UNIVERSITÉ DU QUÉBEC À MONTRÉAL

OBJECTMACRO 2, GÉNÉRATEUR DE TEXTES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

TRAN VAN BA VU ANH LÂM

MARS 2019

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui m'ont suivi tout au long de ma maîtrise particulièrement Étienne qui a pu m'accompagner et m'instruire. Je le remercie également pour les longs débats et les analogies à sa vie personnelle qui ont été très intéressants.

Je remercie Mathilde qui a pu me supporter, m'aider, me conseiller et me motiver depuis ces quelques années à la maîtrise (surtout pour les tâches ménagères). Je la remercie encore une fois parce que, sans elle, je ne serais pas peut être pas parti au Canada pour faire la maîtrise.

Je remercie aussi mes amis : Vincent, Jehan, Agathe, Valentin, Ellen, Théo, Mathieu, Leslie, Kenan et Edouard qui m'ont souvent demandé des nouvelles de mon mémoire et qui m'ont aidé à me décontrater avec des bières, un nombre incalculable de barbecues, des soirées jeux de société, des poutines et des sessions d'escalade.

Je tiens à remercier les personnes de mon laboratoire : Florian, Quentin et Mélanie (qui n'est pas vraiment du laboratoire) qui m'ont supporté et aidé pour la rédaction de ce mémoire tout en restant optimiste (sarcasme) sur la date du dépôt de ce mémoire.

Je tiens particulièrement à remercier également ma mère, mon père, mes frères et ma soeur qui m'ont encouragé et soutenu dans mes études au Canada. Je remercie encore une fois ma famille avec qui j'ai traversé une période très difficile au cours de cette maîtrise.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

TABLE DES FIGURES

LISTINGS

RÉSUMÉ

Nous présentons dans ce mémoire une nouvelle version d'ObjectMacro, un générateur de générateurs de textes. ObjectMacro crée une bibliothèque de classes, à partir d'un fichier de macros, qui sera utilisée par le développeur pour générer du texte.

Cette nouvelle version introduit un langage déclaratif et épuré permettant d'avoir des macros lisibles. Le nouveau modèle objet robuste et flexible facilite la génération de textes pour le développeur. Le modèle s'occupe de toute la partie construction de textes tout en vérifiant statiquement et dynamiquement les éléments ajoutés par l'utilisateur.

Avec cette nouvelle version plus flexible d'ObjectMacro, des cycles statiques ou dynamiques peuvent être introduits par mégarde par le développeur. ObjectMacro détecte les cycles statiques en s'aidant de l'algorithme de Tarjan. Pour les cycles dynamiques, deux algorithmes de détection incrémentale de cycles sont présentés dans ce mémoire dont l'un des deux est expérimenté et comparé avec un algorithme naïf pour conclure que l'algorithme naïf est le plus efficace compte tenu de l'usage anticipé d'ObjectMacro.

Les informations d'une macro peuvent être transmises d'une macro à une autre dans le but de factoriser le corps d'une macro en plusieurs macros réutilisables. Nous proposons la notion d'internes qui sont des paramètres dont le texte dépend du contexte dans lequel l'interne est assigné.

Nous proposons un système permettant de définir différentes versions pour une

même macro pour donner au développeur la capacité de générer du texte différent en utilisant le même modèle objet.

Nous comparons ObjectMacro avec d'autres outils de génération de textes. Nous trouvons que, contrairement à ObjectMacro qui analyse et vérifie statiquement les macros, ces outils analysent dynamiquement les patrons de textes et ne font pas de détection de cycles.

Mots clés : Générateurs de textes, ObjectMacro, SableCC, Macro, Modèle objet, Patron de textes.

INTRODUCTION

De nos jours, la génération de textes est utilisée pour simplifier le travail des développeurs où nous pouvons la retrouver dans différents contextes. Notamment, dans le contexte *Web*, la génération peut correspondre à la génération de plusieurs pages de profils dont la structure ne change pas contrairement aux données de l'utilisateur. Dans le contexte de la compilation, du code JavaDoc est compilé pour générer des pages *web* dont le format des pages reste identique alors que le nom des paramètres ou la description peuvent être différents.

Un générateur de textes permet à un utilisateur de produire du texte dont la structure reste identique tandis que les données contenues peuvent être différentes.

De nombreux outils existent permettant la génération de textes notamment les moteurs de patrons tels que StringTemplate ou Velocity. Dans ces moteurs, les patrons de textes sont compilés lors de la génération du texte.

