UNIFIED MODELING LANGUAGE (UML)

Approche de la décomposition fonctionnelle: Methode Merise

La méthode Merise est une méthode d'analyse et de conception de bases de données. Elle utilise une approche de la décomposition fonctionnelle pour diviser un système en sous-systèmes fonctionnels, et ainsi faciliter sa modélisation.

Demarche

La méthode Merise se compose de plusieurs étapes, notamment :
1. **L'analyse des besoins**
Il s'agit de l'étape de collecte des besoins des utilisateurs, des objectifs du système et des contraintes techniques et fonctionnelles.

2. **La spécification des exigences**
Il s'agit de l'étape de formalisation des besoins en termes de fonctionnalités attendues, d'entités impliquées et de relations entre ces entités.

3. **La conception préliminaire**
Il s'agit de l'étape de définition des sous-systèmes fonctionnels et de leur interdépendance.

4. **La conception détaillée**
Il s'agit de l'étape de conception détaillée de chaque sous-système, en utilisant des diagrammes de flux de données, des diagrammes entité-association, etc.

5. La mise en œuvre
Il s'agit de l'étape de réalisation effective du système, en utilisant un langage de programmation et un système de gestion de base de données.

Avantages

Les avantages de la méthode Merise sont nombreux :
1. Elle permet une **modélisation** claire et précise du système, en utilisant des diagrammes et des représentations graphiques.
2. Elle facilite la **communication** entre les parties prenantes, en utilisant un langage commun pour décrire les besoins et les fonctionnalités attendues.
3. Elle permet une **conception modulaire** et **évolutive** du système, en divisant le système en sous-systèmes fonctionnels.
4. Elle permet une **validation** précoce des choix de conception, en identifiant les erreurs et les incohérences dès les premières étapes.

Inconvenients

Cependant, la méthode Merise présente également certains inconvénients :
1. Elle peut être **coûteuse** et **chronophage**, en raison du grand nombre d'étapes et de documents requis pour la conception complète d'un système.

2. Elle peut être **rigide** et **difficile à adapter à des changements de besoins**, car elle repose sur une analyse préalable exhaustive.
3. Elle peut nécessiter une expertise **technique** et **fonctionnelle** importante pour être mise en œuvre efficacement.

Approche orientée objet avec UML

L'approche orientée objet est un paradigme de programmation qui modélise les systèmes informatiques en utilisant des objets qui ont des caractéristiques (attributs) et des comportements (méthodes). UML (Unified Modeling Language) est un langage de modélisation graphique standardisé utilisé pour concevoir et documenter des systèmes orientés objet.

Démarche :

La démarche de l'approche orientée objet avec UML comprend les étapes suivantes :
1. Analyse des besoins
Collecte des exigences du système et identification des objets clés impliqués.

2. Modélisation des classes
Définition des classes qui représentent les objets et leur structure interne, y compris les attributs et les méthodes.

3. Modélisation des relations
Définition des relations entre les classes, telles que l'héritage, l'agrégation, l'association, etc.

4. Modélisation du comportement
Définition des interactions et du flux de contrôle entre les objets en utilisant des diagrammes de séquence, des diagrammes d'activité, etc.

5. Modélisation de l'architecture
Définition de l'architecture logicielle globale du système en utilisant des diagrammes de composants, des diagrammes de déploiement, etc.

Avantages:

Avantages de l'approche orientée objet avec UML :
1. Abstraction et modularité
Permet de modéliser des systèmes complexes en utilisant des objets abstraits et en les regroupant en modules réutilisables.

2. Réutilisabilité
Les classes et les objets peuvent être réutilisés dans différents projets, ce qui accélère le développement logiciel.

3. Encapsulation

Les objets encapsulent à la fois les données et les méthodes, ce qui permet de maintenir l'intégrité des données et de cacher les détails internes.

4. Flexibilité et extensibilité
Permet des modifications et des ajouts faciles grâce à l'héritage, la polymorphisme et la composition.

5. Communication efficace
UML fournit des diagrammes graphiques qui facilitent la communication entre les parties prenantes techniques et non techniques.

Inconvénients:

Inconvénients de l'approche orientée objet avec UML :
1. Complexité
L'approche orientée objet et UML peuvent être complexes à comprendre et à maîtriser, surtout pour les débutants.

