Table of contents

420-4P1-DM Analyse et conception d'applications 3
Cycle de vie d'un logiciel 4
Analyse des besoins ou la Collecte des exigences 5
Conception 7
Implémentation 9
Test 11
Déploiement 13
Maintenance 15
Procédés de développement 17
Développement en cascade 19
Développement itératif et incrémental 21
Développement agile 23
Développement lean 25
Développement piloté par les tests (TDD) 27
Développement piloté par le comportement (BDD) 29
Développement par intégration continue (CI) 31
Développement par déploiement continu (CD) 33
Analyse des besoins ou la Collecte des exigences 35
Parties prenantes 37
Exploration 38
Backlog 39
User stories 40
Écrire des user stories 42
Liens avec le BDD 44
Exemples 46
Exigences non-fonctionnelles 50
Document vision 51
Exemples de contenu d'un document vision 54
Diagramme de contexte 60
Techniques d'entrevue 61
Analyse et conception architecturale 62
Diagramme de cas d'utilisation 64
Diagrammes de composants 69
Diagrammes de déploiement 70

Différences entre diagrammes de composants et de déploiement 72

420-4P1-DM Analyse et conception d'applications

Pondération du cours: 2-3-3

Description des activités d’enseignement et d’apprentissage :

Ce cours comportera une partie théorique portant sur le cycle de vie du logiciel, les procédés de développement logiciel, plusieurs types de diagrammes UML, la modélisation des données, la modélisation des traitements, les essais et les tests.

Par la suite, les étudiants auront un projet basé sur un cas réel et adapté pour ce cours. Les étudiants devront passer à travers les étapes principales du procédé de développement logiciel : l’exploration, l’analyse, la conception globale, la conception détaillée, les tests et les essais. Cela leur permettra de mettre en pratique la théorie et d’avoir déjà une petite expérience de développement complet d’un logiciel pour le cours 420-5W1-DM.

Cycle de vie d'un logiciel

Le cycle de vie d'un logiciel désigne les différentes étapes du développement d'un logiciel, de sa conception à sa maintenance, en passant par sa mise en œuvre. Voici un résumé des étapes typiques :

1. **Analyse des besoins ou Collecte des exigences :** L'objectif de cette phase est de comprendre et documenter les besoins du client ou de l'utilisateur. Cela implique généralement une interaction directe avec les utilisateurs finaux ou les parties prenantes pour comprendre leurs besoins et leurs exigences.
2. **Conception :** Dans cette phase, les développeurs ou ingénieurs logiciels définissent la façon dont le logiciel va résoudre les problèmes ou répondre aux exigences recueillies lors de la phase d'analyse des besoins. C'est là que la structure et l'architecture du logiciel sont décidées.
3. **Codage :** Lors de cette phase, le logiciel lui-même est créé. Les ingénieurs logiciels écrivent le code source du logiciel, en utilisant des langages de programmation sélectionnés en fonction des spécifications de la phase de conception.
4. **Test :** Le logiciel est testé pour s'assurer qu'il n'a pas de bugs ou d'erreurs, qu'il répond aux exigences spécifiées, et qu'il fournit les fonctionnalités attendues. Ceci comprend également des tests non fonctionnels tels que les performances, la sécurité, etc.
5. **Déploiement :** Une fois que le logiciel est testé et vérifié, il est déployé pour l'utilisateur final. Cela peut impliquer la distribution du logiciel via différents canaux, tels que le téléchargement en ligne, les clés USB ou cartes de mémoire, (anciennement) les CD/DVD, etc.
6. **Maintenance :** À ce stade, le logiciel est surveillé pour résoudre les problèmes qui pourraient survenir une fois qu'il est utilisé en production. Cela inclut des mises à jour, des correctifs de sécurité, des améliorations et des ajustements basés sur les retours des utilisateurs.

Il est important de noter que le cycle de vie du logiciel peut varier d'un projet à l'autre et peut être plus complexe que ce qui est décrit ici, comprenant des iterations et des retours en arrière sur certaines phases dans certains modèles de cycle de vie, comme le modèle Agile.

Analyse des besoins ou la Collecte des exigences

L'analyse des besoins ou la collecte des exigences est la première et cruciale étape d'un cycle de vie du développement logiciel. C'est le processus par lequel on **détermine les attentes des utilisateurs** pour un nouveau logiciel ou une version améliorée d'un logiciel existant.

Ce processus implique une communication approfondie avec les **parties prenantes**, qui pourrait inclure :

1. les **clients**,
2. les **utilisateurs finaux**,
3. les **développeurs de logiciels**,
4. et les **autres parties intéressées**.

L'objectif est de comprendre le problème que le logiciel se propose de résoudre, d' **établir ce qu'on attend** du logiciel en termes

1. de **fonctionnalités**,
2. de **performance**
3. et d' **utilisabilité**,

et d' **évaluer** comment le nouveau logiciel, ou le logiciel amélioré, s'intégrera dans les systèmes et les processus existants.

L'analyse des besoins est généralement réalisée à travers une série

1. de **rencontres**,
2. d' **interviews**,
3. d' **enquêtes**,
4. d' **observation du processus de travail**,
5. et d' **ateliers avec les parties prenantes**.

Pendant ces activités, la liste des exigences fonctionnelles et non-fonctionnelles est créée et affinée.

Les **exigences fonctionnelles** définissent *ce que le logiciel doit faire*. Par exemple, une exigence fonctionnelle pour une application de commerce électronique pourrait être *"l'application doit permettre aux utilisateurs de filtrer les produits par catégorie"*.

Les **exigences non-fonctionnelles**, d'autre part, décrivent *comment le logiciel doit le faire*. Les exigences non-fonctionnelles pourraient inclure des questions de performance, de sécurité, d'accessibilité, de compatibilité, etc. Par exemple, une exigence non-fonctionnelle pour l'application de commerce électronique pourrait être *"l'application doit charger une page de produit en moins de 2 secondes"*.

Après la collecte, les exigences doivent être analysées afin de garantir leur clarté, leur cohérence, leur complétude, et leur faisabilité. Il est essentiel que les analystes de système travaillent en étroite collaboration avec toutes les parties prenantes pour s'assurer que les exigences sont bien comprises et correctement traduites pour le développement.

Enfin, les **exigences doivent être documentées de façon détaillée** afin qu'elles puissent servir de référence pour les développeurs de logiciels, les testeurs de logiciels, les gestionnaires de projet, et aussi les utilisateurs. La documentation doit aller au-delà de la simple énumération des exigences - elle doit également inclure des diagrammes de processus, des maquettes, des scénarios d'utilisation, et tout autre artefact qui peut aider à comprendre, à visualiser et à expliciter les exigences.

En conclusion, l'analyse des besoins et la collecte des exigences est une étape essentielle du développement de logiciels dont le but est de comprendre ce que les utilisateurs attendent du logiciel. Elle forme la base sur laquelle le reste du processus de développement est construit. Il est donc essentiel de prendre le temps nécessaire pour réaliser cette analyse correctement afin de minimiser les risques d'échec du projet.

Conception

La phase de conception d'un logiciel est la deuxième étape clé du cycle de développement logiciel qui suit l'analyse des besoins ou la collecte des exigences. Pendant cette phase, l'équipe de développement planifie la solution à implémenter, répondant ainsi aux exigences du système définies lors de la phase d'analyse.

La conception d'un logiciel est principalement divisée en deux niveaux : la conception de haut niveau (également appelée conception architecturale) et la conception de bas niveau (également appelée conception détaillée).

1. **Conception de haut niveau (Conception architecturale):** La conception de haut niveau est une représentation abstraite du système qui identifie les différents composants et décrit comment ils interagissent entre eux. Le but est de définir une architecture logicielle qui sera à la fois efficace et efficiente, stable, flexible, et qui respecte les exigences fonctionnelles et non fonctionnelles du système.

   L'architecture d'un logiciel est généralement représentée par des modèles, des schémas et des diagrammes qui dépeignent visuellement la disposition générale du système. Les architectures logicielles courantes comprennent les architectures monolithiques, les architectures en couches, les architectures orientées services (SOA), et les microservices.

2. **Conception de bas niveau (Conception détaillée):** La conception de bas niveau est une description détaillée des fonctionnalités individuelles (ou des composants) identifiées lors de la phase de conception architecturale. Cette phase consiste à décider comment le code sera écrit et comment les fonctionnalités seront effectivement mises en œuvre. De plus, cette phase de conception se concentre également sur la détermination de l'algorithme approprié pour atteindre les fonctions spécifiées dans le document de conception de haut niveau.

   Les détails spécifiques dans cette phase incluent la conception de l'interface utilisateur, la conception de la base de données, la conception des classes et autres structures de données, déterminer les méthodes, attributs et relations appropriées pour chaque classe.

   Les concepts tels que la modélisation des données, les diagrammes de classe, les diagrammes de séquence, les diagrammes d'activités font partie intégrante de cette étape. Cette étape sert également à identifier et à résoudre les problèmes potentiels qui pourraient survenir lors de la mise en œuvre des fonctionnalités.

En somme, la phase de conception du logiciel résulte en deux principaux livrables :

1. le **document de conception du système**

2. et le **document de conception détaillé**.

Ces documents servent de guides pour l'équipe de développement lors de la phase de codage. Ils offrent également une documentation précieuse pour la maintenance et l'extension futures du logiciel.

Il est important de noter, toutefois, que **la phase de conception n'est jamais vraiment terminée**. Au fur et à mesure que le logiciel est développé et testé, les concepteurs peuvent découvrir de meilleurs moyens d'implémenter certaines fonctionnalités ou peuvent devoir apporter des modifications en raison de contraintes inattendues. Par conséquent, **tout plan de conception doit être flexible et adaptatif**.

Implémentation

L'étape de codage, ou phase d' *implémentation* (ou *implantation*), est l'une des étapes les plus cruciales et pratiques du cycle de vie du développement logiciel. C'est à ce stade que les spécifications détaillées rédigées et planifiées lors de la phase de conception sont traduites en un code source concret. En d'autres termes, c'est le moment où la conception du logiciel devient réelle.

La phase de codage commence généralement par la **mise en place d'un environnement de développement** approprié, comprenant l'ensemble des outils nécessaires pour écrire, tester et déboguer le code. Cela peut inclure un environnement de développement intégré (**IDE**), des **outils de tests**, des systèmes de **gestion de versions** (comme Git) et des **outils de déploiement**.

Les développeurs, à ce stade, commencent à écrire le code en suivant les spécifications issues de la phase de conception. Ils doivent **tenir compte des règles et des conventions de programmation**, qui ont pour but d'augmenter la lisibilité et la clarté du code, de prévenir les erreurs et de faciliter la maintenance du code à l'avenir.

Pendant cette phase, il est extrêmement important de pratiquer une programmation attentive et de qualité. Le code doit être simple, concis et efficace. Plus le code est simple, plus il sera facile de le déboguer et de le maintenir. Il est également recommandé de commenter le code de manière appropriée pour s'assurer que d'autres développeurs (ou le même développeur à l'avenir) comprennent ce que fait le code.

Pendant le processus de codage, les développeurs peuvent créer les différentes fonctionnalités du logiciel en petits incréments, en testant chaque incrément pour s'assurer qu'il fonctionne comme prévu. Cette approche, connue sous le nom d'intégration continue, peut aider à identifier rapidement et à résoudre les problèmes.

Des pratiques telles que la **revue de code** participent également à la qualité du code. Il s'agit d'un processus où d'autres développeurs vérifient le code pour détecter toute erreur ou faiblesse et examiner si les normes de codage et les exigences du projet ont été respectées.

À mesure que le projet progresse, le code est souvent stocké dans un système de gestion de versions, qui conserve un historique des modifications et permet aux développeurs de travailler sur différentes parties du projet sans se gêner.

L'objectif final de la phase de codage est de traduire les exigences et la conception du système en code, en créant ainsi le logiciel lui-même. Bien que cette phase soit l'une des plus

longues et des plus laborieuses du cycle de vie du développement logiciel, elle est toutefois l'une des plus satisfaisantes, car c'est à ce moment que les idées prennent vie.

Test

L'étape des tests est un élément **primordial** dans le cycle de développement de logiciel. C'est la phase où le code produit pendant l'étape de codage est examiné pour déceler les éventuelles erreurs, bugs, et pour vérifier si le logiciel se comporte comme attendu.

L'objectif principal des tests est d' **assurer la qualité du logiciel** en vérifiant que les fonctionnalités fournies **respectent les exigences identifiées pendant la phase de collecte des exigences**. Cela comprend également de s'assurer que le logiciel **fonctionne efficacement** et est capable de **gérer les exceptions et les erreurs**.

Voici les différentes formes de tests généralement employées dans le processus de test logiciel :

1. **Test unitaire :** Il s'agit de tester les plus petites unités de code indépendamment les unes des autres pour s'assurer qu'elles fonctionnent correctement. Typiquement, cela signifie tester chaque méthode ou fonction d'une classe ou d'un module.
2. **Test d'intégration :** Ce test vise à vérifier si les différentes unités de code fonctionnent bien ensemble. Par exemple, cela peut impliquer de tester l'interaction entre différentes classes, modules, ou services.
3. **Test de système :** Dans ce type de tests, le système entier est testé pour vérifier si toutes les composantes fonctionnent ensemble comme prévu. Cela inclut généralement des tests d'interfaces utilisateur (GUI), des tests de performance, des tests de sécurité, etc.
4. **Test d'acceptation :** C'est une forme de test qui vise à déterminer si le produit est prêt à être livré ou non. Il est habituellement effectué par l'utilisateur final dans un environnement qui simule le monde réel. Il vise à évaluer si le système répond aux exigences business du client.
5. **Test de régression :** Ce type de tests est réalisé pour s'assurer qu'une modification du code (ajout de nouvelles fonctionnalités, correction de bugs, etc.) n'a pas eu d'effets néfastes sur les fonctionnalités existantes du logiciel.

