée gestion de projet. La visibilité, l'inspection et l'adaptation constituent une façon nouvelle de contrôler et suivre les projets. Cependant, d'autres domaines de la gestion de projet ne sont pas abordés par Scrum.

Quelques pratiques souvent associées à Scrum seront présentées dans le chapitre 15 *Estimations, mesures et indicateurs* :

- Estimer la taille des *stories* en points.
- Estimer l'utilité des *stories*.
- Collecter des mesures.
- Produire et utiliser des indicateurs agiles.

### *Ingénierie du logiciel*

La plupart du temps Scrum est utilisé pour développer du logiciel. Cependant Scrum ne prescrit pas de technique d'ingénierie particulière, laissant le choix à l'équipe.

Nous verrons dans le chapitre 16 *Scrum et l'ingénierie du logiciel* que la plupart sont nécessaires et induites par la signification de fini :

- Intégration continue.
- Pilotage par les tests.
- Remaniement du code.
- Programmation en binôme.
- Architecture évolutive.

## **12.2 RISQUES DANS LA MISE EN ŒUVRE DE SCRUM**

Une équipe qui passe à Scrum doit faire face à plusieurs risques.

### *Risque d'oublier les pratiques d'ingénierie*

Une équipe qui se contente du cadre Scrum et qui ne met pas en place de pratiques d'ingénierie pour respecter sa signification de fini (ou qui n'applique pas la pratique « signification de fini ») va se heurter rapidement à des difficultés. Scrum permet de les révéler mais c'est à l'équipe de faire les efforts pour trouver des solutions.

### Risque de perdre le contexte sans adaptation

En appliquant une pratique strictement, comme on l'a compris en lisant des livres ou à partir d'une expérience qui a été racontée, on risque de se couper de l'environnement actuel de l'organisation. Ce n'est pas parce qu'une pratique a fonctionné ailleurs qu'elle marchera à l'identique dans un contexte différent.

Voici un exemple qu'on m'a raconté, anecdotique mais caricatural de la copie d'une pratique sans adaptation au contexte. Une équipe est distribuée sur plusieurs sites en France et aux États-Unis et les réunions se tiennent donc en utilisant un système de vidéoconférence. Un participant au premier scrum quotidien, probablement marqué par le côté *standup* du scrum, demande s'il faut rester debout...

### Risque d'adapter sans discernement

Certaines équipes pensent appliquer Scrum, mais n'utilisent que des bribes de l'ensemble des pratiques. Partant du principe que Scrum s'adapte, elles n'en prennent que ce qui les arrange ou uniquement ce qu'elles ont compris d'un apprentissage trop rapide.

Je rencontre souvent des équipes qui ont démarré Scrum depuis quelques mois, sans formation ni assistance, et qui n'ont conservé que le scrum quotidien et des *sprints* d'un mois. Elles sont un peu perdues et m'appellent pour les aider à s'y retrouver dans les pratiques à mettre en œuvre sur leur projet.

Ken Schwaber définit de la façon suivante la possibilité d'adaptation de Scrum : « *L'équipe suit strictement les règles Scrum, jusqu'à ce que le ScrumMaster considère que l'équipe est suffisamment aguerrie. Dans ce cas il peut proposer des adaptations à l'équipe.* »

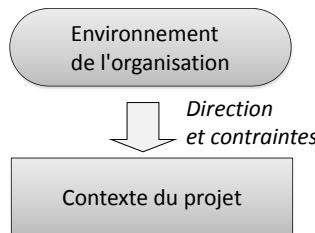
Il faut bien dire que les règles dont parle Ken Schwaber ne sont pas toutes des règles qu'on peut vérifier mais plutôt des recommandations, ce qui laisse une grande part d'interprétation. Ce qui est sûr, c'est qu'une adaptation sans discernement est dangereuse : les équipes qui n'ont pas le recul suffisant risquent d'échouer.

La plupart des pratiques sont utiles pour la plupart des projets, mais elles ne s'appliquent pas partout de la même façon et leur application évolue dans le temps. Pour en savoir plus sur leur application, il faut commencer par définir le contexte des organisations et des projets.

## 12.3 DÉFINIR LE CONTEXTE

Un projet de développement de logiciel s'inscrit dans le contexte d'une organisation. Celle-ci possède un environnement qui influe sur les caractéristiques des projets qui y sont lancés et développés. Les attributs relatifs à la situation d'un projet permettent de définir la façon dont les pratiques agiles peuvent y être appliquées. C'est l'**agilité**

**en situation<sup>1</sup>**, titre de la présentation de Philippe Kruchten et Claude Aubry, lors de l'Agile Tour 2008, sur laquelle s'appuie la notion de contexte.



**Figure 12.2** – L'influence de l'organisation sur les projets

### 12.3.1 Influence de l'organisation

Une organisation possède une culture, un savoir-faire, bref des conditions qui ont une influence sur les projets qui vont appliquer Scrum et l'agilité. Il est bien évident que l'application sera plus difficile dans une entreprise qui a des valeurs et des principes éloignés de ceux définis par le *Manifeste agile*.

On peut relever quelques attributs qui permettent de situer une organisation par rapport à l'agilité :

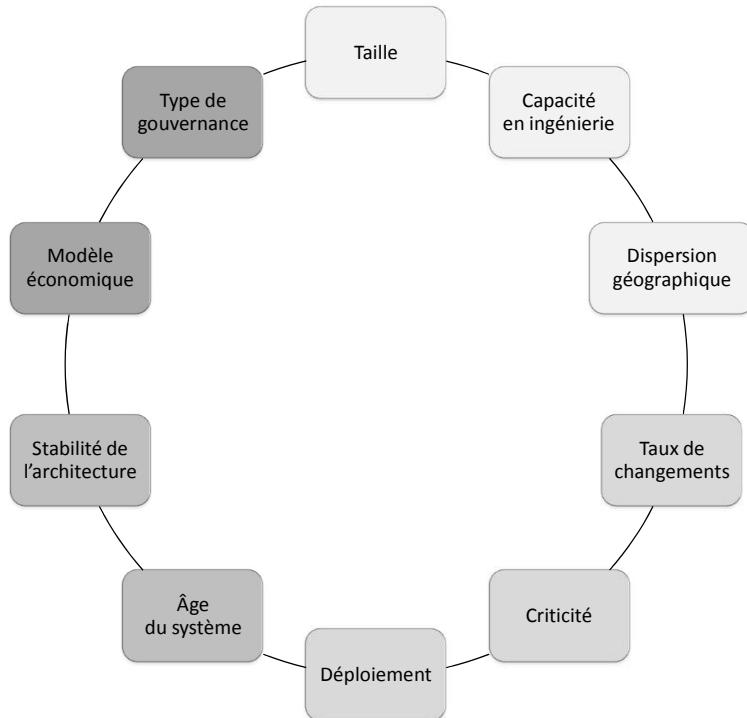
- **Niveau d'innovation** : les entreprises qui développent des produits innovants seront dans un contexte plus favorable que celles qui font des applications plus classiques.
- **Culture** : si la façon de partager un modèle mental, de communiquer et de collaborer est basée sur la confiance entre les gens, cela facilitera une bonne mise en œuvre des pratiques agiles.
- **Domaine métier** : il existe de nombreux domaines bien différents qui ont une influence sur les caractéristiques des projets. Par exemple, dans le domaine de l'aéronautique embarquée, la criticité des applications fera que la mise en œuvre de Scrum demandera plus d'efforts que dans le cas d'une application web.
- **Niveau de maturité** : l'expérience en ingénierie du logiciel est très différente selon les organisations. Une bonne compétence des personnes dans les pratiques d'ingénierie facilite la transition à Scrum.

Les projets qui sont menés dans une organisation sont influencés par sa situation, mais pas forcément tous de la même façon : les entreprises ont de plus en plus une grande hétérogénéité dans leurs projets, c'est pourquoi il convient avant tout de définir le contexte du projet qui va utiliser Scrum.

1. <http://www.sigmat.fr/dotclear/index.php?post/2008/10/17/L-Agilité-en-situation-la-présentation>

### 12.3.2 Contexte du projet

Les attributs de contexte permettent de caractériser un projet, la figure 12.3 présente un exemple de dix attributs significatifs.



**Figure 12.3** – Les attributs définissant le contexte d'un projet

- **Attributs relatifs à l'équipe**
  - **Taille** : le nombre de personnes dans l'équipe développant le produit. La taille de l'équipe est représentative de la taille du projet, comme le sont aussi le budget ou la taille du code.
  - **Capacité en ingénierie** : le niveau des personnes de l'équipe dans les pratiques d'ingénierie de logiciel, qui est une combinaison de la formation et de l'expérience.
  - **Dispersion géographique** : le nombre de bureaux différents accueillant les personnes de l'équipe.
- **Attributs relatifs à la nature du produit** – Ces attributs dépendent essentiellement du domaine métier dans lequel se situe le produit développé.
  - **Taux de changements** : la fréquence des changements sur le produit, à propos de sa technologie ou des besoins des utilisateurs.
  - **Criticité** : l'enjeu humain ou économique en cas de défaillance du logiciel.
  - **Déploiement** : le nombre d'instances du produit final.

- Attributs relatifs à l'état du logiciel
  - Âge du système : la quantité de code déjà existant qu'incorpore le logiciel à développer.
  - Stabilité de l'architecture : le degré selon lequel on peut ajouter des *user stories* au logiciel existant sans revoir l'architecture.
- Attributs liés à l'organisation de développement
  - Modèle économique : projet open source, projet interne, projet sous-traité au forfait, logiciel commercial...
  - Type de gouvernance : les contraintes imposées par l'organisation sur le cycle de vie et le « *reporting* ».

## 12.4 IMPACT DU CONTEXTE SUR LES PRATIQUES

### 12.4.1 Impact par attribut

#### Taux de changement

Un taux de changement élevé est souvent le facteur qui va déclencher le passage à l'agilité. Dans l'esprit des personnes, il y a souvent l'idée : « *on ne sait pas vraiment ce qu'on veut et on est sûrs que ça va changer, alors comme l'approche traditionnelle ne marche pas, essayons une méthode agile* ».

Au-delà de cet effet déclencheur, un taux de changement élevé aura un impact sur quelques pratiques, sans demander d'efforts supplémentaires pour leur mise en œuvre :

- Les *sprints* seront plus courts pour incorporer rapidement les changements.
- L'implication du Product Owner doit être très forte pour définir les priorités dans le *backlog* de produit.
- La planification de *release* doit être révisée régulièrement pour refléter des changements.
- Le *burndown chart* de *release* ne sera pas suffisant et devra être complété par un indicateur (comme le *burnup*<sup>1</sup>) faisant apparaître l'évolution de périmètre.

Au contraire, si les besoins sont stables et l'architecture maîtrisée, les équipes seront peut-être moins enclines à passer à l'agilité ou à relaxer des règles. Mais un faible taux de changement n'empêche pas d'utiliser les pratiques.

#### Criticité

Lorsque le produit développé possède une criticité élevée, la conformité à des normes doit être prouvée, par l'utilisation de méthodes formelles, par des tests intensifs ou par des audits externes.

---

1. Voir le chapitre 15 *Estimations, mesures et indicateurs agiles*

Une criticité élevée a un impact sur :

- Les pratiques de test qui devront être poussées (pour être conforme aux normes du domaine).
- Le backlog qui demandera à être complété avec de la documentation, et la mise en place d'une traçabilité explicite sur les exigences.
- Le rythme des sprints, qui peut être perturbé par les audits qui ont un rythme différent.
- Le rôle de Product Owner : il risque d'être difficile pour une seule personne de bien connaître à la fois les normes à respecter et les fonctions attendues et de définir les priorités.

### Taille de l'équipe

Si le développement du produit implique un grand nombre de personnes, cela affecte :

- La constitution des équipes, puisqu'il faudra organiser l'équipe globale en plusieurs sous-équipes, la taille d'une équipe Scrum étant limitée à une dizaine de personnes.
- La durée des sprints, qui sera plus longue pour synchroniser les travaux des différentes équipes.
- Le mode de communication qui nécessitera plus de formalisme et de documentation.
- Le rôle de Product Owner pour définir et mettre en place son interaction avec les différentes équipes Scrum.
- Le déroulement du cérémonial qui devra être adapté au fonctionnement multi-équipes, pour rendre possible la participation de personnes aux réunions de différentes équipes.

### Dispersion de l'équipe

La distribution géographique rend toutes les pratiques plus difficiles à appliquer et plus susceptibles d'échouer.

La dispersion d'une équipe affecte lourdement le déroulement du cérémonial Scrum. Toutes les pratiques sont à adapter à cette situation, en général par une utilisation massive d'outils permettant la collaboration à distance.

Cela n'interdit pas à ces équipes d'utiliser Scrum : il serait dommage de priver de nombreux projets de ses bienfaits. Dans *Succeeding with Agile*<sup>1</sup>, Mike Cohn présente, dans un chapitre dédié, les façons d'adapter les pratiques Scrum dans le cas d'équipes réparties sur plusieurs sites.

---

1. <http://mikecohnsignatureseries.com/books/succeeding-with-agile>, chapitre 18.



**Figure 12.4** — Une équipe Scrum coupée en deux aura plus de difficultés

### Capacité en ingénierie de l'équipe

Une équipe Scrum doit couvrir, avec l'ensemble de ses membres, toutes les activités nécessaires pour obtenir un produit à la fin d'un *sprint* : elle est dite multifonctionnelle. Une équipe qui ne possède pas assez de compétences techniques en ingénierie de logiciel<sup>1</sup>, cela a des impacts négatifs sur la gestion du projet et la qualité du produit.

Si la capacité de l'équipe en ingénierie n'est pas suffisante, sa mise à niveau est la priorité, par des actions de formation ou d'accompagnement des membres de l'équipe.

### Âge du système

Il est plus facile de mettre en œuvre Scrum sur un logiciel nouveau. Si le produit incorpore des parties de code existant (*legacy*), il faut vivre avec ce code qui est de plus ou moins bonne qualité :

- Les pratiques de test sont impactées : se pose la question des tests sur le code ancien (régression et automatisation).
- La découverte de bugs sur cette partie existante risque de perturber le déroulement des *sprints*.
- La gestion des priorités dans le *backlog* demande des arbitrages incessants entre les bugs, la résorption de la dette technique et les nouveautés fonctionnelles.

---

1. Les pratiques d'ingénierie du logiciel sont présentées au chapitre 16.

### Stabilité de l'architecture

Si l'architecture d'un logiciel est stable, le *backlog* comportera essentiellement des *user stories* qui pourront être développées sans modification importante de la structure du logiciel.

Si ce n'est pas le cas, il sera nécessaire d'inclure de nombreuses *stories techniques* pour stabiliser l'architecture et les pratiques suivantes peuvent être impactées :

- Les *sprints*, dont la durée devra tenir compte de la longueur des travaux techniques.
- La gestion des priorités dans le *backlog*, qui demandera de convaincre le Product Owner de l'importance des *stories techniques*, alors qu'elles n'apportent pas de valeur visible.

### Modèle économique

La façon dont le budget pour le développement est obtenu (ou les ressources dans le cas d'un projet Open Source) influe sur les pratiques. Par exemple, un développement réalisé par un contrat au forfait entre un donneur d'ordre et une SSII complique la mise en œuvre de plusieurs pratiques :

- La personne qui tient le rôle de Product Owner devrait être côté donneur d'ordre et intégrée à l'équipe, ce qui est souvent difficile.
- La planification de *release*, avec les estimations faites par l'équipe qui nécessitent d'avoir acquis la confiance entre le donneur d'ordre et le prestataire.
- Le suivi du projet du point de vue économique, qui demande à tous les partenaires de se mettre d'accord sur les nouveaux indicateurs agiles à utiliser.

### Type de déploiement

Le nombre d'instances déployées varie entre un et des millions selon les applications. Dans le cas de l'hébergement en ligne, appelé SaaS (*Software as a Service*), le déploiement est maîtrisé par l'organisation, qui peut le faire très fréquemment. Un déploiement fréquent a un impact sur :

- Les *sprints*, qui peuvent être très courts.
- La signification de fini.
- L'automatisation des tests.

À l'inverse, un déploiement du logiciel sur des dispositifs diffusés à des milliers d'exemplaires (comme dans le domaine de la téléphonie ou celui des jeux vidéo), rend les mises à jour difficiles, et très coûteuses dans certains cas, ce qui oblige à avoir des tests très poussés en interne.

### Type de gouvernance

En France, les administrations et les grandes entreprises ont une façon de gouverner les projets qui est souvent héritée de Merise dans le domaine tertiaire et du GAM T17

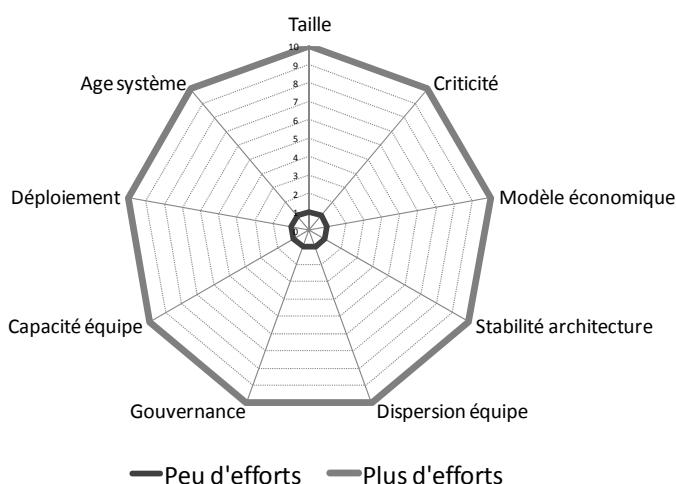
dans le domaine industriel. Bien sûr, chacune a ensuite fait ses adaptations à ces cadres, mais le résultat en matière de gouvernance ne donne pas, en général, dans la légèreté.

Ces structures se lancent maintenant dans l'agilité, mais leur type de gouvernance a un impact sur les pratiques agiles, notamment :

- L'autonomie de l'équipe n'est pas facile à obtenir dans un environnement très hiérarchique.
- La façon de faire le planning et du *reporting* avec Scrum, qu'il faut faire cohabiter avec les pratiques traditionnelles (comme les comités de pilotage), au moins un certain temps.
- Les relations du projet agile avec les autres services, en particulier celui qui s'occupe d'infrastructure, nécessitent de faire de la planification prévoyant les points de synchronisation à l'avance.

### 12.4.2 Situation d'un projet par rapport à l'agilité

L'examen des attributs du projet permet de définir les pratiques à adapter. Une représentation synthétique donne une idée de l'effort nécessaire pour l'adaptation.



**Figure 12.5** — Profil pour l'adaptation à l'agilité  
Les projets qui demandent le moins d'efforts sont au centre.

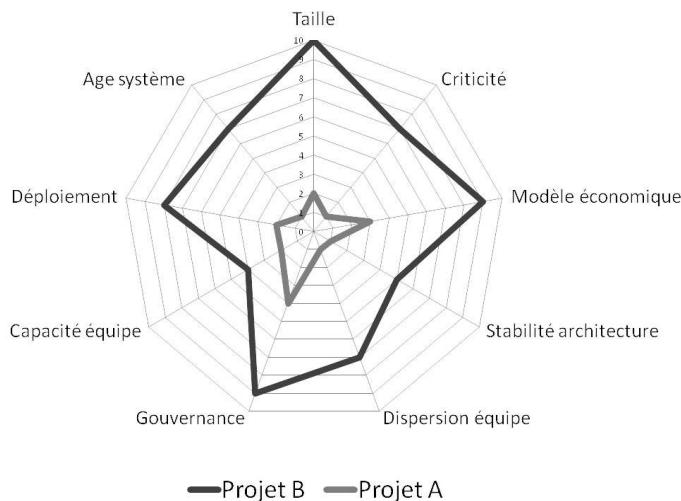
**Attention :** c'est relatif ! Un changement de processus, comme l'est un passage à l'agilité, demande toujours de nombreux efforts.

Un projet typique pour un passage à Scrum sans beaucoup d'adaptations a les attributs suivants :

- Une équipe de sept personnes, toutes dans le même bureau.
- Un logiciel nouveau avec une architecture connue.

- Un produit qui n'est pas critique.
- Un développement fait en interne avec dans l'équipe de bonnes compétences techniques.
- Une gouvernance accommodante.

De nombreux projets se situent dans ce cœur de cible de l'agilité. Mais il en existe beaucoup d'autres qui sont dans une situation demandant plus d'efforts lors de l'introduction de Scrum.



**Figure 12.6** — Le projet B demande plus d'efforts pour appliquer Scrum

Les attributs définissant le contexte évoluent dans le temps. Après quelques mois, certains devraient se rapprocher du centre, comme la gouvernance, mais d'autres peuvent s'en éloigner, comme la taille de l'équipe si le projet grandit. C'est pourquoi l'analyse du contexte devrait être mise à jour à chaque nouvelle *release*.

## En résumé

Scrum ne se vend pas en pack de 6. Sélection et adaptation des pratiques sont les deux mamelles de son application sur un projet.



# 13

## De la vision aux stories

Quand on est dans les *starting-blocks*, il est tentant de partir vite et à fond, mais il y a des risques de faire un faux départ. Les *sprints* avec Scrum, ça se prépare et nous avons vu qu'il y avait une période particulière avant de commencer le premier. C'est de cette période pour le développement d'un nouveau produit dont il est question dans ce chapitre. Les pratiques présentées sont mises en œuvre au début d'un projet, mais gardent de l'importance pendant tout le développement.

### 13.1 LE PRODUIT A UNE VIE AVANT LES SPRINTS

J'ai déjà évoqué quelques-unes des activités à mener, juste avant le premier *sprint*, avec la planification de *release*. Elles nécessitent de disposer d'un *backlog* de produit, mais comment, à partir de l'idée initiale, arriver au *backlog* n'est pas dans la visée de Scrum. Même si Ken Schwaber évoque la notion de vision, il ne la développe pas et dit clairement que Scrum n'aborde pas les pratiques de gestion de produit et aborde peu celles d'ingénierie des exigences. Donc, la voix officielle de Scrum considère que la façon de dérouler la phase amont est en dehors du cadre et suggère d'associer à Scrum des pratiques complémentaires.

Dans la suite de ce chapitre, nous allons aborder les pratiques présentées figure 13.1.



**Figure 13.1** — Les pratiques permettant la constitution du *backlog* initial

Ces pratiques visent à démarrer le premier *sprint* dans de bonnes conditions. Nous avons vu qu'il fallait un *backlog* estimé et priorisé<sup>1</sup>. Pour obtenir ce *backlog*, il est nécessaire d'avoir une bonne vision sur le produit, d'identifier les *features* et les rôles et de décomposer les *features* en *stories*.

## 13.2 CONSTRUIRE UNE BONNE VISION

Dans Scrum la notion de produit est essentielle : on parle de *backlog de produit* et de **produit partiel** à la fin d'un *sprint*, par exemple. La vision du produit futur permet de préparer son développement. Tous les produits ont besoin d'une vision : cela guide les personnes qui participent au développement vers un objectif partagé.

Une vision est l'expression d'une volonté collective de développer un excellent produit. L'absence de vision limite la capacité à s'investir dans un projet.

### 13.2.1 Techniques pour la vision

La meilleure façon d'obtenir une bonne vision est de procéder par des séances de travail en groupe. Un *brainstorming* collectif permet à l'équipe d'obtenir rapidement des parties de la vision, comme la formulation du problème et la position du produit<sup>2</sup>.

#### *Énoncé du problème*

Le vrai problème à l'origine du produit, doit être clairement identifié. On peut utiliser la technique cause-effet, connue par son résultat montré avec un diagramme en arête de poisson<sup>3</sup>. Une technique plus simple consiste à énoncer le problème dans un tableau (tableau 13.1).

**Tableau 13.1**

<b>Le problème de</b>	[décrire le problème]
<b>affecte</b>	[les intervenants affectés par le problème]
<b>Il en résulte</b>	[quel est l'impact du problème]
<b>Une solution réussie permettrait de</b>	[donner quelques bénéfices]

Le but de ce tableau synthétique est de forcer l'équipe à élaborer une formulation concrète du problème.

1. Voir les chapitres 5 *Le backlog de produit* et 6 *La planification de la release*.

2. Un exemple utilisant les différentes pratiques présentées est fourni dans le chapitre 17 *Scrum avec un outil*.

3. [http://fr.wikipedia.org/wiki/Diagramme\\_d'Ishikawa](http://fr.wikipedia.org/wiki/Diagramme_d'Ishikawa)

### Position du produit

La position choisie pour le produit est décrite au plus haut niveau possible. Une technique possible, venant du marketing, est le test de l'ascenseur ou *l'Elevator Statement* (Moore 1991) résumée par le tableau 13.2.

**Tableau 13.2**

<b>Pour</b>	[Public concerné par l'outil]
<b>Qui</b>	[Leur rôle général]
<b>&lt;nom du produit&gt;</b>	[Ce que c'est (outil, système, application...)]
<b>Qui permet</b>	[Utilité]
<b>À la différence de</b>	[Pratique actuelle, concurrence]
<b>Notre produit</b>	[Ce qu'il permet de faire]

La position du produit est présentée en six points et constitue un résumé permettant de comprendre rapidement quelle est la solution proposée. Le nom vient de l'idée qu'une position, destinée à convaincre son chef, doit être exprimée pendant le temps que prend un voyage en ascenseur avec lui.

### 13.2.2 Qualités d'une bonne vision

La vision permet de mobiliser l'équipe vers l'objectif en s'appuyant sur l'intérêt du produit mais aussi sur des dimensions culturelles, psychologiques, voire éthiques. Le choix dépend de la culture de l'organisation et de l'impact qu'aura le produit attendu. La vision doit être ambitieuse pour entraîner les énergies et donner un élan, tout en restant réaliste : ce n'est pas une hallucination. Une bonne vision reprend l'adage de Scrum, l'art du possible.

La vision est partagée avec toutes les personnes intéressées dans le produit. Elle fait la synthèse entre les différents points de vue et fournit le contexte pour tout le monde. Elle présente la stratégie à moyen terme, pour quelques mois. Si le plan de *release* change régulièrement, la vision, elle, devrait rester stable.

La vision s'énonce avec un langage simple et elle est courte : pour que la vision soit lue et partagée il faut qu'elle soit bien résumée et bien écrite. Elle se concrétise par un document court de quelques pages, certains la font même tenir en une seule page. La vision peut aussi prendre la forme de tableaux ou de rubriques d'un wiki. Une personne qui arrive dans le projet se doit de commencer par la découverte de la vision, c'est aussi le premier document lu par quelqu'un qui s'intéresse au produit.

### 13.2.3 Une vision par release

Au début du développement d'un nouveau produit, on ne se pose pas de question : la vision décrit le produit. Le développement se faisant par des *releases* successives, doit-on remettre à jour la vision pour chaque *release* ?

C'est ce que j'ai longtemps fait en ayant une vision unique pour le produit, en l'adaptant à chaque nouvelle *release*. Le problème est qu'on s'y perd entre la vision

de tout le produit et le delta entre deux versions. Un autre problème est de savoir jusqu'où porte la vision, à quel horizon. Un horizon trop lointain risque de donner à la vision trop d'instabilité. C'est pourquoi maintenant, je préconise d'associer une vision à une *release*.

Pour chaque nouvelle *release*, une nouvelle vision : celle du produit à moyen terme, défini par l'horizon de quelques mois correspondant à la *release*.

## 13.3 FEATURES

On peut essayer d'identifier les éléments détaillés du *backlog*. En trouver plusieurs centaines, puis essayer de les ordonner par priorité. Une approche plus efficace consiste à définir une vision du produit, puis à s'intéresser aux *features* : entre la vision, générale, et le *backlog* avec ses *stories*, détaillées pour les *sprints*, un niveau intermédiaire est nécessaire.

### 13.3.1 Justification du terme feature

Dans la littérature relative à l'ingénierie des exigences (agile ou pas), on trouve, en anglais, de nombreux termes pour qualifier des éléments plus gros que des *stories* : *themes*, *epics*, *features*, *minimal marketable features* (MMF).

Après la lecture de *User stories applied* de Mike Cohn, j'ai utilisé un temps *theme* et *epic*. Un *theme* est une collection de *stories*. Une *epic* est une grosse *story*. Je me base maintenant sur le modèle qui me paraît le plus abouti, celui de Dean Leffingwell, résumé dans the « *Big picture for the agile entreprise* <sup>1</sup> ». Il utilise quatre niveaux : *strategic theme*, *epic*, *feature* et *story*. Les deux premiers concernent la vision d'entreprise et sont hors du périmètre de mon propos qui porte sur la vision d'un produit.

Le terme *features* est abondamment utilisé dans le domaine de l'ingénierie des exigences. Il y a même une ancienne méthode qui l'utilise dans son nom : *Feature driven development* ou FDD et on retrouve le terme *feature* dans les visions du RUP (*Rational Unified Process*). Je l'ai traduit pendant plusieurs années par fonctionnalité essentielle. Mais c'est un peu long et désormais je conserve l'anglais *feature* (on prononce « fitcheur »). En fait peu importe les noms, ce qui compte c'est de s'y retrouver.

#### Définition

Une **feature** est un service fourni par le système, observable de l'extérieur, qui répond à un besoin, et dont la description se situe à un niveau tel que toutes les parties prenantes comprennent facilement ce dont il s'agit.

