Table des matières

Table des figures

Introduction

Contexte

Dans le domaine du développement logiciel, la modélisation constitue une étape clé, essentielle pour représenter de manière abstraite la structure et le comportement d'un système avant même l'écriture du code. Le langage UML (Unified Modeling Language) s'est imposé comme un standard de facto, notamment grâce au diagramme de classes, qui permet de décrire les entités principales d'un système, leurs attributs, méthodes et les relations qu'elles entretiennent. Cette modélisation facilite la communication entre les différents acteurs du projet, encourage la réutilisabilité et favorise une approche rigoureuse du développement. Dans le contexte de Model-Driven Design (MDD), les modèles ne sont plus de simples documents de conception, mais deviennent de véritables artefacts centraux du processus de développement. L'idée est de passer du modèle à l'implémentation par génération automatique de code, minimisant ainsi les tâches répétitives et les erreurs humaines.

Problématique

Cependant, dans la pratique, la transition du diagramme UML vers le code reste souvent manuelle, répétitive, chronophage et source d'erreurs, en particulier pour la mise en place des structures de base (ou "boilerplate code"). Ce constat soulève la problématique suivante : *Comment automatiser la transformation d'un diagramme de classes UML en code source, tout en assurant une fidélité sémantique au modèle initial, et en intégrant cette automatisation dans une logique d'agents autonomes capables de collaborer à cette tâche complexe ?*

Objectifs

Le projet UML2Code vise à :

— Développer un outil capable d'analyser automatiquement un diagramme de classes UML exporté depuis Draw.io (au format XML).
— Extraire les entités du modèle : classes, attributs, méthodes, relations (héritage, agrégation, association...).

— Générer automatiquement un squelette de code correspondant en Java Spring et en PHP Laravel, selon les conventions du framework.
— Intégrer cette approche dans une architecture multi-agent, où des agents spécialisés prennent en charge les différentes étapes de l'analyse et de la génération du code.

Méthodologie

Pour atteindre ces objectifs, le projet adopte une approche inspirée des **compilateurs**, dans laquelle le diagramme UML est considéré comme un langage source à analyser et à traduire. L'architecture du système suit une chaîne de traitement modulaire, où chaque composant a un rôle bien défini dans le processus de transformation du modèle en code.

— **Module de parsing** : lit le fichier XML exporté depuis Draw.io et extrait les éléments bruts (balises, attributs) pour produire une représentation structurée, au format JSON.
— **Module d'analyse syntaxique** : vérifie la conformité des éléments extraits selon les règles du méta-modèle UML. Il transforme les données structurées en objets internes représentant les classes, attributs, méthodes, relations, etc.
— **Module d'analyse sémantique** : interprète les structures syntaxiques pour en extraire la signification métier et enrichir la représentation avec des métadonnées utiles à la génération de code (par exemple, les types de relation, les dépendances entre classes, les annotations spécifiques au framework).
— **Modules de génération de code** : chaque module est dédié à un langage ou framework cible (Spring, Laravel, etc.). Il traduit la représentation sémantique en squelette de code conforme aux conventions du framework choisi.

1

Contexte et Fondements

1.1 Focus sur le diagramme de classes dans draw.io

1.1.1 Rôle du diagramme de classes dans la modélisation

Le diagramme de classes constitue l'élément central de la modélisation orientée objet et l'un des diagrammes les plus importants du langage UML (Unified Modeling Language) Booch et al., 2005. Dans le cadre de notre projet UML2Code, nous nous sommes particulièrement intéressés à ce type de diagramme pour plusieurs raisons fondamentales.

Premièrement, le diagramme de classes représente la structure statique d'un système en décrivant ses classes, leurs attributs, leurs méthodes et les relations entre elles. Cette représentation structurelle correspond directement aux concepts de la programmation orientée objet que nous retrouvons dans des frameworks comme Spring (Java) et Laravel (PHP), qui sont les cibles premières de notre générateur de code.

Deuxièmement, contrairement à d'autres diagrammes UML qui capturent des aspects comportementaux ou dynamiques, le diagramme de classes se prête naturellement à une transformation vers du code source Fowler, 2003. Cette caractéristique est essentielle pour notre projet, puisque notre objectif est précisément d'automatiser la génération de code à partir de modèles UML.

Enfin, dans le cycle de développement logiciel, le diagramme de classes joue un rôle prépondérant en servant de contrat entre les concepteurs et les développeurs. Il définit clairement les entités du domaine, leurs propriétés et leurs interactions, fournissant ainsi une base solide pour l'implémentation. En permettant la génération automatique de code à partir de ces diagrammes, notre outil `UML2Code` renforce ce contrat et réduit les écarts potentiels entre conception et implémentation.

Les études empiriques montrent que l'utilisation des diagrammes de classes améliore significativement la qualité du code produit, réduisant les défauts de conception jusqu'à 30% Wang et al., 2024. De plus, ils facilitent la compréhension des systèmes complexes, diminuant le temps d'onboarding des nouveaux développeurs et améliorant la maintenabilité à long terme.

Malgré l'émergence d'approches alternatives comme le C4 Model ou les diagrammes basés sur le texte (PlantUML), le diagramme de classes UML reste l'outil privilégié pour la modélisation structurelle dans les projets orientés objet, particulièrement dans les domaines nécessitant une forte rigueur architecturale comme les systèmes critiques ou les applications d'entreprise.