Il est important de noter qu'un bon processus de test ne se contente pas de trouver des bugs ; il contribue également à les prévenir. Cela signifie que l'activité de test **doit être intégrée à toutes les étapes du cycle de vie** du développement logiciel, et **pas seulement à la fin**.

Enfin, une phase de test réussie aboutira à un logiciel sans bugs (ou du moins avec le moins de bugs possible), qui répond aux attentes des utilisateurs et aux exigences du projet. C'est

une étape indispensable pour assurer la qualité d'un logiciel et pour garantir sa robustesse, son efficacité et sa fiabilité.

Déploiement

Le déploiement d'un logiciel est l'étape finale du cycle de vie du développement du logiciel. Cette phase consiste à rendre le logiciel accessible aux utilisateurs finaux. En d'autres termes, lors du déploiement, le logiciel est transféré du périphérique de développement vers l'environnement de production où les utilisateurs peuvent l'utiliser en conditions réelles.

Le processus de déploiement d'un logiciel comprend plusieurs sous-étapes en fonction de la nature du logiciel et des besoins spécifiques de l'entreprise :

1. **Publication :** Il s'agit de l'étape initiale du déploiement où le logiciel est préparé pour l'utilisation. Cela implique la compilation du code, le conditionnement du logiciel dans un format exécutable et la préparation du programme d'installation ou le package d'installation.
2. **Installation :** C'est le processus d'ajout du logiciel exécutable au matériel informatique de l'utilisateur final. Cela peut impliquer le chargement, l'installation et la configuration du logiciel sur les systèmes de l'utilisateur final. Dans le cas des applications web, cela peut impliquer le déploiement de l'application sur un serveur web ou dans le cloud.
3. **Activation :** Cette étape peut impliquer la configuration du logiciel pour qu'il fonctionne correctement une fois installé. Cela peut inclure la définition des paramètres de configuration, des préférences d'utilisateur ou l'activation du logiciel par une licence.
4. **Mise à jour :** Une fois le logiciel déployé et en usage, il sera mis à jour de temps en temps pour ajouter de nouvelles fonctionnalités, corriger des bugs ou améliorer la performance. De nombreuses entreprises utilisent des approches de déploiement continu pour publier les mises à jour et les améliorations de manière plus fréquente et régulière.

La phase de déploiement comprend également la **documentation** de l'utilisation du logiciel, la **formation** des utilisateurs à l'utilisation du logiciel, le **support technique** pour résoudre les problèmes de déploiement et l' **assistance** à l'utilisateur pour résoudre les problèmes rencontrés lors de l'utilisation.

Il est également crucial pour l'organisation de recueillir des **feedbacks après le déploiement**. Les retours des utilisateurs peuvent être utilisés pour améliorer les futures versions du logiciel.

Le **succès de la phase de déploiement** repose sur un large éventail de facteurs, dont la *qualité* du logiciel, la *préparation de l'infrastructure*, la *documentation* et le *support* fournis aux

utilisateurs et la *compréhension de l'utilisateur final de la façon d'utiliser le logiciel efficacement*.

En somme, le déploiement marque la transition du développement de logiciel à l'exploitation en temps réel, impliquant ainsi une multitude de tâches planifiées et coordonnées pour s'assurer que le logiciel est correctement installé, configuré, exécuté et maintenu.

Maintenance

La maintenance du logiciel est la dernière et la plus longue phase du cycle de vie du développement d'un logiciel. Elle débute après le déploiement du logiciel et se poursuit tant que le logiciel est utilisé. Son objectif est d' **assurer que le logiciel continue à fonctionner efficacement**, sans faille, et **reste pertinent et utile** pour les utilisateurs.

La maintenance du logiciel implique plusieurs activités, parmi lesquelles :

1. **Correction de bugs :** Malgré les tests exhaustifs effectués pendant le développement, il est possible que certains bugs aient échappé à la détection. Ces bugs sont généralement identifiés par les utilisateurs pendant l'utilisation réelle du logiciel. Ils doivent être triés, reproduits, corrigés, testés à nouveau et déployés.
2. **Amélioration des performances :** Parfois, certaines parties du logiciel peuvent avoir besoin d'être optimisées pour améliorer les performances. Cela peut impliquer l'analyse et l'optimisation du code, la mise à jour de certaines bibliothèques ou des ajustements de la configuration.
3. **Mise à jour :** Avec l'évolution rapide de la technologie, du matériel et de l'environnement d'exploitation, le logiciel doit lui aussi évoluer pour rester compatible et performant. Cela peut impliquer des mises à jour pour rester compatible avec les nouveaux systèmes d'exploitation, navigateurs web, normes de sécurité, etc.
4. **Ajout de nouvelles fonctionnalités :** Les besoins des utilisateurs évoluent avec le temps. Ils peuvent demander de nouvelles fonctionnalités ou des modifications des fonctionnalités existantes. L'ajout de nouvelles fonctionnalités implique souvent une miniversion du cycle de vie du développement logiciel à l'intérieur de la phase de maintenance.
5. **Support :** Les utilisateurs peuvent rencontrer des problèmes ou avoir des questions concernant l'utilisation du logiciel. L'équipe de maintenance devrait être disponible pour aider les utilisateurs et résoudre leurs problèmes.

La maintenance du logiciel est généralement classée en quatre types :

- **Maintenance corrective :** Elle implique la correction des bugs et des erreurs découverts après le déploiement du logiciel.
- **Maintenance adaptative :** Elle concerne les adaptations ou modifications du logiciel pour

qu'il reste compatible avec l'évolution de l'environnement.

- **Maintenance préventive :** Il s'agit d'activités visant à éviter les problèmes futurs, comme l'optimisation du code pour prévenir les problèmes de performance.
- **Maintenance évolutive :** Elle implique l'ajout de nouvelles fonctionnalités ou la modification des fonctionnalités existantes pour répondre aux besoins changeants des utilisateurs.

En somme, la maintenance du logiciel est une étape cruciale pour garantir la durée de vie du logiciel et sa valeur continue pour les utilisateurs. Elle nécessite une planification détaillée, une gestion efficace des ressources et un engagement à long terme pour garantir la qualité et l'utilité du logiciel pendant toute sa durée de vie.

Procédés de développement

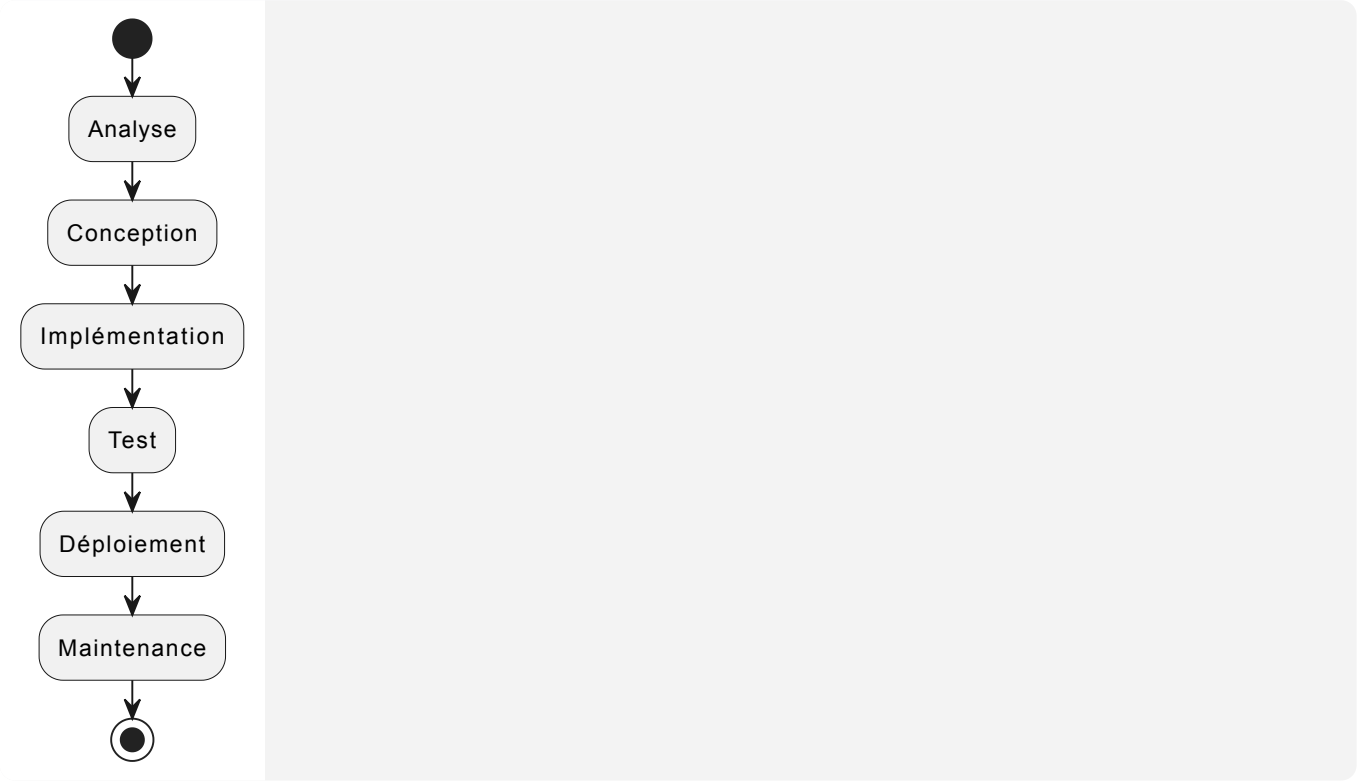
Il existe plusieurs approches pour le développement de logiciels. Voici les plus courants :

1. **Développement en cascade (Waterfall) :** C'est une approche linéaire où le développement progresse à travers des phases définies, comme l'analyse des exigences, la conception, l'implémentation, les tests, le déploiement et la maintenance. Chaque phase dépend de l'accomplissement de la précédente.
2. **Développement itératif et incrémental :** Dans cette approche, le logiciel est développé et livré en petits morceaux ou "incréments". Chaque incrément représente une version complète et utilisable du logiciel, bien que limitée en termes de fonctionnalités.
3. **Développement Agile :** C'est un ensemble de principes pour le développement de logiciels dans lesquels les exigences et les solutions évoluent grâce à la collaboration entre les équipes autogérées et inter-fonctionnelles. Les méthodes agiles favorisent des réponses flexibles au changement.
4. **Développement Lean :** Cette approche se concentre sur l'élimination des déchets dans le processus de développement de logiciels. Les activités qui n'ajoutent pas de valeur au produit sont considérées comme un "gaspillage" et sont donc éliminées pour rationaliser le processus.
5. **Développement piloté par les tests (Test-Driven Development, TDD) :** Dans cette approche, les tests sont écrits avant que le code ne soit développé. Les développements sont ensuite réalisés pour faire passer ces tests.
6. **Développement piloté par le comportement (Behavior-Driven Development, BDD) :** C'est une extension du TDD qui met l'accent sur la description du comportement du logiciel en termes compréhensibles pour les parties prenantes non techniques.
7. **Intégration continue (Continuous Integration) :** Dans cette pratique, les développeurs intègrent régulièrement leur travail dans un dépôt centralisé. Après chaque intégration, le logiciel est automatiquement testé et construit pour détecter les erreurs le plus tôt possible.
8. **Déploiement continu (Continuous Deployment) :** C'est une approche où chaque modification du code qui passe le processus d'intégration continue est automatiquement déployée en production.

Chaque approche a ses avantages et ses inconvénients, c'est pourquoi le choix dépend du contexte du projet, du type de logiciel, de l'équipe de développement, des compétences disponibles, etc. Il est courant que les équipes combinent diverses approches pour créer un processus qui répond le mieux à leurs besoins spécifiques.

Développement en cascade

Le développement en cascade, aussi appelé **modèle en cascade**, est l'un des modèles méthodologiques les plus anciens et les plus compréhensibles pour le développement de logiciels. Dire qu'un processus est en cascade est une autre façon de dire qu'il est **séquentiel**. Dans une cascade, l'eau coule pas à pas, progressivement vers le bas. De la même manière, le processus de développement en cascade est également progressif, avançant à travers différentes phases.



Voici les phases du développement en cascade :

Phase d'analyse des exigences : C'est le premier stade du processus de développement. Dans cette phase, les développeurs communiquent avec les clients et les parties prenantes pour recueillir toutes les exigences des utilisateurs. Ces exigences sont alors documentées de manière exhaustive pour être utilisées comme base pour les phases suivantes du projet.

Phase de conception : Cette phase comprend la conception du logiciel et de son architecture. Les exigences recueillies lors de la phase d'analyse sont utilisées pour concevoir la solution. Cela peut impliquer une conception de haut niveau, où l'architecture générale du système est définie, ainsi qu'une conception de bas niveau, où les détails spécifiques de chaque composant du système sont définis.

Phase d'implémentation : C'est dans cette phase que le code du logiciel est réellement écrit. En utilisant les dessins et les spécifications fournies lors de la phase de conception, les développeurs écrivent le code qui donnera vie au logiciel.

Phase de test : Une fois le code écrit, le logiciel entre dans la phase de test. Ceci est essentiel pour s'assurer que le logiciel fonctionne comme prévu et qu'il est exempt de bugs. Les tests peuvent être effectués à différents niveaux, y compris des tests unitaires, des tests d'intégration, des tests de système et des tests d'acceptation.

