Chapitre 1 : Introduction au génie logiciel

« Si les ouvriers construisaient les bâtiments comme les développeurs écrivent leurs programmes, le premier pic-vert venu aurait détruit toute civilisation. » ***Gerald Weinberg***

1. Pourquoi un cours sur le génie logiciel ?

La spécialité informatique de l'ENSICAEN forme des **développeurs logiciel**, aussi nommés **architectes du logiciel**. Ce sont des ingénieurs professionnels qui conçoivent et mettent à jour les logiciels informatiques.
Pourquoi faire un cours sur le développement de logiciel alors que vous êtes déjà formé à programmer. Pourquoi un cours sur la programmation ne suffit-il pas pour développer un logiciel ?
Nous introduisons ici une différence fondamentale entre programme et logiciel, ainsi que développeur et programmeur :

- 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 code est recopié directement d'Internet. 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 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 d'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 ci-après remplit parfaitement cette tâche :

```
void main() {
  int nb1, nb2;
```

```
  scanf("%d", &nb1);
  scanf("%d", &nb2);
  printf("Résultat = %d\n", nb1 + nb2);
}
```

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 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 fonction du système d'exploitation (OS) sur lequel le programme s'exécutera. À 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 précisément 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 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 implanté sur un OS unique.
- Interfaçant des périphériques qui sont physiquement à disposition au moment de la programmation.

Logiciel

- Portable sur différents OS.

- Réparti 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.

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 (MLOC : 10^6 LOC 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.
  - Noyau Linux 4.1 (2015) : 20MLOC (contre 15 MLOC linux 3.1 (2011)).
  - 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 coût de développement de l'algorithme de recherche de Google est estimé à 1000 années-hommes.

Conséquence de la taille : coût de développement

- Ordre de grandeur :
  - 1 année-homme ≈ 1 650 h.
  - 1 h ≈ 50 €.
  - Productivité (de l'analyse à la production) ≈ 2 à 5 LOC / h.
- Donc, le code suivant met 1 h à traverser tout le cycle de développement et coûte 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 commence après la livraison du logiciel au client.
    - La maintenance doit être envisagée dès sa construction.

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.

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

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 :

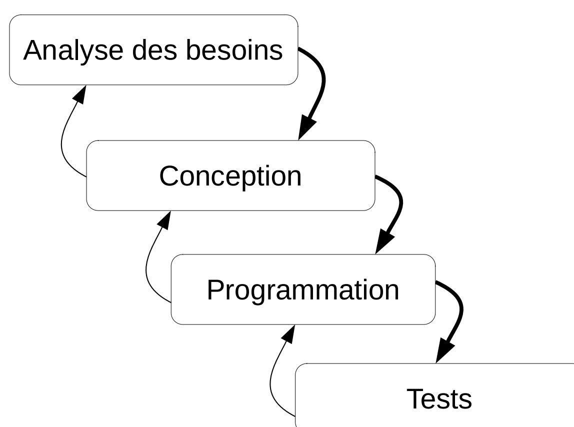
- Découper le temps alloué au projet en une séquence d'étapes parfaitement identifiées de manière à introduire du contrôle intermédiaire.
  - Livrable : *Diagramme de Gantt / Pert.*
- Ne faire qu'une seule chose à la fois à chaque étape.
  - *Métiers spécifiques à chaque étape.*
- Récolter tout le besoin 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.*

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 aide à identifier les briques de base de la conception et les mécanismes pour les assembler.
3. **Un formalisme : Il f**ournit 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.

