

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
Exemples	68
Diagrammes de composants	79

Diagrammes de déploiement	81
Différences entre diagrammes de composants et de déploiement	83
Diagrammes entité-association	84
Diagrammes du domaine	86
Diagrammes d'activités	89
Architectures matérielles et logicielles	94
Architecture Monolithique	97
Architecture en couches	98
Architecture orientée services	99
Architecture microservices	100
Architecture événementielle	101
Architecture pipeline de données	102
Architecture pair-à-pair	103
Architecture MVC	104
Diagrammes de package	105
Diagrammes de classes	107
Prototypage	108
Maquettes (ou prototypes)	109
Conception détaillée	112
Modèles de données	113
Normalisation	115
Dénormalisation	119
Exemple : ORDBMS	121
Diagrammes d'état	137
Diagrammes de séquences	141
Patrons de conception	144
Principes OO	145
Design Patterns : Première Partie	158
Design Patterns : Deuxième Partie	161
Design Patterns : Structuraux	165
Design Patterns : Comportementaux (*Behavioral*)	170

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.

Dépôt Git contenant les fichiers sources de ce site

(https://github.com/profdenis/analyse_4P1)

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 incrément, 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

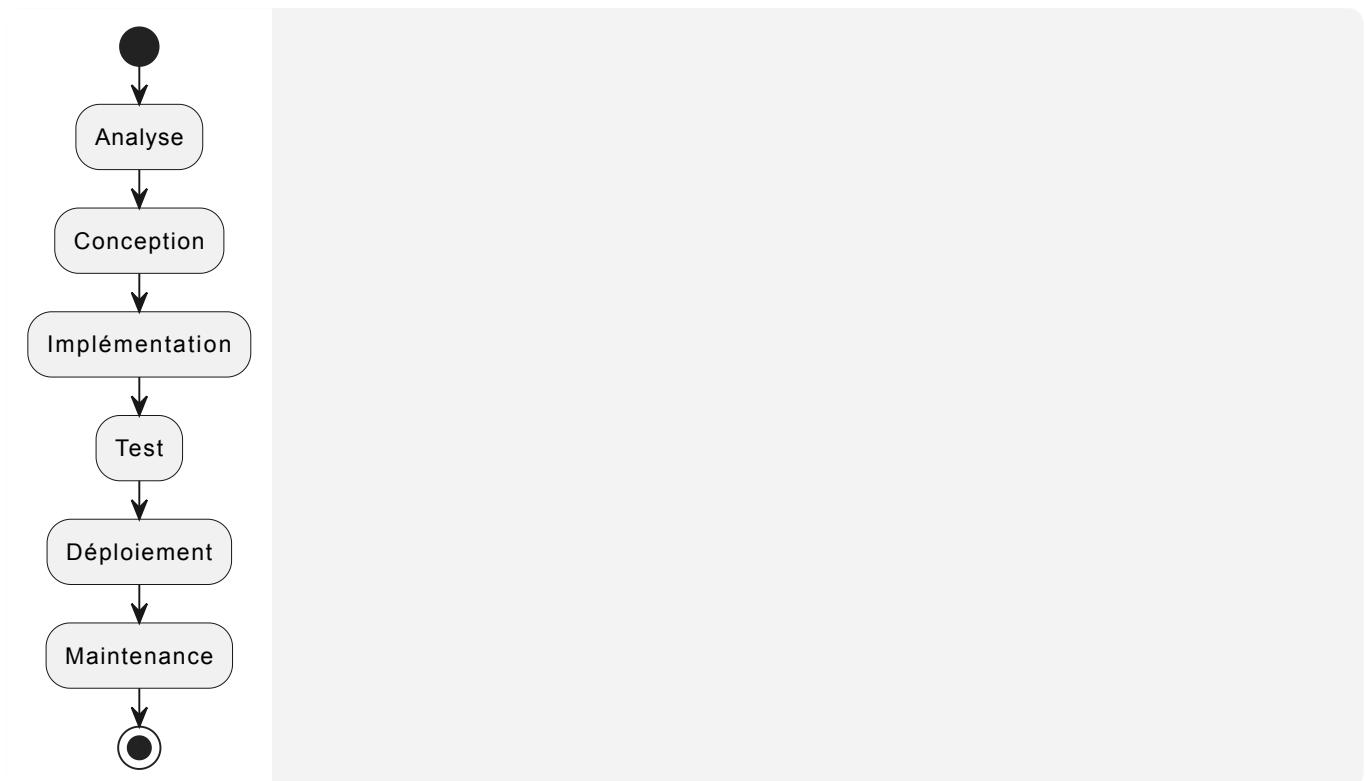
Il existe plusieurs approches pour le développement de logiciels. Voici les plus courants :

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.

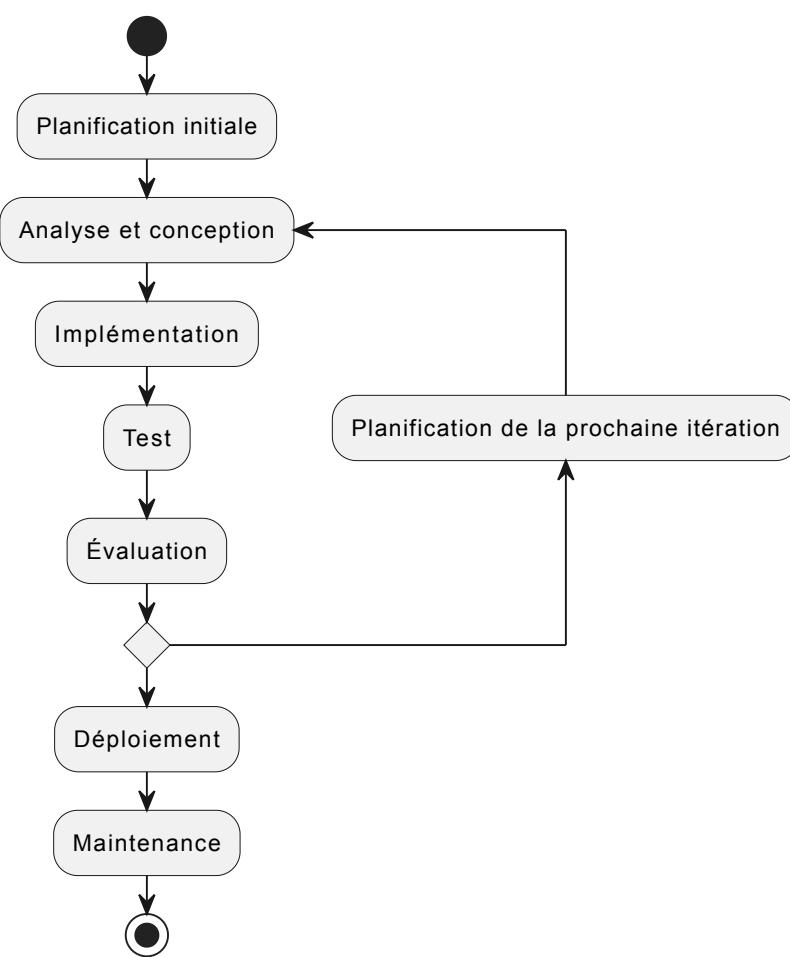
Phase de déploiement : Une fois que tout le code a été testé et que le logiciel a été accepté par les clients ou les utilisateurs, il est prêt à être déployé. Cela signifie que le logiciel est mis en ligne et mis à la disposition des utilisateurs.

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 incrémentums.
- 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 manière consistante.
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 consiste 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 commettent 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, spécifier 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 fournit 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

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.

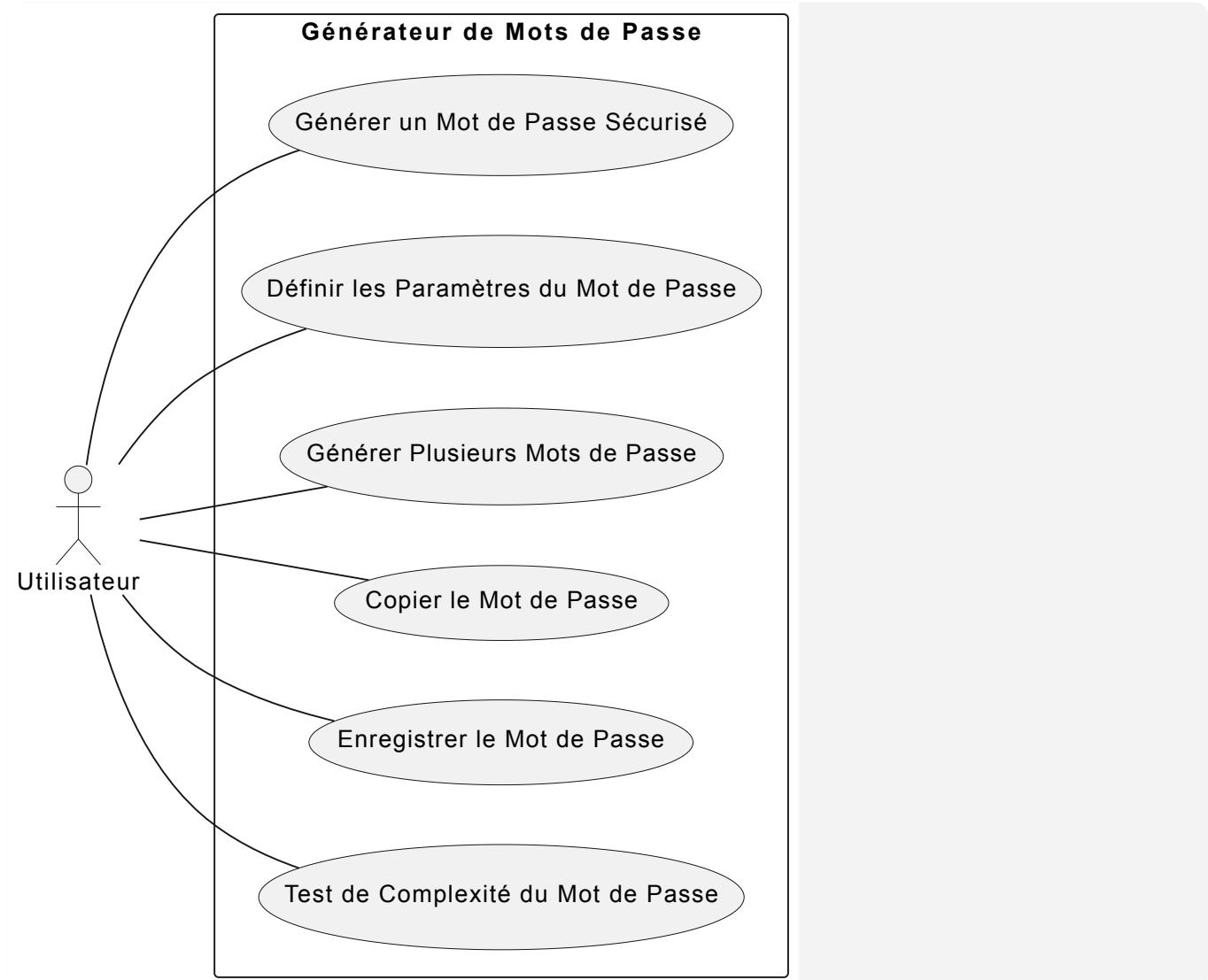
Références

1. Cours d'UML Chapitre 2 (<https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-cas-utilisation#L2>)
2. PlantUML (<https://plantuml.com/en/use-case-diagram>)

Exemples

Générateur de mots de passe

Diagramme



Cas d'utilisation (version courte)

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.

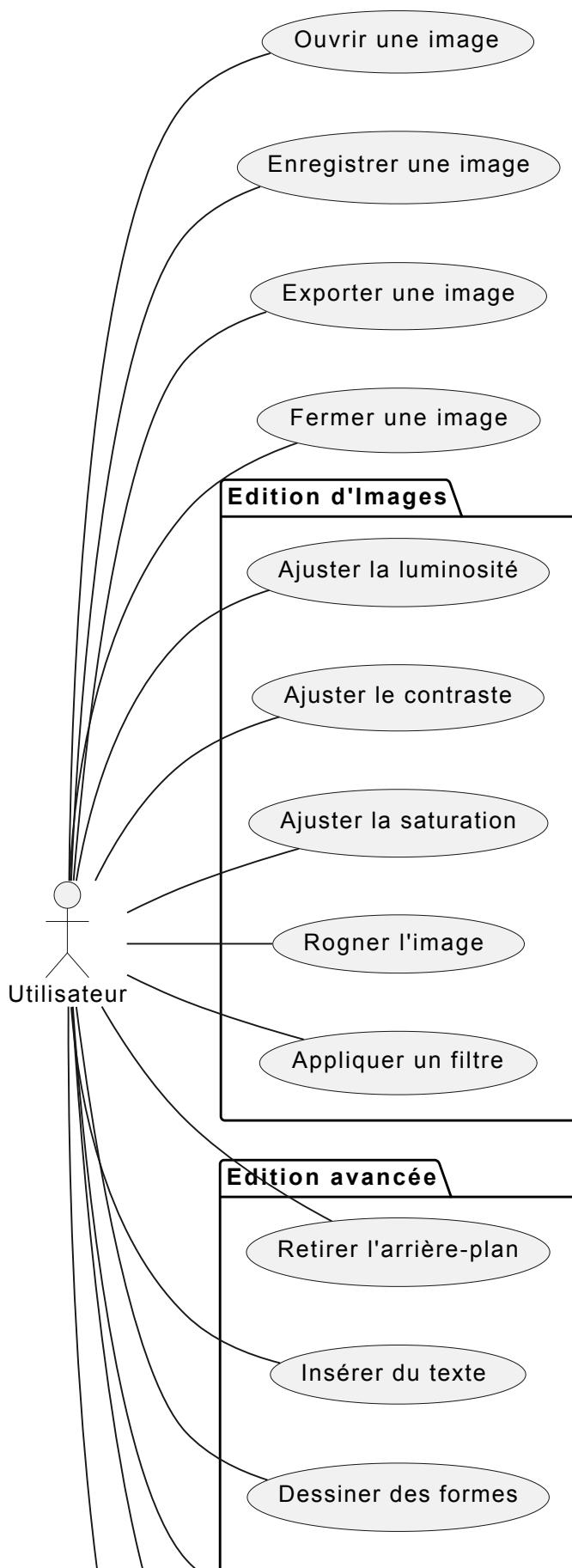
6. Test de complexité du mot de passe

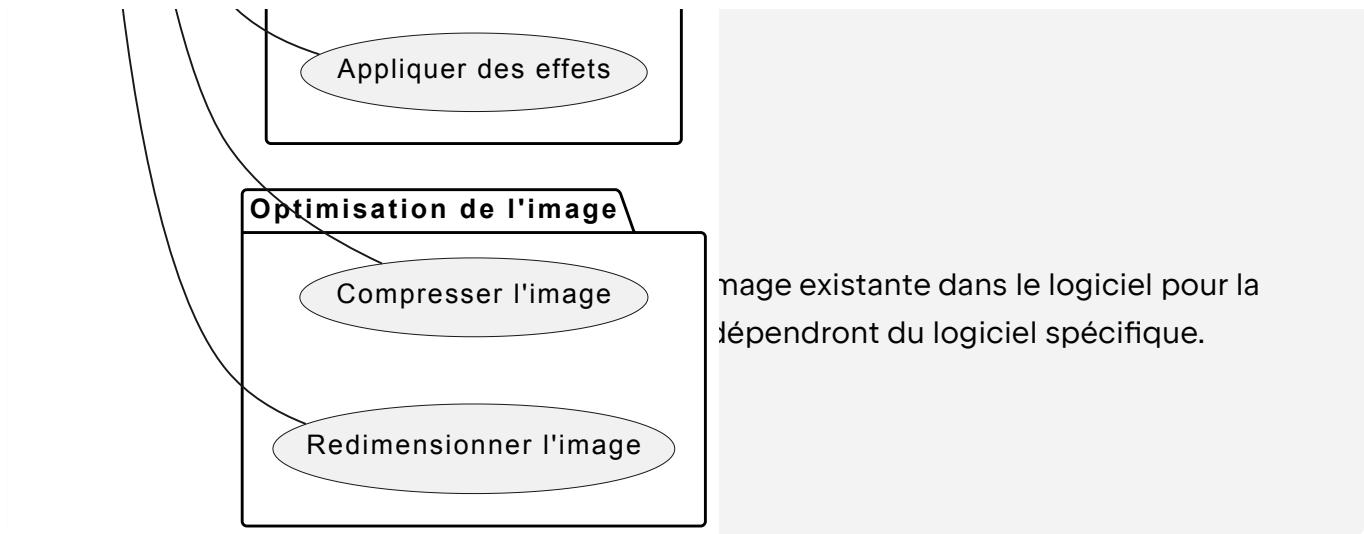
- Acteur : Utilisateur
- Description : L'utilisateur souhaite tester la complexité du mot de passe généré. Le système fournit une analyse de complexité qui donne un score basé sur la complexité du mot de passe.

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

Logiciel de manipulation d'images

Diagramme





- Suite à des modifications sur une image, l'utilisateur a la possibilité d'enregistrer ces changements. Cette action mettra à jour l'image originale avec les modifications réalisées.

3. Exporter une image

- Acteur : Utilisateur
- L'utilisateur peut choisir d'exporter l'image modifiée. Cela permet d'enregistrer les modifications apportées sans altérer l'image originale. L'image peut être exportée sous un nouveau nom de fichier ou dans un format différent.

4. Fermer une image

- Acteur : Utilisateur
- L'utilisateur peut fermer l'image qu'il a ouverte dans le logiciel. Cette action ne quitte pas le programme, seulement la visualisation et la manipulation de l'image actuelle.

5. Ajuster la luminosité

- Acteur : Utilisateur
- L'utilisateur peut modifier la luminosité de l'image en utilisant les outils appropriés du logiciel, pour la rendre plus claire ou plus sombre.

6. Ajuster le contraste

- Acteur : Utilisateur
- L'utilisateur a la possibilité d'augmenter ou de diminuer le contraste de l'image, améliorant ainsi la distinction entre les zones d'ombres et de lumière.

7. Ajuster la saturation

- Acteur : Utilisateur
- L'utilisateur peut augmenter ou diminuer la saturation pour intensifier ou atténuer les couleurs.

8. Rogner l'image

- Acteur : Utilisateur
- L'utilisateur peut couper une partie de l'image pour enlever des zones indésirables ou changer la taille de l'image.

9. Appliquer un filtre

- Acteur : Utilisateur
- Divers filtres peuvent être appliqués par l'utilisateur pour modifier l'apparence générale de l'image, tels que des filtres vintage, noir et blanc, sépia, etc.

10. Retirer l'arrière-plan

- Acteur : Utilisateur
- Dans certains cas, l'utilisateur peut avoir besoin de supprimer l'arrière-plan de l'image. Certains logiciels offrent des outils dédiés pour cette tâche.

11. Insérer du texte

- Acteur : Utilisateur
- L'utilisateur peut ajouter du texte à l'image, le personnaliser en changeant la police, la couleur, la taille, l'orientation, etc.

12. Dessiner des formes

- Acteur : Utilisateur
- L'utilisateur peut dessiner différentes formes (cercles, carrés, lignes, etc.) sur l'image.

13. Appliquer des effets

- Acteur : Utilisateur

- L'utilisateur peut appliquer divers effets spéciaux à l'image pour lui donner un aspect créatif ou artistique.

14. Compresser l'image

- Acteur : Utilisateur
- Pour réduire la taille du fichier de l'image pour un stockage ou un partage plus facile, l'utilisateur peut utiliser une fonction de compression.

15. Redimensionner l'image

- Acteur : Utilisateur
- L'utilisateur peut changer la largeur et la hauteur de l'image selon ses besoins.

Cas d'utilisation (version longue)

1. Cas d'utilisation : Ouvrir une image

- **Objectif:** Charger une image dans le logiciel pour effectuer des manipulations ultérieures.
- **Acteur(s):** Utilisateur
- **Préconditions:** Le logiciel est ouvert et l'image à ouvrir est disponible sur le dispositif de stockage
- **Scénario:** L'utilisateur navigue vers le dossier contenant l'image, sélectionne le fichier d'image et lance l'opération d'ouverture.
- **Postconditions:** L'image est chargée dans le logiciel et prête pour des manipulations.

2. Cas d'utilisation : Enregistrer une image

- **Objectif:** Sauvegarder les modifications effectuées sur l'image.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image a été ouverte et modifiée dans le logiciel.
- **Scénario:** Après avoir réalisé des modifications sur l'image, l'utilisateur lance l'opération de sauvegarde.
- **Postconditions:** Les modifications de l'image sont enregistrées et le fichier d'image

original est mis à jour.

3. Cas d'utilisation : Exporter une image

- **Objectif:** Sauvegarder l'image modifiée dans un nouveau fichier ou un format différent, en conservant l'original intact.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image a été ouverte et modifiée dans le logiciel.
- **Scénario:** L'utilisateur choisit l'option d'exportation, sélectionne le format de l'image et le répertoire de sauvegarde, puis lance l'opération d'exportation.
- **Postconditions:** L'image modifiée est enregistrée sous un nouveau nom de fichier ou dans un nouveau format, l'image originale restant inchangée.

4. Cas d'utilisation : Fermer une image

- **Objectif:** Fermer l'image actuellement ouverte dans le logiciel.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est actuellement ouverte dans le logiciel.
- **Scénario:** L'utilisateur choisit l'option pour fermer l'image.
- **Postconditions:** L'image est fermée, libérant ainsi les ressources utilisées.

5. Cas d'utilisation : Ajuster la luminosité

- **Objectif:** Modifier la luminosité de l'image pour améliorer sa qualité visuelle.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur utilise l'option d'ajustement de la luminosité, sélectionne le niveau souhaité de luminosité et applique le changement.
- **Postconditions:** La luminosité de l'image est modifiée en fonction des ajustements de l'utilisateur.

6. Cas d'utilisation : Ajuster le contraste

- **Objectif:** Modifier le contraste de l'image pour améliorer sa qualité visuelle.

- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur utilise l'option d'ajustement du contraste, sélectionne le niveau souhaité de contraste et applique le changement.
- **Postconditions:** Le contraste de l'image est modifié en fonction des ajustements de l'utilisateur.

7. Cas d'utilisation : Ajuster la saturation

- **Objectif:** Modifier la saturation de l'image pour améliorer sa qualité visuelle ou créer un effet spécifique.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur utilise l'option d'ajustement de la saturation, sélectionne le niveau souhaité de saturation et applique le changement.
- **Postconditions:** La saturation de l'image est modifiée en fonction des ajustements de l'utilisateur.

8. Cas d'utilisation : Rogner l'image

- **Objectif:** Extraire une partie souhaitée de l'image ou supprimer les parties indésirables.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur sélectionne l'outil de rognage, spécifie la zone de l'image à conserver et applique le changement.
- **Postconditions:** Seule la partie sélectionnée de l'image est conservée et sa taille est réduite en conséquence.

9. Cas d'utilisation : Appliquer un filtre

- **Objectif:** Appliquer un filtre prédéfini à l'image pour créer un effet spécifique.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.

- **Scénario:** L'utilisateur sélectionne l'option de filtre, choisit un filtre parmi ceux disponibles et applique le filtre à l'image.
- **Postconditions:** L'image est modifiée pour refléter l'effet du filtre choisi.

10. Cas d'utilisation : Retirer l'arrière-plan

- **Objectif:** Séparer le sujet de l'image de son arrière-plan.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur utilise l'option de suppression de l'arrière-plan et sélectionne le sujet à conserver. Le logiciel supprime l'arrière-plan.
- **Postconditions:** Seul le sujet de l'image est conservé; l'arrière-plan est supprimé.

11. Cas d'utilisation : Insérer du texte

- **Objectif:** Ajouter des légendes, des titres ou d'autres textes à l'image.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur sélectionne l'outil d'ajout de texte, entre le texte souhaité, le positionne sur l'image et applique le changement.
- **Postconditions:** Le texte est ajouté à l'image à l'emplacement spécifié.

12. Cas d'utilisation : Dessiner des formes

- **Objectif:** Ajouter des formes géométriques à l'image.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur sélectionne l'outil de dessin de forme, choisit la forme souhaitée, la place sur l'image et applique le changement.
- **Postconditions:** La forme est ajoutée à l'image à l'emplacement spécifié.

13. Cas d'utilisation : Appliquer des effets

- **Objectif:** Appliquer des effets spéciaux à l'image.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur sélectionne l'option d'effets, choisit un effet parmi ceux disponibles et applique l'effet à l'image.
- **Postconditions:** L'effet choisi est appliqué à l'image, modifiant son apparence en conséquence.

14. Cas d'utilisation : Compresser l'image

- **Objectif:** Réduire la taille du fichier de l'image pour économiser de l'espace de stockage ou pour faciliter le partage.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur sélectionne l'option de compression, définit le niveau de compression souhaité et applique le changement.
- **Postconditions:** La taille du fichier de l'image est réduite en fonction du niveau de compression spécifié.

15. Cas d'utilisation : Redimensionner l'image

- **Objectif:** Modifier les dimensions de l'image.
- **Acteur(s):** Utilisateur
- **Préconditions:** Une image est ouverte dans le logiciel pour l'édition.
- **Scénario:** L'utilisateur sélectionne l'option de redimensionnement, entre les nouvelles dimensions souhaitées et applique le changement.
- **Postconditions:** Les dimensions de l'image sont modifiées pour correspondre aux valeurs entrées par l'utilisateur.