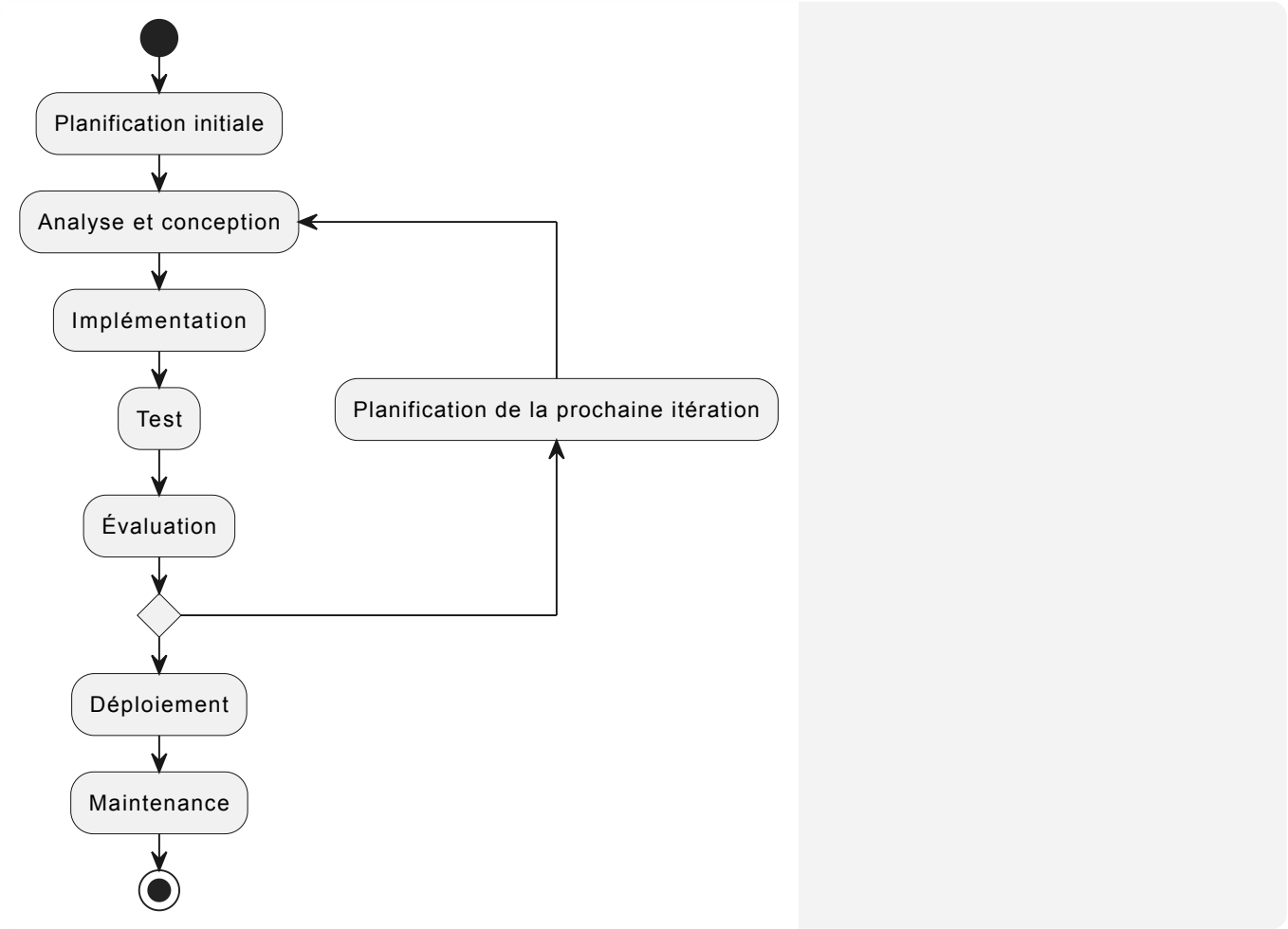
Phase de déploiement : Une fois que tout le code a été testé et que le logiciel a été accepté par les clients ou les utilisateurs, il est prêt à être déployé. Cela signifie que le logiciel est mis en ligne et mis à la disposition des utilisateurs.

Phase de maintenance : Même après le déploiement, le travail n'est pas terminé. Le logiciel peut toujours rencontrer des problèmes ou il peut y avoir des changements dans les exigences. C'est là qu'intervient la phase de maintenance. Les développeurs corrigent les erreurs, s'adaptent aux nouvelles exigences et s'assurent que le logiciel continue de fonctionner comme prévu.

L'un des principaux avantages du modèle en cascade est sa simplicité. Chaque phase a un début et une fin, et le processus progresse de manière linéaire de phase en phase. Cependant, ce modèle a des limites, notamment le fait qu'il **ne gère pas bien les changements d'exigences après le début du processus**. Ainsi, bien qu'il puisse encore être utile dans certains contextes, **de nombreuses équipes adoptent aujourd'hui des modèles plus flexibles et adaptatifs, tels que l'Agile**.

Développement itératif et incrémental

Le développement itératif et incrémental est une méthode de développement de logiciels qui privilégie la livraison de petits morceaux de fonctionnalités finies et testées au lieu de livrer tout le projet en une seule fois à la fin. Cette approche est basée sur le cycle de vie du développement de logiciels qui sont itératifs (signifie que les étapes sont répétées) et incrémentaux (signifie que chaque itération livre du travail qui ajoute aux caractéristiques précédentes).



L'approche du développement incrémental et itératif se décompose comme suit :

1. **Planification initiale :** L'étape de planification initiale comprend la détermination des besoins du système ou du produit et la documentation des exigences. Ces exigences sont alors décomposées en plusieurs blocs de travail ou groupes d'exigences.

2. **Analyse et conception :** Chaque itération du cycle de logiciel commence par une phase d'analyse des requis et de conception. Des exigences spécifiques ou des parties des exigences sont choisies pour cette itération et une analyse détaillée est effectuée. Sur la base des besoins de l'utilisateur, le système est conçu pour répondre à ces exigences.

3. **Implémentation :** Les développeurs de logiciels codent les exigences de la conception pour cette itération. À la fin de cette étape, le produit logiciel devrait avoir des fonctionnalités supplémentaires au fur et à mesure qu'il est développé par incréments.

4. **Test :** Les tests sont une étape importante de chaque itération. Le code est testé pour s'assurer qu'il répond aux exigences définies pour cette itération et qu'il s'intègre correctement avec le code précédemment développé.

5. **Évaluation et planification de la prochaine itération :** Une fois une itération terminée et que le produit a été testé et évalué, l'équipe planifie la prochaine itération. Ce processus comprend une évaluation de l'itération actuelle pour identifier les leçons apprises et comment elles pourraient être appliquées pour améliorer les futures itérations.

Ce cycle de développement se poursuit, chaque itération construisant sur les fonctionnalités livrées dans les itérations précédentes, jusqu'à ce que le produit logiciel final soit complet.

L' **avantage principal de cette approche est que les risques sont minimisés**, car les problèmes peuvent être détectés et corrigés plus tôt dans le cycle de vie. Les utilisateurs voient également le produit évoluer et peuvent fournir des commentaires précieux dès le début du processus. Il permet également de **mieux adapter le produit aux besoins changeants** du client ou du marché.

Cependant, **il nécessite une bonne planification et gestion** pour s'assurer que chaque itération livre des fonctionnalités de valeur et que toutes les parties du produit fonctionnent ensemble de manière cohérente. Un accompagnement constant et une rétroaction constructive de la part des utilisateurs sont également nécessaires pour le succès de cette méthode.

Développement agile

Le développement Agile est une méthodologie de gestion de projet et de développement de logiciels qui met l'accent sur

1. la **collaboration continue**,
2. l' **amélioration continue**,
3. la **souplesse**
4. et la **livraison de produits de haute qualité**.

Il facilite l' **adaptation rapide aux changements** tout au long du processus de développement.

Le développement Agile repose sur douze principes (https://agilemanifesto.org/iso/fr/principles.html) définis dans le Manifeste Agile (https://agilemanifesto.org/iso/fr/manifesto.html), qui incluent entre autres :

1. La satisfaction du client grâce à la livraison rapide et continue de logiciels utiles.
2. L'acceptation du changement, même tardivement dans le développement.
3. Livrer fréquemment des versions fonctionnelles du logiciel.
4. La coopération quotidienne entre les clients et les développeurs tout au long du projet.
5. Construire des projets autour d'individus motivés et leur donner l'environnement et le soutien dont ils ont besoin.
6. Conduire une réflexion en face à face est la méthode la plus efficace de communication.

Ces principes mettent l'accent sur les personnes et les interactions plutôt que sur les processus et les outils, la collaboration avec les clients plutôt que la négociation de contrats, et la réponse au changement au lieu du suivi d'un plan.

Voici comment cela fonctionne :

1. **Planification de produit :** Tout commence avec l'ensemble des besoins du produit, habituellement exprimés sous forme de **user stories**. Ce sont des déclarations courtes et simples qui décrivent ce que l'utilisateur veut faire avec le logiciel du point de vue de

l'utilisateur. Ces *user stories* sont prioritaires et insérées dans le **product backlog**, qui est essentiellement une liste de toutes les fonctionnalités désirées pour le produit.

2. **Planification du sprint :** Un sprint est une période de temps donnée, généralement de deux à quatre semaines, pendant laquelle un ensemble spécifique de fonctionnalités doit être développé. Pendant la planification du sprint, l'équipe sélectionne des *user stories* du product backlog pour être réalisées lors de ce sprint. Les *user stories* choisies sont alors décomposées en tâches spécifiques.

3. **Sprint :** Pendant le sprint, l'équipe travaille sur les tâches programmées. Il est important que l'équipe maintienne une communication constante. Chaque jour, l'équipe se réunit lors d'une réunion appelée **daily scrum** ou **daily stand-up** pour discuter de l'avancement et résoudre les problèmes.

4. **Revues de sprint :** À la fin du sprint, l'équipe présente les fonctionnalités terminées aux parties prenantes lors de la revue de sprint. C'est une occasion pour les parties prenantes de donner des commentaires et d'aider à orienter le développement futur.

5. **Rétrospective de sprint :** L'équipe se réunit pour discuter de ce qui a bien fonctionné, de ce qui n'a pas fonctionné et de comment s'améliorer pour le prochain sprint. Les actions pour améliorer sont établies et mises en œuvre dans le prochain sprint.

6. **Prochain sprint :** Après la rétrospective, l'équipe commence un nouveau sprint, commençant par une autre réunion de planification de sprint.

La méthodologie Agile offre plusieurs avantages. C'est une méthode flexible qui peut facilement s' **adapter aux changements**. Il favorise un **environnement collaboratif** et **implique constamment le client** dans le processus de développement, ce qui garantit que le produit final correspond exactement à ce que le client veut.

Cependant, **il exige une communication et une collaboration solides pour réussir**, la nature flexible de la méthodologie peut également conduire à un élargissement de la portée et à des déviations par rapport aux objectifs initiaux.

La méthodologie Agile n'est pas une solution universelle et n'est pas toujours le choix optimal pour chaque projet de développement de logiciels. Cependant, lorsque c'est le cas, le développement Agile peut conduire à une plus grande satisfaction du client, une livraison plus rapide et plus efficace et une meilleure qualité du produit.

Développement lean

Le développement *Lean* est une méthode de développement de logiciels qui s'inspire des principes de fabrication *lean* mis en œuvre par des entreprises comme Toyota. Il se concentre sur

1. l' **élimination du gaspillage**,
2. l' **amélioration continue**,
3. la **collaboration continue**
4. et la **livraison de produits de haute qualité**.

Voici les étapes de la méthodologie de développement *lean*:

1. **Éliminer le gaspillage (*Waste*) :** Le but est de créer un processus qui ne produit que ce qui est nécessaire pour répondre aux besoins des clients à l'époque où ils en ont besoin. Cela comprend la réduction de tout ce qui n'ajoute pas de valeur, comme des règles de code non utilisées, des réunions inutiles, des fonctionnalités inutilisées, des délais d'attente, etc.
2. **Amplifier l'apprentissage (*Learning*) :** L'accent est mis sur un cycle de rétroaction rapide qui permet à l'équipe d'apprendre rapidement de ses erreurs. Cette approche encourage la résolution rapide des problèmes et favorise l'amélioration continue.
3. **Décider le plus tard possible (*Decide as late as possible*) :** Pour éviter de se tromper, le développement *Lean* préconise de retarder certaines décisions jusqu'à ce que vous ayez le plus d'informations possibles. Cela donne à l'équipe la flexibilité de s'adapter et de répondre aux changements.
4. **Livrer aussi rapidement que possible (*Deliver as fast as possible*) :** En développant de manière incrémentale, l'équipe est en mesure de fournir de nouvelles fonctionnalités aux clients plus rapidement. Cela peut améliorer la satisfaction du client et permettre à l'équipe de recevoir des commentaires plus rapidement pour les cycles d'amélioration.
5. **Autonomisation de l'équipe (Empower the team):** Les équipes Lean sont souvent auto-organisées, ce qui signifie qu'elles ont le pouvoir de prendre des décisions clés concernant la façon dont elles travaillent et le travail à accomplir.

6. **Intégrité du produit (Built-in integrity):** Le développement Lean s'efforce de garantir que le système fonctionne ensemble de manière intégrée et cohérente, répondant aux besoins des clients de manner consistant.
7. **Optimiser le tout (See the whole):** L'équipe doit optimiser l'ensemble du flux de travail, pas seulement des parties isolées. Le but est de voir comment chaque pièce s'insère dans le tout et de chercher à améliorer le processus global.

Dans le monde du développement de logiciels, le Lean se traduit souvent par une suite de meilleures pratiques, y compris une interaction étroite avec les clients, la mesure de l'efficacité grâce à des métriques clés, l'automatisation de la qualité et du test, et le maintien d'une équipe de développement petite et concentrée.

En résumé, la philosophie Lean s'efforce de fournir le plus de valeur avec le moins de gaspillage possible, en se concentrant sur l'efficacité, la flexibilité et la livraison rapide de produits de haute qualité.

Développement piloté par les tests (TDD)

Le développement piloté par les tests (*Test-Driven Development*, **TDD**) est une méthode de développement agile qui vise à améliorer la qualité du code et à faciliter l'entretien. En TDD, le développeur commence par écrire un test pour une fonctionnalité spécifique avant d'écrire le code qui l'implémente.

