1

L'évolution logicielle et le refactoring

Dans la plupart des ouvrages informatiques, nous adoptons le point de vue du créateur de logiciel. Ainsi, les livres consacrés à la gestion de projet informatique décrivent les processus permettant de créer un logiciel de A à Z et s'achèvent généralement à la recette de la première version de celui-ci.

Malheureusement, la création ne représente souvent qu'une petite partie du cycle de vie d'un logiciel. Par exemple, dans le domaine des assurances, des logiciels sont âgés de plusieurs dizaines d'années et continuent d'évoluer au gré des nouvelles réglementations et des nouveaux produits.

L'objectif de ce chapitre d'introduction est de proposer une synthèse de l'évolution logicielle et de montrer l'importance du processus de refactoring pour faire face aux challenges imposés par les forces du changement.

Le chapitre comporte trois grandes sections :

- La première expose la problématique de l'évolution logicielle et insiste sur les moyens de lutter contre ses effets pervers. Le processus de maintenance qui est au cœur de cette problématique est décrit ainsi que le positionnement du refactoring par rapport à cette activité majeure de l'ingénierie logicielle.
- La deuxième section donne la définition du refactoring et résume les objectifs et la typologie des actions de refactoring. Les bénéfices et les challenges de cette activité sont en outre analysés.
- La dernière section positionne le refactoring par rapport aux méthodes agiles, le refactoring se révélant une activité clé au sein de ces méthodes itératives. Bien entendu, le

refactoring ne se limite pas aux méthodes agiles et peut être mis en œuvre dans le cadre de démarches classiques de développement.

La problématique de l'évolution logicielle

En 1968, le phénomène de crise logicielle est identifié lors d'une conférence organisée par l'OTAN. Par crise logicielle, nous entendons la difficulté pour les projets informatiques de respecter les délais, les coûts et les besoins des utilisateurs.

Face à cette crise, de nombreuses solutions sont proposées : langages de plus haut niveau pour gagner en productivité, méthodes de conception permettant d'améliorer l'adéquation entre les fonctionnalités du logiciel et l'expression des besoins des utilisateurs, méthodes de gestion de projets plus adaptées, etc.

Force est de constater que si la situation s'est améliorée depuis, elle reste encore perfectible. D'après le rapport *CHAOS: a Recipe for Success,* du Standish Group, le taux de réussite d'un projet au sein d'une grande entreprise est de 29 % en 2004. Une majorité de projets (53 %) aboutissent, mais sans respecter le planning, le budget ou le périmètre fonctionnel prévus. Les 18 % restants sont constitués des projets purement et simplement arrêtés.

Cette étude ne s'intéresse qu'à la première version d'un logiciel. Or celui-ci va nécessairement devoir changer pour s'adapter au contexte mouvant de ses utilisateurs.

De notre point de vue, le succès d'un projet ne se mesure pas tant à sa capacité à délivrer une première version opérationnelle du logiciel, mais à sa capacité à créer un logiciel assez robuste pour affronter les épreuves des forces du changement.

En un mot, l'évolution logicielle est *darwinienne.* Ce sont les logiciels les mieux adaptés qui survivent.

Le cycle de vie d'un logiciel

La vie d'un logiciel, qu'il soit réalisé à façon au sein d'une entreprise ou à une fin industrielle chez un éditeur, ne s'arrête pas après la livraison de la première version.

À l'instar des êtres vivants, le cycle de vie d'un logiciel connaît cinq grandes phases :

- **Naissance.** Le logiciel est conçu et développé à partir de l'expression de besoins des utilisateurs.
- **Croissance.** De nombreuses fonctionnalités sont ajoutées à chaque nouvelle version en parallèle des correctifs.
- **Maturité.** Le nombre de nouvelles fonctionnalités diminue. Les nouvelles versions sont essentiellement des adaptations et des corrections.
- **Déclin.** L'ajout de nouvelles fonctionnalités est problématique, et les coûts de maintenance deviennent rédhibitoires. Le remplacement du logiciel est envisagé.

- **Mort.** La décision de remplacement est prise. Il peut y avoir une période transitoire, pendant laquelle l'ancien logiciel et le nouveau fonctionnent en même temps. Généralement, on assiste à une migration de la connaissance de l'ancien vers le nouveau logiciel. Cette connaissance est constituée, entre autres, des processus fonctionnels, des règles de gestion et des données. Cet aspect est problématique lorsque la connaissance est enfouie dans l'ancien logiciel et n'est pas documentée.

Le cycle de vie élémentaire d'un logiciel consiste ainsi en une succession de versions, chacune apportant son lot de modifications. Entre deux versions, des correctifs et des évolutions mineures sont réalisés en fonction des anomalies non détectées en recette mais constatées en production.

La figure 1.1 illustre ce cycle de vie.