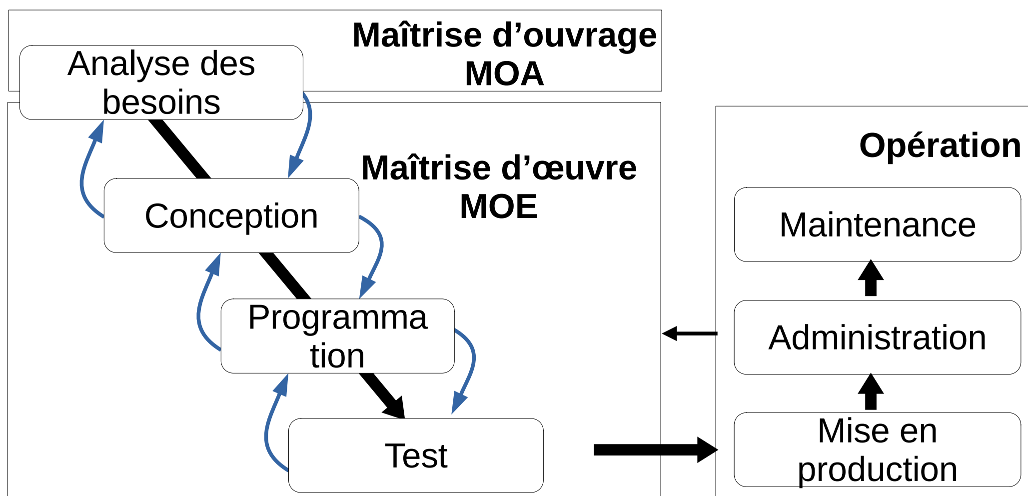
2.3. Une méthode : la méthode en cascade

Appliquée au génie logiciel, la méthode du génie civil aboutit au cycle en 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

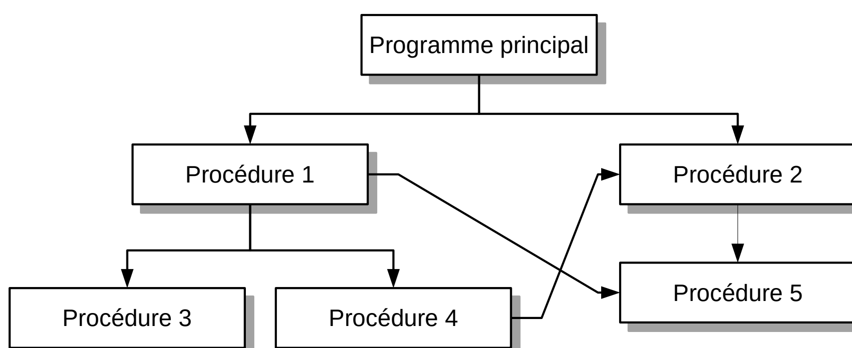
On appelle ouvrage le produit qui résulte d'un projet, ici le logiciel. La maîtrise d'ouvrage (MOA) est à l'origine de l'idée de base du projet et représente à ce titre les utilisateurs finaux à qui l'ouvrage est destiné. Ils sont aussi nommés aussi donneur d'ordre ou client. La maîtrise d'œuvre (**MOE**) 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'enchaînent.

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



Un programme est une séquence d'appel de procédures.

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 organisée 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(A)		*cost*	*times*
1	**for** $j = 2$ **to** $A.length$	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 côté 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)**
   - Écriture du code.
   - Écriture 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.
Conséquence :

- Si l'une des étapes en amont est imparfaite, les étapes en aval seront imparfaites et conduiront à mettre le projet en difficulté. Planifier à long terme, c'est comme minuter son trajet entre Caen et Paris. La moindre déviation pour travaux peut mettre en péril toute la suite du parcours parce que ce décalage peut nous conduire finalement dans les embouteillages quotidiens sur les rocades des villes traversées que l'on avait pourtant prévues d'éviter. La prise en compte de ce décalage devrait être de reconsidérer le plan initial en changeant par exemple l'itinéraire et éviter les villes aux heures d'embouteillage.

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

Il est extrêmement difficile d'estimer la durée/coût d'un projet informatique. Il y a trop d'aléas et d'incertitudes. Le travail en équipe complique encore cette prévision.

- Le mythe de l'année-homme (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.*

- Toutes les tâches ne sont pas divisibles : « Neuf femmes ne font pas un enfant en un mois » (F. Brooks).
- 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. Étant 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 déclarés dans la station spatiale internationale pendant son exploitation.

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 procédure 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 l'analogie avec le 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.
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 de l'Agilité.
La définition résultante 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 50 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).

4.3. Remarques

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

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 mature dans les équipes.

	Rappel en 2000	2015
Projets échoués	23 %	9 %
Projets en difficultés	49 %	52 %
Projets réussis	**28 %**	**39 %**

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 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, l'ingénieur 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 :

1. Le paradigme objet.
2. Le langage UML avec ses principaux diagrammes.
3. Les méthodes de développement itérative et incrémentale en particulier l'agilité.
4. La qualité de code.
5. Les tests logiciel.

5.1. Pourquoi ce cours ?

À 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.

À 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. » Robert Martin.

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) :
  - Être 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.