Le processus TDD suit habituellement ce cycle :

1. **Écrire un test :** Avant de commencer à coder une nouvelle fonctionnalité, le développeur écrit d'abord un test qui décrit comment cette fonctionnalité doit se comporter. **À ce stade, le test échouera**, car la fonctionnalité n'a pas encore été implémentée.
2. **Écrire le code minimum nécessaire :** Le développeur écrit ensuite le code minimum nécessaire pour que le test passe. L'objectif n'est pas de produire un code parfait, mais d'obtenir une fonctionnalité qui réussit le test.
3. **Exécuter le test :** Après avoir écrit le code, le développeur exécute le test. Si le test échoue, le développeur doit réviser et **ajuster le code jusqu'à ce que le test passe**.
4. **Refactoriser le code :** Une fois que le test passe, le développeur refactorise le code pour s'assurer qu'il est clair, simple et dépourvu de toute redondance. Le test est alors exécuté à nouveau pour vérifier que rien n'a été cassé par le refactoring.
5. **Répéter le processus :** Ce cycle de « *test - code - refactorisation*» se répète pour chaque nouvelle fonctionnalité du code.

L'approche TDD offre plusieurs avantages. Tout d'abord, cela conduit à un **code de meilleure qualité et plus fiable**, car chaque fonction est consciencieusement testée. Deuxièmement, cela **facilite l'entretien et la modification du code**, puisque le développeur peut modifier le code en toute confiance, en sachant que d'éventuelles erreurs seraient signalées par les tests. Enfin, il favorise une **meilleure compréhension de la conception et des exigences du système**, parce que les développeurs doivent penser en termes de comportements attendus avant de rédiger le code.

Cela dit, TDD n'est pas sans défis. Écrire des tests en premier peut être difficile pour les développeurs habitués à coder sans tests. De plus, il nécessite une suite de tests complète et bien organisée pour être efficace. Enfin, le TDD peut ralentir la vitesse initiale de

développement, car il faut du temps pour écrire les tests. Cependant, cette vitesse peut être récupérée plus tard grâce à un code de meilleure qualité et à moins de bugs.

Développement piloté par le comportement (BDD)

Le développement piloté par le comportement (*Behavior Driven Development, BDD*) est un sous-ensemble du développement piloté par les tests (Test Driven Development, TDD). Il étend le TDD en fournissant des langages spécifiques à un projet pour écrire des cas de test sous une forme plus naturelle et descriptive.

Voici comment les étapes typiques de la méthodologie BDD se déroulent :

1. **Définir le comportement :** La première étape en BDD consiste à définir le comportement du système depuis le point de vue de l'utilisateur. Les développeurs, en collaboration avec les parties prenantes et les utilisateurs, définissent des scénarios d'utilisation distincts pour chaque fonctionnalité du logiciel. Ces scénarios sont généralement exprimés en utilisant la syntaxe "Given... When... Then...", qui spécifie un contexte, une action et un résultat attendu.
2. **Écrire les tests :** Une fois les comportements définis, les tests sont écrits pour valider ces comportements. L'idée est d'échouer d'abord ces tests puisque le code pour le comportement spécifié n'a pas encore été implémenté.
3. **Implémenter le code :** Cette étape consiste à écrire le code qui passe les tests. L'accent est mis ici sur l'obtention de code qui passe le test et non sur l'obtention d'une solution parfaite.
4. **Exécuter les tests :** Après l'implémentation du code, les tests sont exécutés. Si les tests réussissent, cela signifie que le code implémenté correspond au comportement défini. Si un test échoue, le code est révisé jusqu'à ce que tous les tests soient réussis.
5. **Refactoriser le code :** Enfin, une fois que toutes les fonctionnalités ont été testées et validées, tout code redondant ou complexe est refactorisé en un code plus simple et plus clair.

L'une des principales valeurs ajoutées de BDD est le langage naturel qu'il utilise pour définir les comportements. Cela facilite la communication et la collaboration entre les développeurs, les testeurs, les parties prenantes non techniques et les utilisateurs finaux. De plus, la définition claire des comportements attendus aide à éviter les malentendus et les divergences dans les exigences du logiciel.

Néanmoins, comme tous les processus, BDD a ses défis. Il prend davantage de temps que les approches traditionnelles, les équipes doivent être formées à son utilisation, les scénarios doivent être constamment mis à jour pour refléter les changements dans le code, et il faut de l'expérience pour écrire de bons scénarios.

Développement par intégration continue (CI)

L'Intégration Continue (*Continuous Integration* ou *CI*) est une pratique de développement de logiciels qui c**onsiste à fusionner le travail de tous les développeurs dans un dépôt de code centralisé plusieurs fois par jour**. Le principal avantage de cette approche est la détection rapide des erreurs, ce qui permet de garantir la cohérence du projet. De plus, avec une intégration fréquente, les erreurs sont plus faciles à localiser et à résoudre car elles impliquent généralement un plus petit ensemble de modifications.

Voici un aperçu du processus d'intégration continue :

1. **Codage :** Les développeurs écrivent du code sur leurs machines locales pour implémenter une nouvelle fonctionnalité ou corriger un bug. Ils peuvent le faire en travaillant sur une branche distincte pour isoler leur travail.
2. ***Commit* et *Push*:** Une fois leur code prêt, ils le commitent dans le système de contrôle de version (par exemple, Git), puis ils poussent leurs modifications vers le dépôt centralisé.
3. **Automatisation de la compilation (*Build*) :** Dès que les modifications du code sont reçues, le serveur d'intégration continue démarre automatiquement un "build". Un script de build est généralement utilisé pour compiler le code, générer des packages, créer des versions distribuables du logiciel, etc.
4. **Exécution des tests automatisés :** Après le *build*, la suite de tests automatisée est exécutée pour s'assurer que le nouveau code n'introduit pas d'erreurs et qu'il ne casse pas le code existant. Les tests peuvent inclure des tests unitaires, des tests d'intégration, des tests de chargement, etc.
5. **Rétroaction aux développeurs :** Si le *build* ou les tests échouent, une alerte est envoyée aux développeurs pour corriger le problème dès que possible. Certains systèmes afficheront également le statut du build (réussi ou échoué) sur un tableau de bord pour une visibilité instantanée.
6. **Déploiement :** Si le build et les tests réussissent, le logiciel peut être déployé dans un environnement de test, de staging ou de production, en fonction de la stratégie de l'équipe.

L'objectif de l'intégration continue est de fournir une rétroaction rapide afin que, si un défaut est introduit dans le code de base, il puisse être identifié et corrigé le plus rapidement

possible. Cela rend les bugs plus faciles à corriger (puisque le code a été écrit récemment) et rend le logiciel plus fiable globalement.

Pour bénéficier de l'intégration continue, une organisation doit investir dans une culture de la collaboration, adhérer à des normes de codage cohérentes, maintenir une suite de tests automatisée et utiliser des outils d'intégration continue pour automatiser le processus de build et de test.

Développement par déploiement continu (CD)

Le déploiement continu (*Continuous Deployment* ou *CD*) est une méthode de développement de logiciels qui se concentre sur l'automatisation du déploiement du code dans un environnement de production.

Dans un processus de déploiement continu, chaque changement de code qui passe tous les stades du pipeline de production est déployé directement dans la production, ce qui rend les nouvelles fonctionnalités, mises à jour et correctifs disponibles aux utilisateurs finaux plus rapidement et de manière plus fiable. Le *CD* est une extension du *CI*

Voici comment cela fonctionne :

1. **Soumettre le Code :** Les développeurs soumettent leurs modifications de code dans une révision source centralisée. Dans ce système, le code est vérifié pour la qualité et pour s'assurer qu'il n'y a pas de conflits avec le code existant.
2. **Construction Automatisée :** Les modifications du code déclenchent ensuite une construction automatisée du projet. Cela signifie que le logiciel est compilé, testé, puis empaqueté pour la distribution.
3. **Tests Automatisés :** Avant de déployer, le logiciel passe par des tests automatisés pour s'assurer qu'il fonctionne comme prévu. Cela peut inclure des tests unitaires, des tests de charge, des tests d'intégration, des tests de sécurité, et plus encore.
4. **Déploiement en Production :** Si le logiciel passe tous les tests avec succès, il est automatiquement déployé dans l'environnement de production.
5. **Surveiller & Valider :** Après le déploiement, le logiciel est surveillé pour s'assurer qu'il fonctionne correctement dans un environnement en direct. Si des problèmes sont détectés, des alertes peuvent être envoyées à l'équipe de développement pour une intervention rapide.

L'un des principaux avantages du déploiement continu est qu'il permet une livraison plus rapide des nouvelles fonctionnalités et corrections aux utilisateurs. Parce que le processus est automatisé, le risque d'erreurs humaines lors du déploiement est également réduit.

Cependant, le déploiement continu n'est pas sans défis. Il exige une suite de tests automatisés robuste et fiable pour s'assurer que seules les versions de qualité sont déployées.

Il nécessite également une surveillance et une alerte efficaces pour s'assurer que les problèmes peuvent être rapidement identifiés et corrigés.

Malgré ces défis, pour de nombreuses organisations, les avantages de la capacité à livrer rapidement des améliorations logicielles de haute qualité à leurs utilisateurs l'emportent sur les complications potentielles. Le développement par déploiement continu est maintenant une pratique standard dans de nombreuses entreprises de développement logiciel de premier plan.

Analyse des besoins ou la Collecte des exigences

La phase de planification d'un projet Agile est une étape cruciale qui donne le ton et la direction du développement du logiciel. Contrairement aux méthodologies traditionnelles où la planification a lieu principalement au début du projet, dans l'Agile, **la planification est une activité continue tout au long du cycle de vie du développement**.

La première étape de la planification Agile est de définir **la vision du produit**. Cette vision est généralement déterminée par le *product owner* et fournit un aperçu de haut niveau du but du produit, des problèmes qu'il résoudra, et de qui bénéficiera de son utilisation.

Une fois la vision du produit définie, l'équipe passe à la création d'un backlog de produit. Il s'agit d'une liste organisée d'articles, généralement appelés **user stories**, ou simplement **stories**, qui représentent les fonctionnalités, modifications, améliorations ou corrections du produit à développer. Chaque *user story* contient une description de la fonctionnalité du point de vue de l'utilisateur final, ainsi que les critères d'acceptation définissant ce que signifie qu'une story est terminée. L'ordre des *user stories* dans le backlog est déterminé par leur **priorité**, qui est souvent basée sur **leur valeur pour l'utilisateur final**.

Ensuite vient **la planification de l'itération ou du sprint** (dans le cas de Scrum), qui est une période de temps fixe (généralement de deux à quatre semaines) pendant laquelle l'équipe s'engage à livrer un ensemble de *user stories*. Au cours de la planification du sprint, l'équipe choisit les user stories du backlog de produit à inclure dans le sprint, en tenant compte de leur priorité et de la capacité de l'équipe. Chaque user story est ensuite décomposée en tâches individuelles, qui sont estimées en termes de temps ou d'effort nécessaire pour les accomplir.

La communication est une partie vitale de la phase de planification Agile. Les *stand-ups* quotidiens, également appelés réunions de scrum, permettent à l'équipe de se synchroniser sur leurs progrès et d'identifier les obstacles potentiels. Les sessions de revue et de rétrospective du sprint permettent à l'équipe de discuter de ce qui a bien fonctionné et de ce qui pourrait être amélioré pour les sprints futurs.

Il est important de noter que dans l'Agile, **le plan n'est pas une chose figée**, mais est considéré comme **adaptable et évolutif**. **L'équipe doit être prête à réviser et à ajuster le plan** en fonction des retours d'information des parties prenantes, des résultats des revues du sprint, et des changements dans l'environnement du projet.

En conclusion, la phase de planification Agile est à la fois une activité préliminaire importante et un processus continu qui soutient le développement itératif et incrémental du logiciel. Son objectif est de fournir de la clarté, de l'alignement et une direction flexible à l'équipe tout au long du projet.

Parties prenantes

Dans le processus de développement Agile, les *"stakeholders"* ou **parties prenantes** réfèrent à **toute personne ou groupe qui a un intérêt direct ou indirect dans le produit ou le projet**. Dans la phase de planification du produit, les parties prenantes impliquées peuvent inclure :

1. **Utilisateurs finaux ou clients :** Les personnes qui utilisent le produit final ou en bénéficient. Leurs besoins et préférences influencent fortement les paramètres et fonctionnalités du produit.
2. **Propriétaire du produit :** Le propriétaire du produit (Product Owner) représente la voix du client dans l'équipe et fait le lien entre les parties prenantes et l'équipe de développement.
3. **Équipe de développement :** L'équipe effectuant le travail de développement du produit. Ils sont responsables de transformer le backlog du produit en un produit fini.
4. **Scrum Master :** Le scrum master aide l'équipe à appliquer la méthodologie agile, facilite les communications et résout les problèmes qui peuvent survenir pendant le processus de développement.
5. **Gestionnaires et Leaders de l'entreprise :** Les membres de la direction qui ont un intérêt dans le succès du produit et du projet y sont également inclus.
6. **Autres Équipes au sein de l'organisation :** Les équipes comme les ventes, le marketing, le support client, certaines fois peuvent être impliquées en tant que parties prenantes car le produit final affecte aussi leur travail.

Chacune de ces parties prenantes apporte une perspective valorisée à la phase de planification du produit et leurs retours aident à assurer que le produit final est bien aligné sur les besoins et attentes de ceux qui sont concernés par son développement et son utilisation.

Exploration

La phase d'exploration dans le développement de logiciels Agile est une étape essentielle qui aide à établir une compréhension claire des besoins et des exigences du projet. Cette phase est orientée vers le dialogue et la collaboration avec les parties prenantes pour définir la vision du produit.

La collecte des exigences est généralement réalisée par le biais de *user stories*, qui sont des descriptions simples et concises des fonctionnalités du point de vue de l'utilisateur final. Ces *user stories* sont ensuite rassemblées dans le *Backlog* du produit, une liste organisée et priorisée de toutes les fonctionnalités souhaitées pour le produit.