1.1.2 Rôle de draw.io pour créer les diagrammes de classes

Dans le cadre de notre projet, nous avons choisi **draw.io** (également connu sous le nom de diagrams.net) comme outil de modélisation pour plusieurs avantages spécifiques qui correspondent parfaitement à nos besoins Draw.io, 2023 (voir figure 1.1).

Figure 1.1 – *Outil draw.io*

Accessibilité et facilité d'utilisation : Contrairement à des outils UML professionnels comme Enterprise Architect ou Rational Rose qui présentent une courbe d'apprentissage importante, draw.io offre une interface intuitive basée sur le glisser-déposer qui permet de créer rapidement des diagrammes UML sans formation approfondie. Cette accessibilité était cruciale pour notre projet, car elle permet à un plus large éventail d'utilisateurs de bénéficier de notre générateur de code.

Disponibilité multiplateforme : Draw.io est disponible en version web, application de bureau, et comme plugin pour diverses plateformes collaboratives (Confluence, Google Drive, etc.). Cette ubiquité facilite l'intégration de notre outil dans différents environnements de travail sans imposer de contraintes techniques supplémentaires.

Gratuité et open-source : L'outil est gratuit et son code source est ouvert, ce qui correspond à notre philosophie de développement et permet d'envisager des extensions futures ou des intégrations plus poussées si nécessaire.

Support UML complet : Malgré sa simplicité apparente, draw.io propose une bibliothèque complète d'éléments UML, particulièrement pour les diagrammes de classes. L'outil permet de représenter aisément :

— Les classes avec leurs compartiments (nom, attributs, méthodes)
— Les interfaces et classes abstraites
— Les différents types de relations (association, agrégation, composition, héritage, implémentation)

— Les multiplicités et les rôles dans les associations
— Les packages pour organiser les éléments

Conversion au format XML : Un avantage décisif pour notre projet est la capacité de draw.io à exporter les diagrammes dans un format XML structuré. Cette fonctionnalité est fondamentale pour notre approche, car elle nous permet d'analyser programmatiquement le contenu des diagrammes et de les transformer en code source.

Cette intégration avec draw.io permet aux concepteurs de se concentrer sur la modélisation visuelle tout en bénéficiant de la génération automatique de code fournie par notre outil.

1.1.3 Exploitation du XML

Le format XML généré par draw.io constitue la matière première de notre système UML2Code. Il s'agit d'un document structuré qui encode l'ensemble des informations visuelles et sémantiques du diagramme de classes créé par l'utilisateur Draw.io, 2022.

Contrairement aux formats standardisés comme XMI (XML Metadata Interchange), le XML de draw.io n'est pas spécifiquement conçu pour l'échange de modèles UML. Il s'agit plutôt d'un format général utilisé par la bibliothèque sous-jacente mxGraph pour représenter n'importe quel type de diagramme. Cette particularité présente à la fois des défis et des opportunités pour notre projet.

1.1.3.1 Structure générale du XML de draw.io

L'examen de plusieurs diagrammes exportés depuis draw.io nous a permis d'identifier la structure hiérarchique suivante presentée dans la Figure 1.2 :

```
<mxfile>
  <diagram id="diagram-id" name="Page-1">
    <mxGraphModel>
      <root>
        <mxCell id="0"/> <!-- Cellule racine -->
        <mxCell id="1" parent="0"/> <!-- Couche par défaut -->
        <mxCell id="..."
                style="..."
                vertex="1"
                parent="1">
          <mxGeometry ... />
        </mxCell>
        <!-- Connexions entre éléments -->
        <mxCell id="..."
                style="..."
                edge="1"
                parent="1"
                source="..."
                target="...">
          <mxGeometry ... />
        </mxCell>
      </root>
    </mxGraphModel>
  </diagram>
</mxfile>
```

Figure 1.2 – *Structure XML typique d'un diagramme de classes exporté depuis draw.io*

Les éléments clés de cette structure sont :

— **mxCell** : Élément de base représentant tout objet dans le diagramme (classes, relations, textes)
— **Attribut style** : Chaîne de caractères au format CSS-like définissant l'apparence et, indirectement, la sémantique UML
— **Attributs vertex/edge** : Indiquent si la cellule est un nœud (classe) ou une arête (relation)
— **Attributs source/target** : Pour les relations, référencent les identifiants des classes liées
— **mxGeometry** : Définit la position et la taille des éléments

1.2 Notion de compilation + Analogie avec le projet

Nous expliquerons ici l'analogie fondamentale qui a guidé la conception de notre système UML2Code : **l'approche par compilation**. En effet, notre processus de transformation d'un diagramme de classes UML vers du code source peut être assimilé à la compilation d'un langage de programmation, où le diagramme UML joue le rôle de langage source et le code généré celui de langage cible.

1.2.1 Rappels sur le processus de compilation

Un **compilateur** est un programme qui traduit un code écrit dans un langage source vers un autre langage cible, généralement de plus bas niveau. Le processus de compilation classique, tel qu'illustré dans la Figure 1.3, se décompose en plusieurs phases distinctes Aho et al., 2006.

