Développement orienté objet

Modélisation et diagrammes de classes UML

Qu'est-ce que la modélisation ?

La modélisation est une étape importante dans le développement d'un logiciel. Elle permet de décrire le comportement du logiciel et de le représenter sous une forme compréhensible par tous les acteurs du projet (qui ne sont pas tous développeurs). Il est donc nécessaire de disposer d'un langage de modélisation qui soit compréhensible par tous ; tout en étant suffisamment précis pour décrire le comportement du logiciel.

Unified Modeling Language (UML)

L'UML est un langage de modélisation graphique qui permet de décrire le comportement d'un logiciel. Il est utilisé dans le cadre de la conception orientée objet. Il permet de décrire les classes, les objets, les relations entre les classes et les objets, les cas d'utilisation, les interactions entre les objets ...

UML est définie via une norme. Elle offre un vocabulaire et une syntaxe précise. UML repose sur la définition de **diagrammes** qui permettent de représenter les différents aspects d'un logiciel. Il existe plusieurs types de diagrammes UML. Dans ce cours, nous allons nous intéresser uniquement aux **diagrammes de classes**. La figure ci-dessous liste les différents types de diagrammes UML sous forme d'un diagramme de classes :

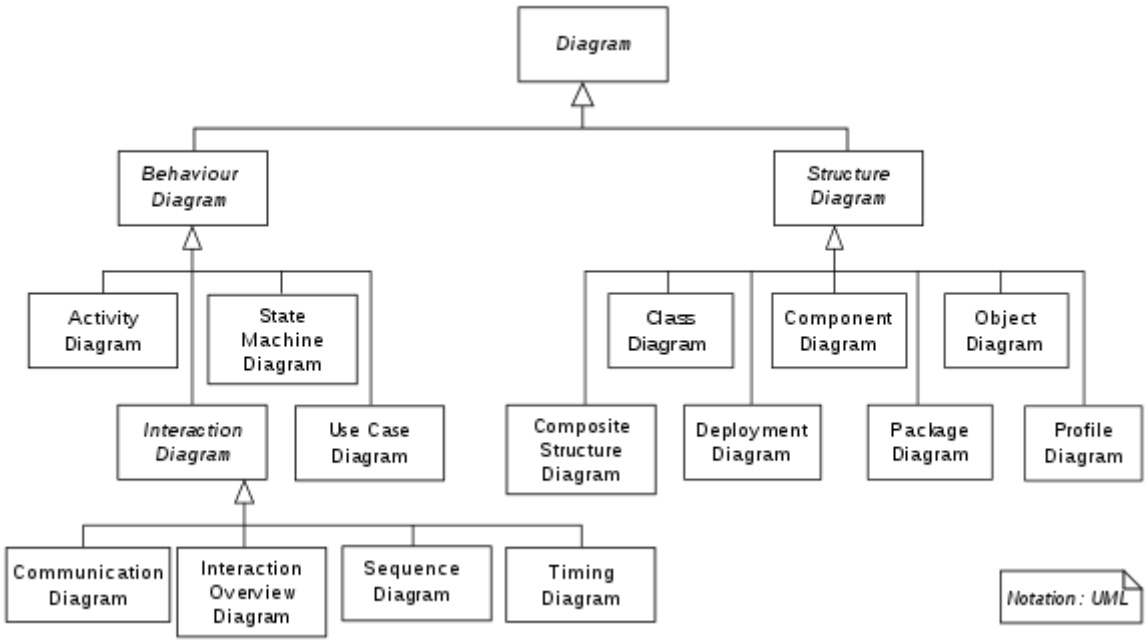


Diagramme de classes

Le diagramme de classes est un diagramme statique qui permet de représenter les classes d'un logiciel, leurs attributs et leurs méthodes. Il permet également de représenter les relations entre les classes.

Classes en UML

L'élément de base d'un diagramme de classes est la classe. Une classe est représentée par un rectangle divisé en trois parties :

- La première partie contient le **nom** de la classe
- La deuxième partie contient les **attributs** de la classe
- La troisième partie contient les **opérations** / **méthodes** de la classe

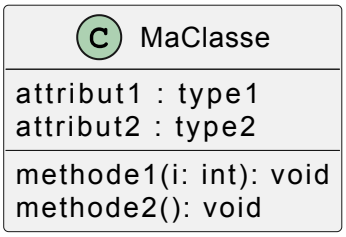


Diagramme de classes et implémentations

Un diagramme de classes ne donne aucune indication sur l'implémentations des classes. Il ne s'agit que d'une description du comportement du logiciel. Il est donc possible d'implémenter les classes de différentes manières en prenant en compte les spécificités du langage utilisé. Le but du diagramme de classes est de décrire l'information contenue dans les classes et les méthodes permettant de manipuler cette information.