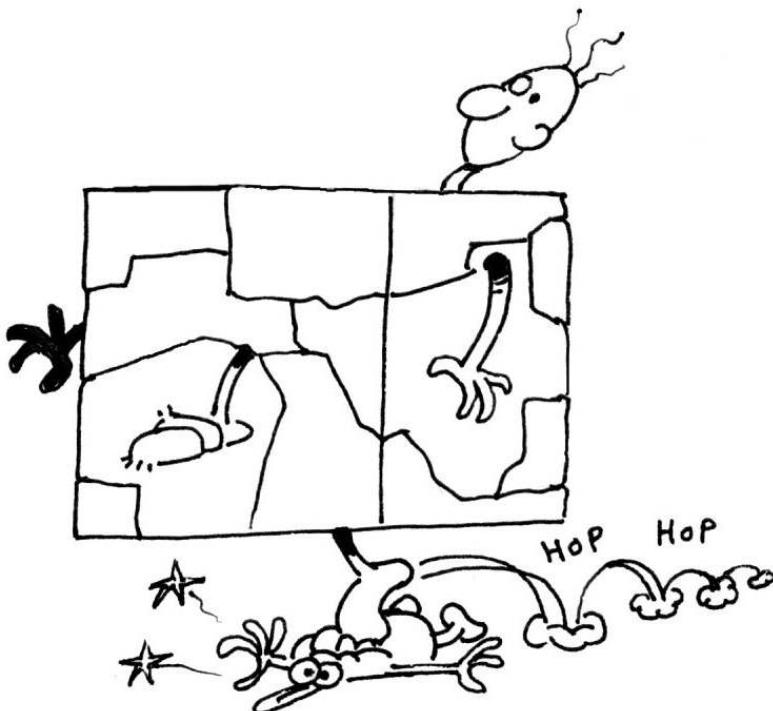
1. <http://scalingsoftwareagility.wordpress.com/2009/08/17/new-whitepaper-the-big-picture-of-enterprise-agility/>

### 13.3.2 Identification des features

L'identification des *features* est faite, de préférence, par des ateliers de travail en groupe, auxquels participent les membres de l'équipe Scrum (si elle est constituée), et les personnes impliquées dans le produit.

« *L'atelier a un côté ludique. Il favorise les échanges, les arbitrages, la levée de certaines incompréhensions. Le format oblige les participants à aller à l'essentiel, à prioriser<sup>1</sup>.* ».

Les ateliers que j'utilise pendant mes formations et sur les projets de mes clients, comme *boîte du produit, souvenir du futur*<sup>2</sup>, *boîte du produit, arbre des features*, permettent à l'équipe d'identifier une liste de *features* initiale de bonne qualité et partagée par tous.



**Figure 13.2** — Le Product Owner lors de l'atelier « boîte du produit ».

Les ateliers sous forme de jeu permettent de canaliser la créativité pour une meilleure définition d'un produit. On trouvera de nombreux projets sur le site Innovation Games<sup>3</sup>.

1. <http://www.qualitystreet.fr/2009/07/29/la-vision-du-produit/>

2. <http://www.aubryconseil.com/post/2007/10/29/320-souvenir-du-futur>

3. <http://innovationgames.com/>

### 13.3.3 Attributs d'une feature

Une *feature* peut posséder les attributs suivants :

- Nom.
- Description.
- Valeur ajoutée (ou utilité) de la *feature*.
- Stories liées (ajoutées quand la *feature* est décomposée en *stories*).
- Taille (généralement obtenue en faisant la somme des estimations de taille des *stories* liées).

### 13.3.4 Features et backlog

Les *features* servent – notamment – à amorcer le *backlog*. La démarche pour obtenir le *backlog* initial d'un nouveau produit consiste en :

- identifier les *features* pour en obtenir de 5 à 20,
- les consolider de façon à constituer des domaines fonctionnels indépendants,
- décider des priorités entre les *features*, par exemple par une session de *priority poker*, présenté ci-dessous,
- pour les deux ou trois *features* les plus prioritaires, décomposer en *stories*, qui sont placées dans le *backlog* initial.

### 13.3.5 Features et priorité

La valeur d'une *feature*, c'est ce qu'elle rapporte à l'organisation, son utilité. C'est un des facteurs utilisés pour déterminer la priorité dans le *backlog*.

Il est extrêmement difficile d'évaluer la valeur financière (en euros) d'une *feature*. En particulier s'il ne s'agit pas d'un produit commercial, il vaut mieux parler d'**utilité**. C'est aussi une responsabilité qu'il est difficile d'exercer en solitaire si on la laisse au Product Owner. C'est pourquoi il est plus efficace d'organiser une réunion sous forme d'atelier et estimer l'utilité des *features* de façon relative, sans unité.

L'atelier peut se dérouler de différentes façons en utilisant différentes techniques. Voici un exemple de ce qui se pratique le plus souvent et qui est appelé *priority poker* :

- chaque participant reçoit un lot de neuf cartes numérotées de 1 à 9 (on peut aussi utiliser les cartes du *Planning Poker*),
- chaque *feature* est étudiée successivement,
- le premier vote porte sur l'intérêt d'avoir la *feature*, chaque participant vote avec une carte, on fait le total des points,
- le deuxième vote porte sur la pénalité relative de ne pas avoir la *feature* dans le produit, chaque participant vote également de 1 à 9,
- on définit l'importance des deux votes, par exemple on peut donner un poids de 3 au premier et le poids 1 au second,
- en faisant la somme des deux votes pondérés, on obtient l'utilité de l'élément.

Pour obtenir l'utilité relative, il faut rapporter cette utilité au total de l'ensemble des éléments, et la priorité d'un élément s'obtient en divisant l'utilité relative par le coût relatif.

Ce genre d'atelier de travail en groupe permet d'établir une première liste de priorités au démarrage d'un projet. Et comme souvent dans les réunions, le bénéfice qui en est tiré vient aussi de la richesse des discussions entre les participants.

Le premier *backlog* de produit est constitué par les *features*, ordonnées par priorité.

## 13.4 RÔLES D'UTILISATEURS

Un produit est destiné à des utilisateurs. La définition des rôles d'utilisateurs permet d'avoir une meilleure connaissance des usages qu'ils pourront faire du produit et d'en tenir compte pour identifier les *features* et les *stories*.

### 13.4.1 Intérêt des rôles

La découverte des *user stories*, comme d'ailleurs celle des cas d'utilisation ou de toute autre technique, se fait en réfléchissant à ce qu'attendent les utilisateurs du logiciel. Il faut donc commencer par identifier ces utilisateurs, au sens large, du logiciel. Les cas d'utilisation utilisent le terme acteur, pour les *stories* on parle de rôle ou de type de rôle ou encore de **rôle d'utilisateur**.

Il faut être vigilant pour ne pas passer à côté de certains rôles, ce qui induirait probablement l'oubli de *stories* importantes.

J'ai fait l'expérience avec deux équipes d'étudiants à qui j'ai demandé d'identifier des *stories* pour une application web. Pour une équipe je ne leur avais pas demandé de réfléchir d'abord à cette notion de rôle et pour l'autre équipe, j'avais exigé une liste de rôles en premier. Dans la première équipe, je n'ai retrouvé que des *stories* liées à l'utilisateur principal, tandis que l'autre, à partir d'une liste de cinq rôles, a identifié bien d'autres *stories*.

Il ne faut pas en rester à un seul rôle qu'on appelle utilisateur, ce n'est pas assez précis, cela va confiner la réflexion à un seul point de vue. Le risque est d'oublier des *stories* et de faire que celles qu'on propose ne seront pas assez spécifiques du problème. Les spécialistes des IHM, les ergonomes, les marketeurs du web le savent bien, il faut aller plus loin et penser aux usages.

### 13.4.2 Identification des rôles

Pour identifier les rôles, on peut s'inspirer des questions suivantes :

- qui fournira, utilisera ou supprimera les informations de l'application ?
- qui utilisera le logiciel ?
- qui est intéressé par une fonction ou un service proposé ?
- qui assurera le support et la maintenance du système ?
- avec quels autres systèmes doit interagir le logiciel à développer ?

Ces questions axées sur le logiciel en développement peuvent aider à démarrer la collecte des rôles.

Dans un esprit de travail collectif, recommandé par les méthodes agiles, il est souhaitable de faire un atelier pour identifier tous les rôles potentiels, puis de les regrouper. Pratiqué juste après l'atelier *features*, un atelier *rôles* permet d'y procéder en une heure environ.

Pour certains types de logiciel, comme le logiciel embarqué, la collecte des rôles est difficile, puisqu'il n'y a pas vraiment d'utilisateurs en relation directe. L'atelier n'en est pas moins utile pour identifier les utilisateurs plus éloignés et les interfaces du logiciel.

Pour compléter la liste, on peut réfléchir aux personnes mal intentionnées qui chercheront à profiter abusivement du logiciel, par exemple les trolls et les spameurs, car il faudra probablement ajouter des *stories* spécialement pour les empêcher.

### 13.4.3 Attributs d'un rôle

Un rôle peut posséder les attributs suivants :

- Nom.
- Description du rôle.
- Sa fréquence d'utilisation du produit.
- Son niveau en utilisation des applications informatiques.
- Le nombre de personnes ayant ce rôle pour le produit final.
- Ses critères de satisfaction dans l'utilisation du produit.

### 13.4.4 Personas

Pour affiner cette notion de rôle utilisateur et bénéficier de la valeur ajoutée d'une conception centrée usage, la méthode des *personas* est particulièrement efficace.

Voilà ce qu'en dit Jean-Claude Grosjean<sup>1</sup>, ergonome agile :

« Un persona, c'est un utilisateur-type, une représentation fictive des utilisateurs cibles, qu'on peut utiliser pour fixer des priorités et guider nos décisions de conception d'interface.

Cette méthode, inventée par Alan Cooper en 1999 dans son best-seller "The Inmates Are Running the Asylum", permet d'offrir une vision commune et partagée par l'équipe des utilisateurs d'un produit, en insistant sur leurs buts, leurs attentes et leurs freins potentiels, et en proposant un format des plus engageants. Les Personas suscitent en effet de l'empathie, un véritable investissement émotionnel : ils prennent très vite une place de choix sur le panneau d'affichage de l'équipe.

Les « rôles utilisateurs » adoptent volontairement un format très synthétique, et restent avant tout sur la relation utilisateur/système : ni éléments fictifs, ni scénario, ni travail d'enquête. La technique des Personas va donc plus loin...elle est aussi plus contraignante.

C'est une démarche avant tout collaborative : les Personas se construisent collectivement au cours de plusieurs ateliers de travail. Cette construction doit impérativement s'appuyer sur des données issues d'entretiens (partie prenantes et futurs utilisateurs), voire de l'observation des cibles et sur l'épluchage de diverses sources (support, presse, études externes, concurrence...) : c'est la spécificité et la force de la méthode !

Quelques conseils :

- Mobiliser l'équipe autour de la démarche.
- Limiter le nombre de Personas.
- Définir idéalement un (deux au maximum) Persona primaire : la cible principale du futur produit.
- Donner vie à vos Personas en créant des fiches contenant au minimum les éléments suivants sans pour autant tomber dans les stéréotypes: prénom, titre/rôle, scénario (le storytelling qui permet de bien rentrer dans la vie du personnage), photo, buts, déclencheurs, freins, features attendues.
- Être vigilant sur le choix de la photo : éviter les célébrités, les proches ou les visages trop parfaits. Votre Persona doit être crédible.
- Communiquer sur vos Personas. »

La technique des personas peut être employée en complément avec Scrum, couplée à des enquêtes sur les comportements, notamment pour les applications web.

## 13.5 USER STORIES

### 13.5.1 Définition

La notion provient d'*Extreme Programming* : une *user story* y est définie comme un rappel pour tenir une conversation qui existe dans le but de faciliter la planification.

---

1. <http://www.qualitystreet.fr/>

Cela montre bien qu'une story doit raconter quelque chose. Ron Jeffries<sup>1</sup> définit trois composants pour une story :

- Une carte pour enregistrer son titre.
- Une conversation pour la raconter.
- Une confirmation pour s'assurer qu'elle est finie, ce qui n'est pas le plus facile.

Pas du tout les mêmes objectifs que le *storytelling* utilisé en marketing et en politique !

Plus simplement on peut dire qu'une *user story* est un morceau fonctionnel qui apporte de la valeur et qui pourra être développé en un *sprint*.

### 13.5.2 Identifier les stories

Le travail de décomposition des *features* en *stories* se fait, de préférence, dans un atelier auquel participe toute l'équipe.

Cet atelier utilise la liste des *features* et la liste des rôles. On part de la *feature* la plus prioritaire pour la décomposer en *stories* dont chacune est enregistrée sur un Post-it, ainsi que le rôle d'utilisateur.

Un atelier permet d'identifier quelques dizaines de *stories* en une session de travail collective de deux à trois heures. À la fin de la réunion, les *stories* pourront être introduites dans le *backlog* de produit.

### 13.5.3 Attributs d'une story

#### *Utilisation du plan type*

Mike Cohn, auteur de livre *User Stories Applied*, préconise l'emploi de la formulation suivante :

**As a <type of user>, I want <some goal> so that <some reason>**  
*(En tant que <rôle d'utilisateur>, je veux <un but> afin de <une justification>)*

La partie justification est optionnelle, parce qu'elle est parfois évidente.

#### **Exemples**

- En tant qu'étudiant, je veux m'inscrire à une formation afin d'obtenir le diplôme.
- En tant que voyageur je veux réserver un billet de train.
- En tant qu'opérateur je veux créer un compte pour un client afin de recevoir son argent.
- En tant qu'organisateur, je veux connaître le nombre de personnes inscrites à la conférence afin de choisir la salle adéquate.

---

1. L'article 3C de Ron Jeffries a été traduit en français par Fabrice Aimetti ; il est disponible sur son blog [www.fabrice-aimetti.fr](http://www.fabrice-aimetti.fr).

Un outil aide à la formulation, par exemple avec trois attributs identifiés pour le rôle, l'exigence et la justification :

- **Rôle** : organisateur
- **But fonctionnel** : connaître le nombre d'inscrits
- **Justification** : savoir si la salle aura la bonne capacité.

Ce plan type permet de se concentrer sur l'utilité apportée par une story à un utilisateur.

#### *Autres attributs d'une story*

- La fréquence d'utilisation de la *story* par le rôle. Une *story* utilisée plusieurs fois par jour devra probablement subir plus de tests qu'une autre utilisée une fois par an.
- Des informations complémentaires sur la *story*. Cela peut-être du texte ou un schéma utile à l'équipe pour le développement, comme un diagramme de séquence UML.
- Les tests<sup>1</sup> associés à la *story*.

### **13.5.4 Techniques de décomposition des stories**

Pour être suffisamment petites et finies en un *sprint*, les *stories* doivent être décomposées. Il existe plusieurs techniques pour cela :

#### *Décomposition par les données*

##### **Exemple avec la story *En tant qu'opérateur je crée un compte***

S'il existe plusieurs types de compte et que la façon de créer les comptes n'est pas la même pour tous, il faudra créer autant de *stories* qu'il y a de types de compte :

- En tant que opérateur je crée un compte espèces
- En tant que opérateur je crée un compte titres

#### *Décomposition selon les actions*

**Exemple** : s'il existe plusieurs opérations sur un compte, il faudra créer autant de *stories* qu'il y a d'opérations ayant un comportement différent :

- En tant que opérateur je crée un compte espèces
- En tant que opérateur je bloque un compte espèces...

Une *story* prête à être développée dans un *sprint* est petite : en moyenne, elle est réalisée en trois jours, tout compris (selon la signification de fini).

---

1. Voir le chapitre 14 *De la story aux tests d'acceptation*

L'intérêt des petites *stories* est double :

- les tests sont plus faciles à identifier,
- cela apporte plus de flexibilité dans la planification car les *stories* décomposées peuvent être planifiées dans des *sprints* différents.

### 13.5.5 Différence avec les use-cases

À la différence des *use-cases*, la technique des *user stories* n'est pas une technique de spécification. Elle repose sur l'idée qu'il n'est pas efficace de rédiger des spécifications détaillées, mais qu'il est préférable :

- de dialoguer pour arriver au détail,
- de rédiger (ou d'écrire) des tests fonctionnels et de les passer pour valider la *story*. Ces tests remplacent la spécification détaillée.

Une autre facette, que n'ont pas les cas d'utilisation, c'est la gestion de projet par l'intermédiaire du *backlog* de produit. C'est pourquoi la question du choix entre cas d'utilisation et *user story* ne se pose pas dans un développement agile : un cas d'utilisation ne remplit pas les conditions pour devenir un élément du *backlog* sélectionné pour un *sprint*.

## 13.6 AMÉLIORER L'INGÉNIERIE DES EXIGENCES

Mais que deviennent les spécifications, les exigences, l'ingénierie des exigences, la traçabilité ?

### 13.6.1 Exigences et spécifications

Pratiquer l'ingénierie des exigences à la mode agile représente un changement dans la façon de procéder comme dans le vocabulaire.

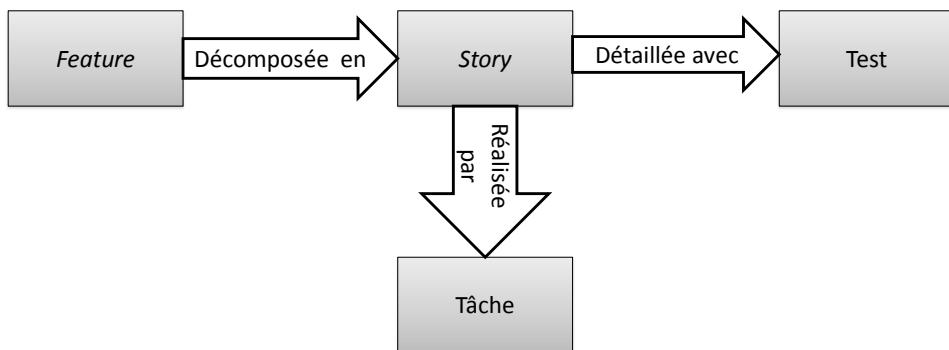
Avec l'avènement des méthodes agiles, on ne parle plus d'exigence, l'élément central devient la *story*. La nouvelle orientation fait que, plutôt d'exiger une capacité à laquelle le système doit se conformer, on se préoccupe essentiellement d'optimiser la valeur apportée aux utilisateurs.

Le gros document de spécification fait au début n'existe plus. Il est remplacé par un *backlog* de produit évoluant régulièrement, dans lequel le comportement attendu est décrit par les *stories*, chacune accompagnée de ses tests d'acceptation.

### 13.6.2 Traçabilité

La traçabilité est la capacité de lier des éléments d'un projet afin d'évaluer l'impact d'une modification d'un élément.

Le fait d'avoir un *backlog* unique dans lequel on trouve toutes les *stories* diminue le besoin de tracer des exigences décrites dans différents documents. Dans le *backlog*, on peut garder la trace entre une *feature* et les *stories* qui sont le résultat de sa décomposition. Scrum apporte aussi la trace entre une *story* et les tâches nécessaires pour la réaliser. La trace entre les *stories* et les tests fonctionnels est assurée : à chaque *story*, on associe plusieurs tests d'acceptation.



**Figure 13.3** — Traçabilité entre *feature*, *story*, *tâches* et *test*

### 13.6.3 Exigences non fonctionnelles

La technique des *user stories* permet d'identifier les exigences fonctionnelles, mais pour un produit logiciel, il y a d'autres exigences, qualifiées de non fonctionnelles, parfois d'exigences techniques. Cela concerne :

- les qualités d'exécution comme l'usabilité, la fiabilité, la performance ;
- les qualités de développement comme la maintenabilité, la portabilité...
- les contraintes de conception, de déploiement, de conformité à des standards...

Il y a plusieurs façons de traiter les exigences non fonctionnelles comme le présentent les paragraphes suivants.

#### Citer les plus importantes dans la vision

Quand une exigence non fonctionnelle a un impact fort sur le produit, elle peut déjà être mentionnée dans la vision.

**Exemples** : on peut trouver dans la vision des contraintes comme la conformité à des standards d'un domaine (comme le DO178B pour l'aéronautique), ou le choix d'un progiciel, ou un type de licence.

### Les mettre dans le backlog

Le *backlog* de produit a vocation à contenir tout ce qui nécessite du travail pour l'équipe, le fonctionnel, comme ce qui est non fonctionnel. L'avantage de tout mettre dans le *backlog* permet d'avoir une source unique, avec des priorités, pour tout ce qu'il y a à faire.

La difficulté, pour une exigence non fonctionnelle, est de faire en sorte qu'elle soit :

- faisable en une itération, selon la signification de fini,
- testable.

Cela demande souvent à découper une exigence générale en petits morceaux. Ce n'est pas toujours facile mais on y arrive pour certaines : la technique consiste à se concentrer sur les tests nécessaires pour la vérifier plus que de sa description.

#### **Exemple**

Plutôt que de dire le logiciel doit être ergonomique (ce qu'on trouve dans des cahiers des charges), on dira : en tant que membre de l'équipe, j'accède à mes tâches en deux clics maximum. Pour une exigence générale comme l'ergonomie, il y aura souvent plusieurs *stories* non fonctionnelles.

Pour mentionner les exigences de performance on s'efforcera d'exprimer ce qui est vérifiable, comme : en tant qu'utilisateur lorsque je me connecte, j'ai un temps de réponse inférieur à deux secondes dans 99 % des cas, même s'il y a déjà 50 utilisateurs connectés, en précisant la configuration du serveur sur lequel faire les tests.

Le Product Owner doit être impliqué dans l'identification de ces *stories* non fonctionnelles : c'est lui le responsable de ce qu'il y a dans le *backlog* et il aura à définir les priorités de ces éléments par rapport aux autres.

### Les définir comme associées à une story

Certaines contraintes peuvent être simplement associées à une *story*, où elles apparaissent comme des cas de test.

**Exemple** : pour une *story* de recherche d'une occurrence d'un mot dans un texte, un cas de test pourrait porter sur la vérification du temps de réponse souhaité en utilisant un gros document.

### Les mettre dans la signification de fini

Les exigences de qualité de développement se déclinent dans la signification de fini. On y trouve aussi des exigences de localisation ou d'utilisabilité qui représentent des contraintes portant sur plusieurs *user stories*.

**Exemple**

IceScrum est un produit utilisé dans le monde entier, il parle anglais (en plus du français). C'est une exigence de localisation. Est-ce qu'elle va dans le *backlog* de produit ? Non ! En tant qu'utilisateur, je veux un produit qui parle ma langue serait une *story* possible, mais elle ne pourrait être faite qu'à la fin quand tout le français serait fait. Or nous voulons que l'anglais, soit fait au fur et à mesure. Chaque fois qu'on ajoute du texte pour une nouvelle *user story*, il doit être accessible en français et en anglais.

Chaque *user story* avec du texte est donc contrainte par cette exigence de localisation. L'exigence « *texte en français et en anglais* » doit être connue de tous les membres de l'équipe : développeurs et testeurs. Une façon de la rappeler à tous est de l'inclure dans la définition de fini.

Il existe d'autres exigences non fonctionnelles du même genre :

- compatibilité avec Firefox, IE7, Chrome et Safari,
- aide en ligne disponible sous forme d'info-bulle,
- compatibilité Java6 et Java5.

Ces exigences non fonctionnelles nécessitent des tests particuliers, avec éventuellement des environnements spécifiques. Cela peut représenter beaucoup de travail.

Porter à la connaissance de toute l'équipe les exigences non fonctionnelles permet d'éviter les surprises. Pour rester avec IceScrum, j'ai appris fortuitement, lors d'une démonstration chez un client, qu'il n'était pas compatible avec IE6. Si nous avions mis en évidence dans la définition de fini ce genre de contraintes, il n'y aurait pas eu de surprise.

#### 13.6.4 Avec quelle équipe ?

Le mieux est de faire participer toute l'équipe à ces travaux et aux différents ateliers organisés avant le lancement du premier sprint. Elle n'est pas toujours constituée à ce moment, cependant il est important que le Product Owner, le ScrumMaster et un architecte soient déjà identifiés et constituent le socle de l'équipe pour les *sprints* suivants.

## En résumé

Pour bien démarrer les *sprints*, il faut disposer d'une vision partagée sur le produit. Une bonne vision présente la position du produit, les rôles d'utilisateurs et une liste de *features*. Cela permet de diffuser à l'équipe la connaissance nécessaire pour développer le produit.

Le *backlog* initial est constitué, en s'appuyant sur la vision, par la décomposition des *features* les plus prioritaires en *stories*.

# 14

## De la story aux tests d'acceptation

À l'occasion d'un audit sur le processus de développement d'une entreprise, j'avais constaté que la documentation relative aux spécifications et aux tests était abondante et, qu'à mon avis, c'était du gaspillage. L'entreprise en question était organisée avec une séparation entre le service des représentants des utilisateurs (maîtrise d'ouvrage) et celui des informaticiens (appelé maîtrise d'œuvre) :

- Le premier rédigeait un document de spécification fonctionnelle du besoin.
- Le second répondait par un document de spécification du logiciel pour montrer ce qu'il avait compris des demandes des utilisateurs.
- Il développait le logiciel et passait des tests unitaires, des tests d'intégration mais aussi des tests fonctionnels écrits en interne.
- L'équipe de test de la maîtrise d'ouvrage rédigeait un cahier de recette, pas fourni au préalable, qu'elle passait sur le logiciel reçu des informaticiens.

Cela faisait quatre documents, tous décrivant le comportement du produit ! C'est pour éviter ce genre de gaspillage que les méthodes agiles préconisent d'avoir un seul référentiel pour les spécifications et les tests.

Un article publié dans le très sérieux magazine IEEE Software<sup>1</sup> fait l'hypothèse de cette équivalence entre les tests et les exigences. La référence au ruban de Möbius illustre comment les deux (exigences et tests) se confondent lorsque le formalisme augmente.

---

1. L'article *Tests and Requirements, Requirements and Tests : A Möbius strip* est signé de Grigori Melnik et de Robert C. Martin, le célèbre Uncle Bob.

C'est l'objectif de ce chapitre de montrer comment les *stories*, accompagnées de leurs tests d'acceptation, constituent une **spécification par l'exemple**.

### Le test n'est pas une phase

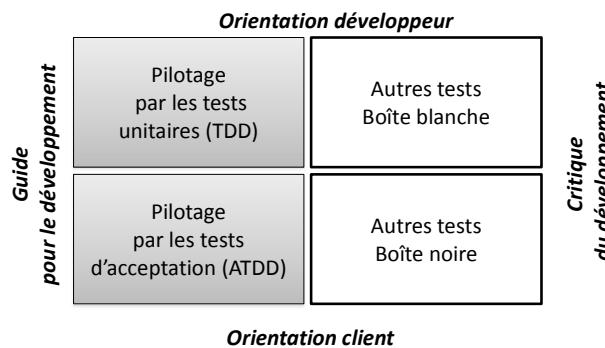
Tester c'est un processus qui consiste à collecter des informations en faisant des observations et en les comparant aux attentes. Avec l'approche itérative de Scrum, le test n'est plus une phase qui se déroule après le développement. Il est intégré dans chaque *sprint*. Dès le premier *sprint*, une équipe commence à tester des *stories*.

Cette façon de procéder permet de réduire le délai entre le moment où une erreur est introduite dans le logiciel et celui où elle est corrigée. On sait depuis longtemps que plus ce délai s'allonge plus le coût de la correction est élevé.

## 14.1 TEST D'ACCEPTATION

Quelle que soit la méthode de développement utilisée, il existe des tests de nature différente. Les méthodes agiles apportent une nouveauté dans la façon de percevoir certains types de tests. En effet, la vue classique du test est la détection d'erreurs a posteriori, après le travail de développement : le testeur, en cherchant des erreurs, vise à critiquer. Or, le test agile a un objectif différent, celui de guider le développeur dans son travail.

Brian Marick met en évidence cette nouvelle vision des tests dans la matrice à quatre quadrants<sup>1</sup> (figure 14.1).



**Figure 14.1** – Quadrants des tests

Cette idée de guide pour le développement transparaît dans le pilotage par les tests (TDD pour *Test Driven Development*), centré sur les tests unitaires<sup>2</sup>. C'est aussi le but

1. <http://www.exampler.com/old-blog/2003/08/21/>

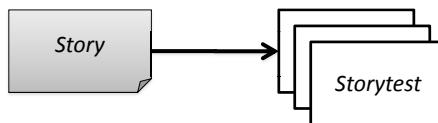
2. Le TDD est abordé dans le chapitre 15 *Scrum et l'ingénierie du logiciel*

pour les tests d'acceptation qui sont des tests orientés client (ATDD, *Acceptance Test Driven Development*).

Le **test d'acceptation** est le processus qui permet d'accepter une story à la fin d'un *sprint*. Il consiste en plusieurs étapes, appliquées à une story :

- Décrire le comportement attendu avec les conditions de satisfaction.
- Transformer ces conditions en cas de test, appelés *storytests*.
- Écrire le code applicatif qui répond au comportement attendu
- Passer les *storytests* sur le code applicatif. En cas d'échec, corriger les tests ou le code.

Ce processus est appelé **pilotage par les tests d'acceptation**.

