Les phases du projet

1. L'expression de besoins et le cahier des charges

Ces deux termes sont analogues, bien qu'en pratique une expression de besoins soit le point de départ à l'élaboration d'un cahier des charges plus complet.

Comme nous l'avons déjà évoqué, le cahier des charges est plus souvent décrié qu'encensé. Et pourtant, il s'agit d'un document indispensable à la réalisation d'un projet. Le principal problème lié à sa rédaction tient dans l'interprétation réputée équivoque que l'on pourrait lui prêter, suivant que l'on est du côté de la maîtrise d'ouvrage ou de la maîtrise d'œuvre. En effet, les clients rencontrent parfois de réelles difficultés à exprimer clairement leurs besoins (on entend même parfois que la meilleure expression correspond au logiciel réalisé). Il arrive aussi que, rompus à l'exercice, les clients restent délibérément flous dans leurs réponses, cherchant ainsi à maximiser leurs marges de manœuvre et à différer leurs choix.

La société en charge de la réalisation poursuit précisément l'objectif opposé : plus vite elle identifie le périmètre de la solution, meilleur sera le rendement du projet, par le truchement de la capitalisation du savoir-faire.

La situation paraît donc inextricable, et pourtant elle s'impose rapidement car le cahier des charges reste la principale annexe technique d'un contrat de développements informatiques.

Il apparaît clairement que les incompréhensions proviennent du caractère « figé » prêté au cahier des charges. Il serait une fin en soi. Or il n'en est rien et la solution tient dans une autre lecture de la remarque faite ci-dessus. Si la meilleure expression d'un besoin client est supportée par un logiciel, elle n'est qu'une dérivation d'un ensemble de spécifications qui découlent toutes du cahier des charges.

Autrement dit, le cahier des charges n'est nullement un document figé mais plutôt un canevas que le processus d'analyse va compléter et transformer jusqu'à la production du logiciel.

Et tous les processus d'analyse commencent par donner une priorité aux éléments du projet. Il n'y a donc pas de raison de presser le client pour bloquer des éléments qui

pourront être définis ultérieurement, sans remettre en cause le déroulé des opérations.

a. Le contenu d'un cahier des charges

Dès lors, que trouve-t-on dans un « bon » cahier des charges ? Cela dépend évidemment du niveau d'avancement de réflexion de son rédacteur, qu'il soit représentant des utilisateurs (maître d'ouvrage) ou prestataire (maître d'œuvre).

Contexte et objectifs stratégiques

Cette section reprend les grandes lignes du *business case* du demandeur ; la situation existante, la cible. Il est habituel de ne pas dépasser quelques lignes pour rappeler le contexte, la rédaction est souvent courte et synthétique.

Objet

Est énumérée dans cette partie la portée du projet confié au maître d'œuvre. Un lotissement peut être mis en place si les responsabilités et les domaines de compétences sont trop vastes. La longueur de cette section varie considérablement d'un projet à l'autre.

Domaine métier

Le domaine métier apporte des précisions sur les règles applicables au projet. Ces règles sont décrites textuellement et font référence à des documents annexes officiels.

Des modèles de base de données constituent également un bon moyen de présenter les aspects métiers, surtout s'ils sont conceptuels (MCD).

Périmètre fonctionnel (généralement confondu avec l'expression de besoins)

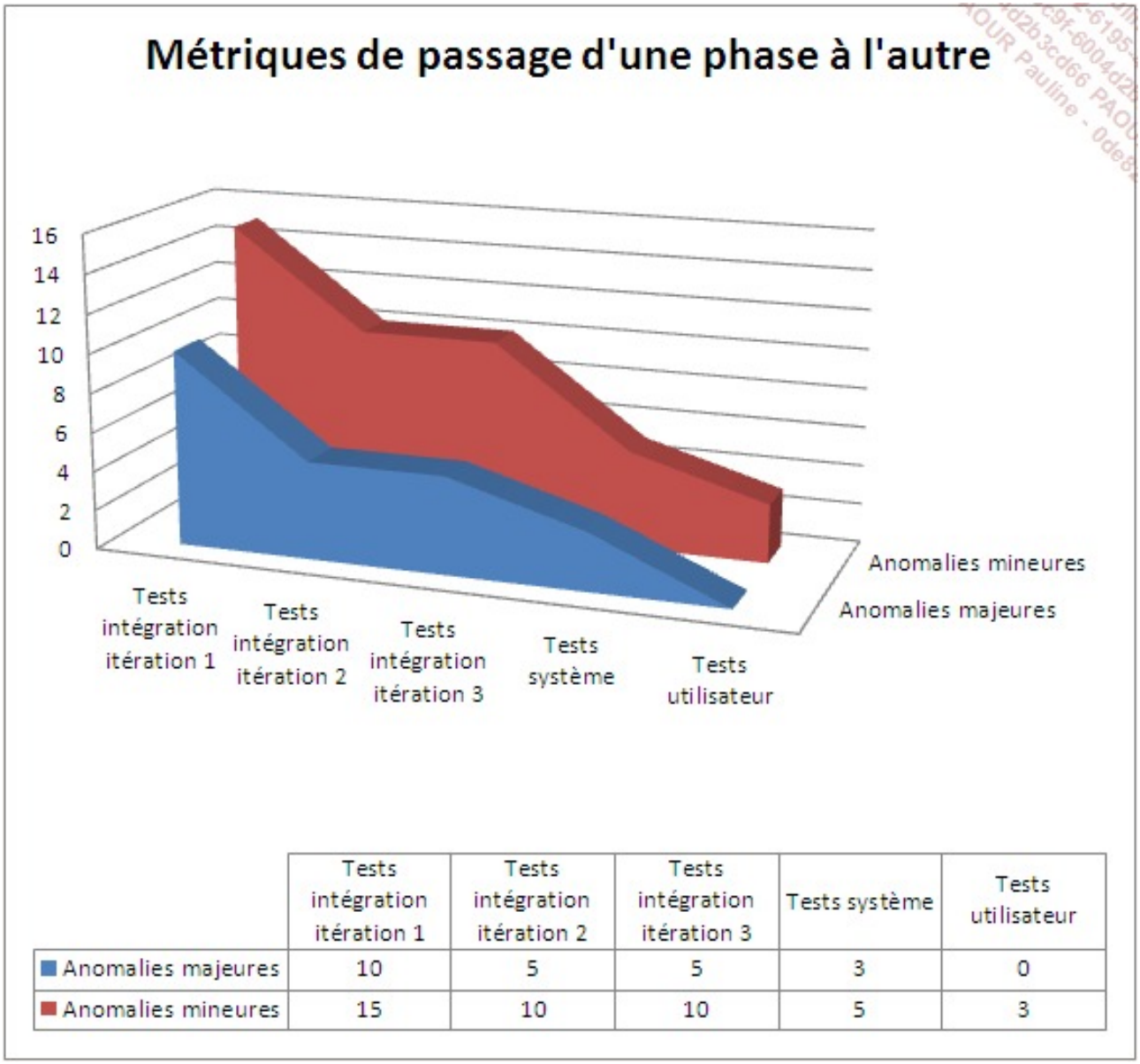
La description du périmètre fonctionnel s'appuie sur quantité de formalismes : diagramme des cas d'utilisation, scénarios, diagrammes de flux de travail, cartographies fonctionnelles…

Il n'est pas rare, au moment de l'élaboration du cahier des charges, d'avoir suffisamment de matière pour développer cette partie du cahier des charges. C'est une bonne démarche dans la mesure où elle clarifie souvent les responsabilités ; cependant, une certaine souplesse doit être conservée quant à son évolution au cours du projet. De plus, il ne faut pas trop anticiper les phases d'analyse pour ne pas s'orienter vers une solution ne répondant pas au besoin réel. Certaines parties peuvent donc être

délibérément esquissées alors que d'autres font l'objet d'une description beaucoup plus poussée.

Critères d'évaluation de la qualité, critère d'achèvement

On peut conditionner, pour les projets au forfait, le passage d'une phase à l'autre par un décompte d'anomalies ouvertes classées par sévérité. Cela fournit une métrique pour l'achèvement des tâches de développement et permet de contrôler la maturité de la solution jusqu'à un niveau de qualité adéquat.



	Tests intégration itération 1	Tests intégration itération 2	Tests intégration itération 3	Tests système	Tests utilisateur
Anomalies majeures	10	5	5	3	0
Anomalies mineures	15	10	10	5	3

Conditions opérationnelles

Ces conditions listent les modalités d'intégration, d'exploitation, de maintenance, de sûreté de fonctionnement, de supervision des applications ciblées par le projet. L'expression de ces conditions se fait à partir de valeurs moyennes cibles sur des KPI (SLA, *Service Level Agreement*).