Le but de ce mémoire est de présenter une nouvelle version du générateur de textes ObjectMacro. ObjectMacro s'appuie sur l'outil SableCC. Ce dernier est un générateur de compilateur. Il permet de générer de analyseurs lexicaux, syntaxiques et des arbres syntaxiques. SableCC utilise une ancienne version d'ObjectMacro pour réaliser la génération de codes. L'ancienne version d'ObjectMacro ne répond plus aux attentes de l'auteur ; c'est pourquoi, ce sujet de maîtrise consiste à développer une nouvelle version d'ObjectMacro.

ObjectMacro est considéré comme un générateur de générateurs de textes puisqu'il produit une bibliothèque de classes dont l'utilisateur se sert pour générer du texte.

L'approche d'ObjectMacro pour générer du texte se distingue des approches utilisées dans les autres moteurs de patrons. ObjectMacro précompile le fichier de patrons pour produire une représentation objet des patrons. Le texte et les règles d'affichages contenus dans le patron sont directement encodés dans la bibliothèque de classes sans que l'utilisateur en prenne compte lors du développement de son générateur de textes. Cela sépare de manière explicite la génération du texte gérée par le modèle et la logique métier développée par l'utilisateur.

0.1 Objectifs

L'objectif général du sujet de maîtrise est le développement d'une nouvelle version complète d'ObjectMacro. Plusieurs objectifs découlent de cet objectif. Les objectifs principaux sont de créer un langage simple et épuré permettant la déclaration des macros et de créer un modèle objet plus puissant et plus flexible que la première version d'ObjectMacro. Un des objectifs est de garder la séparation entre la définition des macros et la génération de textes présente dans la première version d'ObjectMacro. Par ailleurs, nous souhaitons donner la possibilité à un utilisateur de générer du texte différent en utilisant un seul modèle objet.

0.2 Approche utilisée

Afin de choisir les concepts à retenir dans ObjectMacro 2, des expérimentations ont été effectuées. Pour cela, les idées ont été implémentées et par la suite vérifiées afin qu'elles correspondent aux critères recherchés.

Le développement d'ObjectMacro 2 a été effectué avec ObjectMacro 1 pour la génération de code. Par la suite, ObjectMacro 1 a été remplacé par ObjectMacro 2 pour vérifier que le code généré est fonctionnel. À chaque modification du modèle objet, la version du modèle dans le générateur de modèles a été remplacée par la nouvelle version.

0.3 Contributions

Sommairement les contributions de ce mémoire sont :

- L'élaboration d'un modèle objet robuste et flexible pour faciliter la génération de textes.
- La mise en place d'une politique *Fail-Fast* (Shore, 2004) dans l'ensemble du système incluant lors de la détection dynamique de cycles.
- La réalisation d'expérimentations et de comparaisons des algorithmes existants de détection incrémentale de cycles par rapport à un algorithme naïf.
- L'introduction des internes permettant de factoriser une macro et de transférer des informations d'une macro à une autre sans que l'utilisateur ait à s'en occuper.
- La conception d'un système de versions de macro permettant de générer du texte différent avec un même générateur de textes.
- La conception d'une syntaxe épurée permettant de décrire simplement des macros et leur corps.

0.4 Remarque

Tout au long de ce mémoire, lorsque nous évoquons ObjectMacro, nous faisons référence à la deuxième version d'ObjectMacro mis à part les cas dans lesquels une autre version est explicitement mentionnée.

0.5 Structure du mémoire

Dans le chapitre 1, nous commençons par présenter ObjectMacro dans son ensemble en décrivant le processus de génération du texte à partir d'un fichier de macros. Par ailleurs, nous ferons une description exhaustive des fonctionnalités permettant la définition des macros et l'utilisation de la bibliothèque de classes. Le chapitre 2 présente en détail le modèle objet généré par ObjectMacro, ce qui

permettra de comprendre comment le modèle est organisé pour générer du texte. Dans le chapitre 3, nous présenterons les différents cycles qui peuvent être rencontrés et les algorithmes mis en place pour détecter ces cycles. Dans le chapitre 4, nous présenterons les paramètres contextuels définis dans les macros pour montrer le gain en flexibilité apporté. Dans le chapitre 5, nous présenterons comment les versions permettent au développeur de générer du texte différent sans modifier son générateur de textes. Dans le chapitre 6, nous présenterons trois logiciels de génération de textes pour voir les différences, leurs avantages et inconvénients par rapport à ObjectMacro. Pour terminer, nous ferons une synthèse des différents points abordés dans ce mémoire et montrerons des possibilités de travaux futurs.

CHAPITRE I

OBJECTMACRO 2

Dans ce chapitre, nous allons présenter le fonctionnement général d'ObjectMacro en passant par la déclaration de macros jusqu'à la génération du texte. Cela nous permettra de comprendre comment utiliser ObjectMacro et générer le texte d'une macro.

