

GENIE LOGICIEL

Approche Agile **SCRUM**

Cours dispensé par Richard Kalonji
Diplômé de Paris-Dauphine

Table des matières

1. Pourquoi un cours sur le génie logiciel ?	10
1.1. Confusion entre programmeur et développeur	11
1.2. Confusion entre programme et logiciel	12
1.2.1. Différences entre programme et logiciel : l'utilisateur	13
1.2.2. Différence entre programme et logiciel : les cas d'utilisation	13
1.2.3. Différence entre programme et logiciel : la complexité	13
1.2.4. Différence entre programme et logiciel : la taille du code source	14
1.2.5. Différence entre programme et logiciel : la responsabilité des dysfonctionnements	15
1.2.6. Différence entre programme et logiciel : la maintenance	15
1.3. Conclusion préliminaire	15
2. Naissance du génie logiciel (1968)	17
2.1. La crise du logiciel	17
2.2. Le génie logiciel	17
2.3. Une méthode : la méthode en cascade	18
2.3.1. L'écosystème du développement logiciel	19
2.4. Un paradigme : la conception procédurale	20
2.5. Un formalisme : l'algorithmique	20
2.6. Ingénierie logicielle	21
3. Constat d'échec (2000)	22
3.1. Exemples de projets en échec	23
3.2. Causes d'échec	23
3.2.1. Cause d'échec n°1 : Suivre un plan coûte que coûte	23
3.2.2. Cause d'échec n°2 : Estimer la durée d'un projet	24
3.2.3. Cause d'échec n°3 : Définir les besoins au début	24
3.2.4. Cause d'échec n°4 : Réaliser les tests à la fin	24
3.2.5. Cause d'échec n°5 : Raisonner au niveau procédure	25
4. Le génie logiciel aujourd'hui	26
4.1. Constat préliminaire	26
4.2. Repenser le génie logiciel	26
4.3. Remarques	26
4.4. Effet sur la réussite des projets	26
4.5. Ingénierie logicielle	27

5. Conclusion du chapitre	28
5.1. Pourquoi ce cours ?	28
5.2. Pour qui ce cours ?	28
5.3. Que retenir de ce chapitre ?	29
Chapitre 2.....	31
Une méthode : Agilité	31
1. Objectif du chapitre.....	32
2. Mauvaises pratiques du développement logiciel	32
2.1. Planification à long terme : le modèle de développement en cascade	32
2.2. Pourquoi le modèle en cascade ne fonctionne pas pour le développement de logiciels	32
2.2.1. Analyse des besoins au début	33
2.2.2. Conception puis codage	33
2.2.3. Intégration en fin de codage	33
2.2.4. Test après le codage	33
2.3. Une amélioration : le cycle en V	34
2.4. Documentation exhaustive	34
3. Bonnes pratiques du développement : agilité.....	35
3.1. Manifeste Agile	35
3.1.1. Les 4 valeurs de l'agilité (comparées à celles des méthodes prédictives)	35
3.1.2. Les 12 principes de l'agilité.....	35
4. Mise en pratique de l'agilité pour le développement	37
4.1. Cycle de développement itératif : casser la planification à long terme	37
4.1.1. Avantages du cycle itératif	38
4.2. Cycle de développement incrémental : profiter de la malléabilité du logiciel.....	38
4.2.1. Incrément : MVP (Minimum Viable Product) ou POC (Proof Of Concept)	38
4.2.2. Avantages du cycle incrémental	40
4.3. Collaboration client – développeur	40
4.4. Équipe	40
4.5. Intégration continue.....	41
4.5.1. Cycle d'intégration continue (DevOps).....	41
5. Auto-documentation	42
5.1. Modélisations UML	42
6. Exemples de méthodes agiles	43

6.1. L'exemple de la méthode Scrum : vue globale	43
6.2. Les artefacts de Scrum	44
6.2.1. Product backlog (Carnet de produit)	44
6.2.2. Sprint backlog (Carnet de Sprint)	44
6.2.3. Kanban board (Tableau Kanban)	45
6.2.4. Burndown chart (Graphique d'avancement)	45
6.3. Les 4 cérémonies essentielles de SCRUM	46
6.3.1. Le Sprint Planning au début d'une itération (< 8 h)	46
6.3.2. Le Daily Meeting (quotidiennement 15')	46
6.3.3. Le Sprint Review (2 h) à la fin d'une itération	47
6.3.4. La Rétrospective (3 h) à la fin d'une itération	47
7. Que retenir de ce chapitre ?	47
Chapitre 3	48
Qualité logicielle	48
1. Objectif du chapitre	49
2. Notion de qualité logicielle et assurance qualité (QA)	49
2.1. Définitions	49
2.2. Normes de qualité	49
2.2.1. Normes de qualité par des facteurs internes	49
2.2.2. Normes de qualité par des facteurs externes	49
2.3. Bilan	50
3. Le point de vue de l'agilité	51
3.1. Artisanat du logiciel (<i>Software craftsmanship</i>)	51
3.2. Atelier de l'artisan	52
3.2.1. Environnement de développement	52
3.2.2. Outils d'aide à l'analyse de la propreté du code	52
4. Que retenir de ce chapitre	53
Chapitre 4	54
Code propre	54
1. Objectif du chapitre	55
2. Pourquoi faire propre ?	55
2.1. Code propre en proverbes	55
2.2. Pratique traditionnelle du codage	55

2.3. Pourquoi il ne faut pas coder comme cela	55
2.4. Les commentaires sont néfastes au code.....	56
2.4.1. Les commentaires mensongers	56
2.4.2. Les commentaires non informatifs	56
2.4.3. Les commentaires trompeurs.....	56
2.5. Les commentaires rendent le code illisible	57
2.5.1. Commentaires formatés avec des tags HTML	57
2.5.2. Commentaires de fin de bloc.....	57
2.5.3. Commentaires de séparation de parties	58
2.6. Les commentaires décrédibilisent le développeur	58
2.6.1. Les fautes d'orthographe	58
2.6.2. Les copier-coller malheureux	58
3. Code propre	59
3.1. Nouvelle pratique du codage	59
3.2. Note : cas des API publiques	60
3.3. Utiliser judicieusement les commentaires	60
3.3.1. Commentaires indispensables.....	61
3.3.2. Commentaires acceptables	61
3.3.3. Commentaires d'aide à la programmation : TODO / FIXME	61
4. Nommage des identificateurs	62
4.1. Utiliser avantageusement les noms d'identificateur	62
4.1.1. Choisir des noms explicites.....	62
4.1.2. Utiliser des noms plutôt que des commentaires	63
4.1.3. Ne pas en faire trop.....	63
4.2. Respecter des règles de nommage.....	64
4.2.1. Éviter l'ambiguïté	64
4.2.2. Utiliser des noms prononçables, mnémotechniques et partageables.....	64
4.2.3. Utiliser des noms distinguables	64
4.2.4. Ne pas avoir peur de faire des noms longs.....	65
4.2.5. Utiliser une convention pour rendre les noms composés lisibles.....	65
4.2.6. Passer du temps pour le choix des noms.....	65
4.2.7. Nommage des données membres (attributs).....	65
4.2.8. Nommage des fonctions / méthodes	66

4.2.9. Nommage des paquets en Java	67
5. Écrire des fonctions auto-documentées	67
5.1. Fonctions courtes	67
5.2. Fonctions d'une seule chose	67
5.3. Un seul niveau d'abstraction	68
5.4. Bannir le code lourd	68
5.5. Bannir le code naïf.....	69
5.6. Bannir le code de « geek »	69
5.7. Bannir l'humour douteux	69
6. Respecter des standards de formatage de code	70
6.1. Formatage en Java.....	70
6.2. Formatage en général	71
7. Ne pas faire d'optimisation prématurée	72
8. Code propre dans l'industrie (code review)	73
9. Que retenir de ce chapitre ?	74
10. Lecture	74
Chapitre 5	75
Refonte de code (refactoring)	75
1. Objectif du chapitre.....	76
2. Nécessité de la refonte de code	76
2.1. Qu'est-ce que la refonte de code ?	76
2.2. Qu'est-ce que n'est pas la refonte de code ?	76
2.3. Pourquoi faire de la refonte de code ?	76
2.4. Quand faire de la refonte de code ? Une nouvelle façon de coder	77
2.5. Quelques indices de la nécessité d'une refonte : « <i>bad smells in the code</i> »	77
3. Exemples de refonte de code	77
3.1. Factoriser le code redondant	78
3.2. Remplacer un « nombre magique » par une constante symbolique	78
3.3. Supprimer les doubles négations	78
3.4. Remplacer une méthode par un objet méthode	79
3.5. Introduire une variable explicative	79
3.6. Préserver un objet entier	80
3.7. Remplacer les conditions par le polymorphisme.....	80

3.8. Remplacer un code d'erreur par une exception	81
3.9. Remplacer un paramètre par une méthode	81
3.10. Introduire un objet paramètre	82
4. Que retenir de ce chapitre ?	82
5. Lectures	82
Chapitre 8	83
Test logiciel	83
1. Objectif du chapitre	84
2. Généralités sur les tests	84
2.1. Pourquoi tester ?	84
2.2. Bug	84
2.3. Qu'est-ce qu'un test ?	85
2.4. Niveaux de test	85
2.5. Tests dynamiques	86
2.6. Types de tests dynamiques	86
2.7. Exemple fil rouge	87
2.7.1. Étape 1 : Objectif de test	87
2.7.2. Étape 2 : Jeu de test	87
2.7.3. Étape 3 : Fixture	87
2.7.4. Étape 4 : Oracle	87
2.7.5. Étape 5 : Verdict	88
2.7.6. Étape 5 : Test exécutable : Cas de test	88
2.8. Quand s'arrêter de tester ?	88
2.9. Couverture de code	88
2.10. Doit-on écrire des tests pour tout ?	89
2.11. À quelle fréquence dois-je exécuter mes tests ?	89
2.12. Limite des tests	89
3. Test unitaire	90
3.1. Qualités d'un test unitaire : FIRST	90
3.2. Testabilité d'un code	90
3.2.1. Contrôler une entrée indirecte	91
3.2.2. Observer une sortie indirecte	91
3.2.3. Code testable	91

4. Frameworks de test : JUnit	91
4.1. Organisation des codes	92
4.2. Les assertions de base de JUnit	92
4.3. Méthode de test JUnit : @Test.....	92
4.4. Tester la levée d'exception.....	93
4.5. Visualisation du verdict de JUnit	93
4.6. Fixture JUnit : @BeforeEach.....	94
5. Frameworks de test : Mockito	95
5.1. La doublure pour les tests	95
5.2. Pourquoi des doublures ?.....	95
5.3. Exemple d'utilisation du framework Mockito	95
5.4. Exemple d'utilisation pour le substitut pour une classe « ServiceAuthentification »	96
5.5. Exemple d'utilisation pour l'espionnage pour une classe « ServiceAuthentification ».....	96
6. Développement dirigé par les tests.....	97
6.1. Quand tester ?.....	97
6.2. TDD : Test Driven Development	97
6.3. En pratique	97
6.4. Pourquoi écrire les tests avant le code ?	98
6.4.1. Avantages psychologiques	98
6.4.2. Avantages techniques	98
6.5. Pourquoi écrire un seul test à la fois ?.....	98
6.6. Pourquoi commencer par la barre rouge ?.....	98
7. Correction de bug.....	99
7.1. Mauvaise pratique.....	99
7.2. Bonne pratique.....	99
8. Que retenir de ce chapitre ?.....	101

Introduction générale

1. Pourquoi un cours sur le génie logiciel ?

Pourquoi vous faire un cours sur le développement de logiciel alors que vous êtes déjà formés à programmer, autrement dit, pourquoi un cours sur la programmation ne suffit-il pas pour développer un logiciel ?

Nous introduisons ici une différence fondamentale entre programme et logiciel, et par conséquent entre programmeur et développeurs :

- **les programmeurs créent des programmes,**
- **les développeurs créent des logiciels.**

1.1. Confusion entre programmeur et développeur

- Aujourd'hui tout le monde programme

Un programmeur réalise des œuvres personnelles à usage personnel plus ou moins bien réussies.

En faisant une analogie avec le génie civil, le programmeur écrit du code comme le maçon fait de la maçonnerie. Aujourd'hui, tout le monde programme, comme tout le monde peut construire une maison individuelle. Il n'y a pas besoin de connaissances pointues pour faire du code, d'autant que beaucoup de codes sont copiés directement d'Internet ou générés par des outils de génération automatique de code tels que ChatGPT, Copilot ou Gemini.

- Par contre, un développeur produit des ouvrages d'art qui sont hors de portée d'un programmeur.

Personne ne conçoit qu'un maçon fasse une tour d'habitation de 200 m de haut. Cette tâche relève de compétences d'ingénieurs du génie civil. En informatique, il faut des développeurs pour faire des logiciels et pas de simples programmeurs, même si cela paraît moins évident que dans le cas du génie civil.

Toutefois, certains développeurs resteront toute leur vie des programmeurs parce qu'ils n'auront pas la culture du génie logiciel.

En informatique, il y a beaucoup d'inculture qui aboutit à créer des ouvrages qui s'écroulent à la moindre brise de vent.

1.2. Confusion entre programme et logiciel

La confusion entre programmeur et développeur vient de l'amalgame qui est fait entre programme et logiciel.

- Quelle est la différence entre programme et logiciel ?

Pour répondre simplement à cette question, prenons l'exemple d'un programme qui fait l'addition de deux nombres entiers.

Le programme C suivant remplit parfaitement cette tâche :

```
int main() {  
    int nb1, nb2;  
    scanf("%d", &nb1);  
    scanf("%d", &nb2);  
    printf("Résultat = %d\n", nb1 + nb2);  
    return 0;  
}
```

Pour en faire un logiciel, il faudra au minimum ajouter une interface plus conviviale que la ligne de commande pour lire les deux entiers ; dans l'idéal, une interface graphique multilingue qui prend en compte par exemple l'entrée du nombre mille sous la forme 1000, 1.000 ou 1 000 en français, et 1000 ou 1,000 en anglais. Il faudra donc lire les nombres comme des chaînes de caractères, ce qui induit de vérifier que les valeurs entrées sont bien des entiers, protéger le programme contre une attaque de type << *buffer overflow* >> et traiter le cas où le résultat de l'addition des deux nombres dépasse la valeur maximale de la représentation des entiers qui est dépendante du système d'exploitation (OS) sur lequel le programme s'exécutera.

A cela, il faudra ajouter une batterie de tests pour tenter de débusquer les imperfections et se prémunir contre une régression éventuelle.

Ce simple exemple nous montre combien le développement est plus complexe que la programmation.

Listons plus profondément les grandes différences.

1.2.1. Différences entre programme et logiciel : l'utilisateur

Programme

- Un seul utilisateur averti et bienveillant (généralement son créateur).
- L'utilisateur accepte d'utiliser une procédure compliquée et fragile pour exécuter le programme.

Logiciel

- Tout utilisateur final cible est plus ou moins formé sur le logiciel.
- L'utilisateur attend une interface ergonomique rendant l'utilisation du logiciel « naturelle » et « efficace » (on parle d'expérience utilisateur ou **UX**).

1.2.2. Différence entre programme et logiciel : les cas d'utilisation

Programme

- 1 cas d'utilisation cible (*ie*, le cas nominal).
- L'utilisateur connaît les conditions d'utilisation et n'y déroge pas.

Logiciel

- Tous les cas d'utilisation doivent être prévus :
 - nominaux,
 - dégénérés,
 - frauduleux.
- Le cas nominal couvre en général 80 % des fonctionnalités attendues, mais les 20 % des cas restants réclament 80 % du temps de développement.

1.2.3. Différence entre programme et logiciel : la complexité

Programme

- Simple, monolithique et implante sur un OS unique.
- Interfaçant des périphériques qui sont physiquement à disposition au moment de la programmation.

Logiciel

- Portable sur différents OS.
- Reparti sur le réseau.
- Interfaçant des périphériques d'origine et de version diverses non nécessairement disponibles au moment du développement. Il faut alors s'en remettre à la norme sans vraiment pouvoir faire de tests autrement que sur des émulateurs.

1.2.4. Différence entre programme et logiciel : la taille du code source

L'unité de mesure de la taille d'un logiciel est le **LOC** : lines of code (1 MLOC correspond à 40 romans épais).

- **Programme**

- Quelques KLOC.

- **Logiciel**

- Commandes de vol A380 : 1 MLOC (contre 100 KLOC pour l'A320).

- OS Android : 11,8 MLOC.

- Linux kernel 5.8 (2020): 53 MLOC ; Noyau Linux 4.1 (2015) : 20MLOC ;
linux 3.1 (2011) : 15 MLOC.

- Facebook : 62 MLOC.

- Windows 10 : 80 MLOC (contre 40 MLOC Windows 7).

- Google (tous les services internet) : 2 GLOC en 2015.

Conséquence de la taille : développement en équipe

La taille des logiciels oblige à un travail en équipe. L'unité de mesure de la durée du projet en équipe est l'**année-homme** (*man-year*), soit un développement de 1 année pour 1 homme, ou six mois pour 2 hommes

ou toute autre combinaison. Par exemple : le cout de développement de l'algorithme de recherche de Google est estime à 1 000 années-hommes.

Conséquence de la taille : coût de développement

Ordre de grandeur en France :

- 1 année-homme \approx 1 650 h.

- 1 h \approx 50 €.

- Productivité (de l'analyse à la production) \approx 2 à 5 LOC / h.

Donc, le code suivant met 1 h à traverser tout le cycle de développement jusqu'à la production et coute 50 € !

```
void bubbleSort( int array[], int length ) {  
    for (int i = 0; i < length - 1; i++) {  
        for (int j = 0; j < length - 1 - i; j++) {  
            if (array[j] > array[j + 1]) {  
                int temp = array[j];  
                array[j] = array[j+1];  
                array[j+1] = temp;  
            }  
        }  
    }  
}
```

1.2.5. Différence entre programme et logiciel : la responsabilité des dysfonctionnements

- **Programme**

- L'utilisateur accepte la responsabilité des désagréments liés à l'utilisation du programme.