Méthodologie

Une méthodologie de développement ou d'analyse peut parfaitement figurer dans le cahier des charges. En général, il ne s'agit pas d'une recommandation de la maîtrise d'ouvrage mais plutôt d'un choix rendu public du maître d'œuvre.

Planning et éléments financiers

Dans le cadre d'une expression de besoins ou d'un appel d'offres, le planning général se limite habituellement aux bornes du projet, date de démarrage, date de « lancement » (*go live*). Le budget n'est pas toujours communiqué.

Dans le cadre d'une réponse à appel d'offres ou d'une proposition, le macroplanning est un engagement du maître d'œuvre. L'estimation budgétaire n'est souvent communiquée qu'à la toute fin des discussions entre les parties.

Responsabilités et garanties

Elles encadrent les limites d'action du maître d'œuvre et les exigences réciproques des deux parties.

Élaborer un cahier des charges

Un cahier des charges est une synthèse de travaux d'analyse préparatoire. On peut donc organiser une étude des besoins fonctionnels, du métier concerné, de l'architecture technique sous-jacente à l'aide des formalismes et méthodes présentés au début de ce chapitre. Puis, le chef de projet assemble ces éléments et propose des séances de relecture commune pour confronter les points de vue et stabiliser le concept de la solution. Il cherche à dégager une vision partagée et non à aboutir à un consensus.

Il introduit parallèlement les composantes organisationnelles du futur projet et celui-ci est chiffré (estimé) autant que possible.

Même si elle n'est pas annexée au cahier des charges, l'analyse des risques est une étape importante car elle donne la lisibilité nécessaire pour négocier avec les services et

prestataires prenant part au projet.

Les référentiels de spécifications

Une fois le projet lancé, le cahier des charges est progressivement appréhendé par l'équipe et donne lieu à deux documents de spécifications : les dossiers d'intégration fonctionnelle et technique.

Le dossier d'intégration fonctionnelle (DIF)

Ce dossier est maintenu conjointement par la maîtrise d'ouvrage (MOA) et la maîtrise d'œuvre (MOE). Il recense l'ensemble des processus fonctionnels réellement supportés par le logiciel. Chaque processus est détaillé de la façon suivante :

- Acteurs.
- Étapes du processus.
- Conditions de déclenchement.
- Résultats attendus.

Les interfaces avec d'autres logiciels figurent également dans le DIF, elles sont présentées d'un point de vue utilisateur.

Le dossier d'intégration technique (DIT)

Le DIT se concentre sur l'intégration technique de la solution au sein du système d'information. Il détaille les éléments suivants :

- L'architecture applicative du logiciel (autant qu'elle puisse être communiquée au client).
- L'architecture du système s'il est composé de plusieurs applications intégrées les unes aux autres.
- La liste des interfaces techniques avec leurs spécifications (données, formats, modalités de déclenchement).
- Les procédures d'exploitation, de supervision, d'administration, de mise à jour.

2. Le cadrage et les spécifications générales

L'expression de besoins et le cahier des charges traduisent les attentes de la maîtrise d'ouvrage, c'est-à-dire du demandeur du projet. La phase de cadrage a pour objectif de découvrir, clarifier et prioriser ces attentes pour obtenir des spécifications générales réalisables, c'est-à-dire pour lesquelles la maîtrise d'œuvre est capable de s'engager. C'est donc une version adaptée du cahier des charges, partagée entre MOA et MOE.

Par rapport au cahier des charges, les spécifications générales comportent nécessairement une description de la solution à mettre en œuvre. L'élaboration de ce document est généralement menée par des spécialistes (*product owner*, architecte, assistant à maîtrise d'ouvrage) chargés de dialoguer avec les commanditaires du projet.

La phase de cadrage est souvent réalisée par itération (découverte, formalisation, revue, validation) et par domaine d'intérêt fonctionnel ou technique.

3. La conception détaillée

Il s'agit d'un document indispensable à une implémentation maîtrisée du code. Chaque fonctionnalité est détaillée tour à tour :

- Acteurs et objectifs opérationnels.
- Écrans.
- Règles de validation des entrées utilisateur.
- Règles de gestion (logique métier).
- Enchaînements d'écrans.
- Exceptions.
- Bases de données : tables, taille des champs…
- Diagrammes préparatoires tels que diagrammes de séquence UML, de classes…

Il est important de relire périodiquement les spécifications détaillées et de les actualiser au cours du développement.

4. La réalisation

a. Gestion du code source

Quelle que soit la technologie utilisée, langage de programmation, plateforme intégrée, serveur d'application… le code source est un actif fondamental du projet et doit être l'objet de toutes les attentions.

Les solutions de gestion du code source garantissent :

- le développement collaboratif (accès sécurisé et simultané au code source),
- l'archivage sécurisé du code source et la mise en historique de toutes les modifications,
- la duplication du code sous forme de branches (versions parallèles),
- la gestion des tâches de développement en associant demande (ticket) et modification de code source.

Des solutions très performantes (Azure DevOps, Jira, GIT) de gestion du code source sont disponibles sur le marché. Les chefs de projet sont de plus en plus amenés à organiser les activités au travers de ces solutions.

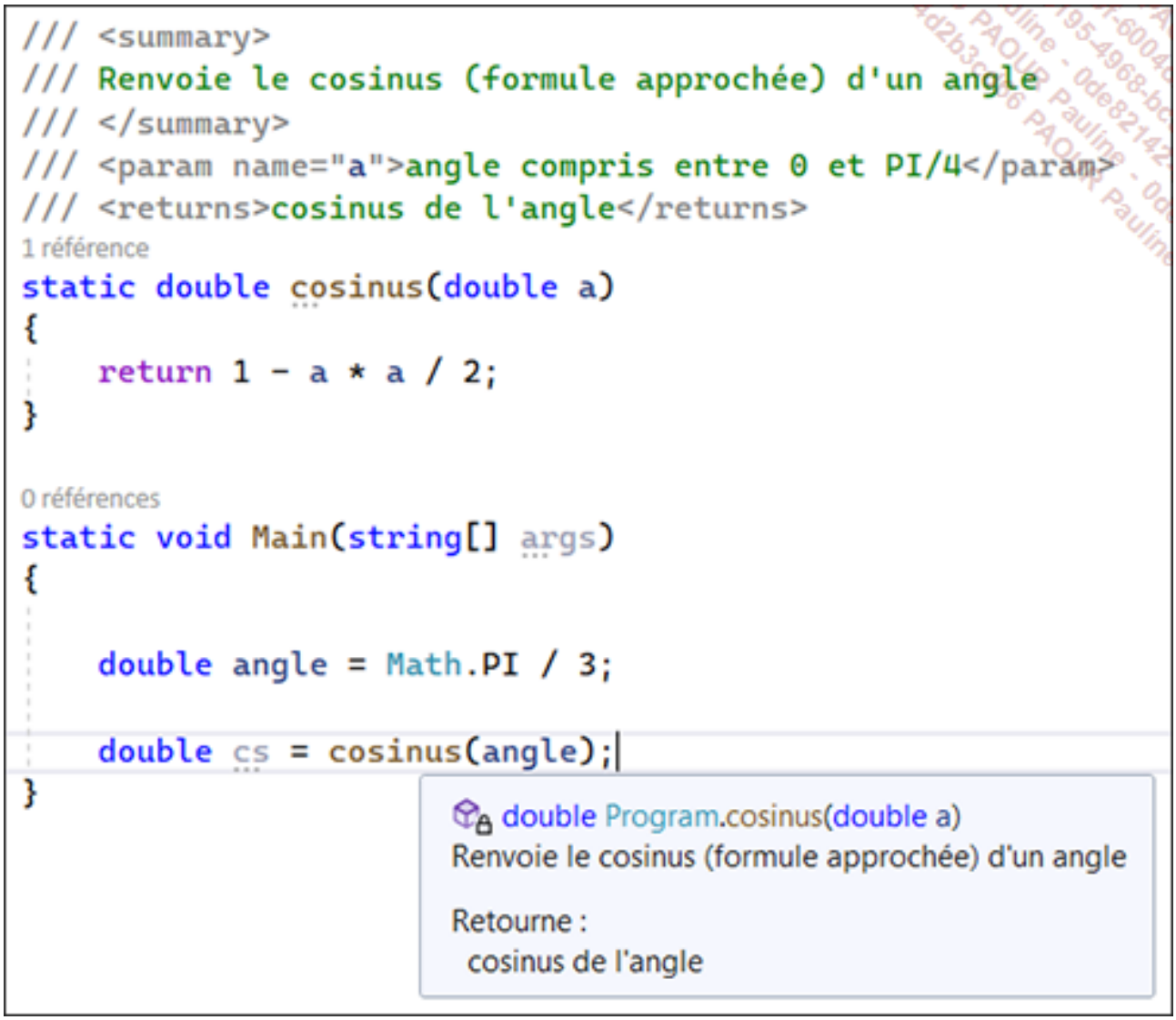
b. Gestion de la documentation liée au codage

Les commentaires

La documentation et les commentaires sont des éléments indispensables, d'égale importance au code lui-même. À vrai dire, les commentaires devraient au moins être rédigés conjointement aux instructions, voire les précéder, car ils expriment la pensée du programmeur dans un style moins formel et moins contraignant qu'un langage de programmation. Les commentaires font partie intégrante du code et n'ont réellement de sens qu'au sein de celui-ci. On peut les comparer à des notes de bas de page dans un document plus littéraire.