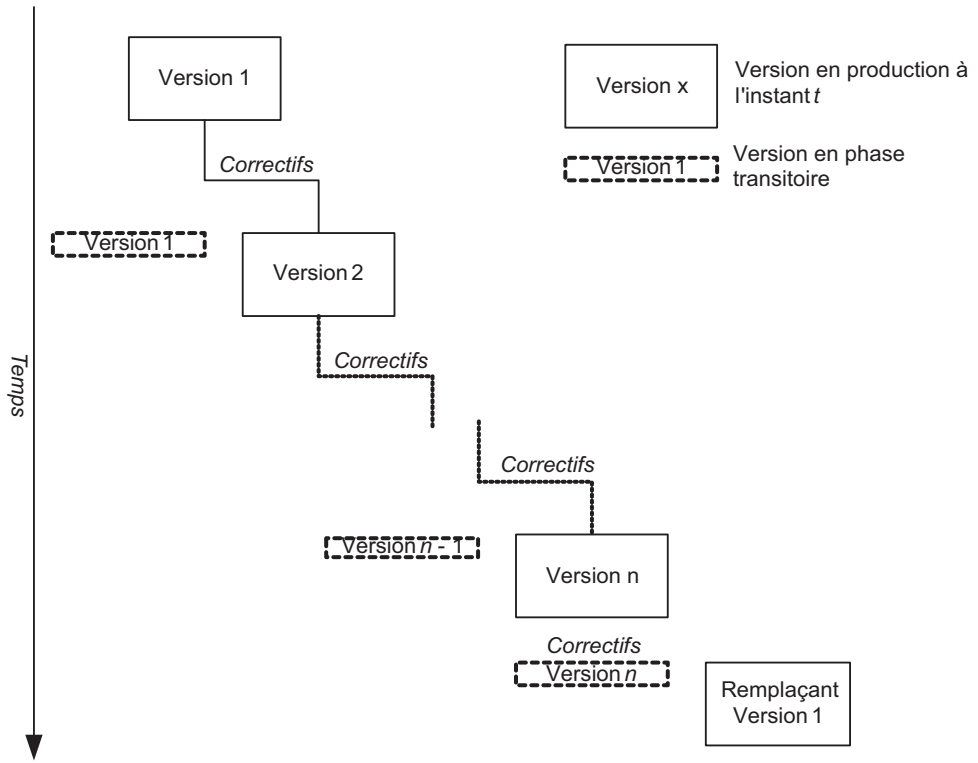

Figure 1.1
Le cycle de vie élémentaire d'un logiciel

Le cycle de vie réel d'un logiciel d'éditeur est plus complexe, la politique commerciale et contractuelle de l'éditeur vis-à-vis de ses clients l'obligeant à maintenir en parallèle plusieurs versions de son logiciel.

Le cycle de vie d'un logiciel d'éditeur a l'allure générale illustrée à la figure 1.2.

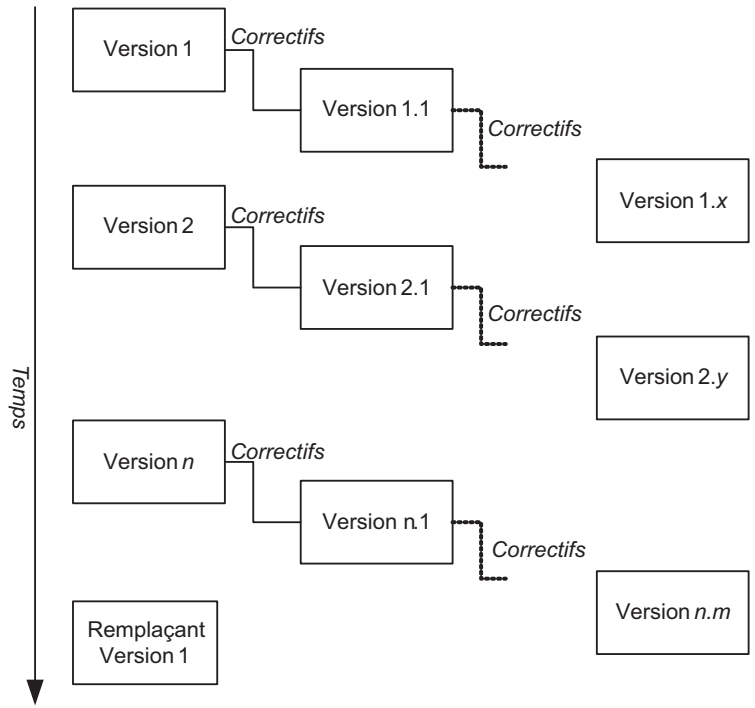

Figure 1.2
Le cycle de vie d'un logiciel d'éditeur

Le cycle de vie d'un logiciel est en fait influencé par le cycle de vie d'autres logiciels. Un logiciel est en effet le plus souvent dépendant d'autres logiciels, comme les systèmes d'exploitation, SGBD, bibliothèques de composants, etc. Le cycle de vie du matériel a aussi une influence, mais dans une moindre mesure, car ce cycle est beaucoup plus long.

Cette interdépendance est un facteur important de complexité pour la gestion du cycle de vie des logiciels, pourtant régulièrement oublié par les chefs de projet. Or, du fait de la généralisation des composants réutilisables, à l'image des frameworks Open Source, et des progiciels, cette problématique doit être prise en compte pour assurer la pérennité des investissements.

Les lois de l'évolution logicielle

Outre le taux d'échec important des projets informatiques, la crise logicielle concerne aussi les immenses difficultés rencontrées pour doter un logiciel d'un cycle de vie long, assorti d'un niveau de service convenable et économiquement rentable.

Le professeur Meir Manny Lehman, de l'Imperial College of Science and Technology de Londres, a mené une étude empirique sur l'évolution des logiciels, en commençant par

analyser les changements au sein du système d'exploitation pour gros systèmes OS/390 d'IBM. Démarrée en 1969 et encore poursuivie de nos jours, cette étude fait émerger huit lois, produites de 1974 à 1996, applicables à l'évolution des logiciels.

Quatre de ces lois de Lehman sont particulièrement significatives du point de vue du refactoring :

- **Loi du changement continuel.** Un logiciel doit être continuellement adapté, faute de quoi il devient progressivement moins satisfaisant à l'usage.
- **Loi de la complexité croissante.** Un logiciel a tendance à augmenter en complexité, à moins que des actions spécifiques ne soient menées pour maintenir sa complexité ou la réduire.
- **Loi de la croissance constante.** Un logiciel doit se doter constamment de nouvelles fonctionnalités afin de maintenir la satisfaction des utilisateurs tout au long de sa vie.
- **Loi de la qualité déclinante.** La qualité d'un logiciel tend à diminuer, à moins qu'il ne soit rigoureusement adapté pour faire face aux changements.

Une des conclusions majeures des lois de Lehman est qu'un logiciel est un système fortement dépendant de son environnement extérieur. Ses évolutions sont dictées par celui-ci selon le principe du feed-back : tout changement dans l'environnement extérieur envoie un signal au sein du logiciel, dont les évolutions constituent la réponse renvoyée à l'extérieur. Cette réponse peut générer elle-même des changements, engendrant ainsi une boucle de rétroaction.

Les forces du changement qui s'appliquent à un logiciel sont tributaires de celles qui s'appliquent à une entreprise. La figure 1.3 illustre quelques forces du changement qui poussent les logiciels à évoluer.

