# 1 Introduction à l'Ingénierie Logicielle

## 1.1 Définition et Nécessité de l'Ingénierie Logicielle

L'ingénierie logicielle est essentielle pour construire des systèmes logiciels fiables, robustes et efficaces. Nous voulons avoir des raisons de faire confiance aux logiciels.

## 1.2 Constat Initial : Analogie Rolls-Royce et Ordinateur

Imaginez si l'industrie automobile avait progressé au même rythme que l'informatique. Comme le soulignait Robert X. Cringely :

> Si l'automobile avait suivi le même cycle de développement que l'ordinateur, une Rolls-Royce coûterait aujourd'hui :
>
> - $100,
> - Consommerait un million de miles par gallon,
> - Et exploserait une fois par an, tuant tous les occupants.

Cette analogie frappante met en lumière l'importance cruciale de l'ingénierie dans le développement de systèmes complexes et critiques.

## 1.3 Pourquoi l'Ingénierie Logicielle est Cruciale

L'ingénierie logicielle est indispensable pour plusieurs raisons, notamment :

- **Problèmes Technologiques :**
  - Les logiciels sont de plus en plus massifs et complexes.
  - Les contraintes de fiabilité et de sécurité sont primordiales.
  - Il existe des interactions complexes entre matériel et logiciel.
- **Problèmes Sociaux :**
  - Les demandes des clients sont souvent imprécises et fluctuantes, nécessitant une communication efficace.
  - Le développement se fait en grandes équipes, ce qui requiert une coordination rigoureuse.
  - Le cycle de vie des logiciels est long et inclut une maintenance conséquente.
  - Les contraintes légales et les problèmes d'image sont des enjeux importants.

## 1.4 Risques Liés aux Logiciels Défectueux

Un logiciel défectueux peut être dangereux et avoir des conséquences graves :

- Pour certains logiciels, des vies sont en jeu, notamment :
  - Dans les transports,
  - La médecine,
  - L'industrie,
  - Le nucléaire,
  - Les missiles...
- De nombreux autres logiciels ont un fort pouvoir de nuisance :
  - Dans les communications,

- Dans les transactions bancaires...

L'évolution des logiciels tend vers des systèmes :

- Avec des composants plus nombreux.
- Présentant un risque d'accumulation de "petits" problèmes.

## 1.5 Fiascos Mémorables et Coûts Associés

De nombreux fiascos mémorables illustrent les conséquences de défauts logiciels :

- **1962 Mariner 1 (Venus) :** Mauvais calcul d'une trajectoire, crash en vol.
- **1985 Therac-25 :** Radiothérapie surdosée. 5 morts.
- **1996 Ariane 5 :** Crash.
- **1997 .com :** Blocage de tous les noms de domaine.
- **1999 Mars Climate Orbiter :** Un satellite à 120 millions $ perdu pour une confusion entre unités.
- **2004 SNCF :** Système de réservation défaillant.
- **2004 Réseaux Bouygues et France Telecom :** Inopérants.
- **2005 Régulateur de vitesse Renault Laguna.**
- **2010 NPfIT :** Coût de 120 milliards £.

De plus, les logiciels peuvent atteindre une taille considérable et engendrer des coûts importants :

- **Windows 7 :**
  - 1200 personnes (rien que pour les programmeurs).
  - Coûts de développement : estimés à 5 milliards.
  - Revenus : estimés à 20 milliards.
  - Engagements légaux sur 15 à 20 ans.
  - Un précédent litige sur Windows Server 98 avait coûté 700 millions.

## 1.6 Procédures de Validation : Questions Clés

Pour garantir la qualité et la fiabilité des logiciels, des procédures de validation rigoureuses sont nécessaires. Les questions clés à se poser incluent :

- Comment justifier et contrôler le processus de développement ?
- Comment garantir la représentativité des tests ?
- Traiter toutes les combinaisons d'options ?
- Comment tester tous les facteurs extérieurs ?
- Qui peut autoriser la mise en service ? Quelle est sa responsabilité ?