La documentation générée

Les compilateurs récents disposent de balises spéciales intégrées comme des commentaires ; elles sont utilisées par les info-bulles et servent aussi à la génération des documentations de références (C# doc, Java doc...).



La documentation Java, C# ou VB.NET est générée automatiquement en compilant l'ensemble des balises de description des classes, méthodes, attributs et packages… Elle est matérialisée par un ensemble de pages web au format HTML ou XML transformé et devient ainsi consultable par un navigateur web.

La documentation intégrée au programme oblige le développeur à apporter des éclaircissements sur son implémentation, à rationaliser le code avant de le livrer, voire à procéder à un minimum de tests unitaires. Pour le langage Java, chaque classe nommée figure dans un fichier distinct. Il est alors logique de démarrer par une documentation sur le rôle de la classe. Le cas des autres langages est un peu différent, car leur syntaxe n'impose pas le même découpage. Le chef de projet doit insister sur l'opportunité d'une telle organisation et inciter les développeurs à ranger les classes dans des fichiers distincts et des répertoires qui correspondent à l'espace de noms (au *package*).

Les méthodes sont également des points d'entrée importants qu'il convient de documenter. Les conventions de nommage des paramètres ne sont pas toujours respectées et, dans bien des cas, insistent plus sur la nature des arguments que sur leur rôle.

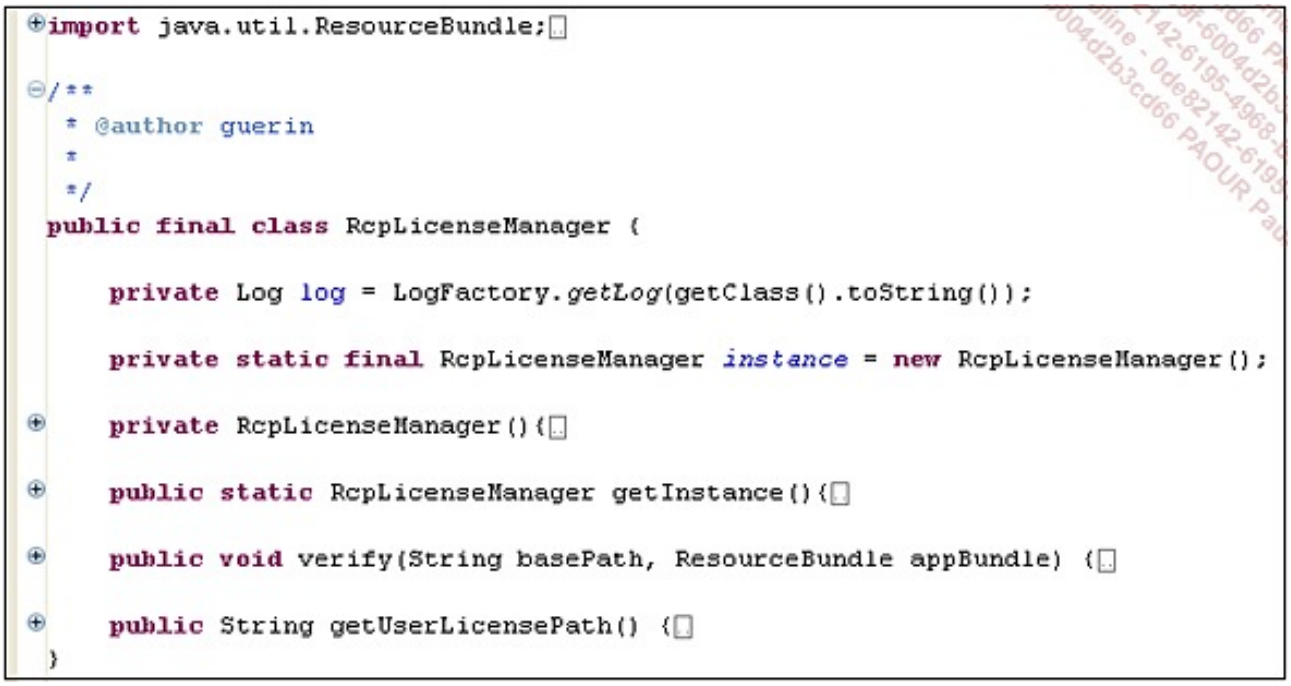
La documentation intégrée pallie ce problème en donnant un moyen au programmeur de décrire le rôle et l'utilité de chaque paramètre. Le type de retour de la méthode et les exceptions levées sont aussi explicités par ce mécanisme.

Voici les balises (*tags*) utilisées pour décrire une classe ou une méthode :

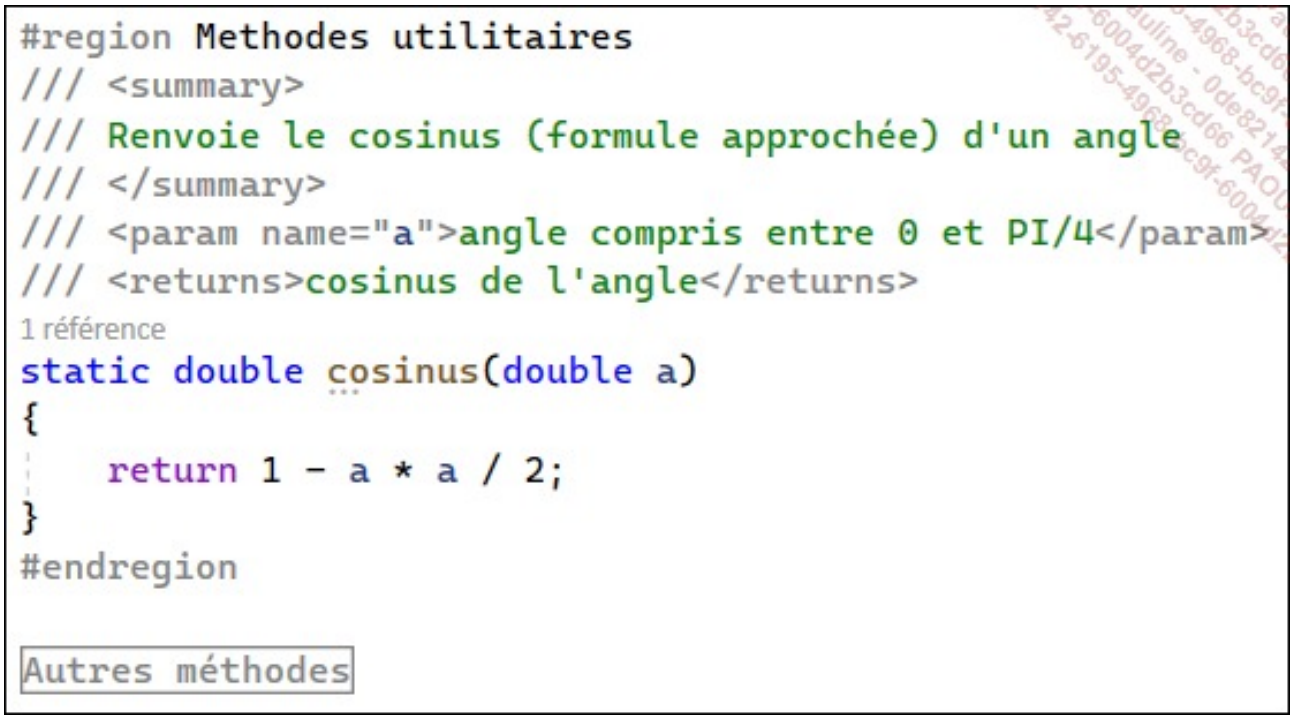
Java	.NET	Rôle
`Texte libre`	`<summary>`	Donne un résumé de l'élément.
`Texte libre`	`<example>`	Indique un exemple de mise en œuvre.
	`<permission>`	Indique le jeu d'autorisations associé à l'élément.
`@throws`	`<exception>`	Énumère les exceptions levées par la méthode.
`Texte libre`	`<remarks>`	Indique une remarque.
`@see @seealso`	`<see> <seealso>`	Fournit des correspondances dans la documentation.
`@author`	`Texte libre`	Auteur du fragment de code.
`@deprecated`	`[Obsolete]`	Attention, dans .NET il s'agit d'un attribut (pas d'une notation de la documentation).
`@return`	`<returns>`	Valeur retournée par la méthode.
`@param`	`<param>`	Paramètre de la méthode.