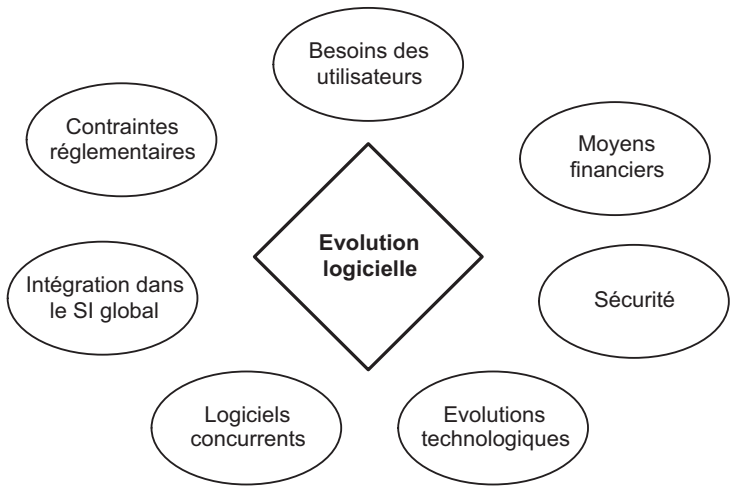

Figure 1.3
Forces du changement et évolution logicielle

L'érosion du design

La loi de la croissance constante aboutit à un phénomène appelé *érosion du design.* Au moment de la conception initiale du logiciel, l'anticipation des fonctionnalités futures est souvent très difficile à moyen ou long terme. De ce fait, le design d'un logiciel découle de choix de conception qui étaient pertinents lors de sa création, mais qui peuvent devenir invalides au fil du temps.

Ce processus d'érosion du design fait partie des phénomènes constatés par les lois de la qualité déclinante et de la complexité croissante. Cette érosion peut devenir telle qu'il devienne préférable de réécrire le logiciel plutôt que d'essayer de le faire évoluer à partir de l'existant.

Un exemple concret de ce phénomène est fourni par le logiciel Communicator, de Netscape, donné en 1998 à la communauté Open Source Mozilla afin de contrer Internet Explorer de Microsoft. Six mois après que le navigateur est devenu Open Source, les développeurs ont considéré que le moteur de rendu des pages, c'est-à-dire le cœur du navigateur, devait être complètement réécrit, son code source étant trop érodé pour être maintenable et évolutif.

La feuille de route de la communauté Mozilla (disponible sur *http://www.mozilla.org/roadmap.html*) donne les raisons du développement de son remplaçant Gecko. En voici un extrait, suivi de sa traduction par nos soins :

> *« Gecko stalwarts are leading an effort to fix those layout architecture bugs and design flaws that cannot be treated by patching symptoms. Those bugs stand in the way of major improvements in maintainability, footprint, performance, and extensibility. Just by reducing source code complexity, Gecko stands to become much easier to maintain, faster, and about as small in dynamic footprint, yet significantly smaller in code footprint. »*
>
> « L'équipe Gecko travaille sur la correction des bogues d'architecture et des erreurs de conception qui ne peuvent être traités en appliquant des patchs sur les symptômes. Ces bogues empêchent des améliorations majeures en maintenabilité, en consommation de ressources, en performance et en évolutivité. En réduisant la complexité du code, Gecko va devenir plus facile à maintenir, plus rapide et à peu près similaire en terme de consommation de ressources bien que plus petit en terme de code. »

En 2004, une deuxième étape a été franchie avec le lancement du navigateur Web Firefox et du client de messagerie Thunderbird, cassant définitivement le monolithisme de la solution Mozilla, jugée trop complexe.

Ce phénomène d'érosion du design est une conséquence de la loi de la croissance constante. En effet, les choix de design initiaux ne tiennent pas ou pas assez compte des besoins futurs puisque ceux-ci ne peuvent généralement être anticipés à moyen ou long terme. De ce fait, le logiciel accumule tout au long de sa vie des décisions de design non optimales d'un point de vue global. Cette accumulation peut être accentuée par des méthodes de conception itératives favorisant les conceptions locales au détriment d'une

conception globale, à moins d'opérer des consolidations de code, comme nous le verrons plus loin dans ce chapitre avec les méthodes agiles.

Même simples, les décisions de design initiales ont des répercussions très importantes sur l'évolution d'un logiciel, pouvant amener à opérer maintes contorsions pour maintenir le niveau de fonctionnalités attendu par les utilisateurs.

Enfin, les logiciels souffrent d'un manque de traçabilité des décisions de design, rendant difficile la compréhension de l'évolution du logiciel. Les décisions de design associées aux évolutions sont souvent opportunistes et très localisées, faute d'informations suffisantes pour définir une stratégie d'évolution.

In fine, à défaut de solution miracle à ce problème fondamental, seules des solutions palliatives sont proposées.

Les forces du changement qui s'appliquent aux logiciels sont sans commune mesure avec celles qui s'appliquent aux produits industriels. Ces derniers sont conçus par rapport à des besoins utilisateur définis à l'avance. Lorsque le produit ne correspond plus à ces besoins, sa production est tout simplement arrêtée. Dans le cadre d'un logiciel, les besoins ne sont pas figés dans le temps et varient même souvent dès le développement de la première version.

L'exemple des progiciels, dont la conception relève d'une approche « industrielle » (au sens produit), est caractéristique à cet égard. Même en essayant de standardiser au maximum les fonctionnalités au moyen d'un spectre fonctionnel le plus large et complet possible, l'effort d'adaptation à l'environnement est très loin d'être négligeable. Par ailleurs, l'arrêt d'un logiciel est généralement problématique, car il faut reprendre l'existant en terme de fonctionnalités aussi bien que de données.

Les solutions palliatives sont mises en œuvre soit *a priori,* c'est-à-dire lors de la conception et du développement initiaux, soit *a posteriori,* c'est-à-dire de la croissance jusqu'à la mort du logiciel.

Les solutions *a priori* font partie des domaines du génie logiciel les plus actifs. Les méthodes orientées objet en sont un exemple. Elles ont tenté de rendre les logiciels moins monolithiques en les décomposant en objets collaborant les uns avec les autres selon un protocole défini. Tant que le protocole est maintenu, le fonctionnement interne des objets peut évoluer indépendamment de celui des autres. Malheureusement, force est de constater que ce n'est pas suffisant, le protocole étant le plus souvent lui-même impacté par la moindre modification du logiciel.

Une autre voie est explorée avec la POA (programmation orientée aspect), complémentaire des méthodes précédentes. La POA cherche à favoriser la séparation franche des préoccupations au sein des logiciels, leur permettant de faire évoluer les composants d'une manière plus indépendante les uns des autres. Deux préoccupations typiques d'un logiciel sont les préoccupations d'ordre fonctionnel, ou métier, et les préoccupations techniques, comme la sécurité, la persistance des données, etc. En rendant le métier indépendant de la technique, il est possible de pérenniser le logiciel en limitant les impacts des forces du changement à une seule des deux préoccupations, dans la mesure du possible.