- **Logiciel**

- Les utilisateurs tiennent les développeurs pour responsables des conséquences néfastes de l'utilisation du logiciel (au moins moralement si non pénalement). Les développeurs doivent donc proscrire toutes les conséquences néfastes de l'utilisation du logiciel :

- perte de données,
- vol de données,
- résultats erronés,
- utilisation frauduleuse.

1.2.6. Différence entre programme et logiciel : la maintenance

- **Programme**

- Le programme est destiné à répondre à une tâche donnée à un moment donné.

- Son évolution dans le temps n'est pas une préoccupation.

- Le travail de programmation est terminé une fois que le programme fonctionne.

- **Logiciel**

- Un logiciel ne s'use pas, mais il se détériore à mesure que les technologies utilisées évoluent, les OS support évoluent et que le besoin des utilisateurs évolue.

- La maintenance évolutive et corrective est une Composante inhérente du développement d'un logiciel.

- L'expérience montre même que la majeure partie du travail de développement commence après la livraison du logiciel au client.

- La maintenance doit être envisagée dès sa construction. En reprenant l'analogie avec le génie civil, c'est au moment de la construction d'un pont suspendu que l'on prévoit le remplacement de câbles de soutènement.

1.3. Conclusion préliminaire

- Un programme peut se réaliser de façon empirique sans méthode.

- Un logiciel ne peut pas se développer sans méthode et un haut niveau d'expertise reposant sur le génie logiciel.

- **Développeur logiciel :**

- Un métier à haut niveau d'expertise reposant sur le génie logiciel.

- Un métier qui s'apprend.

- L'auto-formation donne des programmeurs.

Histoire et creation

2. Naissance du génie logiciel (1968)

2.1. La crise du logiciel

Expression née d'un constat en 1968 : il est incroyablement difficile de réaliser des logiciels satisfaisant la qualité attendue dans les délais prévus. La taille et la complexité conduisent à une incapacité à maîtriser le logiciel.

La raison majeure est l'absence de théorie générale de la construction de logiciel :

« *Nous sommes toujours à la recherche d'une théorie générale de la construction de logiciels, à l'image des équations de la mécanique classique qui permettent de concevoir un pont robuste.* »

Joseph Sifakis (prix Turing 2007).

2.2. Le génie logiciel

En l'absence de théorie générale, il est donc nécessaire de définir des procédures, méthodes et paradigmes pour rationaliser le processus de développement. Le génie logiciel est apparu en 1968 lors d'une conférence à Garmisch en Allemagne à l'initiative de la division des affaires scientifiques de l'OTAN !

Le terme anglais « *software engineering* » est dû à Margaret Hamilton, cheffe du projet du système embarqué du programme spatial Apollo et pionnière du génie logiciel.

Le génie consiste en un ensemble de pratiques régulées pour une corporation professionnelle et basées sur des principes scientifiques et économiques. Le génie logiciel postule donc que les principes du génie peuvent s'appliquer au développement de logiciels.

Définition : *Le génie logiciel est l'ensemble des activités de conception et de mise en œuvre des produits et des procédures qui tendent à rationaliser la production du logiciel et son suivi.*

La source d'inspiration est le génie civil. Le génie civil a démontré toute son efficacité depuis des siècles.

Le génie civil nous apprend que pour gérer efficacement un projet, il faut :

- Découper le temps alloué au projet en une séquence d'étapes parfaitement identifiées de manière à introduire du contrôle intermédiaire. (*Livrable : Diagramme de Gantt*)
- Ne faire qu'une seule chose à la fois à chaque étape. Cela induit des *métiers spécifiques à chaque étape*.
- Récolter tout le besoin avant d'élaborer une solution. (*Livrable : le cahier des charges*)

- Bien réfléchir avant d'agir, en commençant par une phase d'analyse exhaustive avant la phase de programmation qui en découle. (*Livrable : les documentations architecturale et fonctionnelle*)
- Programmer la solution en suivant les documentations techniques (*Livrable : code commenté*)

Le génie logiciel s'organise autour de 3 éléments :

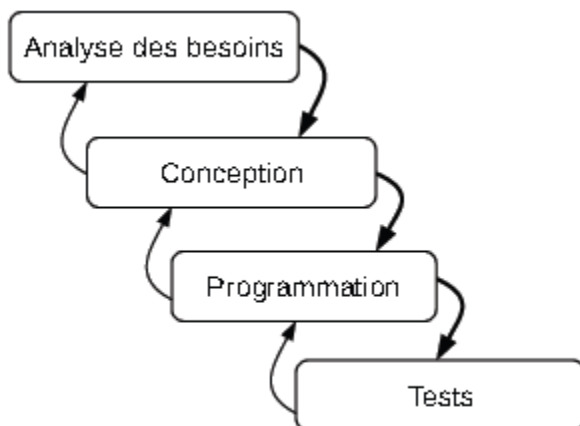
1. **Une méthode** : Elle vise à organiser le travail de développement en équipe et gérer le cycle de vie du logiciel.
2. **Un paradigme** : Il identifie les briques de base de la conception et les mécanismes pour les assembler.
3. **Un formalisme** : Il fournit un langage formel pour parler du code de manière abstraite et non-ambiguë en restant proche des langages de codage.
Ce formalisme est basé sur le paradigme dont il fournit une représentation

2.3. Une méthode : la méthode en cascade

Appliquée au génie logiciel, la méthode du génie civil aboutit à un cycle de développement cascade.

Ce cycle identifie 4 étapes séquentielles :

L'analyse des besoins, la conception, la programmation et les tests.



2.3.1. L'écosystème du développement logiciel

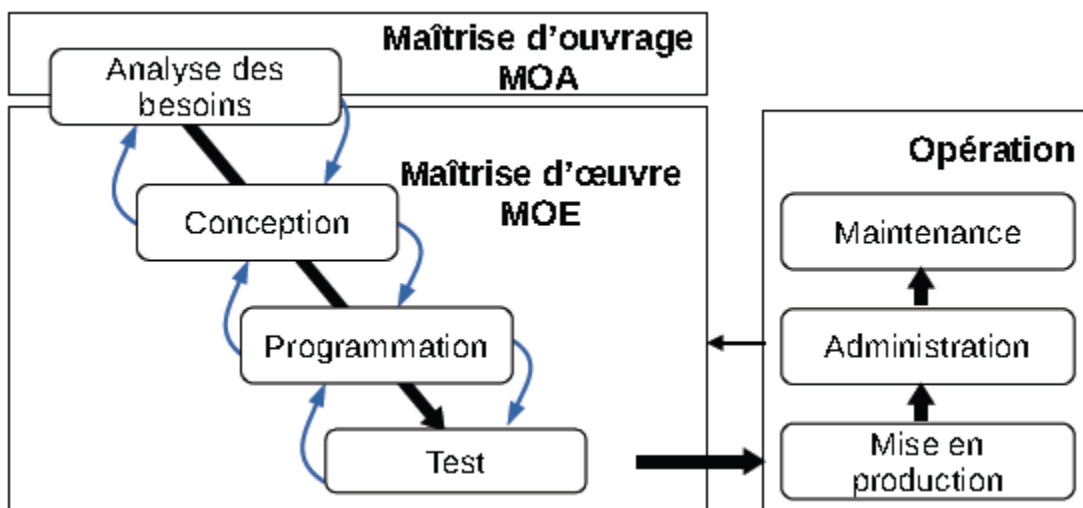
On appelle ouvrage le produit qui résulte d'un projet, ici le logiciel.

La maîtrise d'ouvrage (**MOA**) est à l'origine de l'idée de base du projet et représente à ce titre les utilisateurs finaux à qui l'ouvrage est destiné. Ils sont aussi nommés aussi donneur d'ordre ou client.

La maîtrise d'œuvre (**MOE**) désigne ceux qui réalisent le produit à partir des besoins exprimés par la MOA.

Une fois le code produit et testé par la MOE, il est donné à l'équipe opération.

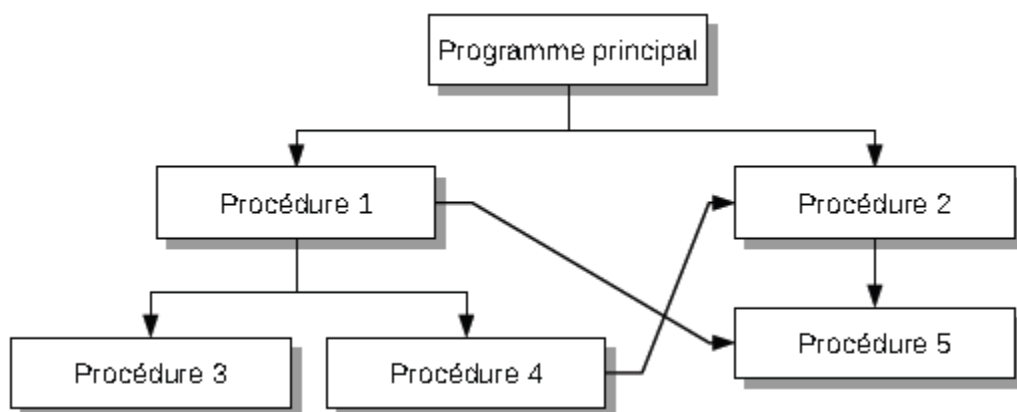
Son rôle est de faire la mise en production du logiciel à partir du code pour son exploitation effective par le client. Cela inclut des tâches de mise en exploitation sur l'infrastructure système du client, de garantie de la stabilité du système, de réponse aux tickets générés par la maintenance ou les remontées clients, de supervision de la conception et du contrôle des processus de production, de la mise en conformité du système et d'assurance sécurité du système.



2.4. Un paradigme : la conception procédurale

Un programme est conçu comme une suite finie de procédures (fonctions) plus ou moins élémentaires qui s'enchainent.

- Briques de base : les procédures paramétrables.
- Assemblage : appel de procédures avec des valeurs pour les paramètres.
- Programme : graphe d'appel des procédures paramétrées.



2.5. Un formalisme : l'algorithmique

L'algorithmique est basée sur un langage semi-formel incluant :

- instructions de base,
- structures de données,

- séquence d'instructions organisées par des structures de contrôle.

Le premier algorithme informatique a été formalisé par Ada Lovelace en 1843 pour calculer les nombres de Bernoulli par une machine.

INSERTION-SORT(<i>A</i>)	<i>cost</i>	<i>times</i>
1 for <i>j</i> = 2 to <i>A.length</i>	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

Algorithme du tri par insertion.

2.6. Ingénierie logicielle

Du cote de la MOE, il en résulte deux métiers spécifiques :

1. Analyste (ingénieurs)

- Analyse des besoins en liaison avec la MOA.
- Conception architecturale.
- Conception fonctionnelle.

L'analyse des besoins fournit le cahier des charges.

Les conceptions architecturale et fonctionnelle fournissent la documentation technique.

2. Programmeur (techniciens)

- Ecriture du code.
- Ecriture des tests.

Le code est produit par les programmeurs sur la base de la documentation technique fournie par les analystes.

3. Opération (ingénieurs et techniciens)

- Mise en production / Infrastructure.
- Maintenance corrective (répondre aux tickets des utilisateurs).
- Mise en conformité / sécurité.

3. Constat d'échec (2000)

En 2000, patatras, malgré l'application du génie logiciel, plus des deux tiers des projets restent non satisfaisants :

Projets échoués	23 %
Projets en difficultés	49 %
Projets réussis	28 %

Source : The Standish Group et 35 000 projets étudiés (2000).

Les projets en difficulté sont des projets qui aboutissent mais qui posent des problèmes tels que :

- une inadéquation des fonctionnalités livrées par rapport aux besoins des utilisateurs,
- le non-respect des délais spécifiés,
- ou encore l'augmentation des coûts.

3.1. Exemples de projets en échec

- *Le projet TAURUS* : La bourse de Londres a renoncé en mars 1993, après quatre ans de développement, au projet informatique Taurus qui devait assurer le suivi complet de l'exécution des transactions. Ce système a coûté directement 60 M£ et les opérateurs sur le marché ont dépensé 400 M£ pour y adapter leurs propres logiciels.

- *Système de paiement d'Atos* : La saturation du système d'autorisation de paiement dépassant 100 € a provoqué en pleine période d'achats de Noël 2001 de longues files d'attente de clients excédés dont beaucoup finiront par abandonner leurs chariots. Les autorisations de débit qui prenaient habituellement quelques dizaines de secondes, nécessitèrent ce jour-là quasiment une demi-heure. Le coût du préjudice pour le seul groupe Leclerc est de 2 M€.

3.2. Causes d'échec

Les causes d'échec sont maintenant bien connues.

3.2.1. Cause d'échec n°1 : Suivre un plan coûte que coûte

Le cycle en cascade définit des étapes séquentielles précises. C'est un engagement sur plusieurs mois (cf. diagramme prévisionnel de Gantt). Il n'y a pas (ou peu) de remise en cause possible par la suite.

En conséquence, si l'une des étapes en amont prend du retard, la planification des étapes en aval seront impactées et conduiront à mettre le projet en difficulté.

3.2.2. Cause d'échec n°2 : Estimer la durée d'un projet

Il est extrêmement difficile d'estimer la durée/cout d'un projet informatique.

Il y a trop d'aléas et d'incertitudes à prendre en compte.

Le travail en équipe complique encore cette prévision.

Ainsi, la mesure de la charge de travail en année-homme est un leurre (cf. « The Mythical Man-Month », Fred Brooks, 1995).

La charge de travail n'est pas linéairement liée au nombre de personnes dans le projet.

L'exemple ci-dessous donne le gain de productivité dans l'entreprise Borland Corporation en fonction du nombre de personnes dans l'équipe.

Taille équipe	Productivité par personne (KLOC/année)	Productivité de l'équipe (KLOC/année)	Gain
1	15.0	15.0	1
2	11.9	23.8	1.6
10	7.0	69.6	4.6
25	5.1	128.2	8.5

Source : Borland Software Corporation.

De plus, toutes les tâches ne sont pas divisibles :

« Neuf femmes ne font pas un enfant en un mois » (F. Brooks).

Enfin et contrairement à l'intuition, ajouter des personnes à une équipe d'ingénieurs dans un projet en difficulté ne permet pas de rattraper le retard. Au contraire, les nouvelles personnes devront être formées sur le logiciel en cours de construction par les membres du projet, ce qui entraînera des retards supplémentaires (loi de Brooks).

3.2.3. Cause d'échec n°3 : Définir les besoins au début

La phase de collecte des besoins au début du projet est illusoire.

- Les clients ne savent pas identifier exactement ce qu'ils veulent.
- Les clients ne savent pas exprimer clairement même les besoins identifiés.
- Les clients changent leurs besoins au cours du projet.

La collecte des besoins est au mieux incomplète et au pire caduque au moment de produire le logiciel.

3.2.4. Cause d'échec n°4 : Réaliser les tests à la fin

Les tests sont généralement la variable d'ajustement du temps. Etant réalisés à la fin, les tests sont faits selon le temps restant (ou pas).

Pourtant, il est impossible de garantir des logiciels sans défaut :

- Estimation : 1 à 10 bugs / KLOC : Windows 10 (80 MLOC) → 80,000 bugs !

- En 2006, 500 bugs ont été détectés dans la station spatiale internationale pendant son exploitation.

Pour poursuivre l'analogie avec le génie civil, les tests sont comme les armatures que l'on ajoute au béton pour le rendre plus solide. Il est criminel de construire une tour de 200 m avec du béton non armé.

3.2.5. Cause d'échec n°5 : Raisonner au niveau procédure

La conception procédurale est inadaptée à la conception de programmes de très grande taille. Le niveau procédural est trop bas pour appréhender la complexité des logiciels d'aujourd'hui.

4. Le génie logiciel aujourd'hui

4.1. Constat préliminaire

Le logiciel n'est pas le matériel. En reprenant la métaphore du génie civil, en développement logiciel, on peut commencer par construire un bungalow avant de le transformer en gratte-ciel et on peut commencer le bungalow par le carrelage de la salle de bain si le client considère que c'est un besoin prioritaire, ce qui est impossible en génie civil. Il doit donc être possible de repenser le génie logiciel en s'émancipant du génie civil.

4.2. Repenser le génie logiciel

La nouvelle vision du génie logiciel puise son essence dans l'*Agilité*. La nouvelle définition du génie logiciel devient : *Le génie logiciel est l'art et l'ensemble des moyens techniques, industriels et humains qu'il faut réunir pour construire, distribuer et maintenir des logiciels.*

Cette définition ajoute une dimension artisanale au développement (cf *software craftsmanship*) en prônant qu'il ne suffit pas qu'un logiciel soit fonctionnel, mais il faut qu'il soit bien conçu. Le développeur travaille comme un artisan, guidé par son talent, son expérience et ses connaissances théoriques puisées dans le partage et la formation. C'est sur ce point que l'expérience des développeurs expérimentés entre en jeu. Il existe aujourd'hui presque 60 ans d'expérience en génie logiciel dont il est nécessaire de s'inspirer. Le développement logiciel est un domaine qui souffre de beaucoup d'incompétence avec beaucoup d'autodidactes qui croient savoir.

Depuis quelques années, le développement logiciel a beaucoup changé y compris au niveau du codage, ce qui conduit à :

- une nouvelle méthode de gestion de projet : **méthode itérative et incrémentale** (agilité),
- un nouveau paradigme de conception logicielle : **conception orientée objet**,
- un nouveau formalisme de modélisation : **UML** (Unified Modeling Language).

Pour l'instant, si les outils tels que ChatGPT peuvent remplacer le travail de programmation, ils ne sont pas capables de remplacer le travail d'ingénierie logicielle.

4.3. Remarques

Le paradigme procédural et le formalisme algorithmique sont encore des principes de base du développement, mais, ils sont une préoccupation plus localisée sur des parties du logiciel.

4.4. Effet sur la réussite des projets

Selon l'étude annuelle du Standish Group, on constate une amélioration lente mais réelle le temps que cette nouvelle vision du génie logiciel mure dans les équipes de développement.