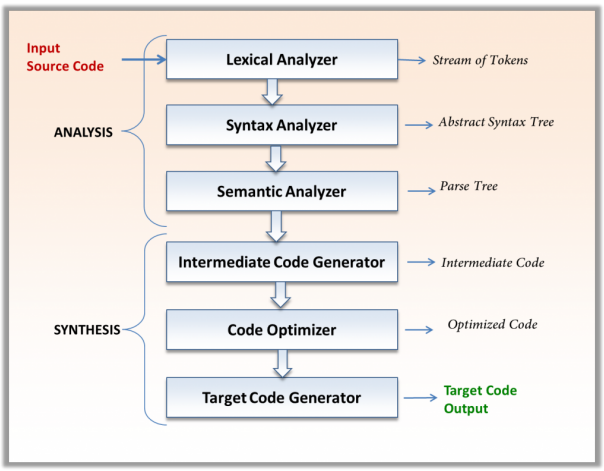

Figure 1.3 – *Les phases classiques d'un processus de compilation*

Les principales phases d'un compilateur sont :

1. **Analyse lexicale** : Décompose le code source en unités lexicales élémentaires appelées tokens (mots-clés, identificateurs, opérateurs, etc.).
2. **Analyse syntaxique** : Organise les tokens selon les règles grammaticales du langage pour construire un arbre syntaxique qui représente la structure hiérarchique du programme.
3. **Analyse sémantique** : Vérifie la cohérence sémantique du programme (typage, déclarations, portée des variables) et enrichit l'arbre syntaxique avec ces informations.
4. **Génération de code intermédiaire** : Traduit l'arbre enrichi en une représentation intermédiaire indépendante de la machine cible.
5. **Optimisation** : Améliore la représentation intermédiaire pour produire un code plus efficace.
6. **Génération de code cible** : Produit le code final dans le langage cible à partir de la représentation intermédiaire.

1.2.2 Analogie avec le projet UML2Code

Notre système UML2Code s'inspire directement de cette architecture en compilation pour transformer les diagrammes UML en code source. Le Tableau 1.1 présente l'analogie que nous avons établie entre les concepts de compilation classique et notre approche.

Concept en compilation	**Équivalent dans UML2Code**
Code source	Fichier XML du diagramme de classes draw.io
Analyse lexicale	Extraction et identification des éléments XML de base (mxCell, attributs)
Tokens	Éléments XML atomiques identifiés
Analyse syntaxique	Organisation des éléments XML en structure hiérarchique (classes, relations)
Arbre syntaxique	Graphe représentant la structure du diagramme UML
Analyse sémantique	Interprétation UML des éléments et vérification de cohérence
Représentation intermédiaire	Modèle objet UML indépendant de draw.io
Optimisation	Réorganisation et normalisation du modèle UML
Génération de code cible	Production du code Spring et Laravel

Table 1.1 – *Analogie entre compilation classique et approche UML2Code*

1.2.3 Vue d'ensemble de l'architecture UML2Code

En nous inspirant de cette analogie, nous avons structuré notre système selon une architecture modulaire en pipeline, comme illustré dans la Figure 1.4. Cette architecture sera explorée en détail dans le chapitre suivant.

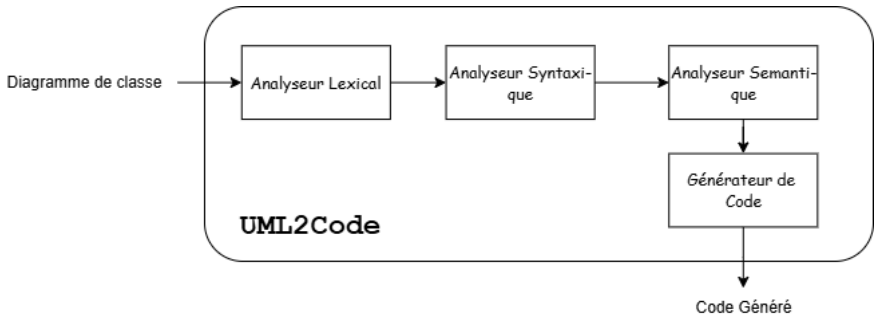

Figure 1.4 – *Architecture modulaire du système UML2Code inspirée des compilateurs*

Notre architecture se compose de quatre modules principaux, chacun correspondant à une phase du processus de compilation :

— **Analyseur lexical** : Responsable de l'extraction et de l'identification des éléments XML de base du diagramme draw.io.
— **Analyseur syntaxique** : Organise les éléments XML en une structure hiérarchique reflétant l'organisation du diagramme UML.

— **Analyseur sémantique** : Enrichit la structure avec des informations de type UML et vérifie sa cohérence.
— **Générateur de code** : Produit le code source dans les langages et frameworks cibles à partir du modèle UML intermédiaire.

L'approche par compilation s'est révélée être un cadre conceptuel puissant pour structurer notre système UML2Code. Elle nous a permis de :

— Décomposer un problème complexe en phases bien définies
— Établir une séparation claire des préoccupations
— Faciliter l'évolution et la maintenance du système
— Fournir un modèle naturel pour l'organisation multi-agent

Dans les chapitres suivants, nous détaillerons l'implémentation de chaque composant de notre architecture, en commençant par l'analyseur lexical qui constitue la première étape de notre pipeline de transformation.