### 1.7 Ordres de Grandeur : Taille et Coût des Logiciels

La taille et le coût des systèmes logiciels peuvent être considérables :

- **Taille des systèmes logiciels :**
  - Système d'exploitation : plusieurs millions (ou dizaines de millions) de lignes de code.
  - Navette spatiale : plusieurs dizaines (ou centaines) de millions de lignes de code.
    - **Coûts de développement :**
      - Codage : $\sim$ 15-20 %
      - Validation et vérification : $\sim$ 40%
      - Spécification et conception : $\sim$ 40 %

## 2 Processus de Développement Logiciel

### 2.1 Processus de Développement en V

Le processus de développement en V est un modèle séquentiel qui met en évidence la relation entre les phases de développement et les phases de test.

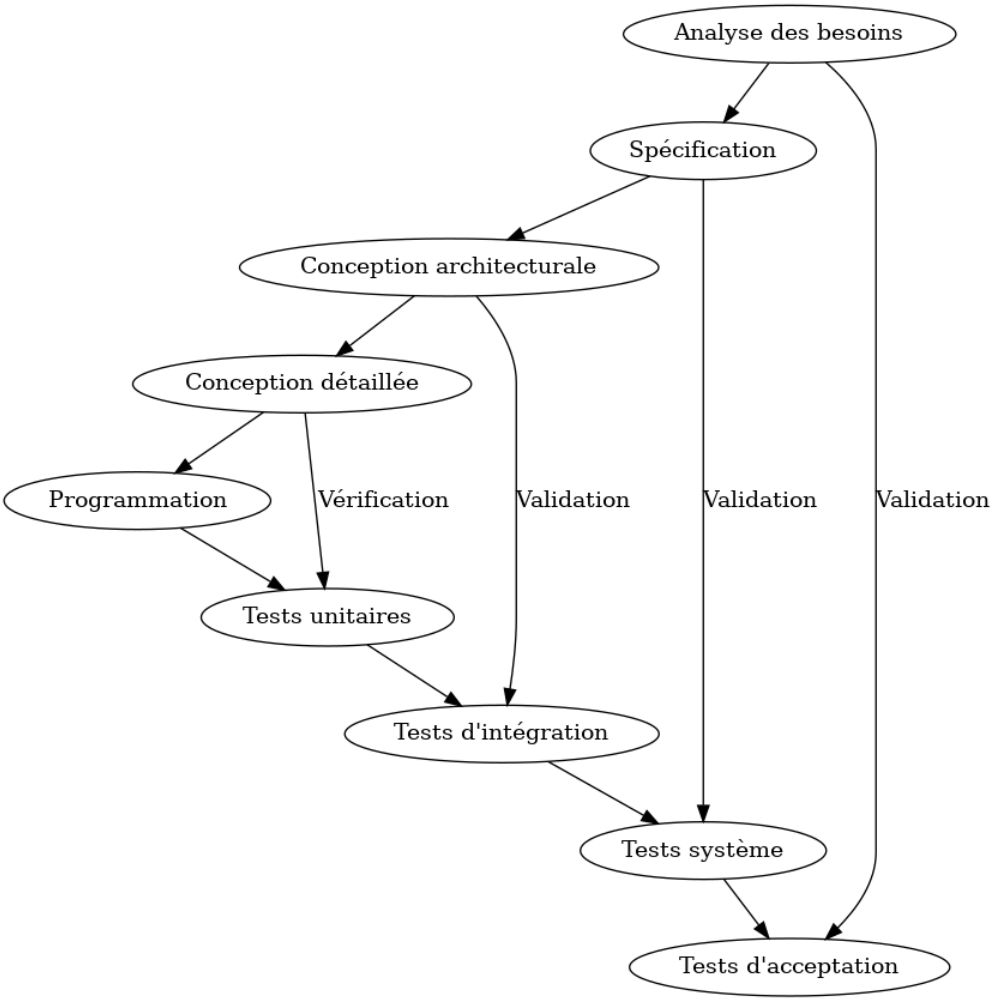


Figure 1: Processus de Développement en V

Le processus se déroule comme suit :

1. **Analyse des besoins :** Définir ce que doit faire le système.
2. **Spécification :** Écrire les spécifications détaillées.
3. **Conception architecturale :** Définir comment le système va être réalisé.
4. **Conception détaillée :** Conception détaillée des composants.
5. **Programmation :** Implémentation du code.
6. **Tests unitaires :** Tests des unités de code.
7. **Tests d'intégration :** Tests de l'intégration des composants.
8. **Tests système :** Tests du système complet.
9. **Tests d'acceptation :** Tests par le client pour acceptation.
10. **Validation et vérification :** S'assurer que le système répond aux besoins et est correctement construit.