Rappel en 2000 2020

	Rappel en 2000	2020
Projets échoués	23 %	11 %
Projets en difficultés	49 %	42 %
Projets réussis	28 %	47 %

Source : The Standish Group. 10 000 projets étudiés (2015)

4.5. Ingénierie logicielle

Il n'y a plus de séparation entre les métiers analyste et programmeur. Ne reste qu'un seul métier dans une équipe MOE : **ingénieur développement** ou plus simplement **développeur**. Les ingénieurs qui composent l'équipe travaillent sur tout le cycle de vie d'une application depuis l'analyse des besoins, la conception, la programmation et les tests. Aujourd'hui, le développeur est même en charge de la production voire du déploiement du logiciel chez le client. On parle **d'ingénieur Devops**.

5. Conclusion du chapitre

Ce cours est axé sur la conception et le codage de logiciels.
Nous aborderons successivement :

3. Les méthodes de développement itérative et incrémentale en particulier l'agilité.
4. La qualité de code.
5. Les tests logiciels.

5.1. Pourquoi ce cours ?

A l'issue du cours, vous serez en mesure de :

- concevoir un logiciel de taille respectable en équipe,
- comprendre une conception modélisée,
- évaluer la qualité d'une conception,
- mettre en application les principes de base de l'agilité pour le développement.

A noter :

- « Les livres d'art ne permettent pas de vous transformer en artistes, pas plus que les livres sur le génie logiciel ne vous transforment en développeur. Ils ne peuvent que vous apporter les outils, les techniques et les processus de réflexion employés par d'autres développeurs qui sont sources d'inspiration. »

5.2. Pour qui ce cours ?

- Pour ceux qui choisiront de développer (MOE) :
- Trouver sa place dans la gestion de projet informatique.
- Se forger une culture du développement logiciel de haut niveau.
- Prendre conscience de l'importance du code et des tests.
- Pour ceux qui choisiront de ne pas développer (MOA, Conseil, Avant-vente, Ingénierie produit) :
- Etre en mesure d'exprimer des besoins et de suivre un développement de logiciel.
- Comprendre comment sont construits les ouvrages à spécifier et apprécier leurs contraintes.
- Gagner en crédibilité face aux personnes de la MOE.

5.3. Que retenir de ce chapitre ?

- Le développement de logiciels est plus que la programmation d'applications.
- Le développement de logiciels ne peut pas se faire sans méthode.
- Le génie logiciel a beaucoup évolué depuis quelques années. Il est défini par :
- Une méthode itérative et incrémentale pour structurer le processus de développement et organiser le travail en équipe.
- Un paradigme de conception basé sur la notion d'objet.
- Un formalisme de modélisation basé sur le langage UML.
- Le génie logiciel est fortement influencé par l'agilité.
- L'agilité repense la planification et la production des artefacts (code, documentation, produit).
- Le développement logiciel doit être appréhendé comme de l'artisanat mais en profitant d'une forte culture théorique et pratique de 50 ans.

Chapitre 2

Une méthode : Agilité

1. Objectif du chapitre

Il ne s'agit pas d'un cours sur l'agilité mais plutôt une introduction à l'agilité pour le développement logiciel.

Ici, nous n'en présentons qu'une approche pragmatique utilisable pour les TP et les projets. A l'issue de ce chapitre, vous serez sensibilisé à :

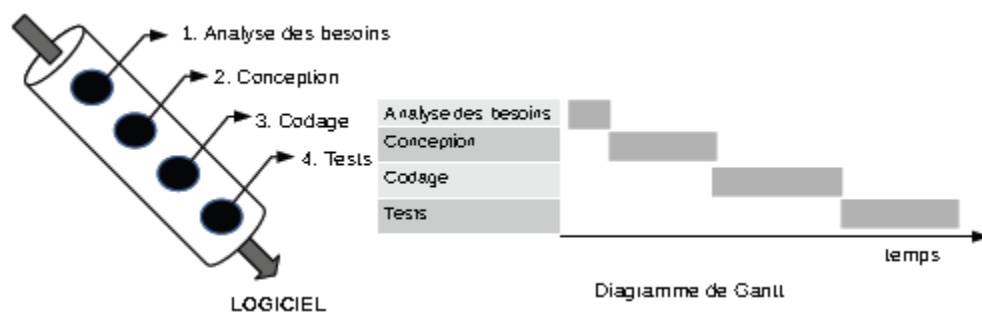
- l'importance d'un cycle de développement itératif et incrémental,
- la nécessité d'un dialogue permanent avec les futurs utilisateurs sur la base d'une version opérationnelle du futur logiciel,
- le rôle central du code,
- l'obligation de tester son code.

Cela doit vous permettre de changer votre façon de développer en TP et en projet pour aborder plus sereinement le travail de conception et de programmation.

2. Mauvaises pratiques du développement logiciel

2.1. Planification à long terme : le modèle de développement en cascade

Le modèle de développement en cascade fait partie des méthodes prédictives issues de génie civil. Il enchaîne séquentiellement 4 étapes. L'ordonnancement temporel des phases se représente sur un diagramme de Gantt qui chronomètre chacune des phases et donne leur organisation. Le diagramme de Gantt revêt une importance fondamentale puisqu'il est contractuel.



Ce modèle de développement a longtemps été enseigné et utilisé en entreprise. Malheureusement, il est la cause de l'échec de nombreux projets professionnels.

2.2. Pourquoi le modèle en cascade ne fonctionne pas pour le développement de logiciels

Outre la difficulté de prévoir le temps consacré à chacune des étapes que nous avons déjà pointées en introduction, les raisons se retrouvent aussi dans chacune des étapes.

2.2.1. Analyse des besoins au début

On se met d'accord avec le client au début du projet sur la liste des fonctionnalités à développer. Cette étape peut durer 2 mois de manière à bien collecter les besoins dans le cahier des charges. Et quand on ressort du tunnel de développement au bout de 3 mois ou 1 an, les besoins du client ont changé ! On appelle « effet tunnel » la période durant laquelle le client n'a aucune nouvelle du futur logiciel.

Un autre effet pervers d'une analyse des besoins en amont est l'invention de besoins. Puisque, par contrat, seules les fonctionnalités listées dans le cahier des charges seront développées, le client est poussé à imaginer des besoins. La conséquence selon une étude du Standish Group en 2006, c'est que près de 45 % des fonctionnalités demandées par les utilisateurs ne sont en fait jamais utilisées. C'est donc du temps, des ressources et des coûts consacrés à ces développements qui sont perdus.

2.2.2. Conception puis codage

On peut très bien découvrir au moment du codage un problème qui remet en cause la conception. Il faut alors recommencer la phase de conception, ce qui impacte la planification prévisionnelle et peut mettre le projet en péril.

2.2.3. Intégration en fin de codage

L'organisation du travail en équipes consiste à décomposer le développement en différentes parties qui vont être distribuées à différentes équipes qui travaillent en parallèle afin d'accélérer le développement du produit. Puis on réunit ces différentes parties à la fin pour faire le produit final. Mais, l'intégration en fin de projet ne fonctionne jamais parce qu'elle génère trop de conflits à régler.

C'est pourquoi cette approche est appelée « intégration big bang » puisque tout explose et rien ne fonctionne.

2.2.4. Test après le codage

Les bugs et les erreurs de conception doivent être détectés le plus tôt possible. Plus le code est avancé, plus la correction est difficile. De plus, la définition des tests en fin de codage est inefficace parce que les objectifs de ce que l'on doit tester ont été oubliés. Enfin, les tests sont très souvent court-circuités pour tenir les délais ce qui rend le logiciel fragile et peu sûr.

2.3. Une amélioration : le cycle en V

L'apport principal du cycle en V se situe au niveau des tests (voir Figure 1). Le meilleur moment pour définir les tests, c'est lors de la spécification des éléments à tester. Ainsi, quand on rédige les spécifications à chaque étape, on rédige en même temps les tests de validation. Toutefois, les tests ne seront implémentés qu'après l'étape de programmation. Mais pour le reste, il n'y a aucune réelle amélioration. Par exemple, du point de vue MOA, le cycle en V est toujours vu comme un tunnel et les tests sont toujours réalisés à la fin.

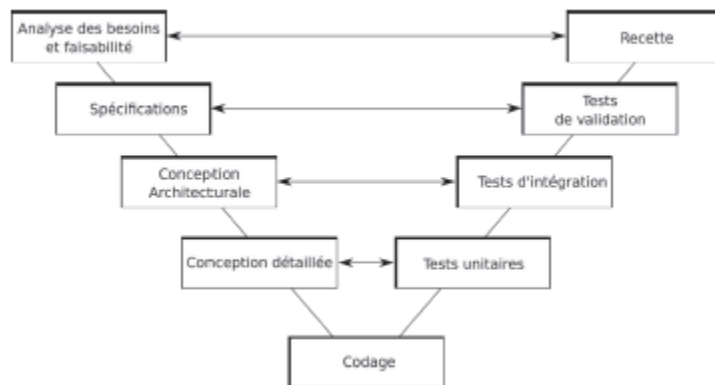


Figure 1 : Le cycle de vie en V.

2.4. Documentation exhaustive

La documentation est un pilier des méthodes prédictives. Elle est nécessaire pour :

- passer à la phase suivante,
- revenir en arrière en cas d'erreur,
- faire la maintenance du logiciel.

Mais en informatique, la documentation s'avère généralement inutile voire néfaste :

- Elle n'est jamais lue. C'est donc du temps précieux perdu. Vous-même, utilisez-vous la documentation qui vous est fournie ?
- Elle est laborieuse à rédiger. Elle a tendance à être bâclée et se révèle alors confuse ou incomplète et donc finalement inutilisable.
- Elle freine les changements. Il faut maintenir la documentation en même temps que les changements.

Mais, comme cela est difficile (voire impossible) à garantir, la documentation devient rapidement obsolète voire nuisible puisqu'elle n'est plus en accord avec ce qu'elle est censée décrire.

Une documentation n'a de sens qu'à un instant donné ou pour quelque chose qui n'évolue pas. Ce n'est pas le cas d'un logiciel professionnel. Par contre, c'est le cas d'une API qui par essence doit rester stable.

3. Bonnes pratiques du développement : agilité

Fort du constat précédent, dix-sept experts du développement d'applications informatiques ont repensé le génie logiciel qu'ils ont nommé Agile, et rédigé un manifeste en 2001.

3.1. Manifeste Agile

3.1.1. Les 4 valeurs de l'agilité (comparées à celles des méthodes prédictives)

Méthodes agiles		Méthodes prédictives
<ul style="list-style-type: none">- Individus et interactions- Logiciel qui fonctionne- Collaboration avec les clients- Adaptation au changement	plutôt que plutôt que plutôt que plutôt que	<ul style="list-style-type: none">- Processus et outils- Documentation exhaustive- Négociation contractuelle- Suivi d'un plan

Remarque : << Bien qu'il y ait de la valeur dans les méthodes prédictives, notre préférence se porte sur les éléments qui se trouvent dans les méthodes agiles. >>

3.1.2. Les 12 principes de l'agilité

Ces quatre valeurs se déclinent en douze principes.

1. **Motivation des équipes** : réalisez les projets avec des personnes motivées, fournissez-leur l'environnement et le soutien dont elles ont besoin et faites-leur confiance pour atteindre les objectifs fixés.

2. **Le dialogue face à face** : privilégiez la colocation de toutes les personnes travaillant ensemble et le dialogue en face à face comme méthode de communication.

3. **Rythme soutenable** : les processus agiles encouragent un rythme de développement soutenable.

Ensemble, les commanditaires, les développeurs et les utilisateurs devraient être capables de maintenir indéfiniment un rythme constant.

4. **Équipes auto-organisées** : les meilleures architectures, spécifications et conceptions émergent d'équipes auto-organisées.

5. **Opérationnel sinon rien** : un logiciel opérationnel est la principale mesure de progression d'un projet.

6. **L'excellence technique** : une attention continue doit être portée sur l'excellence technique et une bonne conception.

7. **La simplicité**. La simplicité (cf, principe KISS : *Keep It Simple, Stupid*), c'est-à-dire l'art de minimiser la quantité de travail inutile, est essentielle.

8. **Satisfaction des clients** : notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à forte valeur ajoutée.

9. **Livraisons fréquentes** : livrez fréquemment un logiciel opérationnel avec des cycles de quelques semaines.

10. **Travail client-développeur** : les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet.

11. **Accepter le changement du besoin** : accueillez positivement les changements de besoins, même tard dans le projet.

12. **Amélioration continue** : à intervalles réguliers, l'équipe réfléchit aux moyens possibles pour devenir plus efficace. Puis elle s'adapte et modifie son mode de fonctionnement en conséquence.

Il ressort de ces douze principes, que l'agilité met en avant les relations humaines débarrasse au maximum des outils numériques de collaboration et considère le développement comme un travail artisanal libère des normes qui en régissent la qualité.

4. Mise en pratique de l'agilité pour le développement

Du côté de la maîtrise d'œuvre (MOE), la mise en pratique des méthodes agiles passe par :

1. un cycle de développement itératif,
2. un cycle de développement incrémental,
3. une collaboration entre le client et l'équipe de développement,
4. une équipe auto-organisée avec un rythme constant,
5. l'intégration continue du travail des développeurs (quasi-quotidienne).

L'idée est de développer le logiciel par itérations pour produire des versions successives du logiciel qui soient exécutables et donc testables en grandeur réelle.

Le logiciel grossit progressivement dans une direction donnée régulièrement par le client.

Le projet est découpé en une suite de petits projets facilement maîtrisables par l'équipe de développement.

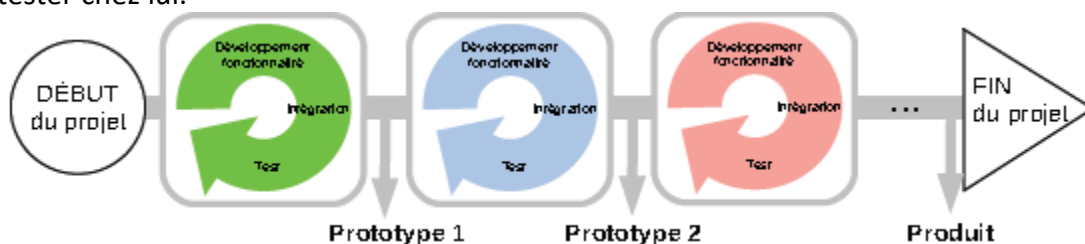
4.1. Cycle de développement itératif : casser la planification à long terme

Le temps total de développement est divisé en **itérations de même durée**.

Une itération est un-mini projet de très courte durée, généralement 2 à 3 semaines.

On y retrouve toutes les étapes du cycle en cascade.

Chaque itération se termine par la livraison d'un prototype opérationnel que le client peut tester chez lui.



Il faut considérer chaque itération comme une fin en soi, comme si le client allait décider d'arrêter le projet à l'issue.

En conséquence :

- Il ne peut y avoir de tâche de développement s'étalant sur plusieurs itérations ; au besoin la découper.
- Le prototype doit être testé dans les conditions de la production et son code doit être propre. Il est pratiquement livrable en état. Il est simplement sous-doté en fonctionnalités.

4.1.1. Avantages du cycle itératif

Du fait que la durée d'une itération est courte, cela réduit considérablement l'impact des problèmes liés à la gestion des projets identifiés plus tôt.

- On avance à petits pas testés. Si on rate un pas, on n'a raté qu'un pas (ie 2 semaines), sans grosses conséquences.
- Chaque pas est validé par le client.
- Le manque de temps conduit à un déficit de fonctionnalités, mais pas à un échec total du projet ni à un logiciel non testé.
- On a toujours une version intermédiaire mais opérationnelle du logiciel à donner au client.

4.2. Cycle de développement incrémental : profiter de la malléabilité du logiciel

Le logiciel est construit par **incrémentation**. Un prototype n'est pas jetable. Au contraire, il sera repris pour être amélioré au cours des itérations suivantes s'il est jugé intéressant par le client ou revu s'il ne satisfait pas le client. Chaque prototype suivant ajoute, supprime et modifie des fonctionnalités au prototype précédent.

4.2.1. Incrément : MVP (Minimum Viable Product) ou POC (Proof Of Concept)

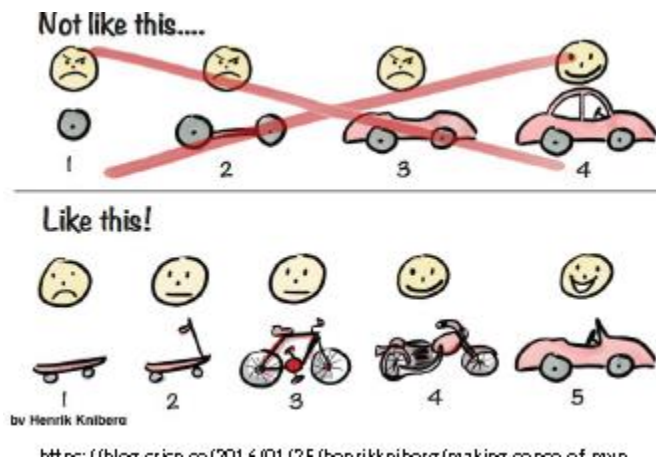
Le produit minimum viable (MVP) est un prototype avec juste assez de fonctionnalités pour satisfaire les premiers clients et fournir une rétroaction pour poursuivre le développement. Le MVP permet de valider ou d'invalider des hypothèses.

On passe d'un développement basé sur des présupposés, << *je pense que le client a besoin de ça* >>, à un développement basé sur des retours factuels, << *le client me demande ça* >>.

Cela permet de maximiser la création de valeur en évitant de louper sa cible et de dépenser des ressources sur des fonctionnalités à faible valeur ajoutée.

Un MVP doit apporter rapidement de la valeur à chaque itération. La valeur peut se résumer par toute fonctionnalité ou amélioration participant à l'atteinte des objectifs métiers.

Le concept de MVP peut être illustré par le dessin ci-dessous qui symbolise le développement d'une voiture (Henrik Kniberg, 2016).



Le scénario du haut, livrer successivement un pneu, un châssis, une caisse et enfin la voiture ne correspond pas à l'idée de MVP. Même si on livre régulièrement des prototypes, ces prototypes ne sont pas utilisables par le client : que voulez-vous qu'un automobiliste fasse d'un pneu ou d'un châssis dans sa cour ?