Les régions

Les ateliers logiciels EDI (environnement de développement intégré) récents analysent la syntaxe à la volée et proposent au programmeur une fonction de masquage des détails ; les instructions d'une méthode peuvent ainsi être cachées pour que l'on puisse visualiser plus aisément la structure d'un module.



Le logiciel Visual Studio dispose, comme Eclipse, de ce mécanisme. Il propose également au développeur de marquer lui-même les zones condensables, que l'on appelle des régions :



```
#region Methodes utilitaires
/// <summary>
/// Renvoie le cosinus (formule approchée) d'un angle
/// </summary>
/// <param name="a">angle compris entre 0 et PI/4</param>
/// <returns>cosinus de l'angle</returns>
1 référence
static double cosinus(double a)
{
    return 1 - a * a / 2;
}
#endregion

Autres méthodes
```

En définissant des régions à l'intérieur d'une fonction, celles-ci peuvent avantageusement se substituer aux commentaires ; la relecture d'un algorithme conséquent s'avère alors nettement plus simple.

c. Les revues de code

La revue de code est une relecture d'un programme ou d'un module dont l'objectif est de contrôler sa qualité et de juger de sa fiabilité. Le chef de projet doit régulièrement procéder à des revues pour s'assurer du respect des normes de développement, du niveau d'avancement des tâches impliquant de la programmation, mais aussi pour détecter des erreurs de codage.

Définir les règles de codage

La majorité des développeurs appliquent constamment leurs propres règles de codage, et on peut remarquer un certain systématisme dans leurs réalisations, alors même qu'elles ne respectent aucune « bonne pratique » ou convention d'écriture. Il est donc fondamental d'indiquer quelles sont ces conventions pour que les développeurs puissent s'y conformer.

Utiliser des outils de contrôle durant le codage

Très souvent, la directive ne suffit pas et il est intéressant de recourir à une liste de

contrôle (*check-list*) que tout un chacun déroule avant d'archiver sa production (*check in*, *commit*, *push*) :

1. Toutes les classes sont commentées avec ///.

2. Toutes les méthodes sont commentées avec /// (y compris les méthodes utilitaires) et les commentaires explicitent le rôle de la méthode.

3. Ajouter des commentaires au sein des méthodes pour décrire l'implémentation des algorithmes.

4. Chaque classe mentionne le nom du développeur et l'historique des modifications.

5. Séparer les blocs d'instructions par des lignes vides. Le bloc débute par un commentaire suivi des instructions correspondantes.

6. Donner une explication au rôle de chaque paramètre sous forme de commentaire ///, sans se contenter d'indiquer le nom du paramètre.

7. Tous les membres d'une classe sont entourés de régions.

8. Les méthodes complexes sont partitionnées en région.

9. La partition d'une classe en région suit la logique fonctionnelle et non la logique technique (éviter le découpage suivant : constructeurs, données, événements, méthodes…).

10. Pas de code mort archivé.

11. Éviter les logiques à trois niveaux ou plus de boucles / tests (*while if while*…).

12. Éviter le code dupliqué ; mais ne créer des méthodes génériques que si cela s'avère fiable et utile.

13. Effacer les directives d'importation inutiles.

14. La compilation ne doit pas générer d'avertissement.

15. Utiliser l'outil de couverture de code pour détecter des incohérences.

16. Toutes les chaînes/labels sont répertoriées dans des fichiers de ressources.

17. Aucune URL en dur, utiliser les méthodes adéquates pour les obtenir.

18. Pas de logique fonctionnelle dans les composants entités.

19. Utiliser des diagrammes de séquence pour former les classes.

20. Éviter les tautologies (si a est vrai = vrai alors).

21. Définir des propriétés ou des accesseurs pour élaborer des logiques de contrôle.

22. Utiliser la convention d'écriture PascalCase (chaque mot d'un identifiant composé débute par une majuscule).

…

Il est vrai que les référentiels de code source tels que Team System peuvent appliquer à la volée une analyse de couverture de code et bloquer l'opération d'archivage, si celui-ci n'est pas conforme. Cette pratique s'avère toutefois délicate à activer dans le cas de la reprise d'un code très loin de satisfaire ces conditions.

Ces démarches responsabilisent le développeur et l'incitent à réfléchir avant de « produire de la ligne ». Progressivement, le responsable technique doit accompagner son équipe non pas pour qu'elle se conforme simplement à cette liste (non exhaustive) mais plutôt pour qu'elle acquière la méthode nécessaire pour y parvenir. C'est une différence capitale entre un simple exécutant et un développeur plus avisé.

Préparer une revue de code

Quel est le bon moment pour une revue de code ? Assurément avant une livraison, autrement l'équipe technique se concentrera légitimement, mais en même temps avec peine, sur la résolution des bugs plutôt que sur sa refactorisation. Le chef de projet choisit un module ou une classe « au hasard » avant la revue de projet et il consigne l'ensemble des anomalies. Il y a statistiquement toujours le même nombre d'anomalies par unité de ligne de code, et cette valeur évolue au cours du temps en fonction de la maturité du projet et de celle de l'équipe.

Effectuer une revue

La revue de code est une activité qui doit rester pédagogique. Il ne s'agit pas de stigmatiser le fruit de telle ou telle personne mais plutôt de responsabiliser l'équipe dans son ensemble pour atteindre un niveau de qualité suffisant. Dans le cas d'équipes peu

coutumières des revues de code, quelques exemples de débogage sur du code mal conçu illustreront à quel point le temps passé à respecter les règles de développement est un temps gagné et non perdu.

5. Les tests et les corrections, l'assurance qualité

Nous avons déjà évoqué l'image que dégagent les tests dans la perception d'un projet ; d'aucuns affirment qu'ils ne sont pas assez nombreux, ou que la plage qui leur est réservée est trop courte.

En réalité, c'est souvent la méthodologie de test qui fait défaut. Nous allons par conséquent découvrir les différentes catégories de tests et leur rôle dans la méthode de test d'un logiciel.

Pour redorer le blason d'une activité souvent mésestimée (test et correction), le vocabulaire a évolué en parlant d'assurance qualité (*Quality Assurance* ou QA en anglais).

6. Les types de tests

Il existe de nombreux protocoles de tests indiqués dans les vérifications d'éléments plus ou moins grands, comme une microfonctionnalité jusqu'à la validation de processus complets (tests de bout en bout).

a. Tests unitaires

Les tests unitaires ont pour objectif de valider le fonctionnement des modules (ou composants) logiciels pris indépendamment les uns des autres. Un module s'assimile à une fonction qui admet des paramètres et fournit de manière déterministe un résultat.

Ces tests sont le plus souvent automatisés tant leur déroulement est simple : l'automate « instancie » le composant à tester, le stimule avec des valeurs choisies à l'avance et compare le résultat avec une valeur théorique, également déterminée à l'avance. Une règle (égalité stricte, inégalité stricte ou large) décide de l'issue du test.

L'environnement Eclipse a popularisé le framework de tests unitaires JUnit, lequel a été porté vers d'autres environnements sous le nom NUnit. Pour sa part, la version Team tester de Visual Studio propose aussi un mécanisme de tests unitaires au fonctionnement relativement comparable à JUnit.

b. Tests d'intégration

Les tests d'intégration valident le fonctionnement d'un sous-système lorsque ses composants sont assemblés les uns aux autres. En principe, ils suivent les tests unitaires et par conséquent ils ne cherchent pas forcément à contrôler la validité qualitative des résultats.

L'assemblage (ou l'intégration) de composants est d'abord un mélange de technologies : C++, Java ou .NET, SQL, XML… Les situations de blocage, d'attente, de contre-performance sont nombreuses et c'est précisément l'objet des tests d'intégration que de les découvrir.

Voici les thématiques développées par ces tests :

Thématique	Exemple de tests
Connectivité	Le serveur SQL peut se connecter au logiciel. Le module de facturation peut être intégré à l'application web.
Sécurité	Le sous-système s'exécute dans les conditions de sécurité de l'environnement de production. L'application démarre l'impression sans nécessiter les droits « administrateur ».
Performance	Le composant de génération des rapports tient la charge avec plus de mille requêtes à l'heure. Le composant de login valide une connexion en moins de 0,1 seconde.
Installation	La base de registre est accessible au module de persistance.
Environnement	Le sous-système de cryptage fonctionne sous Linux et sous Windows.

c. Tests fonctionnels

Les tests fonctionnels (ou *system tests* en anglais) cherchent à vérifier le bon déroulement des processus implémentés dans le logiciel. Ils sont généralement confiés à des utilisateurs, des analystes, des personnes n'ayant pas eu directement en charge le développement desdits processus.