Le reste de ce chapitre est organisé comme suit. Dans la section 1.1, nous allons présenter comment le logiciel ObjectMacro fonctionne dans son ensemble. Dans la section 1.2, nous présenterons un exemple complet d'utilisation du logiciel en partant de la définition d'une macro jusqu'à la génération du texte. Dans la section 1.3, nous allons présenter de façon détaillé comment utiliser le logiciel ObjectMacro par le biais de différents exemples pour générer du texte. Dans la section 1.4, nous présenterons d'autres approches pour construire une macro. Enfin, dans la section 1.5, nous allons conclure ce chapitre en synthétisant les différents points évoqués.

1.1 Fonctionnement général d'ObjectMacro

ObjectMacro est un logiciel de génération de textes offrant un langage permettant de définir des patrons de texte ou `macros`. ObjectMacro prend en entrée un fichier de macros pour générer en sortie une bibliothèque. Cette bibliothèque est une re-

présentation des macros sous forme de classes. L'utilisateur d'ObjectMacro couple cette bibliothèque avec des données ou un programme pour générer du texte qu'il désire.

La figure 1.1 illustre le processus de génération de textes. Dans un premier temps, le fichier de macros est lu et vérifié par ObjectMacro pour s'assurer que les macros définies soient lexicalement, syntaxiquement et sémantiquement correctes. Dans un second temps, une représentation des macros sous forme de bibliothèque de classes, permettant la génération du texte, est générée par ObjectMacro. L'utilisateur d'ObjectMacro associe des données ou un programme pour générer du texte. Nous pouvons considérer qu'ObjectMacro est un générateur de générateur de textes.

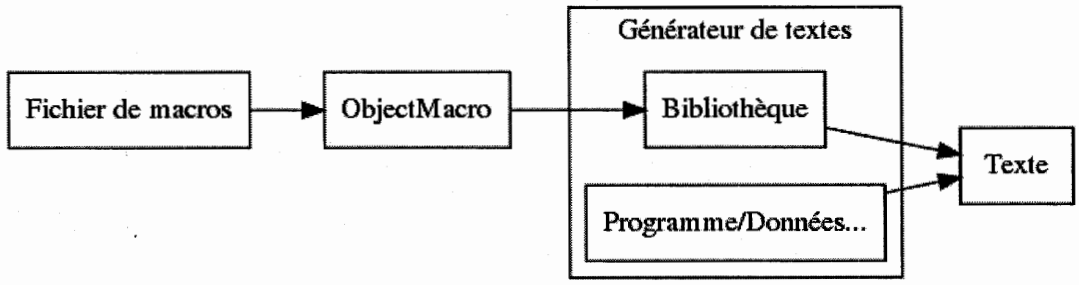

FIGURE 1.1: Processus de génération de textes

1.2 Différentes étapes de génération du texte

Dans cette section, nous allons présenter en détail les différentes étapes pour générer du texte à partir d'un fichier de macros.

Dans un premier temps, il faut définir des macros dans un fichier nommé par exemple `mon_premier_exemple.objectmacro`. Dans notre exemple du listing 1.1, nous définissons une macro simple `presentation_club` qui liste les clubs de sports d'une école.

```
1  Macro presentation_club
2          Param
3              nom_ecole: String;
4              clubs: String, separator="\n";
5  {Body}
6  Liste des clubs a l'ecole {nom_ecole} :
7  {Indent: "\t -"}
8  {clubs}
9  {End}
10 {End}
```

Listing 1.1: Définition de la macro `presentation_club`

Après avoir défini la macro, il faut compiler le fichier en utilisant la ligne de commande comme dans le listing 1.2. Il est possible de générer le modèle objet dans différents langages. Dans notre cas, nous souhaitons générer du Java. C'est pourquoi, il est nécessaire de spécifier le paquetage des classes générées. La commande génère un certain nombre de classes dont une classe liée à la macro `presentation_club` dans le dossier `src` et sous le paquetage `example.macro`. Une classe est générée pour chacune des macros décrites dans le fichier de macros. Dans notre cas, la seule classe de macro générée est donc `MPresentationClub` pour la macro `presentation_club`.

```
$ java -jar objectmacro.jar --target=java --package=example.macro --destination=
    src
```

Listing 1.2: Exemple de commande pour générer le modèle objet

Dans le paquetage `example`, nous créons une classe avec une méthode `main` comme dans le listing 1.3. Le but de la méthode `main` est d'utiliser la macro `presentation_club` pour générer du texte listant les clubs créés dans une école.