2. Surdimensionnement
Pour les petits projets, l'utilisation complète de l'approche orientée objet et UML peut être excessive et entraîner un surdimensionnement.

3. Temps et coût
La modélisation complète d'un système avec UML peut nécessiter du temps et des ressources considérables.

4. Surcharge de documentation
La méthode UML encourage la création de nombreux documents et diagrammes, ce qui peut entraîner une surcharge de documentation.

Etude comparative entre la méthode Merise et UML

Approche 4 vues

UML adopte une approche en 4 vues pour la modélisation des systèmes. Ces vues sont :

1. Vue logique
Cette vue se concentre sur la modélisation des classes, des objets, des relations et des comportements. Les diagrammes de classes, de séquence, d'activité, d'états et de communications font partie de cette vue.

2. Vue des processus
Cette vue se concentre sur la modélisation des processus métier. Les diagrammes de flux de données, de flux de contrôle et de flux de traitements font partie de cette vue.

3. Vue de composants
Cette vue se concentre sur la modélisation des composants logiciels et de leurs relations. Les diagrammes de composants, de déploiement et de packages font partie de cette vue.

4. Vue de déploiement
Cette vue se concentre sur la modélisation de la configuration matérielle et logicielle sur laquelle le système est déployé. Les diagrammes de déploiement font partie de cette vue.

5. Vue des besoins
Cette vue se concentre sur la modélisation des besoins des utilisateurs. Les cas d'utilisation, les diagrammes de séquences d'utilisation, les diagrammes d'activité d'utilisation et les diagrammes de classes d'analyse font partie de cette vue.

Les 13 diagrammes UML

1. Diagramme de cas d'utilisation : Modélise les cas d'utilisation du système.

2. Diagramme de classes : Modélise les classes, les attributs et les relations entre les classes.

3. Diagramme d'objets : Modélise les instances d'objets et leurs relations.

4. Diagramme de séquence : Modélise les interactions entre les objets dans une séquence chronologique.

5. Diagramme de collaboration : Modélise les interactions entre les objets dans un contexte particulier.

6. Diagramme d'états : Modélise les états et les transitions des objets.

7. Diagramme d'activité : Modélise les activités et les actions du système.

8. Diagramme de déploiement : Modélise la configuration matérielle et logicielle sur laquelle le système est déployé.

9. Diagramme de composants : Modélise les composants logiciels et leurs relations.

10. Diagramme de packages : Modélise les packages et leur structure.

11. Diagramme de profil : Modélise les profils UML personnalisés.

12. Diagramme de temps : Modélise les aspects temporels du système.

13. Diagramme d'interaction générale : Modélise les interactions entre les vues.

Classification des diagrammes par aspect et par vue

Les diagrammes UML peuvent être classés en fonction de leur aspect (fonctionnel ou architecture) et de leur vue.

En termes d'aspect, les diagrammes fonctionnels sont ceux qui modélisent les aspects du système qui sont liés aux processus métier, aux besoins des utilisateurs et aux interactions entre les acteurs. Les diagrammes d'architecture modélisent les aspects techniques du système, tels que la structure des classes, les composants logiciels, la configuration matérielle et logicielle, etc.

En termes de vue, les diagrammes UML peuvent être classés en fonction de la perspective qu'ils offrent sur le système. Voici une classification basée sur les vues principales :

1. Vue logique
• Diagramme de cas d'utilisation • Diagramme de classes • Diagramme d'objets • Diagramme de séquence • Diagramme de collaboration

2. Vue des processus
• Diagramme de cas d'utilisation • Diagramme de séquence • Diagramme de collaboration • Diagramme d'états • Diagramme d'activité • Diagramme de flux de données • Diagramme de flux de contrôle • Diagramme de flux de traitements

3. Vue de composants
• Diagramme de composants • Diagramme de déploiement • Diagramme de packages

4. Vue de déploiement
• Diagramme de déploiement

5. Vue des besoins
• Diagramme de cas d'utilisation • Diagramme de séquences d'utilisation • Diagramme d'activité d'utilisation • Diagramme de classes d'analyse