Boîte noire

Dans ce type de test, le logiciel est employé dans les conditions d'un utilisateur qui n'a aucune idée de la façon dont il est implémenté. Le test porte donc sur les fonctionnalités

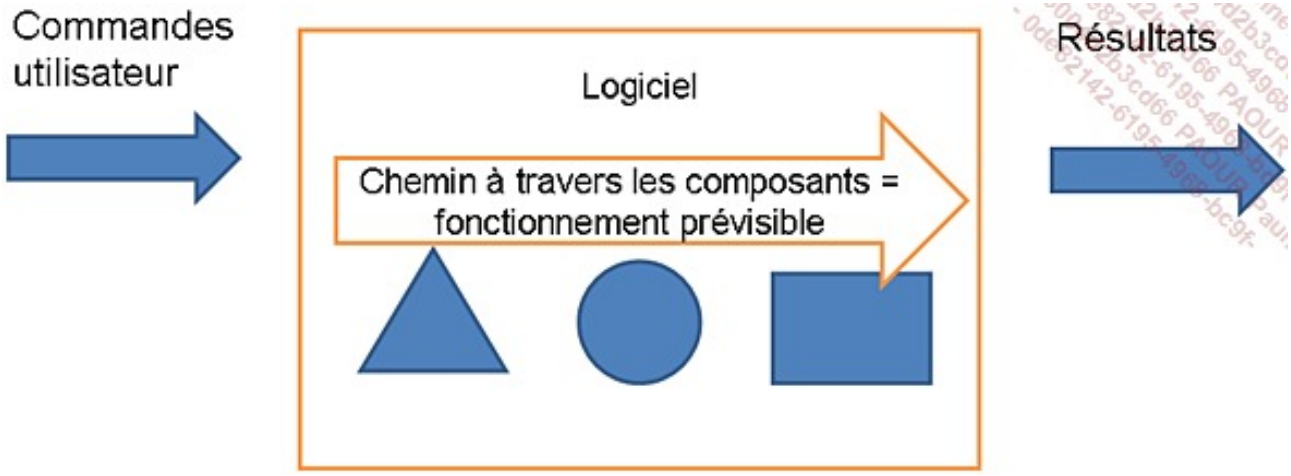
sans a priori technique.

Ce type de test est indiqué lorsqu'il faut valider le fonctionnement des processus sans chercher d'explications techniques ; il fait partie des tests d'acceptation utilisateur (*User Acceptance Test*, UAT) et figure généralement dans le cahier de recette du projet.

Boîte blanche

Dans le cas d'une boîte blanche, on connaît le fonctionnement interne des composants supportant l'exécution des processus à tester. Ces informations permettent de stresser le système pour vérifier que son comportement reste conforme aux prévisions. Dans le cas où le comportement deviendrait non déterministe, ou différent de ce que le fonctionnement nominal laissait présager, le système n'est plus « une boîte blanche ».

Ces tests sont donc conduits par des informaticiens, parfois aidés par les analystes. Les tests de boîte blanche font partie des tests fonctionnels (ils contrôlent des processus entiers) mais les a priori techniques servent à guider des recherches de dysfonctionnement.



d. Le banc d'essai (benchmark)

Ces tests contrôlent le système logiciel dans ses conditions d'exploitation, en faisant varier le volume des données à traiter et le nombre d'utilisateurs simultanés.

Tests de charge et de performances

Les tests de charge vérifient le comportement du système lorsque le nombre d'utilisateurs nominal est atteint. Les transactions portent sur des volumes de données pertinents vis-à-vis des scénarios d'utilisation ciblés pour la production.

Ces tests, s'ils s'avèrent négatifs, mettent en lumière une intégration défectueuse ou, plus grave, une conception technique inadaptée.

En matière de complexité algorithmique, on mesure généralement la contre-performance sous forme d'une expression appelée O. Ceci donne des classes de complexités O(n), O(log(n)), O(n^2)… Pour le concepteur d'un algorithme, il vaut mieux s'en tenir aux performances minimales pour déterminer ce que le module peut donner dans le pire des cas. Toutefois, à l'échelle d'un système, il est parfois réclamé, dans la configuration de production et avec des scénarios nominaux, d'obtenir une certaine performance. Cette performance s'exprime alors sous la forme de temps de réponse, de nombre de transactions par seconde, de vitesse d'affichage…

Les tests de performance servent également à étalonner un logiciel qui doit subir des évolutions technologiques. Il est alors attendu que le nouveau système aura des performances équivalentes à l'ancien.

Tests de stress et de montée en charge

Les tests de stress suivent le comportement du système lorsqu'un grand nombre de transactions est subitement initié. Il s'agit d'une variante des tests de montée en charge qui s'assurent de la qualité de service lorsque le niveau de transactions simultanées augmente « normalement ».

Ces types de tests sont utiles pour paramétrer le logiciel et proposer des gammes de fonctionnement : normal, prioritaire, dégradé… Le système hôte doit toujours garder une réserve pour éviter le déni de service (*Denial Of Service* ou DOS) aux conséquences parfois catastrophiques.

Tests d'endurance

Il s'agit de tests contrôlant l'état du système lorsqu'il est soumis à une utilisation « en charge » prolongée. Les ressources mémoire et système disponibles peuvent ainsi s'éroder et rendre le système hôte instable, voire provoquer un DOS et un crash.

Tests aux limites

Ces tests déterminent comment se comporte le système soumis à un nombre d'utilisateurs supérieur et à des demandes de transactions supérieures aux valeurs nominales. Ils permettent de définir la marge de sécurité avant que le système ne devienne instable.

e. Tests d'acceptation utilisateur (UAT)

Il s'agit de tests réalisés par les futurs utilisateurs (en général des membres de l'équipe projet et des utilisateurs pilotes) sur un système fonctionnant en condition de production. Ces tests sont très importants car ils conditionnent généralement la mise en service du système avec un nombre d'anomalies maximum.

f. Cycles de tests

Les cycles de tests sont des phases du projet pendant lesquelles on ne développe plus de nouvelles fonctionnalités mais qui donnent lieu à des compilations (*build*) régulières.

Un cycle de tests est un processus qui s'appuie sur une version du logiciel et qui comporte les étapes suivantes :

Planification des tests à effectuer	Le cycle concerne une sélection de tests parmi tous ceux qui peuvent être développés. ˘ La méthodologie « ascendante » ou « descendante » peut cibler certains modules et de ce fait exclure du cycle certains tests. ˘ L'accent peut être mis sur certains modules critiques.

Tests	Les tests sont effectués sur la version courante. Les anciennes versions servent de référence pour traquer la régression.
Évaluation des bugs à corriger dans la prochaine version	Les bugs sont identifiés et les qualités suivantes sont évaluées : ˘ Sévérité (majeur, mineur, bloquant, anodin…). ˘ Reproductibilité. ˘ Catégorie fonctionnelle (module, processus…) ou technique (configuration, compatibilité système…). ˘ Existence dans une version antérieure.
Clôture du périmètre de la prochaine version	Parmi tous les bugs identifiés, une partie seulement doit être corrigée dans la version suivante. L'autre partie sera abordée par des versions ultérieures, de manière à optimiser l'engagement de l'équipe de développement sur la résolution des bugs critiques pour avancer le plan de tests.
Évaluation du temps de correction	L'équipe de développement en charge de la nouvelle version doit procéder à une estimation avant même de débuter les corrections. Ainsi, le chef de projet peut organiser la suite des opérations.
Correction	Pendant les corrections et la préparation de la nouvelle version, les tests continuent sur la version actuelle.
Installation de la nouvelle version	À la livraison de la nouvelle version, un nouvel environnement est créé et les notes de version (*release notes*) sont étudiées. Les bugs corrigés sont indiqués afin de procéder aux vérifications.

g. Plans de tests

Le plan de tests organise la phase de tests d'un logiciel. Il cherche à gagner la convergence, c'est-à-dire le moment où l'on corrige plus de bugs que l'on en découvre de nouveaux.

Le plan de tests est indissociable d'un planning qui délimite les opérations dans le temps. Il paraît toujours difficile d'augurer de la qualité du logiciel issu du développement et de fournir une estimation valable. Pourtant, cette estimation est indispensable si l'on veut conserver la mobilisation de l'équipe projet et aboutir dans de bonnes conditions.