Des solutions *a posteriori* vont être présentées tout au long de cet ouvrage. La problématique d'érosion du design est en effet directement adressée par le refactoring. Il faut avoir conscience cependant que ces solutions *a posteriori* ne sont pas un remède miracle et que le refactoring n'est pas la pierre philosophale capable de transformer un logiciel mal conçu en la quintessence de l'état de l'art. La dégénérescence du logiciel peut atteindre un tel degré que seul un remplacement est susceptible de régler les problèmes.

Le rôle de la maintenance dans l'évolution logicielle

Située au cœur de la problématique énoncée par les lois de Lehman, la maintenance est une activité majeure de l'ingénierie logicielle. C'est en partie grâce à elle que le logiciel respecte le niveau d'exigence des utilisateurs et permet, dans une certaine mesure, que les investissements consacrés au logiciel soient rentabilisés.

D'une manière générale, la maintenance est un processus qui se déroule entre deux versions majeures d'un logiciel. Ce processus produit des correctifs ou de petites évolutions, qui sont soit diffusés en production au fil de l'eau, soit regroupés sous forme de versions mineures.

La figure 1.4 illustre les différentes étapes du processus de maintenance pour la gestion des anomalies et des évolutions mineures.

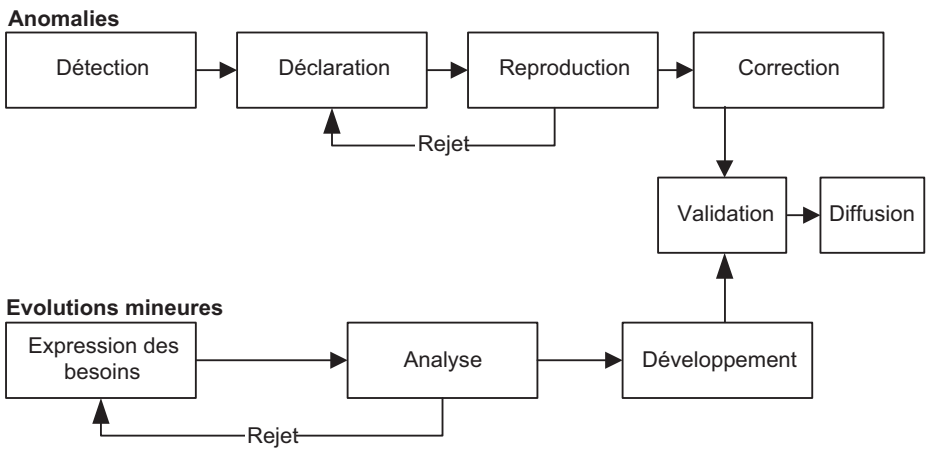

Figure 1.4
Le processus de maintenance

Pour la gestion des anomalies, appelée *maintenance corrective,* le processus de maintenance est très proche de celui d'une recette. La principale différence est que la détection et la déclaration ne sont pas réalisées par des testeurs mais directement par les utilisateurs finals. De ce fait, la tâche de correction est plus difficile. Les déclarations sont généralement moins précises, et la reproductibilité, qui est une condition nécessaire à la correction, est

complexifiée faute d'un scénario précis à rejouer. À cela s'ajoutent les impératifs de production en terme de délais de réaction, de contraintes de diffusion, etc.

Pour la gestion des évolutions mineures, appelée *maintenance évolutive,* le processus de maintenance s'apparente à la gestion d'un petit projet de développement. La principale différence réside dans la phase d'analyse, où une demande peut être rejetée si elle constitue une évolution majeure devant être prise en compte dans la prochaine version du logiciel.

Signalons aussi les deux autres types de maintenance suivants :

- Maintenance préventive : il s'agit de réaliser des développements pour prévenir des dysfonctionnements futurs.
- Maintenance adaptative : il s'agit d'adapter le logiciel aux changements survenant dans son environnement d'exécution.

En conclusion, la caractéristique des activités de maintenance par rapport à un projet de développement est leur durée. Sur l'ensemble de la vie d'un logiciel, la durée de la maintenance est généralement supérieure à celle du développement. Le cycle développement-mise en production est beaucoup plus court pour la maintenance et comprend de nombreuses itérations.

Maintenance et refactoring

Comme vous le verrez tout au long de l'ouvrage, le refactoring est une activité de maintenance d'un genre particulier. Il s'agit non pas d'une tâche de correction des anomalies ou de réalisation de petites évolutions pour les utilisateurs mais d'une activité plus proche de la maintenance préventive et adaptative dans le sens où elle n'est pas directement visible de l'utilisateur.

Le refactoring est une activité d'ingénierie logicielle consistant à modifier le code source d'une application de manière à améliorer sa qualité sans altérer son comportement du point de vue de ses utilisateurs. Son rôle concerne essentiellement la pérennisation de l'existant, la réduction des coûts de maintenance et l'amélioration de la qualité de service au sens technique du terme (performance et fiabilité). Le refactoring est en ce sens différent de la maintenance corrective et évolutive, qui modifie directement le comportement du logiciel, respectivement pour corriger un bogue ou ajouter ou améliorer des fonctionnalités.

Par ailleurs, la maintenance a généralement une vision à court terme de l'évolution du logiciel puisqu'il s'agit de répondre à l'urgence et d'être réactif. Le refactoring est une démarche qui vise à pallier activement les problèmes de l'évolution logicielle, notamment celui de l'érosion. Il s'agit donc d'une activité au long court, devant être dotée d'une feuille de route continuellement mise à jour au gré des changements.

Idéalement, le refactoring doit être envisagé comme un processus continu plutôt que comme un chantier devant être mené ponctuellement. Il peut être effectué en parallèle de la maintenance dès lors qu'il est compatible avec ses contraintes de réactivité. À l'instar de la correction d'erreur, plus un refactoring est effectué tôt moins il coûte cher.

Le périmètre d'intervention du refactoring

Vous avez vu à la section précédente comment situer le refactoring par rapport à la problématique de l'évolution logicielle et à l'activité de maintenance. La présente section se penche sur le périmètre d'intervention du refactoring.