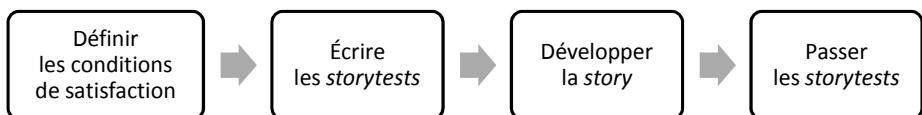


**Figure 14.2** – Une story possède des *storytests*

Une *user story* devrait posséder au moins deux *storytests* associées : un cas de succès et un cas d'échec. Il peut y avoir d'autres cas de tests pour une story, mais un nombre trop important (disons au-delà de huit) est le signe d'une trop grande complexité de la story, qu'il conviendrait de décomposer.

Pour les stories techniques, la notion de test n'est pas pertinente, on en restera au niveau des conditions de satisfaction. De même pour une *user story*, toutes les conditions de satisfaction ne deviennent pas des tests : c'est le cas de celles qui portent sur d'autres artefacts que le produit, par exemple sur de la documentation requise.

## 14.2 ÉTAPES



**Figure 14.3** – Les étapes du processus

### 14.2.1 Définir les conditions de satisfaction

Le principe, pour toutes les méthodes agiles, est qu'une story soit réalisée en une itération. Mais comment savoir si elle est vraiment finie à la fin de l'itération ?

C'est la responsabilité du Product Owner d'accepter (ou non) une story. Pour cela, le moins qu'il puisse faire est de définir ses conditions de satisfaction. Si toutes les

conditions sont satisfaites, la story est acceptée, sinon elle n'est pas considérée comme finie.

Exemple avec la story « Inscription à une conférence ». On peut identifier trois comportements :

- **Inscription acceptée** – C'est le cas de succès, l'inscription d'une personne à une conférence est confirmée.
- **Inscription différée** – L'inscription n'est pas confirmée faute de place et placée en liste d'attente.
- **Inscription refusée** – L'inscription est refusée, la liste d'attente ayant atteint sa limite.

Pour une *user story*, une condition de satisfaction peut être formalisée par un test.

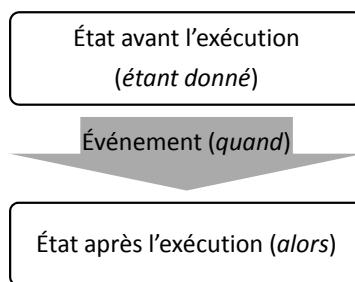
## 14.2.2 Écrire les storytests

### *Formalisme utilisé pour les storytests*

Les différents tests associés à une *story* correspondent à des comportements différents du logiciel. Les comportements diffèrent parce que, en fonction de l'état du logiciel au moment où la *story* est exécutée, les résultats obtenus seront différents. La technique du BDD (*Behaviour Driven Development*<sup>1</sup>) permet de décrire ces comportements.

Chaque test est formalisé avec trois rubriques :

- l'état du logiciel avant l'exécution du test (on parle aussi de précondition ou de contexte du test) ;
- l'événement qui déclenche l'exécution ;
- l'état du logiciel après l'exécution (on parle aussi de postcondition ou de résultat attendu).



**Figure 14.4** — Un test BDD

1. <http://dannorth.net/introducing-bdd>

Le formalisme textuel du BDD est le suivant :

Étant donné le contexte et la suite du contexte  
Quand l'événement  
Alors résultat et autre résultat

Cette façon de faire est particulièrement adaptée à des applications interactives. Elle pousse à avoir des tests courts, puisqu'on y décrit la réponse à un seul événement.

### Exemple avec la story « Inscription à une conférence »

Chaque condition de satisfaction est transformée en test.

#### Exemple pour inscription acceptée

Étant donné l'utilisateur Benji connecté et la conférence AgileToulouse avec le nombre d'inscrits à 134 et la salle A4 d'une capacité de 200 associée à AgileToulouse.  
Quand Benji s'inscrit à AgileToulouse  
Alors l'inscription de Benji est acceptée et le message Vous êtes bien inscrit à AgileToulouse est envoyé à Benji et le nombre des inscrits passe à 135.

#### Exemple pour inscription différée

Étant donné l'utilisateur Pred connecté et la conférence AgileToulouse avec le nombre d'inscrits à 200 et la salle A4 d'une capacité de 200 associée à AgileToulouse et 3 personnes dans la liste d'attente.  
Quand Pred s'inscrit à AgileToulouse  
Alors l'inscription de Pred est refusée et le message Vous êtes en liste d'attente est envoyé à Pred et le nombre des inscrits reste à 200 et le nombre de personnes en liste d'attente passe à 4.

#### Exemple pour inscription refusée

Étant donné l'utilisateur Chipeau connecté et la conférence AgileToulouse avec le nombre d'inscrits à 200 et la salle A4 d'une capacité de 200 associée à AgileToulouse et 20 personnes dans la liste d'attente.  
Quand Chipeau s'inscrit à AgileToulouse  
Alors l'inscription de Chipeau est refusée et le message Il n'y a plus de places disponibles est envoyé à Chipeau et le nombre des inscrits reste à 200 et le nombre de personnes en liste d'attente reste à 20.

### Où stocker les tests ?

Chaque test étant associé à une story, il est considéré comme un attribut de la story et placé avec elle dans le *backlog* de produit (si l'outil employé le permet ; sinon les tests peuvent être mis dans un document annexe mais en gardant la référence aux stories).

## Quand écrire les tests ?

Si la story est réalisée dans l'itération  $n$ , cela implique que les étapes du processus de test d'acceptation s'y déroulent (pour quelques stories, il faut même faire l'étape d'écriture des tests dans l'itération  $n - 1$ ).

Une recommandation est, au moins pour une *story* du *sprint*, que ses tests soient prêts avant le début du *sprint* et que tous les tests soient prêts à la moitié du *sprint*.

Les étapes ne sont pas nécessairement séquentielles, l'ajout de nouveaux tests peut se faire en parallèle avec le développement de la story.

## Qui écrit les tests ?

Scrum met l'accent sur l'équipe sans spécialiser les rôles. Il n'y a pas de rôle de testeur, mais cela ne veut pas dire que l'équipe ne teste pas ! Il existe parfois l'idée que c'est le client qui teste, le client étant représenté par le Product Owner, ce qui peut conduire les développeurs à déléguer au Product Owner tout l'effort de test.

Ce n'est pas une bonne idée : pour des raisons de quantité de travail et de compétences, le Product Owner n'est généralement pas en situation pour s'occuper seul du test d'acceptation et surtout cela doit être un travail collectif.

En fait, peu importe qui rédige les tests, ce qui compte c'est que cela soit fait au bon moment.

### 14.2.3 Développer la story

Le développement de la *story* est mené rapidement pendant le *sprint* ; il dure, en moyenne, trois jours, à plusieurs personnes.

Le pilotage par les tests signifie que l'équipe part des tests d'acceptation pour concevoir et coder la *story*. Pendant le développement, il est fréquent que des *storytests* soient complétés, voire que de nouveaux soient ajoutés.

### 14.2.4 Passer les *storytests*

Pour vérifier que la *story* est bien finie, il faut exécuter ses tests sur la dernière version du logiciel, le **build** courant. Si des tests ne passent pas, la correction est faite aussitôt, l'objectif étant que tous passent avant la fin du *sprint*.

À chaque nouveau *build*, pour éviter les régressions, il conviendrait de repasser tous les tests. C'est une raison pour laquelle il est vite nécessaire de s'intéresser à leur automatisation.

#### Intérêt de l'automatisation

À la première itération, on passe T1 tests. À la deuxième, on passe les T2 nouveaux tests identifiés et on repasse les T1 pour détecter les régressions éventuelles. Cela donne :

Itération 1 : T1  
 Itération 2 : T2 + T1  
 Itération 3 : T3 + T2 + T1  
 Itération n : Tn + .... + T3 + T2 + T1

Avec l'hypothèse d'un nombre moyen  $T_i$  des tests par itération, cela donne pour le nombre de tests à passer :

$$\text{Total} = T_i * n * (n+1) / 2$$

Pour dix itérations avec chacune dix nouveaux tests, le total est de :

$$T_i=10, \quad n=10, \quad \text{Total} = 550$$

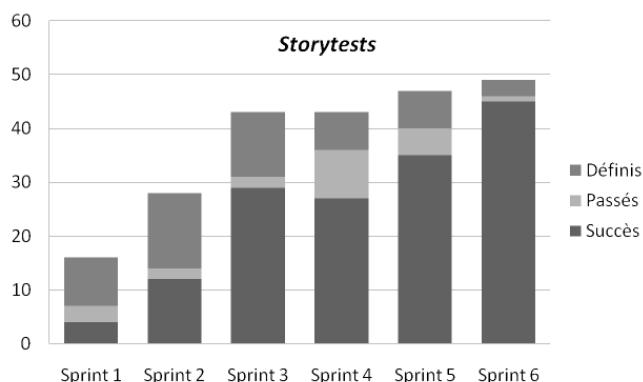
Avec 50 tests par itérations, il devient :

$$T_i=50, \quad n=10, \quad \text{Total} = 2750$$

On imagine que si les tests ne sont pas automatisés, ils ne seront pas tous passés manuellement à chaque sprint et que des régressions peuvent ne pas être détectées.

### *Mesures et indicateurs de suivi de test*

Il n'est pas nécessaire de produire une documentation de rapport des tests comme on en fait dans le développement traditionnel. Il peut être intéressant, lorsque la pratique est mise en place, de faire quelques mesures. Les mesures sur le nombre de *stories* tests existants et sur leur exécution sont importantes pour évaluer la qualité du test. La collecte<sup>1</sup> se fait à chaque *build* et à chaque fin de *sprint*.



**Figure 14.5** — Évolution des tests à chaque sprint – Dans cet exemple, on voit que le nombre de tests passés avec succès a diminué entre le *sprint* 3 et le *sprint* 4, ce qui est le signe d'un problème. L'équipe doit analyser ce qui a causé cette régression et en tirer les conséquences. Cet indicateur met également en évidence qu'il reste des tests en échec à la fin d'un *sprint*.

1. Voir le chapitre 15 *Estimations, mesures et indicateurs*.

## 14.3 GUIDES POUR LE TEST D'ACCEPTATION

À essayer	À éviter
Se servir des tests pour communiquer	Tester une story dans le sprint suivant son développement
Connecter les tests	Stocker les bugs
Planifier le travail de test	

### 14.3.1 Se servir des tests pour communiquer

L'ensemble des stories avec leurs tests remplacent une spécification fonctionnelle détaillée, avec un bénéfice essentiel : la communication est facilitée entre le métier et le développement.

Les membres de l'équipe sont demandeurs de ces tests. Ils s'en servent dans les discussions avec le Product Owner et les testeurs. Ils les complètent si c'est nécessaire. Le référentiel des tests est complété progressivement et toujours à jour.

Cela montre que cette façon de faire – dans façon de faire j'inclus bien plus que du test ; en fait je pense que le mot test est trompeur : au lieu de test d'acceptation, spécification par l'exemple serait probablement plus approprié – est un moyen d'obtenir une meilleure compréhension entre le métier et le développement, et procure un apport absolument fondamental.

### 14.3.2 Tester une story dans le sprint où elle est développée

Un des constats fait en suivant des équipes Scrum qui débutent est que de nombreuses *stories* ne sont pas finies en un *sprint*. Quelques-unes durent même plusieurs *sprints* !

Ce problème est souvent dû à l'accostage développeurs-testeurs. Si un testeur reçoit le logiciel à tester en toute fin de *sprint*, au mieux il découvre des erreurs qui ne pourront pas être corrigées avant la fin du *sprint*, au pire il diffère ses tests au *sprint* suivant.

Ne pas développer, tester et corriger une *story* dans le même *sprint* est un dysfonctionnement sérieux auquel il faut s'attaquer. Pourquoi est-ce un problème ?

- Cela diminue la productivité des développeurs qui doivent se replonger dans le code qui implémente une *story* qu'ils ont développée dans une itération précédente.
- Ce n'est pas satisfaisant pour l'équipe. Elle s'est engagée au début de l'itération à finir une *story* et le résultat montre que ce n'est pas fini.
- Cela rend la planification plus difficile. Une *user story* pas finie est comptée à zéro point pour la vitesse alors que du travail a été effectué dessus. Cette décorrélation entre résultat et travail tend à produire un *burndown chart* de *release* en dents de scie, ce qui peut être perturbant.

Le testeur doit être impliqué dans la planification du *sprint*, s'engager avec le reste de l'équipe et être très réactif. L'équipe doit aussi être capable de refuser d'inclure une story dans un *sprint* si elle estime qu'elle n'est pas suffisamment définie.

### 14.3.3 Ne pas stocker les bugs

Un *story* développée dans un *sprint* est testée dans ce *sprint*. Si un test ne passe pas avec succès, l'équipe doit effectuer la correction au plus vite et au plus tard avant la fin du *sprint*.

Si à la fin du *sprint* tous les tests ne passent pas avec succès, la *story* n'est pas montrée lors de la revue et n'est pas considérée comme finie. On ne stocke pas les bugs, on stocke les tests.

### 14.3.4 Connecter les tests d'acceptation

La technique des « *user stories* » est très efficace couplée à un développement par itérations. Les *stories* alimentent le *backlog* de produit et sont développées pendant l'itération. La tendance est à avoir des *stories* très petites, ce qui présente des avantages en termes de gestion et de suivi. Mais cela a l'inconvénient de rendre les choses plus difficiles à comprendre. Une *story* est à replacer dans un contexte plus large pour que l'on identifie son usage.

En fait une *story* ne suffit pas pour raconter une histoire qui parle aux utilisateurs. Par exemple lors de la revue, à laquelle participent de nombreuses personnes, il convient de préparer une démonstration qui enchaîne des *user stories*. C'est ce qu'on appelle un scénario.

Sur un projet de gestion documentaire auquel j'ai participé, deux scénarios ont été présentés lors de la revue de *sprint*. Nous avions identifié et sélectionné pour ce *sprint* des *stories* comme :

- créer un document,
- télécharger un document existant,
- commenter un document,
- désigner les approuveurs,
- déléguer l'approbation,
- approuver un document.

La démonstration a montré d'abord le cas d'un nouveau document créé et approuvé (scénario 1), puis celui d'un document téléchargé puis délégué dans un mécanisme d'approbation en série (scénario 2).

Souvent les scénarios font référence à des utilisateurs fictifs, appelés *user1* ou *toto*. Il est préférable de choisir de vrais utilisateurs. De la même façon, plutôt que de prendre des documents appelés *doc1*, il vaut mieux s'appuyer sur un exemple réel qui rend les choses plus concrètes et facilite leur compréhension.

Les scénarios sont utiles pour la démonstration en fin de *sprint*, mais c'est mieux de les élaborer bien avant. En début de *sprint*, ils donneront à toute l'équipe le contexte pour le travail de développement.

### 14.3.5 Planifier le travail de test

Pour chaque *story*, on peut identifier deux tâches pour mener à bien le test d'acceptation :

- La spécification des tests de cette *story*, qu'on peut séparer en identification du test et formalisation du test.
- Le passage de ces tests.

C'est du travail qui prend du temps, c'est pourquoi les tâches de test doivent figurer dans le plan du *sprint*.

## Résumé

Le test n'est pas une activité réservée à la fin des développements. Avec les méthodes agiles, les tests d'acceptation sont passés à chaque *sprint*. Le pilotage par les tests d'acceptation pousse même à définir le test d'une *story* avant son développement, pour qu'il serve de spécification par l'exemple à l'équipe.

# 15

## Estimations, mesures et indicateurs

La théorie sur laquelle est basé Scrum (visibilité, inspection, adaptation) nécessite de produire des indicateurs (visibles) pour inspecter et adapter le processus. Dans les chapitres précédents, nous avons vu que l'indicateur emblématique de Scrum était le *burndown* :

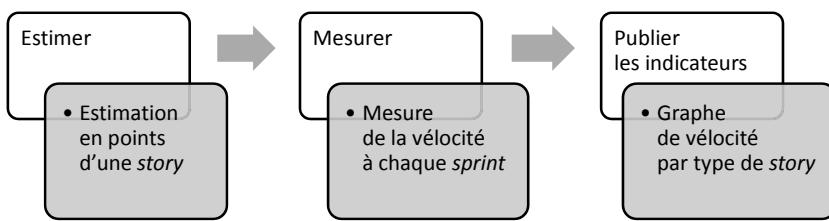
- Le *burndown chart* de *sprint* montre l'évolution du cumul des estimations de reste à faire sur les tâches, sur une base quotidienne.
- Le *burndown chart* de *release* montre l'évolution du cumul des estimations sur les *stories* qui restent à faire, à chaque *sprint*.

Nous verrons que le *burndown chart*, malgré son intérêt, présente des limitations rédhibitoires dans certains contextes pour lesquels d'autres indicateurs sont plus pertinents.

Pour produire des indicateurs, il faut collecter des mesures brutes. Avec Scrum, les mesures les plus importantes portent sur des résultats visibles – un suivi de projet traditionnel porte sur l'avancement de tâches qui ne produisent pas de résultat visible, tandis que le suivi agile s'appuie sur les *stories* finies, qui sont visibles.

Les mesures les plus importantes portent sur des grandeurs qui ne sont pas connues au moment où on en a besoin et qu'il faut donc estimer.

Ce chapitre présente les indicateurs d'un développement avec une méthode agile et montre comment les obtenir, avec quelles mesures et à partir de quelles estimations.



**Figure 15.1** — Exemple d'estimation, mesure et indicateur

## 15.1 ESTIMER LA TAILLE ET L'UTILITÉ

L'estimation est, depuis toujours, un domaine extrêmement difficile dans le développement de logiciel. On sait que le meilleur outil pour estimer se trouve dans l'historique des données collectées (les mesures). Mais il faut bien constater que, dans les projets traditionnels, les mesures sont rarement utilisées par ceux qui estiment. Il faut dire aussi que des mesures ne sont pas souvent collectées et, quand elles le sont, elles ne sont pas toujours utilisables.

Avec Scrum et les méthodes agiles, l'estimation reste un art difficile, mais il y a des différences fondamentales dans la façon de l'aborder :

- **L'estimation est collective** – En particulier, les estimations de taille ou de durée sont faites par ceux qui réalisent.
- **L'estimation se base sur des mesures** – Par exemple, la capacité de l'équipe est estimée à partir de la mesure de la vitesse sur les *sprints* passés.

Nous avons déjà abordé trois situations où l'estimation était pratiquée avec Scrum :

- L'estimation en points des stories pour la planification de *release*.
- L'estimation en valeur des *features* pour aider à définir les priorités.
- L'estimation en heures des tâches pour la planification du *sprint*.

Nous allons revenir sur les deux premières, qui sont les plus importantes et aussi les plus nouvelles.

### 15.1.1 Estimation de la taille des stories en points

#### *La taille du backlog dépend de la taille des stories*

Pour mesurer la taille d'un *backlog*, utile pour planifier, on pourrait se baser sur le nombre d'éléments qu'il contient. Seulement nous avons vu que la décomposition du *backlog* était progressive et qu'à un moment donné, les éléments étaient de taille disparate : le nombre total d'éléments est une mesure, certes intéressante, mais pas assez précise pour avoir une idée du travail qui reste à faire.

C'est pourquoi la taille du *backlog* est obtenue par la mesure de la taille de chaque élément. Évidemment pour planifier, on a besoin de cette mesure avant que la story

ne soit réalisée. La valeur de la taille n'étant pas disponible, il faut l'estimer, et c'est difficile.

### Pourquoi c'est difficile d'estimer la taille d'une story ?

La taille dépend de la compréhension de cette *story* et de la complexité pour la concevoir et la développer – qui est aussi influencée par la qualité de l'architecture et du code. Et toutes ces notions ne sont pas connues avec précision au moment où est généralement faite la première estimation, lors de la planification de release.

L'approche préconisée face à des difficultés est de pratiquer l'estimation en équipe par une séance de *planning poker*<sup>1</sup>.

L'unité d'estimation préconisée est le **point**, sans unité (alors que la pratique courante est de chiffrer en jours). Cela présente l'avantage de différencier les notions de taille et de durée et contraint à pratiquer l'estimation relative, par comparaison.

Une fois chaque story estimée, la taille du *backlog* s'obtient en faisant la somme des tailles des stories qui le composent.

#### De la taille à la durée

Le concept de *timebox* permet de connaître les ressources d'un *sprint*, qui sont fixes si l'équipe est stable et la durée uniforme. C'est là que la **vélocité** intervient : c'est une mesure sur les *sprints* passés qui sert pour estimer la **capacité** de l'équipe pour les *sprints* futurs.

Cette notion permet de faire la relation entre la taille et la durée : il suffit de diviser le total des points à faire pour une *release* par la capacité de l'équipe pour obtenir la date de fin (dans le cas d'un périmètre stable).

**Exemple** : un *backlog* a une taille de 124 points et la capacité de l'équipe est de 26 points. Le nombre de sprints nécessaires est 5 ( $124/26$  arrondi). Connaissant la durée du sprint, 3 semaines, on peut en estimer la durée de l'effort nécessaire, 15 semaines.

La vélocité est évidemment une mesure importante pour les équipes agiles : combinée à la notion de *timebox*, elle rend totalement inutile de collecter des mesures de la durée et du coût de chaque *story* pour planifier à moyen terme.

#### 15.1.2 Estimation de la valeur ou de l'utilité

C'est un précepte de Scrum et des méthodes agiles : on cherche à maximiser la valeur ajoutée. Plus la valeur d'un élément est importante, plus sa priorité est élevée et, comme la priorité définit l'ordre dans lequel les éléments du *backlog* sont réalisés, les éléments avec le plus de valeur sont développés en premier.

1. Le *planning poker* est présenté dans le chapitre 6 *La planification de la release*.

## De la valeur à l'utilité

Mais ce n'est pas facile d'estimer la valeur ajoutée par une story...

Il faut déjà définir ce qu'on entend par la valeur métier : le retour sur investissement, la valeur actuelle nette ? Ensuite, un gros travail d'étude est nécessaire pour estimer la valeur financière que va rapporter un élément du *backlog*.

Personnellement je n'ai pas rencontré d'entreprises qui avaient défini la valeur en euros des fonctions d'un logiciel.

De plus, la valeur est une notion mal comprise : on s'aperçoit que beaucoup la confondent avec le coût. Pour éviter les confusions, il est préférable de changer de vocabulaire et de parler **d'utilité**.

C'est une notion utilisée en économie<sup>1</sup> : l'**utilité** est une mesure du bien-être ou de la satisfaction obtenue par la consommation, ou du moins l'obtention, d'un bien ou d'un service. Elle a aussi l'avantage d'être plus générale : si tous les produits ne visent pas à apporter de la valeur financière, ils ont tous vocation à être utiles. C'est le cas des logiciels Open Source.

## Utilité relative

Comme l'estimation de la taille, celle de l'utilité se fait en points sans unité, et de façon relative. Le Product Owner fait, implicitement, un tri selon l'utilité des stories quand il ordonne son *backlog* par priorité (c'est ce qu'on appelle l'utilité ordinaire).

Pour aller plus loin que l'ordre et obtenir une mesure, il faut évaluer l'utilité de chaque élément (faire de l'utilité cardinale). Il y a plusieurs techniques possibles, comme le *Priority Poker*<sup>2</sup>.

## Sur quels éléments estimer l'utilité

Des *user stories* peuvent être trop petites pour apporter de la valeur par elles-mêmes. C'est pour cette raison que la pratique la plus simple est de définir l'utilité sur des *features* plutôt que sur des *user stories*, ce qui limite le nombre d'éléments à estimer.

À la différence des *user stories*, les *stories* techniques et les défauts, n'ont pas une utilité directe, perceptible par des utilisateurs. Cependant, comme des *user stories* en dépendent, il est possible de leur allouer une utilité indirecte<sup>3</sup>. Les défauts, eux, ont une utilité négative dans la mesure où ils retirent de la valeur à la story sur laquelle ils portent.

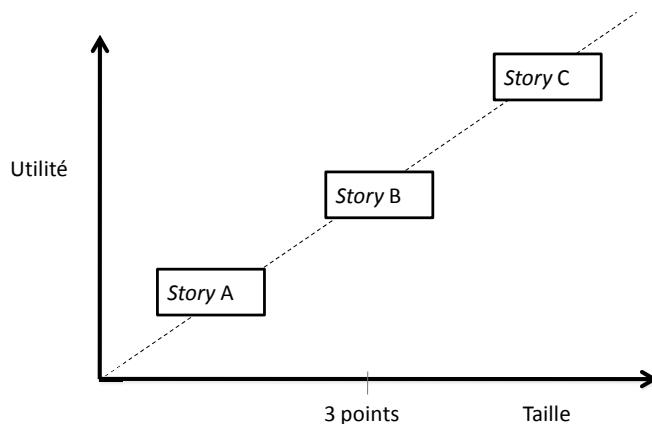
1. <http://fr.wikipedia.org/wiki/Utilit%C3%A9>.

2. Vu dans le chapitre 13 *De la vision aux stories*.

3. Voir à ce sujet les travaux de Philippe Kruchten présentés au Scrum Gathering d'Orlando : [philippe.kruchten.com/kruchten\\_backlog\\_colours.pdf](http://philippe.kruchten.com/kruchten_backlog_colours.pdf).

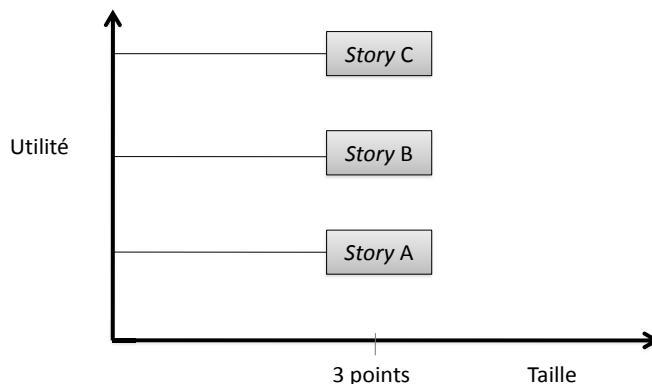
### Utilité et taille

La taille et l'utilité sont statistiquement corrélées : en moyenne, plus la taille est grande plus l'utilité est importante.



**Figure 15.2** — En moyenne, l'utilité augmente avec la taille.

Mais ce n'est évidemment pas vrai pour toutes les *stories* : on connaît tous l'exemple de fonctions faciles à développer qui peuvent être très utiles (ou inversement des « usines à gaz » inutilisables).



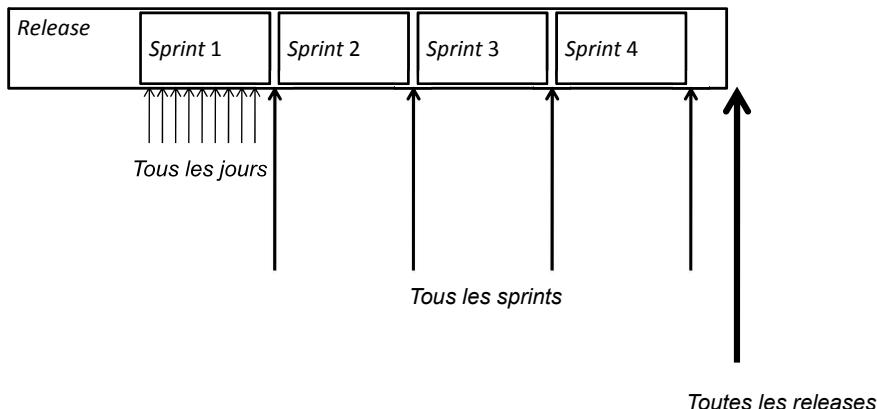
**Figure 15.3** — Trois stories de taille identique peuvent avoir des utilités différentes.

Il est donc intéressant qu'un élément du *backlog* possède deux attributs distincts : sa taille et son utilité.

Le ratio utilité sur taille ( $R = U/T$ ) donne une idée du meilleur retour sur investissement et, plus concrètement, aide à définir les priorités dans le *backlog*. La taille et l'utilité servent donc pour définir la priorité dans le *backlog* et sont collectées pour produire des indicateurs.

## 15.2 COLLECTER LES MESURES

Avec Scrum, les mesures sont collectées au rythme des cycles de régulation : chaque jour, chaque *sprint*, chaque *release*.



**Figure 15.4** – Quand collecter les mesures.

Toutes les mesures présentées ci-après ne sont pas à faire dans tous les projets, c'est à l'équipe de définir celles qui doivent l'être en fonction de ses objectifs et de ses possibilités. Les deux mesures les plus importantes sont celles en relation avec les notions de vélocité et d'utilité.

### 15.2.1 Mesures quotidiennes

Pendant un *sprint*, la collecte suivante peut être faite tous les jours :

- Le nombre d'heures restant à faire pour les tâches du *sprint* non finies (Q1).
- Le nombre de tâches restant à finir (Q2).
- Le nombre de *stories* restant à finir pour ce *sprint* (Q3).
- Le nombre de points de *stories* restant à finir pour ce *sprint* (Q4).

### 15.2.2 Mesures à chaque sprint

À chaque *sprint*, les mesures suivantes peuvent être collectées :

- La capacité estimée au début du *sprint* (S1).
- La vélocité réelle du *sprint* (S2).
- L'utilité ajoutée pendant le *sprint* (S3).
- Le nombre de *stories* restant à faire dans le *backlog* dans chaque état de son « *workflow* » (S4).
- La taille (en points) du reste à faire dans la partie du *backlog* de produit réduite à la *release* courante (S5).