Etude détaillée des diagrammes d’UML

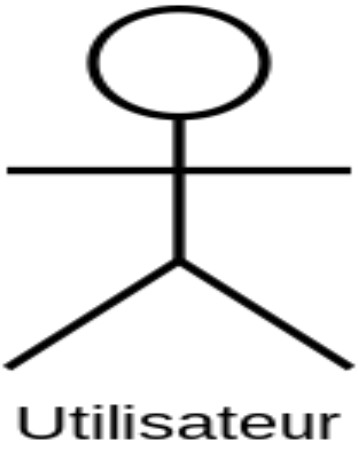
le diagramme de cas d’utilisation modélise le comportement de d’un système et permette de capturer les exigences du système.
Un cas d’utilisation décrit une fonction qu’un système execute pour l’atteindre l’objectif de l’utilisateur.

Objectif

L'objectif d'un diagramme de cas d'utilisation (UC) est de représenter les interactions entre les acteurs (utilisateurs) d'un système et les différentes fonctionnalités qu'ils peuvent utiliser. Le diagramme de cas d'utilisation permet de comprendre les besoins des utilisateurs et de définir les fonctionnalités principales du système.

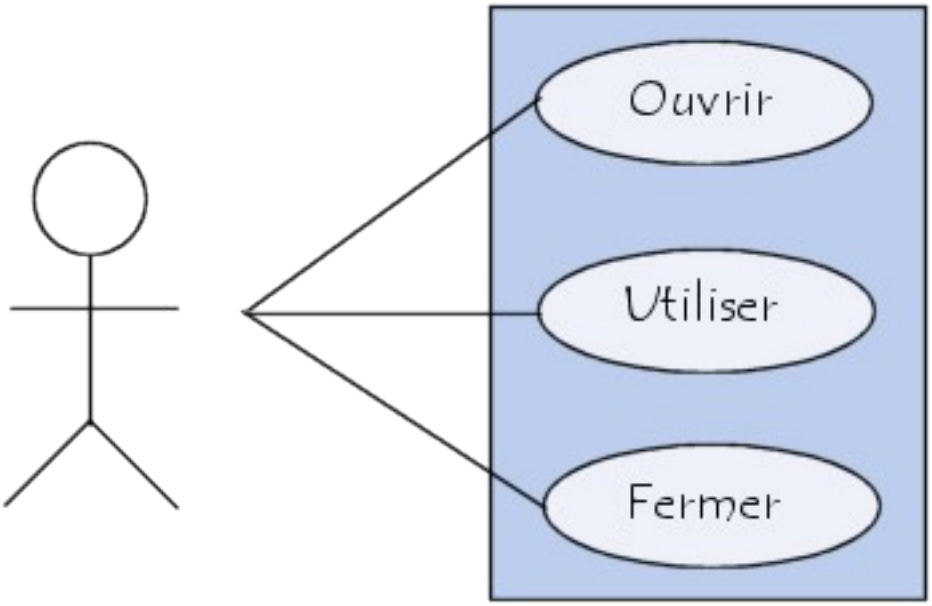
Acteur

Un acteur représente un rôle ou une entité externe qui interagit avec le système. Il peut s'agir d'un utilisateur humain, d'un autre système, d'un périphérique, ou même d'un autre logiciel. Les acteurs sont généralement représentés par des icônes d'humains ou de boîtes sur le diagramme de cas d'utilisation.



Notion de Use Case

Un cas d'utilisation représente une fonctionnalité ou un ensemble d'actions que le système peut effectuer pour répondre aux besoins d'un acteur. Il décrit l'interaction entre l'acteur et le système du point de vue de l'utilisateur. Chaque cas d'utilisation est représenté par une ellipse sur le diagramme de cas d'utilisation.



Les types de cas d’utilisation

Cas d'utilisation principal :
Il représente une fonctionnalité principale du système et décrit l'interaction entre l'acteur et le système pour atteindre un objectif spécifique.

Cas d'utilisation inclus

Il représente une fonctionnalité qui est incluse (ou réutilisée) dans d'autres cas d'utilisation. Il permet d'éviter la duplication de fonctionnalités communes entre les différents cas d'utilisation.

Cas d'utilisation étendu

Il représente une fonctionnalité facultative qui peut être ajoutée à un cas d'utilisation principal. Il permet d'étendre les fonctionnalités de base d'un cas d'utilisation principal en ajoutant des étapes ou des actions supplémentaires.