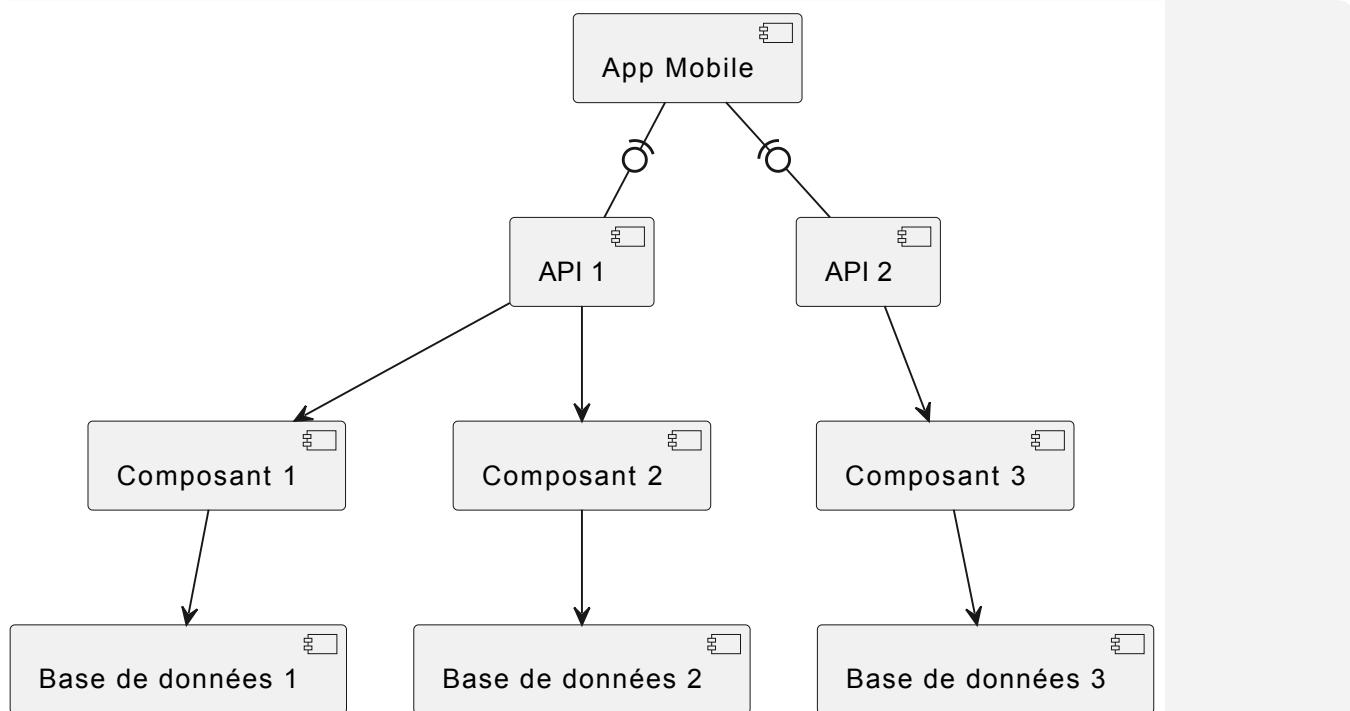
Diagrammes de composants

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/>)

Références

1. Cours d'UML Chapitre 8 (<https://laurent-audibert.developpez.com/Cours-UML/?page=diagrammes-composants-deploiement>)
2. PlantUML (<https://plantuml.com/en/component-diagram>)

Diagrammes de déploiement

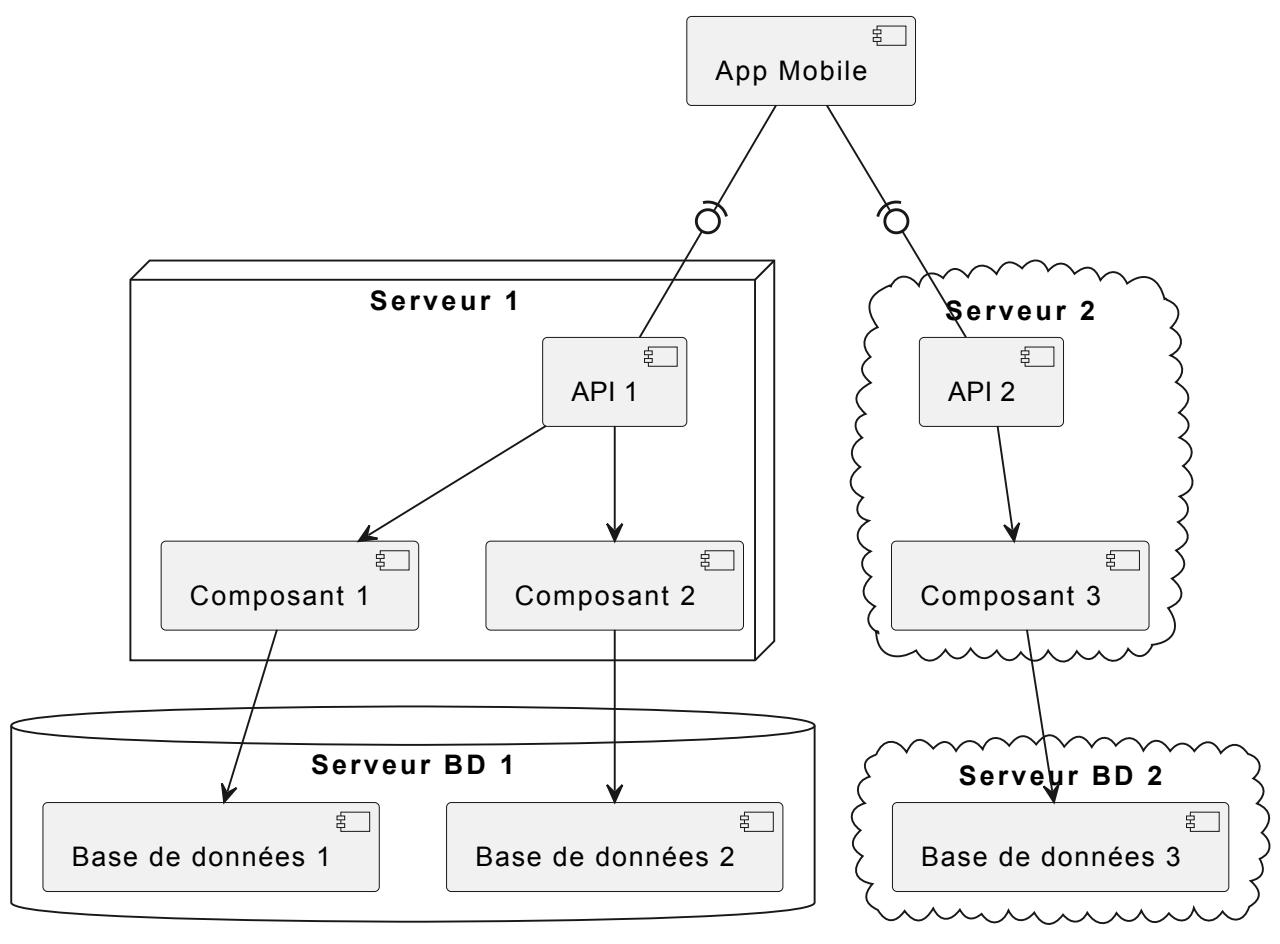
Un diagramme de déploiement est un type de diagramme utilisé en UML qui décrit l'**architecture du matériel** utilisé dans les systèmes et la façon dont les éléments logiciels sont assignés à ces parties matérielles. C'est fondamentalement une représentation graphique de la configuration physique et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

Références

1. Cours d'UML Chapitre 8 (<https://laurent-audibert.developpez.com/Cours-UML/?page=diagrammes-composants-deploiement>)
2. PlantUML (<https://plantuml.com/en/deployment-diagram>)

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).

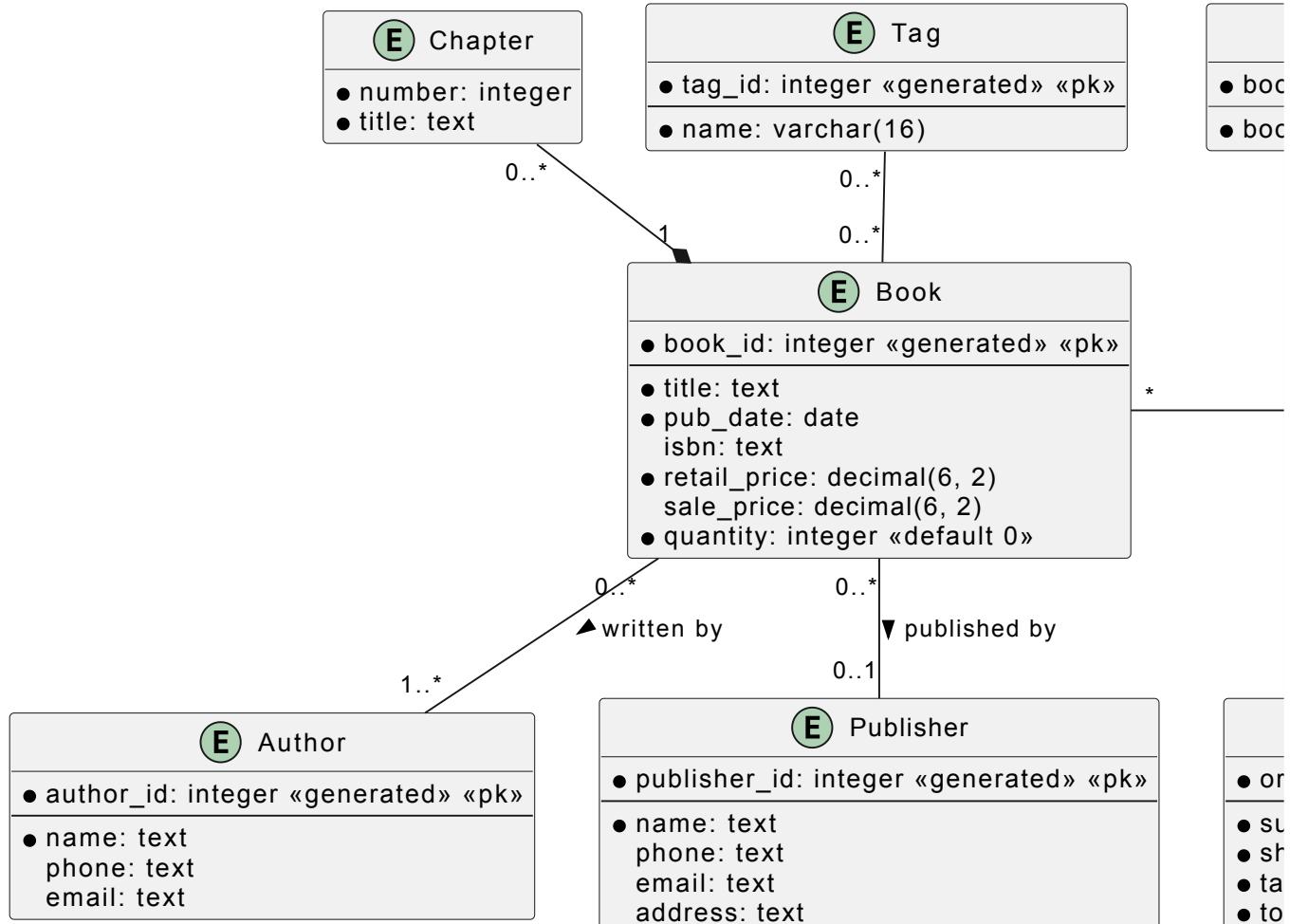
Diagramme de déploiement

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.

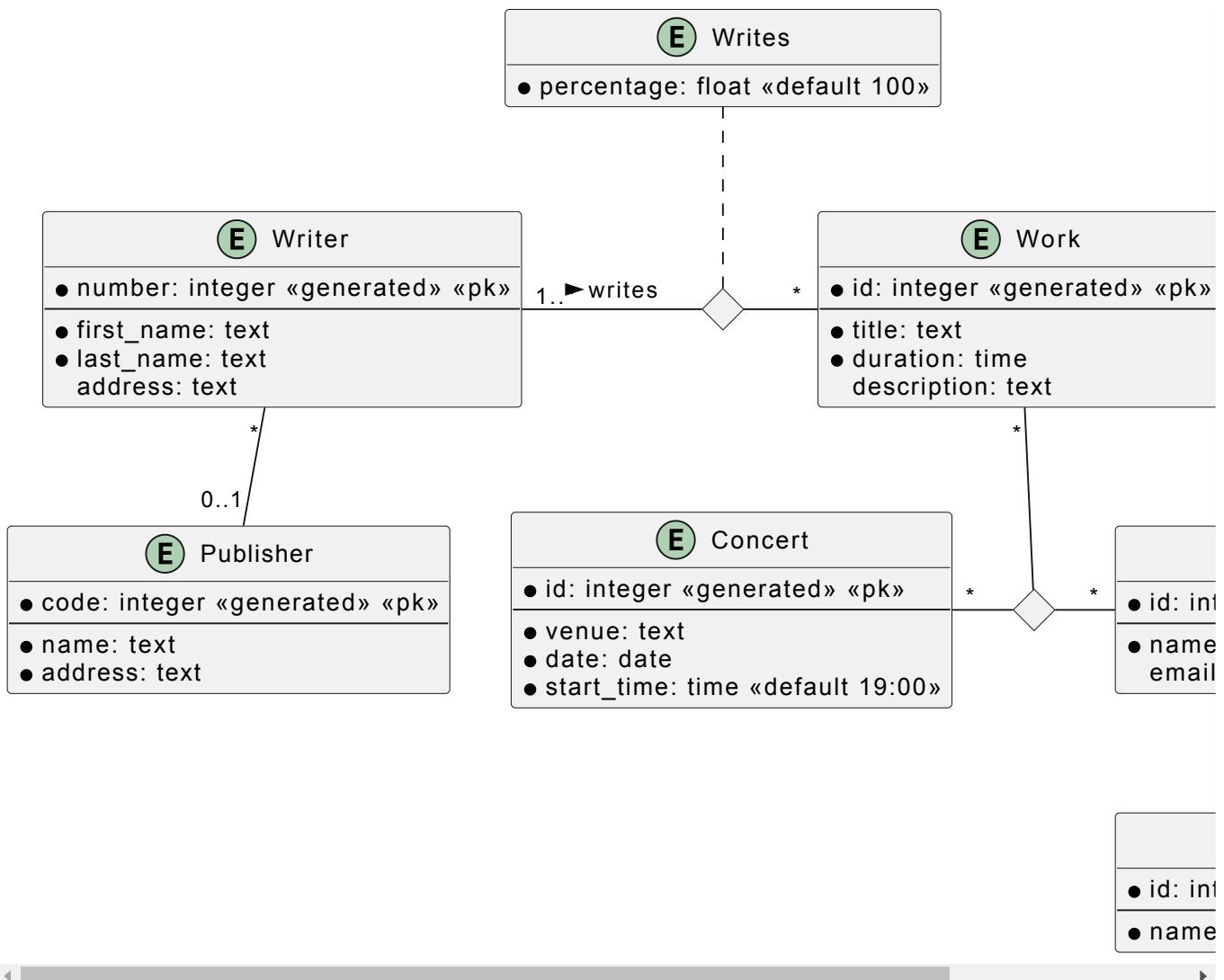
En conclusion, alors que le diagramme de composants décrit le 'quoi' (les composants) et le 'comment' (leurs relations) du système, le diagramme de déploiement décrit le **où** (la disposition des composants logiciels sur l'infrastructure matérielle) et le **comment** (comment ils interagissent à ce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

Diagrammes entité-association

Exemple 1 : Livres



Exemple 2 : Musique



Références

1. PlantUML (<https://plantuml.com/en/ie-diagram>)

Diagrammes du domaine

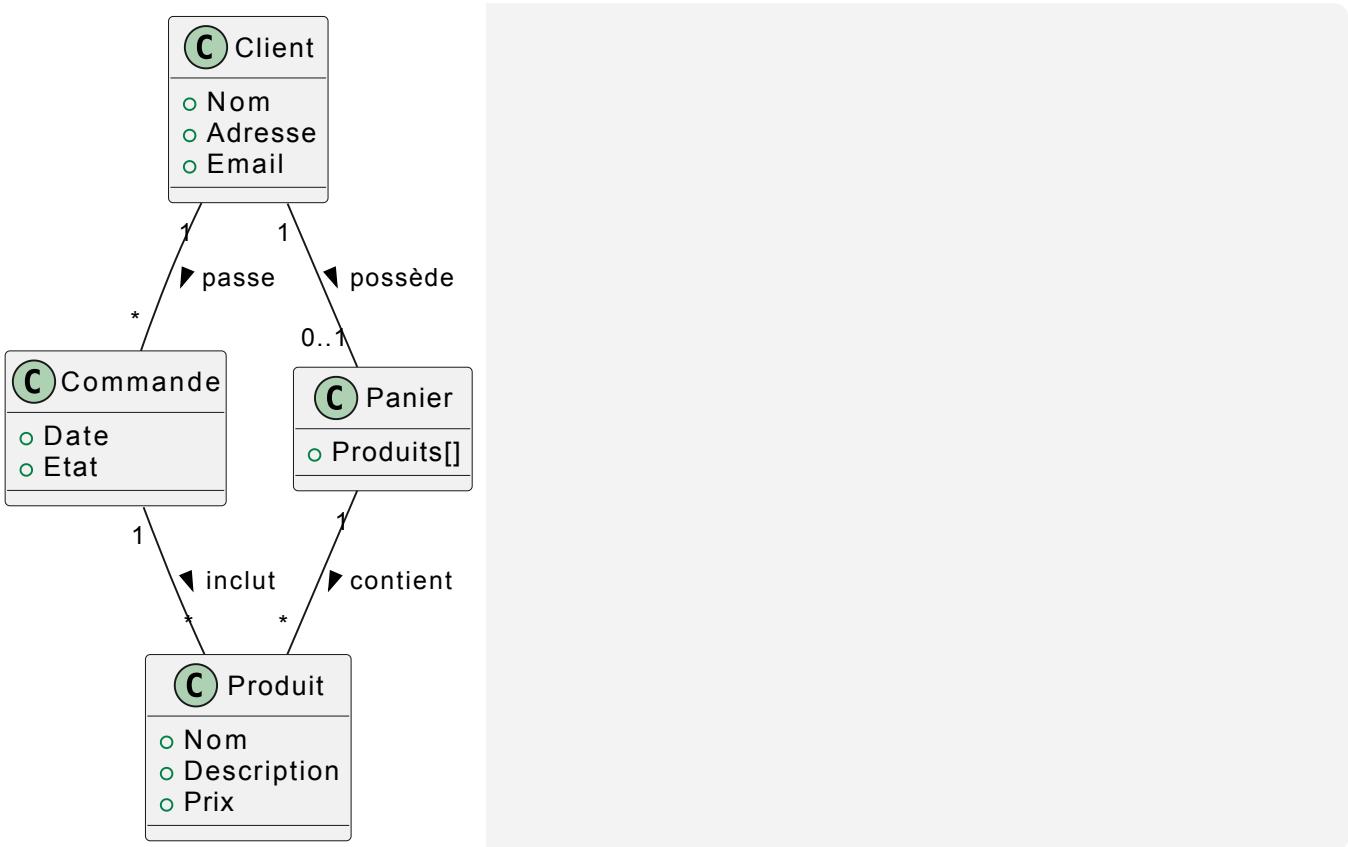
Un diagramme de domaine UML, aussi connu sous le nom de *modèle de domaine*, de *diagramme de classe conceptuelle* ou *diagramme d'analyse orientée objet*, représente le vocabulaire et les concepts clés pertinents pour un domaine spécifique. Il sert principalement à modéliser les concepts de haut niveau dans le domaine du problème et leurs relations. Contrairement aux diagrammes de classes techniques, il ne décrit pas comment le système est implémenté.

Un diagramme de domaine UML englobe généralement les éléments suivants :

- **Classes de domaine:** Ces classes représentent des concepts clés dans le domaine du problème. Par exemple, dans le domaine du commerce électronique, nous pourrions avoir des classes de domaine comme "Produit", "Client", "Commande", etc.
- **Associations:** Les associations sont des liens qui connectent les classes de domaine et illustrent la nature de leur relation. Par exemple, une relation "identifiée par" peut exister entre la classe "Client" et la classe "IdentifiantClient".
- **Attributs:** Les attributs sont des propriétés définies dans les classes qui décrivent les caractéristiques des concepts du domaine. Par exemple, la classe "Produit" pourrait avoir des attributs comme "nom", "prix", "description", etc.

Les diagrammes de domaine sont très utiles tout au long du processus de développement de logiciel, pour comprendre les requis de haut niveau, pour communiquer avec les parties prenantes non techniques, et comme base pour des modèles plus détaillés comme les diagrammes de classes.

Exemple : Site web de commerce en ligne



Lien entre un diagramme de domaine et un diagramme entité-association

Un diagramme de domaine UML et un diagramme Entité-Association (EA) sont deux outils de modélisation de données qui visent à représenter les données et leurs relations d'une manière conceptuelle et abstraite. Ces deux types de diagrammes sont utilisés pour analyser et définir les besoins en termes de stockage et de manipulation des données pour un système. Cependant, ils diffèrent par plusieurs aspects :

- Focus:** Un diagramme de domaine UML se concentre sur les "objets" du monde réel (ou les concepts du domaine), indépendamment de la manière dont ils sont stockés ou gérés par le système. Par contre, un diagramme EA se concentre sur les "entités" qui doivent être stockées en tant que données, ainsi que sur leur structure et leurs relations.
- Associations:** Dans un diagramme de domaine UML, les associations entre les classes représentent des liens conceptuels et logiques entre les objets du monde réel. Dans un diagramme EA, les relations représentent comment les différentes entités sont liées dans la base de données.
- Utilisation:** Les diagrammes de domaine UML sont souvent utilisés lors de la phase de conception initiale du projet pour aider à comprendre les besoins et les concepts du domaine du problème. Les diagrammes EA, quant à eux, sont principalement utilisés lors

de la conception et de la modélisation de bases de données. Lors de la conception d'un système ayant une base de données en son centre, il est possible de développer un diagramme entité-association avant les diagrammes de domaine et les diagrammes de classe.

En résumé, un diagramme de domaine UML est généralement plus abstrait et orienté vers l'utilisateur, tandis qu'un diagramme EA est plus technique et orienté vers la conception de bases de données. Les deux types de diagrammes peuvent se compléter dans un projet de développement de logiciel.

Diagrammes d'activités

Un diagramme d'activités UML est un type de **diagramme comportemental** qui visualise le workflow d'activités parmi les composants d'un système. Il illustre le déroulement d'une opération et montre comment l'opération est séparée en actions qui se produisent en parallèle ou en série. Le diagramme sert à modéliser le contrôle du flux dans un système en mettant l'accent sur la séquence et les conditions d'exécution.

Vous pouvez utiliser un diagramme d'activités UML lorsque vous voulez :

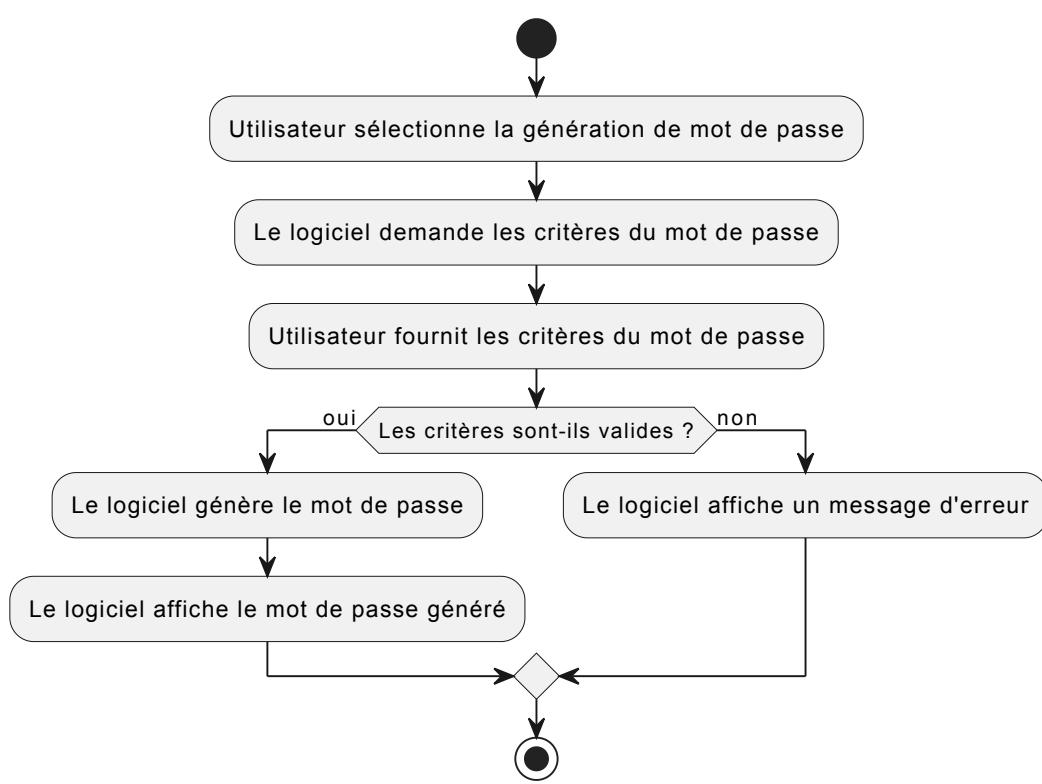
1. Modéliser le flux de contrôle d'un système.
2. Capturez les processus d'affaires et les flux de travail.
3. Visualisez les étapes d'une opération ou d'un processus.
4. Analysez les opérations existantes et trouvez des moyens d'améliorer ces opérations.
5. Représenter des algorithmes graphiquement

Voici comment vous pouvez créer un diagramme d'activités UML :

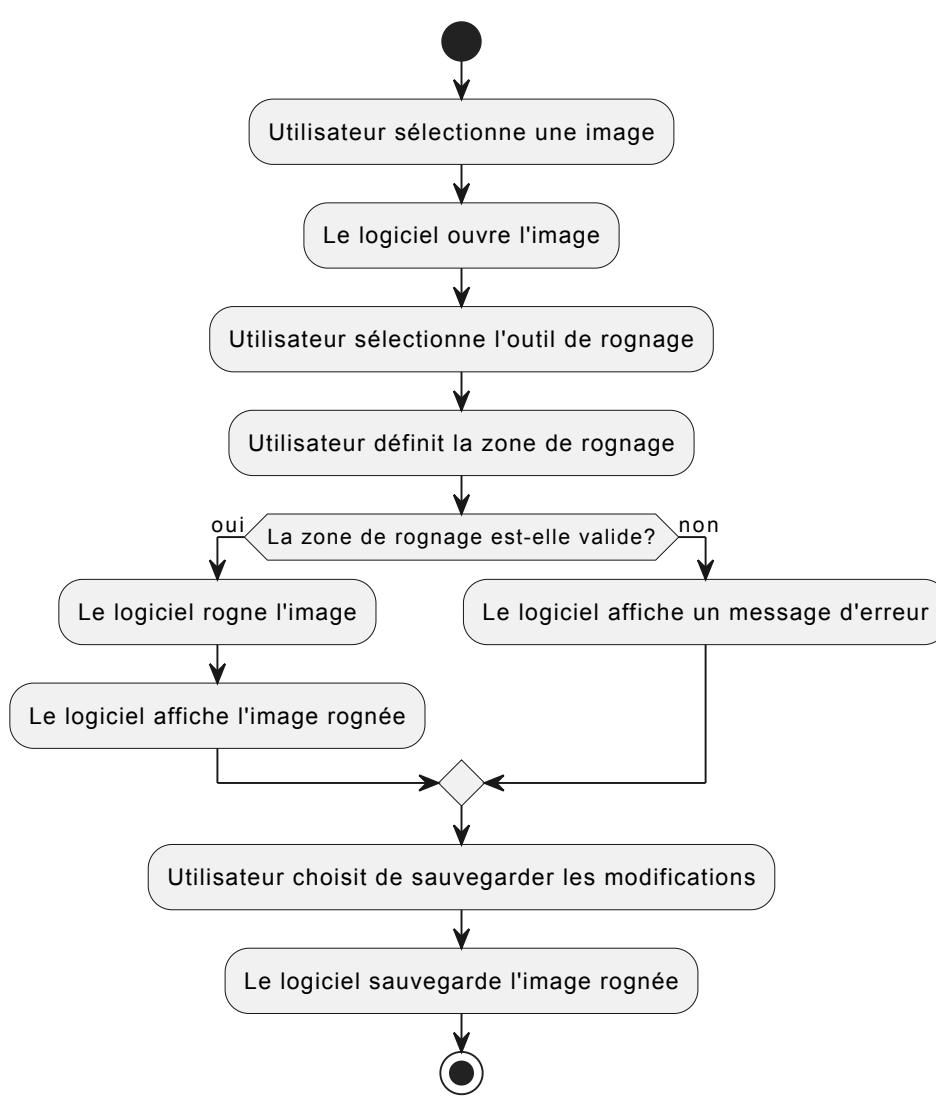
1. **Définition des activités:** Commencez par identifier toutes les tâches ou activités qui seront effectuées dans la procédure que vous modélisez.
2. **Séquence des activités:** Après avoir identifié les activités, il faut déterminer leur ordre d'exécution.
3. **Ajout des flèches de contrôle de flux:** Des flèches sont ajoutées entre les activités pour montrer la direction dans laquelle l'opération se déplace.
4. **Ajout des diamants de décision:** Ajoutez un diamant chaque fois que le chemin du flux de contrôle peut bifurquer en fonction d'une décision.
5. **Examen et mise à jour:** Une fois le diagramme initial créé, il faut le revoir et le mettre à jour pour s'assurer qu'il représente avec précision le processus.