- Le nombre de points total dans le *backlog*, y compris ce qui est fini (S6).
- Le nombre de cas de test d'acceptation écrits et parmi eux, ceux passés avec succès et en échec (S7).

### 15.2.3 Mesures à chaque release

Les mesures de fin de *release* sont les mêmes que celles décrites pour les *sprints* et peuvent être obtenues par le cumul des *sprints* qui composent la *release*.

### 15.2.4 Autres mesures possibles

Dans certains cas, d'autres mesures, faites à chaque *sprint*, peuvent aussi être utiles, de façon ponctuelle (la décision de les commencer et de les arrêter se prend lors de la rétrospective) :

- Le nombre de *builds* produits pendant le *sprint*, pour une équipe qui n'est pas encore passée à l'intégration continue.
- Le nombre d'obstacles restant à éliminer à la fin du *sprint*, pour une équipe qui n'arrive pas bien à les éliminer.
- Les ressources consommées pour le *sprint*, dans le cas où toute l'équipe n'est pas à plein temps sur le projet.
- L'exposition au risque, pour les projets critiques.

À côté de ces mesures orientées gestion de projet, des mesures sur la qualité du code (complexité, taux de commentaires...) sont elles toujours nécessaires.

## 15.3 UTILISER LES INDICATEURS

### 15.3.1 Indicateurs pour le suivi du sprint

L'indicateur représentatif de Scrum est le *burndown chart*. Dans sa forme usuelle, il est réalisé avec les heures restant à faire (Q1). Pour des équipes aguerries, l'estimation des tâches en heures constitue du gaspillage et des variantes possibles sont d'utiliser les mesures Q2, Q3 ou Q4 qui sont obtenues plus facilement.

Une autre possibilité avec ces variantes est de représenter l'avancement plutôt que le reste à faire. Un indicateur intéressant, obtenu avec la mesure sur les *stories* (Q4), est le *burnup* de *sprint* en points.

Ces graphiques<sup>1</sup> sont uniquement destinés à l'équipe dans le suivi de son sprint, ils n'ont pas d'intérêt pour des intervenants extérieurs.

---

1. Voir le chapitre 8 *Le Scrum quotidien*

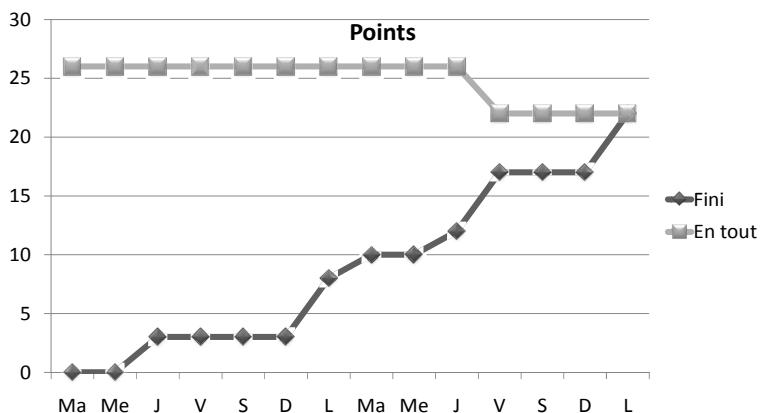


Figure 15.5 — Un burnup de sprint en points

### 15.3.2 Indicateurs pour le suivi du produit

#### Vélocité des sprints

Le graphe de la figure 15.6 présente l'historique de la vélocité (S2) de chaque sprint.

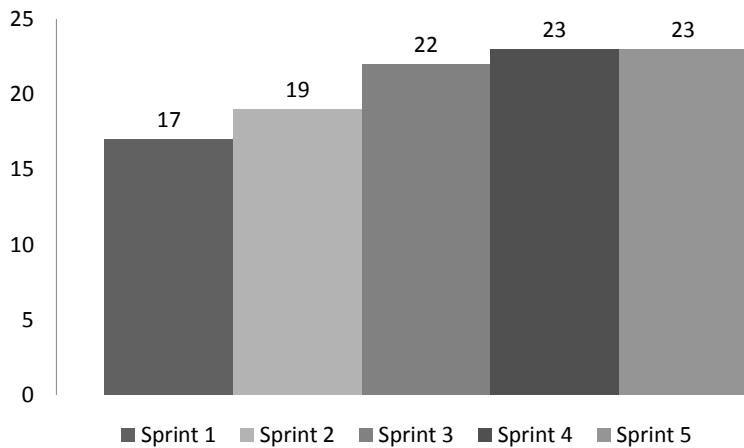


Figure 15.6 — La vélocité des sprints

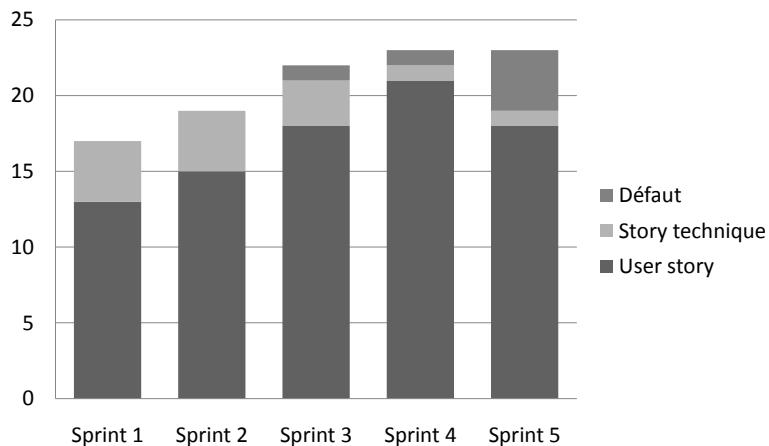
**Usage :** pour estimer la capacité de l'équipe et faire la planification de la release.

**Quand l'utiliser :** dès que possible et tout au long du développement.

**Tendance souhaitée :** croissance après la constitution d'une nouvelle équipe, puis stabilisation. Une diminution de la vélocité après est souvent le signe d'une dette technique.

**Risques :** changements dans les ressources disponibles par sprint, tendance à vouloir une croissance continue de la vélocité (ce qui peut nuire à la qualité et provoquer de la dette technique).

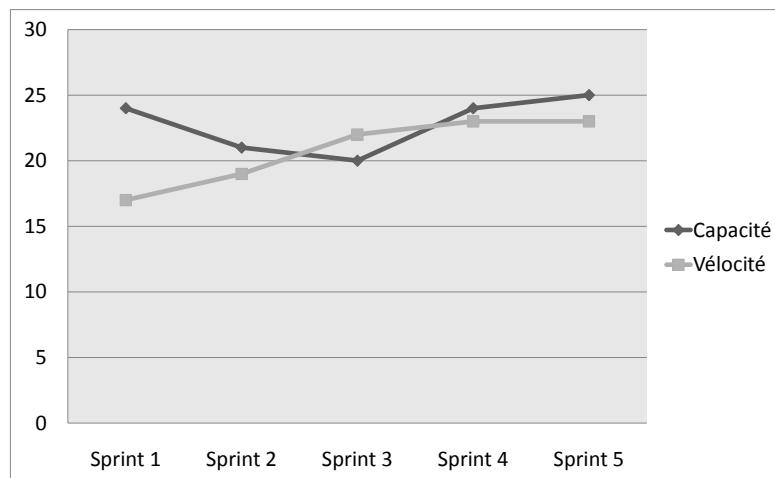
Une variante est de montrer la vitesse par type de story. La visualisation des types de story permet de constater l'importance prise par les stories techniques (plus importante au début d'un nouveau développement) et les défauts (qui peuvent arriver après plusieurs sprints, mais dont la proportion doit rester réduite).



**Figure 15.7** – Historique de vitesse par type de story

### Vélocité vs capacité

Le graphique présente, pour chaque sprint deux points : le premier est la capacité qui avait été prévue au début du sprint (S1), lors de la réunion de planification, le deuxième est la vitesse mesurée à la fin du sprint (S2).



**Figure 15.8** – La vitesse comparée à la capacité

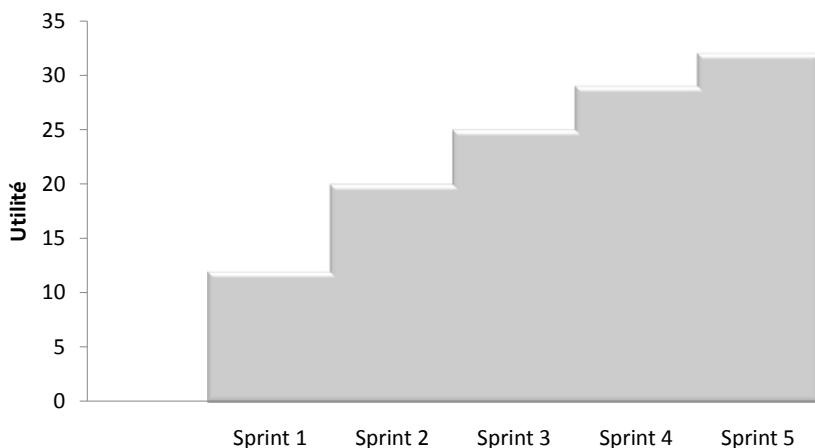
Usage : destiné à l'équipe.

**Quand l'utiliser :** quand l'équipe obtient une vélocité toujours différente de la capacité estimée lors de la réunion de planification du *sprint*. En général, les équipes sont optimistes et la vélocité réelle est inférieure à la capacité prévue.

**Tendance souhaitée :** les deux courbes doivent converger après quelques *sprints* : la vélocité n'est pas systématiquement en dessous (ou au-dessus) de la capacité. On arrête de l'utiliser quand l'équipe a progressé dans ce sens.

### Utilité par sprint

Le diagramme montre l'utilité, cumulée *sprint* après *sprint* (à partir de la mesure S3).



**Figure 15.9** — L'utilité cumulée.

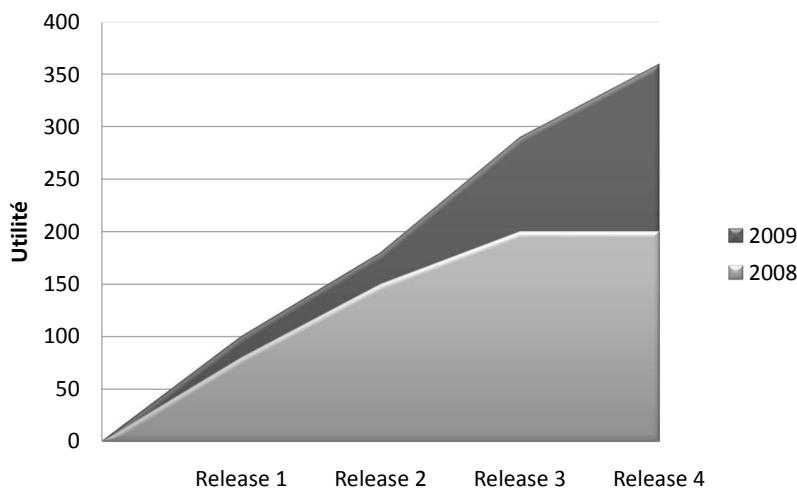
**Usage :** permet de visualiser l'utilité du produit *sprint* après *sprint* et de prendre des décisions sur la fin de la *release* (si le produit présente suffisamment d'utilité, on peut décider de le mettre en production).

**Quand l'utiliser :** dès que possible, mais cela demande beaucoup d'efforts pour estimer l'utilité et la mesurer.

**Tendance souhaitée :** croissance régulière. En principe une bonne gestion des priorités fait que les stories ayant le plus d'utilité sont faites en premier, ce qui donne la forme en escalier avec de plus grandes marches au début.

### Utilité par release

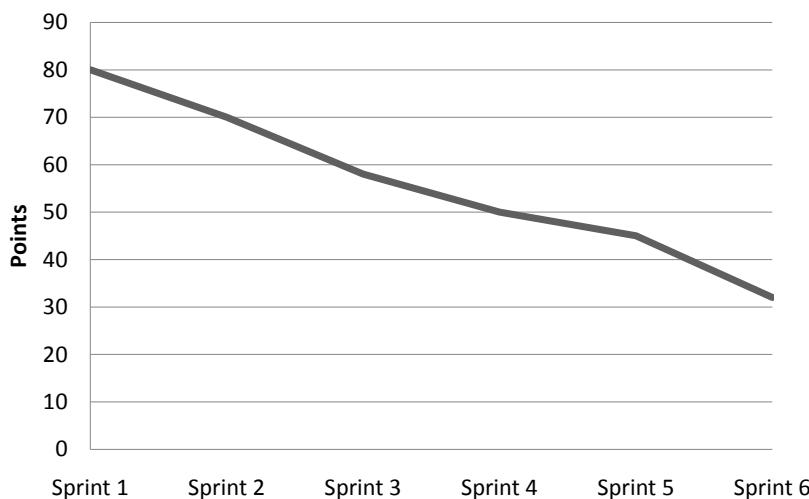
C'est une variante plus simple qui ne porte que sur l'utilité vraiment fournie aux utilisateurs, à la fin d'une *release*. Il faut donc plusieurs *releases* pour que l'indicateur soit significatif.



**Figure 15.10** – Utilité mesurée à chaque *release* présentée pour deux années (l'année 2008 n'a produit que trois *releases*).

### Burndown de produit

Le *burndown chart* montre la taille ce qui reste à faire dans le *backlog*, *sprint* après *sprint*.



**Figure 15.11** – Un *burndown* de produit

**Usage** : permet au Product Owner et aux intervenants de déterminer ce qui reste à faire dans le développement.

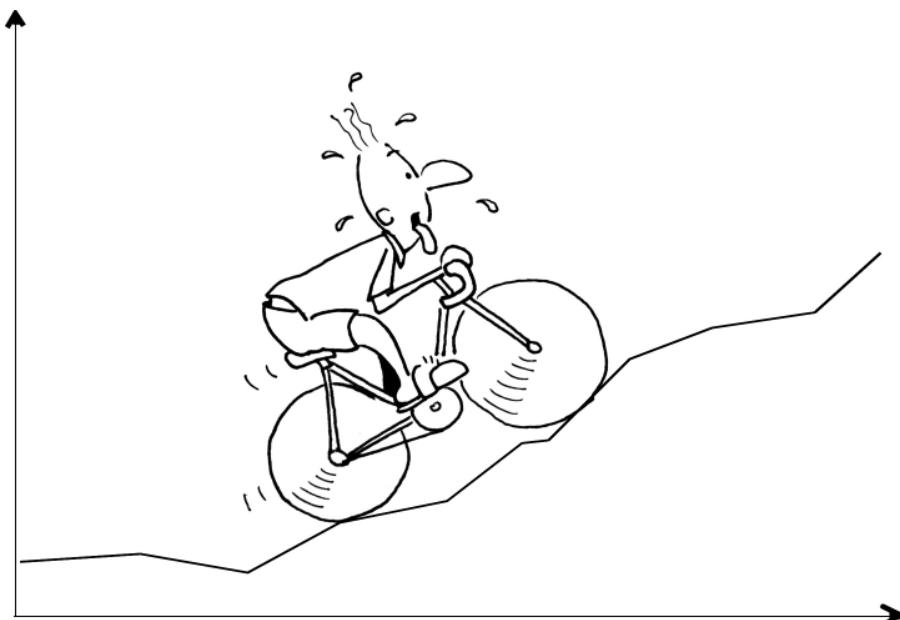
**Quand l'utiliser** : dès que possible.

**Tendance souhaitée** : décroissance régulière (si le périmètre est stable)

**Risques** : on ne voit pas l'influence des changements de périmètre, ce qui rend le graphe difficile à comprendre ; en cas de variation de périmètre importante, la courbe peut remonter. Dans ce cas, le *burnup* permet un meilleur suivi.

### Burnup de produit

J'ai remarqué que la plupart des gens préfèrent les courbes qui montent à celles qui descendent.



**Figure 15.12** — Un *burnup* montre plus l'effort qu'un *burndown* !

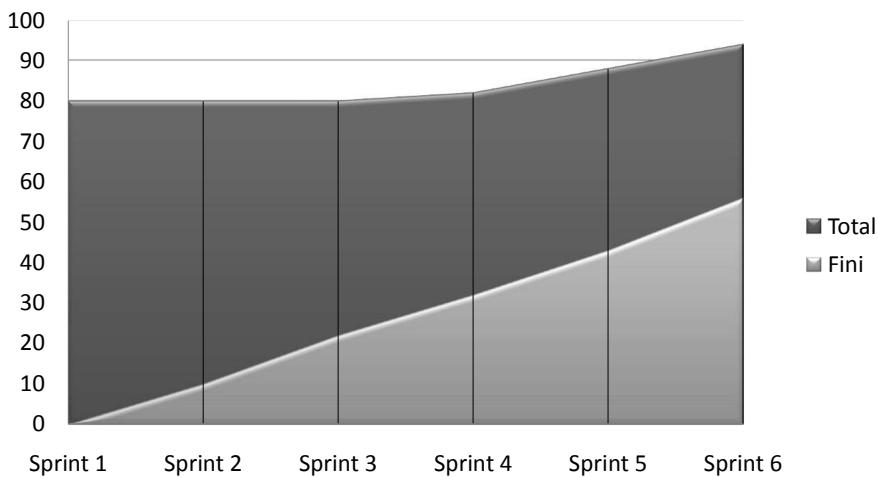
Le *burnup* de produit possède deux courbes : une qui montre ce qui est fini (elle ne descend jamais, il n'y a pas de vitesse négative !) et l'autre tout le travail contenu dans le *backlog*, y compris ce qui est fini (elle peut monter mais aussi descendre : si on supprime des stories qui avaient été déjà estimées, si on ré-estime à la baisse).

Le *burnup* est basé sur les mesures S2 (cumulée) et S6.

**Usage** : permet de visualiser l'avancement et le reste à faire.

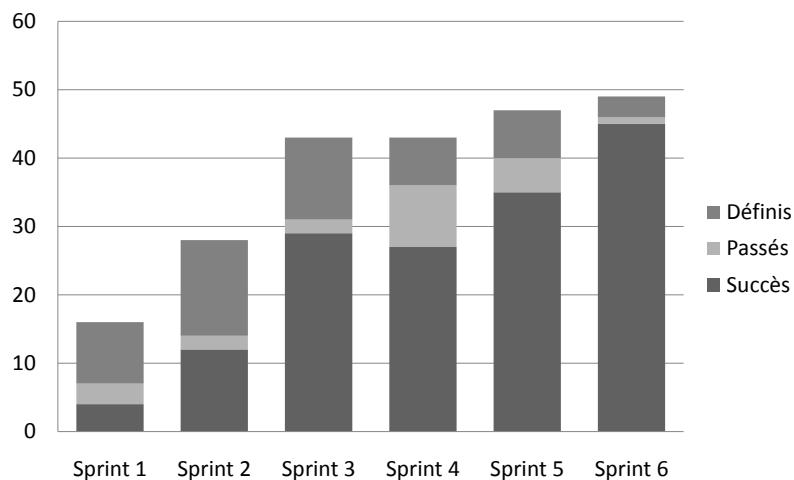
**Quand l'utiliser** : quand le périmètre évolue (ce qui est fréquent), il est préférable au *burndown*.

**Tendance souhaitée** : il est possible que les deux courbes montent en parallèle, cela veut simplement dire qu'on ajoute des éléments dans le *backlog* au même rythme que leur réalisation dans les *sprints*. En fait la tendance souhaitée dépend du contexte.

**Figure 15.13** — Le burnup chart

### Tests d'acceptation

Le diagramme montre le cumul des tests d'acceptation existants et ceux passés avec succès (S7), *sprint après sprint*.

**Figure 15.14** — Le diagramme de tests (*running tested features*)

**Usage** : il permet de s'assurer que les *story* sont testées.

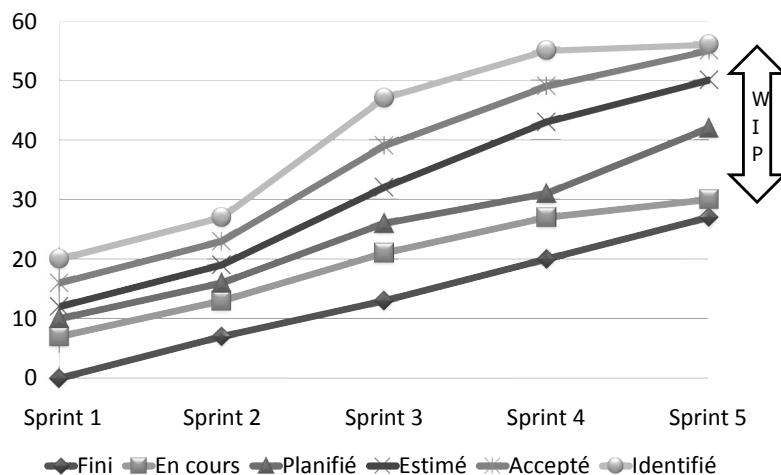
**Quand l'utiliser** : il pousse à faire plus de test d'acceptation et on peut arrêter de le produire quand la pratique est acquise.

**Tendance souhaitée** : croissance régulière dès les premiers *sprints*.

### Diagramme de flot cumulé

Le diagramme montre le cumul de *stories* dans chaque état, *sprint* après *sprint*. Le diagramme de flot cumulé n'est pas basé sur les points mais sur le nombre de *stories* dans chaque état. La mesure est faite à l'activation du sprint, avec six valeurs :

- le nombre de *stories* en tout dans le *backlog*,
- parmi celles-ci, celles qui sont acceptées ou estimées ou planifiées ou en cours ou finies,
- parmi celles-ci, celles qui sont estimées ou planifiées ou en cours ou finies,
- parmi celles-ci, celles qui sont planifiées ou en cours ou finies,
- parmi celles-ci, celles qui sont en cours ou finies,
- parmi celles-ci, celles qui sont finies.



**Figure 15.15** — Le diagramme de flot cumulé

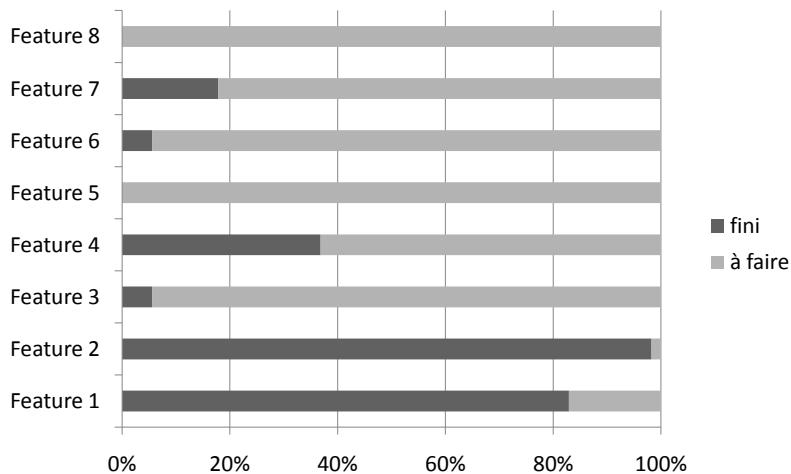
Pour les adeptes du *Lean*, ce diagramme permet de déceler des goulots d'étranglement et de mesurer le débit et le travail en cours (*WIP*, *Work In Process*<sup>1</sup>).

### Parking lot

Le diagramme montre le pourcentage de finition des *stories* associées à un domaine fonctionnel (*feature*).

Il intéressera en particulier les intervenants, spécialisés dans un domaine fonctionnel, participant à la revue de *sprint*.

1. <http://leansoftwareengineering.com/2008/06/12/queue-utilization-is-a-leading-indicator/>

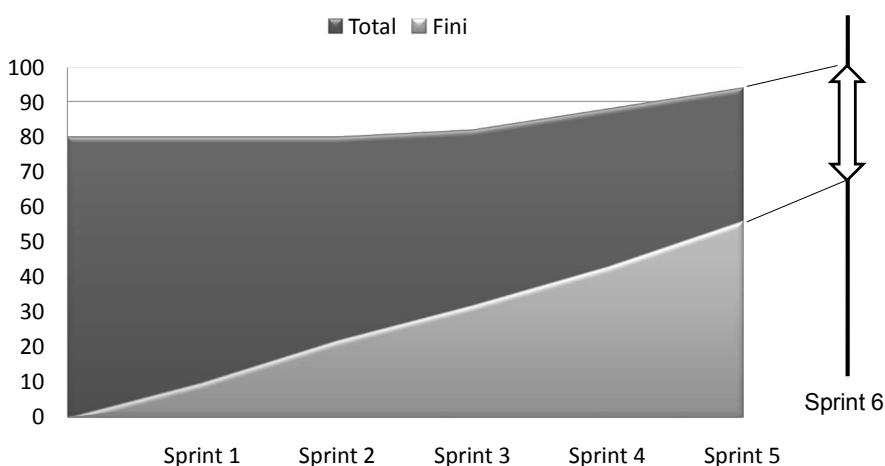


**Figure 15.16** — Le diagramme de *parking lot*

### 15.3.3 Indicateurs pour le suivi de la release

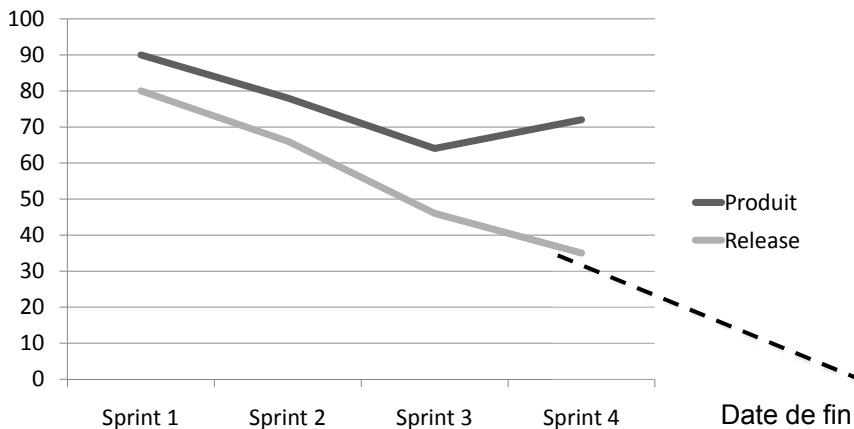
Les indicateurs pour le suivi d'une *release* montrent, en plus de l'avancement, la comparaison par rapport à la cible. Leur objectif est de mettre en évidence des déviations par rapport aux objectifs et de permettre la prise de décision pour des ajustements. Les indicateurs suivants sont utilisables pour le suivi de la *release* : *burndown chart*, *burnup*, *parking lot*, en les adaptant en fonction du type de *release*.

**Release à date fixée** : la cible se définit par une date. On ajoute aux indicateurs présentés pour le produit une barre verticale montrant la date cible.



**Figure 15.17** — La *release* se termine au *sprint 6* : en prolongeant les courbes sur la barre verticale, on identifie la part du *backlog* qui restera à faire.

**Release à périmètre fixé** : chaque story dans le backlog doit disposer d'un attribut supplémentaire indiquant la *release* cible. Les mêmes indicateurs sont produits, en restreignant à la *release*.



**Figure 15.18** — Le burndown de *release* porte sur le sous-ensemble du backlog à finir pour la *release*.

## 15.4 GUIDES POUR ESTIMER, MESURER ET PUBLIER LES INDICATEURS

À essayer	À éviter
Collecter les mesures dès le début d'un développement	Considérer une estimation comme un engagement
Utiliser un outil pour la collecte	Mesurer le temps consommé
Expliquer les indicateurs	

### 15.4.1 Une estimation n'est pas un engagement

Avec les méthodes agiles, une différence fondamentale par rapport à la gestion de projet classique est que les estimations sont ajustées régulièrement. Il convient alors de considérer les estimations faites au début d'un projet comme un budget pour une solution possible, pas comme un engagement sur une solution spécifique.

Cette pratique permet de :

- Faciliter la première estimation, car on sait qu'on pourra ré-estimer si nécessaire.
- Se passer de la marge de protection parfois ajoutée arbitrairement à une estimation par ceux qui craignent d'avoir des reproches en cas de dépassement du chiffre annoncé.

- Éviter qu'une personne qui a fini un travail « en avance » ne le fasse durer de peur d'être accusée de sur-estimation.
- Diminuer le doute. Une estimation est toujours fausse, mais de moins en moins au fur et à mesure des ré-estimations.
- Éviter de perdre du temps à mesurer le réel consommé, à le comparer à la première estimation et à proposer une justification (généralement sans intérêt) de l'écart constaté.

**Si les membres de l'équipe ne s'engagent pas sur leurs estimations, à quoi s'engagent-ils ?**

Une estimation possède une part d'incertitude, cela n'a pas vraiment de sens de s'engager quand l'incertitude est trop grande.

Plutôt que de contraindre une personne à s'engager sur quelque chose qu'elle ne partage pas forcément, l'agilité s'appuie sur la responsabilité individuelle dans un collectif qui partage des valeurs communes : courage, communication, simplicité, *feedback*, respect.