Pour compiler le programme, nous utilisons la commande du listing 1.4. C'est ensuite avec la commande du listing 1.5 que nous exécutons le programme. Le

```
1  package example;
2
3  import example.macro.*;
4
5  public class Example {
6
7    public static void main (String[] args) {
8      Macros macros = new Macros();
9
10     MPresentationClub mPresentationClub = macros.newPresentationClub();
11     mPresentationClub.addNomEcole("Sacré-Coeur");
12
13     mPresentationClub.addClubs("Club d'échec du Sacré-Coeur");
14     mPresentationClub.addClubs("Club de jeux de société");
15     mPresentationClub.addClubs("Club de football");
16
17     System.out.println(mPresentationClub.build());
18   }
19 }
```

Listing 1.3: Exemple d'utilisation de la bibliothèque générée

programme affiche le résultat du listing 1.6.

Présentement, les fichiers de macros sont limités aux caractères `ASCII` tandis que Java prend en compte les caractères Unicode. C'est pourquoi, dans le listing 1.6, le texte obtenu du code Java contient des accents alors que le texte des macros n'a pas d'accent.

1.3 La génération du texte

Dans cette section, nous allons présenter la définition et l'utilisation des macros. Pour cela, nous verrons comment déclarer une macro à la section 1.3.1, ses paramètres à la section 1.3.2, et comment utiliser ces macros dans le modèle objet généré tout au long de cette section. Nous allons présenter à la section 1.3.3 comment définir le corps de la macro et les différents éléments contenus dans le corps

```
$ javac example/Example.java -d classes
```

Listing 1.4: Compilation du programme

```
$ java -cp classes example.Example
```

Listing 1.5: Exécution du programme

```
Liste des clubs a l'ecole Sacré-Coeur :
    - Club d'échec du Sacré-Coeur
    - Club de jeux de société
    - Club de football
```

Listing 1.6: Sortie standard du programme

d'une macro en plus du texte.

1.3.1 La déclaration d'une macro

Comme nous pouvons le voir dans le listing 1.7, une macro est définie minimalement par le mot réservé `Macro`, un nom et un corps. Le corps d'une macro commence par la commande `{Body}` et se termine avec la commande `{End}`. Dans ce corps, nous retrouvons le texte de la macro.

```
1 Macro salutation
2 {Body}
3 Bonjour !
4 {End}
```

Listing 1.7: Déclaration d'une macro

Pour chacune des macros, nous obtenons une classe. Pour la macro `salutation`, nous obtenons la classe `MSalutation`. Pour créer des nouveaux objets, il suffit d'initialiser une usine à objets nommée `Macros` comme nous pouvons le voir dans le listing 1.8. Nous présenterons en détail dans le chapitre 5 ce qu'est l'usine à objets.

C'est ensuite avec l'appel de la méthode `build` sur l'instance de `MSalutation` que nous obtenons le texte représenté dans le listing 1.9.

```
1 Macros macros = new Macros();
2 MSalutation mSalutation = macros.newSalutation();
3 System.out.println(mSalutation.build());
```

Listing 1.8: Génération du texte de la macro `salutation`

```
Bonjour !
```

Listing 1.9: Résultat après exécution du listing 1.8

1.3.2 La définition des paramètres

Une macro peut avoir des paramètres. Chaque paramètre correspond à une liste d'éléments. Il y a deux types de paramètres ; ceux de type chaîne de caractères et les paramètres de type macro. Il faut utiliser le modèle objet généré à partir des macros pour alimenter ces listes d'éléments. Nous allons voir un exemple pour les paramètres de type chaîne de caractères ainsi qu'un autre exemple pour les paramètres de type macro.

1.3.2.1 Les paramètres de type chaîne de caractères

Le listing 1.10 nous montre comment définir un paramètre de type chaîne de caractères et comment utiliser ce paramètre dans le corps d'une macro. Pour alimenter ces paramètres avec des données dans le générateur de textes, il suffit de faire appel aux méthodes d'ajouts commençant toutes par le préfixe `add` auquel s'ajoute le nom du paramètre concerné. Le listing 1.11 représente un exemple de programme permettant d'ajouter des éléments de type chaîne de caractères dans une instance de macro. Le résultat est illustré dans le listing 1.12.

```
1 Macro salutation
2     Param
3         prenom : String;
4         nom : String;
5 {Body}
6 Bonjour {prenom} {nom} !
7 {End}
```

Listing 1.10: Déclaration de paramètres chaînes de caractères

```
1 Macros macros = new Macros();
2 MSalutation mSalutation = macros.newSalutation();
3
4 mSalutation.addPrenom("Pierre");
5 mSalutation.addNom("Londubas");
6
7 System.out.println(mSalutation.build());
```

Listing 1.11: Ajout d'éléments de type chaîne de caractères

```
Bonjour Pierre Londubas !
```

Listing 1.12: Résultat de l'ajout de chaînes de caractère

1.3.2.2 Les paramètres de type macro

Un paramètre est considéré de type macro lorsque sa définition correspond à un ou plusieurs noms de macros. Nous disons qu'un paramètre fait référence à des macros. Un paramètre ne peut pas à la fois faire référence à des macros et être de type chaîne de caractères.