Par contre, dans le scénario du dessous, les prototypes livrés sont utilisables par le client qui peut faire des retours sur l'utilisabilité du prototype. Ce scénario profite de la malléabilité du code.

Le premier livrable est un skateboard. Pensez au skateboard comme une métaphore de la plus petite chose que vous pouvez mettre dans les mains des utilisateurs et obtenir de vrais retours. Il est développable en une itération et permet d'avoir des premiers retours des utilisateurs sur le besoin réel.

On peut apprendre par exemple que le skateboard couvre le besoin réel de se déplacer mais qu'il manque de stabilité.

Le deuxième prototype ajoute alors cette dimension et on obtient une trottinette.

A la place du skateboard, on aurait pu proposer un ticket de bus. Si cette solution couvrait le besoin complètement, le projet serait réalisé en une itération.

Quand le prototype n'est pas forcément porteur de valeur visible immédiatement pour le client, on parlera de prototype de type preuve de concept (POC). C'est le cas par exemple lorsqu'une itération est consacrée à une étude pour lever des verrous technologiques (Implantation d'une partie du logiciel sur GPU).

4.2.2. Avantages du cycle incrémental

Encore une fois, l'important c'est que chaque prototype soit une version livrable du logiciel, sous-doté en fonctionnalités, mais opérationnel pour l'utilisateur qui peut repartir avec et faire ses essais.

Les avantages :

- Donne rapidement de la valeur au produit.
- Le client voit une évolution rassurante et constante du produit.
- Réduit le temps de mise sur le marché (*time to market*).

4.3. Collaboration client – développeur

Le client (utilisateur / donneur d'ordre / MOA / AMOA) est un partenaire à part entière de l'équipe de développement. Il est fortement impliqué dans le cycle de développement (quotidiennement dans l'idéal).

C'est lui qui définit les priorités et décide des choix.

4.4. Équipe

Une équipe Agile présente les caractéristiques suivantes :

- Taille réduite (< 10 personnes) de manière à garder une relation de proximité.
- On découpera en plusieurs équipes sur de très gros projet.
- Formée d'ingénieurs développeurs.
 - Auto-organisée sans chef de projet.
 - Rythme soutenable. Ne prendre qu'une liste de tâches réalisables dans l'itération.
 - Dialogue face à face.
 - Motivation des équipes.
 - Propriété collective du code.

4.5. Intégration continue

L'intégration du travail de chaque développeur dans la version commune doit être faite quasi quotidiennement ; en fait à chaque fois que la fonctionnalité (petite) dont il est en charge est complètement développée.

Le but est bien évidemment d'éviter la mauvaise surprise de l'effet « big bang » de l'intégration en fin d'itération.

Il n'existe qu'une version courante du logiciel partagée par tous les développeurs. Cette version est opérationnelle et testée. Chaque développeur en possède une copie de travail. Il développe sa fonctionnalité sur sa copie en parfaite isolation. A la fin du développement de la fonctionnalité, il ajoute sa contribution à la version commune.

4.5.1. Cycle d'intégration continue (DevOps)

A chaque fois que le développeur pousse sa contribution dans la version commune, cela déclenche des mécanismes automatiques de vérification de la compilation et de la non-régression de la version commune.

Cela peut aussi aller jusqu'au déploiement automatique chez le client (*déploiement continu*). On parle de **DevOps** quand l'équipe gère le produit de sa définition jusqu'à sa mise en production. Cela suppose que la version doit être compilable et testable sans intervention humaine. Dès qu'une erreur survient dans le cycle de DevOps, que ce soit une erreur de compilation, de test ou de déploiement, l'équipe se mobilise en urgence pour corriger l'erreur afin de toujours avoir une version commune opérationnelle. La figure 2 suivante schématise le cycle de DevOps.

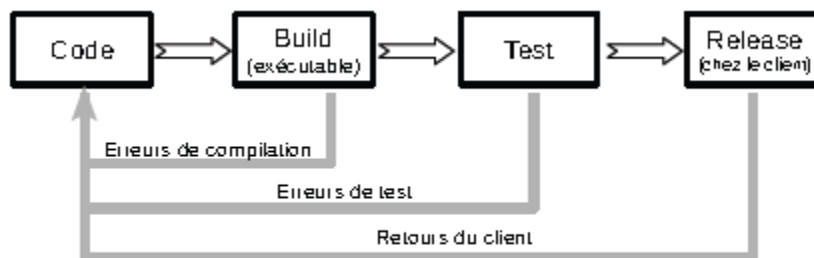


Figure 2. Cycle d'intégration (déploiement) continue.

5. Auto-documentation

En agilité, seul compte le code. Il n'y a plus, ou peu, de production de documentation.

Les documentations UML et textuelles mentent parce qu'elles ne sont pas toujours synchronisées avec les modifications qui sont apportées au code.

Au contraire, le code ne ment pas.

L'auto-documentation consiste à écrire le logiciel avec une architecture et du code qui se lisent comme une documentation. Nous reparlerons de la façon d'écrire du code dans le chapitre concené.

5.1. Modélisations UML

Les modélisations UML ne sont plus utilisées pour faire la documentation des spécifications. Par contre, elles restent utilisées pour échanger entre développeurs et avancer dans la compréhension du domaine et des besoins à un instant donné. On utilise pour cela, une version allégée d'UML ; les 6 diagrammes principaux sont en général suffisants.

Ce ne sont que des modélisations temporaires, mais elles doivent être exactes pour éviter l'ambiguïté et doivent rester simples pour apporter une valeur claire.

Si on a besoin de documentation, par exemple lors de la reprise de code, alors on utilise la retro-ingénierie du code (*reverse engineering*).

6. Exemples de méthodes agiles

Il n'existe pas une mais plusieurs méthodes agiles, dont les principales sont :

- **eXtreme-Programming** (aussi nommée XP) : la méthode pionnière dont la plupart des principes sont repris par les suivantes,

- **Kanban** : la méthode la plus simple et la plus connue par son fameux tableau (TODO/DOING/DONE).

- **Scrum** : la méthode la plus utilisée en France.

La mise en place concrète de l'agilité au sein d'une organisation mixe en général plusieurs méthodes.

Ce qui importe, c'est que l'application de l'agilité soit elle-même agile : il faut s'approprier l'agilité mais en respectant ses valeurs et ses principes.

6.1. L'exemple de la méthode Scrum : vue globale

Scrum est considérée comme un framework, c'est-à-dire un cadre général pour le développement de projet.

La Figure 3 résume les éléments de la méthode.

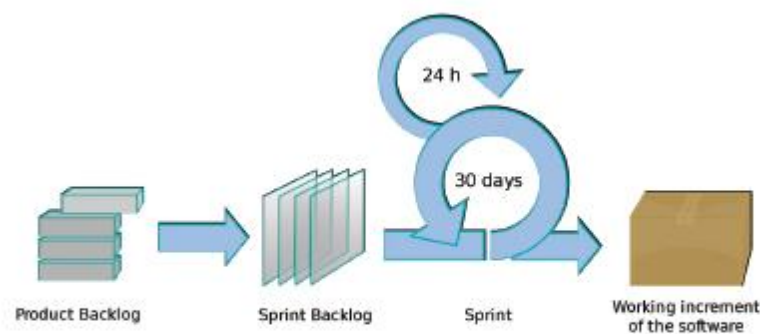


Figure 3

re 3 : Vue générale de la méthode Scrum.

Figure 3
méthode Scrum.

: Vue générale de la

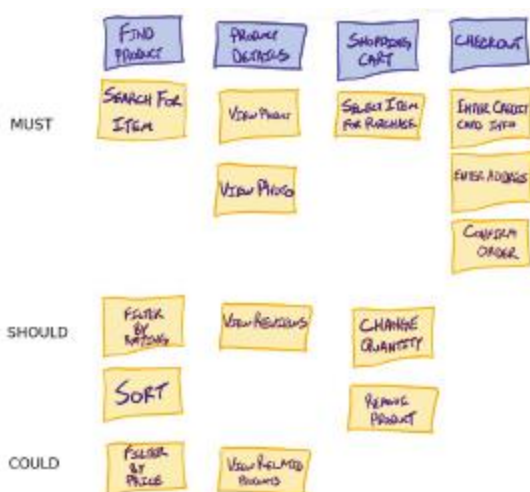
6.2. Les artefacts de Scrum

6.2.1. Product backlog (Carnet de produit)

Ce carnet formalise la vision du produit (ie, le logiciel) que le client souhaite réaliser sous la forme d'une liste de fonctionnalités à développer. Il contient :

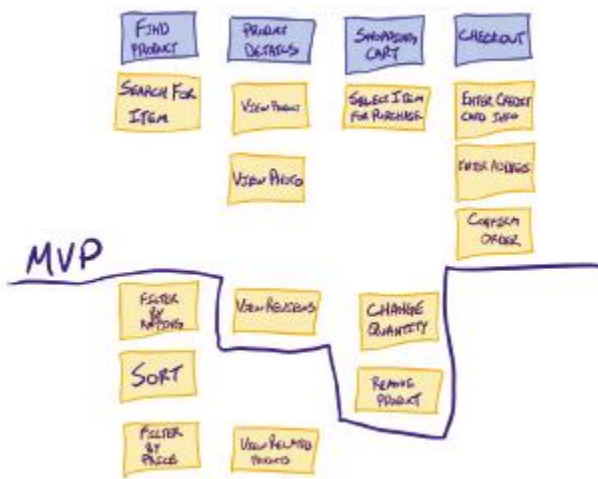
- Les exigences fonctionnelles et non fonctionnelles du produit avec une priorité entre elles (MUST, SHOULD, COULD).
- Ce carnet de produit donne une vision globale du fonctionnement du futur produit : il doit raconter l'histoire du futur produit.

Il se présente sous la forme d'une liste d'activités (« Find Product ») qui se déclinent en une liste de tâches (« Search for item »).



6.2.2. Sprint backlog (Carnet de Sprint)

Parmi les tâches du carnet de produit, il faut choisir celles qui seront développées au cours de l'itération. Le principe est de commencer par les exigences qui apportent le plus de valeur ajoutée au client/utilisateur (dans les « must »). Cette liste doit dessiner les contours du MVP. Pour cela, les tâches seront choisies un peu partout dans les activités et pas seulement dans une seule activité.

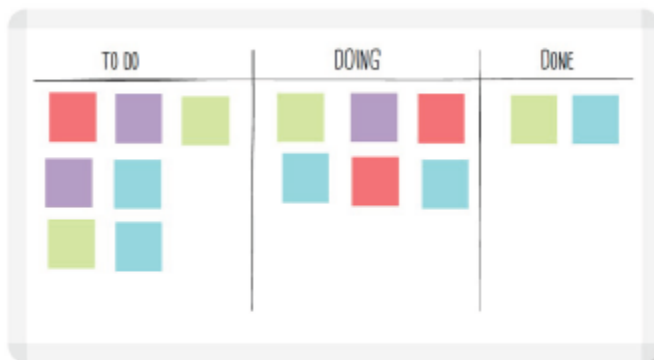


6.2.3. Kanban board (Tableau Kanban)

On reconnaît ici le fameux tableau d'organisation du travail d'une équipe. Chaque développeur prend une tâche de la travée TODO et la fait passer de DOING à DONE.

Tous les développeurs ont accès à ce tableau et voient les tâches accomplies par les autres développeurs.

Il est possible d'ajouter d'autres travées, comme TESTING par exemple.

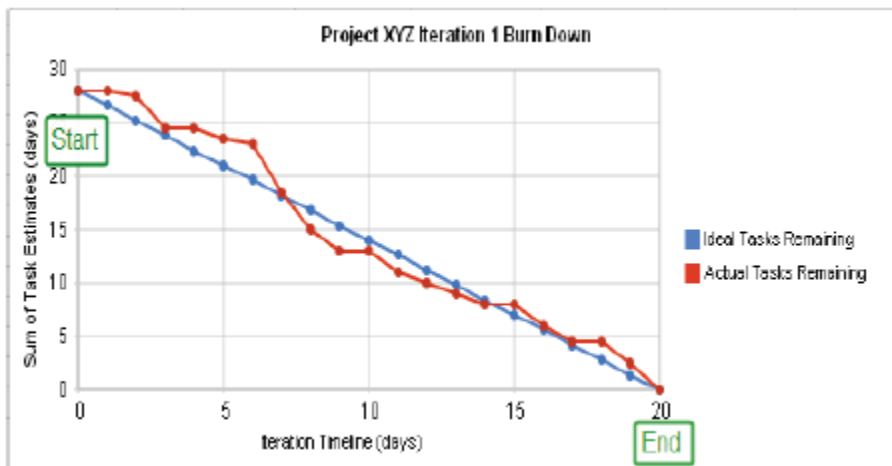


6.2.4. Burndown chart (Graphique d'avancement)

Ce graphique donne l'évolution de quantité de travail restante par rapport au temps alloué. Le travail restant se situe sur l'axe vertical, alors que le temps est sur l'axe horizontal.

Il est utilisé à l'issue d'une itération pour améliorer la manière d'estimer le volume de tâches réalisables par l'équipe en une itération et aucunement à évaluer le volume de travail réalisé par l'équipe.

D'ailleurs, pour éviter ce travers, la quantité de travail estimé pour chaque tâche est estimée en termes de points sans unité.



6.3. Les 4 cérémonies essentielles de SCRUM

Scrum est bâti sur des rituels simples mais efficaces qui permettent de rythmer la gestion de projet.

Malgré leur caractère simpliste, ils sont en fait très efficaces pour donner de la cohésion au projet.

6.3.1. Le Sprint Planning au début d'une itération (< 8 h)

Cette réunion a pour but de construire le carnet de sprint (*sprint backlog*) à partir d'une sélection des tâches du carnet de produit (*product backlog*). Elle inclut aussi l'estimation des points sur les tâches choisies. Les grosses tâches doivent être décomposées en sous-tâches plus simples.

6.3.2. Le Daily Meeting (quotidiennement 15')

Lors de cette réunion quotidienne de début de journée, chaque membre décrit ce qu'il a fait hier, les obstacles rencontrés et ce qu'il compte faire aujourd'hui. Toutefois, on n'y discute pas des problèmes ; cela est fait à l'issue de cette réunion avec les personnes concernées.

6.3.3. Le Sprint Review (2 h) à la fin d'une itération

Cette réunion a pour but de faire la démonstration du prototype auprès du client et de récolter ses retours. Le Backlog (*cahier de produit*) est alors mis à jour.

6.3.4. La Rétrospective (3 h) à la fin d'une itération

C'est une réunion interne à l'équipe. L'objectif est d'inspecter la gestion de projet menée à l'itération précédente, afin de déterminer ce qui a bien fonctionné et ce qui est à améliorer. Le burndown chart (*graphique d'avancement*) fournit un des outils pour cette réunion. L'équipe déduit un plan d'actions d'amélioration qu'elle mettra en place lors de l'itération suivante.

7. Que retenir de ce chapitre ?

Les méthodes agiles proposent de nouvelles façons d'aborder le développement logiciel. En particulier, le développement suit un cycle itératif et incrémental.

- Chaque itération correspond au développement de plusieurs petites fonctionnalités qui sont intégrées aussitôt au logiciel, testées et mises en production.
- Chaque itération doit se terminer par la livraison d'une version opérationnelle et testée du logiciel en cours.
- Chaque incrément doit ajouter une valeur pour le client : le MVP.
- Chaque itération est une fin en soi : un mini-projet.

Tout ceci fait, que les méthodes Agiles sont plus vues comme des méthodes de gestion de produit que des méthodes de gestion de projet.

Attention : Les méthodes agiles ne renient pas les méthodes prédictives telles que le cycle en V, mais dans la mesure du possible elles privilégient les valeurs agiles.

Chapitre 3

Qualité logicielle

*« Le débogage est deux fois plus difficile que l'écriture de code.
Donc, si vous écrivez du code de la manière la plus intelligente possible,
vous n'êtes, par définition, pas assez intelligent pour le déboguer. »*

Brian Kernighan

1. Objectif du chapitre

Ce chapitre est consacré à la présentation de la notion de qualité logicielle avec un focus particulier sur le point de vue des méthodes agiles.

A l'issue de ce chapitre :

- Vous vous rendrez compte que, pour le développeur, la qualité logicielle résulte plus d'une démarche artisanale que de l'application de normes.
- Vous serez sensibilisé à la notion de qualité de code.
- Vous saurez tirer parti des indicateurs de qualité pour renforcer la qualité de vos codes.

2. Notion de qualité logicielle et assurance qualité (QA)

2.1. Définitions

La **qualité** englobe l'ensemble des caractéristiques d'un logiciel qui affecte sa capacité à satisfaire des besoins exprimés ou implicites en termes de fonctionnalités, délais et coûts.

Une appréciation globale de la qualité tient autant compte de :

- facteurs externes, directement observables par l'utilisateur,
- facteurs internes, observables par les ingénieurs de développement.

L'**assurance qualité** est un ensemble d'activités planifiées et systématiques de toutes les actions nécessaires pour fournir une assurance que la qualité du logiciel est conforme aux exigences et aux attentes établies.

2.2. Normes de qualité

La qualité logicielle est régie par des normes, issues de la famille ISO 9000.

2.2.1. Normes de qualité par des facteurs internes

Citons deux normes emblématiques :

- La norme ISO/CEI 90003 est la ligne directrice de l'application de la norme ISO 9001 à l'acquisition, la fourniture, le développement, l'exploitation et la maintenance de logiciels informatiques et aux services de soutien connexes.

Elle se base entièrement sur l'approche prédictive de la gestion de projet.

- La norme ISO/CEI 12207 décrit un modelé pour le processus du cycle de vie du logiciel qui reprend globalement le modelé en V.