Notez que les diagrammes d'activités UML sont particulièrement utiles pour les systèmes de logiciels, mais ils peuvent également être utilisés pour comprendre les processus d'affaires et d'autres systèmes non-logiciels.

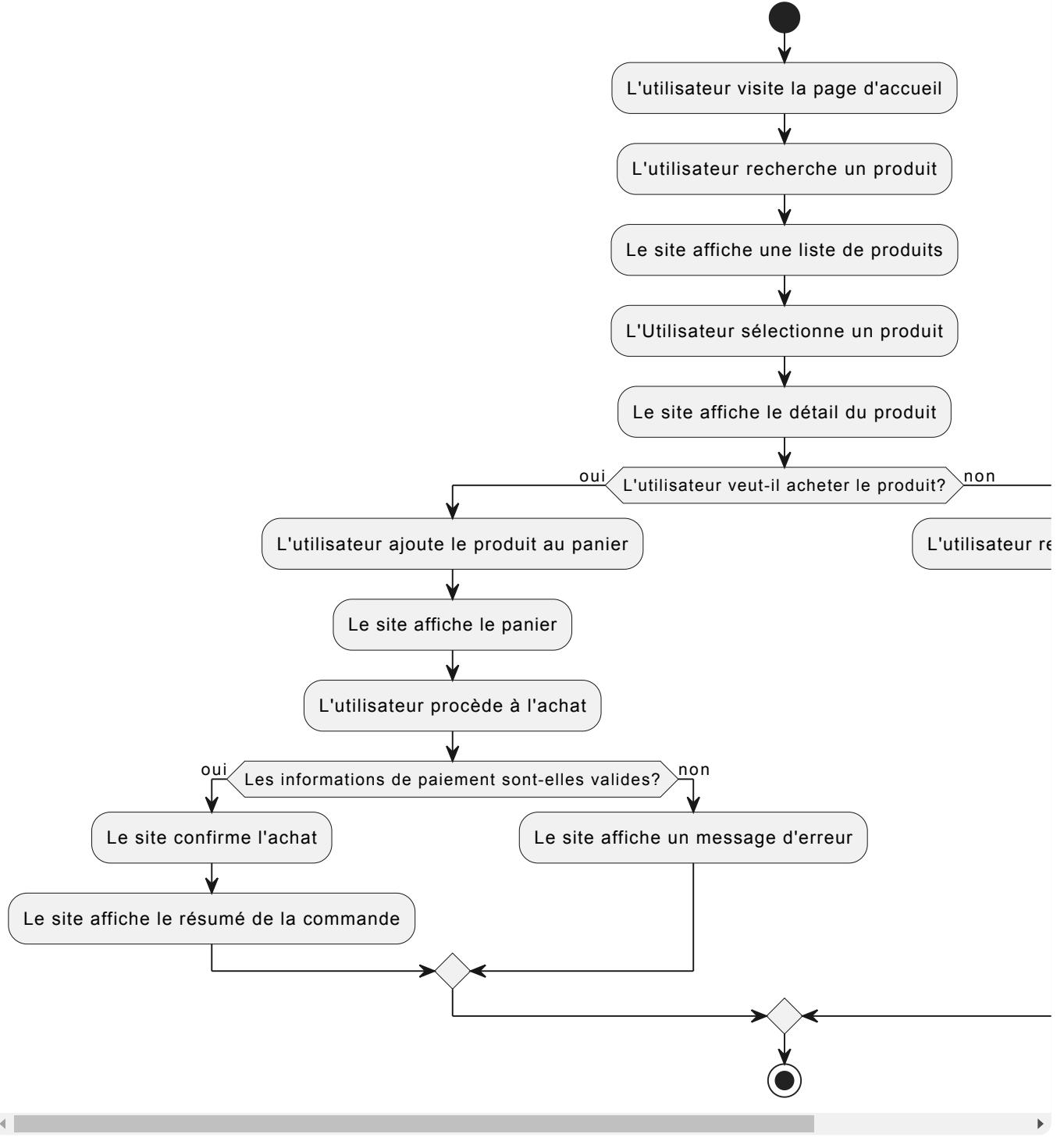
Exemple 1 : Génération de mots de passe



Exemple 2 : Rogner une image



Exemple 3 : Achat d'un produit sur un site web



Références

1. Cours d'UML Chapitre 6 (<https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-activites>)

2. PlantUML (<https://plantuml.com/en/activity-diagram-beta>)

Architectures matérielles et logicielles

Architecture matérielle

Lors de la conception d'un système ou d'un logiciel, plusieurs critères liés à l'architecture matérielle doivent être pris en compte pour assurer une performance optimale, une compatibilité appropriée, et une évolutivité future. Voici les principaux critères à considérer :

1. **Type de système cible:** Il faut déterminer quel type de matériel le logiciel va cibler. Cela pourrait être des ordinateurs personnels, des serveurs, des systèmes embarqués, des appareils mobiles, des systèmes distribués, etc.
2. **Capacités du système:** Cela comprend des facteurs comme la puissance de traitement (nombre et type de processeurs, vitesse d'horloge), la quantité et le type de mémoire (RAM, disque dur), la capacité de stockage, le type de système d'exploitation, les capacités graphiques, etc.
3. **Réseau:** Quel type de connexions réseau le système nécessite-t-il ? Cela pourrait inclure des connexions filaires ou sans fil, la bande passante nécessaire, les protocoles réseau à utiliser, etc.
4. **Exigences en matière de périphériques:** Si le logiciel va interagir avec des périphériques spécifiques, quels sont-ils ? Cela pourrait inclure des imprimantes, des scanners, des caméras, des lecteurs de code-barres, des capteurs, etc.
5. **Exigences en termes de puissance et de consommation énergétique:** Si le logiciel est destiné à être utilisé sur des dispositifs alimentés par batterie ou des systèmes à faible consommation d'énergie, la consommation d'énergie peut être un facteur important.
6. **Évolutivité:** Il est important de considérer comment le système peut être mis à niveau ou étendu en termes de matériel à l'avenir. Cela peut inclure l'ajout de plus de mémoire, l'augmentation de la capacité de stockage, ou le passage à des processeurs plus rapides.
7. **Résilience et tolérance aux pannes:** Ce critère est crucial pour les systèmes critiques. La manière dont le système se comportera en cas de panne d'un composant matériel donne une indication sur le besoin d'une conception redondante.

8. Sécurité: Cela comprend la sécurité physique du matériel lui-même, ainsi que des considérations comme le chiffrement matériel, le stockage sécurisé des données sensibles, etc.

Prendre en compte ces critères lors de la conception d'un système aidera à garantir que le résultat final répondra aux besoins et attentes des utilisateurs tout en offrant une performance et une fiabilité optimales.

Architecture logicielle

Voici quelques-uns des principaux modèles d'architecture logicielle :

- 1. Architecture Monolithique:** Dans ce type d'architecture, le logiciel est construit comme un seul bloc cohérent. Toutes les fonctionnalités du logiciel sont gérées et servies à partir d'une seule application. Toutefois, cette architecture peut devenir complexe et difficile à gérer au fur et à mesure que l'application grandit.
- 2. Architecture en Couches (ou n-tiers):** Cette architecture divise l'application en composants logiques séparés ou "couches". Chaque couche a une fonctionnalité spécifique, comme la présentation, la logique métier, les opérations de données, etc. Les couches interagissent entre elles de manière séquentielle, en minimisant la dépendance et en maximisant la séparation des responsabilités.
- 3. Architecture Orientée Services (SOA):** Dans l'architecture SOA, l'application est conçue autour de services indépendants, mais interconnectés. Chaque service est autonome et réalise une fonction spécifique. Les services peuvent communiquer entre eux et être réutilisés dans différents contextes, favorisant la modularité et la flexibilité.
- 4. Architecture Microservices:** C'est une variante de l'architecture SOA où chaque service est conçu pour être complètement indépendant et peut fonctionner de manière autonome. Chaque microservice est responsable d'une fonctionnalité spécifique et peut être développé, déployé, et mis à l'échelle indépendamment des autres services.
- 5. Architecture événementielle (*Event-Driven*, EDA):** Dans ce modèle, le flux du logiciel est déterminé par des événements tels que les entrées utilisateur, les capteurs ou d'autres programmes. Cette architecture est adaptée pour les applications en temps réel et les systèmes hautement réactifs.
- 6. Pipeline d'Architecture Data-Flow:** Dans cette architecture, l'application est représentée comme une série d'opérations de traitement des données. Chaque opération est un "filtre"

qui transforme les données avant de les transmettre à l'opération suivante dans la "pipeline".

7. Architecture Peer-to-Peer (P2P): Dans ce modèle, chaque nœud d'un réseau fonctionne à la fois comme un client et un serveur, partageant et consommant des ressources dans un réseau décentralisé.

8. Architecture Modèle-Vue-Contrôleur (MVC): Ce modèle de conception qui divise une application en trois composants interconnectés : le *Modèle* (qui gère les données et la logique métier), la *Vue* (qui présente les données à l'utilisateur), et le *Contrôleur* (qui interprète les entrées de l'utilisateur et met à jour le *Modèle* et la *Vue* en conséquence). Cette séparation des responsabilités favorise la réutilisation du code, simplifie la maintenance et permet une représentation visuelle multiple et cohérente des mêmes données.

Chacun de ces modèles a ses propres forces et faiblesses, et le choix de l'architecture logicielle appropriée dépendra des besoins spécifiques du projet.

Architecture Monolithique

L'architecture monolithique est un modèle de conception de logiciel où toutes les fonctionnalités et les composants de l'application sont unifiés en une seule entité indivisible ou monolithe. Dans ce cadre, chaque composant ou service du logiciel est interconnecté et interdépendant. Tous les composants de l'architecture monolithique fonctionnent ensemble pour atteindre l'objectif global de l'application.

Dans une application monolithique, la logique de l'interface utilisateur, la logique métier et les opérations de base de données sont toutes contenues dans une seule base de code et sont interdépendantes. Cela signifie que si un seul composant du logiciel a besoin d'être mis à jour ou modifié, c'est tout le logiciel qui doit être redéployé, et non juste le composant concerné.

Bien que cela puisse sembler peu efficace ou rigide, l'architecture monolithique a en réalité plusieurs avantages. Tout d'abord, les applications monolithiques sont souvent plus simples à développer et à tester puisqu'il n'y a pas de dépendances inter-services ou de communication réseau complexe à gérer. De plus, ces applications peuvent être plus performantes parce qu'elles évitent la latence de communication entre les services.

Cependant, l'architecture monolithique a également des inconvénients. En raison du couplage étroit de ses divers composants, une application monolithique peut être difficile à mettre à l'échelle ou à maintenir à mesure qu'elle grandit. De plus, si un seul composant du système rencontre une erreur ou tombe en panne, l'ensemble du système est souvent affecté.

En résumé, l'architecture monolithique peut être un excellent choix pour les applications plus petites à moyennes, ou lorsque vous débutez un projet. Cependant, pour les grandes applications à fort trafic, une architecture plus modulaire comme les microservices peut être préférable.

Architecture en couches

L'architecture en couches, également appelée architecture n-tiers, est un modèle d'architecture logicielle qui partitionne une application en des «couches» logiques distinctes. Chaque couche a une responsabilité spécifique et fonctionne de manière distincte des autres.

Typiquement, une architecture en couches sera divisée en trois composantes principales : la couche de présentation, la couche de logique métier (ou couche de traitement) et la couche d'accès aux données.

1. **La couche de présentation**, aussi appelée couche interface utilisateur (UI), est responsable de l'interaction avec l'utilisateur. Tout ce que l'utilisateur voit et interagit appartient à cette couche, comme l'interface graphique ou l'interface utilisateur web.
2. **La couche de logique métier**, aussi appelée couche de traitement, contient toute la logique d'entreprise que l'application doit exécuter. Cette couche fonctionne comme un médiateur entre la couche de présentation et la couche d'accès aux données.
3. **La couche d'accès aux données** gère la persistance et le stockage des informations. Cette couche se connecte directement à la base de données et exécute les requêtes nécessaires. Elle ensuite renvoie les résultats à la couche de logique métier.

L'un des avantages clés de l'architecture en couches est la séparation des préoccupations, qui favorise l'organisation et la modularité du code. Cela permet aux développeurs d'apporter des modifications à une couche sans affecter les autres. Par exemple, on pourrait modifier la façon dont l'application stocke ses données sans affecter la logique métier ou l'interface utilisateur.

Cependant, il est important de noter qu'une couche ne doit interagir qu'avec la couche immédiatement en dessous ou au-dessus d'elle, ce qui peut limiter la flexibilité de l'application.

En somme, l'architecture en couches est un modèle couramment utilisé qui favorise la modularité, la flexibilité et la réutilisabilité du code, tout en maintenant une structure organisée.

Architecture orientée services

L'architecture orientée services, plus connue sous son acronyme SOA (pour Service-Oriented Architecture), est un modèle d'architecture logicielle qui divise une application en services distincts mais interconnectés. Chacun de ces services est autonome et réalise une fonction spécifique. Cela signifie qu'une seule application peut être conçue comme un ensemble de services indépendants qui interagissent pour réaliser une tâche globale.

Les services dans une architecture SOA sont définis par leurs capacités, ils sont indépendants les uns des autres, et communiquent grâce à un protocole d'échange de données bien défini, généralement HTTP ou AMQP. Chaque service est autonome et ne dépend pas directement des autres, ce qui favorise une conception modulaire et réutilisable de l'application.

Un des avantages principaux de l'architecture SOA est qu'elle favorise la réutilisabilité. Parce que chaque service est autonome et bien défini, il est possible de réutiliser des services dans différentes parties de l'application, ou même dans différentes applications. Cela peut réduire le temps de développement et favoriser la cohérence dans l'ensemble du système.

SOA aide également à l'évolutivité du système. Étant donné que chaque service est indépendant, il est possible de mettre à l'échelle ou de modifier des services individuels sans affecter le reste du système. Si un service en particulier nécessite plus de ressources, il peut être mis à l'échelle en ajoutant simplement plus d'instances de ce service.

Cependant, SOA n'est pas sans inconvénients. La communication entre les services peut être coûteuse en termes de performance, et la gestion des erreurs peut être complexe dans un système distribué. De plus, l'interdépendance des services peut entraîner des problèmes de performance et affecter l'utilisation des ressources.

Dans l'ensemble, SOA est un modèle d'architecture efficace pour la création de systèmes flexibles et réutilisables, mais il nécessite une planification et une gestion minutieuse pour éviter les problèmes liés à l'interdépendance des services et à la communication coûteuse entre eux.

Architecture microservices

L'architecture microservices est un modèle d'architecture logicielle qui structure une application comme une collection de services faiblement couplés, autonomes et évolutifs. Chaque microservice est un composant logiciel qui a un but précis et qui peut être développé, déployé, géré et mis à l'échelle indépendamment.

Dans une architecture microservices, chaque service est conçu pour faire une chose bien définie et le fait bien. Un microservice est responsable d'une fonctionnalité spécifique ou d'un ensemble de fonctionnalités dans l'application, comme gérer les utilisateurs ou traiter les paiements. Il peut être développé en utilisant le langage de programmation le plus adapté à sa tâche, ce qui contribue à la flexibilité du système.

Un des principaux avantages des microservices est la facilité de développement et de déploiement. Comme chaque service est autonome, une équipe de développement peut travailler sur un service sans interférer avec le travail d'une autre équipe sur un autre service. De plus, si une partie de l'application a besoin de changer, seulement le microservice concerné a besoin d'être modifié et redéployé, sans affecter le reste de l'application.

Un autre avantage est la scalabilité. Si un service particulier a besoin de plus de ressources en raison d'une charge de travail plus importante, ce service peut être mis à l'échelle (augmenté en termes de ressources) indépendamment des autres.

Cependant, l'architecture microservices présente aussi des défis. La coordination et la gestion de multiples services peuvent être complexe. De plus, cette architecture introduit une latence de communication entre services. En outre, la complexité des opérations, la gestion des données, le déploiement et le débogage sont d'autres défis auxquels sont confrontées les architectures microservices.

Néanmoins, avec une bonne gestion et en utilisant les bonnes pratiques et outils, cette architecture offre une grande flexibilité, une scalabilité et une résilience, rendant possible la réalisation de grandes applications avec des systèmes compliqués.

Architecture événementielle

L'architecture événementielle (Event-Driven, ou architecture pilotée par les événements), est un modèle d'architecture logicielle où le flux du logiciel est déterminé par des événements tels que les actions de l'utilisateur, les déclencheurs de capteurs ou les messages d'autres programmes. Ceci est particulièrement populaire dans les environnements d'interface utilisateur moderne, les systèmes en temps réel et les environnements serveurs.

Dans une architecture pilotée par les événements, un morceau de logiciel (un "émetteur" d'événements) crée ou "émet" un événement en réponse à un changement d'état spécifique, tel qu'une action de l'utilisateur ou un déclencheur de capteur. Cet événement est ensuite reçu par un ou plusieurs "écouteurs d'événements", qui prennent des mesures en réponse.

Les avantages de ce type d'architecture incluent la faible couplage et la scalabilité. Comme les émetteurs et les écouteurs d'événements sont généralement indépendants l'un de l'autre, des modifications peuvent être apportées à un émetteur sans nécessairement affecter ses écouteurs, et vice versa. Cela facilite également l'ajout de nouvelles fonctionnalités : un nouvel écouteur d'événements peut être ajouté sans modifier l'émetteur.

L'architecture pilotée par les événements est également très efficace pour gérer des charges de travail élevées, car elle permet un traitement asynchrone des événements. Cela signifie que le système peut continuer à accepter et à traiter d'autres événements pendant qu'un événement est en cours de traitement.

Cependant, cette architecture peut rendre le flux de contrôle plus difficile à comprendre et à gérer, car le traitement est réparti entre de nombreux écouteurs d'événements, fonctionnant potentiellement de manière asynchrone.

En somme, l'architecture pilotée par les événements est idéale pour les applications qui doivent gérer un grand nombre d'événements, particulièrement dans des situations où la réactivité et la performance sont importantes.

Architecture pipeline de données

L'architecture pipeline data-flow, aussi connue sous le nom d'architecture d'écoulement de données ou pipeline de données, est une conception d'architecture logicielle qui permet aux données de passer à travers une séquence d'opérations ou de "stages" où chaque étape est conçue pour réaliser une opération spécifique sur les données. C'est un modèle largement utilisé dans le traitement des données, comme le traitement du signal, la traduction de langage de programmation, et le traitement en lots dans les systèmes Big Data.

Dans l'architecture de pipeline de données, chaque étape est conçue pour assurer une tâche spécifique. La sortie de chaque étape est ensuite transmise à l'étape suivante comme entrée, créant ainsi un "pipeline". Chaque étape du pipeline est généralement exécutée de manière asynchrone, augmentant ainsi l'efficacité du système.

Un des grands avantages de l'architecture pipeline de données est son efficacité pour le traitement des grandes quantités de données. Comme chaque étape du pipeline est exécutée en parallèle et de manière asynchrone, le système peut traiter de grands volumes de données de manière rapide et efficace.

De plus, l'architecture du pipeline de données facilite le découplage et la modularité. Chaque stage est indépendant des autres stages, donc des modifications dans une étape n'affectent pas les autres. Cela facilite l'ajout, la modification ou la suppression d'étapes dans le pipeline sans perturber l'ensemble du système.

Cependant, le débogage et la gestion d'une architecture de pipeline de données peut être complexe. De plus, comme chaque étape dépend de la sortie de l'étape précédente, une erreur ou une défaillance à n'importe quelle étape peut affecter l'ensemble du pipeline.

En conclusion, l'architecture en pipeline data-flow est une solution puissante pour manipuler efficacement de grands volumes de données, mais elle requiert une planification minutieuse et attentionnée pour éviter les éventuels problèmes de débogage et de gestion.

Architecture pair-à-pair

L'architecture pair-à-pair (Peer-to-Peer, P2P) est un modèle d'architecture distribuée qui partage les ressources et responsabilités parmi un ensemble de nœuds équivalents, appelés "pairs", sans nécessiter d'administrateur centralisé ou de serveur fixe. Dans une architecture peer-to-peer, chaque nœud, ou "pair", agit à la fois en tant que client et serveur, ce qui permet de partager des ressources et des services entre pairs.

Un des avantages majeurs d'une architecture peer-to-peer est sa scalabilité. Contrairement aux architectures client-serveur traditionnelles, où ajouter plus de clients peut surcharger le serveur, une architecture P2P devient plus efficace avec l'ajout de plus de pairs, car chaque pair supplémentaire augmente la capacité globale du réseau.

En plus, une architecture P2P est résiliente et robuste. Dans un système peer-to-peer, il n'y a pas de point unique de défaillance, ce qui signifie qu'un nœud peut tomber en panne sans perturber l'ensemble du système. Les données sont distribuées parmi les pairs, rendant le système plus résilient à la défaillance d'un nœud.

Cependant, les architectures peer-to-peer présentent également des défis. Les problèmes de sécurité sont prévalents dans les systèmes P2P, car tous les nœuds ont un accès équivalent, et il peut être difficile de garantir la fiabilité de tous les pairs. De plus, la performance et la qualité de service peuvent être inégales, car elles dépendent des capacités et de la connectivité de chaque pair.

Pour conclure, une architecture Peer-to-Peer est une conception d'architecture efficace et robuste pour la distribution de données, avec de fortes capacités de scalabilité et de résilience à la défaillance. Cependant, elle requiert une gestion de sécurité efficace et peut présenter des défis en termes de performance et de qualité du service.

Architecture MVC

L'architecture Modèle-Vue-Contrôleur (MVC) est un modèle d'architecture logicielle très couramment utilisé pour la conception d'applications, en particulier les applications web et mobiles.

Dans l'architecture MVC, une application est divisée en trois composants principaux : le Modèle, la Vue et le Contrôleur.

1. **Le Modèle (Model)** gère les données et la logique métier de l'application. Il communique avec la base de données pour récupérer, insérer et mettre à jour les informations. En même temps, il implémente les règles et la logique métier qui conduisent les processus de traitement des données.
2. **La Vue (View)** est responsable de l'affichage des données à l'utilisateur, c'est-à-dire de la présentation des données. La vue prend les données du modèle et les affiche d'une manière que l'utilisateur peut comprendre. Cela peut inclure des tableaux, des diagrammes, du texte brut ou toute autre forme de visualisation de données.
3. **Le Contrôleur (Controller)** sert d'intermédiaire entre le Modèle et la Vue. Il gère les entrées de l'utilisateur (par exemple les clics de souris, les pressions de touches, les actions vocales) et les traduit en actions appropriées à exécuter dans le Modèle. Une fois que le Modèle a effectué les changements nécessaires aux données, le Contrôleur met à jour la Vue pour refléter ces nouveautés.

L'architecture MVC offre plusieurs avantages, comme la séparation des préoccupations (SoC) : chaque composant a une responsabilité claire, ce qui rend l'application plus facile à développer, tester, et maintenir. De plus, il permet à plusieurs vues d'utiliser le même modèle, ce qui facilite la réutilisation du code.

Cependant, pour des applications plus complexes, l'architecture MVC peut devenir difficile à gérer car le Contrôleur peut finir par gérer beaucoup de logique d'affaires, ce qui l'entraîne à devenir volumineux et difficile à maintenir.