Comme nous pouvons remarquer dans le listing 1.13, `sports_pratiques` fait référence à une autre macro : `sport` qui possède un paramètre de type chaîne de caractères. Dans le générateur de textes, il faut également faire appel aux méthodes d'ajouts comme dans le listing 1.14. Dans le cas des macros, une nouvelle instance de `MSport` doit être créée pour l'ajouter à l'instance `MPresentation`.

```
Macro presentation
    Param
        sports_pratiques: sport;
{Body}
Bonjour !
{sports_pratiques}
{End}

Macro sport
    Param
        nom_sport : String;
{Body}
Je pratique {nom_sport}.

{End}
```

Listing 1.13: Déclaration de paramètres de type macros

```
...
MPresentation mPresentation = macros.newPresentation();
MSport mSport = macros.newSport();
mSport.addNomSport("du rugby");
mPresentation.addSportsPratiques(mSport);

mSport = macros.newSport();
mSport.addNomSport("de l'escalade");
mPresentation.addSportsPratiques(mSport);

System.out.println(mPresentation.build());
```

Listing 1.14: Ajout d'éléments de type macro

```
Bonjour !
Je pratique du rugby.
Je pratique de l'escalade.
```

Listing 1.15: Résultat de l'exécution du listing 1.14

Le texte de chaque instance de la macro `sport` se termine par un retour à la ligne. Ce retour chariot se trouve dans la définition de la macro `sport` à la ligne 14 du listing 1.13 causant un retour à la ligne pour chaque instance. Ce retour à la ligne apparaîtra à chaque référence de la macro `sport`. En revanche, ce retour à ligne n'est pas désiré sur toutes les références ; il cause l'ajout d'une ligne vide finale dans le listing 1.15. C'est pourquoi, des options peuvent être définies localement pour contrôler le rendu du texte d'une référence. Nous allons présenter dans la section 1.3.2.3 ces options.

1.3.2.3 Les options d'affichage

Les options d'affichage permettent de formater le rendu du texte généré de manière locale, c'est-à-dire, seulement sur le paramètre où l'option est définie. Pour le moment, quatre options sont définies dans ObjectMacro :

- `separator`
- `before_first`
- `after_last`
- `none`

1.3.2.4 L'option `separator`

L'option `separator` permet d'ajouter le texte spécifié en tant que séparateur de chaque élément contenu dans un paramètre. Dans le cas où la liste ne contient qu'un seul élément, le séparateur n'est pas affiché. Dans le cas du listing 1.16, un retour à la ligne est ajouté entre les instances de `sport` contenues dans le paramètre `sports_pratiques`, mais nous pouvons noter qu'il n'est pas ajouté après la dernière instance de `sport` dans le listing 1.17.

```
1  Macro presentation
2      Param
3          sports_pratiques: sport, separator="\n";
4  {Body}
5  Bonjour !
6  {sports_pratiques}
7  {End}
8
9  Macro sport
10     Param
11         nom_sport : String;
12 {Body}
13 Je pratique {nom_sport}.
14 {End}
```

Listing 1.16: L'option `separator`

```
Bonjour !
Je pratique du rugby.
Je pratique de l'escalade.
```

Listing 1.17: Résultat de l'exécution du listing 1.14 avec les macros du listing 1.16

1.3.2.5 L'option `before_first`

L'option `before_first` permet d'ajouter le texte spécifié avant le premier élément du paramètre. Lorsque la liste contient au minimum un élément, le texte est affiché avant le texte du premier élément. Si la liste est vide, le texte n'apparaît pas. L'option `before_first` est appliquée sur le paramètre `sports_pratiques` dans le listing 1.18. Le texte « `Liste de mes sports :` » s'affiche avant le texte du paramètre `sports_pratiques`. Le résultat est illustré dans le listing 1.19.