Cas d'utilisation abstrait

Il représente un cas d'utilisation qui est utilisé pour regrouper des fonctionnalités communes entre plusieurs cas d'utilisation. Il ne peut pas être utilisé directement, mais doit être spécialisé par d'autres cas d'utilisation.

Les liens (ou relations) entre cas d’utilisation

Inclusion (include)

Il indique qu'un cas d'utilisation inclut les actions d'un autre cas d'utilisation. Cela signifie que le cas d'utilisation inclus est une partie nécessaire pour accomplir le cas d'utilisation principal.

Extension (extend)

Il indique qu'un cas d'utilisation peut étendre ou ajouter des fonctionnalités à un autre cas d'utilisation. Cela permet de décrire des variations ou des options facultatives dans le scénario d'un cas d'utilisation.

Généralisation (generalization)

Il représente une relation d'héritage entre les cas d'utilisation. Il permet de regrouper plusieurs cas d'utilisation similaires sous une forme générale (cas d'utilisation abstrait) et de spécialiser ces cas d'utilisation en fonction de leurs différences spécifiques.

Formalisme d’un diagramme d’UC

Un diagramme de cas d'utilisation se compose de plusieurs éléments :

Acteur

Il est représenté par une icône d'humain ou de boîte et est placé à l'extérieur du diagramme. Il représente un rôle ou une entité externe qui interagit avec le système.

Cas d'utilisation

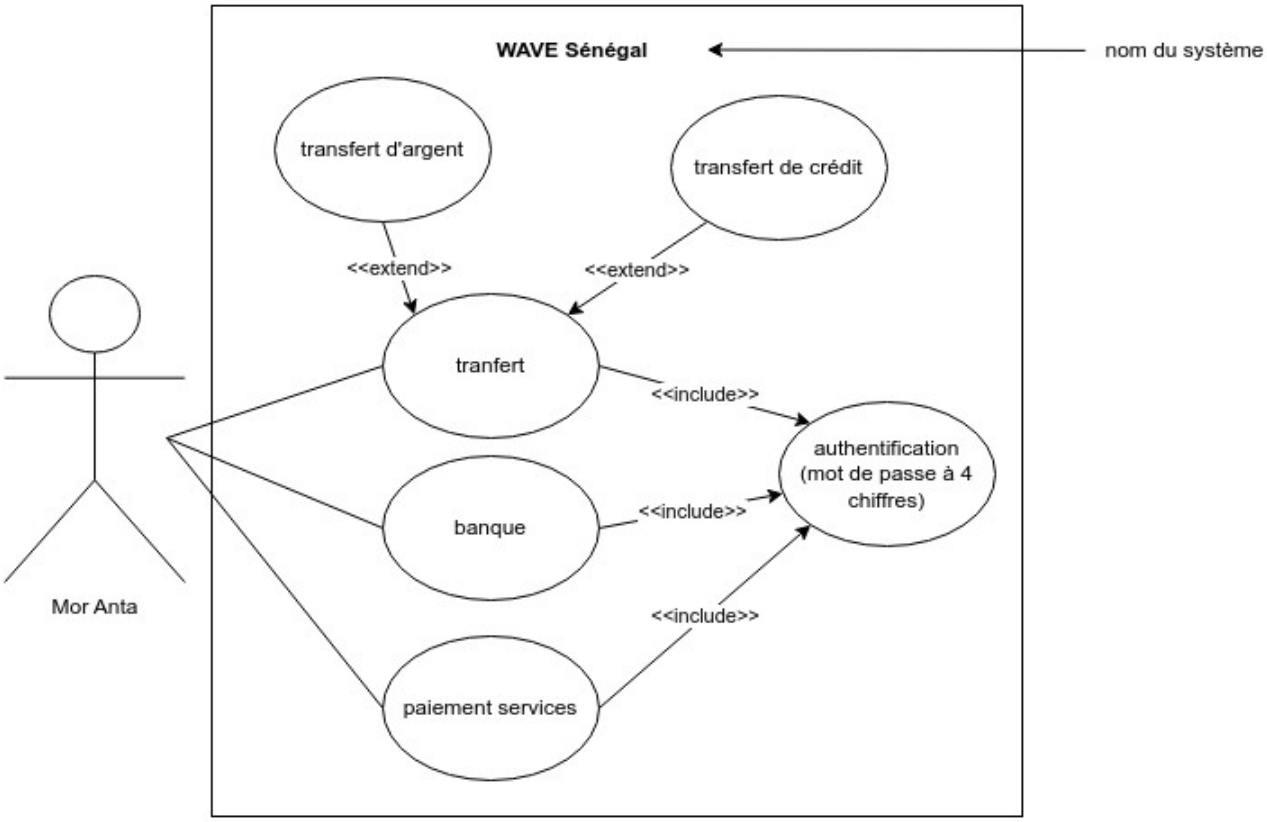
Il est représenté par une ellipse et est placé à l'intérieur du diagramme. Il représente une fonctionnalité ou une interaction spécifique entre l'acteur et le système.