2.2.2. Normes de qualité par des facteurs externes

- La norme ISO/CEI 25010 décrit la qualité d'un logiciel à partir de 35 indicateurs de performance (*Key Performance Indicator*, KPI) couvrant 8 caractéristiques :



- La capacité fonctionnelle représente la mesure dans laquelle un produit ou un système fournit des fonctions qui répondent aux besoins exprimés et implicites lorsqu'il est utilisé dans des conditions spécifiées.
- L'efficacité des performances représente la performance par rapport à la quantité de ressources utilisées dans des conditions indiquées.
- La compatibilité est le degré avec lequel un produit, un système ou un composant peut échanger des informations avec d'autres produits, systèmes ou composants, et/ou exécuter ses fonctions requises, tout en partageant le même environnement matériel ou logiciel.
- L'utilisabilité est la mesure dans laquelle un produit ou un système peut être utilisé par des utilisateurs spécifiés pour atteindre des objectifs spécifiés avec efficacité, efficacité et satisfaction dans un contexte d'utilisation spécifiée.
- La fiabilité est le degré avec lequel un système, un produit ou un composant exécute des fonctions spécifiées dans des conditions spécifiées pendant une période de temps spécifiée.
- La sécurité est le degré avec lequel un produit ou un système protège les informations et les données afin que les personnes ou autres produits ou systèmes disposent d'accès aux données en fonction de leurs types et niveaux d'autorisation.
- La maintenabilité représente le degré d'efficacité et d'efficacité avec lequel un produit ou un système peut-être modifié pour l'améliorer, le corriger ou l'adapter aux changements de l'environnement et des exigences.
- La portabilité est le degré d'efficacité et d'efficacité avec lequel un système, un produit ou un composant peut être transféré d'un matériel, d'un logiciel ou d'un autre environnement opérationnel ou d'utilisation à un autre.

2.3. Bilan

Malheureusement, en pratique ces normes de qualité n'aident en rien à la production de logiciel de qualité.

- La norme ISO/CEI 90003 préconise une méthode de gestion de projet prédictive, mais nous l'avons vu elle est inadaptée au cas du logiciel.
- La norme ISO/CEI 12027 se base sur un modèle de développement en V. Mais, nous l'avons vu plus tôt, ce modèle n'offre pas les meilleures garanties de succès des projets.
- La norme ISO/CEI 25010 se présente comme une liste de caractéristiques à mesurer sur le produit fini. Mais, en développement logiciel, il n'existe pas d'indicateurs objectifs et incontestables pour mesurer la valeur de ces caractéristiques et donc pas d'outils permettant de graduer la qualité d'un logiciel.

Finalement, la mesure de ces indicateurs (KPI) est largement empirique et donc la qualité reste globalement subjective, ce qui va à l'encontre de l'ambition des normes.

3. Le point de vue de l'agilité

L'agilité renonce à l'application de ces normes qu'elle considère comme contre-productives pour se tourner vers les valeurs humaines dans les approches du développement.

- Facteurs externes : Le cycle de vie est basé sur un cycle itératif et incrémental qui repose sur une collaboration étroite entre l'équipe de développement et le client qui est le juge de la qualité.

- Facteurs internes : La qualité d'un logiciel résulte de la qualité de son code. Il ne suffit pas qu'un logiciel soit efficace, il faut en plus qu'il soit bien conçu. Cette vision est mise en pratique de façon emblématique par la méthode Agile *eXtreme Programming* (XP). En agilité, l'ingénieur développeur se voit comme un artisan du logiciel qui prône l'amour du travail bien fait.

Ainsi, les caractéristiques énumérées dans la norme ISO/CEI 25010 fournissent les points de focalisation de l'attention du développeur en recherche de qualité de code mais en aucun cas des mesures de qualité (il n'y a plus de notion de KPI).

3.1. Artisanat du logiciel (*Software craftsmanship*)

En l'absence de mesures objectives, l'artisanat du logiciel met l'accent sur les compétences de codage des développeurs.

Le code produit par l'équipe est considéré comme un << chef-d'œuvre >>.

Les facteurs de qualité du code à privilégier :

- **Simplicité** (Principe KISS : *Keep it Simple, Stupid*)
- Le code doit être << évident >>.
- **Propreté**
- Le code doit se lire comme sa propre documentation.
- **Testabilité**
- Le code doit être couvert par des tests automatiques.

3.2. Atelier de l'artisan

Pour aider à la qualité du code, il faut que l'équipe de développement se dote :

- d'un environnement de développement professionnel,
- des outils d'aide à l'analyse de la propreté du code.

3.2.1. Environnement de développement

Le développement nécessite des environnements de développement sophistiqués.

Cela inclut de façon non exhaustive :

1. IDE : Environnement de Développement Intégré (e.g.,: IntelliJ, CLion, Android Studio, Visual Studio)

- Permet une Edition avancée du code.
- Permet la compilation interactive.
- Permet le management de code.

2. Logiciel de gestion de versions (e.g.,: Git et *gitlab*).

- Partage de la production.
- Suivi de versions.

3. Moteur de production (i.e., super makefile) (e.g., pour le Java : Gradle ou Maven).

- Compilation.
- Déploiement.

4. Outils d'intégration continue (e.g., à l'école GitLab-CI/CD)

- Vérification automatique de l'opérationnalisation du logiciel.

5. Des outils de déverminage et de profilage de code.

3.2.2. Outils d'aide à l'analyse de la propreté du code

Pour aider à la propreté du code, les informaticiens se sont dotés d'outils d'analyse automatique de code. S'il est impossible, à l'heure actuelle, de mesurer la qualité d'un logiciel, il est possible de critiquer la qualité d'un code.

En Java, les outils les plus utilisés sont :

- **Checkstyle** : vérification des règles et conventions de codage.
- **PMD** : similaire à Checkstyle mais plus focalisé sur les problèmes potentiels de codage comme le code non utilisé ou sous-optimisé, la taille et la complexité du code.
- **FindBugs** : détection des bogues potentiels, des problèmes de performance, ou des mauvaises habitudes de codage.
- **SonarQube** : une référence dans le domaine de l'analyse de code (une combinaison de tous les outils précédents).

Ces outils d'analyse sont intensément utilisés pour identifier les parties de code qui doivent être retravaillées.

Ils peuvent être ajoutés sous forme de plugins dans les IDE. Je vous recommande d'installer au moins le plugin SonarQube.

Il existe des outils d'analyse de code nativement dans l'IDE IntelliJ IDEA :

- Voir le menu Analyze::inspect code.
- Il faut le configurer avec ses préférences (voir la plateforme pour un exemple de fichier avec une configuration reprenant les directives données en ODL).

4. Que retenir de ce chapitre

- La qualité logicielle est impossible à mesurer.
- Les normes de qualité issues de la famille ISO 9000 n'apportent aucune aide pour développer des logiciels de qualité.
- Si l'on s'intéresse à la qualité du point de vue des développeurs, les facteurs de qualité sont ;
 - La simplicité
 - La propreté du code
 - La testabilité
- En l'absence de mesure objective de ces facteurs, l'assurance qualité consiste à :
 - Adopter une posture d'artisan qui considère que la qualité du logiciel résulte de la qualité du code.
 - Utiliser des environnements de développement sophistiqués qui permettent d'obtenir des critiques de la qualité du code.
 - Les indicateurs de performance (KPI) ne doivent être considérés que comme des alarmes de parties de code à retravailler et pas comme des mesures de la qualité du logiciel.

Chapitre 4

Code propre

« Codez toujours comme si le type qui sera chargé de maintenir votre code est un violent psychopathe qui sait où vous habitez. »

John Woods

1. Objectif du chapitre

Ce chapitre définit la notion de code propre à travers les principes et la pratique.
Le codage a beaucoup changé ces 10 dernières années, ce que ne reflète pas l'énorme quantité de codes sales que l'on trouve sur Internet.

À l'issue de ce chapitre :

- Vous serez capable d'écrire du code de qualité professionnelle.
- Vous serez en mesure de reconnaître d'un coup d'œil du mauvais code.
- Vous serez sensibilisé à l'approche artisanale du logiciel (*software craftsmanship*) !

2. Pourquoi faire propre ?

2.1. Code propre en proverbes

- « Nous passons plus de temps à lire du code qu'à l'écrire. » Anonyme.
- « *N'importe quel programmeur peut écrire du code que l'ordinateur comprend. Les bons programmeurs écrivent du code que les humains peuvent comprendre.* » Martin Fowler.
- « *Don't comment bad code - rewrite it.* » B. W. Kernighan et P. J. Plaugher.

2.2. Pratique traditionnelle du codage

La plupart des manuels de programmation insistent sur des règles d'or :

- Commentez vos codes. Codez l'algorithme puis documentez-le avec des commentaires de code.
- Dans le corps d'une fonction, séparez les étapes de l'algorithme par des lignes vides afin de faire apparaître la structure de l'algorithme.
- Documentez vos programmes.
- Décrivez le principe de l'algorithme dans un cartouche en-tête du fichier.
- Commentez chaque fonction dans un cartouche en détaillant les paramètres et la valeur de retour.

La raison invoquée pour ces règles, c'est que les commentaires et la documentation sont indispensables au développeur qui maintiendra votre code, qui peut être soi-même des mois plus tard.

2.3. Pourquoi il ne faut pas coder comme cela

Cette pratique du codage paraît logique et pourtant elle est erronée :

- Tout développeur rechigne à faire de la documentation et des commentaires.

Cela a une conséquence sur la qualité et la maintenance des commentaires / documentations qui sont produits.

- Les commentaires créent du bruit dans le code.
- Les commentaires et la documentation prennent la plupart du temps.
- La documentation n'est jamais lue, c'est donc un temps précieux qui est perdu.

Tout cela va à l'encontre de la maintenabilité du code qui est pourtant le but recherché à travers cette pratique.

2.4. Les commentaires sont néfastes au code

2.4.1. Les commentaires mensongers

Le code évolue toujours plus vite que les commentaires. Les IDE permettent de renommer les variables, fonctions, classes, etc., mais pas les commentaires. Les commentaires désuets sont plus néfastes à la maintenance du code que pas de commentaire du tout.

Exemple de code trouvé dans des logiciels professionnels :

```
// Returns null if register doesn't exist
public void registerItem( Item item ) throws NoRegistryException {
...
}
// Always returns true for this device
public boolean isAvailable( ) {
return false;
}
```

2.4.2. Les commentaires non informatifs

Parce que chaque donnée ou fonction membre doit être commentée, la plupart des commentaires ne font que parodier le code.

```
// Default constructor
protected AnnualDateRule() { }
// The day of the month.
private int dayOfMonth;
// Returns the day of the month.
public int getDayOfMonth() {
return dayOfMonth;
}
```

2.4.3. Les commentaires trompeurs

Un commentaire bâclé peut cacher une partie des informations.

Par exemple, le commentaire suivant dit que la fonction retourne le statut de la lumière, mais omet de dire que si la lumière est éteinte elle réinitialise la lumière.

```
/*
 * Returns if light is on.
 */
bool getLightStatus( Light light) {
if (!light.isOn()) {

resetLight(light);
}
return light.isOn();
}
```

2.5. Les commentaires rendent le code illisible

2.5.1. Commentaires formatés avec des tags HTML

Ils sont illisibles. Il ne faut pas confondre commentaire avec documentation

```
/*
 * Task to run fit tests.
 * This task runs fitness tests and publishes the results.</p>
 * <pre>Usage:
 * &lt;taskdef name=&quot;execute-fitness-tests&quot;
 * classname=&quot;fitness.ant.ExecuteFitnessTestsTask&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 * resource=&quot;tasks.properties&quot; /&gt; <p/>
 * &lt;execute-fitness-tests
 * suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 * fitnessreport=&quot;8082&quot;
 * resultsdir=&quot;${results.dir}&quot;
 * resultshtmlpage=&quot;fit-results.html&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * </pre>
 */
```

2.5.2. Commentaires de fin de bloc

Ils sont inutiles pour les petites fonctions. Les grandes fonctions doivent être découpées en sous-fonctions

```
public static int main( String args[] ) {
try {
while ((line = in.readLine()) != null) {
lineCount++;
charCount += line.length();
String words[] = line.split("\\W");
wordCount += words.length;
} //while
System.out.println("wordCount = " + wordCount);
System.out.println("lineCount = " + lineCount);
System.out.println("charCount = " + charCount);
} // try
catch (IOException e) {
System.err.println("Error:" + e.getMessage());
} //catch
} //main
```

2.5.3. Commentaires de séparation de parties

Ils sont agressifs et obscurcissant. De plus, ils sont redondants pour qui connaît le langage de programmation.

```
public static SomeModule extends AbstractInteractionModule {
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
// VARIABLES
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
private int SomeVo _vo;
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
// CONSTRUCTOR
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
public SomeModule() {
}
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
// PUBLIC FUNCTIONS
// xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
public void someFunction() {
}
}
```

2.6. Les commentaires décrédibilisent le développeur

2.6.1. Les fautes d'orthographe

```
int i; /* Counter variabble for "for" loop. */
int t; /* Total of additions for calculaton */
int d; /* Individual number for calclatuion */
/* "for" loop */
for (i=0; i<100; i++) { /* incrément i by one until hunderd */
d = f(); /* get the calue for d */
t = t + d; /* ad it to t */
}
```

2.6.2. Les copier-coller malheureux

```
/* The version. */
private String version;
/* The licenceName. */
private String licenceName;
/* The version. */
private String info;
```

3. Code propre

3.1. Nouvelle pratique du codage

Le code est la seule chose qui soit réellement importante.

Il doit donc être le centre des attentions du développement :

Le code est la seule chose qui soit maintenue.

- Le code ne ment pas.
- Le code doit être sa propre documentation.
- Le code doit se lire comme une documentation.

Il faut supprimer tout le reste.

- Supprimer les commentaires.
- Supprimer la documentation sur tout ce qui est susceptible d'évoluer dans le temps.

Toutefois, supprimer les commentaires et la documentation nécessite de dépenser du temps et de l'énergie pour cela.

Que penseriez-vous d'un chirurgien qui ne se laverait pas les mains avant une opération sous prétexte que cela prend du temps.

Dans les sections qui suivent, nous présentons les règles générales d'écriture de code portant sur le nommage explicite des identificateurs et l'écriture des fonctions (méthodes) auto-documentées.

3.2. Note : cas des API publiques

Une API publique (ie, *Application Programming Interface*) est une bibliothèque de code qui est utilisée par d'autres développeurs pour coder leur logiciel. Contrairement au logiciel, il est impératif d'avoir une documentation de type *Doxygen* ou *Javadoc* qui indique comment utiliser les éléments de l'API. Les classes et les fonctions doivent être précédées d'un cartouche donnant toute information permettant leur exploitation.

Quand on participe à l'écriture d'API publiques la documentation est une **obligation**. Mais, dans ce cas, la production de cette documentation est une tâche de développement au même titre que le code. C'est un travail à part entière et il faut donc y consacrer du temps.

```
/**
 * Computes the matching between two region maps.
 * Only the common regions are kept in the result.
 * @param regionMap1 a regular region map.
 * @param regionMap2 a regular region map.
 * @result the region map with the common regions.
 */
RegionMap matching( RegionMap regionMap1, RegionMap regionMap2 ) {
...
}
```

Il faut noter que le prototype des fonctions, méthodes et classes d'une API est stable par essence. Il n'y a donc pas de risque d'obsolescence des commentaires. On ne change pas le prototype d'une fonction devenue obsolète, mais on l'annote << *deprecated* >> et on construit une nouvelle fonction.

3.3. Utiliser judicieusement les commentaires

Tous les commentaires ne sont pas inutiles. Certains sont même indispensables. Mais ils doivent rester rares.

3.3.1. Commentaires indispensables

- Compléments d'information sur des instructions non auto-documentables.

```
// Format matched hh:mm:ss GMT, MMM dd, yyyy
```

```
Pattern timeMatcher = Pattern.compile(
"\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

3.3.2. Commentaires acceptables

- Commentaire de mise en garde.

```
// Warning: do not remove this method. It is kept for the sake of compatibility.
void getMaximum( ArrayList<Cell> cells ) {
...
}
```

3.3.3. Commentaires d'aide à la programmation : TODO / FIXME

- Ce sont des traces laissées dans le code pour y revenir plus tard. Tous les IDE reconnaissent ces marqueurs. Toutefois, ces commentaires sont provisoires et doivent être supprimés dans la version poussée en production.

```
// TODO Change the sort algorithm to heap sort algorithm.
public void sort( Ordonable list ) {
...
}
```

4. Nommage des identificateurs

4.1. Utiliser avantageusement les noms d'identificateur

Les noms sont partout : identificateurs, nom de projet, de paquets... Il faut les rendre explicites. Ce sont les premiers commentaires d'un programme.

4.1.1. Choisir des noms explicites

- Un bon nommage rend caduque le commentaire.

Avant

```
int d; // elapsed time in days
```

Après

```
int elapsedTimeInDays;
```

- Un nommage judicieux participe à auto-documenter le code.

Avant

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : _theList) {  
        if (x[0] == 4) {  
            list1.add(x);  
        }  
    }  
    return list1;  
}
```

Après

```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<>();  
    for (Cell cell : _gameBoard) {  
        if (cell.isFlagged()) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```

4.1.2. Utiliser des noms plutôt que des commentaires

- Remplacer les commentaires par du code.

Avant

```
/ Check to see if the employee is eligible for full benefits
if ((_employee.flags & HOURLY_FLAG) && _employee.age > 65) {
/* ... */
}
```

Après

```
private final boolean isEligibleForFullBenefits() {
return (_employee.flags & HOURLY_FLAG) && _employee.age > 65;
}
if (isEligibleForFullBenefits()) {
/* ... */
}
```

On pourrait croire qu'il y a un surcout à l'exécution du fait de l'ajout d'une fonction. Mais avec le mot clé final devant la fonction, le compilateur peut procéder à une optimisation de type inline.

- Utiliser des fonctions ou des variables plutôt qu'un commentaire.

Avant

```
// does the module from the global list <module> depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(module.getSubSystem()))
```

Après

```
List<Module> moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = module.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

On pourrait croire qu'il y a un surcout de variables et donc d'occupation mémoire. Mais le compilateur construit lui-même ces variables intermédiaires : donc il n'y a aucun surcout à l'exécution.

4.1.3. Ne pas en faire trop

- Dans l'exemple ci-dessous, la première version suffit. Le renommage de la variable de boucle dans la deuxième version est inutile. La portée de la variable de boucle est limitée, donc on peut se contenter d'un nom générique pour la boucle tel que 'i'.