Le refactoring étant un processus délicat, nous donnons une synthèse dont il importe de bien mesurer les bénéfices et les risques liés à sa mise en œuvre. Ainsi, la réduction de la complexité du code *via* un refactoring entraîne normalement une diminution des coûts de maintenance. Cependant, en fonction de l'importance des modifications à apporter au code, les coûts de transformation et les risques peuvent devenir rédhibitoires.

Pour toutes ces raisons, il est essentiel d'anticiper le refactoring dès la conception d'un logiciel.

Les niveaux de refactoring

Nous pouvons considérer trois niveaux de refactoring, selon la complexité de sa mise en œuvre : le refactoring chirurgical, le refactoring tactique et le refactoring stratégique.

Le refactoring chirurgical

Par refactoring chirurgical, nous entendons la réalisation d'opérations de refactoring limitées et localisées dans le code source.

Il s'agit de refondre quelques composants sans altérer leurs relations avec le reste du logiciel. Nous utilisons typiquement les techniques de base détaillées dans la première partie de cet ouvrage.

De telles opérations sont tout à fait réalisables dans le cadre de la maintenance puisque leur périmètre limité ne justifie pas la mise en place d'une structure de projet *ad hoc*. Elle est particulièrement pertinente pour consolider certaines opérations de maintenance réalisées pour répondre à l'urgence mais nuisant à la qualité générale du logiciel.

Le refactoring tactique

Par refactoring tactique, nous entendons la refonte complète de quelques composants repensés tant dans leur fonctionnement interne que dans leurs relations avec le reste du logiciel. Il peut s'agir, par exemple, d'introduire des design patterns dans le code, comme nous l'expliquons au chapitre 6.

Ce type d'opération est plus difficile à réaliser de manière intégrée à la maintenance. L'effort de test pour valider le refactoring peut en effet devenir rapidement incompatible avec la réactivité nécessaire aux corrections de bogues ou aux évolutions demandées en urgence. Cependant, la structure projet à mettre en place reste généralement limitée.

Le refactoring stratégique

Par refactoring stratégique, nous entendons la remise à plat de la conception du logiciel afin de l'adapter aux exigences présentes et futures des utilisateurs. Dans ce cadre, l'ensemble des techniques présentées dans cet ouvrage sera vraisemblablement mis en œuvre.

La frontière entre ce refactoring et une réécriture pure et simple du logiciel est si ténue qu'il est essentiel de bien réfléchir aux objectifs à atteindre. La démarche de refactoring stratégique est évidemment inutile si, *in fine,* nous devons réécrire le logiciel.

La phase d'analyse du logiciel doit être particulièrement soignée de manière à définir et chiffrer une stratégie de refonte. Cette stratégie doit être comparée à une réécriture ou à un remplacement par un progiciel et pondérée par les facteurs de risque liés à chacune de ces démarches.

Le processus de refactoring

Le processus de refactoring compte les quatre phases principales suivantes, dont la première est généralement mise en œuvre dès la création du logiciel :

1. Mise en place de la gestion du changement.
2. Analyse du logiciel.
3. Refonte du code.
4. Validation du refactoring.

Mise en place de la gestion du changement

Le développement d'un logiciel nécessite l'utilisation d'une infrastructure pour gérer les changements qu'il subira tout au long de sa vie. Cette infrastructure de gestion du changement est articulée autour de trois thèmes :

- Gestion de configuration, qui concerne la gestion des versions successives des composants du logiciel.
- Gestion des tests, qui permet de valider le fonctionnement du logiciel en regard des exigences des utilisateurs.
- Gestion des anomalies, qui concerne la gestion du cycle de vie des anomalies détectées par les recetteurs ou les utilisateurs finals.

Cette infrastructure est généralement mise en place lors de la création du logiciel. Elle peut être artisanale, c'est-à-dire gérée manuellement sans l'aide d'outils spécialisés, dans le cadre de projets de taille modeste.

La mise en place d'une telle infrastructure peut sembler au premier abord inutilement coûteuse et lourde à mettre en œuvre. Si ce raisonnement peut se tenir pour la première version, il n'en va pas de même si nous considérons le cycle de vie dans sa globalité. La contrainte forte du refactoring, qui consiste à ne pas modifier le comportement du logiciel du point de vue des utilisateurs, nécessite une infrastructure de gestion du changement

solide afin de se dérouler avec un maximum de sécurité (grâce, par exemple, à la possibilité de faire marche arrière au cours d'une opération de refactoring non concluante). Dans cette perspective, la mise en place d'une telle infrastructure est vite rentabilisée.

Analyse du logiciel

L'analyse du logiciel consiste à détecter les composants candidats au refactoring au sein du code du logiciel. Cette analyse est à la fois quantitative et qualitative.

L'analyse quantitative consiste à calculer différentes métriques sur le logiciel. Ces métriques sont comparées avec les règles de l'art pour identifier les zones problématiques. Malheureusement imparfaites, puisqu'elles peuvent signaler des anomalies là où il n'y en a pas (fausses alertes) et passer à côté de problèmes sérieux (faux amis), ces métriques ne bénéficient pas de la même fiabilité que les métriques physiques, par exemple.

De ce fait, l'analyse qualitative est cruciale pour le refactoring puisque c'est par le biais de ses résultats que nous pouvons décider des zones à refondre. Cette analyse consiste essentiellement à auditer le code manuellement ou à l'aide d'outils et à revoir la conception du logiciel pour détecter d'éventuelles failles apparues au cours de la vie du logiciel.

Pour optimiser cette phase d'analyse qualitative, qui peut se révéler extrêmement coûteuse et donc peu rentable, nous effectuons des sondages guidés par les résultats obtenus lors de l'analyse quantitative.

À partir de la liste des zones à refondre identifiées lors de cette phase, nous devons décider quelles seront celles qui devront effectivement l'être. Pour cela, il est nécessaire d'évaluer, pour chacune d'elles, le coût de la refonte, le gain attendu en terme de maintenabilité ou d'évolutivité et les risques associés à cette refonte.

Les risques peuvent être déduits d'une analyse d'impact. Une modification strictement interne à une classe est généralement peu risquée alors que la mise en œuvre d'un design pattern peut avoir des impacts très importants sur l'ensemble du logiciel.