1.3.2.6 L'option `after_last`

L'option `after_last` permet d'ajouter le texte spécifié après le dernier élément de la liste. Lorsque la liste contient au minimum un élément, le texte est affiché

```
1  Macro presentation
2      Param
3          sports_pratiques: sport, before_first="Liste de mes sports : \n",
               separator="\n";
4  {Body}
5  Bonjour !
6  {sports_pratiques}
7  {End}
8
9  Macro sport
10     Param
11         nom_sport : String;
12 {Body}
13 Je pratique {nom_sport}.
14 {End}
```

Listing 1.18: L'option `before_first`

```
Bonjour !
Liste de mes sports :
Je pratique du rugby.
Je pratique de l'escalade.
```

Listing 1.19: Résultat de l'exécution du listing 1.14 avec les macros du listing 1.18

après le texte du dernier élément. Si la liste est vide, le texte n'est pas affiché. L'option `after_last` est appliquée sur le paramètre `sports_pratiques` dans le listing 1.20. Le texte « `Ce sont mes sports favoris.` » s'affiche après le texte du dernier élément du paramètre `sports_pratiques`. Le résultat est illustré dans le listing 1.21.

1.3.2.7 L'option none

L'option `none` permet d'ajouter le texte spécifié lorsqu'un paramètre ne contient aucun élément. L'option `none` est appliquée au paramètre `sports_pratiques` dans le listing 1.22. Pour que l'option `none` s'applique et que le texte « `Je ne pratique`

```
1  Macro presentation
2      Param
3          sports_pratiques: sport, after_last="\nCe sont mes sports favoris.",
               separator="\n";
4  {Body}
5  Bonjour !
6  {sports_pratiques}
7  {End}
8
9  Macro sport
10     Param
11         nom_sport : String;
12 {Body}
13 Je pratique {nom_sport}.
14 {End}
```

Listing 1.20: L'option `after_last`

```
Bonjour !
Je pratique du rugby.
Je pratique de l'escalade.
Ce sont mes sports favoris.
```

Listing 1.21: Résultat de l'exécution du listing 1.14 avec les macros du listing 1.20

`aucun sport.` » s'affiche, le générateur de textes ne doit pas ajouter d'élément dans le paramètre `sports_pratiques` comme dans le listing 1.23. Le résultat est illustré dans le listing 1.24.

1.3.2.8 L'intérêt des options d'affichage

Les options d'affichage permettent de contrôler le rendu du texte (cf. section 1.3.2.3). L'autre but des options est d'enlever toute la complexité liée à la présentation du texte dans le générateur pour le déplacer dans la déclaration des macros. Cela évite au développeur du générateur de textes de se soucier comment le texte va être généré. Il ne se soucie que des éléments à ajouter dans les listes.

```
1  Macro presentation
2      Param
3          sports_pratiques: sport, none="Je ne pratique aucun sport.", separator="\
               n";
4  {Body}
5  Bonjour !
6  {sports_pratiques}
7  {End}
8
9  Macro sport
10     Param
11         nom_sport : String;
12 {Body}
13 Je pratique {nom_sport}.
14 {End}
```

Listing 1.22: L'option none

```
1 Macros macros = new Macros();
2 MPresentation mPresentation = macros.newPresentation();
3 System.out.println(mPresentation.build());
```

Listing 1.23: Générateur de textes sans ajout d'instances de la macro `sport`

```
Bonjour !
Je ne pratique aucun sport.
```

Listing 1.24: Résultat après exécution du listing 1.23 avec les macros du listing 1.22

Dans le listing 1.25, quatre options sont définies sur le paramètre `sports_pratiques`. Dans le listing 1.26, nous ajoutons seulement un seul élément dans la liste du paramètre `sports_pratiques`.

Comme nous pouvons voir dans le listing 1.28, le texte des options `before_first` et `after_last` apparaissent car il y a un seul élément. En ajoutant deux éléments à la liste des `sports_pratiques`, comme dans le listing 1.27, les options précédentes

```
Macro presentation
    Param
        sports_pratiques: sport, none="Je ne pratique aucun sport", separator="\n
            ", before_first="Je vais vous parler de mon/mes sport(s) que je
            pratique.\n", after_last="\nEt vous quels sont les sports que vous
            pratiquez ?";
{Body}
Bonjour !
{sports_pratiques}
{End}

Macro sport
...
```

Listing 1.25: Exemple global des options

```
Macros m = new Macros();
MPresentation mPresentation = m.newPresentation();
MSport mSport = m.newSport("du rugby");
mPresentation.addSportsPratiques(mSport);

System.out.println(mPresentation.build());
```

Listing 1.26: Ajout d'un seul élément dans le paramètre `sports_pratiques`

```
...
MPresentation mPresentation = macros.newPresentation();
MSport mSport = macros.newSport();
mSport.addNomSport("du rugby");
mPresentation.addSportsPratiques(mSport);

mSport = macros.newSport();
mSport.addNomSport("de l'escalade");
mPresentation.addSportsPratiques(mSport);

System.out.println(mPresentation.build());
```

Listing 1.27: Ajout de deux éléments dans le paramètre `sports_pratiques`

```
Bonjour !
Je vais vous parler de mon/mes sport(s) que je pratique.
Je pratique du rugby.
Et vous quels sont les sports que vous pratiquez ?
```