Ligne de liaison (relation)

Elle relie l'acteur aux cas d'utilisation et peut avoir une flèche pour indiquer la direction de l'interaction. Les types de liens les plus couramment utilisés sont l'inclusion (include) et l'extension (extend).

Annotations

Elles sont utilisées pour fournir des informations supplémentaires, telles que des contraintes, des descriptions ou des notes, pour clarifier le diagramme.



Exemple de diagramme de cas d'utilisation

Description Textuelle d’un cas d’utilisation

La description textuelle d'un cas d'utilisation fournit des détails sur les différentes étapes et interactions entre l'acteur et le système. Elle est généralement structurée en plusieurs sections :

Objectif

Cette section décrit l'objectif principal du cas d'utilisation, c'est-à-dire ce que l'acteur souhaite accomplir en utilisant cette fonctionnalité.

Scénario Nominal

Cette section décrit les étapes normales et les interactions attendues entre l'acteur et le système pour accomplir l'objectif du cas d'utilisation. Elle présente le déroulement typique du cas d'utilisation sans prendre en compte les situations exceptionnelles.

Scénario Alternatif

Cette section décrit les variations ou les alternatives possibles dans le scénario nominal. Elle présente les différentes voies que le cas d'utilisation peut emprunter en fonction de certaines conditions ou décisions prises par l'acteur ou le système.

Scénario d'Exception

Cette section décrit les situations exceptionnelles ou les erreurs qui peuvent survenir pendant l'exécution du cas d'utilisation et comment elles doivent être gérées. Elle présente les actions spécifiques à entreprendre pour résoudre les problèmes ou traiter les erreurs.

Structure d'une fiche descriptive d'un Cas d'Utilisation (UC)

Une fiche descriptive d'un cas d'utilisation peut être structurée de la manière suivante :

Titre : Le titre du cas d'utilisation, qui décrit brièvement la fonctionnalité ou l'interaction.

Identifiant : Un identifiant unique pour le cas d'utilisation, qui peut être utilisé pour référencer le cas d'utilisation dans d'autres documents ou systèmes.

Acteurs : La liste des acteurs impliqués dans le cas d'utilisation.

Objectif : Une brève description de l'objectif principal du cas d'utilisation.

Préconditions : Les conditions qui doivent être remplies avant que le cas d'utilisation puisse être exécuté.

Post-conditions : Les états du système après l'exécution du cas d'utilisation.

Scénario Nominal : Le scénario normal décrivant les étapes et les interactions entre l'acteur et le système.

❖ Description textuelle du cas d'utilisation "S'inscrire"

Cas d'utilisation	s'inscrire
Acteur	visiteur
Objectif	Permet à un visiteur d'être un membre.
Pré condition	Consulter le site web
Post condition	Créer un compte dans le site web
Scénario nominal	1. le visiteur choisit de s'inscrire 2. Le système affiche le formulaire correspondant 3. le visiteur remplie le formulaire 4. Le système vérifie les données saisies 5. Le système affiche l'espace du membre 6. L'instance de cas d'utilisation se termine
Exception	Si un champ lui manque le saisit ou présente une erreur de saisie, le système affiche un message d'erreur.

Diagramme de Classe

Le diagramme de classe est l'un des diagrammes les plus couramment utilisés en UML. Il permet de représenter la structure statique d'un système en mettant l'accent sur les classes, les objets, les attributs et les relations entre eux.