Des ateliers, des interviews ou des discussions sont souvent organisés avec les parties prenantes pour préciser les détails de chaque *user story*. Chaque User Story est accompagnée de critères d'acceptation, qui définissent les conditions que la fonctionnalité doit remplir pour être considérée comme achevée.

Contrairement à d'autres approches, l'Agile suppose que les exigences évolueront tout au long du projet. Par conséquent, l'exploration est une activité continue, permettant d'adaptation aux changements de l'environnement du projet ou des besoins des utilisateurs.

Backlog

Dans la méthodologie de développement agile, **le backlog du produit est une liste de toutes les tâches, fonctionnalités, corrections de bugs et autres exigences du système**, qui sont nécessaires pour réaliser un projet de logiciel ou de produit. C'est de loin **le document le plus important pour toute équipe travaillant en méthodologie agile.**

Chaque *user story* dans le *backlog* représente une exigence spécifique du point de vue de l'utilisateur final. **Les user stories sont classées par ordre de priorité**, des plus importantes aux moins importantes, et sont utilisées pendant les réunions de planification de sprint pour déterminer ce qui doit être réalisé lors du prochain sprint.

Dans le backlog du produit, les user stories en haut de la liste sont généralement plus détaillées et davantage prêtes pour le développement, tandis que celles qui sont plus bas peuvent nécessiter plus de détails ou de clarification.

Il faut noter que **le backlog du produit est un document vivant qui change et évolue au fur et à mesure que le projet avance** et que de nouvelles informations ou exigences sont découvertes.

User stories

Dans la méthodologie Agile, les exigences fonctionnelles sont principalement représentées par des *user stories*. Une *user story* est une description de haute qualité d'une fonctionnalité telle qu'elle est perçue par l'utilisateur. **Elle favorise la pensée centrée sur l'utilisateur**, l'accent étant mis sur qui a besoin de la fonction, ce qu'elle doit faire et pourquoi elle est nécessaire.

Cependant, il est à noter que toutes les exigences fonctionnelles ne peuvent pas toujours être décomposées en *user stories*, notamment dans le cas de systèmes complexes ou de fonctions plus techniques. Dans de tels cas, d'autres formes de représentation des exigences pourraient être utilisées, telles que les cas d'utilisation, ou des approches spécifiques comme *Behaviour Driven Development* (BDD).

Les user stories sont un outil utilisé en développement Agile pour capturer une description simple, concise et orientée- utilisateur d'une fonctionnalité d'un produit. Elles sont écrites à partir de la perspective de l'utilisateur final, en décrivant une partie de la fonctionnalité que cet utilisateur souhaite pour atteindre un certain objectif.

Le format traditionnel d'une user story est: *"En tant que (un rôle/utilisateur), je veux (un objectif) afin que (une raison/bénéfice)"*. Par exemple, *"En tant que client, je veux pouvoir rechercher des articles par nom afin de trouver rapidement ce que je cherche."*

Comment sont-elles utilisées ?

Les *user stories* sont utilisées comme un moyen de **capturer les besoins des utilisateurs sans entrer dans les détails techniques spécifiques** de comment la fonctionnalité sera mise en œuvre. Elles sont souvent écrites sur des fiches ou des tickets et utilisées pour nourrir le *backlog* de produits Agile, qui est une liste priorisée de toutes les fonctionnalités à développer.

Les *user stories* jouent également un rôle important dans la planification de sprint, où l'équipe de développement choisit une sélection de *user stories* à partir du *backlog* du produit pour développer pendant le sprint.

À quoi servent-elles ?

Les *user stories* servent plusieurs objectifs :

- **Simplicité**: Elles aident à garder le processus de planification simple et flexible en décrivant les fonctionnalités en termes d'utilisation plutôt qu'en spécifications techniques.

- **Communication**: Elles favorisent la communication entre les membres de l'équipe et leur permettent de comprendre les besoins des utilisateurs.
- **Priorisation**: Lorsqu'elles sont utilisées dans un backlog de produits, les user stories aident à donner la priorité aux fonctionnalités en fonction de la valeur ajoutée à l'utilisateur.
- **Base pour les tests d'acceptation**: Les user stories peuvent être utilisées pour créer des tests d'acceptation en définissant le critère d'acceptation ou le comportement attendu.

En somme, **les *user stories* permettent de capturer les besoins de l'utilisateur de manière simple et axée sur l'utilisateur et aident à guider le développement du produit.**

Écrire des user stories

Les *user stories* sont courtes, elles sont des descriptions simples d'une fonctionnalité du point de vue de l'utilisateur final. Elles fonctionnent comme un moyen de créer une liste de tâches centrée sur l'utilisateur pour le développement de produits.

Voici les composantes essentielles d'une *user story*:

1. **Le rôle de l'utilisateur**: Cela définit qui est l'utilisateur de la fonctionnalité.
2. **Le but**: C'est ce que l'utilisateur veut accomplir.
3. **La valeur**: C'est la raison pour laquelle l'utilisateur a besoin de cette fonctionnalité.

Une *user story* est généralement formulée selon le modèle suivant :

```
En tant que <rôle>, je veux <objectif/désir> afin que <bénéfice>.
```

Par exemple :

```
En tant que client, je veux pouvoir me connecter au site avec Facebook
afin de faciliter le processus de connexion.
```

Il est également recommandé d'ajouter des critères d'acceptation à votre *user story*. Ce sont des détails supplémentaires, souvent sous forme de liste, qui précisent le comportement attendu de la fonctionnalité. Par exemple :

```
Critères d'acceptation :
1. Quand un client visite la page de connexion, une option "Se
connecter avec Facebook" est visible.
2. Si le client clique sur "Se connecter avec Facebook", il est
redirigé vers une page de connexion Facebook.
3. Après une connexion réussie avec Facebook, le client est redirigé
vers notre site et est connecté.
```

En écrivant des *user stories*, vous vous assurez que **vous développez des fonctionnalités qui correspondent aux besoins réels des utilisateurs**, en vous mettant à leur place. **Utilisez des**

termes simples, compréhensibles par tous les membres de l'équipe et évitez autant que possible le jargon technique.

Liens avec le BDD

Le BDD (*Behavior-Driven Development*, ou développement piloté par le comportement) est une pratique de développement de logiciels qui encourage la collaboration entre développeurs, QA et non-techniciens ou parties prenantes commerciales dans un projet logiciel. Il se concentre sur la réalisation de l'objectif du comportement des fonctionnalités d'un produit logiciel.

Les *user stories* fournissent une description simple et compréhensible des exigences pour une fonctionnalité particulière d'un produit logiciel du point de vue de l'utilisateur final.

Le lien entre les *user stories* et BDD devient manifeste au niveau des scénarios d'acceptation. Ces scénarios reflètent comment une fonctionnalité donnée devrait se comporter pour être considérée comme satisfaisante pour l'utilisateur final. Ils aident à clarifier les exigences et font également souvent office de test d'acceptation.

Dans BDD, ces scénarios sont exprimés sous la spec BDD "Given, When, Then" format qui ressemble à ceci :

- **Étant donné (Given)** (état initial du système)
- **Quand (When)** (une action effectuée par l'utilisateur)
- **Alors (Then)** (le résultat attendu de l'action)

Par exemple, en utilisant la *user story* précédente :

```
En tant que client, je veux pouvoir me connecter au site avec Facebook
afin de faciliter le processus de connexion.
```

Un scenario BDD pour cela pourrait être formulé comme suit :

```
Étant donné  que je suis un client visitant la page de connexion
Quand        je clique sur l'option "Se connecter avec Facebook"
Alors        je devrais être redirigé vers la page de connexion
Facebook
Et           après une connexion réussie, je devrais être redirigé
vers le site et être connecté.
```

En résumé, les *user stories* et BDD sont étroitement liés. **Les *user stories* aident à définir la fonctionnalité et les scénarios BDD déterminent le comportement attendu de cette fonctionnalité**.

Exemples

Rappel: dans un premier temps, on peut seulement écrire la description des user stories, et dans un deuxième temps, écrire les critères d'acceptation.

Générateur de mots de passe

Voici quelques exemples de User Stories pour un logiciel qui génère des mots de passe :

1. **Génération de mot de passe de base**
   - En tant qu'utilisateur, je veux pouvoir générer un nouveau mot de passe afin que je puisse créer rapidement un mot de passe sécurisé.
   - Critères d'acceptation :
     1. Un nouveau mot de passe est généré chaque fois que l'utilisateur le demande.
     2. Le mot de passe généré est une chaîne aléatoire de caractères.
2. **Personnalisation de la longueur du mot de passe**
   - En tant qu'utilisateur, je veux spécifier la longueur de mon mot de passe généré afin que je puisse répondre à des exigences spécifiques de longueur de mot de passe.
   - Critères d'acceptation :
     1. L'utilisateur peut entrer une longueur de mot de passe souhaitée.
     2. Le mot de passe généré correspond à la longueur spécifiée par l'utilisateur.
3. **Inclusion de types de caractères spécifiques**
   - En tant qu'utilisateur, je veux choisir d'inclure des chiffres, des caractères spéciaux et des lettres majuscules ou minuscules dans mon mot de passe généré afin de répondre à des exigences de complexité spécifiques.
   - Critères d'acceptation :
     1. L'utilisateur peut choisir d'inclure des chiffres, des caractères spéciaux, des lettres majuscules et/ou minuscules.
     2. Le mot de passe généré inclut les types de caractères spécifiés par l'utilisateur.

4. **Exclusion de caractères similaires**

- En tant qu'utilisateur, je veux pouvoir exclure des caractères similaires (comme "0" et "O" ou "l" et "1") dans les mots de passe générés pour éviter la confusion.
- Critères d'acceptation :
  1. L'utilisateur peut choisir d'exclure des caractères similaires.
  2. Le mot de passe généré n'inclut pas de paires de caractères similaires si l'utilisateur a choisi cette option.

5. **Copie du mot de passe généré**

- En tant qu'utilisateur, je veux pouvoir copier facilement le mot de passe généré dans le presse-papiers afin de pouvoir l'utiliser sans avoir à le taper manuellement.
- Critères d'acceptation :
  1. L'utilisateur peut copier le mot de passe généré dans le presse-papiers en un seul clic.
  2. Le mot de passe reste disponible dans le presse-papiers pour être collé ailleurs.

6. **Génération de plusieurs mots de passe en une fois**

- En tant qu'utilisateur, je veux générer une liste de plusieurs mots de passe à la fois afin de pouvoir en choisir un que j'aime.
- Critères d'acceptation :
  1. L'utilisateur peut spécifier le nombre de mots de passe qu'il souhaite générer à la fois.
  2. La quantité spécifiée de mots de passe uniques est générée.

Manipulation d'images

Voici quelques exemples de User Stories pour un logiciel de manipulation d'images :

1. **Ouverture de fichiers image**

- En tant qu'utilisateur, je veux ouvrir un fichier image depuis ma machine locale pour pouvoir effectuer des modifications sur l'image.
- Critères d'acceptation :

1. L'utilisateur peut parcourir les fichiers de son système local.
2. L'utilisateur peut sélectionner et ouvrir un fichier image dans le logiciel.
3. Les formats d'image courants (par exemple, .jpeg, .png, .bmp) sont supportés.

2. **Redimensionnement de l'image**

- En tant qu'utilisateur, je veux être capable de redimensionner une image à une taille spécifique pour que je puisse l'adapter à mes besoins.
- Critères d'acceptation :
  1. L'utilisateur peut spécifier les nouvelles dimensions en pixels.
  2. Le logiciel redimensionne l'image conformément à la nouvelle taille spécifiée.
  3. L'image redimensionnée est affichée à l'utilisateur.

3. **Conversion de format de l'image**

- En tant qu'utilisateur, je veux convertir une image d'un format à un autre afin que je puisse l'utiliser dans différentes applications qui peuvent nécessiter des formats spécifiques.
- Critères d'acceptation :
  1. L'utilisateur peut choisir parmi une liste de formats d'image disponibles pour la conversion.
  2. Le logiciel convertit l'image selon le format spécifié.
  3. L'image convertie est enregistrable par l'utilisateur.

4. **Rognage de l'image**

- En tant qu'utilisateur, je veux pouvoir rogner une portion de l'image afin de me concentrer sur l'élément le plus important de l'image.
- Critères d'acceptation :
  1. L'utilisateur peut sélectionner une portion de l'image à rogner avec un outil de sélection.
  2. Le logiciel rogne la portion d'image sélectionnée et affiche le résultat à l'utilisateur.

5. **Rotation de l'image**

- En tant qu'utilisateur, je veux pouvoir faire pivoter une image pour changer son orientation.
- Critères d'acceptation :

  1. L'utilisateur peut spécifier l'angle de rotation pour l'image.
  2. Le logiciel fait pivoter l'image conformément à l'angle spécifié.
  3. L'image pivotée est affichée à l'utilisateur.

6. **Application de filtres à l'image**

- En tant qu'utilisateur, je souhaite avoir la possibilité d'appliquer divers filtres à mes images, afin de pouvoir améliorer ou modifier leur apparence selon mes préférences.
- Critères d'acceptation :

  1. Une liste de filtres disponibles est présentée à l'utilisateur.
  2. L'utilisateur peut sélectionner un filtre et l'appliquer à l'image.
  3. L'image avec le filtre appliqué est affichée à l'utilisateur.

Exigences non-fonctionnelles

Les exigences non fonctionnelles, aussi appelées "attributs de qualité", spécifient les critères globaux du système, tels que la performance du système, la sécurité, et l'interface utilisateur. Elles ne sont pas liées à des fonctionnalités spécifiques, mais caractérisent plutôt la performance et les qualités du système dans son ensemble.