## 2.2 Phases de Développement et Documents Associés

Chaque phase du développement logiciel est associée à des documents spécifiques :

- **Phase d'analyse des besoins :**
  - Étude du contexte, contraintes de performance, ergonomie, portabilité.
  - Produit : Cahier des charges (requirement specification).
- **Phase d'analyse des spécifications :**
  - Définir ce que le système doit faire (et pas comment le faire).
  - Produit : Modèle d'analyse (analysis model).
- **Phase de conception architecturale :**
  - Décomposition en "composants".
  - Conception de l'intégration et des tests d'intégration.
  - Produit : Conception architecturale (architectural design).
- **Phase de conception détaillée :**
  - Choix des algorithmes et structures de données, définition des interfaces, conception des tests.
  - Produits : Documents de conception, interfaces de code, fragments de code, pseudo-code, prototypes.
- **Phase de codage :**
  - Implémentation du code.
  - Produit : Code.
- **Phase de test unitaire :**
  - Test indépendant de chaque méthode (comportement bas-niveau).
  - Produit : Protocoles de validation, documentation.
- **Phase de test d'intégration :**
  - Exécution des scénarios de tests conçus pendant l'analyse (comportement haut-niveau).
  - Produit : Protocoles de validation, documentation.
- **Phase de test système :**
  - Test global en conditions réelles, mesure de performances.
  - Produit : Protocoles de validation, documentation.
- **Phase de déploiement :**
  - Déploiement chez le client.
  - Produit : Test, inspection des documents et des normes qualité.

## 2.3 Activités Transverses

Chaque phase de développement contient des activités transverses :

- **Production de documentation :** Manuels d'utilisateur, de référence, d'installation...
- **Validation / Vérification :** Chaque phase produit et valide le document correspondant.

## 2.4 Difficultés Particulières du Développement Logiciel

Le développement logiciel présente des difficultés spécifiques :

- **Problèmes dépendants des applications :**
  - Synchronisation des processus.
  - Nature et volume des données, problèmes algorithmiques.
  - Interactions matériel/logiciel.
  - Contraintes de temps-réel.
- **Facilité de modification et imprévisibilité des conséquences :**
  - Il est facile de modifier un programme, mais beaucoup moins de prédire les conséquences.
  - Un changement mineur peut être catastrophique.
  - Il est difficile de connaître les probabilités des différentes situations.
  - Comment mesurer la qualité ?

## 2.5 Modèles de Développement Alternatifs : Modèle en Spirale

Le modèle en spirale est une alternative au modèle en V, mettant l'accent sur l'itération et la gestion des risques.

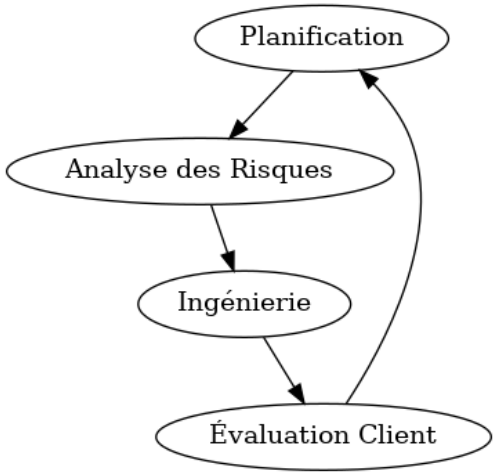


Figure 2: Modèle de Développement en Spirale

Caractéristiques du modèle en spirale :

- Intégration et validation progressive.
- Retours utilisateur rapides.
- Diminution des risques.
- Meilleure visibilité.
- Méthode  Agile .
- Modèle itératif.
- Cycle court de production pour mieux interagir avec le client.

## 2.6 Importance de la Maintenance

La maintenance logicielle est une phase essentielle, souvent complexe et coûteuse, après le déploiement. Elle comprend plusieurs aspects :

- **Correction** des bugs critiques.
- **Adaptation** à de nouveaux OS, matériels, problèmes de performance.
- **Évolution** : intégration de nouvelles fonctionnalités.
- Pire : éventuellement maintenir plusieurs versions en parallèle !