Listing 1.28: Résultat de l'exécution du listing 1.26 avec les macros du listing 1.25

```
Bonjour !
Je vais vous parler de mon/mes sport(s) que je pratique.
Je pratique du rugby.
Je pratique de l'escalade.
Et vous quels sont les sports que vous pratiquez ?
```

Listing 1.29: Résultat de l'exécution du listing 1.27 avec les macros du listing 1.25

```
Bonjour !
Je ne pratique aucun sport.
```

Listing 1.30: Résultat de l'exécution du listing 1.23 avec les macros du listing 1.25

ainsi que l'option **separator** sont appliquées comme nous pouvons le voir dans le listing 1.29.

Dans le cas où aucun élément est ajouté, comme dans le listing 1.23, les deux options précédentes ne s'affichent pas mais l'option **none** est appliquée. le listing 1.30 illustre le résultat avec aucun élément dans le paramètre. L'application des options dépend du nombre d'éléments ajoutés dans la liste du paramètre **sports_pratiques**, la façon dont est utilisée le modèle objet impacte alors l'application des options.

1.3.3 La définition du corps de la macro

Dans cette section, nous allons présenter la composition du corps de la macro. Nous allons expliquer les directives retrouvées dans le corps de cette macro. Par la suite, nous verrons comment ObjectMacro différencie le texte des directives.

1.3.3.1 La composition du corps

Le corps d'une macro est composé de textes, de différentes commandes (`directives`) d'ObjectMacro ainsi que des références de paramètre. Tous les éléments du corps qui ne sont pas du texte se distinguent par des accolades comme nous pouvons le voir dans le corps de la macro du listing 1.25 à la ligne 6. La directive `{Body}` et la directive `{End}` sont toujours en début de ligne. ObjectMacro soulève une erreur de compilation si les deux directives ne sont pas en début de ligne. Cela sépare de manière lisible la définition des paramètres d'une macro du corps de celle-ci.

1.3.3.2 L'insertion de macro

L'insertion de macro passe par l'utilisation de la commande `{Insert:}` dans une macro. Cette commande peut apparaître dans n'importe quel texte dans la définition ou dans le corps de la macro. Elle permet d'ajouter le contenu d'une macro à l'endroit où elle est utilisée. Dans le corps de la macro `presentation`, le corps de la macro `entete` est insérée à la ligne 5 dans le listing 1.31. Ce texte apparaît toujours dans toutes les instances de la macro `presentation`. Le résultat est illustré dans le listing 1.32.

1.3.3.3 L'indentation du texte

L'indentation du texte s'effectue grâce à la commande `{Indent:}`. Le texte et les références de paramètre entre les deux commandes `{Indent:}` et `{End}` sont indentés. Dans l'exemple du listing 1.33, pour chaque ligne entre les deux commandes, une tabulation suivie d'un tiret est ajoutée au début de chaque ligne comme nous pouvons le constater dans le listing 1.34.

```
1  Macro presentation
2      Param
3          sports: sport, separator="\n";
4  {Body}
5  {Insert: entete}
6  Bonjour !
7  {sports}
8  {End}
9
10 Macro entete
11 {Body}
12 Copyright 2018 UQAM
13 {End}
14
15 ...
```

Listing 1.31: Utilisation de la commande `Insert`

```
Copyright 2018 UQAM
Bonjour !
Je pratique du rugby.
Je pratique de l'escalade.
```

Listing 1.32: Résultat après exécution du listing 1.27 avec les macros du listing 1.31

1.3.3.4 Différenciation du texte et des directives ObjectMacro

Le texte dans une macro peut représenter du code comme du code Java ou du code C#. Dans ces langages de programmation, l'utilisation de l'accolade est constante. Il faut dans ObjectMacro pouvoir écrire les accolades sans causer de conflits avec les directives ObjectMacro.

Pour donner le maximum de flexibilité à l'utilisateur, ObjectMacro distingue le texte et les directives grâce à une définition explicite des jetons des commandes. Pour cela, nous avons défini une convention pour les commandes dans le corps

```
1  Macro presentation
2      Param
3          sports_pratiques: sport, separator="\n";
4  {Body}
5  Bonjour !
6  {Indent: "\t - "}
7  {sports_pratiques}
8  {End}
9  {End}
10
11 ...
```

Listing 1.33: Utilisation de la commande `{Indent:`

```
Bonjour !
    - Je pratique du rugby.
    - Je pratique de l'escalade.
```

Listing 1.34: Résultat après exécution du listing 1.27 avec les macros du listing 1.33

d'une macro. Elle correspond à une accolade suivie du nom avec la première lettre en majuscule ainsi que deux points `{Nom:`. De plus, dans la grammaire d'ObjectMacro, la commande `{Insert:` et la commande `{Indent:` sont définies chacune comme un jeton.