Voici comment les exprimer :

1. **Performance**: Les exigences de performance spécifient le temps de réponse, le débit, le nombre d'utilisateurs simultanés, etc. Par exemple, "Le système doit pouvoir gérer 1000 utilisateurs simultanément sans ralentissement notable".
2. **Fiabilité/Disponibilité**: Il s'agit de la quantité de temps pendant laquelle le système est opérationnel et du temps nécessaire pour se rétablir en cas de panne. Par exemple, "Le taux de disponibilité du système doit être de 99,99%".
3. **Maintenabilité**: Les exigences doivent préciser à quelle vitesse les ajouts ou modifications peuvent être effectués. Par exemple, "Les nouvelles fonctionnalités doivent pouvoir être ajoutées au système sans plus de deux semaines de travail de développement".
4. **Sécurité**: Ces exigences précisent le niveau de sécurité que le système doit offrir en termes d'accès non autorisé, de perte de données et de confidentialité. Par exemple, "Toutes les données utilisateur doivent être cryptées pour protéger la confidentialité des informations".
5. **Interface utilisateur**: Les exigences peuvent préciser certaines propriétés de l'interface utilisateur, telles que l'ergonomie, le confort de lecture, la simplicité. Par exemple, "L'interface utilisateur doit être disponible en français et en anglais".
6. **Conformité aux régulations et standards**: Ces exigences impliquent que le système doit être conforme à certaines lois ou réglementations. Par exemple, "Le système doit être conforme à la RGPD".

Cependant, il est important que toutes ces exigences soient mesurables et testables. Par conséquent, spécifiers les de manière claire et précise est crucial pour le développement du logiciel.

Document vision

Introduction

But

Ce document offre un aperçu de haut niveau du logiciel XYZ, en esquissant le but, l'étendue et la base d'utilisateurs prévus.

Portée

Ce document décrira le développement prévu du logiciel XYZ, une application innovante conçue pour [le but du logiciel].

Parties prenantes

- Commanditaire du projet : [Nom]
- Propriétaire du projet : [Nom]
- Équipe de développement : [Nom]
- Utilisateurs finaux : [Description des utilisateurs]

Vue d'ensemble du produit

Perspective du produit

Le logiciel XYZ sera une application autonome ciblant [l'audience cible]. Son principal objectif est de [fonctionnalité/principal objectif du logiciel].

Fonctions du produit

- **Fonction 1** - [Décrire la première fonction principale du produit]
- **Fonction 2** - [Décrire la deuxième fonction principale du produit]
- [...]

User stories

User stories (avec ou sans critères d'acceptation). Possiblement dans un document en annexe.

Classes et caractéristiques des utilisateurs

- **Classe d'utilisateur 1** - [Décrire le premier type d'utilisateur et leurs caractéristiques]
- **Classe d'utilisateur 2** - [Décrire le deuxième type d'utilisateur et leurs caractéristiques]
- [...]

Diagramme de contexte

Caractéristiques du produit

- **Caractéristique 1** - [Décrire la première caractéristique du produit]
- **Caractéristique 2** - [Décrire la deuxième caractéristique du produit]
- [...]

Environnement de fonctionnement

Le logiciel fonctionnera sur les plateformes Windows 10 et Linux, exécutant la version 21 de Java JRE.

Contraintes de conception et d'implémentation

Le logiciel doit être développé à l'aide de Java. Le cycle de développement doit respecter [Méthodologie de Développement Spécifique], et doit obéir à tous les [Contrôles de Qualité Spécifiques].

Documentation utilisateur

Une documentation utilisateur détaillée et une aide en ligne seront fournies avec le package du logiciel.

Hypothèses et dépendances

- Le logiciel est dépendant des utilisateurs finaux ayant la version 21 de Java JRE sur leur

système.

- [...]

Exemples de contenu d'un document vision

Exemples de fonctions de produits

Fonctions du logiciel de gestion de projets

1. **Planification de projet** - Permet aux utilisateurs de définir des échéances et des jalons pour les tâches et sous-tâches, créant ainsi un calendrier de projet visuel.
2. **Suivi des progrès** - Fournit un aperçu en temps réel de l'état d'avancement des tâches du projet, permettant ainsi une gestion efficace des ressources.

Fonctions du logiciel de traitement de texte

1. **Édition de texte** - Donne aux utilisateurs les outils nécessaires pour modifier le texte, y compris changer la taille et le type de police, ajouter des couleurs, etc.
2. **Vérification grammaticale** - Analyse le texte pour les fautes d'orthographe et de grammaire et suggère des corrections.

Fonctions du logiciel de commerce électronique

1. **Gestion de l'inventaire** - Permet la gestion intégrée de l'inventaire, y compris le suivi des stocks, la mise à jour des quantités, etc.
2. **Passerelle de paiement** - Intègre des options de paiement pour traiter les transactions en ligne de manière sécurisée.

Ces exemples sont assez génériques et devraient être adaptés pour correspondre à la nature spécifique et unique de votre projet.

Exemples de classes d'utilisateurs et de leurs caractéristiques

1. **Administrateurs** - Ils ont le niveau le plus élevé d'autorisation et peuvent effectuer des tâches telles que la gestion des utilisateurs, la configuration du logiciel et la visualisation

des rapports globaux. Ils possèdent des compétences techniques approfondies et une compréhension claire du fonctionnement interne du logiciel.

2. **Utilisateurs réguliers** - Ils forment le principal groupe d'utilisateurs du logiciel. Ils utilisent les fonctionnalités de base du logiciel pour effectuer leurs tâches quotidiennes. Ils peuvent avoir des compétences techniques variées, allant de basiques à intermédiaires.

3. **Utilisateurs externes** - Parfois, une partie du logiciel peut être accessible à des utilisateurs qui ne sont pas directement associés à l'organisation, tels que les clients ou les partenaires. Ces utilisateurs peuvent accéder à des fonctionnalités spécifiques et limitées en fonction des permissions qui leur sont accordées. Ils peuvent ou ne pas avoir de compétences techniques.

4. **Utilisateurs d'API** - Il s'agit de systèmes tiers qui interagissent avec le logiciel via une interface de programmation d'application (API). Ils peuvent récupérer, créer, mettre à jour et supprimer des données à partir du logiciel. Ces utilisateurs ont des compétences techniques très développées, étant souvent des développeurs ou des ingénieurs logiciels travaillant sur l'intégration du logiciel avec d'autres systèmes.

Chaque classe d'utilisateurs aura probablement des interactions et des expériences différentes avec le logiciel, et leurs besoins et leur feedback seront différents, influençant ainsi la conception et le développement futurs du produit logiciel.

Exemples de caractéristiques de produits

Logiciel de gestion de projets

1. **Tableau de bord du projet**: Affiche une vue d'ensemble de tous les projets en cours avec des informations clés comme l'état, les membres de l'équipe, les échéances, etc.

2. **Gestion des tâches**: Permet l'attribution de tâches, le suivi du temps et l'établissement de priorités pour gérer efficacement le travail.

Logiciel de traitement de texte

1. **Formatage de texte riche**: Permet d'appliquer des styles de texte tels que gras, italique, souligné, etc.

2. **Insertion d'images et de graphiques**: Offre la possibilité d'ajouter des images ou des graphiques dans le document.

Logiciel de commerce électronique

1. **Panier d'achat**: Permet aux clients de sélectionner plusieurs articles à acheter en même temps.
2. **Évaluations et commentaires**: Permet aux clients de noter et de commenter les produits, améliorant l'expérience d'achat des autres clients.

Logiciel de suivi du temps

1. **Chronomètre**: Permet aux utilisateurs de commencer et de stopper un chronomètre pour suivre le temps passé sur une tâche.
2. **Rapports**: Fournit des aperçus détaillés de l'utilisation du temps pour une analyse plus approfondie.

Exemples d'environnements de fonctionnement

1. **Logiciel de bureau**: Peut fonctionner sur divers systèmes d'exploitation tels que Windows 10, MacOS Big Sur et diverses distributions Linux. Les spécificités dépendent du logiciel lui-même et de sa compatibilité avec ces systèmes d'exploitation.
2. **Application Web**: S'exécute dans un navigateur web et devrait être compatible avec les navigateurs les plus populaires (Google Chrome, Mozilla Firefox, Safari et Microsoft Edge). De plus, cette application nécessite souvent un serveur web et une base de données pour fonctionner.
3. **Application mobile**: Fonctionne sur des systèmes d'exploitation mobiles. Dans la plupart des cas, cela se résume à Android et iOS, qui sont les deux principaux systèmes d'exploitation mobiles sur le marché.
4. **Logiciel embarqué**: Fonctionne sur un appareil spécifique et est souvent conçu pour une plate-forme matérielle spécifique. Par exemple, un logiciel pour une machine à laver, une voiture ou un drone.
5. **Service en nuage**: Fonctionne sur une plate-forme de cloud computing comme AWS (Amazon Web Services), Microsoft Azure ou Google Cloud Platform. Le logiciel est généralement accessible via le web ou une API.

6. **Logiciel de serveur**: Fonctionne sur un serveur et fournit des services à d'autres logiciels ou appareils sur le réseau. Les exemples incluent les serveurs de bases de données, les serveurs web et les serveurs de fichiers.

Rappelez-vous, chaque logiciel a son propre environnement de fonctionnement unique en fonction de son but, de sa plate-forme cible et de ses besoins en ressources.

Exemples de contraintes de conception et d'implémentation

Les contraintes de conception et d'implémentation sont des limites spécifiées ou imposées sur les options de conception et d'implémentation d'un produit. Voici quelques exemples :

1. **Contraintes de plateforme**: Le logiciel doit être développé pour être compatible avec une certaine version de l'environnement d'exécution Java (comme Java 8 ou Java 11) ou un système d'exploitation spécifique (par exemple, Windows 10).
2. **Contraintes de performance**: Le logiciel doit être capable de gérer un certain nombre d'utilisateurs concurrents, ou d'effectuer une certaine fonction en moins d'une certaine quantité de temps.
3. **Contraintes de sécurité**: Le logiciel doit être conforme à des normes de sécurité spécifiques, comme la norme de sécurité des données de l'industrie des cartes de paiement (PCI DSS) pour le logiciel de traitement des paiements, ou le règlement général sur la protection des données (RGPD) pour le logiciel utilisé par les entreprises de l'UE.
4. **Contraintes d'accessibilité**: Le logiciel doit être accessible aux utilisateurs ayant différents types de handicaps, conformément aux directives WCAG 2.1.
5. **Contraintes légales et de conformité**: Le logiciel doit respecter les lois et régulations liées à son champ d'application, comme la HIPAA pour les systèmes de santé aux États-Unis ou la Loi Informatique et Libertés en France.
6. **Contraintes de ressources**: Limitations liées au budget, au temps, à la technologie disponible ou aux compétences du personnel. Par exemple, le projet doit être complété en six mois avec une équipe de cinq développeurs.
7. **Contraintes architecturales**: Le système doit être conçu en utilisant une certaine architecture, comme une architecture microservices, ou doit être compatible avec une architecture existante.

8. **Contraintes d'interface utilisateur (UI)**: Le logiciel peut nécessiter une compatibilité avec certaines résolutions d'écran, tailles de police, ou styles de couleur.

Exemples de types de documentation utilisateur

Documentation en ligne et/ou papier, sous-forme de

1. site web
2. fichiers PDF
3. tutoriels (texte, vidéos)
4. blog
5. wiki
6. FAQ
7. manuel d'utilisateur
8. *cheat sheet*

Exemples d'hypothèses et dépendances

Les hypothèses et les dépendances sont les conditions préalables aux décisions de développement et de déploiement du produit. Ce sont les facteurs qui pourraient affecter la mise en œuvre du plan ou le succès du produit.

Voici quelques exemples :

1. **Hypothèses**:
   - *Technologiques*: On suppose par exemple que la majorité des utilisateurs utiliseront un navigateur moderne compatible avec les dernières normes web (HTML5, CSS3).
   - *Commerciales*: On suppose que le marché cible pour le produit est stable et croissant, et que la demande pour cette catégorie de produits continuera d'augmenter au cours des prochaines années.
   - *Ressources disponibles*: On suppose que l'équipe de développement aura accès à toutes les compétences nécessaires pour le projet, par exemple des développeurs Java expérimentés.

2. **Dépendances**:

- *Externes*: Le succès du produit peut dépendre de partenaires ou de fournisseurs tiers. Par exemple, une application mobile peut dépendre d'un fournisseur de paiement pour les transactions en app.
- *Technologiques*: Le développement du produit peut dépendre de technologies spécifiques ou de logiciels externes. Par exemple, le déploiement de l'application pourrait dépendre de plateformes cloud spécifiques comme AWS ou Azure.
- *Réglementaires*: Il peut y avoir des dépendances réglementaires, par exemple, le produit doit être en conformité avec des lois provinciales et/ou fédérales sur la gestion des données privées.

Les hypothèses et dépendances devraient être réévaluées régulièrement pendant le cycle de vie du produit, car les conditions peuvent changer.

Diagramme de contexte

Un diagramme de contexte est un outil important de la modélisation de systèmes qui fournit **une vue de haut niveau de l'interaction entre un système et son environnement externe**. Il est souvent utilisé dans le cadre de l'analyse des exigences dans le processus de développement logiciel.

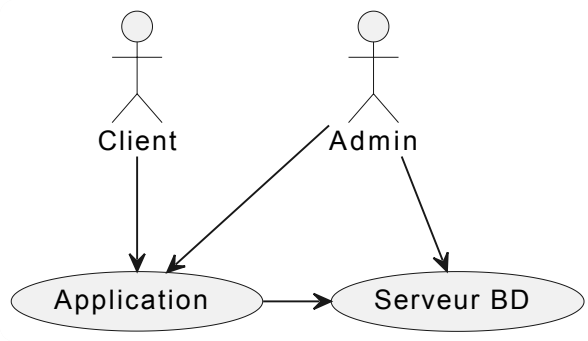
Le diagramme de contexte montre spécifiquement :

- Le **système considéré** (généralement représenté par un cercle ou une boîte au centre du diagramme).
- Les **acteurs externes** (qui peuvent être des personnes, des systèmes ou d'autres entités) qui interagissent avec le système. Ces acteurs sont généralement représentés comme des boîtes ou des cercles à l'extérieur du système.
- Les **interactions** ou les **canaux de communication** entre le système et les acteurs extérieurs. Ces interactions sont souvent représentées par des flèches ou des lignes qui montrent la direction du flux d'information ou des interactions.