L'engagement est possible quand l'incertitude diminue : lors de la réunion de planification du *sprint*, l'équipe s'engage à réaliser les *stories* sélectionnées parce qu'en identifiant les tâches elle a réduit le degré d'incertitude.

On peut considérer aussi que lors du scrum quotidien chacun s'engage devant ses pairs : lorsqu'une personne de l'équipe dit ce qu'elle va faire d'ici le prochain scrum, elle l'annonce devant toute l'équipe et cela constitue un engagement fort.

### 15.4.2 Pas de mesure du temps consommé

L'estimation du temps qu'il reste à passer sur une tâche est plus importante que la mesure du temps qu'on y a passé.

La pratique habituelle de gestion de projet consiste à estimer la durée de la tâche avant de la commencer, de relever les heures passées effectivement et d'analyser les écarts constatés.

Avec Scrum, on estime aussi la tâche, et on la ré-estime régulièrement (tous les jours si nécessaire !) pendant son déroulement. Le résultat s'appelle le reste à faire (RAF). On ne se préoccupe pas des heures consommées. Une erreur serait de croire que le RAF est l'estimé au départ moins le consommé. En effet on peut avoir estimé une tâche à vingt heures, avoir travaillé deux heures dessus et se rendre compte que c'est plus facile que prévu et que dix heures suffiront pour finir la tâche.

Lorsque je présente la position de Scrum sur ce sujet, j'entends : à quoi servirait de faire des estimations si on ne mesure pas le temps effectivement passé ? Les estimations servent d'abord à planifier. On peut très bien avoir des estimations sans mesure du temps réellement passé. C'est exactement ce qu'on fait avec Scrum. Ce qui nous intéresse, c'est le reste à faire, pas le relevé des heures. C'est avec

l'évolution du reste à faire que des décisions de planification peuvent être prises, par exemple ajuster l'objectif d'un *sprint* en ajoutant ou supprimant des tâches.

La mesure individuelle du temps passé présente de nombreux inconvénients : elle prend du temps, elle n'est pas fiable, elle pousse à considérer que l'objectif d'un projet est de tenir une estimation plutôt que de produire quelque chose,

De plus, elle ne contribue pas à développer l'esprit d'équipe, chacun essayant de se justifier en invoquant le temps qu'il annonce avoir passé ; l'affichage des différences d'heures entre membres de l'équipe peut dégrader l'ambiance.

Enfin, cela ne sert pas à grand-chose. En effet, si on observe un décalage entre une estimation et les heures effectivement passées, on ne peut pas dire si c'est un problème d'estimation ou de compétence ou de définition de la tâche.

Quand une équipe applique Scrum pour la première fois, on constate que certains membres de l'équipe ont beaucoup de réticences à donner une estimation de leurs tâches. Une des raisons est la peur d'être jugé sur la qualité de leur estimation et sur leur productivité individuelle. Cette tendance – naturelle – à considérer une estimation comme un engagement diminue quand le relevé des heures consommées n'est pas mis en exergue et qu'il est substitué par le reste à faire.

#### 15.4.3 Collecter les mesures dès le début d'un développement

La publication d'indicateurs pertinents contribue à l'amélioration des pratiques en permettant à l'équipe d'évaluer les progrès qu'elle accomplit. Elle permet aussi de promouvoir l'usage de Scrum dans l'organisation. C'est pourquoi il est préférable de rendre possible la production d'indicateurs dès le début d'un développement en mettant en place la collecte des mesures les plus importantes, comme la vitesse. L'utilité est une mesure encore plus importante, cependant il faut être conscient qu'elle demande beaucoup plus d'efforts et une grande maturité de l'organisation dans la définition du produit.

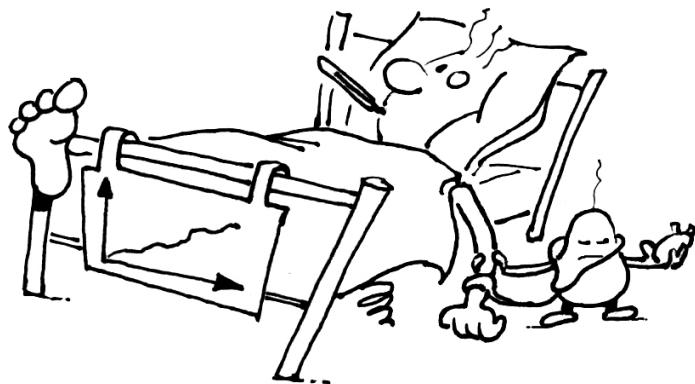
#### 15.4.4 Considérer un outil pour la collecte

La collecte des données prend du temps si elle est faite manuellement. Avec un outil, la collecte est automatique. Les mesures et indicateurs proposés, c'est un critère à prendre en compte dans le choix de l'outil.

#### 15.4.5 Expliquer les indicateurs

Il ne suffit pas de produire des graphiques, il convient de sélectionner les plus pertinents pour les personnes auxquelles ils sont destinés. Les plus simples sont les meilleurs.

Il est aussi nécessaire, en particulier les premières fois, d'expliquer la signification de ces indicateurs.



**Figure 15.19** – Un indicateur simple

## En résumé

Les mesures agiles sont différentes. Elles diffèrent dans leur nature avec l'importance donnée à la valeur ajoutée (ou utilité) et aux résultats visibles (vitesse calculée avec les stories finies).

L'utilité et la vitesse reposent sur des estimations, faites de façon relative, sans unités.

Les indicateurs, élaborés à partir des mesures, sont utiles pour le suivi des plans ; ils servent aussi à l'équipe pour qu'elle évalue ses progrès et améliore ses pratiques.



# 16

## Scrum et l'ingénierie du logiciel

Scrum s'est largement diffusé pour les développements de logiciel et, après l'enthousiasme des premiers utilisateurs, on constate que certaines équipes ont des difficultés à obtenir des produits de qualité. Début 2009, Martin Fowler, James Shore et Robert Martin, des gourous de l'agilité, y sont allés de leur publication sur le thème « *Scrum sans pratiques d'ingénierie, c'est risqué* ».

Ils ont raison : Scrum est un cadre et, selon le domaine, il est impératif d'y ajouter des pratiques complémentaires. En fait, même si Scrum ne présente pas explicitement ces pratiques, leur usage est induit par la définition de fini : pour respecter ce que signifie fini, une équipe est poussée à appliquer ces pratiques d'ingénierie.

Une erreur serait de considérer que ces pratiques sont optionnelles et que la qualité du produit n'est pas un objectif de Scrum. Le code doit être de meilleure qualité possible : certains seront peut-être déçus, mais Scrum ne rend pas possible le développement « à l'arrache ».

C'est le sujet de ce chapitre de présenter comment Scrum doit être complété avec des pratiques d'ingénierie technique du logiciel. Nous aborderons rapidement les pratiques les plus courantes et surtout, nous montrerons comme elles s'intègrent au cadre Scrum.

## 16.1 PRATIQUES AUTOUR DU CODE

### 16.1.1 Intégration continue

L'intégration continue est une pratique de développement logiciel qui conduit les membres d'une équipe à intégrer leur travail fréquemment ; habituellement chaque personne le fait au moins quotidiennement, ce qui conduit à avoir plusieurs intégrations par jour.

#### *Une pratique indispensable*

L'intégration régulière du code de chaque développeur est une pratique essentielle pour le développement itératif.

Comme le dit Martin Fowler : « *Je crois que toutes les équipes devraient pratiquer l'intégration continue. Les outils sont gratuits. Le seul prix à payer c'est d'apprendre* ».

Une intégration produit un *build*, qui peut être utilisé pour passer des tests et permettre le *feedback* du Product Owner. Pendant un *sprint*, une équipe produit de nombreux *builds*.

L'intégration continue garantit des progrès dans l'application en assurant que le code récemment ajouté fonctionne bien avec le *build* précédent déjà validé.

À chaque *commit*<sup>1</sup> d'un développeur, un outil placé sur le serveur d'intégration lance un *build*, suivi de contrôles. L'enchaînement est généralement le suivant :

- Compiler et vérifier le *build*.
- Lancer les tests unitaires.
- Lancer les tests d'intégration et les tests d'acceptation.
- Produire le rapport, avec les erreurs éventuelles.

Si le *build* ne passe pas, l'équipe s'arrête pour résoudre rapidement le problème et relancer la fabrication d'un *build* stable : il ne faut jamais laisser un *build* « cassé » pour éviter de propager des erreurs.

#### *Quand la mettre en place ?*

Il est préférable que l'intégration continue soit en place avant le premier *sprint*. Si l'équipe n'en dispose pas et décide de la mettre en place pendant une *release*, l'installation du serveur et du logiciel constitue une story technique qui va dans le *backlog* de produit. Ce sera à l'équipe de négocier sa priorité avec le Product Owner, en lui expliquant tout l'intérêt qu'il va en tirer : avec l'intégration continue, un Product Owner peut tester les stories dès qu'elles sont développées.

L'intégration continue permet d'avoir à tout moment un logiciel qui marche, ce qui motive l'équipe et aussi les utilisateurs. Elle augmente la transparence en évitant que des travaux d'une personne restent ignorés du reste de l'équipe pendant un certain temps.

---

1. Le *commit* est le fait, pour un développeur, de mettre le résultat de son travail dans l'espace commun à toute l'équipe.

### 16.1.2 Pilotage par les tests

Un développeur qui écrit du code a pour responsabilité de tester son code morceau par morceau : c'est ce qu'on appelle le test unitaire<sup>1</sup>.

La pratique du pilotage par les tests (*Test Driven Development*, TDD) va plus loin : il s'agit d'écrire les tests avant d'écrire le code et de profiter de la présence de tests automatiques pour améliorer le code.

#### *Test en premier*

Le programmeur écrit d'abord les tests unitaires d'un composant : cela lui permet de réfléchir au comportement attendu de ce composant. Ensuite, il écrit le code pour que les tests passent avec succès ; avoir écrit le test avant permet de rester simple au niveau du code. Il continue ainsi en ajoutant de nouveaux tests, puis le code minimal pour qu'ils passent.

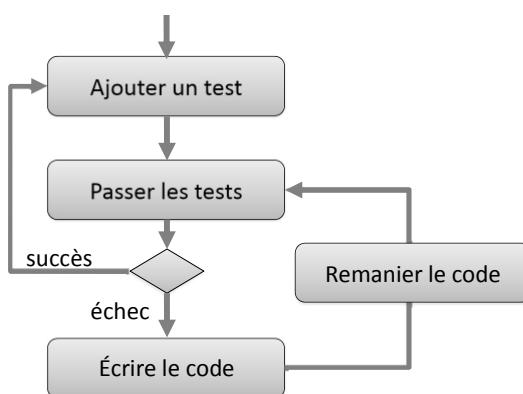
Cette pratique est en fait plus une méthode de conception que de codage, dans la mesure où la réflexion sur le test guide le développeur vers une solution.

Le pilotage par les tests est accompagné du remaniement de code (*refactoring*).

#### *Rémantien du code*

Le remaniement de code consiste à améliorer le code sans changer son comportement. L'objectif est d'améliorer sa qualité en simplifiant et optimisant la solution actuelle. À l'issue de chaque modification, il convient de lancer les tests à nouveau pour s'assurer que le comportement reste celui attendu. Le remaniement de code peut se pratiquer sans élaborer les tests en premier, mais c'est l'intégration des deux qui constitue le pilotage par les tests.

Pilotage par les tests = Tests écrits en premier et remaniement du code



**Figure 16.1** — La pratique du pilotage par les tests

1. Les autres types de test, et notamment le test d'acceptation, ont été présentés dans le chapitre 15.

## Quand commencer ?

Le mieux est de mettre en place les conditions permettant de faire du pilotage par les tests dès le début d'un développement, avant le premier *sprint* d'une *release*. Le niveau de pratique souhaité est renseigné dans la signification de fini : par exemple, si l'équipe considère qu'il n'est pas nécessaire d'avoir des tests unitaires pour certaines parties du code, c'est l'endroit où elle le rend explicite.

Dans le plan de *sprint*, des tâches de test unitaire peuvent être identifiées ou incluses dans celles de codage, selon l'expérience de l'équipe : si elle est novice, il est préférable de les rendre explicites.

Si l'équipe décide de mettre en œuvre ces pratiques au milieu d'une *release*, sur un logiciel déjà existant, il va exister un passif, c'est-à-dire que des parties du logiciel n'auront pas de tests unitaires et auront besoin d'être remaniées. La question qui se pose est la résorption de ce passif, appelé aussi **dette technique**. Comme il est difficile, et pas forcément utile, d'arrêter le développement de toutes les nouvelles *stories* pour se consacrer au remaniement du code, il convient de définir les composants qui doivent être remaniés et de les ordonner par priorité.

Le travail à faire pour remanier et écrire des tests sur des parties existantes prend du temps, c'est pourquoi les travaux doivent être identifiés comme des *stories* techniques et ont leur place dans le *backlog* de produit. Le Product Owner, qui définit les priorités, doit être impliqué pour que ces *stories*, qui visent à améliorer la qualité du produit, soient prises en compte au bon moment.

**Exemple :** une équipe se rend compte pendant le *sprint n* que la qualité du code n'est pas satisfaisante. Lors de la rétrospective du *sprint n*, elle décide que le remaniement est l'action prioritaire. Pendant le *sprint n + 1*, une tâche d'étude est créée pour définir les composants nécessitant du remaniement, puis une réunion a lieu pour fixer les priorités. Lors de la planification de *release* du *sprint n + 2*, en présence du Product Owner, ces *stories* techniques portant sur le remaniement de composants sont ajoutées au *backlog* de produit. Selon leur priorité, certaines seront faites dans le *sprint n + 2*, d'autres plus tard.

### 16.1.3 Programmation en binôme

La programmation en binôme est une pratique qui consiste à mettre deux développeurs devant un seul poste de travail, de façon à ce que leur collaboration améliore la qualité du code.

Un développeur tient le clavier et programme pendant que l'autre observe l'écran, de façon active. La collaboration naît de la différence des points de vue : le premier est orienté vers les détails du code tandis que le second a le recul sur la structure de l'ensemble du programme.

Cette pratique peut être étendue au-delà des développeurs : la constitution de binômes développeur-testeur s'avère également efficace. C'est pourquoi le binômage

(travail en binôme) est plus une pratique de collaboration qu'une pratique réduite à la programmation.



**Figure 16.2** — Pour éviter les dissonances, il vaut mieux une seule personne au clavier

### *Quand en faire ?*

C'est à l'équipe de décider de quelle façon elle met en œuvre la pratique de binômage. Il n'est pas nécessaire d'en faire toute la journée et il est important de permutter régulièrement entre les rôles dans un binôme et entre les personnes pour les binômes de l'équipe.

Utilisé avec discernement, le travail en binôme contribue à améliorer la qualité du produit : le besoin en remaniement sera moindre sur les parties faites en paire.

## **16.2 PRATIQUES DE CONCEPTION**

Autrefois, on distinguait la conception préliminaire de la conception détaillée. Maintenant le terme architecture du logiciel est largement employé. L'architecture est relative à l'organisation des composants et leurs interactions. Pour faire simple, l'architecture est globale et permet de guider des choix de conception fait sur un composant ou pour développer une story.

### 16.2.1 Architecture évolutive

En caricaturant les points de vue, on dirait qu'avec une méthode traditionnelle on est censé élaborer toute l'architecture au début et qu'avec une méthode agile, l'architecture commence avec la première itération et se poursuit régulièrement.

Dans un cadre Scrum, s'il n'est pas recommandé d'être dans le premier cas en figeant très tôt l'architecture, il n'est pas interdit de faire de l'architecture avant le premier *sprint* (heureusement) : la quantité d'architecture nécessaire dépend de chaque produit.

Quelle que soit la quantité d'architecture faite avant le premier *sprint*, il faut partir sur l'idée qu'il y aura encore des travaux à mener pendant les *sprints* : l'architecture est évolutive.

Les gros travaux constituent des *stories* techniques et vont dans le *backlog* de produit. Des travaux d'architecture peuvent demander l'assistance ponctuelle d'un expert, il convient d'anticiper leur planification pour s'assurer de sa disponibilité. Comme pour toutes les *stories* qui n'apportent pas directement de la valeur, une négociation avec le Product Owner est nécessaire pour le convaincre de l'importance de ces travaux pour l'ordonnancement par priorité.

S'il existe un document d'architecture, sa mise à jour se fait à chaque *sprint* et cela fait l'objet d'une entrée dans la définition de fini et se concrétisera comme une tâche dans le plan de *sprint*.

L'architecture évolutive pousse à un changement dans les rôles. Typiquement, on peut identifier deux postures :

- L'architecte qui prend les grandes décisions en suivant les tendances technologique, mais ne participe pas aux travaux de l'équipe (il reste dans sa tour d'ivoire).
- L'architecte qui montre l'exemple en « mettant les mains dans le cambouis » et collabore intensivement avec les autres membres de l'équipe.

Avec Scrum et les méthodes agiles, c'est la seconde posture qui est privilégiée.

### 16.2.2 Conception émergente

La pratique de conception émergente se concrétise par des travaux de conception faits régulièrement, à chaque *sprint*. Nous avons évoqué deux moments où de la conception était faite pendant les *sprints* :

- Pendant la réunion de planification de *sprint*, la conception d'une *story* est faite de façon collective, lors de l'identification des tâches. Elle peut être représentée par un diagramme de séquence montrant les interactions entre les composants collaborant à la réalisation de la *story*.
- L'écriture des tests en premier (pour le pilotage par les tests) participe à la conception ; cette activité est menée pour développer une *story*, par un développeur ou en paire.

Il peut aussi être nécessaire de mener des travaux d'étude ou d'exploration technique pendant un *sprint*. C'est ce qu'on appelle un *spike*.

Le *spike* est utilisé quand l'équipe ne sait pas estimer correctement une *story*. En général, si elle ne sait pas l'estimer, c'est qu'elle ne connaît pas de solution technique pour cette *story* et c'est l'objectif du *spike* d'en identifier une.

Le besoin est identifié lors d'une réunion de planification de *release* (au moment de l'estimation). Le *spike* est alors ajouté au *backlog*, comme *story* technique, estimé et priorisé.

À la fin du *sprint* incluant le *spike*, on devrait :

- avoir défini une solution,
- être capable d'estimer le coût de développement (sa taille en points, en fait) de la *story* objet de l'étude, pour aider le Product Owner à décider quoi en faire.

Il arrive aussi que le *spike* amène à décomposer la *story* initiale en plusieurs autres, plus petites.

## 16.3 MAINTENANCE

### 16.3.1 Il n'y a pas de phase de maintenance

Dans les organisations, on sépare souvent le premier développement d'un produit des mises à jour venant après sa mise en production. On parle de phase de maintenance pour les travaux postérieurs au premier développement et bien souvent les équipes, les procédures et les outils sont différents.

Avec Scrum, cette distinction n'existe pas : les mises à jour sont produites par les *releases* successives. Au cours de ces *releases*, c'est toujours le cadre Scrum et les pratiques agiles qui sont appliqués, le même *backlog* qui continue de vivre et de préférence les mêmes équipes qui développent.

Un point-clé des phases de maintenance est la gestion des bugs et demandes d'évolution.

### 16.3.2 Gestion des bugs

La gestion des bugs, ou plus exactement des défauts, varie selon les projets. Même si l'objectif ultime avec une méthode agile est de ne pas laisser de défauts dans le code, dans la vraie vie des projets il y a toujours des défauts. Et il faut s'en occuper, en gardant à l'idée que c'est moins cher de les corriger tôt que tard.

## Défaut sur une story en cours

Un défaut trouvé sur une *story* en cours de développement dans le *sprint* est une condition de satisfaction en échec : la *story* n'est pas finie. L'équipe ajoute les tâches qu'il faut pour corriger le défaut (code, test) et les réalise pour finir la *story* avant la fin du *sprint*. Le défaut ne va pas dans le *backlog* de produit et encore moins dans un outil de « *bugtracking* », ce serait de la perte de temps et d'énergie.

## Défaut sur une story considérée comme finie

On peut trouver un défaut sur une *story* développée dans un *sprint* précédent et qui a été considérée comme finie. À tort, mais c'est la vie. Rentrent aussi dans cette rubrique les défauts qui portent sur des parties développées avant que le projet mette en place un processus agile.

Un défaut enlève de l'utilité à une *story*, plus ou moins selon sa gravité :

- Un défaut critique empêche le fonctionnement d'une ou plusieurs *stories*, dont l'utilité devient nulle.
- Un défaut majeur ne permet pas un fonctionnement normal et fait perdre une grande partie de l'utilité à la *story*.
- Un défaut mineur fait perdre un peu de valeur à une *story* en rendant son utilisation plus difficile.

Le traitement des défauts varie selon les projets, voici un exemple de ce qui est fait pour le développement du logiciel Open Source IceScrum :

- **Traitements des défauts critiques** – Un défaut critique est traité de façon prioritaire. Dès qu'un membre de l'équipe informé d'un défaut l'estime critique, il crée une tâche dans le plan de *sprint*. Il lui associe un reste à faire d'une heure. Aussitôt, dans l'équipe de développement, une personne (ou un binôme) arrête son travail en cours pour étudier le défaut, identifier sa cause et corriger le défaut.

Si cela est fait en une heure, il suffit alors de déclarer la tâche finie. Si le travail demande plus d'une heure, il faudra alors identifier les tâches nécessaires pour la résolution et les ajouter dans le *backlog* de *sprint*.

Il est probable que cela aura un impact négatif sur la vitesse du *sprint*. C'est pourquoi il convient de ne déclarer comme critique ce qui est vraiment catastrophique pour l'usage du produit, sans solution de contournement.

- **Traitements des défauts majeurs** – Un défaut majeur suit le même processus au début. La différence c'est que si la correction n'est pas finie en une heure, on crée une entrée dans le *backlog* de produit, avec comme type défaut. Le défaut sera alors estimé et corrigé dans un prochain *sprint*.
- **Traitements des défauts mineurs** – Un défaut mineur va directement dans le *backlog* de produit.

Les défauts sont donc collectés dans le *backlog* et suivent la même vie que les autres *stories* : ils sont estimés et priorisés. C'est au Product Owner de décider si la correction d'un bug (non critique) est plus importante que le développement d'une nouvelle *user story*.

## En résumé

L'usage de pratiques d'ingénierie du logiciel est obligatoire pour une équipe Scrum qui développe un produit logiciel.

Les pratiques de développement venant d'*Extreme Programming* comme l'intégration continue, le pilotage par les tests et la programmation en binôme s'intègrent bien dans le cadre Scrum.

Avec Scrum, l'équipe fait de l'architecture évolutive et de la conception émergente. Il n'y a pas de distinction entre le premier développement et les suivants : il n'y a pas de phase de maintenance.



# 17

## Scrum avec un outil

En France on a sûrement trop tendance à mettre l'outil au centre du développement. Je me souviens d'une grande société dans le domaine aéronautique où des armées de développeurs passaient leur temps à outiller la méthode. Sur cet aspect, les méthodes agiles vont dans le sens d'un rééquilibrage, en faisant passer l'outil après la maîtrise des pratiques. Mais certains vont trop loin en interprétant le premier énoncé du *Manifeste agile* « *les personnes et leurs interactions sont plus importantes que les outils et les processus* » comme une recommandation de ne pas utiliser d'outils pour le développement agile.

Il faut évidemment des outils pour développer ; par exemple les pratiques d'intégration continue et de pilotage par les tests ne peuvent être mises en œuvre qu'avec les outils qui vont bien.

En ce qui concerne les pratiques Scrum, le besoin d'outil dépend du contexte. Évidemment, dans le cas d'équipes regroupées dans le même espace, le besoin d'outil est moins fort que dans le cas d'équipes distribuées.

L'objectif de ce chapitre est de montrer comment un outil peut assister la mise en application de Scrum et de l'agilité.

### 17.1 LES OUTILS SCRUM

#### 17.1.1 Les outils non informatiques

##### Cartes

Les cartes sont en fait des fiches bristol sur lesquelles on écrit les *stories*, une *story* par carte. Cet usage est courant dans les premières expériences que font les équipes avec les *user stories*.

J'ai utilisé des cartes pour mes premiers *backlogs* avec Scrum. C'est pratique pour classer les *stories* par priorité quand on est Product Owner, ainsi que pour écrire des

détails sur le dos de la carte au fil des réunions. En revanche, c'est difficile à faire partager sauf à les accrocher ou coller au mur.

### Notes collantes

Quitte à coller autant choisir des notes qui se collent facilement : les Post-it.

Dans une entreprise, on reconnaît l'utilisation de Scrum à la présence de notes collantes de couleur sur les murs des bureaux. Quand une équipe dispose d'une espace de travail ouvert, c'est l'outil le plus efficace pour communiquer entre tous les membres, surtout en ce qui concerne les tâches du *sprint*.

L'utilisation de notes collantes est particulièrement recommandée pour la gestion des tâches d'un *sprint*. Le tableau des tâches est l'endroit privilégié pour la tenue des scrums quotidiens.

Le Post-it se colle plus facilement que la carte et permet de jouer avec les couleurs. En revanche, l'inconvénient, par rapport à la carte, est qu'il n'a qu'une face utilisable. Et surtout le Post-it peut s'envoler, ce qui est fâcheux pour la pérennité des informations qu'il contient.

À Toulouse, le vent d'Autan qui souffle par rafales est l'ennemi des tableaux des tâches avec des Post-it. Je me souviens d'une ouverture de fenêtre après une réunion de planification du *sprint* qui a entraîné une chasse au Post-it, avec l'équipe à quatre pattes pour les récupérer.

Les notes collantes sont aussi très utiles pour le travail collectif et créatif fait lors des différents ateliers permettant de constituer le *backlog* ; dans ce cas la pérennité n'est pas nécessaire au-delà de la réunion.

Mon conseil est d'utiliser les Post-it pour les ateliers de travail et, quand c'est possible, pour la liste des tâches du *sprint*. Pour bichonner un *backlog* de produit, les Post-it ne sont pas suffisants.

### 17.1.2 Les tableurs ou assimilés

Le premier outil informatique utilisé pour Scrum est le tableur, pour gérer le *backlog* de produit. Les attributs sont des colonnes et les *stories* les lignes. Je trouve beaucoup d'inconvénients à l'utilisation d'un tableur. Le plus important est que c'est un outil qui ne favorise pas le partage entre les personnes de l'équipe.

Le tableur en ligne de GoogleDocs permet de pallier cet inconvénient. Je l'ai utilisé sur plusieurs projets, avec une certaine réussite.

Dans certaines entreprises, l'accès à ce type d'applications n'est pas autorisé. Le *backlog* peut alors être un tableur mis sur un Intranet, mais généralement l'utilisation n'est pas partagée.



**Figure 17.1** — Product Owner essayant de capitaliser après un atelier avec Post-it

Un outil permettant le partage sur un Intranet que j'ai utilisé sur plusieurs projets, parce que c'était le choix de mes clients, est SharePoint. Cet outil permet de gérer des listes, ce qui est plus adapté qu'un tableau pour un *backlog* : l'avantage par rapport à une feuille de calcul est que la liste est visible sur le portail sans qu'il soit nécessaire d'ouvrir un fichier. Cependant c'est très pénible à utiliser dans une optique Scrum, notamment pour la gestion des priorités, pour la production des *burndown charts*...

On arrive vite à des limites rédhibitoires pour tous ces outils, qui sont détournés de leur usage d'origine pour faire du Scrum : ils n'apportent aucune aide méthodologique.

### 17.1.3 Les outils spécifiques

Il existe de nombreux outils dédiés à Scrum et notamment à la gestion du *backlog* de produit. En cherchant un peu, on en trouve à plusieurs dizaines, c'est un sujet qui semble inspirer les innovateurs. Le ticket d'entrée, comme on dit, n'est pas très élevé pour réaliser un outil qui gère simplement un *backlog*.

Il existe des outils commerciaux et des outils libres, des outils fonctionnant selon différents modes, avec différentes technologies.

Je ne vais pas présenter les outils commerciaux, je vais prendre un outil Open Source que je connais bien – j'en suis le Product Owner – pour donner un aperçu de l'aide que peut apporter un outil.

## 17.2 UN EXEMPLE AVEC ICESCRUM

La version R2# 14 d'IceScrum<sup>1</sup>, sortie en juillet 2009, a été utilisée pour l'exemple.

L'exemple va permettre de simuler le développement d'un site communautaire pour une association qui organise des événements. Il s'agit de l'association Omega qui organise tous les trimestres des séminaires d'information. Ces séminaires accueillent des professionnels, des étudiants et enseignants.

### 17.2.1 Les rôles Scrum

Pour réaliser le site Omega, imaginons des membres de l'association qui travaillent à mi-temps sur ce développement. Ils se retrouvent pour des réunions mais travaillent chez eux la plupart du temps. Clodio va jouer le rôle de Product Owner, au moins au début. Il se connecte sur IceScrum et c'est lui qui crée le produit (dans l'outil un projet et un produit se confondent).

Date de début - 26/07/2009						
		juillet				
lun.	mar.	mer.	jeu.	ven.	sam.	dim.
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

**Figure 17.2** — L'assistant de création

### Une gestion dynamique des rôles

La personne qui crée le produit devient automatiquement Product Owner et Scrum-Master : elle possède ces deux rôles en même temps, pour lui faciliter l'usage du logiciel, en attendant que d'autres membres le rejoignent pour constituer l'équipe. Le principe, dans IceScrum, est de laisser une grande liberté à l'équipe et aux personnes qui la composent. Les membres de l'équipe s'inscrivent eux-mêmes et chacun choisit son rôle.