Objectif

L'objectif d'un diagramme de classe est de visualiser les classes et les objets d'un système, ainsi que leurs attributs et leurs relations, afin de faciliter la compréhension de la structure du système et de ses différentes composantes.

Notion de Classe/Objet

Une classe est un modèle ou un plan à partir duquel les objets sont créés. Elle définit les caractéristiques communes et les comportements partagés par un groupe d'objets similaires. Par exemple, une classe "Voiture" peut définir les attributs et les méthodes communs à toutes les voitures.

Un objet est une instance spécifique d'une classe. Il représente une entité concrète existante dans le système. Par exemple, une instance de la classe "Voiture" pourrait être une voiture spécifique avec des valeurs d'attributs spécifiques.

Attributs

Les attributs sont les caractéristiques ou les données associées à une classe. Ils représentent les propriétés ou les variables que les objets de cette classe possèdent. Par exemple, une classe "Voiture" peut avoir des attributs tels que "marque", "modèle" et "couleur".

Méthodes

Les méthodes représentent les actions ou les comportements que les objets de la classe peuvent effectuer. Elles définissent les opérations ou les fonctions qui peuvent être exécutées sur les objets. Par exemple, une classe "Voiture" peut avoir des méthodes telles que "démarrer", "accélérer" et "freiner".

Encapsulation

L'encapsulation est un concept clé en programmation orientée objet qui implique de regrouper les attributs et les méthodes connexes au sein d'une classe et de cacher les détails internes de la classe aux autres parties du système. Cela permet de protéger les données et de contrôler l'accès aux méthodes.

Visibilité ou Portée d'un attribut ou d'une Méthode

La visibilité ou la portée d'un attribut ou d'une méthode définit l'accès et la visibilité de cet élément dans le système. UML définit les visibilités suivantes :

Public (+) : L'attribut ou la méthode est accessible depuis n'importe où dans le système, y compris à l'extérieur de la classe.

Protégé (#) : L'attribut ou la méthode est accessible uniquement à l'intérieur de la classe elle-même et des classes dérivées.

Privé (-) : L'attribut ou la méthode est accessible uniquement à l'intérieur de la classe elle-même.

Package (~) : L'attribut ou la méthode est accessible à l'intérieur du package (groupe de classes) auquel la classe appartient.

La spécification de la visibilité se fait en préfixant l'attribut ou la méthode avec le symbole correspondant (+, #, -, ~) lors de la représentation sur le diagramme de classe.

Surcharge

La surcharge (overloading) est une notion qui permet à une classe d'avoir plusieurs méthodes portant le même nom, mais avec des paramètres différents. Cela permet de fournir différentes versions d'une même méthode, adaptées à des situations différentes.

Relation entre Classes

One-to-Many (Un-à-plusieurs)
C'est une relation où une instance d'une classe est associée à plusieurs instances d'une autre classe. Par exemple, une classe "Équipe" peut avoir une relation one-to-many avec une classe "Joueur", où une équipe peut avoir plusieurs joueurs.

Many-to-One (Plusieurs-à-un)
C'est l'inverse de la relation one-to-many, où plusieurs instances d'une classe sont associées à une seule instance d'une autre classe. Par exemple, plusieurs joueurs peuvent appartenir à une seule équipe.

Many-to-Many (Plusieurs-à-plusieurs)
C'est une relation où plusieurs instances d'une classe sont associées à plusieurs instances d'une autre classe. Par exemple, une classe "Étudiant" peut avoir une relation many-to-many avec une classe "Cours", où un étudiant peut s'inscrire à plusieurs cours et un cours peut avoir plusieurs étudiants inscrits.

One-to-One (Un-à-un)
C'est une relation où une instance d'une classe est associée à une seule instance d'une autre classe, et vice versa. Par exemple, une classe "Personne" peut avoir une relation one-to-one avec une classe "Passeport", où une personne a un seul passeport et un passeport est associé à une seule personne.

Héritage

L'héritage est un mécanisme qui permet à une classe de hériter des attributs et des méthodes d'une autre classe. La classe qui hérite est appelée classe dérivée ou classe fille, et la classe dont elle hérite est appelée classe de base ou classe mère. L'héritage permet de créer une hiérarchie de classes, où les classes dérivées héritent des caractéristiques communes de la classe de base, tout en pouvant ajouter ou redéfinir des fonctionnalités spécifiques.