2

Architecture UML2Code

L'architecture du système UML2Code s'articule autour d'une approche modulaire inspirée des principes de compilation, où chaque composant assume une responsabilité spécifique dans la chaîne de transformation du diagramme UML vers le code source. Cette architecture en pipeline garantit une séparation claire des préoccupations tout en facilitant la maintenance et l'évolutivité du système.

2.1 Analyseur lexical

L'analyseur lexical constitue la première phase du processus de transformation. Son rôle fondamental consiste à convertir le fichier XML exporté depuis draw.io en une représentation structurée exploitable par les modules suivants. Cette phase correspond à l'analyse lexicale dans la théorie des compilateurs, où le texte source est décomposé en unités lexicales atomiques.

2.1.1 Problématiques du format XML draw.io

Le format XML généré par draw.io présente des défis spécifiques qui distinguent notre approche d'un parsing XML classique. Contrairement aux formats standardisés comme XMI (XML Metadata Interchange) spécifiquement conçus pour l'échange de modèles UML, le XML de draw.io utilise le format mxGraph, initialement développé pour représenter des diagrammes génériques.

Cette particularité implique plusieurs complexités :

— **Hétérogénéité sémantique** : Les éléments UML (classes, attributs, méthodes, relations) sont tous représentés par des balises `mxCell` génériques, nécessitant une analyse contextuelle pour déterminer leur nature véritable.
— **Imbrication variable** : La structure hiérarchique ne correspond pas directement à la sémantique UML. Par exemple, les attributs d'une classe ne sont pas des enfants directs de la balise représentant la classe, mais sont liés par des références d'identifiants.

— **Encodage des propriétés visuelles** : Les propriétés sémantiques UML (type de relation, visibilité, etc.) sont encodées dans des chaînes de caractères au format CSS-like, nécessitant un parsing spécialisé.

2.1.2 Algorithme de parsing et normalisation

Le processus de parsing XML s'articule en quatre phases principales :

Algorithm 1 Parsing XML draw.io

```
1 : procedure ParseXMLDrawIO(fichier_xml)
2 :     // Phase 1 : Conversion XML vers structure de données
3 :     document_xml ← Parser(fichier_xml)
4 :     données_json ← ConvertirXMLversJSON(document_xml)
5 :     // Phase 2 : Normalisation des identifiants
6 :     données_normalisées ← NormaliserClés(données_json)
7 :     // Phase 3 : Extraction des éléments mxCell
8 :     diagramme ← ExtraireStructureDiagramme(données_normalisées)
9 :     éléments_mxcell ← ExtraireÉlémentsMxCell(diagramme)
10 :    // Phase 4 : Identification des éléments racines
11 :    (id_racine, id_sous_racine) ← IdentifierÉlémentsRacines(éléments_mxcell)
12 :    return (éléments_mxcell, id_racine, id_sous_racine)
13 : end procedure
```

Cette approche systématique permet de transformer la structure XML complexe en une représentation intermédiaire plus facilement manipulable par les phases suivantes du pipeline.

2.1.3 Identification et classification des tokens

La phase de tokenisation transforme les éléments mxCell en tokens typés selon leur rôle dans le diagramme UML. Cette classification s'appuie sur l'analyse des attributs XML et des propriétés de style.

Les critères de classification principaux sont :

— **Tokens de classe** : Éléments possédant l'attribut `vertex="1"` et un parent correspondant à la couche principale du diagramme
— **Tokens de relation** : Éléments avec `edge="1"` et des références `source` et `target` valides
— **Tokens d'attribut/méthode** : Éléments enfants des classes, différenciés par la présence ou l'absence de parenthèses dans leur valeur

Le processus de classification peut être représenté conceptuellement comme suit :

Algorithm 2 Classification des tokens

```
1 : procedure ClassifierTokens(éléments_mxcell, id_sous_racine)
2 :     tokens_classes ← ∅
3 :     tokens_relations ← ∅
4 :     tokens_membres ← ∅
5 :     for élément in éléments_mxcell do
6 :         if EstClasse(élément, id_sous_racine) then
7 :             tokens_classes ← tokens_classes ∪ {élément}
8 :         else if EstRelation(élément, id_sous_racine) then
9 :             tokens_relations ← tokens_relations ∪ {élément}
10 :        else if EstMembre(élément) then
11 :            tokens_membres ← tokens_membres ∪ {élément}
12 :        end if
13 :    end for
14 :    return (tokens_classes, tokens_relations, tokens_membres)
15 : end procedure
```

2.1.4 Format de sortie de l'analyseur lexical

L'analyseur lexical, implémenté dans la classe `Lexer`, a pour objectif de convertir un fichier XML draw.io, souvent complexe et peu structuré, en une représentation JSON exploitable. Cette étape est cruciale car elle constitue la base sur laquelle reposent toutes les analyses ultérieures, en particulier syntaxique et sémantique.

Le format XML de draw.io utilise des balises génériques `mxCell`, ce qui oblige à identifier le rôle de chaque élément (classe, relation, attribut, méthode) à partir d'heuristiques basées sur leurs styles, leurs parents, ou leur contenu textuel. Une fois ces heuristiques appliquées, l'analyseur produit une structure JSON organisée autour de deux clés : `classes` et `relationships`.