1. Disponible en ligne sur le site IceScrum : [www.icescrum.org](http://www.icescrum.org).

Les rôles de ScrumMaster et Product Owner restent dynamiques, c'est-à-dire qu'ils peuvent changer si cela est nécessaire : par exemple, lors d'une absence, un membre de l'équipe peut prendre le rôle de ScrumMaster ou de Product Owner.

Les personnes qui sont intéressées par le produit sans participer à son développement prennent le rôle de StakeHolder.

### *Des responsabilités selon les rôles Scrum*

Dans IceScrum comme dans Scrum, la notion d'équipe est primordiale : l'usage de l'outil n'est pas réservé à une seule personne qui en devient le spécialiste. Dans cette optique, la spécificité des rôles est limitée à l'essentiel :

- Un équipier peut créer des *stories* dans le *backlog* de produit, créer des tâches dans le plan de *sprint*, créer des tests d'acceptation, noter le résultat des tests et enregistrer un obstacle.
- Le ScrumMaster a la responsabilité supplémentaire de gérer l'élimination des obstacles et d'indiquer qu'un nouveau *build* est utilisable.
- Le Product Owner est le seul habilité à créer une *release*, à définir les *features*, à changer les priorités dans le *backlog* et à déclarer une *story* finie.

Les *stakeholders* peuvent ajouter une *story* et ont l'accès en lecture à tout le reste, en particulier les rapports graphiques.

## **17.2.2 Démarrage d'une *release***

Avant d'activer le premier *sprint*, il faut définir ce que va faire le produit. L'équipe se rencontre physiquement pour définir la *roadmap*, le plan de *release* et le *backlog* initial ; les informations sont collectées dans IceScrum au fur et à mesure.

### *Création de releases dans la vue roadmap*

Une *roadmap* montre la vie du produit à long terme. Elle est présentée sur une ligne de temps, avec les *releases* successives et les *sprints* contenus dans ces *releases*.

Pour Omega, l'équipe décide de lancer le développement en juillet avec une première *release* qui est espérée en octobre, pour la grande fête de l'association.

La *roadmap* est alimentée par la création de la première *release*.

Après discussion, l'équipe envisage de sortir une deuxième version R2 en décembre.

Ces deux *releases* sont créées avec la date de fin fixée, qui pourra être modifiée ultérieurement.

Roadmap

Nom de la release	R1
But de la release	Permettre les inscriptions pour la conférence annuelle.
Durée d'un sprint en jours	21
Vélocité	20.0

Date de début de la release						
		juillet				
lun.	mar.	mer.	jeu.	ven.	sam.	dim.
			1	2	3	4
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Date de fin de la release						
		octobre				
lun.	mar.	mer.	jeu.	ven.	sam.	dim.
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Figure 17.3 – Créeation de la *release* R1 qui se termine le 10 octobre



Figure 17.4 – La *release* R1 apparaît dans la vue *Roadmap*



Figure 17.5 – La roadmap avec les deux *releases*

## Vision

Lors d'une réunion, l'équipe Omega définit :

- l'énoncé du problème,
- la position du produit.

Le Product Owner entre ces informations dans la vision associée à la première *release*, avec deux tableaux.

**Tableau 17.1** — Définir le problème

<b>Le problème de</b>	l'organisation artisanale de l'association.
<b>Affecte</b>	les organisateurs et les membres de la communauté.
<b>L'impact du problème est</b>	un manque de visibilité sur les événements.
<b>Une solution réussie permettrait de</b>	présenter une vitrine de l'association plus attractive, ce qui permettrait d'avoir plus de participants aux événements.

**Tableau 17.2** — Donner la position du produit

<b>Pour</b>	la communauté.
<b>Qui</b>	est intéressée par les activités de l'association.
<b>OMEGA</b>	est une application web.
<b>Qui permet</b>	d'élargir l'écho de l'association.
<b>À la différence de</b>	la pratique actuelle : un site statique simple.
<b>Notre produit</b>	offre une vitrine en ligne où l'on peut connaître les manifestations organisées et s'y inscrire.

## Rôles d'utilisateurs

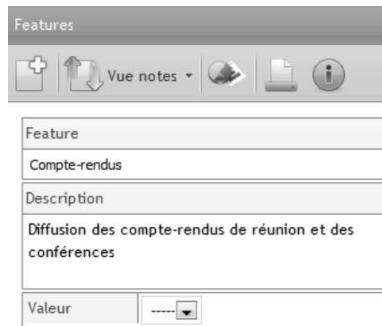
IceScrum permet de collecter des informations sur les rôles et sur la façon dont ils sont susceptibles d'utiliser le produit.

Id	Nom du rôle d'utilisateur	Nombre d'instances	Description	Critères de satisfaction	Niveau d'informatique	Fréquence d'utilisation
29	Organisateur	2-5	Personne qui décide des dates des séminaires et les annonce	Gain de temps avec l'application, inscriptions facilitées et informations accessibles	Moyen	Semaine
30	Sponsor	1	Personne qui participe financièrement ou matériellement à l'organisation de manifestations	La manifestation lui permet d'obtenir de nombreux contacts	Faible	Mois
31	Visiteur	100+	Personne qui vient sur le site de la communauté	Accès facile aux informations	Moyen	Jour
32	Membre	11-100	Personne qui s'est inscrite comme membre de la communauté	Il peut contribuer facilement	Elevé	Mois
33	Animateur	6-1C	Personne qui fait une présentation à une conférence		Elevé	Mois

**Figure 17.6** — Les rôles d'utilisateurs et leurs caractéristiques pour Omega

## Features

Dans IceScrum, une *feature* est gérée dans une liste indépendante, la liste des *features*. En cas de nouveau produit, l'approche est descendante et les *features* sont identifiées en premier.



**Figure 17.7** — Crédit d'une *feature*

Pour Omega, les *features* sont identifiées en groupe et le Product Owner les ajoute dans la vue Features.

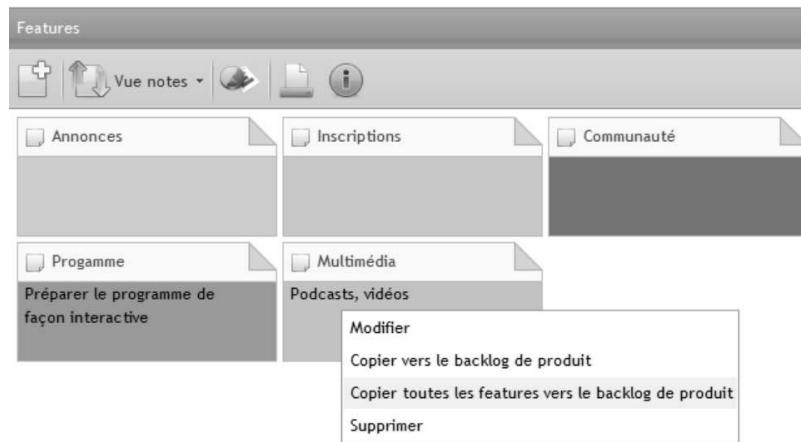
Features				
		Vue tableau ▾		
Id	Feature	Valeur		Description
30	Annonces			
31	Inscriptions			
32	Communauté			
33	Compte-rendus		Diffusion des compte-rendus de réunion et des conférences	
34	News		Ecrire des news pour la communauté	
35	Programme		Préparer le programme de façon interactive	
36	Multimédia		Podcasts, vidéos	

**Figure 17.8** — La liste des *features* pour Omega

Chaque *feature* peut être repérée par une couleur, pour faciliter l'identification des *stories* qui y seront associées. L'attribut valeur permet d'exprimer la valeur ajoutée par une *feature*. Après avoir identifié les *features*, entre une dizaine et une vingtaine en général, on peut les copier dans le *backlog* pour l'initialiser. Il est possible de les copier toutes ou seulement quelques-unes (figure 17.9).

## Backlog de produit initial

Le *backlog* de produit IceScrum montre à l'ouverture les *stories* qui sont à prioriser. L'outil permet de filtrer les éléments du *backlog* selon leur état. La vue « à prioriser » regroupe les *stories* qui ne sont pas encore planifiées, c'est-à-dire celles qui ne sont pas associées à des *sprints*.

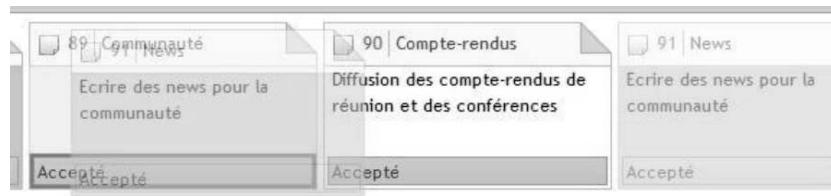


**Figure 17.9** — Menu pour copier les *features* dans le *backlog*

## Priorités

Comme les *features* ont été copiées dans le *backlog*, une première passe sur les priorités s'effectue à ce niveau-là, sur environ une vingtaine d'éléments. Pour Omega, le Product Owner fait une première passe sur les priorités.

Dans cette vue, le Product Owner définit les priorités en déplaçant les notes décrivant les *stories*, la priorité la plus élevée étant en haut à gauche de la vue.



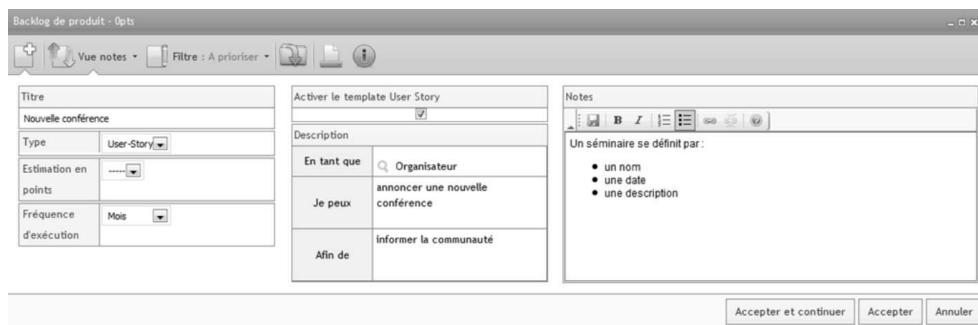
**Figure 17.10** — Le changement de priorité s'effectue par glisser-déplacer d'une *story*

Dans l'exemple précédent la note News est déplacée de sa position et lâchée sur une note à sa gauche : elle va passer plus prioritaire que la note sur laquelle elle est déposée.

## Ajout de *stories*

Lors de l'ajout d'une nouvelle *story*, on peut définir son type (*feature*, *user*, *technical*, défaut), donner une estimation (mais c'est généralement fait lors du *planning poker*) et décrire la *story*.

Dans le cas d'une *user story*, la description est facilitée par la mise à disposition du template « *en tant que rôle, je peux...* ». L'outil propose, pour le rôle, un de ceux rentrés dans la liste des rôles utilisateurs.



**Figure 17.11** — Création de la story « Nouvelle conférence »

Lors du démarrage, le *backlog* initial peut comporter jusqu'à une cinquantaine d'éléments, les plus prioritaires étant ceux dont la granularité est la plus fine. Pour Omega, le travail de décomposition en stories s'effectue sur les trois *features* les plus prioritaires : Annonces, Inscriptions et Compte-rendus. Toute l'équipe participe à l'identification des stories et peut les entrer dans l'outil. Il est aussi possible de décomposer une story en plusieurs (figure 17.12).



**Figure 17.12** — Décomposition d'une story

Pour la story Programme, un schéma a été fait en réunion, il est pris en photo et l'image est associée à la story (figure 17.13).

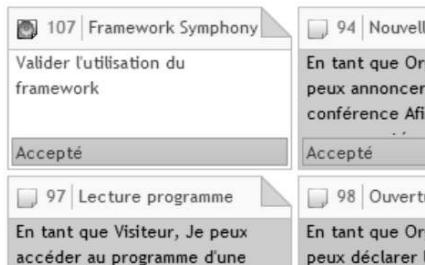
Le développement Omega étant nouveau, il n'y a pas de story de type défaut dans le *backlog*. L'équipe identifie une story technique qui porte sur un *framework* suggéré par le spécialiste des architectures web (figure 17.14).

### Estimation

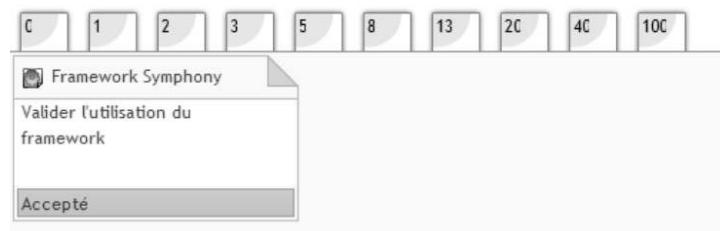
L'estimation des stories se fait avec un *planning poker*. Comme l'équipe Omega ne pouvait pas se regrouper facilement, le *planning poker* a été fait à distance, avec toute l'équipe connectée à IceScrum.

Titre	
Programme	
Type	User-Story
Estimation en points	-----
Fréquence d'exécution	Jour
Fichier attaché	tree.jpg

**Figure 17.13** — Association d'un fichier à une *story*



**Figure 17.14** — Une *story* technique dans le *backlog* de produit



**Figure 17.15** — Début d'une séance de *planning poker* sur IceScrum

À l'issue de la séance d'estimation, le *backlog* est prêt pour la planification de la *release*.

Backlog de produit - 68pts					
Vue notes • Filtre : A prioriser •					
107   Framework Symphony	94   Nouvelle conférence	95   Accès conférence	87   Programme	96   Relance	
En tant que Organisateur, Je peux utiliser le framework	En tant que Organisateur, Je peux annoncer une nouvelle conférence Afin de informer la	En tant que Organisateur, Je peux indiquer le lieu et fournir le plan d'accès à la conférence	En tant que Organisateur, Je peux fournir le programme Afin de attirer des curieux	En tant que Organisateur, Je peux définir les rappels pour les relances des inscriptions	
Estimé 5	Estimé 2	Estimé 1	Estimé 3	Estimé 5	Estimé 5
97   Lecture programme	98   Ouverture des ins...	100   Inscription	99   Suivi inscriptions	101   Annulation inscr...	
En tant que Visiteur, Je peux accéder au programme d'une conférence	En tant que Organisateur, Je peux déclarer les inscriptions pour une conférence ouvertes	En tant que Visiteur, Je peux minscrive à une conférence Afin de réserver ma place	En tant que Organisateur, Je peux connaître le nombre de personnes inscrites à une	En tant que Visiteur, Je peux annuler mon inscription Afin de laisser la place à un autre	
Estimé 8	Estimé 2	Estimé 3	Estimé 5	Estimé 2	Estimé 3
102   Compte-rendus de...	103   Dépôt de présent...	104   Lecture compte-r...	105   Dépôt de photos	106   Fil RSS	
En tant que Organisateur, Je peux faire le compte-rendu des éléments du programme d'une	En tant que Animateur, Je peux déposer ma présentation	En tant que Visiteur, Je peux accéder aux compte-rendus des conférences passées	En tant que Participant, Je peux déposer des photos d'une conférence	En tant que Visiteur, Je peux m'inscrire au fil RSS afin de être au courant des événements	
Estimé 8	Estimé 3	Estimé 5	Estimé 13	Estimé 3	
88   Inscriptions	91   News	89   Communauté	93   Multimédia	92   Programme	
Écrire des news pour la communauté			Podcasts, vidéos	Préparer le programme de façon interactive	
Accepté	Accepté	Accepté	Accepté	Accepté	

Figure 17.16 – Le *backlog* de produit initial pour Omega

### Création de sprints dans le plan de release

Le plan de *release* montre la vie du produit à moyen terme. On y trouve les *sprints* qui composent la *release*. Le plan repose sur l'association des *stories* du *backlog* aux différents *sprints*.

Maintenant qu'elle connaît mieux ce qu'il y a à faire et comment le faire, l'équipe Omega décide d'une durée des *sprints*. Ce sera trois semaines.

Le Product Owner enregistre cette durée de 21 jours dans les caractéristiques de la *release*. Les *sprints* sont générés automatiquement, dans le bloc de temps que constitue la *release*. Ils sont visibles dans la vue Roadmap (figure 17.17).

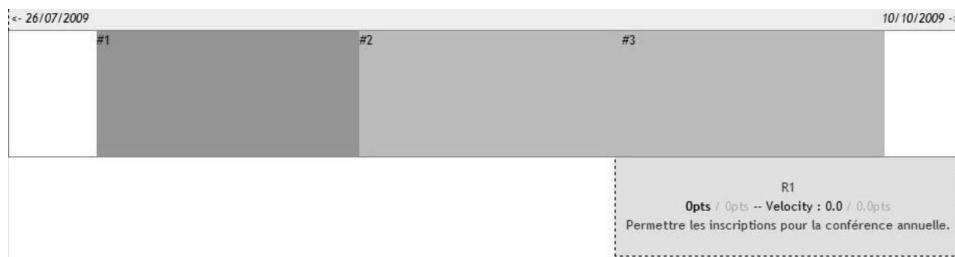
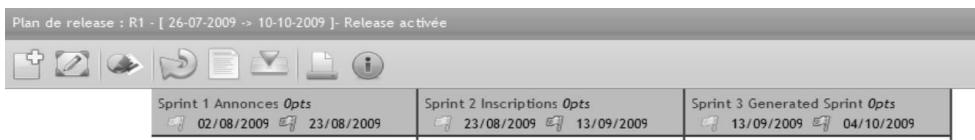


Figure 17.17 – Trois *sprints* créés dans la *release* R1

L'équipe définit collectivement le but des deux premiers *sprints* : ils vont porter sur les annonces et les inscriptions.



**Figure 17.18** — Le but des *sprints* défini dans le plan de *release*

### Planification de release

Les *sprints* de la *release* étant créés et le *backlog* estimé et priorisé, la planification peut s'effectuer, en tenant compte de la vélocité. Avec IceScrum la planification de *release* peut se faire de façon manuelle ou automatique. Au démarrage, lorsque la vélocité n'a pas encore été mesurée, il est préférable de procéder à la planification manuelle.

La planification de *release* consiste à associer des *stories* du *backlog* à des *sprints* de la *release*. Pour cela, il faut donc que le *backlog* contienne des *stories* estimées : en effet pour planifier, il faut avoir estimé et seules les *stories* estimées seront candidates à être planifiées. Il faut aussi avoir créé des *sprints*. Il sera possible plus tard de changer d'avis (figure 17.19), en dissociant toutes les *stories*, ou uniquement celles d'un *sprint* ou des *stories* individuelles ou en déplaçant une *story* d'un *sprint* à un autre.

Pour Omega la première planification de *release* est faite de façon manuelle. L'association se fait avec le plan de *release* ouvert et le *backlog* en vue réduite (*widget*). Les *stories* sont déplacées sur le *sprint* dans lequel on envisage de les réaliser.

### Lancement de la release

IceScrum fournit une assistance facilitant le lancement de la *release*, en montrant avec différentes vues les différents aspects du projet (figure 17.20).

L'équipe Omega passe en revue les éléments dont elle dispose avant de lancer les *sprints* :

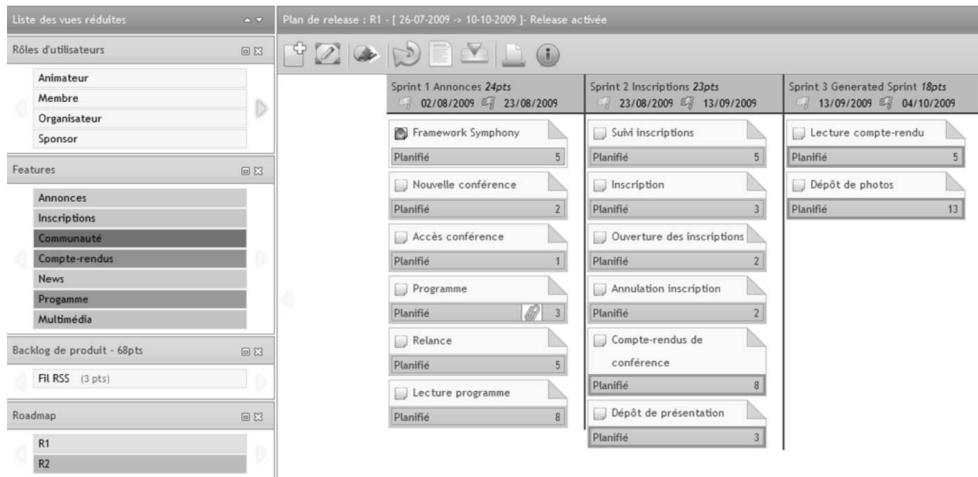
- la composition de l'équipe, avec les rôles de ScrumMaster et Product Owner définis, et les Stakeholders (partie prenantes),
- la roadmap montrant les deux *releases* prévues,
- le plan de la première *release* avec les *sprints* et les *stories* associées,
- les *features*,
- le *backlog* de produit.

Avant de « *sprint*er », l'équipe Omega se met d'accord, collectivement, sur ces éléments.

Dans IceScrum, les *releases*, comme les *sprints*, peuvent être créées à l'avance, ensuite pour lancer la *release* il faut l'activer, puis une fois finie la clore. Pour faciliter l'utilisation au démarrage, la première *release* est automatiquement activée.



**Figure 17.19** – Association de la story Lecture compte-rendu au sprint 3 par glisser-déposer



**Figure 17.20** – Vue sur Omega avec le plan de release en fenêtre principale et les vues réduites

### 17.2.3 Déroulement des sprints

#### Planification de sprint

À partir de la vue Roadmap ou du plan de *release*, on accède au premier *sprint*. Les stories planifiées (planification de *release*) apparaissent dans la colonne de gauche (figure 17.21).

Story	En attente	En cours
108   *Sprint 1*		
Planifié		
107   Framework Symphony		
Valider l'utilisation du framework		
Planifié 5		
94   Nouvelle conférence		
En tant que Organisateur, Je peux annoncer une nouvelle conférence Afin de informer la communauté		
Planifié 2		
95   Accès conférence		
En tant que Organisateur, Je peux indiquer le lieu et fournir		
Planifié		

Figure 17.21 – Plan de *sprint* avec les stories associées sur la gauche

La vue *sprint* reprend le principe du tableau des tâches avec des Post-it.

L'équipe Omega organise la réunion de planification du *sprint*, au cours de laquelle tout le monde identifie des tâches. Il est possible de créer des tâches « *storyless* », associées à la pseudo story du *sprint*.

Story	En attente
108   *Sprint 1*	
Planifié	
107   Framework Symphony	
Valider l'utilisation du framework	
Planifié 5	
94   Nouvelle conférence	
En tant que Organisateur, Je peux annoncer une nouvelle conférence Afin de informer la communauté	
Planifié 2	
95   Accès conférence	
En tant que Organisateur, Je peux indiquer le lieu et fournir	
Planifié	
Doc d'architecture	
ToDo:	
a a	
Déployer sur le ser...	
ToDo: 2	

Figure 17.22 – La tâche « doc d'architecture » n'est pas associée à une story mais au *sprint*

Une fois que les tâches ont été créées, leur reste à faire et leur réalisateur éventuellement définis, il reste à activer le *sprint*, ce qui enregistre l'engagement de l'équipe en fin de réunion.

### Tableau des tâches virtuel

Au cours du *sprint*, les tâches sont actualisées avec le reste à faire mis à jour et d'autres tâches peuvent être créées. Les membres de l'équipe Omega mettent à jour le tableau des tâches virtuel à distance, en déplaçant les tâches dans les colonnes En cours ou fini (figure 17.23).

Backlog de sprint #1 - [ R1 ] - [ 02-08-2009 -> 23-08-2009 ] - Sprint activé			
Story	En attente	En cours	Fini
108   *Sprint 1*	Doc d'architecture ToDo: a a	Déployer sur le ser... ToDo: 2	
107   Framework Symphony	doc ToDo: 2	proto ToDo: 12	étude ToDo: 0
Valider l'utilisation du framework			
En cours 5			
94   Nouvelle conférence	écrire tests d'acce... ToDo:	faire un diagramme ... ToDo: 2	
En tant que Organisateur, Je peux annoncer une nouvelle conférence Afin de informer la communauté	coder le formulaire ToDo: 3		
En cours 2	code et TU couche s... ToDo: 8		

**Figure 17.23** — Le tableau des tâches virtuel du *sprint* 1

Un développeur accède aux tâches qu'il a prises et peut voir les tâches libres pour en choisir une qu'il va prendre. S'il travaille sous Eclipse, il peut accéder à ses tâches avec le connecteur Mylyn, à partir de son environnement de développement.

Le *burndown chart* de *sprint*, actualisé tous les jours, est accessible à tous.

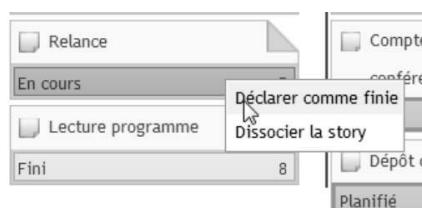
### Obstacles (impondérables)

Un obstacle qui ralentit le travail peut être consigné dans la vue Impondérable. Chacun peut en ajouter et le ScrumMaster a pour mission de les éliminer.

### Revue de *sprint*

Le Product Owner a pour responsabilité de vérifier qu'une *story* est finie. Si c'est le cas, il la déclare finie (figure 17.24), sinon il peut la dissocier ou la glisser dans un autre *sprint*.

Une fois toutes les *stories* contrôlées, à la fin de la revue de *sprint*, le *sprint* est clos et IceScrum calcule la vitesse.



**Figure 17.24** – La story Relance est déclarée finie

### Sprints suivants

Une fois le premier *sprint* clos, le deuxième est planifié, comme le premier, lors de la réunion de planification et activé en fin de réunion, avec l'engagement de l'équipe. De retour sur le plan de *release*, on a un aperçu de ce qui a été fini dans le *sprint 1* et de ce qui est à faire dans le *sprint 2* (figure 17.25).

Sprint 1 Annonces 24.0pts		Sprint 2 Inscriptions 23pts	
		02/08/2009	23/08/2009
<input checked="" type="checkbox"/> Framework Symphony	Fini	5	<input checked="" type="checkbox"/> Ouverture des inscriptions
<input checked="" type="checkbox"/> Nouvelle conférence	Fini	2	<input checked="" type="checkbox"/> En cours
<input checked="" type="checkbox"/> Accès conférence	Fini	1	<input checked="" type="checkbox"/> Inscription
<input checked="" type="checkbox"/> Programme	Fini	3	<input checked="" type="checkbox"/> En cours
<input checked="" type="checkbox"/> Relance	Fini	5	<input checked="" type="checkbox"/> Suivi inscriptions
<input checked="" type="checkbox"/> Lecture programme	Fini	8	<input checked="" type="checkbox"/> Annulation inscription
			<input checked="" type="checkbox"/> En cours
			<input checked="" type="checkbox"/> Compte-rendus de conférence
			<input checked="" type="checkbox"/> En cours
			<input checked="" type="checkbox"/> Dépôt de présentation
			<input checked="" type="checkbox"/> En cours

**Figure 17.25** – Le *sprint 1* est fini, le *sprint 2* en cours

### 17.2.4 Les tests d'acceptation

#### Description des tests

La vue Test permet de définir les tests d'acceptation des stories. Un cas de test est associé à une story, qui en possède en général plusieurs. Un cas de test peut être décrit de façon informelle, comme une condition de satisfaction, ou de manière plus précise.

En plus de permettre l'expression d'une condition informelle, IceScrum propose une formalisation (figure 17.26) de type BDD (*Behavior Driven Development*).

Activer le template de Test	
<input checked="" type="checkbox"/>	
Spécifications de test	
Etant donné	Monsieur Cherbonneau, l'amphi Concorde à 200 places et 200 personnes déjà inscrites
Quand	il s'inscrit à la conférence de septembre
Alors	son inscription est refusée et il reçoit un message qui lui dit que c'est complet

**Figure 17.26** — Un *story* test à la façon BDD

### Passage des tests

Pour qu'une story soit considérée comme finie, il faut au minimum que ses tests passent avec succès. IceScrum permet de noter le résultat des tests d'acceptation (passés en dehors de l'outil). Les cas de test sont glissés dans la colonne correspondant au résultat du test.



**Figure 17.27** — Pour la *story* 98, le test *Inscription acceptée* est un succès et celui *Refus salle complète* est un échec

### 17.2.5 Mesures et indicateurs

L'outil fait automatiquement la collecte des mesures et permet de visualiser les indicateurs présentés dans le chapitre 15 *Estimations, mesures et indicateurs (burndown, burnup, vélocité, capacité, flot cumulé, test, parking lot)*.



**Figure 17.28** — IceScrum, vous en prendrez bien un cornet ?

### En résumé

Une équipe Scrum utilise des outils simples, adaptés à son contexte. Obligatoire pour les équipes distribuées, un outil devient vite nécessaire pour gérer le *backlog* de produit.

Un outil dédié à Scrum, comme IceScrum, présente l'avantage d'apporter une aide méthodologique et de faciliter la production des rapports.