Il faut d'abord déterminer la méthodologie d'ensemble ; celle-ci peut dépendre du modèle de développement qui impose ou non la mise en œuvre de tests d'intégration et de tests fonctionnels à chaque itération. Va-t-on tester à chaque itération, à chaque cycle de tests, l'ensemble des fonctionnalités et processus du logiciel ? Il est probable que non, et les possibles retards engendrés lors du développement permettent rarement d'aborder les tests avec cette approche. Il va donc falloir se focaliser sur certaines parties du logiciel. Les méthodes de tests ascendantes et descendantes peuvent guider le chef de projet dans l'organisation de son plan de tests.

Sont ensuite détaillés les cas de test, autrement dit les tests eux-mêmes. Un cas de test doit impérativement être identifié (de manière non équivoque et unique) et rappeler le module ou le processus sur lequel il porte.

Cas de test n°	Identification du test.
Module ou processus	Partie du logiciel concernée par le test.
Type de test	Fonctionnel, intégration, technique, stress…
Préconditions	Séquence d'opérations à appliquer avant de démarrer le test. Jeu d'essai à utiliser.
Séquence	Opérations à dérouler pendant le test.
Résultats attendus	Mesures et affichages objectifs.
Durée estimée pour le test	

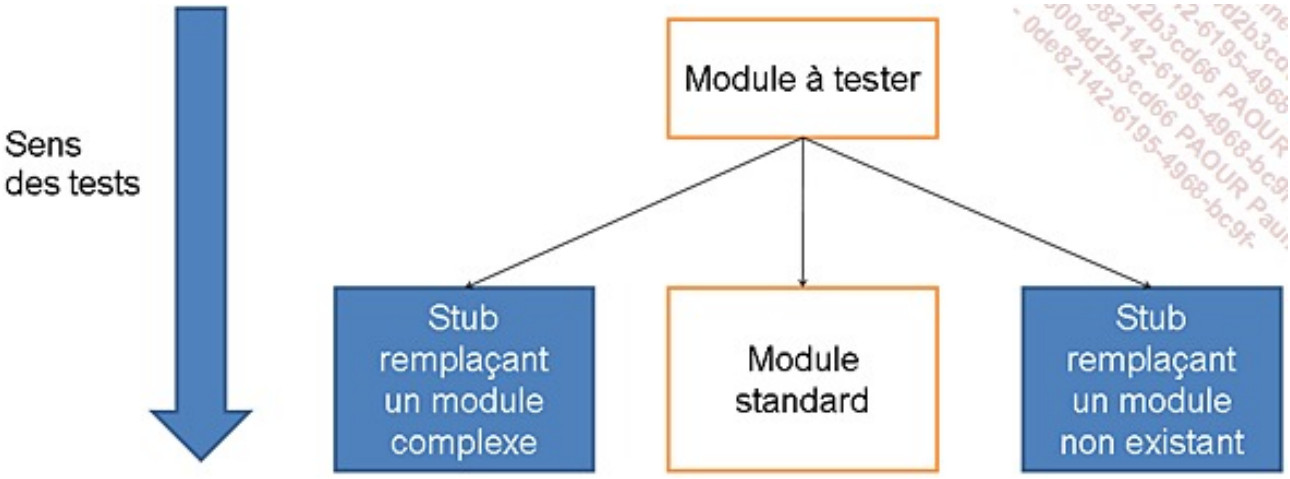
Les cas de test sont aisément saisis dans des feuilles de calcul. Ils peuvent également être déclinés dans différentes versions, et d'ailleurs être rattachés au référentiel de code source du projet.

Les jeux d'essai constituent les données représentatives sur lesquelles vont porter les tests. Leur mise au point réclame un temps non négligeable qu'il faut intégrer dans la planification de la campagne de tests.

Finalement, le chef de projet organise ses cycles de tests en fonction des remontées de bugs successives mais aussi des disponibilités fluctuantes de l'équipe projet. Il n'est pas rare, en effet, que les développeurs soient partagés entre les tests et les activités de correction, et qu'ils soient parfois moins disponibles à l'issue de la phase de développement (engagement sur d'autres projets, congés…). À chaque cycle, il faut déterminer le nombre potentiel de jours réservés aux tests, en additionnant la charge maximum de l'équipe de test. Ceci va donner une indication sur l'importance du périmètre testé et finalement influencer la fréquence des versions correctives.

h. Stratégie de tests descendants

Cette stratégie cherche à intégrer ou identifier des modules défectueux en partant « du haut », c'est-à-dire de l'interface graphique ou du module fonctionnellement le plus élevé. Les modules sous-jacents sont parfois remplacés par des souches (*stubs*) qui se substituent à des modules non encore développés ou bien non testés.

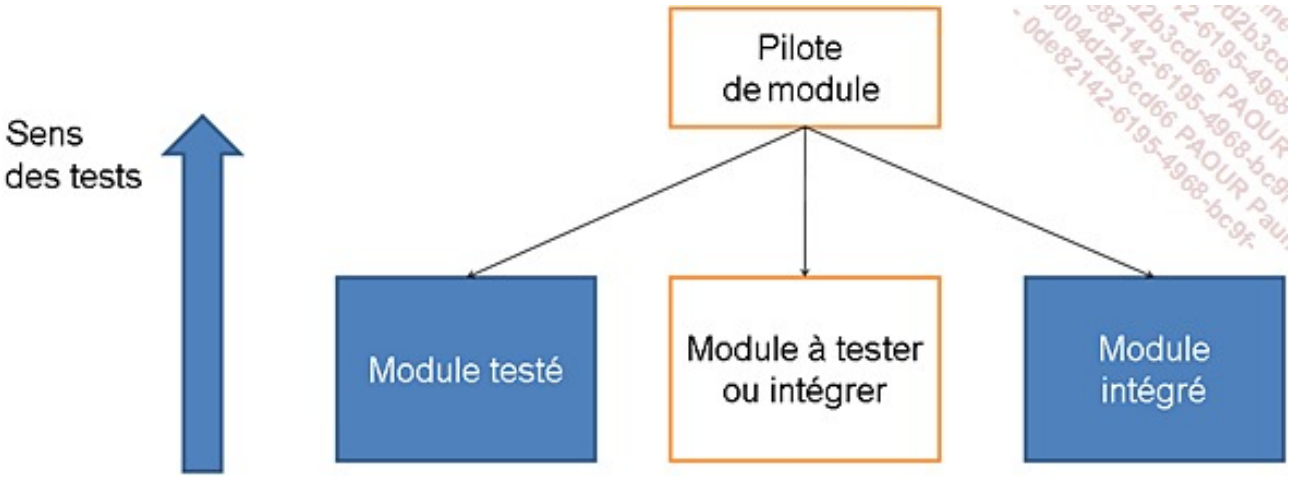


Les tests descendants s'appuient sur les processus utilisateur et permettent de découvrir quels maillons de la chaîne font défaut. Ils servent aussi à valider des interfaces de haut niveau (écrans, services…) avant de procéder au développement de l'ensemble de l'application.

i. Stratégie de tests ascendants

Les tests ascendants vont au contraire s'appuyer sur des modules existants pour remonter la chaîne fonctionnelle. Cette technique est naturellement employée par les développeurs en phase de tests unitaires ou dans les premières étapes de l'intégration.

Dans cette approche, les modules sont sollicités par des pilotes (*drivers*) aux fonctionnalités souvent limitées. C'est le cas notamment des frameworks de tests unitaires JUnit.



j. Tests de non-régression

La régression apparaît lorsque la correction d'un bug fait ressurgir des bugs préalablement résolus. Les tests de non-régression déterminent l'absence de régression.

D'une manière générale, les cycles de tests doivent concerner des ensembles complets de tests, pour que l'équipe de développement puisse les aborder et les traiter dans leur globalité. Autrement, il est plus que probable que la correction d'un bug isolé déclenche une série de problèmes déjà traités.

Comme il est presque impossible de tester l'ensemble de l'application à chaque cycle de tests, la régression est mieux traquée lorsque les tests sont automatisés. Les frameworks de tests unitaires sont ainsi très utiles pour piloter a minima des tests fonctionnels automatisés. Il existe également des logiciels de test d'interface graphique mais leur coût apparaît souvent prohibitif. Cependant, cette situation devrait évoluer favorablement car l'offre s'étoffe d'année en année.