Première version

```
for (int i = 1; i < INDEX_SIZE; ++i) {
setMapIndex(i);
}
```

Deuxième version

```
for (int mapIndex = 1; mapIndex < INDEX_SIZE; ++mapIndex) {
setMapIndex(mapIndex);
}
```

4.2. Respecter des règles de nommage

4.2.1. Éviter l'ambiguïté

- Ne jamais utiliser la minuscule l ou la majuscule O comme nom de variable. Ils se confondent respectivement avec les chiffres 1 et 0.

```
int a = l;  
if (O == l) {  
    a = O;  
} else {  
    l = 0;  
}
```

4.2.2. Utiliser des noms prononçables, mnémotechniques et partageables

- Les noms imprononçables rendent le code confus.

Avant

```
public final class DtaRcrd102 {  
    private Date _genymdhms;  
    private Date _modymdhms;  
    private final String _pszqint = "102";  
    /* ... */  
}
```

Après

```
public final class Customer {  
    private Date _generationTimestamp;  
    private Date _modificationTimestamp;  
    private final String _recordId = "102";  
    /* ... */  
}
```

4.2.3. Utiliser des noms distinguables

L'intérêt de noms rapidement distinguables est de pouvoir :

- utiliser les outils de recherche informatisés,
- bénéficier de la complétion automatique des IDE.

Par exemple les noms suivants sont difficilement distinguables et recherchables :

XYZControllerForEfficientHandlingOfStrings

XYZControllerForEfficientStorageOfStrings

De façon générale, il faut éviter la convention de nommage dite des *Schtroumpfs* (*Smurf Naming Convention*),

quand presque toutes les classes ont le même préfixe, par exemple un *SmurfAccountView* transmet un *SmurfAccountDTO* au *SmurfAccountController*. Le *SmurfId* est utilisé pour récupérer un *SmurfOrderHistory* qui est passé au *SmurfHistoryMatch* avant de le transmettre à *SmurfHistoryReviewView* ou *SmurfHistoryReportingView*.

4.2.4. Ne pas avoir peur de faire des noms longs

Un nom long explicite est meilleur qu'un nom court énigmatique ou un commentaire.

En général, vous avez droit à un minimum de 255 caractères quel que soit le langage et le compilateur.

4.2.5. Utiliser une convention pour rendre les noms composés lisibles

Par exemple, le syntagme `ceciestunidentificateur` sans séparateur de mot est difficilement lisible. En adoptant une convention de séparation des mots, il devient plus lisible. Il existe plusieurs conventions :

- Notation PascalCase : `CeciEstUnIdentificateur`
- Notation camelCase : `ceciEstUnIdentificateur`
- Notation snake_case : `Ceci_est_un_identificateur`
- Notation SCREAMING_SNAKE_CASE : `CECI_EST_UN_IDENTIFICATEUR`
- Notation kebab-case : `ceci-est-un-identificateur`

Dans un langage, la convention à utiliser dépend du type de l'identificateur : variable, fonction, classe, paquet, etc. Elle est décrite dans le manuel associé au langage pour les langages les plus récents.

4.2.6. Passer du temps pour le choix des noms

Il faut essayer différents noms et vérifier leur pertinence en contexte. Les IDE modernes, tels que IntelliJ ou CLion, rendent le changement de nom trivial (faire *Shift+F6* sur l'identificateur).

4.2.7. Nommage des données membres (attributs)

Il faut utiliser une notation des données membres basée sur une astuce pour éviter des bugs de programmation dont certains sont difficilement détectables.

Bug classique (détectable par les IDE)

```
public class Bottle {  
    private int volume;  
    public Bottle( int volume) {  
        volume = volume;  
    }  
}
```

La correction immédiate est d'utiliser le `this`. Mais nous verrons plus loin qu'il ne faut pas faire cela.

```
public class Bottle {  
    private int volume;  
    public Bottle( int volume) {  
        this.volume = volume;  
    }  
}
```

Bug vicieux (indétectable par les IDE et pas de protection possible)

Soit le code où `name` est une donnée membre de la classe :

```
@Override
private void setFeatures( String pname, int age ) {
    this.name = name.toUpperCase()
    this.age = age;
}
```

De toute évidence, le développeur a voulu utiliser `pname` et pas `name`, mais le compilateur ne lui est d'aucune aide pour détecter cette erreur.

Autre bug vicieux (détectable par les IDE mais trop tard)

Soit l'instruction :

```
private int toto( int a ) { f(a,x); }
```

Puis, on renomme le paramètre `a` en `x`.

```
private int toto( int x ) { f(x,x); }
```

L'IDE détecte un masquage d'une donnée membre par un paramètre. En réaction, on renomme `x` en `t` pour éviter cela, mais il est trop tard, le bug est créé.

Solution pour le nommage des données membres

Utiliser l'astuce du préfixe `'_'` devant chaque nom de donnée membre.

```
private String _nom;
```

- Avantages

- Distinguer d'un coup d'œil une donnée membre d'un paramètre ou d'une variable.
- Profiter de la complétion automatique des IDE pour ne proposer que des données membres.
- Le compilateur permet de détecter les bugs précédents (*laissé en exercice*).

Remarque : Android utilise le préfixe `'m'` devant chaque donnée membre, mais nous le trouvons moins discret.

En C++, certains utilisent plutôt `'_'` en suffixe des données membres, mais, cela ne permet pas de bénéficier de la complétion automatique des IDE. Enfin, Python et Dart utilisent le préfixe `'_'` pour indiquer que le membre est privé et par conséquent comme toute donnée membre doit être privée, le langage impose donc un `'_'` devant le nom de toute donnée membre.

4.2.8. Nommage des fonctions / méthodes

Le nom des fonctions doit être un verbe ou une phrase intentionnelle :

- `depositPayment()`, `deletePage()`, ou `save()`.

Il faut utiliser les standards `get`, `is` et `set` pour les accesseurs et mutateurs.

```
List<Artist> artists = song.getArtist();
if (artists.isEmpty()) {
    song.setTag("Unknown artists");
}
```

4.2.9. Nommage des paquets en Java

En Java, les paquets doivent être nommés à partir de l'adresse web (URL) de l'équipe de développement à l'envers (+ snake_case) :

- org.eclipse.swt.graphics
- com.suptech.ecole.mon_projet.model
- com.suptech.ecole.mon_projet.view

5. Écrire des fonctions auto-documentées

5.1. Fonctions courtes

Le corps des fonctions doit être court (< 20 lignes). Un indice d'une fonction trop longue est la difficulté de donner un nom ou la nécessité de mettre des commentaires. S'il vous semble nécessaire d'ajouter des commentaires au code, divisez le corps de la fonction en sous-fonctions : << *Don't comment bad code, rewrite it.* >> Brian Kernighan (*auteur du 1er livre sur le C qui reste la référence*).

5.2. Fonctions d'une seule chose

Le corps d'une fonction ne doit pas contenir de ligne vide. Les lignes vides sont souvent utilisées pour aérer le code de la fonction en séparant les différentes parties. C'est un indice qu'il faut décomposer la fonction en sous-fonctions.

5.3. Un seul niveau d'abstraction

Le corps d'une fonction doit rester à un seul niveau d'abstraction.

Exemple d'un code sale mélangeant deux niveaux d'abstraction :

Avant

```
public int[] process() {  
    int[] array = getArray(); // 1er niveau d'abstraction  
    for (int i = 0; i < array.length; i++) {  
        if (array[i] < array[i - 1]) {  
            swap(array, i - 1, i);  
        }  
    }  
    removeTies(array); // 2e niveau d'abstraction  
    return array;  
}
```

Après

```
public int[] process() {  
    rearrange(array);  
    removeTies(array);  
    return array;  
}
```

5.4. Bannir le code lourd

- N'utilisez pas de **this.donnée** ou **this.methode()**.

L'astuce de nommage des données membres permet d'éviter cela.

Exemple de mauvaise pratique dans les constructeurs :

Avant

```
public class Bottle {  
    private int volume;  
    public Bottle( int volume) {  
        this.volume = volume;  
    }  
}
```

Après

```
public class Bottle {  
    private int _volume;  
    public Bottle( int volume) {  
        _volume = volume;  
    }  
}
```

- Le `this` utilise en préfixe rend l'écriture très lourde :

Avant

```
this.discriminant = sqrt(this.b * this.b - 4 * this.a * this.c);
```

Après : cette seconde écriture est plus lisible que la première :

```
_discriminant = sqrt(_b * _b - 4 * _a * _c);
```

- Le mot `this` ne doit être utilisé que comme argument d'une fonction ou pour l'appel d'un constructeur dans un autre constructeur.

Note

Remarque : en Python, il faut toujours écrire `self.donnee` ou `self.methode()`.

5.5. Bannir le code naïf

Le code naïf vous fait perdre toute crédibilité vis-à-vis des autres développeurs.

Code naïf :

```
(1) if (boolean == true)
(2) if (boolean != false)
(3) if (test) {
    return true;
} else {
    return false;
}
```

Code propre :

```
(1) if (boolean)
(2) if (boolean)
(3) return test;
```

En Java, les deux valeurs `true` et `false` ne doivent jamais être utilisées dans les tests, mais uniquement pour les affectations.

5.6. Bannir le code de « geek »

L'exemple typique est celui des '*Yoda conditions*' :

```
(1) if (185 == height)
(2) if (null == pointer)
```

C'est comme si vous écriviez : si 185 est sa taille et si nul est le pointeur. Le génie logiciel est le contraire du geekisme.

5.7. Bannir l'humour douteux

- L'humour est à manier avec précaution. En général si on relit des parties de code c'est pour y trouver des bugs – si en plus il y a un humour douteux sur ces lignes cela peut énerver.

- exemple trouvé dans du code professionnel :

```
#define TRUE FALSE // Happy debugging suckers
```

- ou encore :

```
int _pigeons = 0; // les clients de l'entreprise
```

6. Respecter des standards de formatage de code

Le formatage correspond à la mise en forme du code.

- Il doit améliorer la lisibilité du code.
- Il n'y a pas de norme mais des standards liés au langage.
- Respecter les règles générales vues en ODL.
- Il est important que tous les membres d'une même organisation partagent le même standard.

Chaque langage définit ses propres règles de formatage qu'il est impératif de respecter même si vous trouvez cela « moche ».

Il reste toutefois des adaptations possibles. Pour automatiser le respect de ce formatage, les IDE peuvent être configurés avec un fichier partageable par tous les membres de l'équipe (pour IntelliJ et Clion voir le fichier fourni sur la plateforme pédagogique du cours de génie logiciel).

6.1. Formatage en Java

Le format d'indentation professionnel en Java utilise l'écriture à l'Égyptienne ou l'accolade d'ouverture est à la fin de la ligne de déclaration :

```
int method( int p ) {  
    if (test) {  
    } else {  
  
    }  
    for (int i = 0; i < 10; i++) {  
    }  
}
```

Par contre, n'utilisez surtout pas, qui est par contre utilisé en C# :

```
if (test)  
{  
}  
else  
{  
}
```

Le « else » n'est pas un début d'instruction. Ce format occupe trop de lignes sur l'éditeur.

6.2. Formatage en général

Toujours encadrer les instructions unilignes par des **accolades**.

- Voir la faille de sécurité SSL sur les systèmes iPhone, iPad, iPod et Mac du 8 janvier 2014.

- La trop fameuse source de bug :

```
if (condition)
    statement
    other statement
```

puis si l'on supprime temporairement l'instruction sous le if, on introduit alors un bug :

```
if (condition)
    // statement
    other statement
```

- Le problème de l'imbrication de else (*dangling else*) :

```
if b0 then if b1 then S0 else S1
```

Que fait cette instruction ?

- Autre exemple avec des if imbriquées :

```
if (cond1)
    if (cond2)
        doSomething();
```

Maintenant, considérons que nous voulons faire doSomethingElse() quand cond1 n'est pas remplie. Donc :

```
if (cond1)
    if (cond2)
        doSomething();
    else
        doSomethingElse();
```

ce qui est évidemment faux puisque le else est associé avec if interne.

7. Ne pas faire d'optimisation prématurée

Les critères de performance s'opposent souvent aux critères de maintenabilité.

Une erreur classique est de rechercher en premier lieu la performance maximale, quitte à sacrifier la lisibilité du code et augmenter les risques d'erreurs.

Ces pseudo-optimisations n'auront généralement pas d'impact perceptible.

Les codes optimisés produisent plus facilement des erreurs difficiles à debugger.

Les développeurs sont obligés de passer beaucoup de temps pour trouver ces erreurs et les corriger. Ce temps perdu pour développer de nouvelles fonctionnalités ou d'optimiser des parties du code qui ont un réel problème de performance.

Le compilateur est bien souvent meilleur que le développeur pour l'optimisation des instructions.

- Ne pas faire d'optimisation que le compilateur peut faire.
- Pas de compromis à la lisibilité.

Par exemple, les optimisations ci-dessous sont inutiles, le compilateur fait lui-même ces optimisations :

```
perimeter = 6.28 * radius; // mis pour perimeter = 2*Math.PI*radius;  
int x = y << 1; // mis pour int x = y * 2;
```

Cela ne veut pas dire qu'il ne faut rien optimiser. Mais, les parties à optimiser sont analysées en particulier avec un profilage de code (*profiler*). Bien souvent l'optimisation consiste à changer l'algorithme et pas le code.

Toutefois, quand l'optimisation est crédible mais rend le code obscur :

1. Isoler le code optimisé dans une fonction ou une classe à part.
2. Commenter ce code à l'aide d'un cartouche comme dans le cas d'une API.

8. Code propre dans l'industrie (code review)

Chaque équipe de développement définit ses propres règles de formatage et de codage propres dans le respect des standards internationaux. Pour garantir le respect de ces conventions de codage, l'étape d'intégration d'une contribution d'un développeur est précédée d'une étape de relecture croisée du code (*code review*).

Par exemple :

- Chez **Ubisoft** : relecture par pair (ie un autre membre de l'équipe). Il s'agit de vérifier la conformité du code aux conventions d'écriture de l'entreprise Mais aussi de diffuser de la connaissance du code entre les programmeurs du projet.
- Chez **IBM** : relecture par un comité avant intégration (inclut la vérification des tests).
- Dans vos projets : faire de la relecture de code par un autre membre du groupe avant l'intégration.

9. Que retenir de ce chapitre ?

- Les commentaires sont une source de bruit : ils doivent être éliminés pour la plupart.
- La contre-partie est la propreté du code :
- Respect des standards de mise en forme.
- Nommage des identificateurs.
- Structuration en fonctions, courtes, explicites et à un seul niveau d'abstraction.
- Le code doit toujours être propre, comme c'est le cas d'une table d'opération pour un chirurgien.

10. Lecture

- Robert C. Martin, << *Clean Code - A Handbook of Agile Software Craftsmanship* >>, Prentice Hall, 2009.

Chapitre 5

Refonte de code (refactoring)

*« Si déboguer c'est supprimer des bugs, alors programmer ne peut être que de les ajouter. » **Edsger Dijkstra***

1. Objectif du chapitre

Ce chapitre présente la refonte de code (*code refactoring*), qui vise à améliorer sans cesse la qualité du code, et souligne son influence sur la façon de coder.

A l'issue de ce chapitre :

- Vous serez en mesure de lutter contre le pourrissement du code (*rotten code*).
- Vous saurez refondre du « code pourri » (*rotten code*).

2. Nécessité de la refonte de code

2.1. Qu'est-ce que la refonte de code ?

Définition : La refonte de code est le processus qui consiste à changer le code d'un système logiciel de telle manière qu'il n'altère pas le comportement extérieur du code, tout en améliorant sa structure interne.

- Elle va de pair avec la notion de « code propre ».
- Sa mise en œuvre nécessite :
- l'utilisation d'environnements de développement approprié (cf. menu « *refactor* » de IntelliJ).
- la présence de tests exécutables qui vont permettre de vérifier que l'on n'introduit pas de régression

2.2. Qu'est-ce que n'est pas la refonte de code ?

La refonte de code ne correspond pas à l'introduction de nouvelles fonctionnalités.

Elle ne correspond pas, non plus, à l'optimisation des performances qui conduit souvent à un code qui devient difficile à comprendre.

2.3. Pourquoi faire de la refonte de code ?

- Pour ne pas accumuler de **dette technique**.

- Pour améliorer la logique de conception du logiciel.
- Pour rendre le système plus simple à comprendre et donc à maintenir, étendre et vérifier. C'est le principe KISS : « Keep It Simple, Stupid ». Attention, faire simple, c'est compliqué et relevé généralement d'un processus d'affinage. Plusieurs refontes peuvent être nécessaires avant d'atteindre la simplicité.
- Pour aider à trouver les bugs de type Schrodinbug .

Remarque : L'amélioration du code correspond principalement à la suppression de code : « *La perfection n'est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à enlever.* » Antoine de Saint-Exupéry.

2.4. Quand faire de la refonte de code ? Une nouvelle façon de coder

Elle s'inscrit pleinement dans l'approche Agile. Elle intervient au moment de la création et au moment de la maintenance du code. Elle profite alors de la malléabilité du code.

- **Lors du développement.** Cela constitue une nouvelle façon de coder :

1. Codage « Quick and Dirty » : coder rapidement la fonctionnalité (code « sale ») en restant focalisé sur le développement de la fonctionnalité.
2. Coder les tests et vérifier que le code passe ces tests.
3. Refondre le code pour le rendre propre en s'assurant que les tests passent encore et éviter ainsi la régression de code.

- **Lors de la reprise du code** pour ajouter une nouvelle fonctionnalité ou corriger un bug. Profiter de cette reprise de code pour améliorer sa structure en application de la règle des boy-scouts : « Laissez le campement plus propre que vous l'avez trouvé en arrivant ».

2.5. Quelques indices de la nécessité d'une refonte : « *bad smells in the code* »

- Duplication de code.
- Longues méthodes.
- Grandes classes.
- Longue liste de paramètres.
- Nécessite de commenter le code.

3. Exemples de refonte de code

Voici une sélection de cas de refonte de code extraite du catalogue de Martin Fowler (<http://refactoring.com/catalog/index.html>).