Structure du JSON généré

Listing 2.1 – *Extrait typique du format JSON retourné par le Lexer*

```
{
  "classes": {
    "5XhS": {
      "name": "User",
      "type": "class",
      "attributes": [],
      "methods": [],
      "geometry": {
        "@x": "120", "@y": "100", "@width": "160", "@height": "80"
      }
    }
  },
```

```
  "relationships": [
    {
      "id": "Y2bM",
      "name": "owns",
      "source": "5XhS",
      "target": "K1u9",
      "edge": "1",
      "style": "endArrow=diamondThin;endFill=1",
      "args": [],
      "type": "",
      "multiplicity": "",
      "source_role": null,
      "target_role": null,
      "source_multiplicity": null,
      "target_multiplicity": null,
      "is_navigable_to_source": false,
      "is_navigable_to_target": false
    }
  ]
}
```


Signification des champs

— `classes` : contient l'ensemble des classes détectées. Chaque clé correspond à l'identifiant unique de la classe dans le XML, et chaque valeur est un dictionnaire décrivant :

  — `name` : le nom de la classe extrait du contenu textuel.
  — `type` : indique s'il s'agit d'une classe, d'une interface, etc.
  — `attributes`, `methods` : initialement vides, ces champs sont remplis par des éléments analysés plus loin.
  — `geometry` : contient les coordonnées visuelles (optionnel), utilisées pour la génération graphique ou l'analyse contextuelle.

— `relationships` : chaque relation identifiée dans le diagramme est représentée par un dictionnaire contenant :

  — `source` et `target` : identifiants des classes reliées.
  — `style` : chaîne encodant les informations visuelles (flèche, remplissage, etc.) interprétées plus tard pour déterminer le type (association, composition…).
  — `multiplicity`, `role`, `navigability` : initialisés à vide, ils seront complétés ultérieurement, notamment par l'agent IA.

Utilisation et justification

Cette représentation intermédiaire permet une séparation claire entre la logique de parsing et l'analyse métier. Elle est stockée dans un fichier `lexer.json` à des fins de traçabilité et de debug, et est directement consommée par l'analyseur syntaxique.

Elle a été conçue comme un format pivot volontairement partiel mais extensible, facilitant la relecture humaine et les traitements automatisés.

2.1.5 Gestion des références et des dépendances

Un aspect critique de l'analyse lexicale concerne la résolution des références entre éléments. Les relations UML référencent leurs classes source et cible par des identifiants qui peuvent pointer vers des éléments intermédiaires plutôt que directement vers les classes.

Cette problématique nécessite un algorithme de résolution des références capable de remonter la hiérarchie des éléments pour identifier les véritables entités UML impliquées dans une relation. Le système implémente une technique de parcours itératif qui résout ces références indirectes, garantissant ainsi l'intégrité des relations dans le modèle résultant.

2.2 Analyseur syntaxique

L'analyseur syntaxique organise les tokens produits par l'analyseur lexical selon les règles structurelles du langage UML. Cette phase correspond à la construction de l'arbre syntaxique dans la théorie des compilateurs, où les tokens sont assemblés selon une grammaire formelle.

2.2.1 Modélisation de la grammaire UML

La grammaire UML que nous considérons peut être formalisée selon les règles de production suivantes en notation BNF (Backus-Naur Form) :

```
Diagramme ::= Classe* Relation*
Classe ::= Identificateur Attribut* Méthode*
Attribut ::= Visibilité Identificateur ':' Type
Méthode ::= Visibilité Identificateur '(' Paramètre* ')' ':' Type
Relation ::= TypeRelation Classe Classe Multiplicité?
TypeRelation ::= Héritage | Association | Agrégation | Composition
Visibilité ::= '+' | '-' | '#'
```

Cette grammaire capture les éléments essentiels des diagrammes de classes UML tout en restant suffisamment flexible pour accommoder les variations introduites par draw.io.

2.2.2 Construction de l'arbre syntaxique

Le processus de construction syntaxique procède par analyse descendante récursive, construisant progressivement un arbre dont les nœuds représentent les entités UML et les arêtes leurs relations de composition structurelle.

Algorithm 3 Construction de l'arbre syntaxique

```
1 : procedure ConstruireArbreSyntaxique(tokens_classes, tokens_relations, tokens_membres)
2 :     arbre_syntaxique ← NouvelArbre()
3 :     // Phase 1 : Construction des nœuds classe token_classe in tokens_classes
4 :     nœud_classe ← CréerNœudClasse(token_classe)
5 :     arbre_syntaxique.AjouterNœud(nœud_classe)
6 :     // Phase 2 : Attachement des membres aux classes token_membre in tokens_membres
7 :     classe_parent ← RésoudreClasseParent(token_membre)
8 :     if classe_parent ≠ ∅ then
9 :         if EstAttribut(token_membre) then
10 :            attribut ← CréerAttribut(token_membre)
11 :            classe_parent.AjouterAttribut(attribut)
12 :        else if EstMéthode(token_membre) then
13 :            méthode ← CréerMéthode(token_membre)
14 :            classe_parent.AjouterMéthode(méthode)
15 :        end if
16 :    end if
17 :    // Phase 3 : Création des arêtes pour les relations token_relation in tokens_relations
18 :    relation ← CréerRelation(token_relation)
19 :    arbre_syntaxique.AjouterArête(relation)
20 :    return arbre_syntaxique
21 : end procedure
```

2.2.3 Format de sortie de l'analyseur syntaxique

À partir de la structure JSON issue du Lexer, l'analyseur syntaxique applique une logique de regroupement, de typage et d'organisation pour reconstruire des entités cohérentes représentant des classes UML, leurs attributs, méthodes et relations structurées.