Les coûts de maintenance peuvent être 2 à 4 fois supérieurs au développement initial. Un besoin de tests de régression sur les nouvelles versions est crucial, avec des descriptions précises des tests (entrées, sorties, contexte) et un besoin d'automatisation.

# 3 Modélisation UML : Cas d'Utilisation et Diagrammes de Séquence

## 3.1 Cas d'Utilisation

Les cas d'utilisation sont utilisés pour comprendre les besoins du client et rédiger le cahier des charges fonctionnel.

### 3.1.1 Objectif et Éléments de Description

**Objectif :** Comprendre les besoins du client pour rédiger le cahier des charges fonctionnel.

**Trois questions clés pour définir un cas d'utilisation :**

1. Définir les **utilisations principales** du système : à quoi sert-il ?
2. Définir l'**environnement** du système : qui va l'utiliser ou interagir avec lui ?
3. Définir les **limites** du système : où s'arrête sa responsabilité ?

**Éléments de description d'un cas d'utilisation :**

- Diagramme de cas d'utilisation.
- Description textuelle des cas d'utilisation.
- Diagrammes de séquence des scénarios d'utilisation.

### 3.1.2 Exemple de Cas d'Utilisation : Commander

Considérons un système de vente en ligne. Un cas d'utilisation typique est "Commander".

**Scénario : Commander**

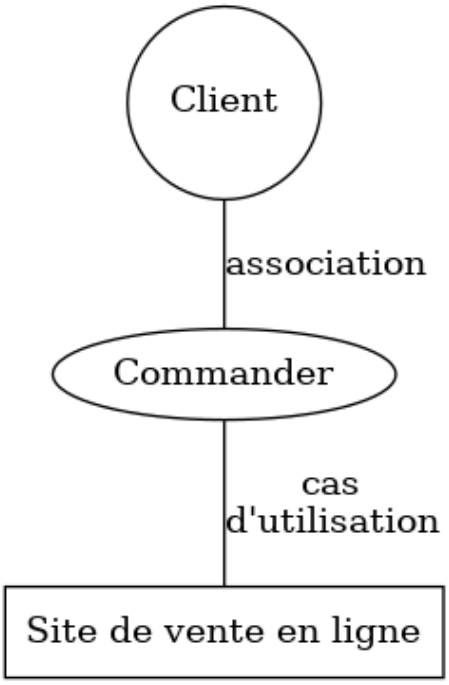


Figure 3: Diagramme de Cas d'Utilisation : Commander

**Description textuelle du scénario "Commander" :** Le client s'authentifie dans le système, puis choisit une adresse et un mode de livraison. Le système indique le montant total de sa commande au client. Le client donne ses informations de paiement. La transaction est effectuée et le système en informe le client par e-mail.

#### 3.1.3 Acteurs et Scénarios

**Acteur :** Entité qui interagit avec le système (personne, chose, logiciel extérieur au système). Il représente un rôle (plusieurs rôles possibles pour une même entité) et est identifié par un nom de rôle.

**Cas d'utilisation :** Fonctionnalité visible de l'extérieur, action déclenchée par un acteur, identifiée par une action (verbe à l'infinitif).

**Scénario principal (cas nominal) :** Description des étapes normales du cas d'utilisation. **Scénarios alternatifs (cas d'erreur, variations) :** Déviations du scénario principal.

### 3.2 Relations entre Cas d'Utilisation

Il existe différents types de relations entre les cas d'utilisation :

#### 3.2.1 Inclusion (≪includes≫)

Un cas d'utilisation en inclut un autre. Le scénario du cas d'utilisation inclus est inséré dans le scénario du cas d'utilisation incluant. Utilisé lorsqu'il y a un scénario commun à plusieurs cas d'utilisation.

#### 3.2.2 Extension (≪extends≫)

Un cas d'utilisation étend un autre cas d'utilisation. Le cas d'utilisation extension est déclenché conditionnellement durant l'exécution du cas d'utilisation étendu. Le cas d'utilisation extension est optionnel pour le cas d'utilisation étendu.

### 3.2.3 Généralisation

Un cas d'utilisation est une spécialisation d'un cas d'utilisation plus général. Tout ou partie du scénario du cas d'utilisation général est spécifique au cas d'utilisation particulier.

## 3.3 Diagrammes de Séquence

Les diagrammes de séquence représentent graphiquement la chronologie des échanges de messages entre les acteurs et le système.