Classe abstraite

Une classe abstraite est une classe qui ne peut pas être instanciée directement, mais sert de classe de base pour d'autres classes. Elle peut contenir des méthodes abstraites, qui sont des méthodes déclarées mais n'ont pas d'implémentation dans la classe abstraite. Les classes dérivées de la classe abstraite doivent fournir une implémentation pour les méthodes abstraites.

Redéfinition

La redéfinition (override) est un mécanisme qui permet à une classe dérivée de fournir une implémentation différente pour une méthode héritée de la classe de base. Lorsque la méthode est appelée sur une instance de la classe dérivée, la version redéfinie de la méthode est exécutée plutôt que la version héritée.

Polymorphisme

Le polymorphisme est un concept clé de la programmation orientée objet qui permet à une variable d'un type de classe de prendre différentes formes et d'exécuter des comportements différents en fonction du type réel de l'objet auquel elle est associée.

Interface

Une interface est une spécification de méthodes qu'une classe doit implémenter. Elle définit un ensemble de méthodes et de constantes que les classes implémentant cette interface doivent fournir. Une interface fournit un contrat pour les classes, en spécifiant les méthodes qu'elles doivent implémenter, mais ne fournit pas d'implémentation concrète de ces méthodes.

Transformation du diagramme de classes (sans utiliser d'outil)

Un diagramme de classes est un type de diagramme UML qui montre la structure d'un système logiciel en montrant les classes, leurs attributs et leurs relations. Les diagrammes de classes peuvent être transformés dans d'autres formats, tels que XML et JSON, sans utiliser d'outil.

Pour transformer un diagramme de classes en XML :

1.Identifiez les classes dans le diagramme de classes.

2.Pour chaque classe, identifiez les attributs et leurs types.

3.Pour chaque classe, identifiez les relations avec les autres classes.

4.Écrivez un document XML qui définit les classes, leurs attributs et leurs relations.

Par exemple, le diagramme de classes suivant montre une classe appelée Personavec deux attributs, name et age :

```
[Person]
name: String
age: Integer
```

Le document XML suivant définit la Personclasse:

Pour transformer un diagramme de classes en JSON:

1.Identifiez les classes dans le diagramme de classes.

2.Pour chaque classe, identifiez les attributs et leurs types.

3.Pour chaque classe, identifiez les relations avec les autres classes.

4.Écrivez un document JSON qui définit les classes, leurs attributs et leurs relations.

Par exemple, le document JSON suivant définit la Personclasse :

```
{
  "name": "John Doe",
  "age": 25
}
```

Le code suivant crée un objet de la Personclasse et définit les valeurs des attributs `name` et `age`:

```
var person = new Person();
person.name = "John Doe";
person.age = 25;
```

Le code suivant crée un lien entre deux objets de la `Person` classe:

```
var person1 = new Person();
var person2 = new Person();

person1.friend = person2;
```

Cas particulier : Héritage

Un système hérité est un système logiciel qui n'est plus en cours de développement actif. Les systèmes hérités peuvent être difficiles à maintenir et à étendre. Une façon de rendre un système hérité plus gérable est de le transformer en un diagramme de classes. Cela peut être fait en créant manuellement un diagramme de classes pour le système hérité ou en utilisant un outil pour générer un diagramme de classes à partir du code source du système hérité.

Cas particulier : Composition

La composition est un type de relation entre des classes dans lesquelles une classe contient une autre classe. La classe qui contient l'autre classe est appelée le **conteneur** et la classe qui est contenue est appelée le **contenu** . La classe contenue est dite membre **de** la classe conteneur.

Par exemple, une Carclasse peut avoir une Engineclasse comme membre. La Engineclasse serait contenue dans la Carclasse.

Cas particulier : Agrégation

L'agrégation est un type de relation entre des classes dans lesquelles une classe contient une autre classe. La différence entre l'agrégation et la composition est que la classe contenue dans une relation d'agrégation n'appartient pas à la classe conteneur. La classe contenue peut exister indépendamment de la classe conteneur.

Par exemple, une Universityclasse peut avoir une Departmentclasse en tant qu'agrégation. La Departmentclasse serait contenue dans la Universityclasse, mais elle pourrait aussi exister en dehors de la Universityclasse.