Cette étape est assimilable à la construction d'un arbre syntaxique abstrait, dans la tradition des compilateurs.

Structure des classes générées

Les objets retournés sont des instances de la classe `Class`, définie dans le module `models`. Ils encapsulent les membres de classe et la hiérarchie d'héritage détectée dans le diagramme.

Listing 2.2 – *Structure des objets Class retournés par l'analyseur syntaxique*

```
@dataclass
```

```
class Class:
    name: str                       # Nom de la classe
    attributes: List[Attribute]     # Liste des attributs
    methods: List[Method]           # Liste des méthodes
    aggregations: List[Attribute] # Relations d'agrégation
    compositions: List[Attribute] # Relations de composition
    parent: Optional[str]           # Classe parente (héritage)
    implements: List[str]           # Interfaces implémentées
```

Typage et analyse des signatures

Le contenu des cellules (attributs et méthodes) est analysé à l'aide d'expressions régulières adaptées aux conventions UML textuelles :

— Attribut : `+ id : int`
— Méthode : `- getName() : String`

Utilisation

Cette représentation fortement typée permet d'effectuer une analyse sémantique rigoureuse, mais aussi d'associer chaque classe à des responsabilités claires en fonction de son contenu et de ses relations (ex : entité métier, DTO, service, etc.).

2.3 Analyseur sémantique

L'analyseur sémantique constitue la troisième phase du pipeline UML2Code. Il a pour objectif d'enrichir la structure syntaxique avec des informations sémantiques permettant une génération de code réaliste, fidèle aux intentions du diagramme UML et adaptée au framework cible (ex. Spring Boot, Laravel).

Cette étape peut être vue comme un interpréteur qui transforme une simple structure syntaxique en un véritable **modèle objet métier**, intégrant des éléments comme la cardinalité des relations, la navigabilité, les rôles, et les annotations spécifiques à la plateforme.

2.3.1 Entrée et sortie

— **Entrée** : une liste d'objets `Class` issus de l'analyse syntaxique, ainsi qu'une liste brute de relations.
— **Sortie** : la même liste de classes, mais enrichie de :
  — relations correctement typées (composition, agrégation, etc.),
  — attributs ajoutés dans les classes cibles ou sources selon la navigabilité,
  — références croisées résolues et typées,
  — annotations techniques selon le framework.

2.3.2 Enrichissements appliqués

1. Interprétation des relations UML

Chaque relation est analysée pour déterminer sa nature (héritage, association, agrégation, composition, dépendance). Ce traitement repose sur :

— les styles visuels extraits depuis draw.io (flèche pleine, losange vide, etc.)
— l'orientation de la flèche et ses métadonnées
— la chaîne `style` (ex : `endArrow=diamondThin;endFill=1`)

Ce traitement permet notamment d'enrichir les champs :

— `parent` pour l'héritage,
— `aggregations` et `compositions` pour les relations fortes,
— création d'attributs d'association selon la navigabilité.

2. Intervention de l'agent IA

La qualité des relations UML extraites depuis draw.io est souvent dégradée ou ambiguë : les rôles, multiplicités et types de relations ne sont pas explicitement exprimés. Pour pallier ces lacunes, l'analyseur sémantique intègre un **agent intelligent**, basé sur un grand modèle de langage (LLM), capable d'interpréter le contexte des relations et de les enrichir de manière autonome.

Objectif. L'agent IA permet de compléter automatiquement les propriétés sémantiques manquantes dans une relation UML :

— les **multiplicités** (`1`, `0..1`, `0..*`, etc.),
— les **rôles** associés à chaque extrémité (nom du lien depuis la perspective de chaque classe),
— la **navigabilité** de la relation,
— et surtout le **type technique de la relation** (ex : `@OneToMany`, `@ManyToOne`, etc.) dans une perspective de génération de code JPA ou Eloquent.

Fonctionnement. L'agent IA est déclenché à chaque fois qu'une relation de type `association` est détectée. Il reçoit en entrée :

— le nom des deux classes concernées,
— les attributs et méthodes de chaque classe (permettant de contextualiser la relation),
— les métadonnées disponibles sur la relation (identifiants, nom, style visuel, etc.).

Ces éléments sont convertis en un prompt structuré qui est envoyé à un modèle de langage (comme GPT, Gemini, ou DeepSeek) via une API. La réponse est un JSON enrichi contenant :

— les noms de rôles à utiliser,

— les directions de navigabilité,
— les cardinalités attendues,
— et les annotations techniques à appliquer.

Listing 2.3 – *Extrait de prompt envoyé à l'agent IA*

```
{
  "source_class": {
    "name": "User",
    "attributes": ["id: Long", "username: String"]
  },
  "target_class": {
    "name": "Post",
    "attributes": ["id: Long", "content: String"]
  },
  "relation": {
    "name": "writes",
    "style": "endArrow=diamondThin;endFill=1"
  }
}
```

Listing 2.4 – *Extrait de sortie IA enrichie*

```
{
  "relationship_type": "OneToMany",
  "source_role": "author",
  "target_role": "posts",
  "source_multiplicity": "1",
  "target_multiplicity": "0..*",
  "is_navigable_to_target": true,
  "is_navigable_to_source": false
}
```

Avantages.