### 3.3.1 Diagramme de Séquence d'Analyse

**Objectif :** Représenter les interactions entre l'utilisateur et le système du point de vue de l'utilisateur, au niveau de l'analyse des besoins.

**Éléments :**

- Acteurs (représentés par des icônes).
- Système (représenté par une colonne).
- Messages informels (noms liés aux cas d'utilisation).
- Mise en avant des données utiles au scénario (arguments).

### 3.3.2 Diagramme de Séquence de Conception

**Objectif :** Décrire la réalisation des cas d'utilisation sur le système représenté par le diagramme de classes, au niveau de la conception.

**Éléments :**

- Acteurs.
- Objets (instances des classes).
- Messages (appels d'opérations).

**Principes de base :**

- Vie de chaque entité représentée verticalement.
- Échanges de messages représentés horizontalement.

## 3.4 Types de Messages

### 3.4.1 Message Synchrone

L'émetteur est bloqué en attente du retour du message. Représenté par une flèche pleine.

### 3.4.2 Message Asynchrone

L'émetteur n'est pas bloqué et continue son exécution. Représenté par une flèche ouverte.

### 3.4.3 Message Réflexif

Un objet s'envoie un message à lui-même.

## 3.5 Alternatives et Boucles

### 3.5.1 Alternative

Représentation de conditions à l'envoi d'un message (si... alors... sinon...). Noté avec un bloc `alt`.

### 3.5.2 Boucle

Représentation de la répétition d'un enchaînement de messages. Noté avec un bloc `loop`.

## 3.6 Référence à un Autre Diagramme

Possibilité de faire référence à un autre diagramme pour décomposer la complexité. Noté avec `ref`.

# 4 Modélisation UML : Diagrammes de Classes

## 4.1 Classes, Objets, Attributs et Opérations

### 4.1.1 Classes

Une classe est un regroupement d'objets de même nature (mêmes attributs et mêmes opérations). C'est un modèle ou un plan pour créer des objets.

### 4.1.2 Objets

Un objet est une instance d'une classe. Il représente une entité concrète ou abstraite du domaine d'application. Décrit par :

- Identité (adresse mémoire).
- État (attributs) : caractéristiques de l'objet, associées à une valeur.
- Comportement (opérations) : services que l'objet peut offrir.

### 4.1.3 Attributs

Un attribut est une caractéristique partagée par tous les objets de la classe. Il est associé à chaque objet une valeur et possède un type simple (int, bool, string, date, etc.).

### 4.1.4 Opérations

Une opération est un service qui peut être demandé à tout objet de la classe. Elle représente un comportement commun à tous les objets de la classe. Il ne faut pas confondre opération et méthode (implantation de l'opération).

## 4.2 Associations entre Classes

Une association représente une relation binaire entre classes.

#### 4.2.1 Multiplicités

Les multiplicités contraignent le nombre d'objets liés par une association.

- **1 :** Exactement un.
- **0..1 :** Au plus un (zéro ou un).
- **1..* :** Au moins un (un ou plusieurs).
- **0..* ou *** : Zéro ou plusieurs.
- **n..m :** Entre n et m.

#### 4.2.2 Rôles

Les rôles nomment les extrémités d'une association et permettent d'accéder aux objets liés par l'association à partir d'un objet donné.

#### 4.2.3 Agrégation

L'agrégation est une association particulière entre classes, de type composant-composite, où une classe prédomine sur l'autre. C'est une agrégation faible : le composite fait référence à ses composants et la durée de vie du composite et des composants sont indépendantes.

#### 4.2.4 Composition

La composition est une forme d'agrégation forte. Le composite *contient* ses composants. La création ou destruction du composite entraîne la création ou destruction de ses composants. Un objet ne fait partie que d'un seul composite à la fois.

### 4.3 Héritage, Interface et Polymorphisme

#### 4.3.1 Héritage

L'héritage permet de construire une classe à partir d'une classe plus générale (super-classe) en partageant ses attributs, opérations et contraintes. Une sous-classe spécialise ou raffine une super-classe.

#### 4.3.2 Interface

Une interface est une liste d'opérations constituant un contrat à respecter par les classes réalisant l'interface. Ce n'est pas une classe, et elle ne peut pas servir à créer des objets. Toutes les opérations d'une interface sont abstraites.