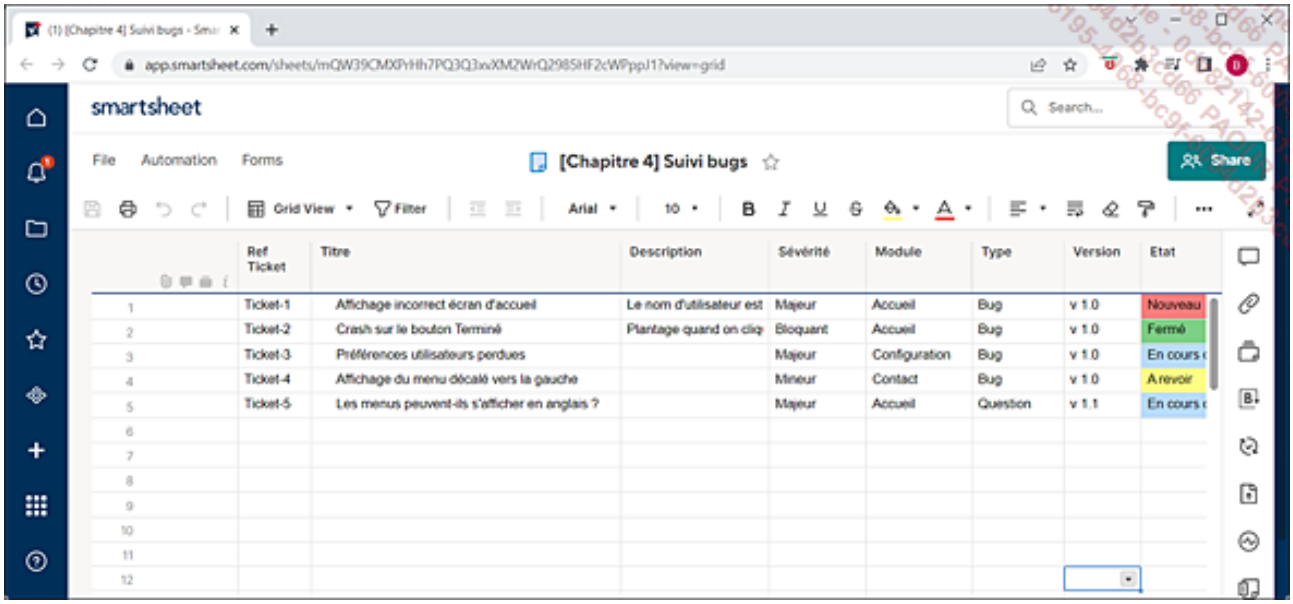
k. Suivi des anomalies

Depuis leur apparition dans des formules open source, les logiciels de suivi de bug (*bug trackers*) se sont imposés au sein de la panoplie du chef de projet. Ils permettent de partager les informations relatives aux bugs rencontrés et de faciliter leur reproduction (circonstances de détection).

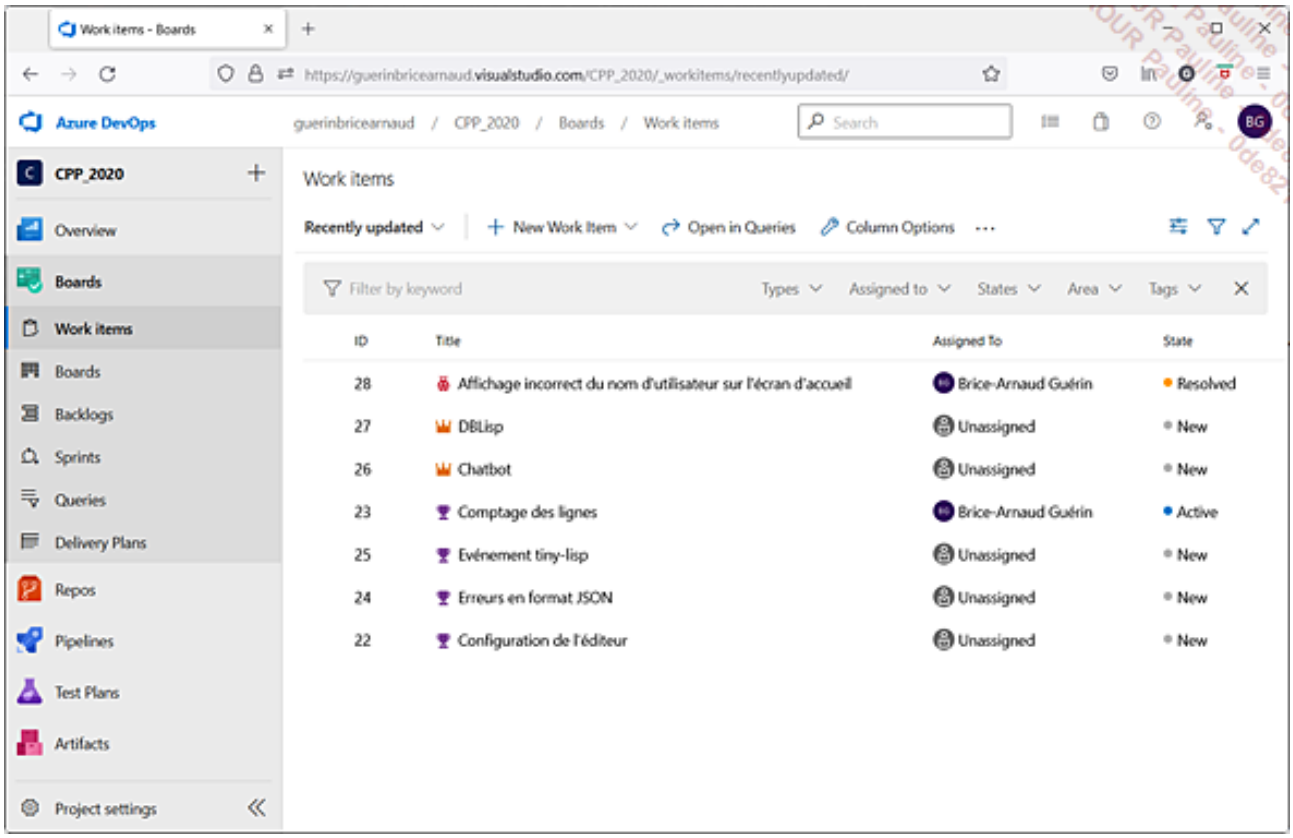
Les *bug trackers* servent également à établir des statistiques sur les versions en cours et à venir.

Le marché propose deux types d'outils de suivi d'anomalies : les outils autonomes comme Mantis ou Smartsheet, et les outils intégrés à une usine logicielle comme Azure DevOps ou Jira.

Les outils autonomes ont l'intérêt d'être utilisables avec n'importe quel type de projet et n'importe quelle technologie de développement. Ils sont caractérisés par une grande souplesse dans la configuration et ne nécessitent aucune compétence technique pour leur mise en œuvre. À titre d'illustration, la copie d'écran suivante est issue du logiciel Smartsheet :



Les outils intégrés comme Azure DevOps sont au contraire associés à un outil de développement (Visual Studio par exemple). Ils s'avèrent particulièrement efficaces dans le suivi des corrections puisque la résolution d'un ticket est associée à la modification d'une partie du code. Il est donc très facile de retrouver l'historique des corrections et d'identifier quelles parties du logiciel ont dû être modifiées pour apporter telle ou telle correction.