Types de base

On peut utiliser les types de base suivants :

- `int` : entier (`number` en TypeScript)
- `float` : nombre à virgule flottante (`number` en TypeScript)
- `double` : nombre à virgule flottante double précision (`number` en TypeScript)
- `boolean` : booléen (`boolean` en TypeScript)
- `char` : caractère (`string` en TypeScript)
- `string` : chaîne de caractères (`string` en TypeScript)
- `void` : type vide (`void` en TypeScript)

Remarque : il n'y a pas forcément de correspondance directe entre les types de base en UML et en TypeScript. Il s'agit alors de choisir le type de base le plus adapté ou vice versa. Cela sera aussi le cas pour d'autres concepts.

Convention sur les noms

- le nom d'une classe commence par une majuscule
- le nom d'un attribut ou d'une méthode commence par une minuscule
- les types de base (`int`, `float`, `double`, `boolean` ...) sont écrits en minuscule
- il n'y a pas d'espace dans les noms des classes, attributs et méthodes

- les noms composés sont écrits en **camelCase** (la première lettre de chaque mot est en majuscule sauf la première pour les attributs et les méthodes)

Déclaration d'un attribut

Syntaxe

nomAttribut: TypeAttribut{[multiplicité]} {= valeurInitiale}

Remarque : une information entre {} indique que l'élément est facultatif

- multiplicité = nombre de valeurs dans une collection (tableau, liste) :

min..max, n, min..*, *

Indication : `min`, `max` et `n` sont des entiers positifs ou nul ; `*` indique que la collection peut contenir un nombre quelconque de valeurs (éventuellement 0)

Exemples

i: int attribut `i` de type entier

p: Point attribut `p` de type `Point`

tabI: int[*] tableau d'entiers de taille quelconque

tabP: Point[4] tableau de 4 points

x: double = 0.0 réel double précision privé initialisé à 0

Constructeur

Syntaxe

NomClasse({LISTE_PARAMS}))

- LISTE_PARAM = paramètres de l'opération séparés par des virgules
- Syntaxe d'un paramètre : nomParamètre: TypeParamètre

Exemples

Point() constructeur sans paramètre de la classe `Point`

Point(x: int, y: int) constructeur avec deux paramètres de type entier de la classe `Point`

Constructeur en TypeScript

En TypeScript, le constructeur est une méthode spéciale nommée `constructor` qui est définie dans la classe et ne porte donc pas le nom de la classe.

Déclaration d'une méthode

Syntaxe

nomMéthode({LISTE_PARAMS}) {: TypeRetour}

- LISTE_PARAM = paramètres de l'opération séparés par des virgules
- Syntaxe d'un paramètre : nomParamètre: TypeParamètre

Exemple

affiche() méthode sans paramètre et sans valeur de retour (en UML on peut omettre `void`)

affiche(i: int): void méthode avec un paramètre de type entier et sans valeur de retour

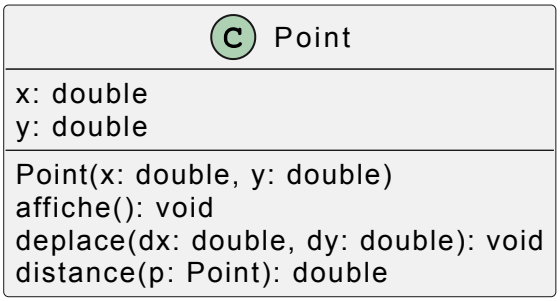
affiche(): int méthode sans paramètre et avec une valeur de retour de type entier

Exemple de classe en TypeScript et en UML

Classe `Point` en TypeScript

```
class Point {
    x: number;
    y: number;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }

    public affiche(): void {
        console.log("(" + this.x + ", " + this.y + ")");
    }

    public deplace(dx: number, dy: number): void {
        this.x += dx;
        this.y += dy;
    }

    public distance(p: Point): number {
        return Math.sqrt((this.x - p.x) ** 2 + (this.y - p.y) ** 2);
    }
}
```

Classe `Point` en UML


Point
x: double
y: double
Point(x: double, y: double)
affiche(): void
deplace(dx: double, dy: double): void
distance(p: Point): double