# 18

## La transition à Scrum

La première fois que j'ai accompagné une grande entreprise dans la transition à Scrum, on m'avait bien recommandé de ne pas citer Scrum ni d'utiliser son vocabulaire. Le *backlog* s'appelait le référentiel des exigences et les *sprints* étaient des itérations. C'était avant que Scrum ne soit à la mode et il ne fallait pas heurter les thuriféraires de la méthodologie officielle. Maintenant, avec la ScrumMania, les transitions se font avec plus de publicité.

J'ai rencontré d'autres situations pour le passage à Scrum : avec un projet pilote, sur un projet en pleine crise, à l'initiative de la direction pour tous les projets...

Dans tous les cas, adopter Scrum implique pour une organisation un changement dans la façon de travailler. Nous avons vu comment mettre Scrum en œuvre pour le développement d'**un** produit avec **une** équipe. Mais une équipe n'est jamais indépendante du reste de l'organisation et, pour élargir l'usage de Scrum à plusieurs équipes et plusieurs produits, il faut organiser la conduite du changement vers plus d'agilité.

C'est sur cette transition d'une organisation à Scrum que porte ce chapitre. Parfois elle est relativement facile, souvent elle très difficile. Pour certaines organisations, on pourrait parler de mutation, comme on parle de mutation industrielle : un changement économique et social brusque et spectaculaire, qui entraîne une modification profonde des structures.

### 18.1 LE PROCESSUS DE TRANSITION

Il existe de nombreuses variations sur la façon de faire la transition :

- **Selon la rapidité de mise en œuvre** : très vite pour toute l'organisation (*big bang*) ou progressivement en commençant avec un projet pilote.

- **Selon le contenu** : toutes les pratiques dès le début, ou des pratiques introduites graduellement.
- **Selon l'initiateur** : Scrum lancé par la hiérarchie (*top down*), ou l'initiative venant de l'équipe (*bottom up*).

### 18.1.1 Avec qui faire la transition ?

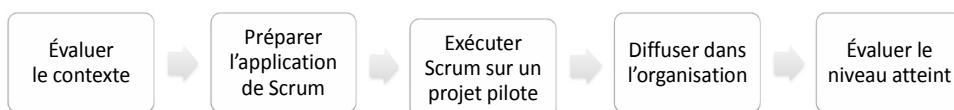
Si une équipe peut être à l'initiative du passage à Scrum, une transition plus massive passe par l'implication de la direction ; dans le cas d'organisation de taille importante, elle doit être gérée comme un véritable projet.

Quand la transition est un projet, il convient de constituer une équipe pour la mettre en œuvre. Il ne s'agit pas d'avoir un groupe de personnes à plein temps comme sur les projets de développement, mais il leur faut néanmoins du temps clairement alloué à la transition. La constitution de l'équipe dépend de la structure et de la taille de l'organisation. On peut y retrouver, par exemple :

- Le PDG (ou un dirigeant), qui est potentiellement bien placé pour tenir l'équivalent du rôle de ScrumMaster dans l'équipe de transition (un Product Owner n'est pas nécessaire).
- Des experts méthodes ou processus.
- Le ScrumMaster du projet pilote sur lequel Scrum sera appliqué en premier.
- Un ou plusieurs consultants extérieurs, experts dans la transition à Scrum.

### 18.1.2 Cycle de transition

Le cycle typique d'une transition dans une grande organisation passe par les étapes suivantes présentées figure 18.1.



**Figure 18.1** – Les étapes de la transition

Une approche cohérente est que le projet de transition à Scrum se fasse aussi en appliquant Scrum. C'est pourquoi le processus de transition est itératif, pendant l'étape d'exécution sur un projet, et basé sur le *feedback*. Dans le même esprit, l'équipe impliquée dans ce projet de transition utilise les artefacts Scrum : elle dispose d'un *backlog*, fait un plan à moyen terme et gère une liste d'obstacles.

### 18.1.3 Backlog d'amélioration des pratiques

Le *backlog* d'amélioration des pratiques (ou *backlog* de transition) est la liste des travaux à faire pour la transition. Il est géré comme le *backlog* de produit, avec des priorités qui définissent l'ordre dans lequel seront appliquées les pratiques. On y trouve des pratiques ou des façons d'améliorer certaines pratiques, portant sur des outils par exemple.

Le plan d'adoption du projet de transition est équivalent au plan de *release* Scrum, on y trouve le contenu du *backlog* projeté sur les *sprints* du projet de transition.

### 18.1.4 Obstacles d'organisation

La force de Scrum est d'exposer les obstacles qui ne manqueront pas d'arriver lors de la transition et qui vont freiner ou arrêter le déploiement de Scrum. Les obstacles identifiés peuvent être de nature différente :

- Sur les pratiques de Scrum.
- Venant de personnes.
- Venant de l'organisation et de son type de gouvernance.
- Sur les pratiques complémentaires de Scrum, notamment les pratiques d'ingénierie.

L'équipe de transition est chargée d'éliminer ces obstacles par des actions au niveau de l'organisation.

## 18.2 ÉTAPES DU PROCESSUS DE TRANSITION

### 18.2.1 Évaluer le contexte

Le but de cette étape est de connaître la situation de l'organisation au moment où s'effectue la transition. Qu'est-ce qui la pousse à passer à Scrum ? Quel est l'environnement de l'organisation, de différents points de vue ? Quel est le contexte des projets développés ?

C'est l'équipe de transition qui évalue le contexte, il faut donc la constituer en premier.

#### Définir la raison du passage à Scrum

Il faut avoir une idée claire de la raison du changement : passer à Scrum parce que c'est à la mode n'est pas une justification suffisante.

Pour commencer, il est utile d'établir une liste des problèmes auxquels est confrontée l'organisation. Cela peut être fait par une sorte de rétrospective sur les derniers projets réalisés. Les participants à ces projets peuvent être conviés pour une séance de réflexion collective dont le but est d'identifier cette liste de points à améliorer. Une

autre possibilité est de demander à un expert externe d'effectuer des interviews des intervenants majeurs dans les développements.

**Exemples de problèmes remontés** : on ne tient jamais les délais, il y a une pression terrible en fin de projet, la qualité est déplorable parce que les tests sont négligés...

La mise en évidence des problèmes aidera à motiver les personnes impliquées dans le changement, puisqu'elles comprendront ce que la transition à Scrum cherche à résoudre.

### *Évaluer le contexte de l'organisation et des projets*

L'organisation est évaluée selon ses conditions d'environnement : culture, niveau d'innovation, domaine métier et ses caractéristiques, niveau de maturité dans ses pratiques d'ingénierie.

Le contexte de quelques projets typiques est évalué en utilisant les dix attributs significatifs<sup>1</sup> (ou d'autres que l'équipe de transition aura sélectionnés) : taille, criticité, modèle économique, stabilité de l'architecture, dispersion de l'équipe, mode de gouvernance, capacité de l'équipe, âge du logiciel, taux de changements et mode de déploiement.

## **18.2.2 Préparer l'application de Scrum**

### *Choisir le projet pilote*

L'étude du contexte de quelques projets facilite le choix de celui qui va être le premier à mettre Scrum en application. C'est à l'équipe de transition de décider, avec les résultats de l'étape « *Évaluer le contexte* », et en s'appuyant sur des critères complémentaires :

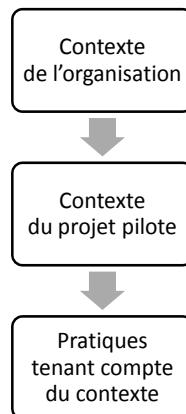
- Il est plus facile de mettre en œuvre Scrum au début du développement d'un nouveau produit qu'en plein milieu d'un projet, cela évite la complexité due à la prise en compte de l'existant, en termes d'outils et d'artefacts notamment.
- Le projet pilote doit être significatif, ce n'est pas un projet jouet, mais il ne doit pas être trop critique non plus avec les risques liés au changement.

Il est possible de lancer simultanément plusieurs projets pilote, mais évidemment cela demandera plus de disponibilité de la part de l'équipe de transition. C'est la distinction, faite par le Lean, entre le Kaizen et le Kaikaku. Le Kaizen correspond à l'amélioration continue et progressive et le Kaikaku à la transition radicale et rapide. Pour certaines organisations, le Kaikaku est la meilleure solution, mais il demande une implication considérable de la direction et l'appel massif à des consultants externes.

1. Voir le chapitre 12, *Adapter Scrum au contexte*, pour le détail de ces attributs.

### *Adapter les pratiques au contexte*

À partir de l'évaluation du contexte, il s'agit de définir précisément la façon dont les pratiques vont être mises en œuvre en tenant compte des contraintes du projet pilote.



**Figure 18.2** – Du contexte aux pratiques

Une façon systématique de procéder est de prendre les pratiques Scrum une à une et de déterminer, en équipe, la façon concrète de la mettre en œuvre. Il convient aussi d'y ajouter les pratiques complémentaires nécessaires<sup>1</sup>.

En procédant de la sorte, l'équipe élaboré la liste de tous les travaux à faire pour que les pratiques soient mises en œuvre sur le projet pilote : formation, logistique, outils, appel à des experts extérieurs...

Le résultat est la constitution du premier *backlog* d'amélioration des pratiques, qui, comme le *backlog* de produit, est priorisé.

### *Formation initiale*

Le premier besoin en formation concerne l'organisation : il est important qu'elle connaisse suffisamment Scrum et les méthodes agiles pour être convaincue qu'ils peuvent s'appliquer dans son contexte et avoir des pistes sur la façon de faire la transition. Une formation d'une journée, allant à l'essentiel, permet d'apporter ces réponses. Elle s'adresse à toutes les personnes qui pourront être en contact avec celles pratiquant Scrum.

Une fois que la décision est prise de démarrer un projet avec Scrum, l'équipe du projet pilote doit être formée à l'application concrète de Scrum. Le ScrumMaster et le Product Owner, qui font partie de l'équipe, y participent. Des travaux pratiques de mise en situation sont nécessaires, portant sur le projet. Ce ne serait pas une bonne idée que de former uniquement le ScrumMaster : toute l'équipe a besoin de la formation.

1. Voir le chapitre 13 *Adapter Scrum au contexte*

Selon le niveau de l'équipe en pratiques techniques (intégration continue, pilotage par les tests...), des formations à ces technologies peuvent être nécessaires avant de commencer le projet pilote.

En résumé :

- pour les intervenants qui ne participent pas directement à un projet, une formation d'une journée de sensibilisation qui va à l'essentiel ;
- pour tous les membres d'une équipe qui commencent un projet, une formation pratique de mise en application de Scrum en trois jours, à compléter par l'apprentissage de pratiques techniques si c'est nécessaire ;
- pour le ScrumMaster ou le Product Owner, participation à la formation de trois jours, complétée éventuellement par l'assistance d'un expert Scrum, au moins sur les premiers *sprints* du projet pilote.

### **Constitution de l'équipe pilote**

Pour le choix du ScrumMaster et du Product Owner, des critères ont été présentés dans des chapitres précédents ; pour les membres de l'équipe pilote voici des pistes :

- Inclure des personnes qui adhèrent à la culture « agile » et sont ouverts au changement.
- Inclure des volontaires qui ont envie de faire partie de l'équipe.
- Construire l'équipe avec des spécialistes généralistes (quelqu'un avec des spécialités techniques qui cherche activement à acquérir de nouvelles compétences dans sa spécialité aussi bien que dans de nouveaux domaines).
- Inclure des experts pour le travail spécialisé nécessaire à court terme, notamment pour l'architecture et l'ergonomie.

### **18.2.3 Exécuter Scrum sur un projet pilote**

Une fois le projet pilote démarré, la transition se déroule avec les deux projets actifs simultanément :

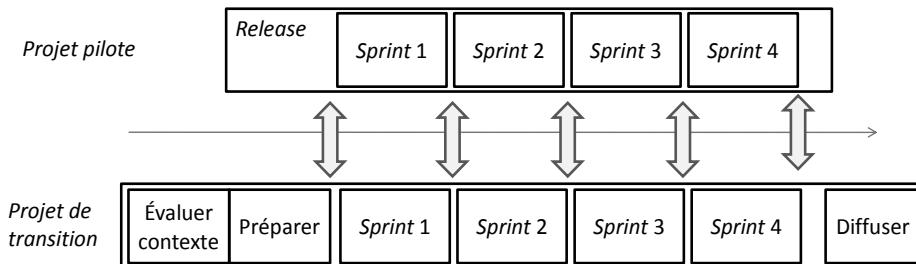
- Le projet de transition.
- Le projet pilote.

Les deux se synchronisent régulièrement : un projet pilote dure environ trois mois avec quatre à six *sprints* et le projet de transition suit le même rythme.

Le *feedback* venant du projet pilote alimente les réflexions de l'équipe de transition. L'équipe du projet pilote dispose du support de l'équipe de transition pendant le *sprint*. L'assistance peut se concrétiser par une présence à quelques réunions, des réponses aux questions sur Scrum, du *coaching* pour le Product Owner, des formations courtes à des techniques qui vont être utiles pour l'équipe...

La rétrospective, à la fin de chaque *sprint* du projet pilote, est faite en présence de l'équipe de transition. Elle se poursuit ensuite par une réunion de l'équipe de transition qui fait le point sur la transition à Scrum. Le *backlog* d'amélioration est mis à jour, en

fonction des remontées de la rétrospective. Les actions pour le prochain *sprint* sont décidées.



**Figure 18.3** – Synchronisation entre projet pilote et projet de transition

#### 18.2.4 Diffuser dans l'organisation

À la fin de la première *release* du projet pilote, si c'est un succès, l'objectif est d'étendre l'usage de Scrum et ses bénéfices à un sous-ensemble significatif de l'organisation cible.

##### Capitalisation

La liste des pratiques qui ont été appliquées et celles des obstacles rencontrés permet d'avoir un état des lieux avant une diffusion plus grande. L'équipe de transition a collecté ces informations au fur et à mesure du déroulement et peut organiser une rétrospective plus poussée à la fin du projet pilote pour compléter la capitalisation sur l'expérience.

##### Diffusion de la connaissance

Les résultats du projet pilote sont communiqués largement dans l'entreprise, pour commencer à créer une communauté et déclencher de nouvelles vocations dans les équipes.

Les participants au projet pilote sont un vecteur idéal pour diffuser la connaissance dans l'organisation (figure 18.4). Ils peuvent être dispatchés dans les nouveaux projets qui vont utiliser Scrum.

##### Scrum de scrums

La généralisation de Scrum va probablement entraîner son utilisation des projets de plus grande taille, nécessitant des ressources importantes, allant au-delà d'une équipe Scrum (limitée à une dizaine de personnes). L'usage de Scrum reste possible, en constituant plusieurs (sous-)équipes. La pratique Scrum permettant la collaboration d'équipes participant au développement d'un produit est connue sous le nom de « *scrum de scrums* ».



**Figure 18.4** — Le ScrumMaster d'un projet pilote heureux de montrer sa vélocité et son *burndown chart*

Les réunions du cérémonial sont appliquées de manière récursive. En ce qui concerne le scrum quotidien, un représentant de chaque équipe participe, après la réunion locale à son équipe, au scrum de scrums. La réunion permet d'exposer et résoudre les problèmes de synchronisation entre les équipes.

Le développement à grande échelle engendre, bien entendu, et pas seulement avec Scrum, de multiples questions d'organisation :

- Comment constituer les équipes ?
- Comment partager le travail entre les équipes ?
- Combien de Product Owners ?
- Les *sprints* de chaque équipe sont-ils synchronisés ?

Une réponse rapide est qu'il vaut mieux constituer des équipes selon un découpage fonctionnel (par *feature*), garder un seul *backlog* et un seul Product Owner et avoir des *sprints* de même durée et synchronisés. Mais chaque situation est différente et il y a bien d'autres questions qui se posent.

Le lecteur pourra trouver des compléments sur le sujet dans les écrits de Dean Leffingwell (*Scaling Agility*) ou dans le chapitre *Scaling Agile* de *Succeeding with Agile* de Mike Cohn.

### Formations complémentaires

Le démarrage de nouveaux projets est l'occasion d'organiser de nouvelles formations, tenant compte des résultats du pilote.

- Les équipes des nouveaux projets Scrum qui vont démarrer suivent la formation pratique de trois jours.
- C'est aussi le bon moment pour une formation spécifique, destinée aux futurs Product Owners et ScrumMasters, d'une durée de deux jours.
- La diffusion est complétée par une nouvelle formation de sensibilisation (un jour) pour les nouveaux membres de l'organisation qui seront en relation avec les nouvelles équipes Scrum.

#### 18.2.5 Évaluer le niveau atteint

Il s'agit de préparer une diffusion générale, une fois que l'utilisation est répandue.

Une entreprise qui effectue une transition a une idée des progrès effectués en effectuant la collecte de mesures pertinentes<sup>1</sup> et en évaluant le niveau (d'agilité) atteint ; cela sert de base pour une diffusion plus large.

Il convient de choisir un sous-ensemble significatif de mesures à faire pour chaque nouveau projet. Il est souhaitable d'ajouter aux mesures quantitatives des mesures qualitatives, comme la satisfaction des utilisateurs (et aussi celle des développeurs).

**Attention :** la plupart des mesures ne permettent pas de comparer plusieurs projets. Par exemple, la vitesse est intrinsèque d'une équipe et ne peut pas servir à évaluer son efficacité.

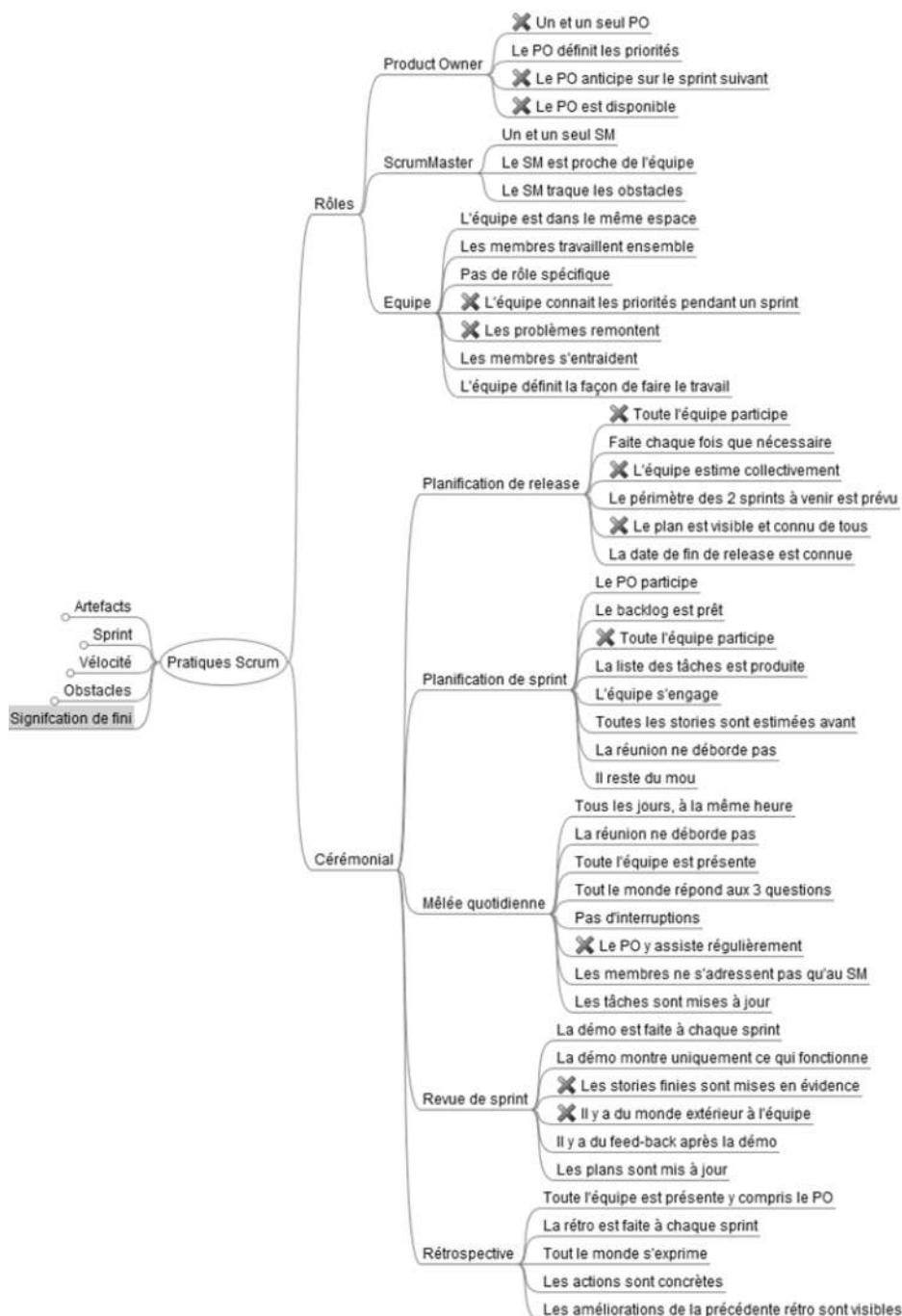
En plus des mesures sur le produit, la diffusion de Scrum doit s'accompagner de mesures sur le processus, c'est-à-dire sur la façon dont les équipes mettent en œuvre les pratiques.

Pour collecter ces mesures sur le processus, l'équipe de transition peut élaborer un questionnaire permettant d'évaluer le niveau d'application de la pratique. Pour chaque pratique, l'équipe se situe en répondant à quelques questions.

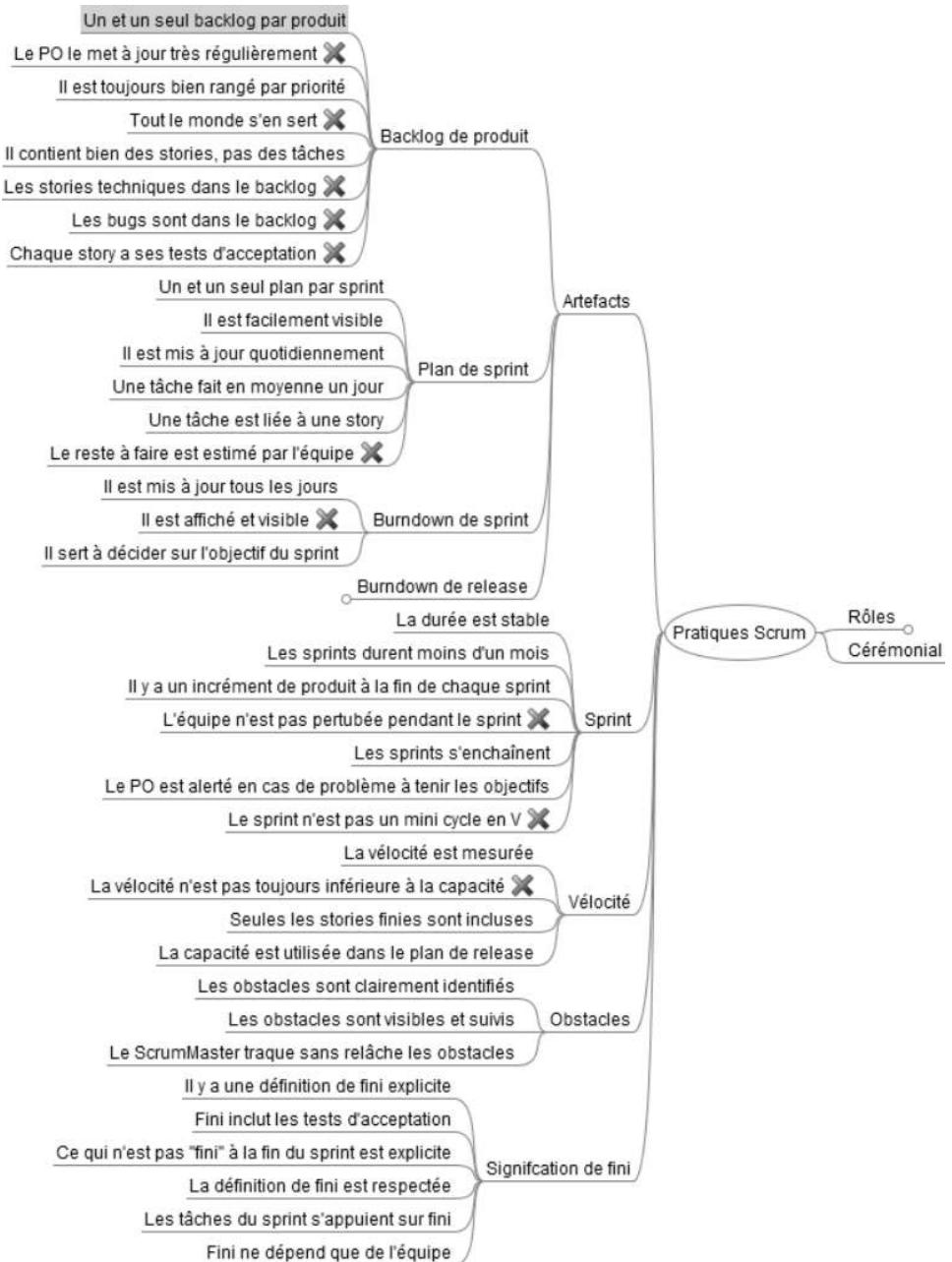
Les figures 18.5 et 18.6 présentent un exemple d'évaluation des pratiques Scrum, sous forme de carte heuristique.

---

1. La liste des mesures possibles est présentée dans le chapitre 15.



**Figure 18.5** — Première partie de la carte heuristique avec l'évaluation des pratiques sur les rôles et le cérémonial



**Figure 18.6** — Deuxième partie de la carte heuristique avec l'évaluation des pratiques sur les artefacts, les obstacles, la signification de fini, le *sprint* et la vitesse.

## 18.3 IMPACTS SUR L'ORGANISATION

La diffusion dans une organisation de grande taille a des impacts importants, parce que Scrum véhicule une vision qui est, la plupart du temps, radicalement différente sur la façon de s'organiser et de gérer les ressources humaines. En voici trois exemples.

### 18.3.1 L'évaluation individuelle est contre-productive

Esther Derby (dans l'article *Performance without Appraisal*<sup>1</sup>) pourfend l'évaluation individuelle au mérite. Elle cite notamment Deming, bien connu pour ses travaux sur la qualité (et pour sa roue<sup>2</sup>) : « *The idea of merit rating is alluring. The sound of the words captivates the imagination: pay for what you get ; get what you pay for ; motivate people to do their best, for their own good. The effect is exactly the opposite of what the words promise. Everyone propels himself forward, or tries to, for his own good, on his own life preserver. The organization is the loser. Merit rating rewards people who do well within the system. It does not reward attempts to improve the system* » W. Edwards Deming, *Out of the Crisis*.

Esther Derby propose de supprimer les entretiens annuels d'évaluation couramment pratiqués dans les entreprises et fournit des réponses intéressantes aux questions qui en découlent :

- comment déterminer le salaire de chacun,
- comment donner une promotion à quelqu'un ou au contraire « virer » une personne,
- comment les gens sauront qu'ils doivent s'améliorer.

Scrum, par ses mécanismes de régulation et la transparence apportée, contribue à privilégier une approche collective et à se passer des évaluations individuelles.

### 18.3.2 Pas de multitâches

Le multitâche pour une personne, c'est le fait de suspendre une tâche en cours alors qu'elle n'est pas encore finie pour passer à une autre, qualifiée de plus prioritaire. Dans nos organisations, le multitâche est un fait courant, parce que c'est souvent l'usage que :

- une personne travaille sur plusieurs projets en même temps (lors d'une formation, j'ai rencontré une personne qui travaillait sur six projets en parallèle).
- une personne travaille sur un nouveau développement mais aussi sur la maintenance du logiciel. Le bug urgent à corriger immédiatement qui ne va pas manquer d'arriver est le déclencheur du changement de contexte et cela va provoquer inéluctablement des soucis au projet de nouveau développement.

---

1. <http://www.scrumalliance.org/articles/50-performance-without-appraisal>

2. [http://fr.wikipedia.org/wiki/Roue\\_de\\_Deming](http://fr.wikipedia.org/wiki/Roue_de_Deming)

- un responsable surgisse en proclamant qu'une nouvelle tâche est devenue la priorité absolue.

Il faut à tout prix éviter le gaspillage d'énergie et de temps dû au multitâche. Les méthodes agiles donnent quelques bons conseils pour y arriver :

- ne pas affecter une personne à plusieurs projets,
- définir des priorités et les rendre visibles à travers le *backlog*,
- éviter de perturber une équipe pendant un *sprint*.

### 18.3.3 Spécialistes vs généralistes

Les équipes agiles sont plus efficaces avec des généralistes ou des spécialistes qui se généralisent.

Thierry Crouzet (qui a écrit un billet sur *Généralistes contre spécialistes*<sup>1</sup>) défend avec brio l'idée que les généralistes sont plus utiles que les spécialistes dans les périodes de changement.

Cette idée est défendue par le mouvement agile chaque fois que la constitution d'une équipe est évoquée. Par exemple, le tout petit nombre de rôles dans Scrum y pousse : pas d'architecte par exemple mais seulement des membres de l'équipe.