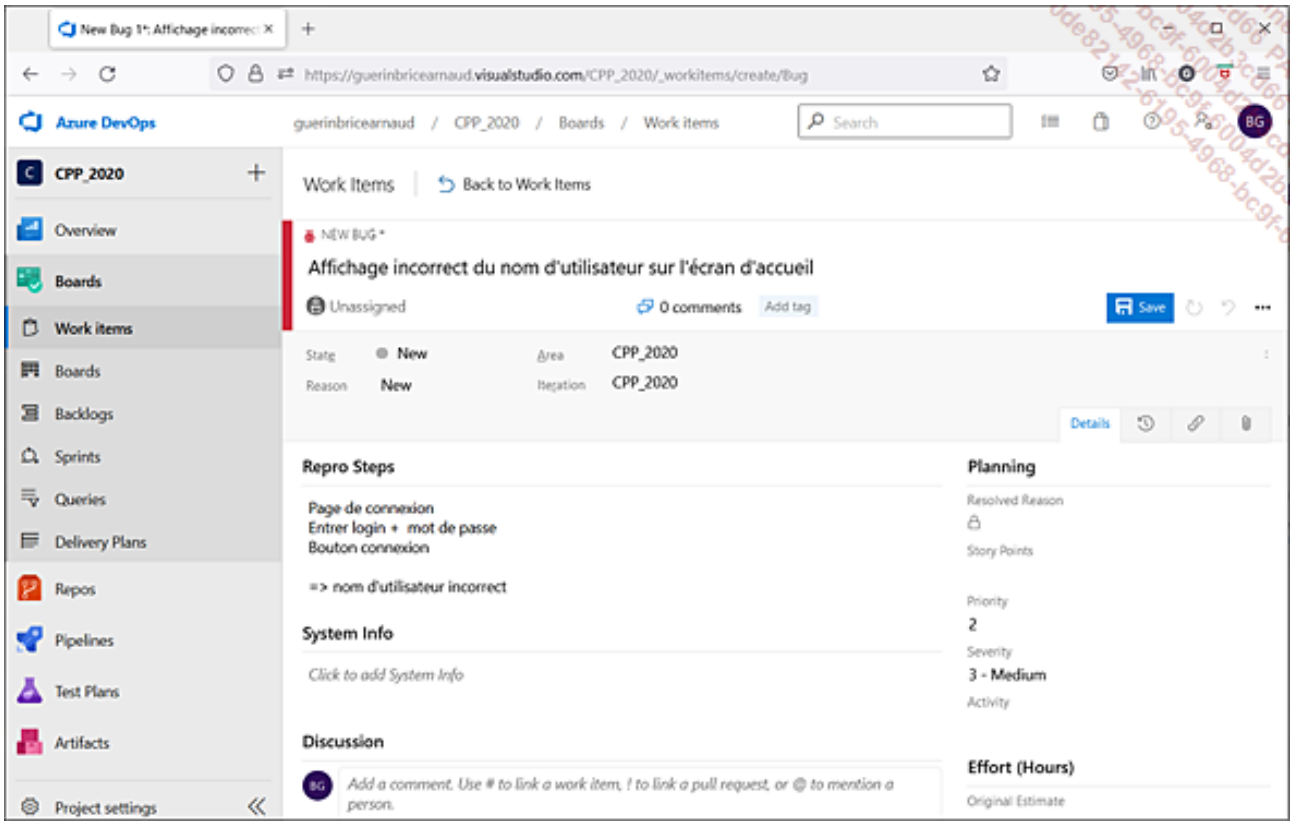
Création de projets et de versions

Les projets sont organisés en versions, et parfois aussi en sous-projets. Ces caractéristiques facilitent l'organisation de la campagne de tests en limitant la visibilité des bugs au périmètre du projet et en synchronisant les plans de tests avec les cycles de correction.

Enregistrement de défauts (bugs)

Pour plus de commodité dans un contexte international, l'interface graphique bascule dans la langue choisie par l'administrateur pour celui qui rapporte les bugs. Le formulaire présente presque toujours les mêmes champs, titre, sévérité, reproductibilité…

Des fichiers complémentaires (bases de données d'essai, fichiers d'exemples, copies d'écran) peuvent être joints à la fiche de défaut pour expliciter les circonstances de sa détection.



Établissement de rapports

Les outils de *bug tracking* proposent de nombreux rapports dont dépend la réalisation du plan de tests :

Progression de la résolution	Statistiques sur les temps de résolution. Nombre de bugs ouverts/résolus. Statistiques sur les fonctions (catégories) du logiciel affectées par des bugs.
Contrôle des versions et des affectations	Liste des bugs par version, par personne, par catégorie.

Pour être traitées, les fiches de défaut sont elles-mêmes exportables au format tableur ou traitement de texte.

7. Gestion des versions

Depuis la généralisation des projets liés à Internet, les cycles de développement se sont considérablement raccourcis. Tout du moins, il est fréquent de procéder en plusieurs étapes pour atteindre le but que l'on s'est fixé : c'est l'essence même des méthodes itératives.

Chaque étape correspond à une « version » d'un logiciel, ou *release* en anglais.

a. Production d'une version

Les numéros de version

Mis à part la mode des logiciels étiquetés par l'année de leur probable commercialisation (Visual Studio 2022, Symantec 2023...) et dont la forme peut varier, nombre d'utilisateurs et de développeurs se réfèrent plutôt au numéro de version. Ce numéro est généralement composé de deux segments, le majeur et le mineur.

Le premier nombre correspond à l'indice majeur de version du logiciel. Le suivant à l'indice mineur. En principe, les évolutions de l'indice mineur correspondent à des maintenances successives d'une version majeure et de ce fait n'introduisent pas de rupture dans le périmètre fonctionnel de l'application. Ainsi, les versions 10.1 et 10.2 d'un logiciel partagent vraisemblablement le même niveau de fonctionnalités (notion de compatibilité ascendante).

Cette identification est parfois complétée d'autres indices :

- Le *build number*.
- Le *build date*.

Ces indices ne sont pas standardisés ; l'un désigne le nombre d'assemblages du logiciel (ou bien le nombre de secondes depuis minuit au moment de l'assemblage, cela dépend) et l'autre indique le nombre de jours écoulés au moment de l'assemblage depuis une date de référence. L'identification des fichiers exécutables comporte ainsi une mention à quatre segments :

Assembly Name	Version
CrystalDecisions.Web	10.5.3700.0
CrystalDecisions.Web	10.2.3600.0
CrystalDecisions.Wind...	10.5.3700.0
CrystalDecisions.Wind...	10.2.3600.0
cscompmgd	7.0.5000.0
cscompmgd	8.0.0.0
CustomMarshalers	1.0.5000.0
CustomMarshalers	2.0.0.0
dao	10.0.4504.0
DTEParseMgd	9.0.242.0
EnvDTE	8.0.0.0
EnvDTE80	8.0.0.0
ESRI.ArcGIS.3DAnalyst	9.2.3.1380
ESRI.ArcGIS.3DAnalyst	9.2.0.1324
ESRI.ArcGIS.3DAnaly...	9.2.3.1380

Détail des numéros de version des DLL

Les release notes

Chaque publication d'une version est accompagnée d'une notice (*release note*) qui récapitule les éléments importants :

- Identification de la version.
- Périmètre et modifications (ou *change log* en anglais).
- Résultats des tests.
- Nature des éléments livrés (code source, package d'installation…).
- Procédures d'installation.
- Avertissements.
- Emplacement des packages livrés (FTP, disque partagé…).

Les *release notes* prennent des formes très variées. Il s'agit parfois d'un simple fichier texte, parfois de notes assez volumineuses. Elles sont généralement sauvegardées dans le référentiel de code source, accompagnées des livrables qui composent la version.

b. Montée de version

Chaque nouvelle version est le fruit d'une évolution du logiciel. Cette évolution concerne des corrections de bugs et de nouvelles fonctionnalités intégrées au logiciel. Il est fréquent d'assortir une version de scripts et d'assistants d'installation destinés à automatiser la mise à niveau des éléments de la solution : base de données, répertoires, fichiers de configuration…

La mise en œuvre de cet ensemble de dispositifs est appelée montée de version.

c. Livraison continue

L'intégration continue est une technique de fabrication de logiciel qui vise à produire le plus fréquemment possible des versions, qu'elles compilent ou pas d'ailleurs. Évidemment, l'objectif est de mettre automatiquement à la disposition de l'ensemble de l'équipe ces versions intermédiaires. C'est la raison pour laquelle cette section s'intitule Livraison continue !

L'intégration continue réclame une bonne organisation, une grande qualité dans le développement mais aussi des méthodes de test capables d'absorber le rythme des versions générées.

8. Le lancement

a. Industrialisation

L'industrialisation est souvent sous-estimée dans le plan de développement. Elle joue pourtant un rôle essentiel, car elle valide la configuration technique cible du logiciel. Sans cette phase, les logiciels s'avèrent inopérants une fois transplantés hors des postes de développement.

Il s'agit de définir toutes les gammes opératoires pour déployer et mettre en service le système logiciel, et autant que possible d'automatiser toutes ces procédures. Les outils disponibles dépendant largement des technologies utilisées et du type de configuration sur site (*on-premise*) ou depuis le Cloud.

Une indication peut toutefois s'avérer primordiale : la réalisation des activités liées à l'industrialisation peut représenter jusqu'à 40 % de l'effort du projet, et ce d'autant plus que les technologies employées impliquent de déployer des éléments sur serveur ou sur le poste de l'utilisateur. Dès lors, on comprend le bénéfice et le succès des solutions Cloud qui limitent le déploiement local de composants tout en automatisant la quasi-totalité des tâches liées à l'industrialisation. Naturellement, ce choix n'est pas neutre puisqu'une fois la solution configurée sur un Cloud, la migration vers un autre environnement représente également un coût non négligeable.

b. La mise en production

La mise en production (également appelée MEP par les développeurs) consiste à installer la solution logicielle dans son environnement d'exécution (en configuration de test, de préproduction, de production…). Toutefois, avec la généralisation des techniques de livraison continue, les développeurs utilisent plutôt le terme de changement (*change*) pour exprimer la mise en production de petites parties du système.

c. La mise en service

Une fois le système livré (c'est la mise en production), il faut le mettre en fonctionnement et servir les utilisateurs. La mise en service est une opération technique qui comprend l'enregistrement des utilisateurs, l'ouverture de droits d'accès… Cette étape s'accompagne volontiers d'un support à la prise en main du système, support composé de modules de formation, de documentations utilisateur et de séances de découverte. On désigne par *change management* cette étape de prise en main du système par les utilisateurs qui suit la mise en service.