— **Automatisation intelligente** : permet de compléter des diagrammes partiels sans intervention manuelle.
— **Adaptabilité** : peut s'ajuster selon le framework cible (Spring, Laravel, etc.) pour proposer des annotations spécifiques.
— **Robustesse** : améliore considérablement la qualité sémantique des modèles dérivés, même à partir d'entrées imprécises.

Limites.

— **Dépendance à une API externe** : nécessite un accès internet stable et peut engendrer un coût selon l'outil utilisé.

— **Résultats non déterministes** : les réponses de l'IA peuvent varier, ce qui implique l'utilisation de règles de post-traitement pour garantir la cohérence.

Ce système permet de faire évoluer le parser UML2Code vers un interpréteur **semi-intelligent** capable de déduire des règles sémantiques à partir d'un simple graphe de classes visuel.

3. Résolution et validation des types

Le système vérifie que chaque type mentionné dans un attribut ou une méthode :

— correspond à un type primitif valide (`int`, `String`, `boolean`, etc.)
— ou correspond au nom d'une autre classe déclarée dans le modèle
— ou est une collection typée : `List<User>`, `Set<Product>`, etc.

Une table des symboles est construite pour résoudre les identifiants et assurer la cohérence des types.

2.3.3 Exemple d'objet enrichi

Listing 2.5 – *Exemple de classe après enrichissement sémantique*

```
Class(
    name="User",
    attributes=[
        Attribute(visibility="private", name="id", _type="Long"),
        Attribute(visibility="private", name="username", _type="String")
    ],
    methods=[
        Method(visibility="public", name="getUsername", _type="String", args=[])
    ],
    aggregations=[
        Attribute(visibility="private", name="posts", _type="List<Post>")
    ],
    compositions=[
        Attribute(visibility="private", name="profile", _type="Profile")
    ],
    parent="Person",
    implements=["Serializable"]
)
```

2.3.4 Utilisation

Le modèle enrichi est encapsulé dans un objet `Project`, qui constitue la donnée d'entrée principale du générateur de code. C'est ce modèle qui détermine :

— la structure des entités métiers
— les annotations à appliquer pour chaque relation
— les couches à générer dans le squelette final (domain, controller, repository, etc.)

Ce découplage garantit une grande extensibilité : un même modèle sémantique peut être réutilisé pour générer du code pour différents frameworks ou architectures.

2.4 Générateur de code

Le **générateur de code** transforme le modèle sémantique enrichi en code source fonctionnel pour les frameworks ciblés. Cette phase constitue l'aboutissement du pipeline de transformation et doit produire du code respectant les conventions et bonnes pratiques de chaque plateforme.

2.4.1 Architecture du système de génération

Le système de génération adopte une architecture basée sur le pattern Strategy, où différents générateurs spécialisés peuvent être sélectionnés selon le framework cible. Cette architecture garantit l'extensibilité du système tout en maintenant une interface uniforme.

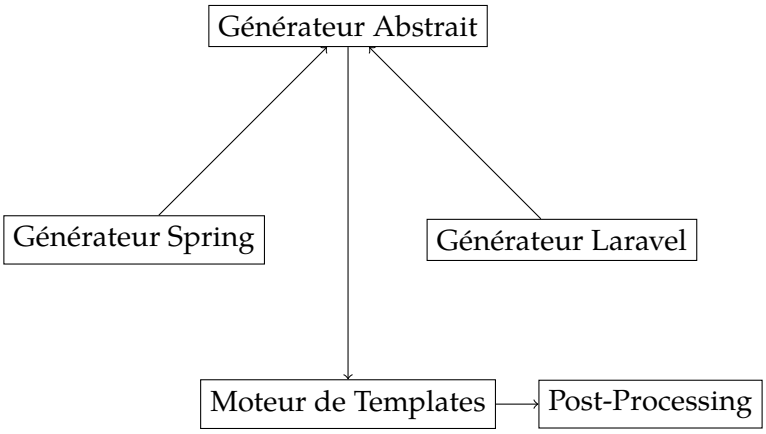

Figure 2.1 – *Architecture du système de génération*

2.4.2 Stratégies de génération par framework

Chaque framework impose ses propres conventions et patterns architecturaux. Le système de génération implémente des stratégies spécialisées qui adaptent le modèle UML générique aux spécificités de chaque plateforme.

Stratégie Spring Boot

- Application du pattern Repository/Service/Controller
- Génération des annotations JPA pour la persistance
- Configuration automatique des beans Spring
- Implémentation des patterns de validation Bean Validation

Stratégie Laravel

— Application du pattern MVC Laravel
— Génération des modèles Eloquent avec relations
— Configuration des migrations de base de données
— Implémentation des FormRequests pour la validation

2.4.3 Système de templates et moteur de génération

Le moteur de génération utilise un système de templates sophistiqué permettant la personnalisation fine du code produit. Ce système s'appuie sur le moteur Jinja2 étendu avec des fonctions spécialisées pour la génération de code.