Diagrammes de package

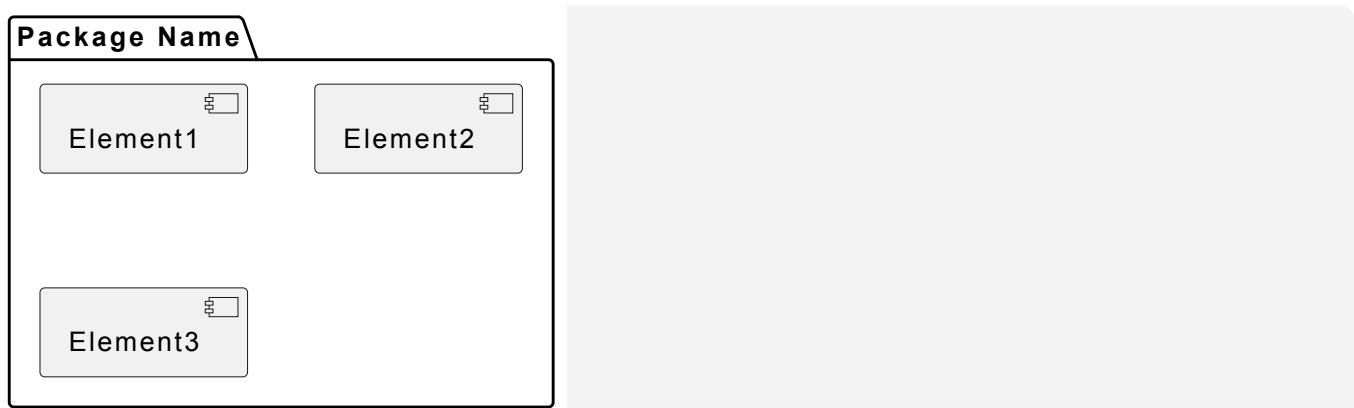
Un diagramme de package UML est une sorte de diagramme structurel utilisé pour représenter les packages ou les espaces de noms. Ces packages regroupent des éléments UML similaires tels que des classes, des diagrammes, des cas d'utilisation, des activités, etc.

Ces diagrammes sont utilisés pour diviser un logiciel complexe en plusieurs paquets gérables, ce qui facilite sa compréhension, son développement et sa maintenance. Chaque paquet peut être développé indépendamment puis assemblé pour créer un logiciel complet. Les packages peuvent également être réutilisés dans différents projets.

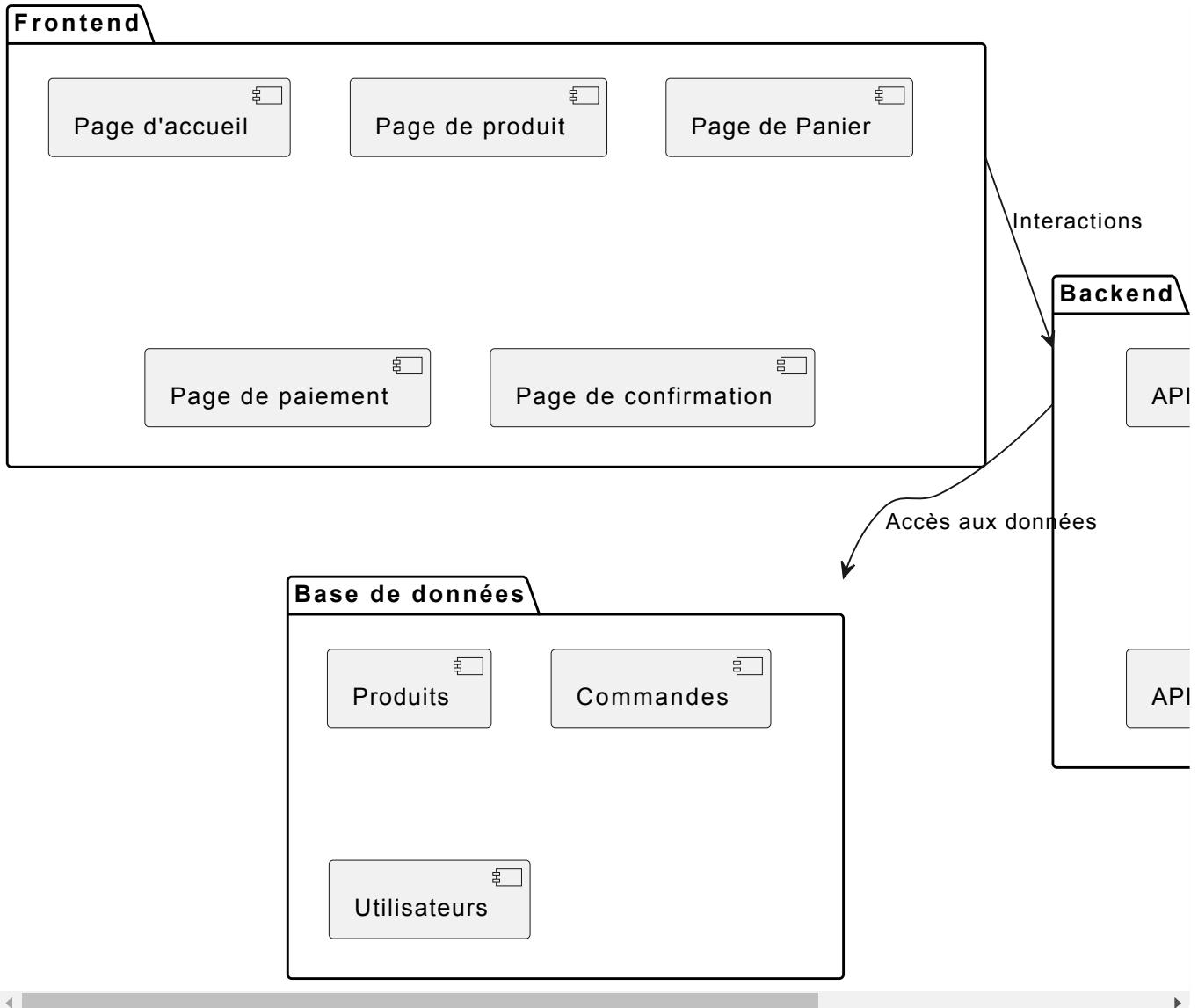
Utiliser un diagramme de package peut aider à :

1. Organiser un modèle UML en regroupant des éléments structurels liés.
2. Réduire la complexité en cachant les détails inutiles.
3. Réutiliser des artefacts UML en intégrant des packages préexistants dans un nouveau modèle.
4. Modéliser des dépendances entre différents éléments.

Exemple de structure d'un diagramme de package :



Exemple : Site web de commerce en ligne



Cet exemple de diagramme de package UML divise le projet en trois parties principales : *Frontend*, *Backend* et *Base de données*.

Le package *Frontend* comprend différentes pages que les utilisateurs peuvent visualiser et avec lesquelles ils peuvent interagir.

Le package *Backend* contient les API ou services nécessaires pour interagir avec la Base de données.

Le package *Base de données* comprend les différentes tables ou collections de données requises pour le projet.

Diagrammes de classes

Les différents types de diagrammes de classes UML

Les diagrammes de classes UML sont un type unique de diagramme dans la notation UML. Cependant, ils peuvent être catégorisés en fonction de leur utilisation ou de leur niveau de détail.

1. **Diagramme de classes conceptuel** (Diagramme du domaine) : Ce type de diagramme est utilisé au tout début de la conception, lorsque vous essayez simplement de déterminer quels éléments devraient être dans le système et comment ils devraient être liés.
2. **Diagramme de classes spécifique**: Il s'agit d'une version plus détaillée d'un diagramme de classes conceptuel. Il inclut des détails spécifiques sur les propriétés et les méthodes que chaque classe utilisera.
3. **Diagramme de classes de mise en œuvre**: Ce type de diagramme de classes est utilisé par les programmeurs pour savoir comment coder le système. Il inclut tous les détails que le programmeur doit connaître, tels que les types de données pour les propriétés et les noms de méthodes, ainsi que toute relation d'héritage entre les classes.

Il est important de noter que, bien que ce soit des catégories utiles pour le processus de conception, dans le système UML officiel, toutes sont simplement appelées "diagrammes de classes".

Dans un diagramme de classes UML, vous pouvez également représenter des relations comme l'association, l'héritage (généralisation), l'agrégation, la composition et la dépendance.

Références

1. Cours d'UML Chapitre 3 (<https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes>)
2. PlantUML (<https://plantuml.com/en/deployment-diagram>)

Prototypage

Le prototypage dans la conception de logiciels est une stratégie de développement qui implique la création de versions initiales simplifiées d'un logiciel, connues sous le nom de prototypes. Ces prototypes fonctionnent comme des brouillons ou des maquettes de la version finale du logiciel et sont généralement utilisés pour aider les développeurs et les parties prenantes (comme les clients ou les utilisateurs) à explorer les idées, à identifier les problèmes potentiels, à définir les exigences et à tester les fonctionnalités avant que le développement complet ne commence.

Voici deux types principaux de prototypage utilisés en développement de logiciels :

1. **Prototypage throwaway (ou jetable):** Ici, un modèle rapide est construit pour comprendre les exigences de l'utilisateur. Il n'est pas censé être évolutif ou être une base pour le produit final. Il est principalement utilisé pour recueillir des commentaires et des informations de l'utilisateur. Après avoir recueilli les informations, le prototype est jeté, et le système réel est développé avec une meilleure compréhension des exigences.
2. **Prototypage évolutif (ou incrémental):** Dans cette méthode, le prototype est construit et amélioré de façon continue, jusqu'à ce qu'il devienne la version finale du logiciel. Il est utilisé lorsque les exigences de l'utilisateur sont bien comprises. Cela permet à l'équipe de développement de produire des versions rapidement et d'obtenir des retours d'information précieux pour chaque étape du développement.

La pratique du prototypage a plusieurs avantages. Entre autres, elle augmente l'implication de l'utilisateur dans le développement du produit, facilite l'identification des erreurs et des omissions dans les premiers stades, améliore la spécification des exigences du système et permet d'obtenir une rétroaction rapide.

Maquettes (ou prototypes)

Types de maquettes

Lors de la conception d'un logiciel, divers types de maquettes (ou prototypes) peuvent être utilisés pour aider à visualiser et à tester différentes facettes du système. Voici quelques exemples :

1. **Maquette statique:** La maquette la plus simple et la plus rapide à créer, souvent créée avec du papier et un stylo ou un logiciel graphique. Elle sert à établir l'aspect général et la disposition de l'interface utilisateur sans inclure de fonctionnalités interactives.
2. **Maquette interactive:** Similaire à une maquette statique, mais avec des éléments interactifs. Cela permet aux utilisateurs de cliquer sur différents éléments de l'interface utilisateur, généralement pour naviguer entre différentes vues ou écrans.
3. **Prototype horizontal:** Ce type de maquette montre une vue large de l'ensemble du système, mais sans beaucoup de profondeur. Il est utile pour visualiser l'interface utilisateur à un haut niveau et pour comprendre le flux général à travers le système.
4. **Prototype vertical:** Contrairement au prototype horizontal, le prototype vertical se concentre sur une petite portion du système mais avec beaucoup de détails. Il est utile pour explorer en profondeur les fonctionnalités et les interactions spécifiques.
5. **Maquette à haute fidélité:** Un prototype à haute fidélité est une version très détaillée et interactive du logiciel. Il peut être basé sur le codage réel et offrir une fonctionnalité complète ou presque complète.
6. **Maquette à faible fidélité:** Ces prototypes sont plus simplistes et ont souvent peu de simulation interactive. Ils sont utilisés pour représenter rapidement des idées et recevoir des commentaires tôt dans le processus de conception.

Chacun de ces types de maquettes a ses avantages et ses inconvénients, et peut être plus approprié à différents stades du processus de conception. Généralement, il est préférable de commencer par des maquettes à faible fidélité et de progresser vers des maquettes à haute fidélité à mesure que le concept du logiciel se précise.

Outils

Il existe de nombreux outils que vous pouvez utiliser pour créer des maquettes de logiciel. Le choix de l'outil dépendra de vos besoins spécifiques, tels que le niveau de détail que vous souhaitez, la facilité d'utilisation, et combien vous êtes prêt à dépenser. Voici quelques outils populaires :

1. **Balsamiq**: C'est un excellent outil pour créer des maquettes à faible fidélité rapidement. Il a une interface utilisateur simplifiée qui permet de créer des écrans en utilisant une variété d'éléments d'interface utilisateur préfabriqués.
2. **Sketch**: Un outil de design vectoriel pour macOS qui est excellent pour créer des maquettes à haute fidélité. Il offre une large gamme d'options de personnalisation pour les éléments d'interface utilisateur.
3. **Adobe XD**: Il s'agit d'un outil de conception d'interface utilisateur qui permet aux concepteurs de créer des maquettes, des prototypages interactifs et des kits d'interface utilisateur. Il est disponible pour Windows et macOS.
4. **Figma**: Un outil de conception d'interface utilisateur basé sur le cloud qui permet de créer des maquettes à haute fidélité et de collaborer en temps réel.
5. **InVision**: Cet outil permet de créer des maquettes interactives à partir de vos designs. Il offre également des fonctionnalités de collaboration et de gestion des commentaires.
6. **Axure RP**: C'est un outil de conception professionnel qui permet de créer des maquettes hautement interactives et dynamiques.
7. **Marvel**: Un outil de conception d'interface utilisateur qui vous permet de créer des maquettes, des prototypes interactifs et des tests utilisateur.

Ces outils offrent tous une gamme de fonctionnalités qui peuvent vous aider à concevoir et à présenter l'interface utilisateur et le flux d'utilisation de votre application logicielle avant de commencer le développement réel.

Utilisations

Tous les outils mentionnés précédemment, comme Balsamiq, Sketch, Adobe XD, Figma, InVision, Axure RP, et Marvel, peuvent être utilisés à la fois pour le prototypage jetable et évolutif. Le type de prototypage que vous choisirez dépendra de votre processus de conception et de développement spécifique, ainsi que des besoins de votre projet.

Prototypage jetable: Balsamiq est souvent utilisé pour le prototypage jetable car il est facile à utiliser et permet de créer rapidement des maquettes à faible fidélité. Cela permet de

présenter rapidement une idée pour recueillir des commentaires et de jeter le modèle ensuite. Adobe XD, Sketch et Figma peuvent également être utilisés pour créer des maquettes plus détaillées à haute fidélité, qui sont ensuite jetées et ne sont pas directement utilisées dans le développement.

Prototypage évolutif: InVision, Figma, Adobe XD, Axure RP sont souvent utilisés pour le prototypage évolutif. Ces outils offrent des fonctionnalités plus avancées pour la création de maquettes interactives à haute fidélité. Avec ces outils, vous pouvez continuer à affiner et à développer vos maquettes en réponse aux commentaires des utilisateurs, et ceux-ci peuvent même servir de base pour le développement de l'interface utilisateur finale.

Il est important de souligner que ces outils sont juste des moyens pour visualiser et tester des idées pendant le processus de conception. Le code derrière ces maquettes n'est généralement pas réutilisable pour le produit final dans le développement de logiciels. L'idée de prototypage évolutif dans ce contexte est plus sur l'évolution du design qu'une évolution du code du prototype vers le produit final.

Conception détaillée

La conception détaillée ou de bas niveau d'un logiciel utilise souvent les types de diagrammes suivants pour représenter visuellement différents aspects de l'architecture du logiciel :

1. **Diagramme de classes:** *spécifique et/ou de mise en oeuvre.*
2. **Diagramme de séquence:** interaction et messages entre les objets.
3. **Diagramme d'activités:** comme dans la conception architecturale, mais pour des scénarios plus complexes.
4. **Diagramme d'états:** cycle de vie d'un objet ou d'un processus.
5. **Diagramme de composants:** comme dans la conception architecturale, mais révisé et possiblement plus détaillé.
6. **Diagramme de déploiement:** comme dans la conception architecturale, mais révisé et possiblement plus détaillé.

Modèles de données

Les modèles de données servent à décrire comment les données sont structurées dans une base de données.

Principaux modèles de données

1. **Modèle relationnel:** Les données sont organisées en tables (relations). Chaque tableau représente une entité et les relations entre les entités sont exprimées par rapport entre les tables. Les SGBD (systèmes de gestion de bases de données) relationnels sont très populaires.
2. **Modèle objet-relationnel:** Une extension du modèle relationnel qui permet aux développeurs d'intégrer leurs propres types de données, opérateurs et fonctions dans le SGBD, étendant ainsi les capacités du modèle de données relationnel. PostgreSQL est un exemple d'un SGBD qui supporte ce modèle.
3. **Modèle hiérarchique:** Les données sont organisées en arborescence, avec un nœud racine, des nœuds enfants et ainsi de suite. Chaque nœud correspond à une entité et chaque lien représente une relation entre deux entités.
4. **Modèle en réseau:** C'est une extension du modèle hiérarchique où chaque nœud peut avoir plusieurs parents. Cela permet une représentation plus complexe des relations entre données.
5. **Modèle orienté objet:** Les données sont organisées sous la forme d'objets, qui sont des instances de classes. Les relations entre les objets sont exprimées par des associations, de l'héritage et de la composition.
6. **Modèle de document:** Utilisé dans les bases de données NoSQL, les informations sont organisées en documents, généralement au format JSON.
7. **Modèle orienté graphe:** Les données sont conservées sous forme de graphes avec des nœuds, des arêtes et des propriétés.

Principaux types de bases de données

1. **SQL (Structured Query Language) ou SGBD relationnel ou objet-relationnel:** Basées sur le modèle relationnel objet-relationnel des données. MySQL, PostgreSQL, Oracle, et SQL

Server en sont quelques exemples.

2. **NoSQL:** Ils sont capables de gérer un grand volume de données réparties sur de nombreux serveurs. Ils comprennent les bases de données de documents comme MongoDB, les bases de données de graphes comme Neo4j, les bases de données de colonnes comme Cassandra, et les bases de données de clés-valeurs comme Redis.
3. **NewSQL:** Concilie les avantages des bases de données NoSQL pour le traitement des big data avec la garantie d'ACID (atomicité, cohérence, isolation, durabilité) dont bénéficient les bases de données SQL. Exemple : VoltDB, CockroachDB.
4. **Bases de données en mémoire:** Elles conservent l'ensemble de leurs données en RAM et sont donc extrêmement rapides. Exemple : Redis, Memcached.

Normalisation

La normalisation des bases de données est un processus qui organise les tables de données et leurs relations pour réduire les redondances et améliorer l'intégrité des données. L'objectif est d'isoler les données de manière à ce qu'elles soient conservées dans un emplacement unique et soient accessibles via un point d'intersection unique.

La normalisation se fait généralement en plusieurs étapes, appelées formes normales, chacune ayant un objectif spécifique.

Voici les trois premières formes normales avec des exemples:

- 1. Première forme normale (1NF):** Les tables sont organisées de façon à éliminer les duplications de données. Chaque table doit avoir une clé primaire, chaque colonne doit être atomique (indivisible), et chaque enregistrement doit être unique.

Par exemple, si vous avez une table "Étudiants" avec les colonnes "ID étudiant", "Nom étudiant", "Cours suivis", et si un étudiant suit plusieurs cours, ce n'est pas en 1NF car la colonne "Cours suivis" aurait plusieurs valeurs. Pour atteindre la 1NF, nous devons diviser "Cours suivis" en plusieurs enregistrements.

- 2. Deuxième forme normale (2NF):** Les tables doivent être en 1NF plus toutes les colonnes non-clés doivent être pleinement dépendantes de la clé primaire. En d'autres termes, elles ne doivent pas dépendre d'une partie seulement de la clé primaire (dans le cas des clés primaires composites).

Si nous avons une table "Inscriptions" avec "ID étudiant", "Nom étudiant", "ID cours", "Nom du cours", "Note", la clé primaire est composée de "ID étudiant" et "ID cours". Cependant, "Nom étudiant" dépend seulement de "ID étudiant" et "Nom du cours" dépend uniquement de "ID cours", alors cette table n'est pas en 2NF. Pour atteindre la 2NF on doit diviser cette table en deux: "Étudiant" et "Cours".

- 3. Troisième forme normale (3NF):** Les tables doivent être en 2NF et toutes les colonnes doivent être directement dépendantes de la clé primaire.

Si nous avons la table "Étudiants" avec "ID étudiant", "Nom étudiant", "Code postal", "Ville", "État". Ici, "ville" et "état" dépendent du "code postal", pas directement de "ID étudiant". Pour mettre cela en 3NF, on doit diviser cette table en deux tables "Étudiants" et "Code postal".

Il existe d'autres formes normales comme BCNF, 4NF, 5NF (ou PJ/NF), 6NF, mais les trois formes normales décrites ci-dessus sont les plus souvent utilisées. La normalisation doit être équilibrée car une trop grande normalisation peut entraîner une complexité excessive et une mauvaise performance.

Exemples

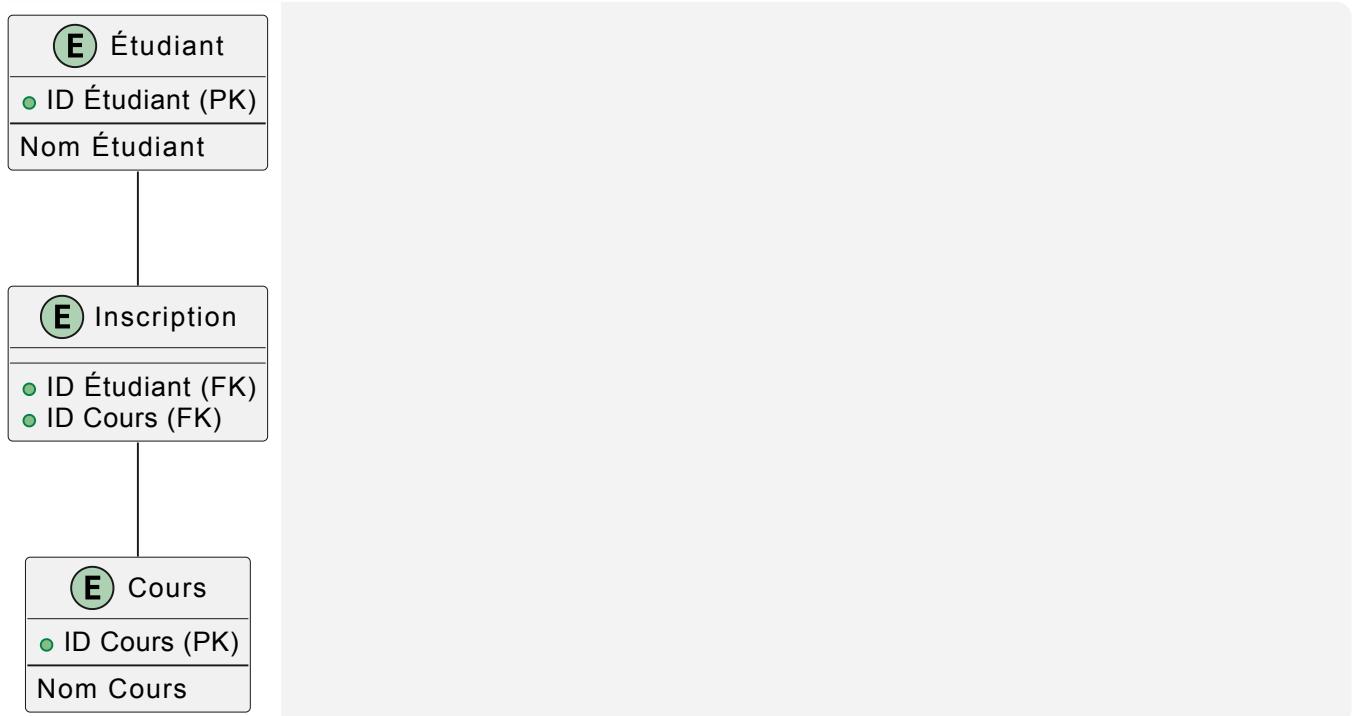
1NF

Si on a une table "Étudiants" avec les colonnes "ID étudiant", "Nom étudiant", "Cours suivis", et si un étudiant suit plusieurs cours, ce n'est pas en 1NF car la colonne "Cours suivis" aurait plusieurs valeurs. Pour atteindre la 1NF, nous devons diviser "Cours suivis" en plusieurs enregistrements.

Table Étudiant :

ID étudiant	Nom étudiant	Cours suivis
1	Alice	Maths, Physique
2	Bob	Maths, Anglais

PlantUML pour la 1NF :



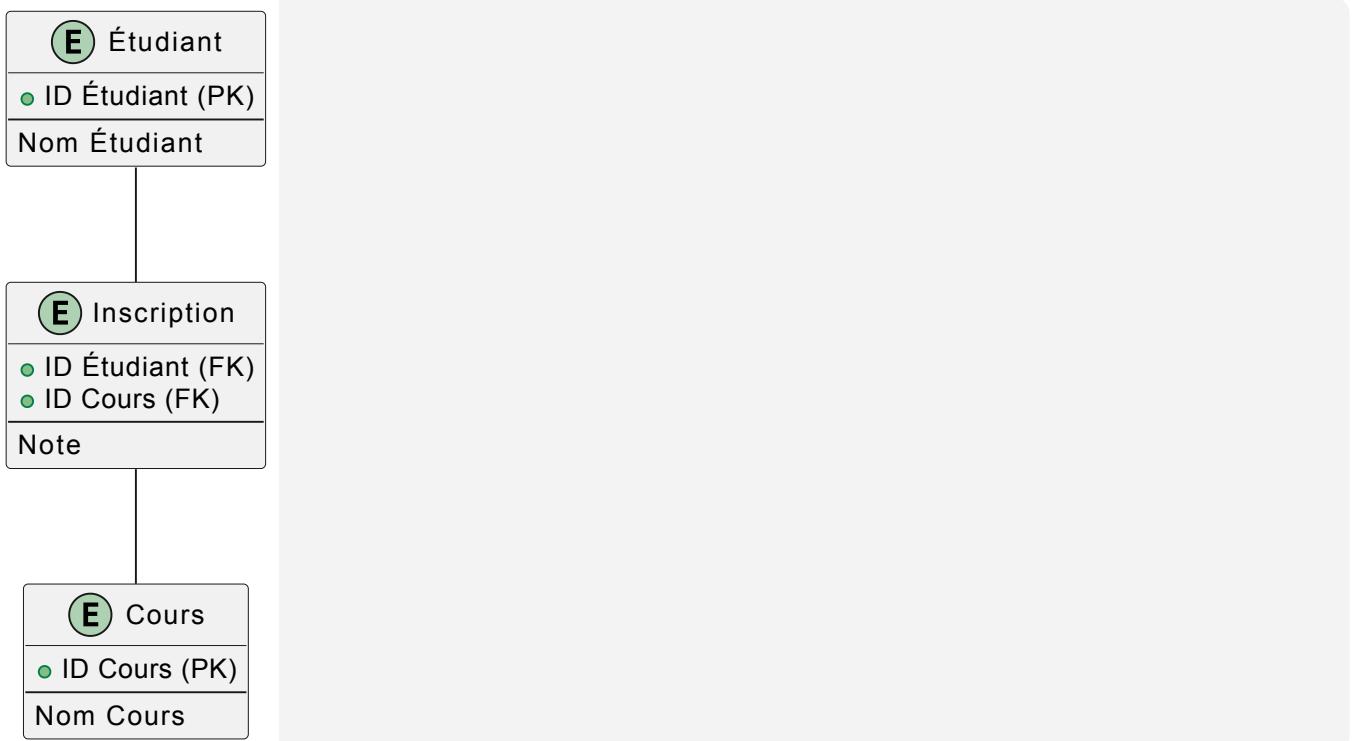
2NF

Si on a une table "Inscriptions" avec "ID étudiant", "Nom étudiant", "ID cours", "Nom du cours", "Note", la clé primaire est composée de "ID étudiant" et "ID cours". Cependant, "Nom étudiant" dépend seulement de "ID étudiant" et "Nom du cours" dépend uniquement de "ID cours", alors cette table n'est pas en 2NF. Pour atteindre la 2NF on doit diviser cette table en deux : "Étudiant" et "Cours".