#### 4.3.3 Polymorphisme

Le polymorphisme est la capacité pour une opération d'être définie différemment dans différentes sous-classes, tout en restant une opération spécifique à la sous-classe. Il nécessite la définition d'une opération abstraite dans la super-classe.

### 4.4 Opérations Abstraites

Une opération abstraite est une opération non définie pour une classe, car il est impossible de la définir pour tous les objets de la classe. L'opération abstraite est définie en italique dans les diagrammes. Une opération abstraite implique que la classe contenant l'opération abstraite est une classe abstraite.

# 5 Contraintes et Invariants

## 5.1 Représentation des Contraintes et Invariants

Le diagramme de classes représente la structure du système, mais ne permet pas de représenter directement les contraintes et invariants. Les contraintes et invariants sont des propriétés portant sur les éléments du modèle, qui doivent être vérifiées à tout instant.

## 5.2 Contraintes sur les Attributs

Les contraintes sur les attributs spécifient des restrictions sur les valeurs autorisées pour les attributs. Elles peuvent être représentées sous forme de notes dans le diagramme, de texte accompagnant le diagramme, ou en utilisant l'OCL (Object Constraint Language).

## 5.3 Contraintes sur les Associations

Les contraintes sur les associations spécifient des restrictions sur les relations entre les classes. Elles peuvent également être représentées de différentes manières, y compris avec l'OCL.

## 5.4 Object Constraint Language (OCL)

L'OCL est un langage de contraintes formel associé à UML, permettant d'exprimer de manière précise les contraintes et invariants liés au diagramme de classes.

# 6 Conception et Programmation Structurée

## 6.1 Principes de la Programmation Structurée

La programmation structurée est un paradigme de programmation visant à améliorer la clarté, la qualité et le temps de développement d'un programme en utilisant des structures de contrôle de flux spécifiques et en évitant l'utilisation excessive d'instructions de branchement inconditionnel (goto).

## 6.2 Structures de Contrôle Restreintes

Pour raisonner sur un programme, il est essentiel d'utiliser des structures de contrôle restreintes :

- **Abstraction** : Appels de fonctions.
- **Induction** : Fonctions récursives, itération.
- **Raisonnement par cas** : Types algébriques, Pattern matching.

Éviter l'utilisation de `goto` pour maintenir un flux de contrôle clair et prévisible.

## 6.3 Organisation en Couches

Organiser le logiciel en couches permet de mieux gérer la complexité. Chaque couche de niveau n accède uniquement aux primitives de la couche n-1, juste en dessous. Cela favorise l'abstraction et l'indépendance entre les différentes parties du système.

### 6.4 Raffinement Successif

La conception par raffinements successifs consiste à décomposer un problème complexe en sous-problèmes plus simples, et à raffiner progressivement les solutions à chaque niveau. Chaque raffinement correspond à un choix de conception.

### 6.5 Types Abstraits de Données

Utiliser des types abstraits de données (TAD) permet de manipuler des données sans se soucier de leur représentation concrète. Un TAD est défini par un ensemble d'opérations. L'utilisateur d'un TAD n'a pas besoin de connaître l'implémentation interne.

### 6.6 Patrons de Conception Architecturaux

Les patrons de conception architecturaux sont des solutions éprouvées à des problèmes de conception récurrents. Ils permettent de structurer l'architecture d'un logiciel de manière modulaire et flexible. Exemples : Cacher la représentation des données (c.f. ADT), les fonctions internes d'un composant. Les interfaces doivent être restreintes et bien définies.

## 7 Documentation

### 7.1 Importance des Commentaires et de la Documentation

Ajouter des commentaires au code source est essentiel pour faciliter la compréhension et la maintenance du code. Les commentaires aident :

- À comprendre le code source.
- À reprendre le code plus tard.
- À expliquer l'utilisation des classes et méthodes.
- À documenter les améliorations, problèmes à corriger, etc.

Utiliser des outils de documentation comme Javadoc pour générer automatiquement de la documentation à partir des commentaires.

### 7.2 Annotations

Les annotations (commençant par `@`) permettent d'ajouter des métadonnées au code source. Exemple : `@Override` pour forcer la redéfinition d'une méthode. Javadoc utilise des annotations comme `@param` et `@return` pour spécifier les paramètres et la valeur de retour des méthodes.