3.1. Factoriser le code redondant

Vous voyez la même structure de code à plus d'un endroit et qui porte la même sémantique. C'est l'application du principe DRY (*Don't Repeat Yourself*). La duplication de code posera un problème de maintenance plus tard. Quand il s'agira de corriger un bug dans la partie dupliquée ou de la faire évoluer, il n'y a plus rien d'automatique qui rappellera qu'il faudra faire les mêmes modifications au code dupliqué.

- Rassemblez le code commun dans une méthode à part entière.

Avant :

```
int total(int a, int b) {  
    return a + b; // Duplication  
}  
double average(int a, int b) {  
    int sum = a + b; // Duplication  
    return sum / 2;  
}
```

Après :

```
int total(int a, int b) {  
    return a + b;  
}  
double average(int a, int b) {  
    int sum = total(a, b);  
    return sum / 2;  
}
```

Après la refonte du code, si l'on doit ajouter du code pour traiter le cas du dépassement de la valeur limite des entiers, les deux occurrences profitent de cet ajout automatiquement.

3.2. Remplacer un « nombre magique » par une constante symbolique

Vous avez un nombre avec un sens particulier (*magic number*).

- Créez une constante, nommez-la en fonction de son rôle, et remplacez le nombre par cette constante.

Avant :

```
double potentialEnergy( double mass, double height ) {  
    return mass * 9.81 * height;  
}
```

Après :

```
static final double GRAVITATIONAL_CONSTANT = 9.81;  
double potentialEnergy( double mass, double height ) {  
    return mass * GRAVITATIONAL_CONSTANT * height;  
}
```

3.3. Supprimer les doubles négations

Vous avez une condition avec une double négation.

- Rendez cette condition positive.

Avant :

```
if (!item.isNotFound()) { }
```

Après :

```
if (item.isFound()) { }
```

3.4. Remplacer une méthode par un objet méthode

Vous avez une longue méthode qui utilise des variables locales.

- Transformez la méthode en une classe interne de telle manière que les variables locales deviennent des attributs de cette classe puis décomposez-la en sous-méthodes privées (alias *abstraction hypostatique*).

Avant :

```
public final class Order {  
    float price(int a, Obj o) {  
        double primaryBasePrice;  
        double secondaryBasePrice;  
        double tertiaryBasePrice;  
        // long code  
    }  
}
```

Après :

```
public final class Order {  
    float price() {  
        return new PriceCalculator.compute();  
    }  
    private final class PriceCalculator {  
        private double _primaryBasePrice;  
        private double _secondaryBasePrice;  
        private double _tertiaryBasePrice;  
        protected static float compute() {  
            // long code qui utilise en outre la méthode method1, method2...  
        }  
        private int method1() {  
            ... // utilise les attributs  
        }  
        ...  
    }  
}
```

3.5. Introduire une variable explicative

Vous avez une expression conditionnelle difficilement compréhensible.

- Mettez le résultat de l'expression, ou une partie de cette expression, dans une variable temporaire avec un nom qui explicite son rôle.

Avant :

```
if (platform.indexOf("MAC") > -1  
    && browser.indexOf("IE") > -1  
    && wasInitialized() && resize > 0) {  
    // code  
}
```

Après :

```
boolean isMacOs = platform.indexOf("MAC") > -1;  
final boolean isIEBrowser = browser.indexOf("IE") > -1;  
final boolean wasResized = resize > 0;  
if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {  
    // code  
}
```

3.6. Préserver un objet entier

Vous utilisez plusieurs valeurs d'un objet et passez ces valeurs comme paramètres d'une méthode.

- Envoyez l'objet en entier.

Avant :

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

Après :

```
withinPlan = plan.withinRange(daysTempRange());
```

3.7. Remplacer les conditions par le polymorphisme

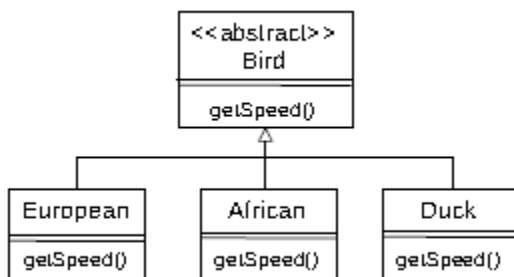
Vous avez une expression conditionnelle qui sélectionne différents comportements en fonction du type de l'objet.

- Créez des sous-classes et déplacez chaque partie de l'expression conditionnelle dans une méthode redéfinie dans une sous-classe puis utilisez le polymorphisme.

Avant :

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN: return getBaseSpeed();
        case AFRICAN: return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case DUCK: return (_isWood) ? 0 : getBaseSpeed();
    }
}
```

Après :



3.8. Remplacer un code d'erreur par une exception

Vous avez une méthode qui retourne un code spécial pour indiquer une erreur.

- Levez une exception à la place.

Avant :

```
int withdraw( int amount ) {  
    if (amount > _balance) {  
        return -1; // Cas erreur  
    }  
    _balance -= amount;  
    return 0;  
}
```

Après :

```
void withdraw( int amount ) throws BalanceException {  
    if (amount > _balance) {  
        throw new BalanceException();  
    }  
    _balance -= amount;  
}
```

3.9. Remplacer un paramètre par une méthode

Vous avez un objet qui appelle une méthode, et passe le résultat comme paramètre d'une autre méthode.

- Supprimez le paramètre et laissez le receveur appeler la méthode.

Avant :

```
int basePrice = _quantity * _itemPrice;  
discountLevel = getDiscountLevel();  
double finalPrice = discountedPrice(basePrice, discountLevel);
```

Après :

```
int basePrice = _quantity * _itemPrice;  
double finalPrice = discountedPrice(basePrice);  
// getDiscountLevel() est appelée à l'intérieur de discountedPrice()
```

3.10. Introduire un objet paramètre

Vous avez un groupe de paramètres qui vont naturellement ensemble.

- Remplacez les paramètres par un objet qui les rassemble.

Avant :

```
amountInvoicedIn(start: Date, end: Date)
amountReceivedIn(start: Date, end: Date)
amountOverdueIn(start: Date, end: Date)
```

Après :

```
amountInvoicedIn(DateRange)
amountReceivedIn(DateRange)
amountOverdueIn(DateRange)
```

4. Que retenir de ce chapitre ?

- Le code logiciel est malléable. Il faut en profiter pour faire une amélioration continue de la propreté et de la qualité du code.
- La refonte de code permet d'améliorer la structure interne du code.
- La refonte s'inscrit après le codage d'une fonctionnalité et au moment de la reprise de code (correction de bug, ajout d'une fonctionnalité).
- Elle ne peut pas se faire sereinement sans tests dynamiques qui permettent de se prémunir contre la régression. C'est l'objet du chapitre suivant.

Savoir quand et comment faire de la refonte de code est un art et une quête perpétuelle.

Le travers du « geek » est le « *refactoring* » qui est le processus qui consiste à prendre un morceau de code bien conçu et grâce à une série de petites modifications irréversibles, à le rendre impossible à maintenir par quiconque sauf par lui-même.

5. Lectures

- Martin Fowler, *Refactoring « Improving the Design of Existing Code »*, Addison-Wesley Professional, 1999.
- Consultez la liste des cas de refonte de code :
- Martin Fowler, « *Refactoring Home Page* », <http://refactoring.com/catalog/index.html>

Chapitre 8

Test logiciel

*« Durant le débogage, les novices insèrent du code correctif, alors que les experts suppriment du code défectueux. » **Richard Pattis***

1. Objectif du chapitre

Ce chapitre porte sur une initiation aux tests logiciels et en particulier aux tests dynamiques.

A l'issue de ce chapitre :

- Vous serez sensibilisé à l'importance des tests logiciels dynamiques.
- Vous serez capable de construire du code testable.
- Vous saurez programmer des tests unitaires et utiliser des «< bouchons >> (ie, *mock*).

2. Généralités sur les tests

2.1. Pourquoi tester ?

- **Seul moyen pour chasser les bugs.** Le code zéro bug n'existe pas, sauf exceptions. Les bugs sont inhérents à l'activité de codage (rappel de l'estimation : 1 à 10 bugs / KLOC). De plus, il n'existe pas de preuve formelle pour des programmes quelconques, donc pas de programme automatique pouvant calculer une preuve de programme. Mais il est possible de détecter certains bugs en testant les programmes pour limiter la casse.
- **Éviter les mauvaises surprises** comme la découverte de bugs après la mise en production.
- **Prémunir contre la régression de code.** Les tests sont aussi des garde-fous pour détecter la régression qu'un développeur peut introduire de façon involontaire lorsqu'il modifie le code du logiciel. Les tests de non-régression assurent que les modifications du code lors de la refonte de code, de l'ajout de fonctionnalités ou de la correction de bugs n'affectent pas le bon fonctionnement des fonctionnalités préexistantes.
- **Rassurer le développeur.** Lorsque le code a atteint une certaine complexité, on commence à craindre les modifications.
- **Rassurer le client.** Le produit final contient des preuves de test du produit.

2.2. Bug

Un bug est un défaut de conception ou de réalisation d'un logiciel qui provoque un dysfonctionnement.

Le mot a été popularisé en 1947 par **Grace Hopper** pionnière de l'informatique.

Il existe plusieurs catégories de bug :

- **Bohrbug** (inspire de l'atome de Niels Bohr) : un bug qui a toutes les bonnes propriétés, en particulier, il est répétable dans les mêmes conditions. C'est le bug classique.

- **Heisenbug** (inspire du principe d'incertitude de Heisenberg) : un bug dont le comportement est modifié quand on essaye de l'isoler. Le cas typique est celui des exécutions sous devermineur. Comme le devermineur réserve plus de mémoire à l'exécution que l'exécution normale, une erreur d'adressage mémoire qui apparaît lors de l'exécution du logiciel (le fameux « Segmentation fault ») peut ne pas se révéler sous devermineur parce que, par malchance, la mauvaise adresse utilisée fait partie du bloc de mémoire réservé.
- **Mandelbug** (inspire des fractales de Mandelbrot) : un bug dont les étapes pour le reproduire sont si complexes qu'il semble se reproduire de façon aléatoire et chaotique. Par exemple, les situations de compétition entre « thread » peuvent entraîner des Mandelbug ou le comportement du programme est différent à chaque fois que celui-ci est exécuté.
- **Schrödinbug** (inspire du chat de Schrodinger) : un bug qui ne se révèle pas à l'exécution, mais qui est découvert lorsque quelqu'un relit le code source ou utilise le logiciel d'une façon inhabituelle.

2.3. Qu'est-ce qu'un test ?

Un test est un procédé de vérification et validation (V&V).

- Vérification : **le logiciel fonctionne-t-il correctement ?**
- Définition ISO 9000 : confirmation par l'examen et la fourniture de preuves objectives que des exigences spécifiées ont été remplies.
- Validation : **a-t-on construit le bon logiciel ?**
- Définition ISO 9000 : confirmation par l'examen et la fourniture de preuves objectives que les exigences, pour un usage ou une application voulue, ont été remplies.

2.4. Niveaux de test

On distingue globalement trois niveaux de test, ce que schématise la pyramide des tests.

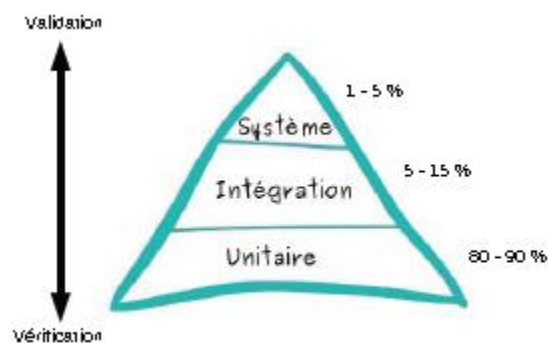


Figure 1 : La pyramide des tests.

Figure 1 : La pyramide des tests.

De bas vers le haut, on distingue :

1. Tests unitaires : Ils ont pour but de guider le développeur. Ce sont des tests en isolation des unités de développement.
- Erreurs recherchées : les erreurs de codage et les erreurs fonctionnelles dans les unités.

2. Tests d'intégration : Ils ont pour but de guider l'équipe de développement. Ce sont des tests qui portent sur l'assemblage des unités.

- Erreurs recherchées : les erreurs d'interface entre unités.

3. Tests système : Ils ont pour but de critiquer le produit. Ils testent le système dans son ensemble et en particulier l'interaction homme machine (IHM).

- Erreurs recherchées : l'absence ou défaillance de fonctionnalités.

- En production, on distingue plusieurs niveaux de tests du système.

- Version alpha : tests auprès d'utilisateurs internes au projet.

- Version beta : tests auprès d'utilisateurs externes mais avertis.

Les tests unitaires sont essentiellement de l'ordre de la vérification tandis que les tests systèmes sont essentiellement de l'ordre de la validation.

2.5. Tests dynamiques

Dans ce qui suit, nous nous focalisons sur les tests dynamiques. Un test dynamique est un bout de code qui est exécuté avec l'intention de vérifier ou valider un bout de code fonctionnel.

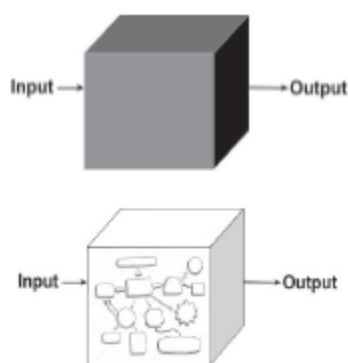
Remarque : le test logiciel inclut aussi des tests manuels qui ne sont pas présentés ici (Voir les quadrants de test Agile dans la littérature).

2.6. Types de tests dynamiques

1. Tests **boîte noire** (*black-box testing*)

- L'écriture des tests se fait sans connaître la structure interne du code à tester.

- Ils ne s'intéressent qu'aux entrées et sorties.



2. Tests **boîte blanche** (*white-box testing*)

- L'écriture des tests tient compte de la structure interne de l'unité testée.

En particulier, ils testent chaque chemin possible dans le code, par exemple chaque branche d'un 'if'.

2.7. Exemple fil rouge

On suppose une classe Human possédant les méthodes publiques suivantes :

- void setAge(int age)
- void setAgeLimit(int age)
- boolean isAdult() throws Exception, qui :
- lève une exception si $\text{age} \notin [0, \text{ageLimit}]$
- retourne true si $\text{age} \in [18, \text{ageLimit}]$
- retourne false si $\text{age} \in [0, 18[$

2.7.1. Étape 1 : Objectif de test

On choisit une caractéristique ou une fonction à tester : c'est l'objectif de test.

- Par exemple : On décide de tester la méthode isAdult() dans le cas nominal ou le paramètre age est dans $[18, \text{ageLimit}]$.

2.7.2. Étape 2 : Jeu de test

Ensuite, on choisit les données pour le test : c'est le jeu de test.

Pour notre exemple, il faut choisir :

- une valeur pour le seuil d'âge limite, par exemple $\text{ageLimit} = 150$.
- une valeur pour le paramètre age qui soit dans l'intervalle : $[18, \text{ageLimit}]$, par exemple : $\text{age} = 35$.

2.7.3. Étape 3 : Fixture

Avant d'exécuter le code du test, il faut amener l'objet dans l'état attendu : c'est la fixture.

- Dans notre exemple : création de l'objet Human et positionnement de son état.

```
Human h = new Human();  
h.setAgeLimit(150);
```

2.7.4. Étape 4 : Oracle

On compare le résultat obtenu au résultat attendu : c'est l'oracle. Il est implémenté par une assertion, ie, une expression booléenne censée être vraie.

- Dans notre exemple :
- h.isAdult() retourne true

2.7.5. Étape 5 : Verdict

On en déduit si le test a réussi ou échoué : c'est le verdict.

- si on récupère true : le test passe
- si on récupère false : le test échoue
- s'il y a une levée d'exception non attendue, c'est une erreur, le test est inconclusif.

2.7.6. Étape 5 : Test exécutable : Cas de test

L'ensemble du test exécutable s'appelle un cas de test.

Définition IEEE 610 : Un cas de test est un ensemble de **données de test**, de **pré-conditions** d'exécution et de **résultats attendus** développés pour un objectif, tel qu'exécuter un chemin particulier d'un programme (test de type boîte blanche) ou vérifier le respect d'une exigence spécifique (test de type boîte noire).