Table Inscriptions :

ID étudiant	Nom étudiant	ID cours	Nom du cours	Note
1	Alice	101	Maths	A
1	Alice	201	Physique	B
2	Bob	101	Maths	C
2	Bob	202	Anglais	D

PlantUML pour la 2NF :



3NF

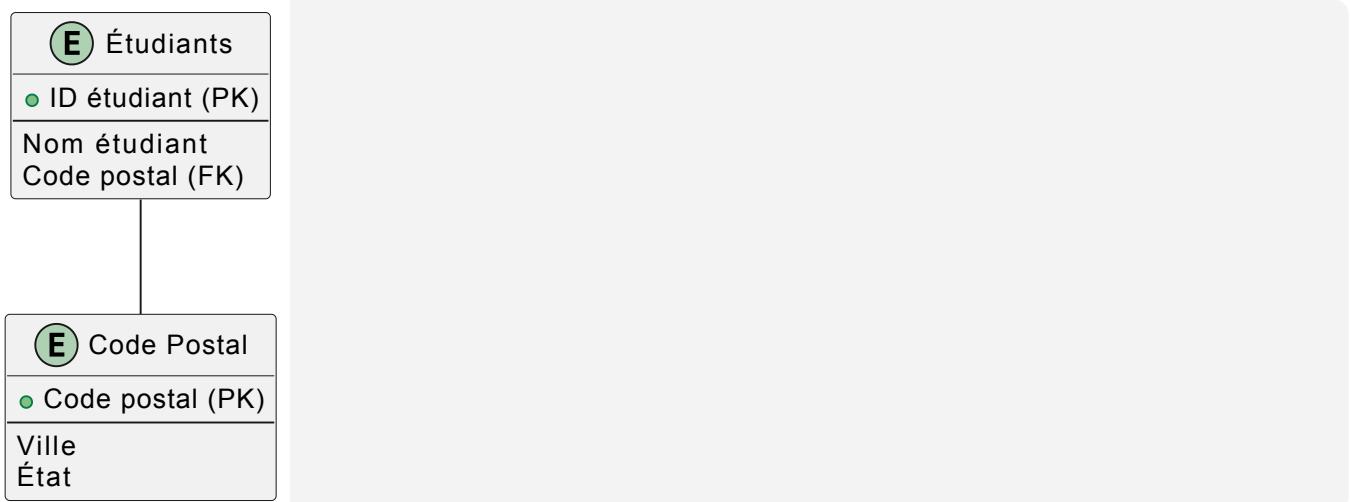
Si nous avons la table "Étudiants" avec "ID étudiant", "Nom étudiant", "Code postal", "Ville", "État". Ici, "ville" et "état" dépendent du "code postal", pas directement de "ID étudiant". Pour mettre cela en 3NF, on doit diviser cette table en deux tables "Étudiants" et "Code postal".

Table "Étudiants" :

ID étudiant	Nom étudiant	Code postal	Ville	État
1	Alice	75000	Paris	France
2	Bob	69000	Lyon	France
3	Charlie	75000	Paris	France

Dans cette situation, ni "Ville" ni "État" ne sont directement dépendants de "ID Étudiant". Ils dépendent plutôt du "Code postal". Pour passer à la 3NF, on crée une nouvelle table "Code postal" et on modifie la table "Étudiant" en conséquence.

PlantUML pour la 3NF :



Dénormalisation

La **dénormalisation** est le processus de réintroduction de la redondance dans une base de données relationnelle normalisée, généralement pour améliorer les performances en réduisant le nombre de jointures nécessaires pour extraire les données.

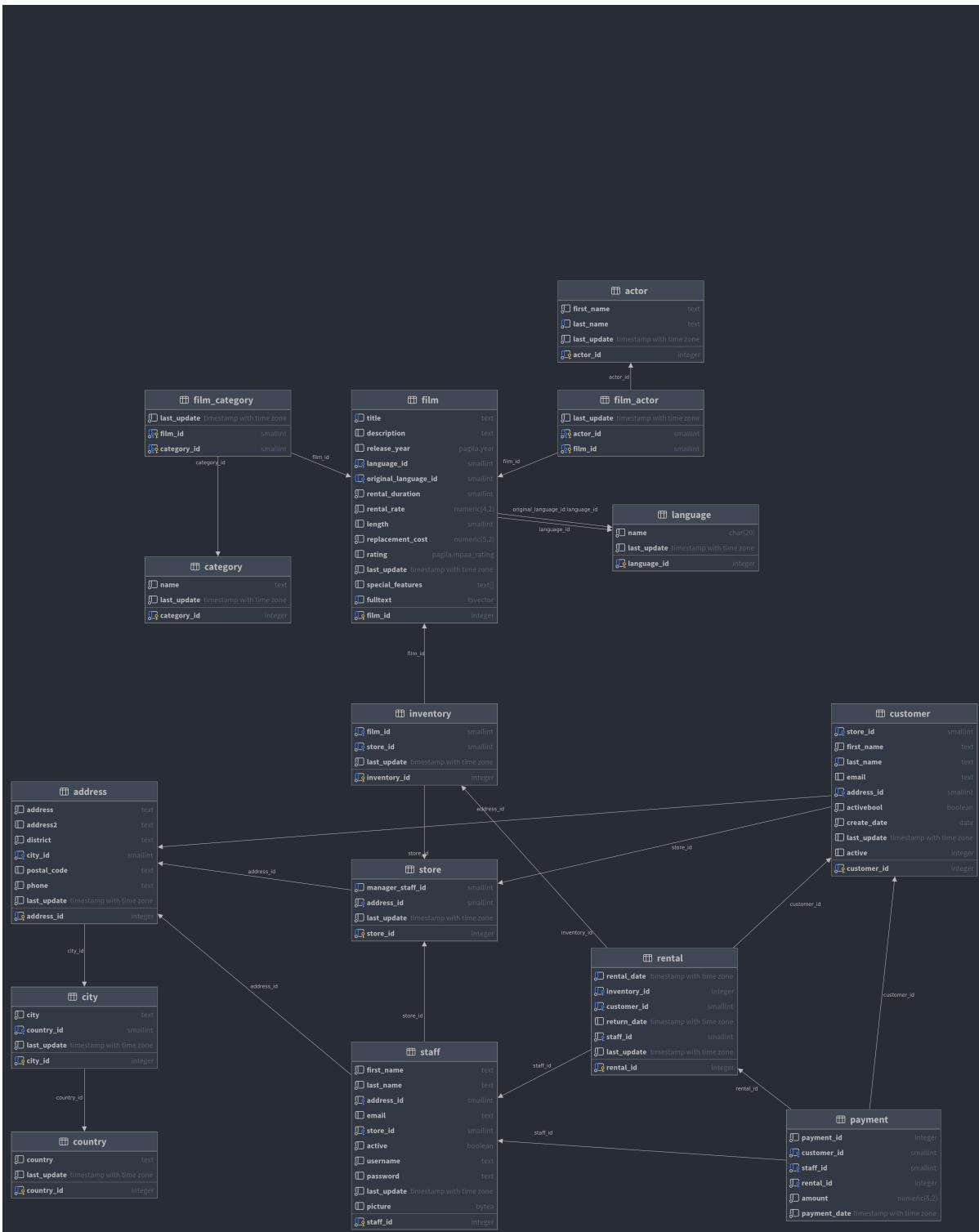
Voici quelques raisons pour lesquelles vous pourriez envisager une dénormalisation :

- 1. Amélioration de la performance des requêtes de lecture:** Si votre application a pour objectif des performances de lecture rapides et nécessite moins d'écritures, la dénormalisation peut aider à accélérer vos requêtes de lecture en réduisant le nombre de jointures nécessaires et en permettant un accès plus direct aux données.
- 2. Réduction de la complexité:** Même si la normalisation réduit la redondance et améliore la logique du stockage des données, elle peut rendre certains types de requêtes ou d'analyses plus complexes. Parfois, une certaine redondance (comme le stockage pré-calculé d'un total) peut rendre une requête beaucoup plus simple.
- 3. Compatibilité avec des non-SGBD:** Pour interagir avec certains systèmes ou services qui ne sont pas des SGBD, comme les systèmes de recherche en texte intégral, vous pouvez avoir besoin de données dénormalisées.
- 4. Optimisation pour les systèmes de bases de données distribuées (par ex. NoSQL):** De nombreux systèmes de bases de données distribuées favorisent les modèles de données dénormalisées en raison de la façon dont ils répartissent les données sur le réseau.

Il convient cependant de noter que la dénormalisation n'est généralement pas la première méthode d'optimisation à envisager et qu'elle doit être utilisée avec prudence. Elle peut en effet introduire des anomalies de données, augmenter la complexité des écritures de données et occuper plus d'espace de stockage. Avant de dénormaliser, il convient d'examiner les autres options d'optimisation et de bien comprendre les conséquences de la dénormalisation.

Exemple

Comment pourrait-on dénormaliser la BD suivante ?



Pagila

Exemple : ORDBMS

ORDBMS: Object-Relational DBMS

Contacts normalisés

contacts1 Schéma relationnel

```
DROP SCHEMA IF EXISTS contacts1 CASCADE;
CREATE SCHEMA contacts1;
SET search_path TO contacts1;

CREATE TABLE users (
    uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    username VARCHAR(15) NOT NULL,
    email TEXT NOT NULL,
    firstname TEXT,
    lastname TEXT,
    emergency_contact TEXT
);
```

```
INSERT INTO users (username, email, emergency_contact)
VALUES ('denis', 'denis.rinfret@example.com', 'help@example.com'),
       ('minh', 'minh@example.com', 'contact@example.com');
```

```
SELECT * FROM users;
```

uid	username	email	firstname	lastname	emergency_contact
1	denis	denis.rinfret@example.com	None	None	help@example.com
2	minh	minh@example.com	None	None	contact@example.com

Que devons-nous faire pour permettre à un utilisateur d'avoir plus d'un contact ?

1. Si le nombre de contacts est fixe, disons n contacts, alors nous pourrions avoir n colonnes de contacts, tant que n est petit.
2. Mais si n n'est pas petit ou si n est inconnu, alors nous devons avoir une autre table, une table contacts, pour préserver la première forme normale.
3. La première forme normale stipule que chaque valeur de colonne doit être atomique.

contacts2 Schéma relationnel

```
DROP SCHEMA IF EXISTS contacts2 CASCADE;
CREATE SCHEMA contacts2;
SET search_path TO contacts2;

CREATE TABLE users (
    uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    username VARCHAR(15) NOT NULL,
    email TEXT NOT NULL,
    firstname TEXT,
    lastname TEXT
);

CREATE TABLE contacts (
    cid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    email TEXT NOT NULL,
    uid INTEGER REFERENCES users(uid)
)
```

```
INSERT INTO users (username, email)
VALUES ('denis', 'denis.rinfret@example.com'),
       ('minh', 'minh@example.com');

INSERT INTO contacts (email, uid)
VALUES ('help@example.com', 1), ('minh@example.com', 1),
       ('contact@example.com', 2), ('ha@example.com', 2);
```

```
SELECT * FROM users;
```

uid	username	email	firstname	lastname
1	denis	denis.rinfret@example.com	None	None
2	minh	minh@example.com	None	None

```
SELECT * FROM contacts;
```

cid	email	uid
1	help@example.com	1
2	minh@example.com	1
3	contact@example.com	2
4	ha@example.com	2

Jointure nécessaire pour obtenir toutes les données

```
SELECT * FROM users u INNER JOIN contacts c ON u.uid = c.uid;
```

uid	username	email	firstname	lastname	cid	email_1	uid_1
1	denis	denis.rinfret@example.com	None	None	1	help@example.com	1
1	denis	denis.rinfret@example.com	None	None	2	minh@example.com	1
2	minh	minh@example.com	None	None	3	contact@example.com	2
2	minh	minh@example.com	None	None	4	ha@example.com	2

1. Les jointures peuvent être lentes.
2. Mais sans normalisation, les données peuvent être redondantes et des anomalies peuvent apparaître.
3. Une SGDB object-relationalle (SGBDOR, ou ORDBMS) peut aider.

Contacts dénormalisés

contacts3 Schéma objet-relationnel

```

DROP SCHEMA IF EXISTS contacts3 CASCADE;
CREATE SCHEMA contacts3;
SET search_path TO contacts3;

CREATE TABLE users (
    uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    username VARCHAR(15) NOT NULL,
    email TEXT NOT NULL,
    firstname TEXT,
    lastname TEXT,

```

```
    emergency_contacts TEXT[]  
);
```

```
INSERT INTO users (username, email, emergency_contacts)  
VALUES ('denis', 'denis.rinfret@example.com',  
       ARRAY ['help@example.com', 'minh@example.com']),  
       ('minh', 'minh@example.com',  
        ARRAY ['contact@example.com', 'ha@example.com']);
```

```
SELECT uid, emergency_contacts FROM users;
```

uid	emergency_contacts
1	["help@example.com", "minh@example.com"]
2	["contact@example.com", "ha@example.com"]

```
SELECT uid, username, email, firstname, lastname,  
      unnest(emergency_contacts)  
FROM users;
```

uid	username	email	firstname	lastname	unnest
1	denis	denis.rinfret@example.com	None	None	help@example.com
1	denis	denis.rinfret@example.com	None	None	minh@example.com
2	minh	minh@example.com	None	None	contact@example.com
2	minh	minh@example.com	None	None	ha@example.com

Trouver les utilisateurs avec un courriel spécifique comme premier contact d'urgence

```
SELECT uid, username, email
FROM users
WHERE emergency_contacts[1] = 'help@example.com';
```

uid	username	email
1	denis	denis.rinfret@example.com

Trouver un utilisateur avec un courriel spécifique comme contact d'urgence (n'importe quelle position)

```
SELECT uid, username, email
FROM users
WHERE 'ha@example.com' = ANY (emergency_contacts);
```

uid	username	email
2	minh	minh@example.com

Trouver les utilisateurs qui sont listés comme contact d'urgence d'autres utilisateurs

```
SELECT uid, username, email
FROM users
WHERE email IN (SELECT unnest(emergency_contacts)
                 FROM users);
```

uid	username	email
2	minh	minh@example.com

```
SELECT distinct uid, username, email
FROM users
      INNER JOIN
        (SELECT unnest(emergency_contacts) AS emergency_email
         FROM users) AS all_contacts
      ON email = emergency_email;
```

uid	username	email
2	minh	minh@example.com

Sans unnest ni sous-requêtes

```
SELECT u1.uid, u1.username, u1.email
FROM users u1 INNER JOIN users u2
      ON u1.email = ANY (u2.emergency_contacts);
```

uid	username	email
2	minh	minh@example.com

Types définis par les utilisateurs

User-Defined Types (UDT)

contacts4 Schéma objet-relationnel

```
DROP SCHEMA IF EXISTS contacts4 CASCADE;
CREATE SCHEMA contacts4;
SET search_path TO contacts4;

CREATE TYPE contact_type AS ENUM ('emergency', 'friend',
                                    'family', 'colleague');

CREATE TYPE contact AS (
    email TEXT,
    type contact_type
);
```

```
CREATE TABLE users (
    uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    username VARCHAR(15) NOT NULL,
    email TEXT NOT NULL,
    firstname TEXT,
    lastname TEXT,
    contacts contact[]
);
```

```
INSERT INTO users (username, email, contacts)
VALUES ('denis', 'denis.rinfret@example.com',
        ARRAY[('help@example.com', 'emergency')::contact,
              ('minh@example.com', 'friend')::contact]),
        ('minh', 'minh@example.com',
        ARRAY[('contact@example.com', 'family')::contact,
              ('ha@example.com', 'colleague')::contact]);
```

```
SELECT uid, contacts FROM users;
```

uid	contacts
1	{"(help@example.com,emergency)","(minh@example.com,friend)"}
2	{"(contact@example.com,family)","(ha@example.com,colleague)"}

```
SELECT uid, username, email,
       (unnest(contacts)::contact).*
FROM users;
```

uid	username	email	email_1	type
1	denis	denis.rinfret@example.com	help@example.com	emergency
1	denis	denis.rinfret@example.com	minh@example.com	friend
2	minh	minh@example.com	contact@example.com	family
2	minh	minh@example.com	ha@example.com	colleague

```
SELECT uid, username, email,
       (unnest(contacts)::contact).email AS contact_email,
       (unnest(contacts)::contact).type AS contact_type
FROM users;
```

uid	username	email	contact_email	contact_type
1	denis	denis.rinfret@example.com	help@example.com	emergency
1	denis	denis.rinfret@example.com	minh@example.com	friend
2	minh	minh@example.com	contact@example.com	family
2	minh	minh@example.com	ha@example.com	colleague

Trouver tous les contacts d'urgence d'un utilisateur

```
SELECT *
FROM (SELECT uid, username, email,
    (unnest(contacts)::contact).email AS contact_email,
    (unnest(contacts)::contact).type AS contact_type
    FROM users) AS temp
WHERE contact_type = 'emergency';
```

uid	username	email	contact_email	contact_type
1	denis	denis.rinfret@example.com	help@example.com	emergency

Trouver les utilisateurs sans contacts d'urgence

```
SELECT uid, username, email
FROM users
WHERE 'emergency' NOT IN (SELECT (unnest(contacts)::contact).type);
```

uid	username	email
2	minh	minh@example.com

```
SELECT uid, username, email
FROM users
```

```
WHERE 'emergency' != ALL (SELECT unnest(contacts)::contact).type;
```

uid	username	email
2	minh	minh@example.com

JSONB à la place de tableaux

contacts5 Schéma objet-relationnel

```
DROP SCHEMA IF EXISTS contacts5 CASCADE;
CREATE SCHEMA contacts5;
SET search_path TO contacts5;

CREATE TABLE users (
    uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    username VARCHAR(15) NOT NULL,
    email TEXT NOT NULL,
    firstname TEXT,
    lastname TEXT,
    contacts jsonb
);
```

```
INSERT INTO users (username, email, contacts)
VALUES ('denis', 'denis.rinfret@example.com',
        '{"emergency": "help@example.com", "friend": "minh@example.com"}'),
        ('minh', 'minh@example.com',
        '{"family": "contact@example.com", "colleague": "ha@example.com"}');
```

```
SELECT uid, contacts FROM users;
```

uid	contacts
1	{"friend": "minh@example.com", "emergency": "help@example.com"}
2	{"family": "contact@example.com", "colleague": "ha@example.com"}

Trouver les utilisateurs avec au moins un contact d'urgence

```
SELECT uid, username, email
FROM users
WHERE 'emergency' IN (SELECT jsonb_object_keys(contacts));
```

uid	username	email
1	denis	denis.rinfret@example.com

```
SELECT uid, username, email
FROM users
WHERE contacts ? 'emergency';
```

uid	username	email
1	denis	denis.rinfret@example.com

```
SELECT uid, username, email, contacts->'emergency' AS emergency
FROM users
WHERE contacts ? 'emergency';
```

uid	username	email	emergency
1	denis	denis.rinfret@example.com	help@example.com

Trouver les utilisateurs sans contacts d'urgence

```

SELECT uid, username, email
FROM users
WHERE NOT(contact ? 'emergency');

```

uid	username	email
2	minh	minh@example.com

Schéma relationnel avec les types de contact

contacts6 Relational Schema

```

DROP SCHEMA IF EXISTS contacts6 CASCADE;
CREATE SCHEMA contacts6;
SET search_path TO contacts6;

CREATE TABLE users (
    uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    username VARCHAR(15) NOT NULL,
    email TEXT NOT NULL,
    firstname TEXT,
    lastname TEXT
);

```

```

CREATE TYPE contact_type AS ENUM ('emergency', 'friend',
                                    'family', 'colleague');

```

```

CREATE TABLE contacts (
    cid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    email TEXT NOT NULL,
    type contact_type NOT NULL,
    uid INTEGER REFERENCES users(uid)
);

```

```

INSERT INTO users (username, email)
VALUES ('denis', 'denis.rinfret@example.com'),

```

```

('minh', 'minh@example.com');

INSERT INTO contacts (email, type, uid)
VALUES ('help@example.com', 'emergency', 1), ('minh@example.com',
'friend', 1),
('contact@example.com', 'colleague', 2), ('ha@example.com',
'family', 2);

```

```
SELECT * FROM users;
```

uid	username	email	firstname	lastname
1	denis	denis.rinfret@example.com	None	None
2	minh	minh@example.com	None	None

```
SELECT * FROM contacts;
```

cid	email	type	uid
1	help@example.com	emergency	1
2	minh@example.com	friend	1
3	contact@example.com	colleague	2
4	ha@example.com	family	2

```

SELECT *
FROM users u
INNER JOIN contacts c ON u.uid = c.uid;

```

uid	username	email	firstname	lastname	cid	email_1	type	uid_1
1	denis	denis.rinfret@example.com	None	None	1	help@example.com	emergency	1
1	denis	denis.rinfret@example.com	None	None	2	minh@example.com	friend	1
2	minh	minh@example.com	None	None	3	contact@example.com	colleague	2
2	minh	minh@example.com	None	None	4	ha@example.com	family	2

Trouver les utilisateurs sans contacts d'urgence

```
SELECT uid, username, email
FROM users
WHERE uid NOT IN (SELECT uid
                   FROM contacts
                   WHERE type = 'emergency');
```

uid	username	email
2	minh	minh@example.com

```
SELECT users.uid, username, email
FROM users LEFT JOIN (SELECT uid
                      FROM contacts
                      WHERE type = 'emergency') AS temp
                      ON users.uid = temp.uid
WHERE temp.uid IS NULL;
```

uid	username	email
2	minh	minh@example.com

Comment traitons-nous les relations plusieurs à plusieurs ?

- Dans une base de données relationnelle, nous avons besoin d'une table supplémentaire entre les 2 tables.
 - Par exemple, nous avons besoin d'une table entre "Users" et "Contacts", contenant des clés étrangères pour les ID d'utilisateur et les ID de contact.
- Comment récupérons-nous toutes les données ?
 - Avec 2 jointures.
- Qu'en est-il des autres modèles de données ?
 - Avec des tableaux ?
 - Avec JSONB ?

Il n'y a pas de solutions miracles pour les relations plusieurs à plusieurs

Diagrammes d'état

Un diagramme d'état, également connu sous le nom de diagramme d'état-transition, est un type de diagramme comportemental dans la modélisation orientée objet. Il montre le cycle de vie d'un objet et comment l'état de l'objet change en réponse à des événements internes ou externes.

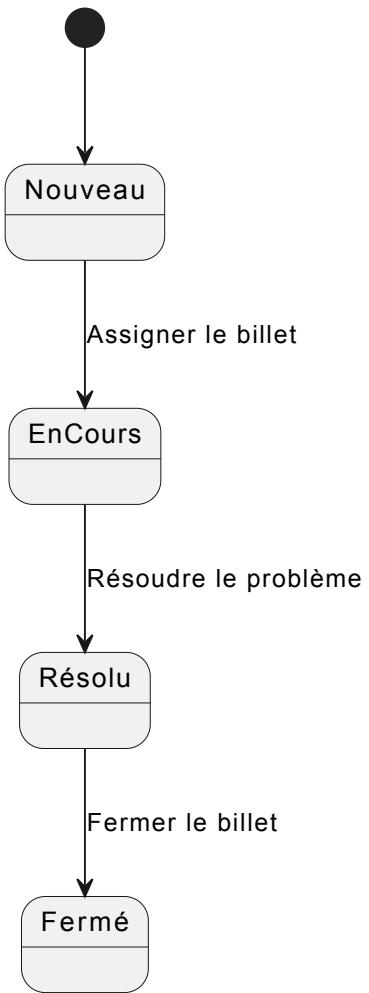
Le diagramme d'état est composé de :

- **Les États:** Ils représentent les conditions distinctes dans le cycle de vie d'un objet. Un objet change d'état en réponse à un événement.
- **Les Transitions:** Elles représentent le passage d'un état à un autre. Elles sont déclenchées par des événements et indiquent un changement d'état.
- **Les Événements:** Ce sont des occurrences spécifiques qui déclenchent des transitions. Ils peuvent être initiés par des acteurs externes ou par des conditions internes ou temporelles.

Les diagrammes d'état sont très utiles lors de la conception d'un logiciel pour comprendre le comportement complexe d'un objet particulier, où l'objet peut être dans l'un des nombreux états et pourrait changer d'état en réponse à un nombre de signaux ou d'événements différents. Les événements ne sont pas toujours représentés dans les diagrammes d'état.