Refonte du code

Une fois les zones à refondre sélectionnées, la refonte du code peut commencer. Le souci majeur lors d'une opération de refactoring est de s'assurer que les modifications du code source n'altèrent pas le fonctionnement du logiciel. De ce fait, avant toute modification du code, il est nécessaire de mettre au point une batterie de tests sur le code originel si elle n'existe pas déjà. Ils seront utiles pour la phase de validation.

Lorsque les tests de non-régression sont validés sur le code originel, nous pouvons effectuer la refonte proprement dite. Pour faciliter la validation du refactoring, il est recommandé de réaliser une validation à chaque opération de refactoring unitaire. Si nous accumulons un grand nombre de modifications non testées, toute erreur détectée devient plus difficile à corriger.

Validation du refactoring

La phase de validation du refactoring consiste à vérifier si la contrainte de non-modification du logiciel a été respectée.

Pour réaliser cette validation, nous nous reposons sur la batterie de tests mise en place en amont de la refonte. Il est donc important que ces tests soient le plus exhaustifs possible afin de valider le plus de cas de figure possible.

Ces tests de validation soulignent une fois de plus l'importance de la gestion du changement pour un logiciel. Lors de la création du logiciel, des tests ont été spécifiés pour effectuer sa recette. Si aucune démarche de capitalisation des tests n'a été mise en place, le coût du refactoring n'est pas optimisé puisqu'une partie de ceux-ci doivent être recréés *ex nihilo.*

Une fois la refonte validée, nous pouvons livrer la nouvelle version du logiciel aux utilisateurs finals. Comme pour toute nouvelle version, une partie de l'équipe projet doit être maintenue le temps nécessaire pour assurer le transfert de compétence vers l'équipe de maintenance. Des bogues seront en effet inévitablement détectés par les utilisateurs et devront être rapidement corrigés.

Bénéfices et challenges du refactoring

Comme nous l'avons vu aux sections précédentes, le refactoring d'un logiciel n'est jamais une opération anodine. Il est essentiel de mesurer les bénéfices et challenges liés à sa mise en œuvre pour décider de l'opportunité de son utilisation.

Les bénéfices du refactoring

L'objectif du refactoring est d'améliorer la qualité du code d'un logiciel. En améliorant la qualité du code, nous cherchons à optimiser sa maintenabilité et son évolutivité afin de rentabiliser les investissements tout au long de la vie du logiciel.

Pour améliorer la maintenabilité d'un logiciel, nous cherchons principalement à réduire sa complexité et à diminuer le nombre de lignes de code à maintenir. Ce dernier point consiste souvent à chasser les duplications de code au sein du logiciel. Comme le montre l'étude de cas de la partie III de l'ouvrage, cette traque aux dupliquas peut être particulièrement lourde.

Pour améliorer l'évolutivité d'un logiciel, garante de sa pérennité, nous cherchons principalement à faire reposer le logiciel sur des standards. Par ailleurs, le respect des meilleures pratiques, pour beaucoup formalisées sous forme de design patterns, est souvent un gage de meilleure évolutivité du logiciel. La mise en œuvre de ces modèles ne va toutefois pas sans poser de problèmes, comme nous le verrons à la section suivante.

Les challenges du refactoring

Le premier challenge du refactoring consiste à… le « vendre ». Dans la mesure où le refactoring n'offre pas de gains directement visibles des utilisateurs finals, ceux-ci ont

une certaine difficulté à y adhérer. Si les utilisateurs finals sont détenteurs des budgets affectés au logiciel, leur tentation de favoriser les évolutions par rapport à une consolidation du code existant est évidemment forte.

Pour les convaincre de réaliser une ou plusieurs opérations de refactoring, il est nécessaire de bien connaître les coûts associés à la maintenance et à l'évolution du logiciel et d'être en mesure, suite à une analyse, de démontrer que l'investissement dans le refactoring est rentable.

L'idéal est d'injecter des opérations de refactoring tout au long du processus de maintenance de manière à rendre l'opération plus indolore pour les utilisateurs finals. Malheureusement, ce n'est pas toujours possible, ce mode de fonctionnement dépendant fortement de la qualité initiale du logiciel. Compte tenu des dépassements de délai ou de budget que l'on constate dans la majorité des projets informatiques, la tendance reste à mettre le logiciel en production le plus rapidement possible.

Le deuxième challenge du refactoring réside dans l'évaluation des risques à l'entreprendre comparés à ceux de ne rien faire. Dans certains cas, il peut être préférable de laisser les choses en l'état plutôt que de se lancer dans une opération dont les chances de succès sont faibles.

Il est donc nécessaire, dans la mesure du possible, d'évaluer le périmètre de la refonte en analysant les impacts liés aux modifications du code source. Cela n'est malheureusement pas toujours possible, car les composants d'un logiciel peuvent être extrêmement dépendants les uns des autres, induisant des effets de bord difficilement contrôlables lorsque l'un d'eux est modifié.

Le refactoring n'est efficace que sur un logiciel dont les fondements sont sains. Si les fondements du logiciel sont mauvais, le refactoring n'est pas la démarche adaptée pour y remédier. Seule une réécriture ou un remplacement par un progiciel, que nous pouvons espérer mieux conçu, permet de solutionner ce problème.

Le troisième challenge du refactoring est de motiver les équipes de développeurs pour refondre le code. Ce type d'opération peut être perçu, à tort de notre point de vue, comme une tâche ingrate et peu valorisante. Par ailleurs, les développeurs font toujours preuve de réticence pour modifier le code d'un autre, du simple fait de la difficulté à le comprendre.

Pour les projets de refactoring lourds, qui ne peuvent être intégrés à la maintenance, il est important de bien communiquer pour démontrer l'intérêt de la démarche et justifier le défi qu'elle représente pour les développeurs.

Anticipation du refactoring

La meilleure façon d'anticiper le refactoring est de bien concevoir le logiciel dès l'origine afin de lui donner le plus de flexibilité possible face aux évolutions qu'il connaîtra tout au long de sa vie. Pour cela, il est nécessaire de se reposer sur les meilleures pratiques en la matière, formalisées notamment sous forme de design patterns.

Il est important de séparer au maximum les préoccupations (métiers et techniques, par exemple), de manière à limiter les effets de bord lors d'un refactoring. Cette séparation peut reposer sur la POA, mais ce n'est pas obligatoire. Il existe d'autres techniques, comme l'inversion de contrôle proposée par les conteneurs légers de type Spring.