```
Human h = new Human(); // fixture
h.setAgeLimit(150); // donnée de test
try {
    h.setAge(35); // donnée de test
    if (h.isAdult()) { // oracle
        System.out.println("test passe"); // verdict
    } else {
        System.out.println("test échoué");
    }
} catch (Exception e) {
    System.out.println("erreur, test inconclusif");
}
```

2.8. Quand s'arrêter de tester ?

Les tests sont incomplets par nature. On a testé isAdult() pour une valeur d'entrée seulement, mais il faudrait les tester toutes. Ici, ce n'est pas possible puisque que $\text{age} \in \mathbb{N}$. Savoir quand s'arrêter est une question d'expérience. Un indice toutefois, la **couverture** des tests.

2.9. Couverture de code

La couverture de code est une mesure qui permet d'identifier la proportion du code testé. Elle correspond au ratio de code source qui est exécuté quand une suite de test est lancée.

Il existe des utilitaires de couverture de code dans pratiquement tous les langages (par exemple Jacoco en Java).

2.10. Doit-on écrire des tests pour tout ?

Non, uniquement pour les choses qui peuvent raisonnablement être source de bug. On n'écrit pas de test pour des instructions qui peuvent être vérifiées par l'OS, l'environnement d'exécution ou le compilateur.

Prenons l'exemple de la classe AClass suivante :

```
public class AClass {  
    int _x;  
    public AClass(int x) { _x = x; }  
    int getX() { return _x; }  
    void setX(int x) { _x = x; }  
}
```

On ne teste pas `getX()` et `setX()`. Tester `getX(setX(y))==y` revient à tester `_x=y`, c'est-à-dire à tester le compilateur.

2.11. À quelle fréquence dois-je exécuter mes tests ?

Exécutez le test unitaire aussi souvent que possible, idéalement à chaque changement dans le code.

2.12. Limite des tests

Mais, le test ne certifie pas le code : << *Le test de programme peut être utilisé pour prouver la présence de bugs, mais jamais leur absence* >> – Edsger Dijkstra.

Parfois le code est faux, mais les tests aussi !

- Question : Doit-on faire des tests de tests ?

3. Test unitaire

Le test unitaire vise deux objectifs de vérification d'une unité en isolation.

1. Vérifier le bon fonctionnement d'une unité du logiciel.
2. Se prémunir contre la régression de code dans l'unité.

En programmation orientée objet, l'**unité** est la **classe**. A chaque classe du code source, on associe une autre classe qui la teste.

Les tests unitaires testent une seule classe et sont indépendants les uns des autres (test en isolation). Cela permet de s'assurer que si un test échoue, c'est la classe testée qui est fautive.

Par exemple, on teste les méthodes publiques et protégées de la classe `TelecommandeTV` en isolation. Si on teste la classe `TelecommandeTV` en utilisant une télévision dans les tests, c'est du **test d'intégration**.

3.1. Qualités d'un test unitaire : FIRST

Un test doit impérativement posséder les qualités suivantes :

- **[F]**ast (Rapide) : plusieurs centaines de tests par seconde.
- **[I]**solated (Isole) : le test ne doit dépendre d'aucun autre test. Les tests peuvent être exécutés dans n'importe quel ordre.
- **[R]**epeatable (Répétable) : chaque exécution doit produire le même résultat quel que soit le moment ou l'environnement d'exécution. Cela interdit par exemple d'utiliser des valeurs aléatoires non reproductibles.
- **[S]**elf-validating (Auto-évaluable) : pas de recours à un utilisateur pour l'oracle.
- **[T]**imely (Juste à temps) : programme des que l'on a la connaissance sur la fonctionnalité.

3.2. Testabilité d'un code

On n'écrit pas du code testable comme du code classique.

Prenons l'exemple d'une alarme qui se déclenche à une heure butoir dans un agenda.

- On part du très mauvais code suivant :

```
public final class Agenda {  
    public void check() {  
        if (System.currentTimeMillis() > 100) {  
            new Bell().ring();  
        }  
    }  
}
```

Voyez-vous pourquoi il est mauvais ? Ce code est mauvais parce qu'il n'est pas testable automatiquement.

3.2.1. Contrôler une entrée indirecte

La méthode check() a deux données de test :

- la date à laquelle l'alarme se déclenche,
- la date courante.

Mais, dans check() la date courante est nécessairement fournie par System.

Mais, System étant incontrôlable, **ce code est mauvais parce qu'il n'est pas testable.**

La date courante est une entrée indirecte de check() qu'il faut pouvoir contrôler.

3.2.2. Observer une sortie indirecte

L'objectif du test est :

- Vérifier que l'alarme se déclenche si la date courante dépasse la date butoir.

Quel est l'oracle ?

- Ecouter si on entend ou pas quelque chose. Mais ce n'est pas auto-évaluable.
- Observer si la méthode ring() de Bell a été appelée. Mais l'objet Bell est créé dans check(), il n'est pas observable.

Dans les deux cas, **ce code est mauvais parce qu'il n'est pas testable.** L'interaction avec Bell est une sortie indirecte de check(), qu'il faut pouvoir observer.

3.2.3. Code testable

Solution : Encapsuler et externaliser des dépendances pour rendre ce code testable.

```
public final class Agenda {
    public Agenda(int limit, Clock clock, Bell bell) {
        _clock = clock; _bell = bell; _limit = limit;
    }
    public Agenda() {
        _clock = System; _bell = new Bell(); _limit = 100;
    }
    public void check() {
        if (_clock.getTimeInMillis() > _limit) {
            _bell.ring();
        }
    }
}
```

Il y a deux constructeurs. Le constructeur par défaut est pour l'exécution normale et le constructeur avec paramètres contrôlables est pour les tests.

4. Frameworks de test : JUnit

JUnit est un exemple de framework de test (peut être le plus connu). Il permet de faciliter l'écriture de test pour le langage Java. Ce qu'il offre :

- des assertions expressives pour automatiser le verdict,
- la visualisation du verdict,
- la possibilité de lancer facilement les tests.

4.1. Organisation des codes

Les tests ne doivent pas être dans la source de l'appliquatif.

Une organisation classique contient les dossiers suivants :

- **bin** (ou **build**) pour les exécutables.
- **src** pour le code source applicatif.
- **test** pour le code source des tests.

L'organisation en paquets des tests suit celle des sources.

Par exemple, pour tester la classe `com.suptech.ecole.projet.MaClasse` du dossier `src`, on trouve la classe `com.suptech.ecole.projet.MaClasseTest` dans le dossier `test`.

Avantages :

- Pas de pollution des sources par les tests.
- Permet de livrer l'appliquatif avec ou sans les tests.

4.2. Les assertions de base de JUnit

Les assertions servent à décrire l'oracle d'un cas de test. Ce sont des variations autour du `assert` :

- `assertTrue(boolean condition)`
- `assertFalse(boolean condition)`
- `assertEquals(Object expected, Object actual)`
- `assertEquals(double expected, double actual, double delta)`
- `assert[Not]Same(Object expected, Object actual)`
- `assert[Not]Null(Object actual)`
- `assertArrayEquals([] expecteds, [] actuals)`
- `fail()`...

4.3. Méthode de test JUnit : `@Test`

L'écriture d'une méthode de test dans une classe de test représentant un cas de test, doit être : annotée par `@Test`.

- publique, sans paramètre et de type de retour `void`.

Exemple reprenant le cas d'étude `Human` :

```
@Test
public void test_is_an_adult_when_age_greater_than_18() {
    Human h = new Human();
    h.setAgeLimit(150);
    h.setAge(35);
    assertTrue(h.isAdult());
}
```

4.4. Tester la levée d'exception

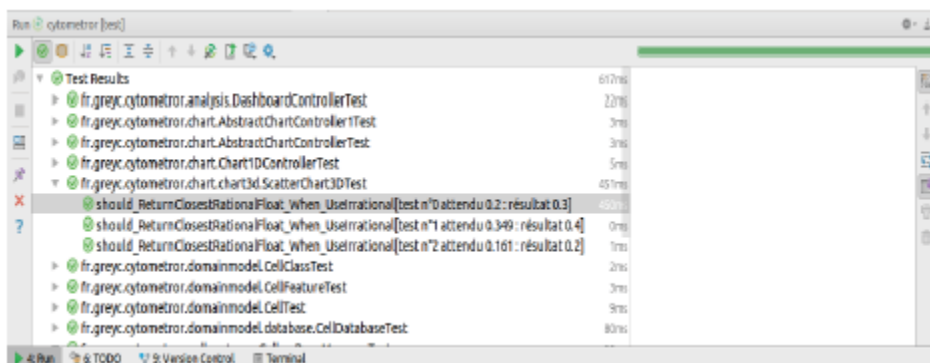
Exemple : vérifier que l'appel de la méthode `setAge()` lève une exception quand le paramètre est hors limite. Si l'exception est levée alors le verdict est positif.

```
@Test
public void test_throw_exception_if_age_is_greater_than_the_maximum()
    Assertions.assertThrows(OutOfLevelException.class,
        () -> {
            Human h = new Human();
            h.setAgeLimit(150);
            h.setAge(151);
        });
}
```

4.5. Visualisation du verdict de JUnit

Le verdict est représenté graphiquement dans un IDE :

- Le test passe : **barre verte**.
- Le test échoue : **barre rouge**.
- Levée d'une exception attendue : **barre verte**.
- Levée d'une exception inattendue : **barre rouge**.



4.6. Fixture JUnit : @BeforeEach

Préambule des tests : une méthode annotée par @BeforeEach.

Elle est appelée avant chaque méthode de test de la classe.

- Elle sert à factoriser la fixture des tests si elle est commune à tous les tests de la classe.

```
public final class test_level_management {  
    private Human _human;  
  
    @BeforeEach  
    public void setUp() throws Exception {  
        _human = new human();  
        _human.setAgeLimit(150);  
    }  
  
    @Test  
    public void test_age_greater_than_max(){  
        Assertions.assertThrows(OutOfLevelException.class,  
            () -> {  
                h.setAge(151);  
            });  
    }  
}
```

5. Frameworks de test : Mockito

Mockito est le plus connu des frameworks qui permettent l'écriture de « doublures » pour le langage Java.

5.1. La doublure pour les tests

Une doublure remplit deux rôles :

1. Le substitut (*fake*) : une classe qui est une implémentation partielle et qui retourne toujours les mêmes réponses selon les paramètres fournis sans fournir de code pour cela (donc pas de bug inséré).
2. L'espion (*spy*) : une classe qui vérifie l'utilisation d'une classe doublée après son exécution.

Remarque : Les doublures peuvent aussi être employées dans le code fonctionnel pour remplacer une classe non encore développée.

Dans ce cas, on parle de « bouchon » (*mock* en anglais).

5.2. Pourquoi des doublures ?

Les doublures offrent une solution pour développer des tests qui nécessitent :

- un composant dont le code n'est pas encore disponible.
- p. ex : persistance des données.
- un composant difficile à mettre en place.
- p. ex : une base de données.
- un comportement exceptionnel d'une classe.
- p. ex : déconnexion dans un réseau.
- un composant dont le code est lent (*empêche le F de FAST*).
- p. ex : construction d'un maillage.
- une fonction qui a un comportement non-déterministe.
- p. ex : réseau.

5.3. Exemple d'utilisation du framework Mockito

On suppose la classe `MaClasse` à doubler :

1. D'abord on importe Mockito dans le fichier test :

- `import static org.mockito.Mockito.*;`

2. On crée l'objet de la classe à tester en utilisant les doublures à la place des objets réels avec la méthode `mock()` :

- `MaClasse mc = mock(MaClasse.class);`

3. On décrit le comportement attendu de la doublure :

- `when(mc.maMethode()).thenReturn(56);`
- `when(mc.maMethode()).thenThrow(new Exception());`

4. On vérifie que l'interaction avec les doublures est correcte :

- `mc.verify()` avec les bons tests.

5.4. Exemple d'utilisation pour le substitut pour une classe « ServiceAuthentification »

Le test suivant vérifie que la méthode vérifie() de la classe MessagerieUtilisateur fonctionne correctement quand on donne un mot de passe correct ou non.

Le problème, c'est que la méthode fait appel à une classe tierce ServiceAuthentification.

Il est donc nécessaire d'utiliser une doublure pour garantir la qualité << isolée >> du test.

```
@Test
public void testLireMessagesAvecBonMotDePasse() {
    ServiceAuthentification sa = mock(ServiceAuthentification.class);
    when(sa.verifie("toto", "mdp")).thenReturn(true);
    when(sa.verifie("toto", "mauvais_mdp")).thenReturn(false);
    // On introduit la doublure dans la classe à tester.
    MessagerieUtilisateur msg = new MessagerieUtilisateur(sa);
    // Oracle
    assertTrue(msg.lireMessages("toto", "mdp"));
    assertFalse(msg.lireMessages("toto", "mauvais_mdp"));
}
```

5.5. Exemple d'utilisation pour l'espionnage pour une classe « ServiceAuthentification »

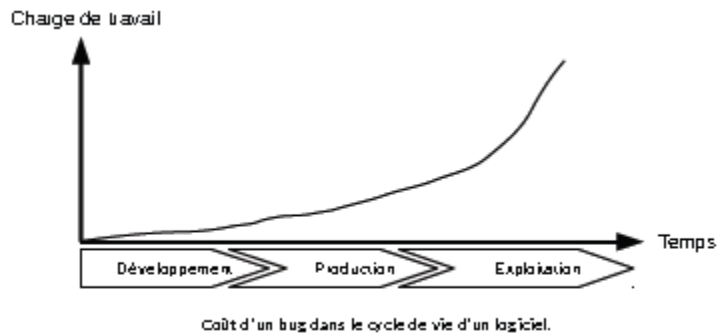
Le test suivant vérifie que la méthode vérifie() de la classe MessagerieUtilisateur fait bien appel à la méthode vérifier() de la classe tierce ServiceAuthentification exactement une fois pour s'exécuter.

```
@Test
public void testLireMessages() {
    ServiceAuthentification sa = mock(ServiceAuthentification.class);
    Mockito.when(sa.verifier("toto", "mdp")).thenReturn(true);
    Mockito.when(sa.verifier("toto", "mauvais_mdp")).thenReturn(false);
    // étape 1 : on introduit la doublure
    MessagerieUtilisateur msg = new MessagerieUtilisateur(sa);
    // étape 2 : on lance le traitement
    msg.lireMessages("toto", "mdp");
    msg.lireMessages("toto", "mauvais_mot_de_passe");
    // étape 3 : on vérifie que la méthode vérifier() a bien été // appelée exactement une fois avec ces
    paramètres.
    Mockito.verify(sa, times(1)).vérifier("toto", "mdp");
}
```

6. Développement dirigé par les tests

6.1. Quand tester ?

Le coût de correction d'un bug est exponentiel avec l'avancement dans le cycle de vie du projet. Il faut donc tester son logiciel le plus tôt possible.



6.2. TDD : Test Driven Development

Et si, les tests étaient programmés avant la fonctionnalité ? On aboutit à une nouvelle façon de développer : le développement dirigé par les tests (TDD). On écrit le code du test juste avant d'écrire le code de la fonctionnalité à tester.

La règle d'or du TDD : << Ne jamais écrire une ligne de code fonctionnel sans qu'une ligne de code de test ne l'exige. >>

6.3. En pratique

- **Le mantra du TDD : Toujours garder la barre verte pour garder le code propre.**

- Red → Green → Refactor → Green

- Plus précisément :

1. Ecrire un test pour une fonctionnalité à développer.
2. Exécuter et constater que la barre est **rouge**.
3. Ecrire le code qui permet de faire passer le test (et rien que ce code).
4. Lancer le test et vérifier qu'il passe : barre **verte**.
5. Remanier le code : garder la barre **verte**.
6. Relancer tous les tests précédents : garder la barre **verte**.

6.4. Pourquoi écrire les tests avant le code ?

6.4.1. Avantages psychologiques

- Travailler plus sereinement.
 - Si un bug apparaît, il sera détecté très tôt par les tests.
- Grâce au code testé:
- On est serein.
 - On planifie mieux son travail.
 - On évite les paniques de dernière minute.
 - Rester focalisé sur la tâche.
 - Oblige à réfléchir à ce que doit faire le code avant de l'écrire.

6.4.2. Avantages techniques

- Garder un temps de développement constant. Tout au long du développement, il y aura le même temps consacré au test et le même temps consacré au développement.
- On passe beaucoup moins de temps à débogué.
- On a toujours quelque chose à montrer au client, dont des tests. Il est impossible de livrer du code non testé. Les tests de **non-régression** sont directement inclus.
 - Quand on écrit le test, on ne se réfère qu'à la spécification et pas à l'implémentation (test boîte noire). Il y a moins de biais et moins de dépendance au code.
 - Le code de test sert aussi de documentation du code.

6.5. Pourquoi écrire un seul test à la fois ?

- Développement itératif.
- Optique du «< petit pas >>».
- Le test représente une partie du contrat total de la fonctionnalité testée.
- On ne passe au développement de la fonctionnalité suivante que lorsque la précédente a été validée.
- Pourquoi ne pas écrire tout le code fonctionnel d'un coup ?
- On n'écrit que le code qui a besoin d'être validé par un test sinon on risque d'introduire du code applicatif non testé ou d'ajouter des fonctionnalités inutiles.

6.6. Pourquoi commencer par la barre rouge ?

Il s'agit d'éviter les «< happy tests >>» : des tests qui s'exécutent avec succès (barre verte) alors qu'ils ne devraient pas (barre rouge).

Les causes possibles des happy test sont :

- Le test est construit par copier-coller d'un autre test sans modification.
- Le test est construit sans l'annotation `@Test` en JUnit.
- Le test ne teste pas réellement la méthode cible.

7. Correction de bug

La correction de bugs occupe une part importante de la charge d'un développeur.

7.1. Mauvaise pratique

- Pratique classique (parfois enseignée)
 1. Truffer le code d'appels d'affichage (*printf*) pour localiser la partie du code fautive.
 2. Lancer le programme pour espérer isoler les lignes de bugs.
 3. Corriger le bug en modifiant et relançant le programme autant de fois que nécessaire.
 4. Une fois le bug corrige, effacer les appels d'affichage.
- Une amélioration sera d'utiliser le débogueur (p. ex. gdb en C).
 1. Exécuter le programme sous débogueur.
 2. Ajouter des points d'arrêt pour examiner les valeurs des variables.
 3. Corriger le bug une fois localisé.
- Mais même si la deuxième méthode est plus efficace que la première, elles sont toutes les deux inefficaces :
 - Plus le code est développé, plus le temps de recherche de bug est important.
 - Il ne prémunit pas contre une régression future (n'empêche pas le bug de se reproduire).
 - L'introduction de lignes de code peut changer le comportement du programme (cf. Heisenbug).

7.2. Bonne pratique

1. Ecrire des tests pour détecter le bug tel qu'il est décrit.
 2. Corriger le code applicatif afin de passer les tests.
- Ainsi, les tests permettent aussi d'éviter que le bug ne se reproduise plus tard.

8. Que retenir de ce chapitre ?

- Les tests sont une obligation. Ils visent deux objectifs :
 - chasser les bugs,
 - mettre des garde-fous contre la régression. La régression de code est inhérente à l'activité de développement selon les méthodes agiles puisqu'elle inclut de la refonte de code.
- Les tests doivent forcément accompagner le code d'un logiciel. Un code sans test est inutile. En empruntant la métaphore du génie civil, les tests sont les armatures du béton. On sait en génie civil, qu'il est impossible de faire une tour de 200 m sans béton armé.
- Il y a trois types de test selon le niveau considéré.
- Unitaire : test en isolation (utilisation de bouchons si nécessaire pour isoler). Ce sont le plus nombreux.
- Intégration.
- Système.
- Les tests doivent être écrits au même moment que l'écriture de la fonctionnalité, que ce soit juste avant ou juste après
- Le code des tests étant du code, il doit donc être propre au même titre que le code source applicatif.