Le processus de génération basé sur templates peut être conceptualisé comme suit :

Algorithm 4 Génération basée sur templates

```
1 : procedure GénérerCode(modèle_sémantique, template_framework, répertoire_sortie)
2 :     moteur_templates ← InitialiserMoteurJinja2()
3 :     contexte_génération ← ConstruireContexte(modèle_sémantique)
          classe in modèle_sémantique.classes
4 :     // Sélection du template approprié
5 :     template ← SélectionnerTemplate(classe, template_framework)
6 :     // Génération du code
7 :     code_généré ← moteur_templates.Rendre(template, classe, contexte_génération)
8 :     // Optimisation et post-processing
9 :     code_optimisé ← OptimiserCode(code_généré)
10 :    code_formaté ← FormaterCode(code_optimisé)
11 :    // Écriture du fichier
12 :    nom_fichier ← ConstruireNomFichier(classe)
13 :    chemin_complet ← répertoire_sortie / nom_fichier
14 :    ÉcrireFichier(chemin_complet, code_formaté)
15 : end procedure
```

2.4.4 Génération des couches architecturales

Le générateur implémente une approche par couches, générant séparément les différents niveaux de l'architecture logicielle. Cette séparation permet d'optimiser le code généré selon les responsabilités de chaque couche.

Couche Domaine

— Entités métier avec validation intégrée
— Services domaine encapsulant la logique métier
— Interfaces de repository définissant les contrats de persistance

— Exceptions métier typées

Couche Infrastructure

— Implémentations des repositories avec technologies de persistance
— Configuration des sources de données et connexions
— Adaptateurs pour les services externes
— Utilitaires transversaux (sérialisation, validation)

Couche Présentation

— Contrôleurs REST avec documentation OpenAPI
— DTOs de transfert avec validation
— Handlers d'exceptions globaux
— Configuration de sécurité

Table 2.1 – *Résumé des étapes du pipeline UML2Code*

Étape	**Entrée**	**Sortie**	**Rôle**
Lexer	XML draw.io	JSON avec `classes` et `relationships`	Transforme le XML brut en structure normalisée interprétable.
Analyseur syntaxique	JSON normalisé	Liste de `Class` et `Relationship` (non enrichis)	Typage et structuration du modèle UML, extraction des signatures.
Analyseur sémantique	Liste d'objets `Class`, `Relationship`	Liste de `Class` enrichis (relations interprétées, rôles, types, annotations)	Ajout d'intelligence métier, validation des types, appel à l'IA pour compléter le modèle.
Générateur de code	Objet `Project`	Fichiers Java (ou autre) dans architecture cible	Produit le code source prêt à être compilé, organisé par couches (Domain, Infrastructure, Presentation).

3

Exemple d'utilisation

4

Perspectives : prise en charge de AUML

Conclusion

Le projet UML2Code représente une avancée significative dans l'automatisation de la transformation des diagrammes de classes UML en code source fonctionnel. En s'appuyant sur une architecture inspirée des compilateurs, le système décompose le processus en phases distinctes et modulaires : analyse lexicale, syntaxique, sémantique, et génération de code. Cette approche permet non seulement de garantir une traduction fidèle des modèles UML, mais aussi d'assurer une intégration harmonieuse avec des frameworks populaires tels que Spring pour Java et Laravel pour PHP.

Les principaux défis relevés dans ce projet incluent la gestion du format XML spécifique de draw.io, l'interprétation précise des styles visuels pour en extraire la sémantique UML, et la génération de code optimisé et conforme aux bonnes pratiques des frameworks cibles. Les solutions proposées, notamment l'utilisation d'algorithmes de parsing et de classification avancés, ainsi que l'adoption d'un système de templates flexible, ont démontré leur efficacité pour produire un code de qualité tout en respectant les contraintes du modèle original.

Les perspectives d'évolution pour UML2Code sont prometteuses, avec notamment l'extension à d'autres types de diagrammes UML (comme les diagrammes de séquence ou d'état) et la prise en charge de frameworks supplémentaires. L'intégration d'une approche multi-agent pourrait également renforcer l'autonomie et la collaboration entre les différentes phases du processus.

En conclusion, UML2Code illustre comment l'automatisation des tâches répétitives dans le développement logiciel peut améliorer la productivité, réduire les erreurs humaines et renforcer la cohérence entre la conception et l'implémentation. Ce projet ouvre la voie à de futures innovations dans le domaine de l'ingénierie dirigée par les modèles, tout en offrant un outil pratique et accessible aux équipes de développement.

Bibliographie

Aho, Alfred V. et al. (2006). *Compilateurs : Principes, techniques et outils*. 2ème. Pearson Education.

Booch, Grady, James Rumbaugh et Ivar Jacobson (2005). *The Unified Modeling Language User Guide*. 2nd. Addison-Wesley.

Draw.io (2022). *Export your diagram to an XML file*. `https://www.drawio.com/doc/faq/export-to-xml`. Accessed : 2025-06-14.

Draw.io (2023). *Create UML class diagrams*. `https://www.drawio.com/blog/uml-class-diagrams`. Accessed : 2025-06-14.

Fowler, Martin (2003). *UML Distilled : A Brief Guide to the Standard Object Modeling Language*. 3rd. Addison-Wesley.

Wang, Linjie et al. (2024). « How LLMs Aid in UML Modeling : An Exploratory Study with Novice Analysts ». In : *arXiv preprint arXiv :2404.17739*. arXiv : `2404.17739 [cs.SE]`.