L'avenir du logiciel doit être anticipé dans la mesure du possible afin de pallier l'érosion logicielle, qui est en elle-même inéluctable. La traçabilité des décisions de conception doit être assurée pour servir de base à l'analyse du logiciel dans le cadre des opérations de refactoring.

Un soin particulier doit être apporté à la programmation et à la documentation du code. Des langages tels que Java disposent de règles de bonne programmation, qu'il est primordial de respecter. Plus le code est correctement documenté, plus il est facile de le refondre puisqu'il est mieux compris.

Enfin, il est nécessaire de capitaliser sur les tests afin de rendre le refactoring efficace. Plus les tests sont complets, plus nous avons des garanties de non-régression du logiciel refondu. Cette capitalisation peut être notamment assurée par la mise en place d'une infrastructure de tests automatisés, comme nous le verrons au chapitre 5. Malheureusement, les tests sont souvent les parents pauvres des développements, car ils ne sont le plus souvent utilisés qu'en tant que variable d'ajustement pour respecter les délais.

Le refactoring au sein des méthodes agiles

Les méthodes agiles sont de nouvelles approches de modélisation et de développement logiciel. Leur objectif fondamental est de produire rapidement des logiciels correspondant aux besoins des utilisateurs en associant ces derniers à un processus itératif favorisant la communication avec les informaticiens.

Ce processus, qui n'est pas sans rappeler les méthodes RAD (Rapid Application Development), nécessite d'introduire des phases de refactoring importantes afin de consolider le code produit à chaque itération.

C'est la raison pour laquelle il nous semble utile d'étudier spécifiquement le rôle du refactoring en leur sein.

Le manifeste du développement logiciel agile

En 2001, plusieurs experts, parmi lesquels Kent Beck, Ron Jeffries et Martin Fowler, un des pères du refactoring, se réunissent lors d'un atelier à Snowbird, aux États-Unis, pour réfléchir à de nouvelles approches de modélisation et de développement logiciel, incarnées par XP (eXtreme Programming) ou DSDM (Dynamic System Development Methodology).

Ils tirent de leurs réflexions un manifeste jetant les bases des méthodes agiles (voir *http://agilemanifesto.org/*), dont voici des extraits en anglais, suivis d'une traduction par nos soins :

> *"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> *– **Individuals and interactions** over processes and tools.*
>
> *– **Working software** over comprehensive documentation.*
>
> *– **Customer collaboration** over contract negotiation.*
>
> *– **Responding to change** over following a plan.*
>
> That is, while there is value in the items on the right, we value the items on the left more."

Traduction :

> « Nous découvrons de nouvelles façons de développer des logiciels en le faisant et en aidant les autres à le faire. Au travers de ce travail, nous en sommes venus à privilégier :
>
> – **Les individus et les interactions** par rapport aux processus et aux outils.
>
> – **Les logiciels qui fonctionnent** par rapport à une documentation complète.
>
> – **La collaboration avec les utilisateurs** par rapport à une négociation contractuelle.
>
> – **La réponse aux changements** par rapport au respect d'un plan.
>
> Ainsi, même si les éléments de droite sont importants, ceux de gauche le sont plus encore à nos yeux. »

De ce manifeste découlent une douzaine de principes devant être respectés par les méthodes agiles :

- Accorder la plus haute priorité à la satisfaction de l'utilisateur grâce à une diffusion rapide et continue d'un logiciel opérationnel.
- Accepter les changements apparus dans l'expression des besoins, même tardivement pendant le développement. Les processus agiles doivent être en mesure de supporter le changement pour garantir l'avantage compétitif du client.
- Diffuser des versions opérationnelles du logiciel à échéance régulière, de toutes les deux semaines jusqu'à tous les deux mois, avec une préférence pour la fréquence la plus courte.
- Faire travailler ensemble quotidiennement utilisateurs et développeurs tout au long du projet.
- Construire le projet avec des personnes motivées en leur fournissant l'environnement et le support dont elles ont besoin et en leur faisant confiance.
- Privilégier la conversation face à face, qui est le moyen le plus efficace pour transmettre de l'information à et dans une équipe de développement.
- Considérer les versions opérationnelles du logiciel comme les mesures principales du progrès.
- Considérer les procédés agiles comme les moteurs d'un développement viable. Sponsors, développeurs et utilisateurs doivent pouvoir maintenir un rythme constant indéfiniment.
- Apporter une attention continue à l'excellence technique et à la bonne conception afin d'améliorer l'agilité.

- Privilégier la simplicité, c'est-à-dire l'art de maximiser le travail à ne pas faire.
- Considérer que les meilleures architectures, expressions de besoins et conceptions émergent d'équipes auto-organisées.
- Réfléchir à intervalle régulier à la façon de devenir plus efficace et agir sur le comportement de l'équipe en conséquence.

La position des auteurs du manifeste est résolument pragmatique. Elle part du constat que rien ne peut arrêter les forces du changement et qu'il vaut mieux composer avec.

En rupture totale avec le célèbre cycle en V, ces principes sont cependant loin de faire l'unanimité auprès des informaticiens et de leurs utilisateurs. Leur mise en œuvre représente par ailleurs de véritables défis pour nos organisations actuelles.

Les méthodes agiles

Plusieurs méthodes agiles sont disponibles aujourd'hui, dont nous présentons dans les sections suivantes quelques-unes parmi les plus significatives :

- XP (eXtreme Programming)
- ASD (Adaptive Software Development)
- FDD (Feature Driven Development)
- TDD (Test Driven Development)

XP (eXtreme Programming)

L'eXtreme Programming est certainement la méthode agile la plus connue. Elle définit treize pratiques portant sur la programmation, la collaboration entre les différents acteurs et la gestion de projet.

Cette méthode fait apparaître dans ses principes fondateurs l'utilisation systématique du refactoring pour garantir une qualité constante aux versions livrées aux utilisateurs.