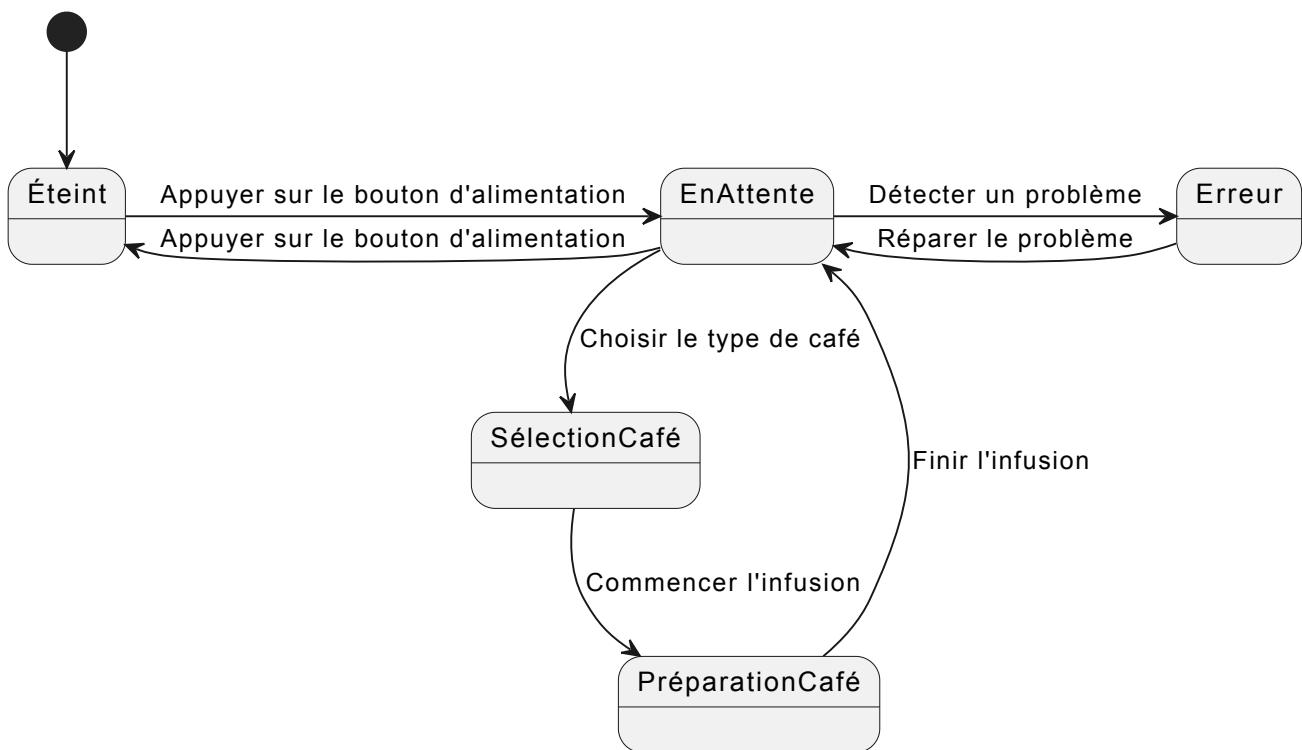
Exemple 1 : le cycle de vie d'un billet dans un système de suivi des problèmes

- Les états : *Nouveau, En cours, Résolu, Fermé.*
- Les transitions (actions, événements) qui font passer le billet d'un état à un autre, par exemple : *Assigner le billet, Résoudre le problème, Fermer le billet.*



Exemple 2 : logiciel pour une machine à café.

- Les états pourraient être : *Éteint, En attente, Sélection du café Préparation du café, Erreur.*
- Les événements pourraient être : *Appuyer sur le bouton d'alimentation, Choisir le type de café, Commencer l'infusion, Finir l'infusion, Détecter un problème, Réparer le problème.*



Différences entre les diagrammes d'activités et les diagrammes d'états

Les diagrammes d'activités et les diagrammes d'états sont tous deux des types de diagrammes de comportement utilisés en UML. Cependant, ils ont des utilisations et des buts légèrement différents.

1. Diagramme d'état (State Diagram)

- Un diagramme d'état est utilisé pour représenter le comportement d'une seule entité (classe, composant, sous-système, etc.).
- Il décrit les différents états d'une entité et comment elle transite d'un état à un autre en réponse à des événements.
- Les diagrammes d'état sont davantage axés sur l'état interne d'une entité et sur la manière dont la modification de cet état provoque des changements de comportement pour cette entité.

2. Diagramme d'activité (Activity Diagram)

- Un diagramme d'activité est utilisé pour modéliser le flux de travail ou le processus d'affaires pour plusieurs entités.

- Il décrit l'ordre séquentiel des actions et les conditions qui dictent le flux entre ces actions.
- Les diagrammes d'activité ont une vision globale et se concentrent sur le flux de contrôle et d'objet à travers la modélisation des activités passées de l'état initial à l'état final. Par exemple, le diagramme d'activités d'un système de commande peut inclure des activités comme "sélectionner des articles", "ajouter au panier", "vérifier", etc.

Dans l'ensemble, les diagrammes d'activité ont une portée plus large, tandis que les diagrammes d'état sont plus concentrés sur un seul objet ou une seule entité.

Références

Cours d'UML (<https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-etats-transitions>) PlantUML (<https://plantuml.com/en/state-diagram>)

Diagrammes de séquences

Un diagramme de séquences, parfois appelé un diagramme de messages, est un type de diagramme utilisé en ingénierie logicielle dans le cadre de la modélisation orientée objet.

Faisant partie du langage de modélisation unifié (UML), le diagramme de séquences est utilisé pour visualiser et documenter les interactions entre les objets dans un système au cours d'un processus ou d'une période de temps spécifiée.

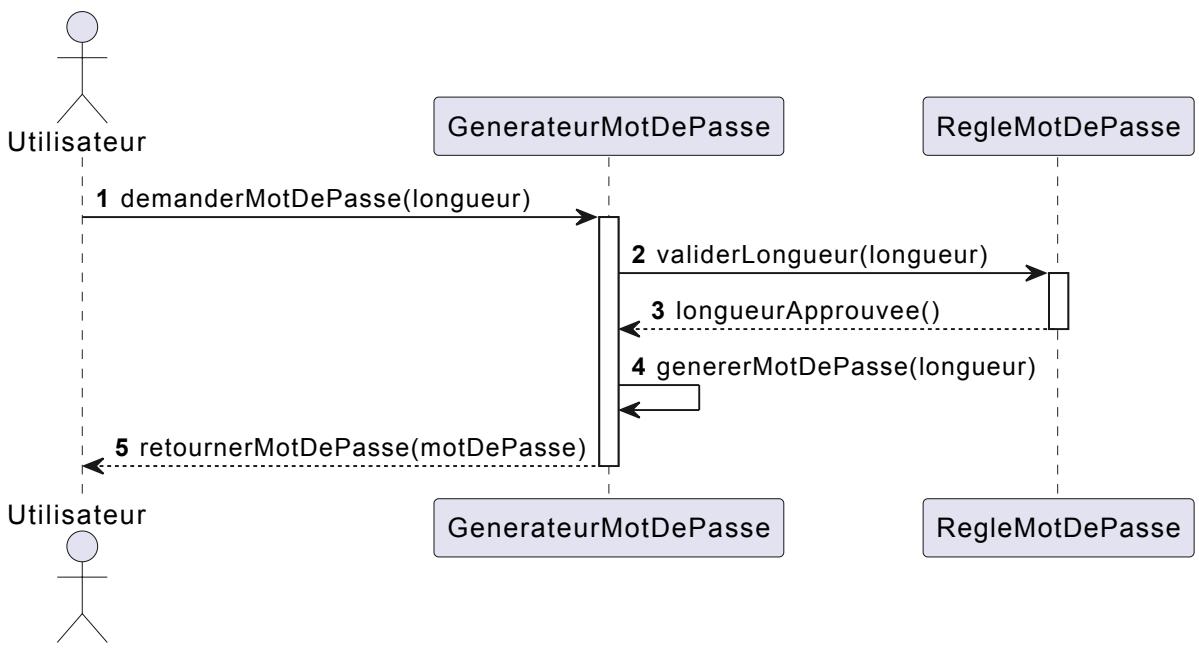
Voici quelques principaux éléments qui composent un diagramme de séquences :

- **Lignes de vie:** Chaque ligne de vie représente un objet ou un acteur dans le processus. C'est généralement une instance d'une classe.
- **Barres d'activation:** Une barre d'activation sur une ligne de vie indique qu'un objet participe à une interaction.
- **Messages:** Les messages, représentés par des flèches entre les lignes de vie, indiquent une interaction ou une communication entre les objets. L'ordre de ces messages est indiqué de haut en bas.
- **Retours:** Ces flèches représentent la réponse d'une interaction.

Un diagramme de séquences est très utile pour comprendre comment les différents éléments d'un système interagissent pour accomplir une tâche spécifique, l'ordre dans lequel ces interactions se produisent, et comment les objets se passent l'information (et dans quelle séquence).

Exemple : Génération de mots de passe

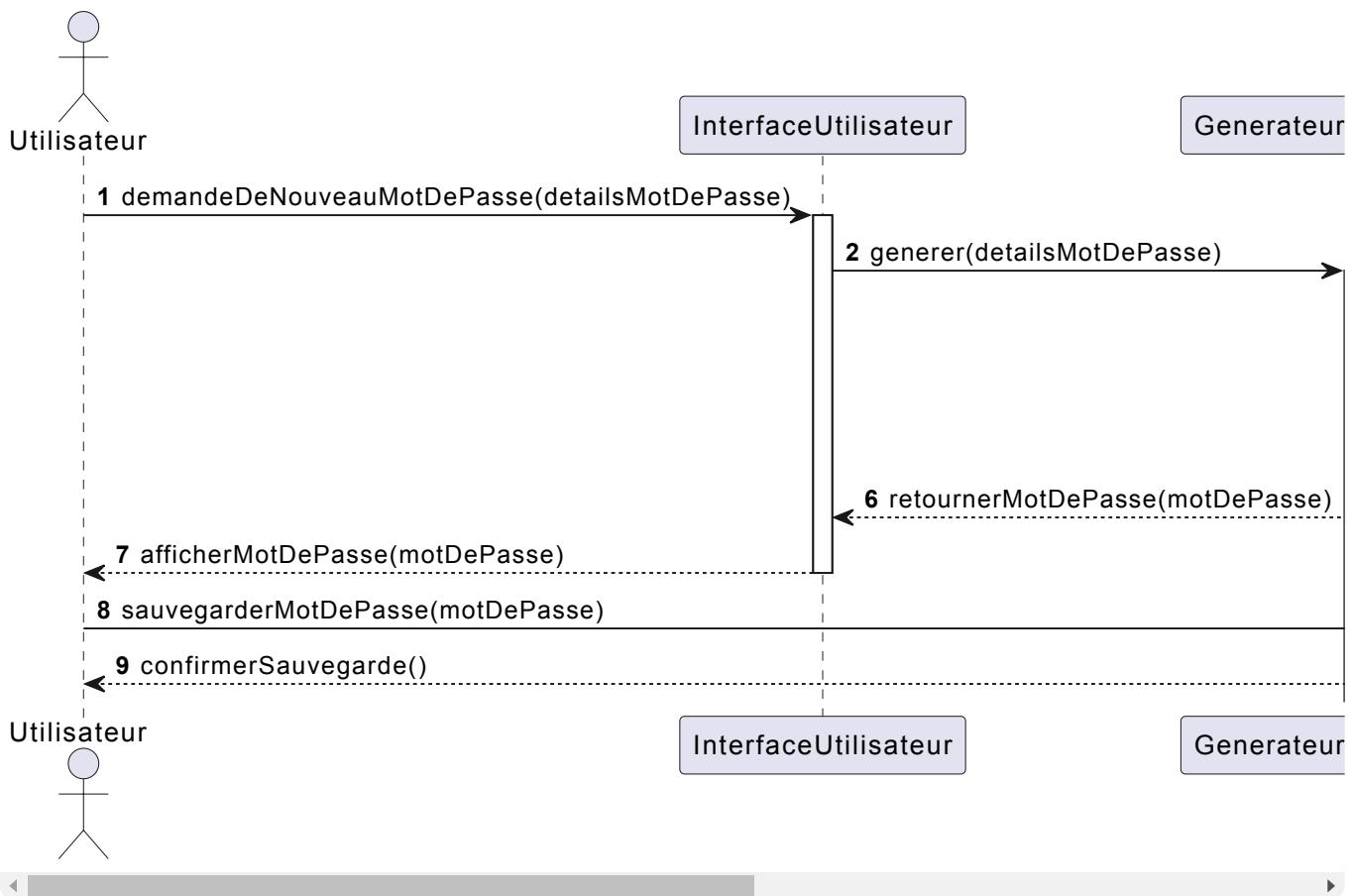
Version courte



Dans cet exemple, l'utilisateur déclenche le processus en demandant un mot de passe d'une certaine longueur au générateur de mots de passe (*GenerateurMotDePasse*). Le générateur de mots de passe valide alors la longueur demandée avec une règle de mots de passe (*RegleMotDePasse*) avant de générer le mot de passe et de le renvoyer à l'utilisateur.

Remarquez que le diagramme de séquence met en évidence l'ordre des interactions et qui initie chaque action. Par exemple, c'est l'utilisateur qui déclenche le processus, et le générateur de mots de passe qui initie la validation de la longueur.

Version longue



Références

1. Cours d'UML Chapitre 7 (<https://laurent-audibert.developpez.com/Cours-UML/?page=diagrammes-interaction#L7-3>)
2. PlantUML (<https://plantuml.com/en/sequence-diagram>)

Patrons de conception

Les *design patterns* ou **patrons de conception** en français, sont des solutions générales réutilisables à des problèmes communs rencontrés dans la conception logicielle. Ce sont essentiellement des *modèles* ou des *schémas* éprouvés qui peuvent aider à résoudre certains problèmes de conception spécifiques.

Ces patrons fournissent un moyen standard de résoudre des problèmes connus et récurrents, ce qui permet aux développeurs de ne pas réinventer la roue à chaque fois qu'ils rencontrent ces problèmes.

Ils peuvent être classés en trois catégories principales :

- 1. Patrons de création:** Ces patrons sont conçus pour aider à la création d'objets tout en cachant les détails de leur création. Ils encapsulent la connaissance sur quels objets concrets le système utilise, mais ils cachent également comment ces instances sont créées et mises ensemble. Des exemples incluent le *Singleton*, *Factory Method*, *Abstract Factory*, *Prototype* et *Builder*.
- 2. Patrons de structure:** Ces patrons concernent la composition de classes ou d'objets pour former de nouvelles structures plus grandes. Ils expliquent comment assembler les objets et les classes pour former des structures plus grandes. Des exemples incluent le *Adapter*, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight* et *Proxy*.
- 3. Patrons de comportement:** Ces patrons se concentrent sur l'interaction entre les objets, comment ils communiquent et comment certains objets sont contrôlés par d'autres. Des exemples incluent le *Chain of Responsibility*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer* (ou *Publish/Subscribe*), *State*, *Strategy*, *Template Method* et *Visitor*.

Les *design patterns* sont un outil de communication puissant entre les développeurs. En disant, par exemple, "Utilisons le patron Strategy ici", vous communiquez beaucoup d'informations sur la manière dont vous envisagez de résoudre un problème de conception.

Principes OO

Clean Code : A Handbook of Agile Software Craftsmanship

Ce livre influent, écrit par Robert C. Martin, qui est également connu sous le nom de "Uncle Bob". L'objectif de ce livre est d'enseigner aux développeurs de logiciels les principes de l'écriture de logiciels de haute qualité. Voici un bref résumé de certains des concepts clés présentés dans le livre :

1. **Code propre:** L'auteur définit le "code propre" comme un code qui a été écrit par quelqu'un qui se soucie réellement de son travail. Pour lui, un bon code peut être lu et amélioré par d'autres développeurs, et a une structure claire.
2. **Noms significatifs:** Les noms pour les fonctions, les variables et les classes doivent être clairs et précis. Ils devraient révéler leur intention et ne laisser aucune place à l'ambiguïté.
3. **Fonctions:** Les fonctions doivent être petites et ne faire qu'une seule chose. Elles devraient seulement avoir un niveau d'abstraction et les commentaires doivent être évités autant que possible. L'auteur explique en détail comment créer des fonctions efficaces.
4. **Commentaires:** En règle générale, les commentaires devraient être évités. Un bon code est auto-documenté. Si vous devez écrire des commentaires pour expliquer votre code, il serait préférable de le réécrire.
5. **Formatage:** Il est essentiel d'avoir un bon format pour votre code. Un bon format facilite la lisibilité et la compréhension du code.
6. **Gestion des erreurs:** L'auteur suggère que les erreurs doivent être gérées correctement en utilisant des exceptions plutôt que des codes d'erreur retournés.
7. **Répétition de code:** La répétition de code est une mauvaise habitude qui peut rendre votre code ennuyeux et difficile à maintenir. Utiliser des abstractions et des méthodes pour éviter la duplication du code.
8. **Principes SOLID:** Il explique également certains principes de la conception orientée objet, tels que les principes SOLID : *Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation et Dependency Inversion*.
9. **Code smells et refactorisation:** Le livre présente également différentes "odeurs de code" (*code smells*) qui indiquent d'éventuels problèmes de conception du code et montre

comment ces problèmes peuvent être résolus par le refactorisation.

En résumé, "Clean Code" est une ressource précieuse pour tout développeur de logiciels cherchant à améliorer ses compétences en codage et à écrire des logiciels de meilleure qualité.

SOLID

<https://en.wikipedia.org/wiki/SOLID> (<https://en.wikipedia.org/wiki/SOLID>)

[https://fr.wikipedia.org/wiki/SOLID_\(informatique\)](https://fr.wikipedia.org/wiki/SOLID_(informatique))
([https://fr.wikipedia.org/wiki/SOLID_\(informatique\)](https://fr.wikipedia.org/wiki/SOLID_(informatique)))

SOLID est un acronyme pour un ensemble de cinq principes de conception de logiciels orientés objet qui aident à rendre les systèmes de logiciels plus compréhensibles, flexibles et maintenables. Voici ce que chaque lettre de l'acronyme SOLID signifie :

1. **S - Principe de Responsabilité Unique (Single Responsibility Principle):** Ce principe stipule qu'une classe ou un module devrait avoir une, et seulement une, raison de changer. En d'autres termes, une classe ou un module devrait avoir une seule responsabilité ou tâche.
2. **O - Principe ouvert/fermé (Open/Closed Principle):** Ce principe stipule que les entités logicielles (classes, modules, fonctions, etc.) devraient être ouvertes à l'extension, mais fermées à la modification. Cela signifie que vous devriez être capable d'ajouter de nouvelles fonctionnalités à une entité sans modifier son code source.
3. **L - Principe de substitution de Liskov (Liskov Substitution Principle):** Ce principe stipule que dans un programme, les objets d'une superclasse devraient pouvoir être remplacés par des objets d'une sous-classe sans affecter l'exactitude de ce programme.
4. **I - Principe de ségrégation d'interface (Interface Segregation Principle):** Ce principe stipule que les clients ne devraient pas être forcés de dépendre des interfaces qu'ils n'utilisent pas.
5. **D - Principe d'inversion de dépendance (Dependency Inversion Principle):** Ce principe stipule que les modules de haut niveau ne devraient pas dépendre des modules de bas niveau. Les deux devraient dépendre des abstractions.

Suivre ces principes peut aider à éviter les mauvaises pratiques de conception, à réduire le couplage dans le code, à rendre le code plus facile à maintenir et à améliorer la lisibilité et la reproductibilité du code.

Responsabilité unique (Single responsibility principle SRP)

Une classe, une fonction ou une méthode doit avoir une et une seule responsabilité.

Selon le principe de responsabilité unique, une classe, un module ou une fonction ne devrait avoir qu'une seule raison de changer. Autrement dit, chaque entité logicielle doit être responsable d'un seul aspect particulier de la fonctionnalité du logiciel.

Pour donner un exemple, supposons que vous ayez un objet dans votre application qui gère la logique de manipulation des utilisateurs (c'est-à-dire, l'ajout, la suppression et la mise à jour des utilisateurs). Si ce même objet est aussi responsable de l'affichage des utilisateurs à l'écran, alors il a deux raisons de changer : une raison technique (modification du code pour manipuler les utilisateurs différemment) et une raison esthétique (modification du code pour changer l'affichage des utilisateurs à l'écran). C'est une violation du principe SRP.

Ce principe encourage à séparer les préoccupations pour améliorer la lisibilité et la maintenabilité

Ouvert/fermé (Open/closed principle)

Une entité applicative (classe, fonction, module ...) doit être fermée à la modification directe, mais ouverte à l'extension.

Le principe Ouvert/fermé, souvent abrégé en OCP (Open/Closed Principle), est un concept important en programmation orientée objet, et est l'un des cinq principes fondamentaux du SOLID, un acronyme pour un ensemble de principes de conception orientée objet conçus pour rendre le code plus compréhensible, flexible et maintenable.

Selon le principe Ouvert/fermé, souvent abrégé en OCP (Open/Closed Principle), les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes à l'extension, mais fermées à la modification. Cela signifie que vous devriez pouvoir ajouter de nouvelles fonctionnalités ou comportements à une entité sans avoir à modifier son code source.

L'intention est de réduire le risque de nouvelles erreurs et réduire les dépendances dans le système. De cette façon, lorsque nous ajoutons de nouvelles fonctionnalités ou changeons le comportement du système, nous ajoutons du nouveau code (extension), nous n'avons pas besoin de modifier le code existant (fermé pour modification).

Un exemple de la manière d'appliquer l'OCP est l'utilisation de **polymorphisme** où la classe de base reste la même et de nouvelles fonctionnalités sont ajoutées en créant de nouvelles classes qui héritent de la classe de base.

Substitution de Liskov (Liskov substitution principle)

Une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela modifie la cohérence du programme.

Ce principe a été introduit par Barbara Liskov en 1987 lors d'une conférence. Le principe stipule que si une classe B hérite d'une classe A, alors nous devrions être capables de remplacer A par B sans affecter le comportement de notre programme. C'est-à-dire, un objet de type superclasse peut être remplacé par un objet de type sous-classe sans perturber l'exactitude ou le comportement du programme.

Par exemple, considérons une classe Oiseau qui a une méthode voler. Maintenant, considérons une sous-classe Pingouin, qui est un type d'oiseau, mais qui ne peut pas voler. Selon le principe de substitution de Liskov, nous devrions pouvoir utiliser un objet Pingouin là où nous utilisons un objet Oiseau dans notre programme. Cependant, si notre programme attend que les oiseaux puissent voler, alors cette substitution causerait un problème, car les pingouins ne peuvent pas voler. C'est une violation du principe de substitution de Liskov.

En substance, le principe de substitution de Liskov encourage à assurer que les sous-classes soient absolument substituables par leurs superclasses sans altérer la logique du programme. Solliciter ce principe lors de la conception de logiciels aide à garantir la polymorphie et augmente la réutilisabilité des modules logiciels.

Ségrégation des interfaces (Interface segregation principle)

Préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale.

L'ISP stipule que " les clients ne devraient pas être forcés de dépendre des interfaces qu'ils n'utilisent pas ". En termes simples, cela signifie que les grandes interfaces monolithiques doivent être évitées en faveur de plus petites, plus spécifiques qui sont plus facilement gérables et utilisables par les classes.

Pour donner un exemple, supposons que vous ayez une interface Imprimante avec les méthodes imprimer, scanner et faxer. C'est bien pour une imprimante multifonction qui peut faire tout cela, mais qu'en est-il d'une imprimante de base qui ne peut qu'imprimer ? Selon l'ISP, il serait préférable d'avoir des interfaces séparées pour chaque fonctionnalité, de sorte que l'imprimante de base n'ait pas besoin de dépendre (ou d'implémenter de manière artificielle) des méthodes scanner et faxer qu'elle ne peut pas utiliser.

L'avantage de l'ISP est qu'il réduit le couplage entre les composants et améliore leur modularité, facilitant ainsi la maintenance, la réutilisation et l'évolution du code.

Inversion des dépendances (Dependency inversion principle)

Il faut dépendre des abstractions, pas des implémentations

Le DIP stipule que :

1. Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre des abstractions.
2. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Par "module de haut niveau", on entend une classe (ou un ensemble de classes) qui est à un niveau d'abstraction élevé, c'est-à-dire plus orientée vers les objectifs métier. Et par "module de bas niveau", on entend une classe (ou un ensemble de classes) qui est à un niveau d'abstraction bas, c'est-à-dire plus orientée vers les détails techniques ou l'infrastructure.

En d'autres termes, plutôt que de laisser les classes de haut niveau (plus proches des opérations métier) dépendre directement des classes de bas niveau (plus proches des opérations techniques), les deux devraient dépendre d'interfaces abstraites. Ainsi, si les détails de la mise en œuvre d'une classe de bas niveau doivent changer, cela n'a aucun impact sur les classes de haut niveau.

L'avantage de ce principe est qu'il contribue à définir un système fortement cohérent mais faiblement couplé, ce qui rend le code plus flexible, plus facile à tester, à maintenir et à faire évoluer.

Autres Principes

DRY / WET

DRY (Don't Repeat Yourself)

Opposé de DRY: WET (Write Everything Twice)

https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

(https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)

https://fr.wikipedia.org/wiki/Ne_vous_répétez_pas

(https://fr.wikipedia.org/wiki/Ne_vous_répétez_pas)

DRY signifie **Don't Repeat Yourself** (*Ne vous répétez pas* en français). C'est un principe de développement logiciel qui vise à réduire la répétition de code. L'idée est que chaque morceau de logique doit avoir un emplacement unique et définitif dans le système. Si vous découvrez que vous écrivez le même code à plusieurs reprises, vous devriez chercher un moyen de le centraliser et de le réutiliser, plutôt que de le dupliquer. Le respect de ce principe rend le code plus lisible, plus efficace et plus facile à maintenir.

À l'opposé, **WET** signifie **Write Everything Twice** (*Écrire tout deux fois* en français) ou **We Enjoy Typing** (*Nous aimons taper* en français). C'est souvent utilisé de manière humoristique pour décrire du code qui viole le principe DRY, c'est-à-dire du code qui a beaucoup de répétitions inutiles.

Il est important de noter que le principe DRY n'encourage pas à éviter toute duplication à tout prix. Parfois, essayer d'éliminer toute duplication peut conduire à une conception trop complexe et difficile à comprendre. Il s'agit plutôt d'éviter la duplication de la logique du système. Une duplication apparemment similaire peut être acceptable si les motivations et les usages derrière elles sont différents.