L' **objectif** du diagramme de contexte est de **simplifier la compréhension du système** en le représentant de manière schématique et de porter une attention particulière aux interactions entre le système et son environnement externe. Il aide à comprendre les limites du système et comment il s'intègre dans un contexte plus large.

C'est un outil particulièrement utile pour les analystes de systèmes et les gestionnaires de projet pour comprendre et représenter les relations entre un système et ses acteurs externes.

Exemple



Techniques d'entrevue

Lors de rencontres avec des clients, les équipes de développement de logiciels utilisent une variété de techniques d'entrevue pour comprendre les besoins et les attentes du client. Voici quelques-unes des techniques les plus couramment utilisées :

1. **Les entrevues structurées :** Celles-ci comprennent un ensemble prédéfini de questions qui sont posées à tous les clients. Elles permettent de collecter des données cohérentes et comparables.
2. **Les entrevues non structurées :** L'intervieweur n'a pas de liste de questions prédéfinies, ce qui permet une conversation plus ouverte et fluide. Cela peut aider à découvrir des perspectives et des idées inattendues.
3. **Les entrevues semi-structurées :** C'est un mélange des deux premières techniques. L'intervieweur a une liste de questions, mais est libre de dévier et de creuser plus profondément sur certaines réponses.
4. **Les groupes de discussion (*focus groups*) :** Ces sessions réunissent plusieurs clients en même temps. Ils sont utiles pour obtenir des informations sur les consensus et les divergences d'opinions parmi un groupe de clients.
5. **Les ateliers ou sessions de remue-méninges (*brainstorming*) :** Ces réunions permettent de recueillir des idées et des besoins de manière collaborative. Les clients sont impliqués activement dans la discussion et la résolution des problèmes.
6. **Le jeu de rôle :** Dans ce type d'entretien, le client est invité à jouer son propre rôle ou un autre rôle dans un scénario donné. Cela peut aider l'équipe de développement à comprendre comment le client interagit avec le système.
7. **Les observations :** Parfois, le meilleur moyen de comprendre comment les clients utilisent un système est simplement de les observer pendant qu'ils l'utilisent.
8. **Le récit (*story-telling*) :** Le client est invité à raconter une "histoire" sur son expérience avec un système ou un service actuel.

Il est important de rappeler que chaque technique a ses propres forces et faiblesses, et le choix de la technique appropriée dépend du contexte de l'entrevue, des objectifs des parties prenantes et de la nature de l'information recherchée.

Analyse et conception architecturale

UML

Qu'est-ce que l'UML ?

L' **UML** (**Unified Modeling Language**), littéralement en français par *Langage de Modélisation Unifié*, est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode normalisée pour visualiser la conception d'un système. Il est largement utilisé dans la modélisation de systèmes et de processus d'entreprise.

Il facilite la communication entre les membres d'une équipe de développement en fournissant une visualisation commune des processus et de l'architecture du système. Étant un langage standard, il est compréhensible par tous les développeurs, quel que soit le langage de programmation utilisé pour mettre en œuvre le système.

L'UML soutient une variété de diagrammes, y compris les diagrammes de classe, d'objets, de composants, de déploiement, d'activités, d'état-transitions et de séquences. Chacun de ces diagrammes offre une perspective différente de la conception du système, et ensemble, ils fournissent un aperçu complet et cohérent du système. Certains de ces diagrammes sont utilisés lors de la *conception architecture*, d'autres lors de la *conception détaillée*.

Qu'est-ce que l'UML n'est pas ?

L'UML n'est pas :

- **Un langage de programmation**: UML est un **langage de modélisation**, pas un langage de programmation.
- **Un processus de développement de logiciels**: Bien que UML puisse être utilisé dans le cadre de différents processus de développement de logiciels, il **ne prescrit pas un processus spécifique**. Des méthodologies comme le *Processus Unifié Rational* (RUP) utilisent UML comme outil, mais UML en lui-même n'est pas un processus.
- **Une garantie de succès du logiciel**: UML est un outil de modélisation qui peut aider les équipes à penser à travers les problèmes et à communiquer des solutions. Cependant, il ne garantit pas que le logiciel développé sera réussi ou sans défaut.
- **Complet**: Bien que UML fournisse une grande variété de diagrammes pour la modélisation de nombreux aspects d'un système, il y a certaines choses qu'il ne peut pas modéliser ou

qu'il ne modélise pas bien. Par exemple, il ne fournit pas une bonne représentation des interfaces utilisateur.

- **Strictement nécessaire**: Pour certains projets, en particulier ceux qui sont plus petits ou moins complexes, utiliser UML peut être plus une question de préférence que de nécessité. Certaines équipes peuvent trouver que l'utilisation d'UML améliore leur processus, tandis que d'autres peuvent réussir sans lui.

Diagrammes pour la conception architecturale

Pour la conception de haut niveau d'un logiciel (aussi appelée architecture logicielle), les diagrammes UML suivants sont particulièrement utiles :

1. **Diagrammes de cas d'utilisation :** Ils sont utilisés pour représenter l'interaction utilisateur-système à un haut niveau en présentant les fonctionnalités clés que le système doit avoir et comment divers utilisateurs ou rôles interagissent avec ces fonctionnalités. PlantUML (https://plantuml.com/en/use-case-diagram)
2. **Diagrammes de composants :** Ils présentent une vue de haut niveau de la façon dont les différents composants ou modules d'un système interagissent entre eux, illustrant la structure générale du système. PlantUML (https://plantuml.com/en/component-diagram)
3. **Diagrammes de déploiement :** Ils montrent la disposition physique et la communication entre le matériel, le logiciel et les connexions réseau. Ils sont essentiels pour la conception de haut niveau, car ils indiquent où les composants logiciels seront déployés. PlantUML (https://plantuml.com/en/deployment-diagram)

Ces diagrammes fournissent une vue abstraite de haut niveau du système en cours de développement et sont essentiels pour comprendre comment le système sera construit et comment il interagira avec ses utilisateurs.

Références

Cours d'UML (https://laurent-audibert.developpez.com/Cours-UML/) PlantUML (https://plantuml.com/en/)

Diagramme de cas d'utilisation

Cours d'UML, chapitre 2 (https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-cas-utilisation#L2)

Différence entre une user story et un cas d'utilisation ?

Une *user story* et un *cas d'utilisation* sont deux techniques utilisées dans le développement logiciel pour comprendre les besoins des utilisateurs. Cependant, ils diffèrent par leur niveau de détail, leur format et leur utilisation.

User story

Une *user story* est une technique utilisée dans le développement Agile pour capturer de manière succincte une exigence logicielle à partir de la perspective de l'utilisateur final. Typiquement, une *user story* a un format concis : "En tant que [utilisateur], je veux [tâche] afin que [bénéfice]". Par exemple : "En tant que client, je veux être en mesure de filtrer les produits par catégorie afin que je puisse plus facilement trouver ce que je cherche".

- Plus axé sur la conversation et la compréhension des besoins des utilisateurs.
- Plus léger et moins formel.
- Fournit un niveau d'abstraction plus élevé.
- Peut manquer de détails techniques.

Cas d'utilisation

Un cas d'utilisation est une technique plus traditionnelle pour capturer les exigences logicielles. Il décrit en détail comment un système doit se comporter en réponse à une interaction spécifique de l'utilisateur ou dans une situation spécifique. Par exemple, un cas d'utilisation pourrait décrire toutes les étapes que l'utilisateur doit suivre pour effectuer un paiement en ligne.

- Plus formel et détaillé.
- Fournit une vue complète de toutes les interactions possibles avec le système.
- Inclut les acteurs, les scénarios possibles, les conditions préalables et postérieures, et les déclencheurs.

- Peut être trop lourd pour certains projets.

En somme, les *user stories* sont généralement plus adaptées au développement Agile rapide, tandis que les cas d'utilisation sont plus adaptés aux méthodologies de développement plus formelles et détaillées. L'utilisation de l'une ou de l'autre dépend de l'approche de développement, des préférences de l'équipe et de la complexité du projet.

Exemple

***User story*:**

```
En tant qu'Utilisateur de la bibliothèque,
Je veux pouvoir rechercher des livres par titre,
Afin de trouver rapidement le livre que je cherche.
```

Cette *user story* fournit une vue de haut niveau de la fonctionnalité souhaitée de l'utilisateur, sans entrer dans les détails de la façon dont cette fonctionnalité sera mise en œuvre.

Cas d'utilisation :

```
Titre : Recherche de livres par titre

Acteurs : Utilisateur de la bibliothèque, Système

Précondition : L'utilisateur est authentifié dans le système.

Déroulement :
1. L'Utilisateur entre dans la section "Recherche de livres".
2. Le Système affiche un champ de recherche.
3. L'Utilisateur entre le titre du livre dans le champ de recherche.
4. Le Système affiche une liste de livres correspondant au titre
recherché.

Postcondition : Une liste de livres correspondant au titre recherché
est affichée à l'utilisateur.

Exception : Si aucun livre ne correspond au titre recherché, le
système affiche un message indiquant "Aucun livre trouvé".
```

Ce cas d'utilisation détaille plus explicitement le comportement attendu du système et l'interaction de l'utilisateur avec celui-ci. Il capture également une condition exceptionnelle à gérer.

Correspondance entre les user stories et les cas d'utilisation

Il n'y a pas nécessairement de correspondance 1 à 1 entre les user stories et les cas d'utilisation. En fait, ils servent des objectifs légèrement différents dans le processus de développement de logiciels.

Un cas d'utilisation est une description détaillée d'une tâche spécifique que l'utilisateur peut accomplir avec le système. Il se focalise sur comment une fonctionnalité particulière sera utilisée pour accomplir un objectif spécifié.

D'autre part, une user story est une description succincte d'une fonctionnalité du point de vue d'un utilisateur final. Elle se focalise sur pourquoi une fonctionnalité donnée est précieuse pour l'utilisateur.

Par conséquent, un cas d'utilisation peut inclure plusieurs user stories. Par exemple, un cas d'utilisation *"Effectuer un achat"* peut inclure plusieurs user stories comme *"Ajouter des articles à un panier"*, *"Entrer les détails de livraison"*, *"Effectuer un paiement"*, etc.

Cependant, dans certains scénarios, il peut y avoir une correspondance 1 à 1 entre les user stories et les cas d'utilisation, mais cela dépend entièrement de la complexité du système et du niveau de détail de la description des fonctionnalités.

Diagrammes de cas d'utilisation

Un diagramme de cas d'utilisation est un type de diagramme comportemental UML (Unified Modeling Language) qui illustre comment un système va être utilisé.

Il détaille les interactions entre le système (représenté par le diagramme) et ses acteurs (les utilisateurs ou autres systèmes) dans le cadre d'un processus. Il met en évidence les différents chemins que les utilisateurs peuvent emprunter lors de l'utilisation d'un système.

Chaque cas d'utilisation représente une fonctionnalité ou un processus spécifique du système. C'est une séquence d'actions, comprenant des variantes, que le système peut effectuer en interagissant avec les acteurs.

Dans un diagramme de cas d'utilisation, les acteurs sont généralement placés à l'extérieur du diagramme, tandis que le système lui-même est représenté par une boîte. Les cas d'utilisation sont représentés à l'intérieur de cette boîte, et les interactions entre les acteurs et le système sont représentées par des lignes reliant les acteurs aux cas d'utilisation correspondants.

Exemple : générateur de mots de passe

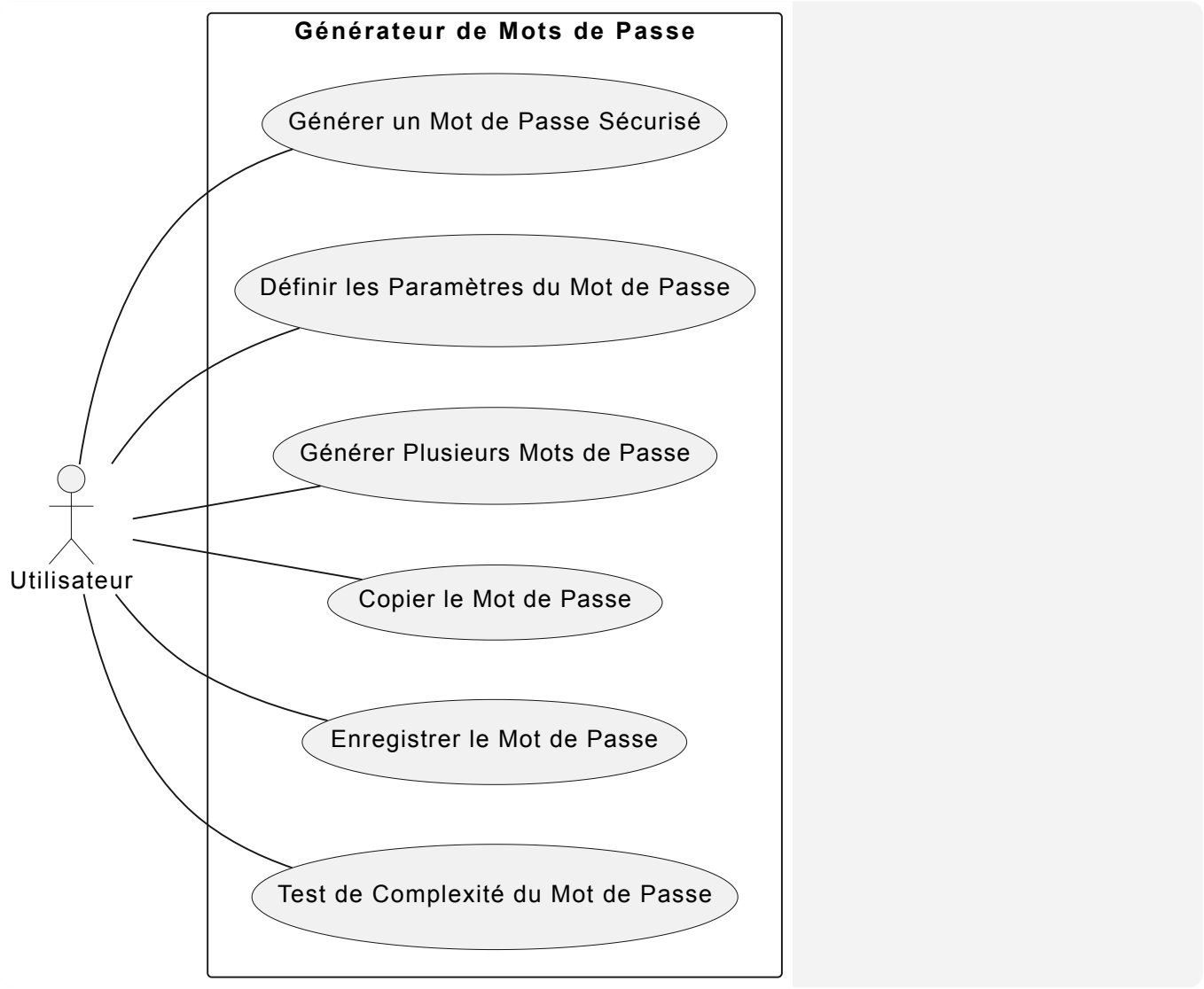
Voici quelques exemples de cas d'utilisation pour un générateur de mots de passe :