Par rapport aux principes du manifeste, l'eXtreme Programming introduit les éléments novateurs suivants :

- **Le développement piloté par les tests.** Les tests unitaires ont une importance majeure dans la démarche XP. Ceux-ci doivent être réalisés avant même le développement d'une fonctionnalité afin de s'assurer de la bonne compréhension des besoins des utilisateurs. En effet, l'écriture de tests oblige à adopter le point de vue de l'utilisateur et, pour cela, à bien comprendre ce qu'il attend.
- **La programmation en binôme.** La programmation en binôme est certainement un des principes les plus perturbants pour les lecteurs habitués aux méthodes de développement traditionnelles. Il s'agit ici de faire travailler les développeurs deux par deux. Chaque binôme travaille sur la même machine et sur le même code afin de traiter plus rapidement les problèmes et d'améliorer le contrôle du code produit. Les binômes ne sont pas fixes dans le temps, et de nouvelles associations se font jour tout au long du projet.

- **La responsabilité collective du code.** Dans une organisation classique, les développeurs se voient généralement attribuer une partie du code du logiciel sous leur responsabilité. Rien de tel avec XP, tout développeur étant susceptible d'intervenir sur n'importe quelle partie du code. De ce fait, la responsabilité du code est collective.
- **L'intégration continue.** Pour garantir une consistance du code produit et livrer rapidement des versions opérationnelles du logiciel aux utilisateurs, l'XP recommande de réaliser des intégrations très fréquentes du code produit par les différents développeurs. La fréquence minimale recommandée est une fois par jour.
- **Le refactoring.** Le refactoring est une activité majeure de l'XP. C'est grâce à lui que la qualité du code est garantie tout au long des itérations. Son rôle est aussi de faire émerger l'architecture du logiciel, depuis les phases initiales jusqu'à sa livraison définitive aux utilisateurs.

ASD (Adaptive Software Development)

L'ASD est fondé sur le principe de l'adaptation continue du fait de la nécessité d'accepter les changements continuels qui s'imposent aux logiciels. Ainsi, l'ASD est organisé autour d'un cycle en trois phases (spéculation, collaboration et apprentissage) en remplacement du cycle classique des projets informatiques (planification, conception et construction).

La spéculation

La spéculation comporte les cinq étapes suivantes :

1. Initialisation du projet, définissant la mission affectée au projet.
2. Planification générale du projet limitée dans le temps. Toute l'organisation du projet est centrée sur le respect de cette limite.
3. Définition du nombre d'itérations à effectuer et de leur date limite de livraison afin de respecter la limite globale du projet.
4. Définition du thème ou des objectifs de chaque itération.
5. Définition en concertation par les développeurs et les utilisateurs du contenu fonctionnel de chaque itération.

La collaboration

Pendant que l'équipe technique livre des versions opérationnelles du logiciel, les chefs de projet facilitent la collaboration et les développements en parallèle afin de respecter le planning du projet et les besoins des utilisateurs.

L'apprentissage

À la fin de chaque itération, une phase d'apprentissage est prévue. Cette phase est destinée à obtenir le feed-back le plus exhaustif possible sur la version livrée afin d'améliorer continuellement le processus. Le focus est mis sur la qualité du résultat, à la fois du point de vue des utilisateurs et du point de vue technique, ainsi que sur l'efficacité du mode de fonctionnement de l'équipe.

FDD (Feature Driven Development)

Contrairement aux méthodes précédentes, le FDD débute par une phase de conception générale, qui vise à spécifier un modèle objet du domaine du logiciel en collaboration avec les experts du domaine.

Une fois le modèle du domaine spécifié et un premier recueil des besoins des utilisateurs effectué, les développeurs dressent une liste de fonctionnalités à implémenter. La planification et les responsabilités sont alors définies.

Le développement du logiciel autour de la liste des fonctionnalités suit une série d'itérations très rapides, au rythme d'une itération toutes les deux semaines au maximum, composées chacune d'une étape de conception et d'une étape de développement.

TDD (Test Driven Development)

Le TDD place les tests au centre du développement logiciel. Il s'agit d'une démarche complémentaire des méthodes plus globales, comme l'XP, mais centrée sur le développement.

Chaque développement de code, même le plus petit, est systématiquement précédé du développement de tests unitaires permettant de spécifier et vérifier ce que celui-ci doit faire. Puisque les tests portent sur du code qui n'existe pas encore, ils échouent si nous les exécutons. Nous pouvons dès lors ne développer que le code nécessaire et suffisant pour que les tests réussissent.

Un refactoring est ensuite effectué pour optimiser à la fois les tests et le code testé, notamment en supprimant la duplication de code. Au fur et à mesure de l'avancée du projet, de plus en plus de tests sont développés, la règle étant que tout nouveau code ajouté ne doive pas les faire échouer.

Les itérations du TDD sont beaucoup plus courtes qu'en XP puisqu'elles s'enclenchent à chaque morceau de code significatif, comme une méthode de classe. Leur fréquence varie donc de quelques minutes à une heure environ.

En procédant de la sorte, nous garantissons le respect des spécifications et la non-régression à chaque itération.

Rôle du refactoring dans les méthodes agiles

Comme nous venons de le voir, le refactoring est une activité clé des méthodes agiles. Un des pères du refactoring, Martin Fowler, est d'ailleurs à l'origine du manifeste du développement agile.

L'ensemble des méthodes agiles fonctionne sur un mode itératif. Cela favorise l'émergence d'une version finale du logiciel adaptée aux besoins des utilisateurs en leur délivrant à chaque itération une version opérationnelle, mais non finalisée fonctionnellement, du logiciel.

Pour ne pas être victime de l'érosion du design, il est fondamental pour les méthodes agiles d'utiliser le refactoring afin de consolider leur code d'une version à une autre. Ce

mode de fonctionnement exige d'anticiper le refactoring, en appliquant dès le départ les meilleures pratiques de conception et de programmation afin de minimiser l'effort de refactoring.

Comme nous le verrons dans les chapitres suivants de cet ouvrage, le processus de refactoring est très bien outillé dès lors que les refontes à mener sont simples. Il est donc important d'assurer l'émergence d'une architecture solide lors des différentes itérations et de ne pas sombrer dans la tentation du *quick and dirty.*

Le processus de refactoring est particulièrement bien anticipé dans les méthodes agiles telles que XP ou TDD grâce à la capitalisation des tests unitaires.

Conclusion

Vous avez vu dans ce chapitre introductif comment le refactoring se positionnait par rapport à aux problématiques générales d'évolution logicielle et de maintenance.

Vous découvrirez dans les chapitres de la première partie de l'ouvrage comment mettre en œuvre le processus de refactoring, depuis la mise en place de l'infrastructure de gestion du changement jusqu'à la validation du logiciel refondu.