KISS

https://en.wikipedia.org/wiki/KISS_principle (https://en.wikipedia.org/wiki/KISS_principle)

https://fr.wikipedia.org/wiki/Principe_KISS (https://fr.wikipedia.org/wiki/Principe_KISS)

Keep it simple, stupid

En développement logiciel, **KISS** est un acronyme pour **Keep It Simple, Stupid** (*Garde ça simple, idiot* en français). Ce principe de conception encourage la simplicité dans la conception et l'écriture du code.

L'idée est d'éviter la complexité inutile. Par complexité inutile, on entend généralement des solutions *sur-conçues* (*over-engineered*, ou trop élaborées), c'est-à-dire qui utilisent des conceptions plus complexes qu'il n'est réellement nécessaire pour résoudre le problème à portée de main. Une solution plus complexe peut être plus difficile à comprendre, à tester et à maintenir, et est également plus susceptible d'introduire des bugs.

Dans la pratique, suivre le principe KISS pourrait signifier de privilégier une conception plus simple même si elle est un peu moins élégante, choisir une solution brute, mais efficace plutôt qu'une solution sophistiquée, mais délicate, ou préférer des structures de code faciles à comprendre et à parcourir à des structures de code plus "intelligentes" mais potentiellement déroutantes.

Il est important de noter que KISS ne signifie pas nécessairement choisir la solution la plus simple à court terme, mais plutôt la solution qui reste simple sur toute la durée de vie du code, y compris sa maintenance et son évolution futures.

Loi de Demeter (LoD)

Principe de connaissance minimale

https://en.wikipedia.org/wiki/Law_of_Demeter
(https://en.wikipedia.org/wiki/Law_of_Demeter)

https://fr.wikipedia.org/wiki/Loi_de_Déméter (https://fr.wikipedia.org/wiki/Loi_de_Déméter)

La Loi de Demeter est un principe de conception en programmation orientée objet qui encourage à réduire les dépendances entre les objets. Aussi connu sous le nom de principe de *moindre connaissance*, il stipule qu'un objet ne devrait pas connaître les détails internes des autres objets.

Formellement, la Loi de Demeter stipule que la méthode M d'un objet O ne devrait invoquer que les méthodes des objets suivants :

- O lui-même
- un objet passé en paramètre à M
- un objet créé par M
- un objet composant de O

En d'autres termes, si vous avez un objet A qui contient un objet B, qui à son tour contient un objet C, la Loi de Demeter suggère qu' A ne devrait pas accéder directement à C. Si vous avez une méthode dans A qui a besoin d'interagir avec C, vous devriez avoir une méthode dans B qui sert d'intermédiaire.

L'idée derrière la loi est de minimiser le couplage entre les objets, en évitant qu'un objet ne connaisse trop les détails internes des autres objets. Cela rend le système dans son ensemble plus modulaire et robuste, car les objets peuvent être modifiés sans affecter grandement les autres objets.

Il est important de noter que la Loi de Demeter est un principe directeur plutôt qu'un ensemble de règles strictes à suivre. Il y aura des situations où il est approprié de rompre cette loi dans l'intérêt d'autres aspects de la conception.

Tell Don't ask

<https://martinfowler.com/bliki/TellDontAsk.html>
(<https://martinfowler.com/bliki/TellDontAsk.html>)

Tell-Don't-Ask is a principle that helps people remember that object-orientation is about bundling data with the functions that operate on that data. It reminds us that rather than asking an object for data and acting on that data, we should instead tell an object what to do.

Tell-Don't-Ask est un principe qui aide les gens à se rappeler que la programmation orientée objet consiste à regrouper des données avec les fonctions qui opèrent sur ces données. Il

nous rappelle qu'au lieu de demander à un objet des données et d'agir sur ces données, nous devrions plutôt dire à un objet ce qu'il doit faire.

Le principe "Tell, Don't Ask" (Dis-le, ne le demande pas) est un principe de conception en programmation orientée objet qui encourage à dire à un objet ce qu'il doit faire, plutôt que de demander à l'objet ses données et de manipuler ces dernières.

Essentiellement, cela signifie que vous devriez essayer d'éviter de demander à un objet son état, de prendre cet état et de faire quelque chose avec lui. À la place, l'objet devrait être responsable de faire quelque chose avec son propre état.

Voici un exemple pour illustrer ce principe :

Considérons deux classes, Car et Driver. Une approche qui **ne suit pas** le principe "Tell, Don't Ask" pourrait ressembler à ceci :

```
public class Driver {  
    public void drive() {  
        Car car = new Car();  
        if (car.getFuel() > 0) {  
            car.moveForward();  
        }  
    }  
}
```

Le problème ici est que la classe Driver doit demander à la voiture combien de carburant elle a, puis décider quoi faire en fonction de cette information.

Une approche qui suit le principe "Tell, Don't Ask" pourrait ressembler à ceci :

```
public class Driver {  
    public void drive() {  
        Car car = new Car();  
        car.moveForward();  
    }  
}  
  
public class Car {  
    public void moveForward() {  
        if (this.fuel > 0) {  
            // code pour avancer  
        }  
    }  
}
```

```
}
```

L'avantage de cette approche est qu'elle décentralise la logique et la responsabilité, rendant le code plus modulaire et facile à maintenir et à comprendre.

Orthogonalité

Le principe d'orthogonalité est un concept de conception logicielle qui stipule que les composants d'un système sont orthogonaux s'ils peuvent être modifiés indépendamment les uns des autres. Autrement dit, changer quelque chose dans une partie du système n'affecte pas d'autres parties du système.

L'idée vient de l'orthogonalité en mathématiques, où deux vecteurs sont orthogonaux s'ils sont perpendiculaires l'un à l'autre, signifiant qu'ils n'ont pas d'impact l'un sur l'autre.

Quand ce principe est appliqué à la conception logicielle, il a pour but de minimiser les dépendances entre les différents modules ou composants. Les avantages de ce principe sont :

- **Isolation des erreurs:** une erreur dans un composant ne se propage pas aux autres.
- **Facilité de compréhension:** chaque composant peut être compris indépendamment des autres.
- **Facilité de modification:** une modification d'un composant n'affecte pas les autres, rendant le développement et la maintenance du système plus faciles.

Par exemple, si vous développez une application avec une interface utilisateur, une base de données, et une logique métier, ces trois éléments devraient être conçus de manière à ce qu'ils puissent être développés et modifiés indépendamment les uns des autres. Ils communiqueraient par le biais d'interfaces clairement définies, sans avoir besoin de connaître les détails de mise en œuvre des autres.

Éviter l'optimisation prématuée

L'optimisation prématuée est la pratique de l'optimisation micro performante d'un code avant de comprendre clairement quels sont les véritables goulets d'étranglement de performance.

"*Éviter l'optimisation prématuée*" est un principe communément cité en programmation, attribué à Donald Knuth, un éminent informaticien. La citation complète de Knuth est :

"Nous devrions oublier les petites économies, disons environ 97% du temps : l'optimisation prématuée est la racine de tous les maux. Pourtant, nous ne devrions pas manquer nos

chances dans ce 3% critique."

Ce principe suggère que la majorité de l'effort en optimisation devrait être consacrée aux zones d'un programme où les gains de performance auront un impact significatif. Ces zones ne peuvent être efficacement identifiées qu'après que les problématiques de performances ont été clairement définies, souvent via le *monitoring* ou le *profilage* du code en question.

Cela aide à éviter de dépenser du temps sur des optimisations minuscules qui n'affectent que très peu les performances globales, tout en rendant potentiellement le code plus complexe et difficile à maintenir. Cela ne signifie pas que les performances ne sont pas importantes, mais qu'elles doivent être abordées de manière systématique et informée.

Règle des 5 lignes

La "règle des 5 lignes" en développement logiciel est une directive heuristique suggérant que les fonctions ou méthodes doivent idéalement être courtes, généralement dans les cinq lignes de code. L'idée derrière cette règle est de promouvoir la lisibilité, la maintenabilité et la simplicité du code.

La logique derrière la règle des 5 lignes inclut :

1. **Lisibilité:** Des fonctions plus courtes sont plus faciles à comprendre d'un coup d'œil, ce qui rend plus simple pour les développeurs de comprendre ce que fait la fonction et comment elle atteint son but.
2. **Maintenabilité:** Les petites fonctions sont généralement plus faciles à maintenir et à modifier. Lorsque les fonctions sont concises, il est plus facile d'isoler et de corriger les bugs, d'ajouter de nouvelles fonctionnalités ou de refactoriser le code sans affecter involontairement d'autres parties du système.
3. **Testabilité:** Les fonctions courtes sont souvent plus faciles à tester, car elles ont tendance à avoir moins de chemins d'exécution et de dépendances. Cela rend plus simple d'écrire des tests unitaires qui couvrent tous les scénarios possibles.
4. **Réduction de la complexité:** Garder les fonctions courtes peut aider à réduire la complexité globale de la base de code. Les fonctions complexes comportant de nombreuses lignes de code sont plus difficiles à comprendre et sont souvent le signe que la fonction fait trop de choses et devrait être décomposée en pièces plus petites et plus gérables.
5. **Amélioration de la réutilisabilité:** Les fonctions courtes sont plus susceptibles d'être réutilisées dans différentes parties de la base de code car elles ont tendance à être plus

ciblées et ont une seule responsabilité. Cela peut conduire à un code plus modulaire et plus facile à maintenir en général.

Il est important de noter que la règle des 5 lignes est une directive plutôt qu'une règle stricte. Bien que viser des fonctions courtes soit généralement bénéfique, il peut y avoir des cas où des fonctions plus longues sont nécessaires ou appropriées. La clé est de prioriser la lisibilité, la maintenabilité et la simplicité dans la conception du code et d'utiliser son discernement lors de la détermination de la longueur appropriée pour une fonction en fonction du contexte spécifique et des exigences du projet logiciel.

Les commentaires de code doivent expliquer pourquoi, pas quoi

Cette règle suggère que les commentaires dans le code devraient **se concentrer sur l'explication du but, de l'intention, ou de la justification derrière le code** plutôt que de décrire ce que fait le code.

Un code bien écrit devrait être auto-explicatif, et les commentaires devraient fournir un contexte ou un éclairage supplémentaire.

Refactorisation de code

La **refactorisation** (*refactoring*) de code est le processus de restructuration du code existant sans en changer le comportement externe afin d'améliorer sa lisibilité, sa maintenabilité et/ou ses performances. Voici quelques règles et directives communes pour la refactorisation de code :

- 1. Règle de trois:** cette règle suggère que si vous vous trouvez en train d'écrire un code similaire pour la troisième fois, il est temps de le refactoriser en une fonction, une méthode ou une classe réutilisable. Elle aide à prévenir la duplication de code et favorise la réutilisabilité.
- 2. Refactoriser tôt, refactoriser souvent:** ce principe préconise de traiter les "odeurs de code" et d'améliorer la qualité du code en continu tout au long du processus de développement, plutôt que de laisser la dette technique s'accumuler. Refactoriser régulièrement de petits morceaux de code peut aider à garder la base de code propre et maintenable.
- 3. Extraire jusqu'à épuisement:** lors de la refactorisation, cette règle conseille aux développeurs de continuer à extraire de plus petites fonctions ou méthodes à partir de plus grandes jusqu'à ce qu'elles ne puissent plus raisonnablement être décomposées davantage. Elle favorise la création de petites unités de code ciblées et réutilisables.

4. **Principes SOLID:** ces principes (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation et Dependency Inversion) fournissent des directives pour écrire un code orienté objet propre, maintenable et extensible. Le respect de ces principes nécessite souvent de refactoriser le code existant pour s'y conformer.
5. **Refactoriser agressivement après avoir ajouté des fonctionnalités:** chaque fois qu'une nouvelle fonctionnalité est ajoutée à la base de code, il est recommandé de passer en revue le code existant et de le refactoriser si nécessaire pour maintenir la qualité et la cohérence du code. Cela aide à prévenir l'accumulation de la dette technique et à s'assurer que la base de code reste propre et maintenable.
6. **Utiliser des outils de refactoring automatisés:** l'utilisation d'outils de refactoring automatisés fournis par des environnements de développement intégrés (IDE) ou des plugins tiers peut rendre le processus de refactoring plus efficace et moins sujet aux erreurs. Ces outils peuvent aider à renommer les variables, à extraire des méthodes et à effectuer d'autres tâches de refactoring courantes avec un minimum d'effort manuel.
7. **Préserver le comportement avec les tests:** lors de la refactoring du code, il est crucial de s'assurer que le comportement du code reste inchangé. L'écriture et la maintenance de tests unitaires exhaustifs peuvent aider à valider que le code refactorisé se comporte comme prévu et à détecter d'éventuels effets secondaires involontaires.

Code smells

En développement logiciel, les "**odeurs de code**" désignent certains modèles ou caractéristiques du code qui peuvent signaler des problèmes plus profonds ou des zones potentiellement améliorables. Ce ne sont pas des bugs en soi, mais plutôt des indicateurs de problèmes de conception ou de violations des bonnes pratiques de codage. Les odeurs de code peuvent rendre le code plus difficile à comprendre, à maintenir et à étendre. Identifier et traiter les odeurs de code par la refactoring peut aider à améliorer la qualité et la maintenabilité du code. Certaines odeurs de code courantes incluent :

1. **Longue Méthode:** Les méthodes ou fonctions qui sont excessivement longues et accomplissent plusieurs tâches. Les longues méthodes sont plus difficiles à comprendre, à tester, et à maintenir.
2. **Grande Classe:** Les classes qui sont trop complexes, avec trop d'attributs et de méthodes. Les grandes classes violent le principe de la responsabilité unique et peuvent être difficiles à gérer.

- 3. Code Dupliqué:** Des fragments de code identiques ou très similaires apparaissant à plusieurs endroits. Le code dupliqué viole le principe Ne vous répétez pas (DRY) et rend la maintenance plus difficile.
- 4. Envie de fonctionnalité:** Lorsqu'une méthode dans une classe accède aux données ou au comportement d'une autre classe plus qu'à sa propre donnée ou son comportement. Cela suggère que la méthode pourrait appartenir à l'autre classe et indique un éventuel problème de conception.
- 5. Obsession Primitive:** Utilisation excessive de types de données primitifs au lieu de créer des classes ou des énumérations personnalisées. L'obsession primitive peut conduire à la duplication de code, à une lisibilité réduite, et à une maintenabilité diminuée.
- 6. Objet Dieu:** Une classe qui sait ou fait trop de choses, devenant souvent un point central de dépendances dans la base de code. Les objets dieux sont difficiles à comprendre, à tester, et à maintenir et peuvent conduire à un code étroitement couplé.
- 7. Commentaires:** Des commentaires excessifs ou trompeurs dans le code, qui peuvent indiquer que le code n'est pas auto-explicatif. Bien que les commentaires puissent être utiles, ils ne doivent pas être utilisés comme substitut à l'écriture de code propre et expressif.
- 8. Regroupements de données:** Lorsque des groupes de champs de données apparaissent ensemble à plusieurs endroits dans la base de code, suggérant qu'ils pourraient être encapsulés dans une classe séparée.
- 9. Déclarations Switch:** Utilisation excessive de déclarations `switch` ou `case`, surtout lorsqu'elles apparaissent à plusieurs endroits dans la base de code. Les déclarations `switch` peuvent rendre le code plus difficile à étendre et à maintenir, violant ainsi le principe ouvert/fermé.
- 10. Intimité Inappropriée:** Classes qui dépendent excessivement les unes des autres en détails internes ou qui ont trop de dépendances. L'intimité inappropriée peut conduire à un code étroitement couplé qui est difficile à refactoriser ou à étendre.

L'identification et le traitement des odeurs de code est une partie essentielle du processus de refactorisation, aidant à améliorer la qualité globale, la lisibilité, et la maintenabilité de la base de code.

Design Patterns : Première Partie

Fabrique (Factory Method, Virtual Constructor)

Intention

Définir une interface pour créer un objet, mais laisser les sous-classes décider quelle classe doit être instanciée. La fabrique permet à une classe de transférer la responsabilité de créer des objets aux sous-classes.

Mise en contexte

Application de dessin, avec la possibilité de dessiner au moins 2 types d'images, des images sur un `Canvas` (voir `ShapesLib`), ou des images de type * AsciiArt*. On peut dessiner des figures géométriques sur ces images.

Détails

1. Menu principal

- Nouveau document
 - Canvas
 - Ascii Art
- Ouvrir
- Fermer
- Sauvegarder

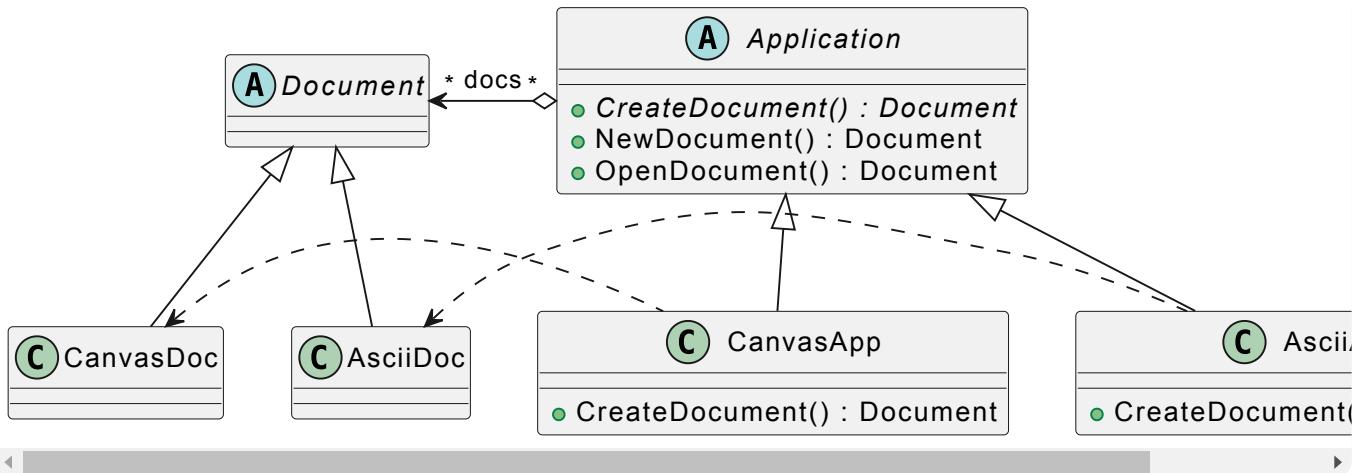
2. L'application doit créer soit une `CanvasApp`, soit une `AsciiApp`

3. Chaque app doit créer un document du bon type, `CanvasDoc` ou `AsciiDoc`

4. La méthode `CreateDoc` est abstraite parce que le type de document créé va dépendre du type d'image choisie

- donc le menu principal doit créer le bon type d'application, mais le reste du code de l'interface graphique va travailler avec les classes abstraites `Application` et `Document`, sans connaître le type exact d'image

- si ce n'était pas le cas, il faudrait faire des `if ... else ...` ou des `switch` partout sur le type d'image pour utiliser les bonnes classes
- en utilisant les classe abstraites, cela permet de faire bon usage du polymorphisme



Fabrique Abstraite (Abstract Factory, Kit)

Intention

Donner une interface pour créer une famille d'objets semblables ou dépendants sans spécifier leur classe concrète.

Mise en contexte

Continuation de l'exemple précédent. Pour créer les figures géométriques (les `Shapes`), doit-on créer les versions `CanvasApp` ou `AsciiApp`? Il faut éviter de faire des `if ... else ...` ou des `switch` partout.

Détails

1. Bouton pour créer une ligne (ou un cercle ou un rectangle...) :

- a-t-on besoin d'un `if` pour savoir quel objet créer exactement ?
- une ligne sur un `Canvas` ou une ligne `Ascii`?

2. Il faut créer 2 `ShapeFactory`, avec des méthodes pour chaque type de `Shape`

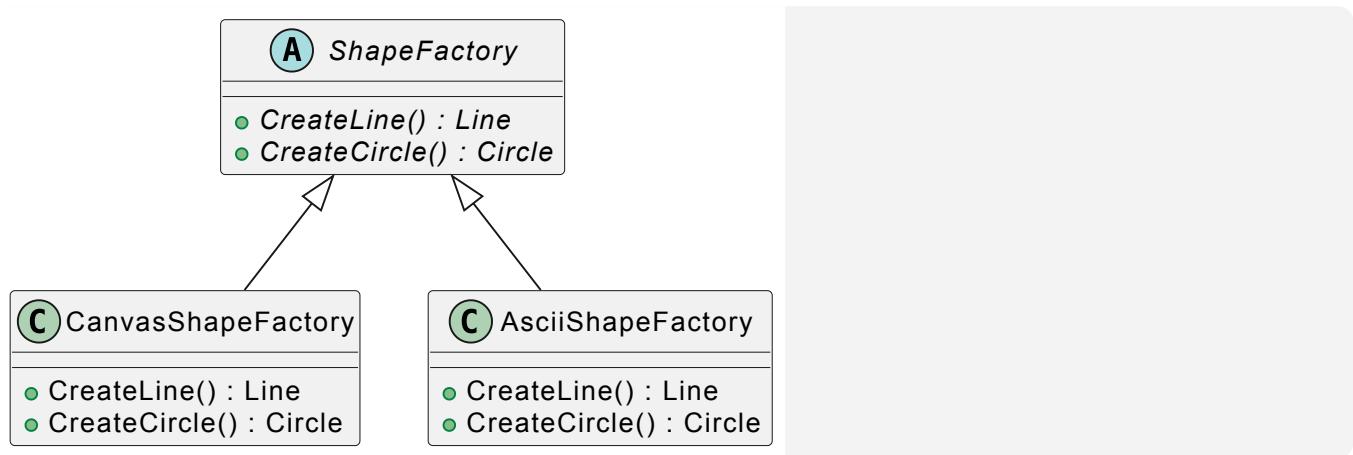
3. `CanvasApp` va créer une `CanvasShapeFactory`

4. `AsciiApp` va créer une `AsciiShapeFactory`

5. La classe abstraite ShapeFactory va être utilisée au niveau de la classe Application

6. Chaque application va avoir une méthode (ou peut-être une propriété) de type ShapeFactory

- App.ShapeFactory.createLine() va retourner une nouvelle ligne, de type Canvas ou Ascii, selon la sous-classe utilisée au moment de la création
- ce code sera utilisé, par exemple, dans le onClick (ou autre gestionnaire d'évènement) d'un bouton, ou d'un item de menu, ou d'un raccourci clavier, ...



Design Patterns : Deuxième Partie

Monteur (Builder)

Intention

Construire un objet complexe étape par étape.

Mise en contexte

Différence avec une fabrique abstraite :

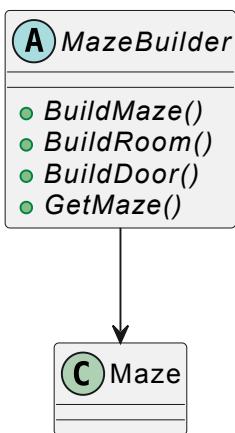
- **Fabrique abstraite:** construire des objets, *simples* ou *complexes*, d'une *même famille*
 - construire des **Shapes** de toutes sortes
- **Monteur:** construire un objet *complexe étape par étape*.
 - Construire des **Shapes complexes** composées de plusieurs autres formes

Détails

Exemple dans GoF (https://en.wikipedia.org/wiki/Design_Patterns): Labyrinthe.

Créer des labyrinthes étape par étape :

1. Construire un labyrinthe vide
2. Ajouter des salles
3. Ajouter des portes
4. etc.



Notes:

1. On pourrait avoir beaucoup plus de méthodes que celles données pour les labyrinthes complexes.
2. Il faudrait évidemment ajuster les signatures de ces méthodes pour y ajouter des paramètres.
3. Il n'est pas obligatoire d'avoir une classe et des méthodes abstraites. On pourrait avoir seulement des classes et méthodes concrètes, avec la possibilité d'avoir des méthodes vides au besoin.

Prototype

Intention

Spécifier les types d'objets à créer en utilisant une instance prototype, et créer de nouveaux objets en copiant ce prototype.

Mise en contexte

Application principale: cloner des objets, comme pour un copier-coller.

Détails

Selon les besoins, on peut effectuer une copie en *surface* ou en *profondeur* d'un objet prototype.

- **Copie de surface (shallow copy):**
 - seulement les références aux attributs ou propriétés sont copiées, donc le clone va partager des références aux mêmes objets avec l'objet original (à condition évidemment

d'avoir des attributs ou propriétés de type référence dans l'objet)

- si les objets partagés sont modifiés, alors l'objet original et ses clones vont voir les modifications

- **Copie en profondeur (*deep copy*):**

- les attributs et propriétés de type référence sont, eux aussi, clonés, donc les références du clone seront vers des objets différents, mais avec le même contenu, que l'objet original
- pour une copie en profondeur complète, il faut effectuer une copie en profondeur sur tous les attributs et propriétés de type référence récursivement

En C#, il a différentes façons de cloner des objets

1. `Object.MemberwiseClone()`: copie de surface

2. `ICloneable.Clone()`: copie de surface ou profonde, selon les besoins

3. Constructeur de copie :

- copie de surface ou profonde, selon les besoins
- un constructeur de copie est un constructeur qui accepte un seul paramètre, du même type que la classe
- exemple : `public Circle(Circle circle)`

Singleton

Intention

S'assurer qu'une classe ne peut avoir qu'une seule instance.

Mise en contexte

Dans certaines applications, il n'est pas logique d'avoir plusieurs instances d'une classe.

Comme dans une application avec interface graphique (mobile ou bureau), il n'est souvent pas logique d'avoir plus d'une fenêtre principale.

Dans d'autres applications, il faut s'assurer de bien gérer les ressources pour éviter d'avoir des applications trop gourmandes en termes de mémoire ou de processeur ou de réseau ...

Comme dans une application qui utilise une base de données, on ne peut avoir qu'un seul

pool de connexions à la base de données. Avoir plusieurs pools de connexions serait plus difficile à gérer et consommerait plus de ressources.

Ou il faut bien synchroniser l'accès à une resource partagée pour éviter de la corrompre. Les applications avec plusieurs fils d'exécution (avec des *thread* ou avec des *asynch/await*) ont normalement besoin de partager certaines structures de données en mémoire, et de synchroniser leur utilisation de ces structures.