On pourrait comprendre qu'une équipe Scrum ne doit pas inclure de spécialistes. Il ne s'agit pas de cela. Dans mes équipes je me suis souvent appuyé sur des spécialistes pour avancer et il serait dommage de se priver de compétences pointues. L'idée est plutôt d'éviter l'hyper-spécialisation. Comme le fait remarquer Crouzet, il y a un risque d'enfermement du spécialiste dans une seule voie. Il y a aussi qu'une équipe de développement qui serait composée de spécialistes ne pourrait pas rester de taille réduite pour couvrir tous les domaines et que la communication y serait bien difficile.

### 18.3.4 Cohabitation avec d'autres processus

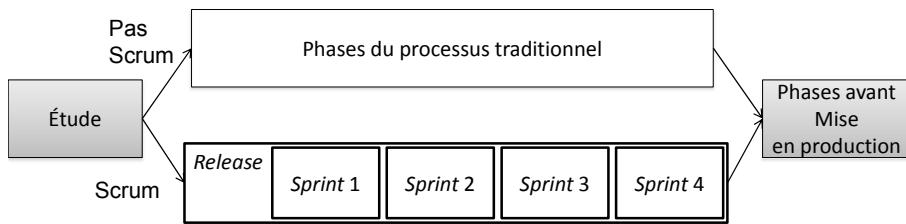
La transition à Scrum est généralement progressive, ce qui signifie qu'il y aura une période de cohabitation avec les processus en place. Des projets qui démarrent auront le choix entre plusieurs options, dont Scrum.

À quel moment se fait ce choix ? Pour qu'il soit pertinent, il convient de le faire après une phase d'étude (ou étude d'opportunité), pour déterminer, par exemple, son contexte avec les attributs présentés précédemment. Cela signifie que la phase d'étude est commune à tous les projets et qu'à la fin de cette phase, le choix du processus, Scrum ou pas, est effectué.

À l'autre bout du cycle de développement, les grandes organisations qui possèdent une infrastructure pour leur système d'information ont généralement des phases balisées avant la mise en production. Dans un premier temps, il est peu probable

1. <http://blog.tcrouzet.com/2008/05/26/generalistes-contre-specialistes/>

que les travaux effectués dans ces phases puissent être inclus dans les *sprints*. C'est pourquoi, après une *release* et ses *sprints*, le produit peut être amené à passer par ces phases traditionnelles.



**Figure 18.7** — Scrum, comme alternative dans le processus global d'une entreprise

## Résumé

Si lancer un projet avec Scrum est relativement facile, la transition d'une organisation est autrement délicate. L'introduction de Scrum se prépare en considérant le changement de processus comme un véritable projet.

# 19

## Scrum en France

Bien que le mot Scrum ait dans ses nombreuses racines le français escarmouche, il est indéniablement d'origine étrangère. La méthode Scrum vient des États-Unis, même si on peut y trouver des racines japonaises<sup>1</sup>, et en remontant plus loin des origines européennes avec l'autogestion. Le but de ce chapitre est de faire l'état de la diffusion de Scrum en France, en se demandant s'il y a des spécificités françaises.

### 19.1 SCRUM À LA FRANÇAISE

#### 19.1.1 Utilisateurs de Scrum

L'usage de Scrum en France est tardif, par rapport à sa diffusion dans le reste du monde. Les premières expériences datent de 2005 et l'essor n'a véritablement commencé que fin 2007. Depuis la croissance est fulgurante. Comme on ne dispose pas d'une mesure des utilisateurs, il faut l'estimer, exercice auquel je m'étais prêté en mars 2009, pour les équipes pratiquant Scrum en France :

- 2005 : moins de dix équipes.
- 2006 : moins de 50 équipes.
- 2007 : moins de 200 équipes.
- 2008 : moins de 800 équipes.
- 2009 : plus de 1 000 équipes.

J'ai présenté cette estimation lors du lancement du groupe des utilisateurs français de Scrum. En effet, ce groupe d'échanges sur Scrum a été lancé début 2009 ; il comptait

---

1. L'allusion au rugby vient d'un article japonais et Scrum s'appuie sur le Lean utilisé en production.

en septembre 2009 plus de 350 membres. Le « Scrum User Group » (SUG) français<sup>1</sup> est une association à but non lucratif qui a pour objet la promotion des pratiques de Scrum sur le territoire français en tant que composante fondamentale des méthodes agiles.

### 19.1.2 Retours d'expérience

On dispose maintenant de nombreux retours d'expérience sur des projets menés avec Scrum en France. Certains sont publiés dans la presse ou présentés à l'occasion de conférences.

Par exemple, à Toulouse, l'association SigmaT<sup>2</sup> a organisé douze conférences entre 2006 et 2009, avec à chaque fois au moins un retour d'expérience sur Scrum. Parmi ces sociétés, des petites et des grandes qui exercent dans différents domaines d'activité. Le bilan de ces utilisations a toujours été présenté comme très positif.

Il existe maintenant d'autres associations dans les régions, comme le CARA<sup>3</sup> pour Rhône-Alpes, ou au niveau national, par exemple le XP Day<sup>4</sup> et l'Agile Tour<sup>5</sup>, qui organisent régulièrement des manifestations sur l'agilité, dont le contenu des sessions porte régulièrement sur Scrum.

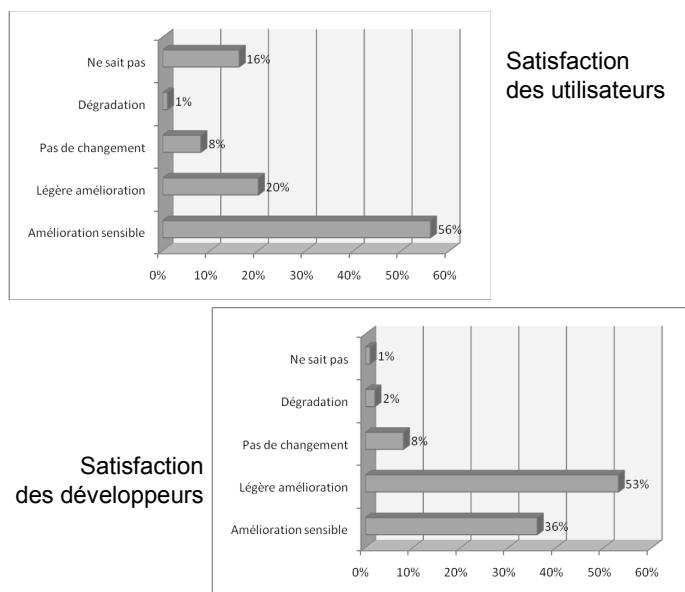
Pour évoquer les retours d'expérience, je peux m'appuyer aussi sur la cinquantaine de projets Scrum dans lesquels j'ai été impliqué, et sur l'enquête du SUG français publiée en juin 2009<sup>6</sup>.

Le SUG français a lancé une enquête au printemps 2009 sur les usages de Scrum (et des méthodes agiles) en France. Les résultats font apparaître un taux de satisfaction élevé, que ce soit de la part des utilisateurs ou des développeurs (figure 19.1).

### 19.1.3 Domaines

Scrum est utilisé dans tous les domaines du développement logiciel. L'enquête du SUG fait apparaître une grande diversité : les méthodes agiles ne sont pas utilisées que pour faire des sites web ! Des domaines *a priori* moins adaptés sont aussi touchés par la vague Scrum. Pour en citer quelques-uns dans lesquels je suis intervenu : administrations du secteur public, banques, studios de jeu vidéo, éditeurs, énergie, automobile, applications M to M (*Machine to Machine*), opérateurs téléphoniques, commande numérique, web, sociétés de service.

- 
1. [www.frenchsug.org](http://www.frenchsug.org)
  2. <http://www.sigmat.fr>
  3. <http://clubagile.org/>
  4. <http://xpday.fr/>
  5. <http://www.agiletour.org/>
  6. Elle est disponible sur le site de l'association du SUG France



**Figure 19.1** — Extraits de l'enquête du SUG en France

Scrum est aussi utilisé dans le domaine de l'industrie. Par exemple, Scrum est utilisé pour un projet de système industriel de contrôle commande de stations électriques chez Areva T&D. Camille Bloch, à l'initiative de l'introduction de Scrum, met en avant l'amélioration de la communication dans l'équipe : « ...Nous nous sommes rendu compte que notre façon de travailler n'était pas optimale. Dans une démarche d'amélioration, nous avons donc exploré avec une petite équipe l'utilisation de Scrum, couplé à des bonnes pratiques d'ingénierie d'XP. Après quelques mois d'expérimentation, nous avons fait appel à Claude pour recadrer notre expérience. Depuis lors, nous évoluons sans cesse pour nous améliorer. Nous avons rencontré de nombreux problèmes, que nous avons résolus ensemble. Le plus grand apport de Scrum a été de nous permettre de développer un réel esprit d'équipe, et de travailler plus sereinement. Notre travail devient plus efficace et le logiciel produit en est de meilleure qualité. Il répond mieux aux besoins du client... »

Scrum est indifférent aux pratiques d'ingénierie, ce qui le rend potentiellement utilisable en dehors du développement de logiciel. Même si cela reste marginal, ces utilisations se multiplient. En lisière du développement, Scrum est utilisé pour gérer des équipes qui font des projets d'infrastructure. J'ai participé à des projets Scrum portant sur le paramétrage de progiciels, à d'autres pour modéliser et documenter des processus ou pour faire de la validation.

En dehors du logiciel, on m'a signalé des usages de Scrum pour de l'éco-facilitation et il semble même que le BTP s'y intéresse.

Les usages personnels de Scrum sont nombreux : les personnes ayant vu l'intérêt de l'approche sur des projets essaient de l'appliquer pour améliorer leur efficacité personnelle. J'ai, bien évidemment, utilisé Scrum pour le gros projet personnel qu'a constitué l'écriture de ce livre.

### 19.1.4 Des particularités locales ?

Existe-t-il une spécificité française rendant Scrum adapté ou non à notre culture ?

En France comme ailleurs, on trouve une grande variété d'organisations et donc de contextes. Par rapport à l'utilisation de Scrum, nous avons vu qu'on pouvait définir des attributs permettant de situer une organisation ou d'un projet. Sont-ils différents en France ?

Les caractéristiques des projets n'ont pas de raison d'être différentes. En France comme ailleurs, il y a de grands et de petits projets, des développements nouveaux et des reprises de logiciel existant.

Sans prétendre généraliser, les attributs suivants peuvent mettre en évidence d'éventuelles spécificités françaises :

- Le type de gouvernance, pour les organisations de type MOA/MOE.
- Le modèle économique, avec les contrats au forfait.
- La culture des équipes de développement.

Les deux premiers, typiques de quelques grandes entreprises ou administrations, peuvent freiner la diffusion de Scrum, le dernier la renforcer.

## 19.2 DES FREINS À LA DIFFUSION ?

### 19.2.1 MOA et MOE ne sont pas agiles

Dans la plupart des grandes organisations françaises, il existe une division entre la Maîtrise d'ouvrage (MOA), qui représente des utilisateurs internes à l'entreprise, et la Maîtrise d'œuvre (MOE), dans laquelle on trouve la Direction des systèmes d'information (DSI). La plupart des sociétés du CAC40 et les administrations sont dans ce schéma.

Après une réunion du Cigref à laquelle il vient d'assister, Louis Naugès<sup>1</sup> note, à propos du rapport nommé *Dynamique de création de valeur par les Systèmes d'Information* : « *Il signe en tout cas la mort définitive d'une invention franco-française qui a fait beaucoup de dégâts : la séparation maîtrise d'ouvrage, maîtrise d'œuvre. Comment imaginer une seconde une collaboration efficace entre informaticiens et métiers s'ils ne travaillent pas ensemble, en permanence ?* »

Effectivement, cette organisation de type MOA/MOE tend à créer une séparation très nette entre deux groupes. Cela contribue aux problèmes suivants :

- de la communication basée sur des documents au détriment de la communication orale,
- des spécifications trop détaillées dès le début du projet,
- des documents redondants entre les deux entités,

1. L'inventeur du mot bureaucratique : <http://nauges.typepad.com>

- des documents qui mélangent le quoi et le comment,
- du temps perdu dans un processus de validation de documentation,
- des tests d'acceptation passés tardivement.

Aussi quand on me demande après une présentation Scrum si c'est la MOA ou la MOE qui doit jouer le rôle de Product Owner, je réponds qu'il faut d'abord se débarrasser de ces notions et de ce qu'elles impliquent avant de passer à Scrum. Il convient de faire tomber les murs et de créer une seule équipe qui contribue au même objectif (qui n'est pas d'écrire de la documentation). Le Product Owner fait partie de cette équipe avec ceux qui analysent, conçoivent, développent, testent, rédigent...

### ***Choix du Product Owner***

La MOA regroupant les personnes qui sont du côté « métier », il paraît logique qu'un Product Owner soit choisi en son sein. Quelqu'un qui a été analyste métier (*Business Analyst*), en assistance à MOA, est un bon candidat pour ce rôle. Mais c'est généralement le CPU (chef de projet utilisateurs) qui est choisi.

Il est impératif d'avoir un Product Owner qui joue vraiment son rôle, c'est une condition *sine qua non* pour la réussite des projets, cela ne se négocie pas. Il n'y a pas vraiment d'alternative : la réussite des projets, qu'on soit agile ou pas d'ailleurs, passe par une disponibilité importante du Product Owner. Certains en sont conscients et disent : *On ne peut pas appliquer les méthodes agiles chez nous, parce que nous ne pourrions pas impliquer suffisamment nos MOA.* »

Renoncer aux bienfaits de l'agilité plutôt que d'essayer d'impliquer plus la MOA, c'est prendre le problème à l'envers. Au-delà de l'agilité, la pratique qui consiste à impliquer les utilisateurs (à travers leur représentant) est considérée comme cruciale pour le succès des projets, depuis des années. Les études le disaient déjà dans les années quatre-vingt-dix.

Alors, une meilleure solution serait de supprimer la MOA et la MOE... en commençant par supprimer les murs qui les séparent.

### ***Supprimer les barrières entre MOA et MOE***

Nous avons vu qu'une bonne façon de renforcer l'esprit d'équipe est d'avoir, au niveau logistique, un espace de travail ouvert. Le Product Owner, même s'il n'y reste pas à plein-temps, sera encouragé à y venir régulièrement.

Une responsabilité habituelle de la MOA est le déroulement des tests de recette. Avec Scrum, il n'y a pas deux équipes, une de développement et l'autre de test, mais une seule équipe accueillant tous les participants au projet, y compris les testeurs, présents dans l'espace de travail.

### ***Scrum sur le terrain, le retour de la communication***

Les retours d'expérience dans ce type d'organisation indiquent tous comme premier résultat du passage à Scrum le retour de la communication entre utilisateurs (MOA) et développeurs (MOE). Pascal Renaut, à l'initiative de l'introduction de Scrum à

l'Électricité de Strasbourg, fait l'analogie avec un couple au bord du divorce qui se reforme :

« (avant Scrum) ...j'en ai entendu de la part de la maîtrise d'ouvrage : « Pas assez vite, pas assez bien ». La faute est bien entendu à partager entre MOA/MOE, comme dans tous les couples, et il est vrai que le traditionnel cycle en V ne peut pas répondre à toutes les attentes. ...

J'ai trouvé en Scrum le moyen de réconcilier ce couple en perdition. Rédaction de l'expression des besoins limitée au plus strict nécessaire (user stories), petits moments de convivialités en équipe pour les estimations et les priorisations, développements dans la foulée, livraisons régulières et on enchaîne comme cela de sprint en sprint.

Il est finalement moins risqué et moins coûteux de travailler MOA/MOE sur un projet commun (et non plus séparément projet MOA puis projet MOE) autour duquel on retrouve une équipe mobilisée, soudée et motivée comme jamais. »

## 19.2.2 Contrats au forfait, le mythe du périmètre fixé

Le malentendu entre contrat dit au forfait et agilité vient de l'idée que le périmètre fonctionnel semble être fixé à l'avance par le client. Mais c'est un leurre : sauf pour de petits projets où on peut se passer de *feedback* ou dans les cas particuliers de spécification formelle, l'expérience montre que c'est impossible. Dans la réalité, le périmètre, d'une part n'est pas strictement défini au début du projet, d'autre part évolue toujours.

Partant de ce constat et du postulat de base qui est l'acceptation du changement, quand on utilise Scrum on considère que le périmètre est la variable d'ajustement.

Il existe de nombreuses façons de rendre les contrats plus agiles. Pour en citer quelques-unes :

- **Décomposer l'appel d'offres en deux parties** : une première pour évaluer la capacité du fournisseur (ou des fournisseurs) et une deuxième avec un engagement sur sa vitesse.
- **Faire un contrat avec une enveloppe fixée et un périmètre évalué par morceaux**, de façon similaire à ce qui se pratique pour les TMA (notion d'unité d'œuvre pour la tierce maintenance applicative).
- **Faire un contrat de façon habituelle, mais en introduisant deux nouvelles clauses** : « changement gratuit » et « gagnant-gagnant ».

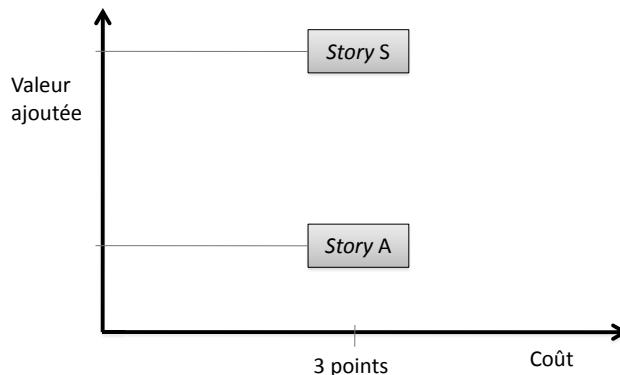
Dans tous les cas, la façon de faire les appels d'offres devrait être adaptée pour introduire une façon de procéder plus agile. Le *backlog* de produit est un élément essentiel de la démarche agile. Il peut, selon les caractéristiques du projet, être annexé à l'appel d'offres ou réalisé en commun entre client et fournisseur.

### La clause « changement gratuit »

Cette clause offre la possibilité au client de changer d'avis sans que cela ne remette en cause le contrat, en restant à prix constant. Le principe est le suivant : à chaque fin de sprint, le client peut changer d'avis sur le contenu du backlog (pour les stories qui n'ont pas encore été développées), à condition que ce qu'il ajoute soit contrebalancé par ce qu'il retire. Cela suppose que :

- les éléments du backlog soient estimés en points,
- le client et le fournisseur sont d'accord sur ces estimations,
- le Product Owner joue vraiment son rôle en participant activement au projet.

L'intérêt pour le client est qu'il peut avoir plus de valeur ajoutée (ou utilité) pour un prix équivalent.



**Figure 19.2** — Plus de valeur pour le même prix en ajoutant la story S et supprimant la story A

En fait, ce type de changement est déjà pratiqué dans la plupart des projets au forfait au cours de négociations entre client et fournisseur, mais en cachette puisque c'est contraire au principe du périmètre fixé à l'avance qui est la base juridique des contrats classiques.

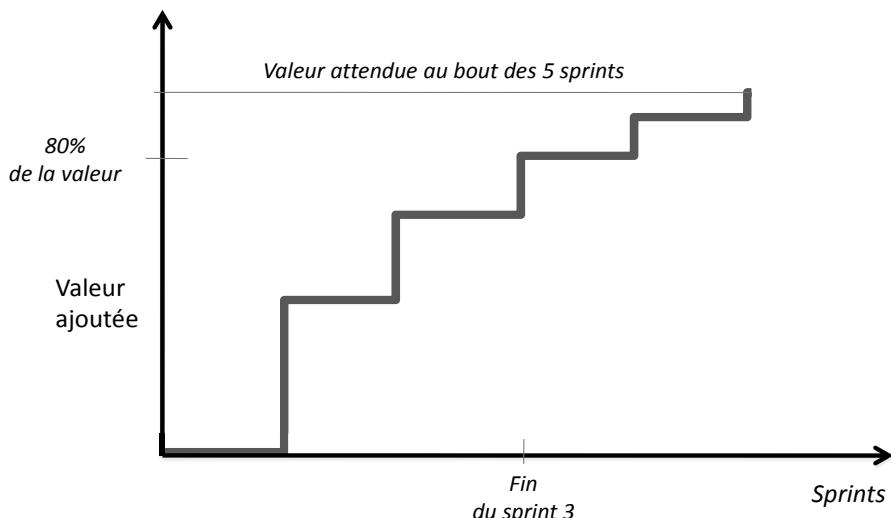
### La clause « gagnant-gagnant »

Cette clause permet au client d'arrêter le contrat à la fin de chaque sprint. Comme l'équipe livre une version à la fin de chaque sprint, le client peut faire du feedback et modifier le backlog (c'est la clause changement gratuit). Il peut aussi se rendre compte que le produit apporte suffisamment de valeur et décider de le déployer en mettant un terme au contrat sans attendre la fin prévue initialement.

Dans ce cas, le fournisseur est payé au prorata des travaux effectués (les éléments du backlog qui sont finis) plus un pourcentage (par exemple 20 %) de la différence entre le prix fixé au départ et la somme obtenue par le prorata.

Tout le monde s'y retrouve :

- le client parce qu'il a un produit qui lui apporte de la valeur et coûte moins cher que prévu (il a par exemple payé 60 % du prix pour un ensemble de stories qui représentent 80 % de son besoin en termes de valeur ajoutée),
- le fournisseur parce qu'il est payé (le pourcentage) alors qu'il ne consomme plus de ressources (cette somme complémentaire sert aussi à dédommager le prestataire qui doit réaffecter ses équipes dans des délais courts et non prévus initialement).



**Figure 19.3** — Le client a plus de 80 % de la valeur attendue à la fin du *sprint 3* pour 60 % du coût envisagé (3 *sprints* sur 5)

Cette clause a un sens parce que les priorités définissant l'ordre de réalisation dans les sprints sont basées sur la valeur. Le client peut ainsi mettre ce qui a le plus d'utilité dans les premières itérations et décider d'arrêter parce que ce qui reste à faire possède moins de valeur que cela ne coûte.

## 19.3 LE FRENCH FLAIR POUR SCRUM

La France a bien adopté le rugby importé d'Angleterre il y a plus de 100 ans et ça a donné le *french flair* qui, bien s'il se perde malheureusement un peu, est toujours la référence du jeu à la française. Le *french flair* est possible par la liberté qui est laissée au joueur pour prendre des initiatives en fonction de la situation plutôt que de suivre aveuglément les schémas de jeu définis à l'avance.

Dans une équipe Scrum, comme dans une équipe de rugby, la liberté de se rebeller doit être permise, voire favorisée. Le bon fonctionnement d'une équipe n'empêche pas

qu'une personne fasse, à un moment du projet, ce dont elle a envie et qui va finalement répondre au besoin de tous. C'est ce qu'on appelle l'intelligence situationnelle.

On peut aussi considérer que la transition vers un processus agile est une rébellion positive. C'est une remise en question des normes établies et de l'autorité hiérarchique ainsi qu'une volonté de faire sauter le vernis des hypocrisies, des traditions surannées et des préjugés tenaces.

Je pense que le développeur français, peut-être par sa culture de descendant de révolutionnaire, garde souvent cette liberté d'initiative.

C'est ce que me confirme Jamel Marzouki<sup>1</sup>, qui après avoir travaillé en France côtoie des équipes américaines et chinoises ; il vit à Chicago depuis plus de dix ans et se déplace souvent à Shangaï : « *Les Français, dans le domaine du logiciel, sont plus enclins à l'évolution et au changement qu'ailleurs ... Je pense que cela est dû à l'éducation : les étudiants reçoivent très tôt une introduction à la culture "Génie Logiciel" dans sa globalité, ne se réduisant pas uniquement aux langages de programmation.* »

De mon côté, j'ai eu moins de contacts avec des développeurs d'autres pays ; mais les quelques Allemands que j'ai vu travailler m'ont paru bien moins agiles que les Français.

À l'époque où j'étais développeur, les initiatives étaient encouragées et il y avait un réel plaisir à réaliser un projet en équipe. C'était il y a longtemps et depuis, il semble que la qualité de vie des informaticiens se soit dégradée en France, au moins dans les grandes structures.

Mon souhait est que Scrum contribue à remettre l'aspect humain au cœur des projets, tout en contribuant à renforcer la qualité des produits développés. C'est dans cet esprit que j'enseigne Scrum à mes étudiants en génie logiciel et, les voyant l'appliquer, je n'ai aucun doute sur l'adéquation des méthodes agiles avec le caractère français

---

1. Au cours d'une conversation personnelle.

# Références bibliographiques

## Webographie

### Articles sur Scrum référencés dans le livre

- En anglais – *Scrum Guide*, Ken SCHWABER, <http://www.scrumalliance.org/resources>.
- En français – *Scrum et XP depuis les tranchées*, Henrik Kniberg, traduit en français par Guillaume Mathias, Bruno Orsier, Emmanuel Etasse, Christophe Bunn, <http://henrik-kniberg.developpez.com/livre/scrum-xp/>.

## Blogs

- En anglais – Il y en a des dizaines ! Un bon nombre apparaissent dans cet agrégateur : <http://www.agilevoices.com/aggregator/sources>.
- En français – Les plus intéressants sur Scrum et les méthodes agiles :  
*Être agile*, Thierry CROS, <http://etreagile.thierrycros.net>  
*Le blog agile d'Alex*, Alexandre BOUTIN, <http://www.agilex.fr/>  
*Quality Street*, Jean-Claude GROSJEAN, <http://www.qualitystreet.fr/>  
Emmanuel CHENU, <http://emmanuelchenu.blogspot.com/>

## Bibliographie

### • En anglais

Mike COHN, *Succeeding with Agile : Software Development using Scrum*, Addison Wesley, 2009.

Mike COHN, *Agile Estimating and Planning*, Prentice Hall, 2005.

Roman PICHLER, *Agile Product Management with Scrum : Creating Products that Customers Love*, Addison Wesley, 2010.

Ken SCHWABER, *Agile Project Management with Scrum*, Microsoft Press, 2004.  
James SHORE, *The Art of Agile Development*, O'Reilly, 2007.

### • En français

Per KROLL et Philippe KRUCHTEN, *Guide pratique du RUP*, Campus Press, 2003.

Véronique MESSAGER ROTA, *Gestion de projet, Vers les méthodes agiles*, Eyrolles, 2007.

Pierre VILLEPREUX et Vincent BLACHON, *L'esprit Rugby*, Pearson Education, 2007.

# Index

## A

agilité 3

## B

*backlog* de produit 29, 55, 57, 71, 124,  
170, 178, 228

élément 62

priorité 60, 170

*build* 121, 212, 225

*burndown chart* 70, 99, 197

de *release* 83

de *sprint* 111

## C

capacité 80, 199

cérémonial Scrum 2

chef

de produit 25

de projet 41

Cohn, Mike 18, 159, 168

cycle

agile 12

de développement 9, 10, 16

de vie 9

de vie Scrum 18

en V 9

## D

défaut 140, 189, 217

dette technique 141, 214

directeur de produit 26

## E

équipe 37, 44, 65, 71, 90, 97, 101, 106,  
131, 179

auto-organisation 1, 51

espace de travail ouvert 91

estimation 75, 192

de *backlog* 76

de la valeur 193

tâches 96

*Extreme Programming* 5

## F

*features* 168, 228

## G

*geeks* 45

## I

*IceScrum* 26, 83, 224

*impediment* 106

intégration continue 212  
itération 10

## J

jalon 9, 15

## L

*Lean* 244  
*Software 5*

## M

*Manifeste agile* 2, 156  
méthode agile 1, 12  
métier 30  
modèle de cycle 9

## O

obstacle 43, 110  
outil 39, 60

## P

*personas* 172  
phases 9, 15  
plan  
    de *release* 81, 123, 232  
    de *sprint* 98, 111  
planification  
    de *release* 70, 144  
    de *sprint* 235  
    de *sprint* 144  
*planning poker* 76, 193  
point 193  
pratiques 3, 4, 152, 243  
priorité 60, 229  
*priority poker* 170  
processus 9, 42

Product Owner 25, 66, 90, 259  
    *backlog* 65  
    disponibilité 33  
    responsabilités 28  
    tests 37

## R

RAF 207  
*refactoring* 213  
référentiel des exigences 56  
*release* 10, 19, 70, 147, 167  
    fin 23, 73, 141  
remaniement de code 213  
rérospective 129, 130, 144  
réunions 2  
    de planification du *sprint* 90  
revue de *sprint* 119, 144  
    démonstration 122  
*roadmap* 225

## S

Schwaber, Ken 5, 19, 35, 92, 151, 155  
Scrum 5, 155  
    utilisateurs 255  
Scrum Alliance 5  
scrum quotidien 106  
ScrumMaster (responsabilités) 42, 179  
signification de fini 134, 139, 178  
*spike* 217  
*sprint* 10, 146, 188  
    but 95  
    de stabilisation 23  
    durée 14, 78  
    rérospective 129  
    zéro 20  
StakeHolder 225  
*standup meeting* 106  
*story* 145  
    test 184  
    type 63

*storyless* 91

Sutherland, Jeff 112

## T

tableau des tâches 236

tâches 95, 115

*technical debt* 141

test d'acceptation 203, 237

*timebox* 13, 74

*timeline* 133

traçabilité 176

## U

*use-cases* 176

*user stories* 173

utilité

cardinale 194

ordinale 194

## V

vélocité 70, 80, 87, 123, 198

vision 28, 166, 227