1. **Générer un mot de passe sécurisé**
   - Acteur : Utilisateur
   - Description : L'utilisateur demande la génération d'un mot de passe sécurisé. Le système génère et affiche le mot de passe.
2. **Définir les paramètres du mot de passe**
   - Acteur : Utilisateur
   - Description : L'utilisateur définit des paramètres de mot de passe, tels que la longueur, l'inclusion de chiffres, de caractères spéciaux, de majuscules, etc. Le système enregistre ces paramètres et les utilise pour générer des mots de passe.
3. **Générer plusieurs mots de passe**
   - Acteur : Utilisateur
   - Description : L'utilisateur demande la génération de plusieurs mots de passe. Le système génère et affiche la liste des mots de passe.
4. **Copier le mot de passe**
   - Acteur : Utilisateur
   - Description : L'utilisateur souhaite copier le mot de passe généré pour l'utiliser ailleurs. Le système fournit une option de copie qui permet à l'utilisateur de copier le mot de passe dans le presse-papiers.
5. **Enregistrer le mot de passe**
   - Acteur : Utilisateur
   - Description : L'utilisateur veut enregistrer le mot de passe généré pour une utilisation future. Le système fournit une fonction permettant à l'utilisateur d'enregistrer le mot de passe dans une base de données sécurisée.
6. **Test de complexité du mot de passe**
   - Acteur : Utilisateur

- Description : L'utilisateur souhaite tester la complexité du mot de passe généré. Le système fournit une analyser de complexité qui donne un score basé sur la complexité du mot de passe.

Évidemment, ces cas d'utilisation peuvent varier en fonction des fonctionnalités spécifiques de votre générateur de mots de passe.

Diagramme



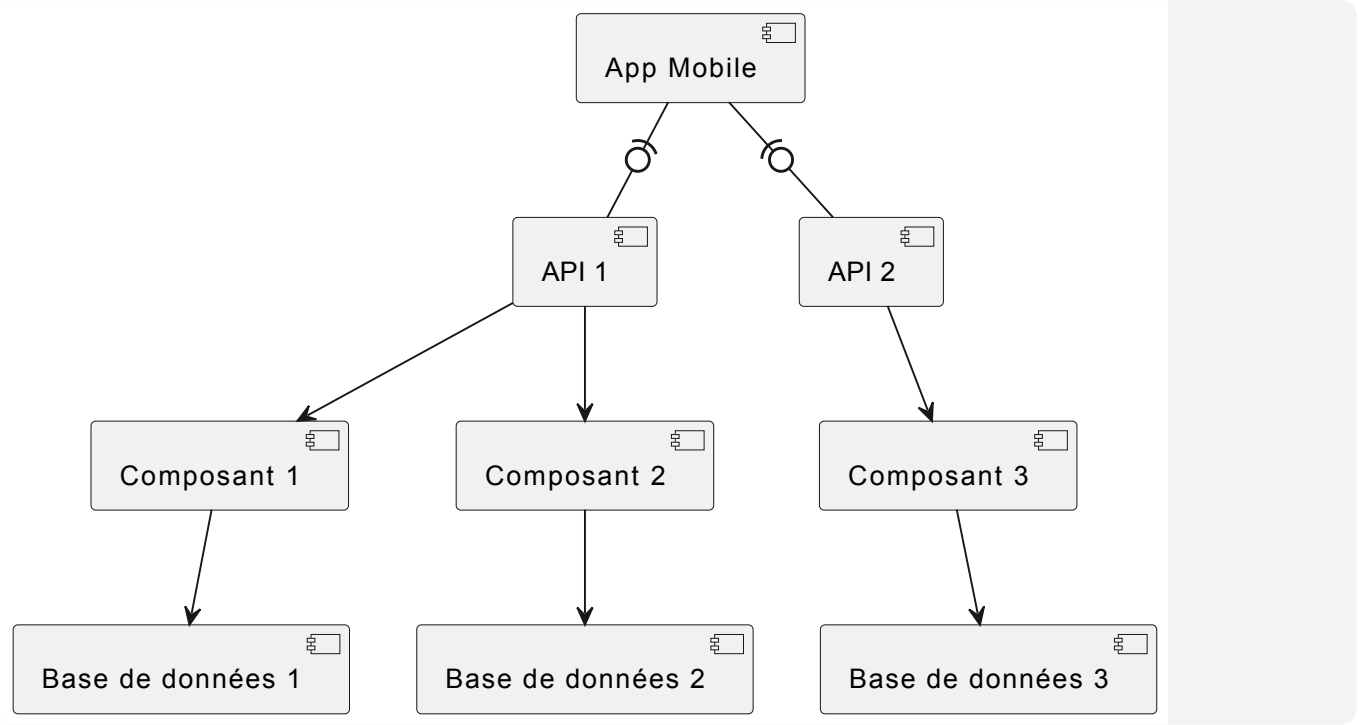
Diagrammes de composants

Un diagramme de composants est un type de diagramme UML (Unified Modeling Language) qui est utilisé pour visualiser l'organisation et les dépendances entre différents composants dans un système. Il fournit une représentation graphique de haut niveau des composants, interfaces et la façon dont ils sont reliés dans un système.

Chaque composant représente une encapsulation indépendante de responsabilité fonctionnelle telle qu'une classe, une interface ou un autre sous-système. Ces composants sont reliés entre eux en utilisant des relations de dépendances binaires qui montrent comment ils coopèrent pour fournir des fonctionnalités au système dans son ensemble.

Le diagramme de composants est généralement utilisé dans le développement de logiciels pour décomposer de grands systèmes en morceaux plus gérables, et pour aider à la planification de la mise en œuvre des systèmes complexes. Il peut également être utilisé pour illustrer l'architecture d'un système existant.

Exemple



Autres exemples (https://creately.com/blog/software-teams/component-diagram-tutorial/)

Diagrammes de déploiement

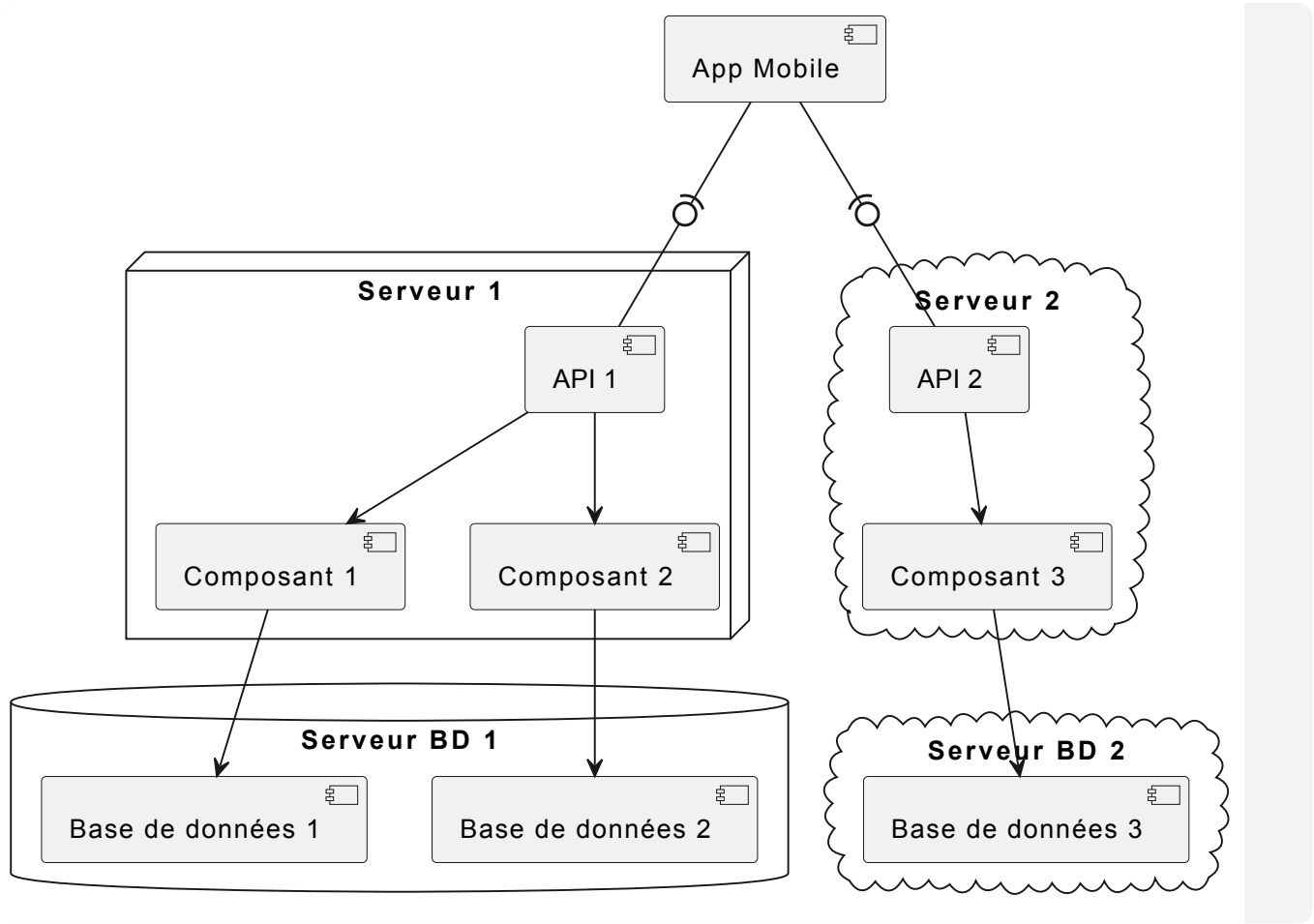
Un diagramme de déploiement est un type de diagramme utilisé en UML qui décrit **l'architecture du matériel** utilisé dans les systèmes et la façon dont **les éléments logiciels sont assignés à ces parties matérielles**. C'est fondamentalement une représentation graphique de la configuration physique et de la distribution du logiciel dans un système.

Dans un diagramme de déploiement, les éléments matériels (comme les serveurs, les machines, les nœuds, les dispositifs, etc.) sont représentés sous forme de nœuds. Les composants du logiciel sont ensuite mappés à ces nœuds pour indiquer où ils sont déployés et exécutés.

Par exemple, un diagramme de déploiement pour une application web pourrait montrer comment l'application est répartie entre un serveur web, un serveur d'application et un serveur de base de données. Il peut également montrer la manière dont ces serveurs sont connectés au réseau et comment ils communiquent entre eux.

Il permet aux architectes et aux développeurs d'avoir une image claire de la manière dont le système sera physiquement déployé et de la façon dont les différents composants du logiciel interagiront entre eux sur cette infrastructure matérielle.

Exemple



Autres exemples (https://creately.com/blog/software-teams/deployment-diagram-templates/)

Différences entre diagrammes de composants et de déploiement

Un **diagramme de composants** et un **diagramme de déploiement** sont deux types de diagrammes utilisés en modélisation UML, mais ils servent à différents objectifs et représentent différents aspects d'un système.

Diagramme de composants

Un diagramme de composants est principalement axé sur le système logiciel et ses composants. Il sert à visualiser, spécifier et documenter les éléments logiciels de haut niveau d'un système, leurs interrelations et leurs dépendances. Les composants peuvent être des parties de code (comme les classes ou les modules), des bases de données, des interfaces utilisateur, des composants tiers, etc. Une relation entre composants pourrait être une dépendance (un composant a besoin d'un autre pour fonctionner correctement) ou une interaction (un composant envoie/reçoit des données à un autre).

Diagramme de déploiement

Un diagramme de déploiement se concentre sur l'aspect matériel et sur la façon dont le logiciel est déployé sur le matériel. Il montre le **où** du système. Il illustre la configuration physique de l'infrastructure matérielle (nœuds), ainsi que la façon dont les composants du logiciel sont répartis sur ces nœuds et comment ils interagissent entre eux. Les nœuds peuvent être des serveurs, des ordinateurs, des terminaux mobiles, des appareils IoT, etc, et leurs relations peuvent comprendre des aspects comme le réseau, la communication, le protocole utilisé, etc.

En conclusion, alors que le diagramme de composants décrit le 'quoi' (les composants) et le 'comment' (leurs relations) du système, le diagramme de déploiement décrit le **où** (la disposition des composants logiciels sur l'infrastructure matérielle) et le **comment** (comment ils interagissent à ce niveau).