Détails

1. On peut mettre tous les constructeurs d'une classe en mode d'accès privé (ou potentiellement protégé), ce qui empêchera d'autres classes de créer des instances de cette classe.
2. Ensuite, on ajoute un attribut privé statique pour contenir la référence à l'unique instance
3. La méthode statique `GetInstance` doit retourner la référence à l'unique instance. La première fois, l'attribut privé statique sera `null`, donc on devra créer l'unique instance au premier appel de `GetInstance`, et la placer dans l'attribut prévu à cet effet, avant de retourner la référence à l'unique instance.
4. Alternativement, on pourrait utiliser une propriété avec un *getter* public qui ferait le même travail de la méthode `GetInstance`, et un *setter* privé qui serait utilisé seulement par le *getter* lors de sa première utilisation.
5. Une autre option serait d'utiliser un constructeur statique en C# (<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors>), ou un bloc d'initialisation statique (<https://www.c-sharpcorner.com/UploadFile/3614a6/static-initialization-block-in-java/>) dans d'autres langages.

Design Patterns : Structuraux

Adaptateur (Adapter, Wrapper)

Intention

Convertir l'interface d'une classe vers une autre interface attendue par le client. Un adaptateur permet à des classes de travailler ensemble qui autrement ne le pourraient pas.

Mise en contexte

Le but est d'isoler l'adaptation d'un sous-système, pour que le client n'ait pas à se soucier des détails du sous-système, et également pour faciliter la substitution du sous-système par un autre qui offre le même type de fonctionnalité, mais possiblement avec une interface différente.

Détails

Exemple avec les bases de données : connexion directe à la BD vs. adaptateur entre le client et la BD.

- Une **connexion directe** à la BD évite d'avoir des couches de trop, donc il est souvent possible d'optimiser des requêtes de meilleure façon et d'utiliser des fonctionnalités de la BD directement.
- Avec un **adaptateur**, il est possible de présenter une interface plus simple au client, qui n'a pas besoin de s'occuper des détails de connexion à la base de données, ni du langage de requêtes (SQL ou autre).
- Également avec un **adaptateur**, il est plus facile de migrer vers des BD différentes, même vers des BD de types différents (relationnelles vs. noSQL, ...).

Un exemple détaillé va être donné dans une solution C# disponible séparément.

Pont (Bridge)

Intention

Découpe une abstraction de son implémentation pour que les deux puissent évoluer indépendamment.

Mise en contexte

Détails

Composite

Intention

Composer des objets dans une structure arborescente pour représenter des hiérarchies tout-parties. Une composite permet aux clients de gérer des objets individuels et des compositions d'objets uniformément.

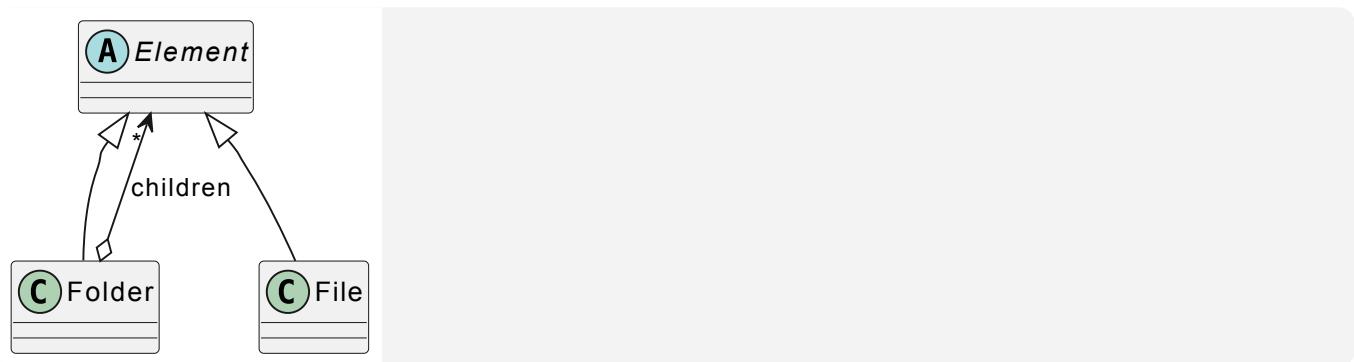
Mise en contexte

Le but est d'avoir des objets qui peuvent contenir d'autres objets qui peuvent eux-mêmes contenir d'autres objets, incluant des objets du même type qu'eux-mêmes.

Détails

Exemple le plus courant : un système de fichiers qui peut contenir des dossiers et d'autres fichiers. Les dossiers peuvent contenir d'autres dossiers ainsi que d'autres types de fichiers.

Deux types d'éléments : dossiers et fichiers. On pourrait aussi avoir plusieurs types de fichiers, qui seraient représentés par des sous-classes de `File` dans le diagramme de classes.



Décorateur (Decorator, Wrapper)

Intention

Attacher des responsabilités additionnelles à un objet dynamiquement. Les décorateurs offrent une alternative flexible à l'extension de classe pour étendre les fonctionnalités.

Mise en contexte

Si on veut rapidement ajouter une propriété ou fonctionnalité à un objet ou une méthode ou une propriété, on peut la décorer (ou l'envelopper) avec un décorateur.

Détails

Exemples :

- En C#, le décorateur `[JsonIgnore]` appliqué à une propriété ou un attribut d'une classe ajoute de l'information qui sera utilisée par le `JsonSerializer` lors de la sérialisation d'un objet. La propriété ou l'attribut sera alors ignoré et ne se retrouvera pas dans le résultat de la sérialisation.
- En développement web, plusieurs *frameworks* utilisent des décorateurs pour envelopper des fonctions ou méthodes pour qu'elles servent de gestionnaire (* handler*) de requêtes sur certaines routes. Une fonction peut être décorée du chemin de la route (`/`, `/data/users/`, ...), de la méthode (`GET`, `POST`, ...), et possiblement d'autres informations. De cette façon, la fonction est enregistrée dans le serveur comme étant le gestionnaire de la route. Si aucune fonction n'est associée à une route, alors le serveur va probablement retourner un `404`.

Facade

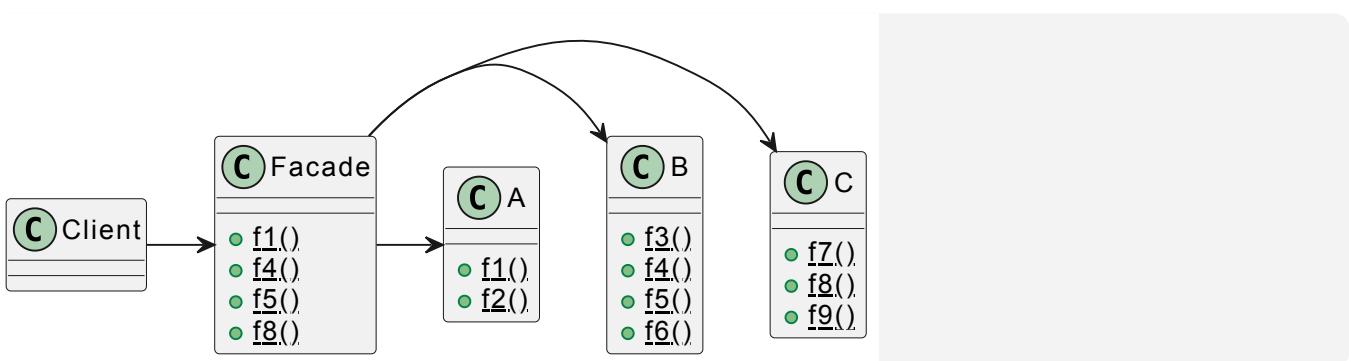
Intention

Offrir une interface unifiée vers un ensemble d'interfaces dans un sous-système. Une facade définit une interface de plus haut niveau qui rend le sous-système plus simple à utiliser.

Mise en contexte

Il est souvent utile de fournir une version simplifiée d'un sous-système qui va être plus facile à utiliser pour des clients qui n'ont pas besoins de toutes les fonctionnalités du sous-système. La facade va inclure seulement ce qui est utile pour le client, ce qui va aider au développement du client puisque qu'il y aura moins de méthodes ou classes inutiles à filtrer par les programmeurs.

Détails



Poids-Mouche (Flyweight)

Intention

Utiliser un système de partage pour pouvoir supporter un grand nombre d'objets détaillés efficacement.

Mise en contexte

Des applications, comme des applications de *CAD* ou des jeux vidéos, ont souvent besoin de charger ou de créer beaucoup d'objets similaires ou même identiques, ce qui peut demander beaucoup de ressources (mémoire et processeur). Le vidangeur (*garbage collector*) va également travailler fort pour libérer la mémoire utilisée par les objets qui ne sont plus utilisés.

Le but est de pouvoir partager et/ou réutiliser des objets, qui contiennent peut-être des images ou d'autres données d'une taille non-négligeable, pour diminuer les ressources utilisées par l'application.

Détails

À voir dans un autre cours.

Proxy (mandataire, substitut, surrogate)

Intention

Offre un substitut ou un espace réservé pour un autre objet pour contrôler son accès.

Mise en contexte

- **Proxy virtuel:** Lors du chargement d'un gros document, qui contient peut-être d'autres documents comme des images ou des vidéos, si on attend que tout le document soit chargé au complet, incluant les images et les vidéos, on pourrait devoir attendre longtemps. Donc, on peut utiliser des proxys pour contrôler le chargement des ressources imbriquées. Pas besoin de charger une vidéo au complet au chargement du document (peut-être un page web), surtout si la vidéo n'est pas visible sur le document et/ou si la vidéo est mise sur pause par défaut au moment du chargement.
- **Proxy de protection:** contrôler l'accès à un objet qui est soumis à des règles d'accès (selon des permissions ou des rôles). On peut par exemple transformer des objets *muables* en objets immuables en permettant seulement des opérations en lecture seule pour certains utilisateurs, mais pas pour d'autres (des admins par exemple).
- **Proxy distant (mandataire distant) :** un proxy réseau peut servir d'intermédiaire pour contrôler ou logger les sites internet permis ou visités à partir d'un réseau interne.

Détails

À venir

Design Patterns : Comportementaux (*Behavioral*)

Itérateur (Iterator, Enumerator, Cursor)

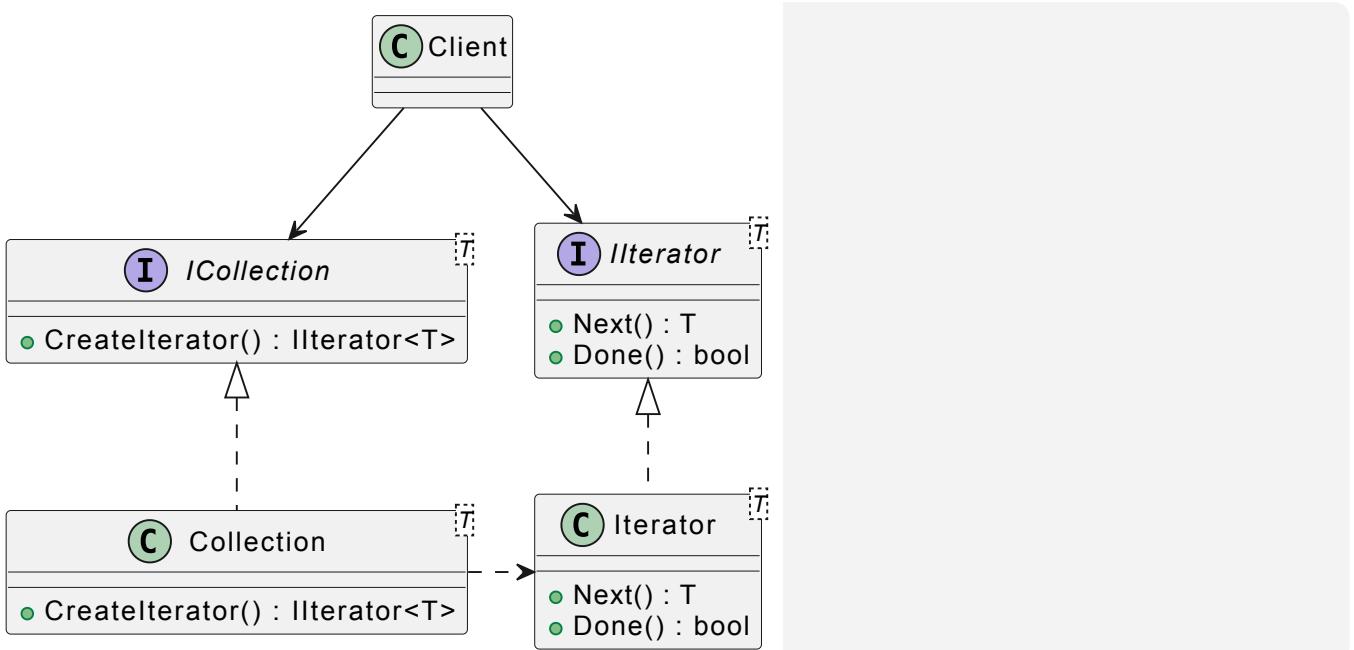
Intention

Fournir un moyen de parcourir séquentiellement les éléments d'un objet composé.

Mise en contexte

On a une collection d'objets (une liste, un tableau, un graphe, ...) et on doit parcourir tous les objets de cette collection, possiblement avec une boucle (for, foreach ou while).

Détails



Pour parcourir tous les éléments d'une Collection, on peut créer un Iterator, et faire une boucle sur l'itérateur tant que Done retourne faux (tant que ce n'est pas fini).

```
ICollection<int> col = new Collection<int>();
//...
IIterator<int> iter = col.CreateIterator();
while (!iter.Done()) {
    int x = iter.Next();
```

```
// ...
```

```
}
```

Chaîne de responsabilité (Chain of responsibility)

Intention

Éviter le couplage de l'émetteur d'une requête au receveur de cette requête en donnant à plus d'un objet la chance de traiter la requête. Chaîner les objets receveurs et passer la requête au suivant sur la chaîne pour traiter la requête à son tour.

Mise en contexte

La chaîne de responsabilité est souvent liée aux applications qui doivent gérer et traiter des événements pour donner la chance d'associer plusieurs gestionnaires (*handlers*) à des événements.

Domaines d'applications :

1. Évènements d'interface graphiques

- Exemple : plusieurs gestionnaires associés à l'évènement *fermeture de fenêtre*
 1. gestionnaire qui vérifie si des fichiers ont été modifiés depuis la dernière sauvegarde et qui propose de les enregistrer avant de quitter
 2. gestionnaire qui libère les ressources utilisées (mémoire, fichiers temporaires, ...) incluant la fenêtre elle-même

2. Requêtes reçues par un serveur web

- Exemple :
 1. *logger* qui enregistre les requêtes reçues dans des fichiers `.log` ou qui les affiche dans la console
 2. gestionnaire d'authentification qui valide un jeton (*token*) qui doit être inclus dans chaque requête sur des pages protégées
 3. gestionnaire qui traite la requête en tant que telle, qui va possiblement accéder à la BD et retourner un document HTML ou JSON dans la réponse

Commande (Command, Action, Transaction)

Intention

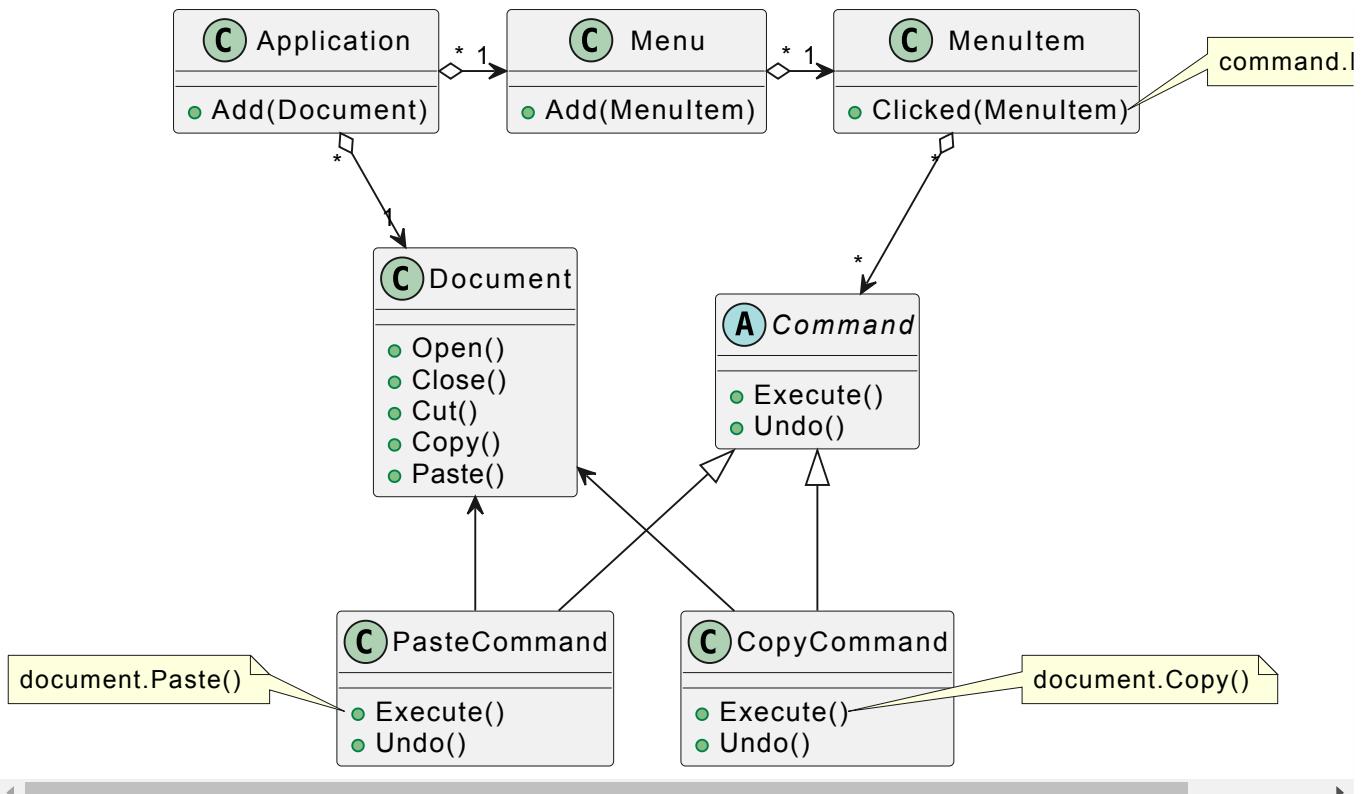
Encapsuler une requête dans un objet qui permet de paramétrer des clients avec des gestionnaires différents, et qui pourraient supporter des fonctions de type *annuler*.

Mise en contexte

Peut-être relié au *design pattern* de la chaîne de responsabilité et d'autres *design patterns*. Chaque gestionnaire dans une chaîne de responsabilité pourrait être représenté sous forme de commande.

Une commande peut aussi être utilisée pour représenter les fonctions de première classe (*first class functions*) dans les langages de programmation qui ne les supportent pas directement. En C#, par exemple, les fonctions fléchées sont des raccourcis pour créer des objets de type Action ou Func. Une Action est essentiellement une fonction qui ne retourne pas de valeur, donc une action est techniquement une procédure, et une Func est une fonction qui retourne une valeur. Une Action ou une Func peut être manipulée comme n'importe quel autre objet, et sont 2 variations du patron *commande* en C#.

Détails



Les `Undo` font l'inverse des `Execute`. Pour supporter les `undo` et `redo`, l'application doit conserver un historique des commandes, et créer des commandes `UndoCommand` et

`RedoCommand` qui vont, en utilisant l'historique de commandes, appeler soit `Undo` ou `Execute` sur les commandes de l'historique selon le cas.

Observateur (Observer, Dependents, Publish-Subscribe)

Intention

Définir une dépendance un-à-plusieurs entre des objets qui lorsqu'un objet change d'état, tous ses dépendants sont avertis et mis à jour automatiquement.

Mise en contexte

Des observateurs peuvent s'enregistrer auprès d'un objet (le sujet), et vont être notifiés quand le sujet change d'état (quand les attributs changent de valeurs). On peut aussi dire que les observateurs s'abonnent, ou sont associés, au sujet.

Interfaces graphiques

Ce patron et ses dérivés, sont souvent utilisés dans les applications avec interfaces graphiques. Quand les données d'un modèle sont modifiées, alors les vues reçoivent des notifications et sont mises à jour automatiquement. Plusieurs variantes ou dérivées plus ou moins complexes sont possibles, comme les architecture MVC (*Model-View-Controller*) et MVVM (*Model-View-ViewModel*).

Avec le MVC, le modèle est le sujet, et la vue est l'observateur. Avec le MVVM, le modèle est encore le sujet, mais l'observateur est normalement dans le * ViewModel*. Il existe aussi plusieurs autres variations pour les interfaces graphiques, dépendamment si elles sont pour des applications bureau, mobile, web, ou un mélange de celles-ci.

Communications

Le patron *Observateur* est aussi utilisé pour les communications de type diffusion (*broadcast*). Les utilisateurs peuvent s'inscrire à un service de diffusion de messages (le sujet), et par le fait même deviennent des observateurs du sujet. Techniquement, ce ne sont pas exactement les utilisateurs qui deviennent les observateurs, mais des objets les représentant sont enregistrés auprès du sujet qui les contactera lors d'une mise à jour. C'est de là que vient le nom *Publier-S'abonner* (*Publish-Subscribe*).

Les notifications poussées (*push notifications*) fonctionnent aussi un peu de la même façon, dans les applications de clavardage par exemple.

Stratégie (Strategy, Policy)

Intention

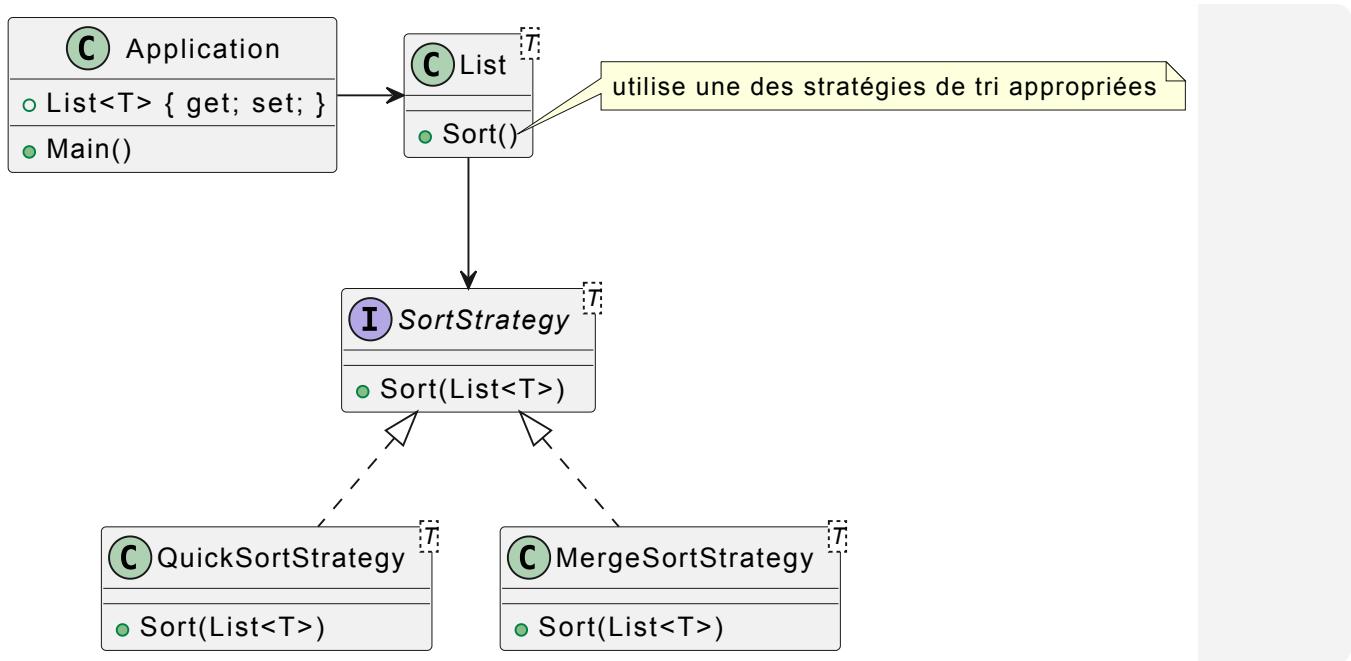
Définir une famille d'algorithmes, encapsuler chaque algorithme, et s'assurer qu'ils sont interchangeables. Permet à un algorithme de varier indépendamment des clients qui l'utilisent.

Mise en contexte

Exemple, à deux niveaux différents, dans le contexte des algorithmes de tri, dans le langage C#.

Au niveau supérieur, il y a une application qui a besoin de trier les éléments dans une liste (ou un tableau ou autre collection). Comme spécifié dans la documentation (<https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.sort?view=net-6.0>), on peut utiliser la méthode `List<T>.Sort()` sur n'importe quelle liste pour la trier. L'algorithme utilisé pourrait être n'importe quel algorithme (on fait confiance aux développeurs d'utiliser un algorithme efficace). Donc, la strategy exacte utilisée est déterminée à l'intérieur dans la classe `List`, et en principe, elle pourrait varier selon les situations. Par exemple, la stratégie utilisée pourrait varier si la liste est courte ou si elle est longue.

Note: ce diagramme ne représente pas exactement ce qui est fait en C#, mais donne l'idée générale du principe.



En C#, il n'est pas possible pour l'application de choisir la stratégie utilisée pour trier la liste, mais on pourrait créer une alternative à `List<T>.Sort()` qui nous permettrait de faire ce choix.

Comparateur

On peut utiliser des **comparateurs** (interface `IComparer<T>`) pour trier une liste selon des critères variables, différents des critères par défaut. Si on ne donne aucun paramètre à `Sort`, l'ordre par défaut sera utilisé pour trier les éléments. Dans ce cas, le type `T` doit implémenter l'interface `IComparable<T>`, qui définit la méthode `compareTo`, pour pouvoir comparer les éléments de la liste et déterminer l'ordre relatif des éléments.

Si `T` n'implémente pas `IComparable<T>`, ou si nous voulons spécifier un ordre différent pour le tri, alors nous pouvons donner un `IComparer` à la méthode `Sort` pour trier les éléments selon l'ordre désiré. Les différents `IComparer` peuvent être considérés comme des stratégies différentes dans le processus de tri des éléments.