Dans le listing 1.35 à la ligne 5, la commande `{Ajout:` est utilisée. En revanche, cette directive n'est pas définie dans la grammaire. ObjectMacro détecte un mot réservé inconnu et soulève alors une erreur syntaxique. Un jeton générique suivant la convention des commandes est défini dans la grammaire pour reconnaître les directives inconnues utilisées dans le corps d'une macro. Cela permet d'avoir un message spécifiant que la commande est inconnue. le listing 1.36 illustre le message d'erreur après la compilation de la macro `presentation` du listing 1.35.

Dans la grammaire d'ObjectMacro, une référence de paramètre est également un

```
1  Macro presentation
2      Param
3          sports_pratiques: sport, separator="\n";
4  {Body}
5  {Ajout: entete}
6  Bonjour !
7  {sports_pratiques}
8  {End}
9
10 Macro entete
11 {Body}
12 Copyright 2018 UQAM
13 {End}
14
15 ...
```

Listing 1.35: Utilisation d'une commande inconnue

```
*** SYNTAX ERROR ***

Line : 5
Char : 1
Syntax error on unexpected unknowncommand token "{Ajout:":
expecting: '{Insert:', '{Indent:', text part, eol, text escape, variable.
```

Listing 1.36: Erreur de syntaxe dû à l'utilisation d'une commande inconnue

jeton. ObjectMacro reconnaît la référence d'un paramètre seulement lorsque le nom du paramètre est contenu entre deux accolades sans espace entre le nom et les accolades. Dans le cas contraire, ObjectMacro détecte du texte comme nous pouvons le voir dans le listing 1.37.

1.4 Les approches alternatives d'ajouts d'éléments

Dans cette section, nous allons présenter les autres approches pour ajouter des éléments dans les paramètres.

```
{Insert: macro} # Commande ObjectMacro
{Insert : macro} # Texte
{nom} # Reference d'un parametre
{ nom } # Texte
{Ajout: macro} # Erreur de syntaxe, jeton de commande inconnu
```

Listing 1.37: Différenciation du texte et des directives ObjectMacro

1.4.1 L'ajout simultané de plusieurs éléments

En couplant le programme avec une base de données ou un autre programme (cf. section 1.1), une liste préconstruite peut venir d'un autre programme. Le modèle objet permet d'ajouter une liste d'éléments en un seul appel de méthode pour éviter à l'utilisateur d'itérer sur tous les éléments.

Pour chaque paramètre défini dans une macro, une méthode d'ajout avec le préfixe `addAll` suivi du nom du paramètre concerné est définie pour permettre d'ajouter plusieurs éléments. Cela n'empêche pas au développeur d'utiliser les autres méthodes d'ajouts comme nous pouvons le voir dans le listing 1.38. Le résultat est illustré dans le listing 1.39.

1.4.2 L'approche par constructeur

ObjectMacro permet l'approche par constructeur pour créer des instances de macro. Chaque macro possède deux constructeurs ; un constructeur par défaut et un constructeur surchargé. Ce constructeur surchargé permet d'ajouter des éléments lors de l'initialisation de la macro. Le constructeur surchargé possède un nombre de paramètres équivalent au nombre de paramètre défini pour la macro. Pour les paramètres de type de chaîne de caractères, une seule chaîne peut être passée en argument. Pour les paramètres de type macro, une liste contenant des éléments du type abstrait `Macro` peut être passée en argument.

```
1  ...
2  List<MSport> sportsPratiques = new LinkedList<>();
3  MSport mSport = macros.newSport();
4  mSport.addNomSport("du rugby");
5  sportsPratiques.add(mSport);
6
7  mSport = macros.newSport();
8  mSport.addNomSport("de l'escalade");
9  sportsPratiques.add(mSport);
10 // Ajout de deux sports
11 mPresentation.addAllSportsPratiques(sportsPratiques);
12
13 mSport = macros.newSport();
14 mSport.addNomSport("du football");
15 // Ajout d'un sport additionnel
16 mPresentation.addSportsPratiques(mSport);
17 System.out.println(mPresentation.build());
```

Listing 1.38: Ajout de plusieurs éléments simultanées

```
Bonjour !
Je pratique du rugby.
Je pratique de l'escalade.
Je pratique du football.
```

Listing 1.39: Résultat après exécution du listing 1.38 avec les macros du listing 1.16

Comme nous pouvons voir dans le listing 1.40, les éléments `MSport` sont créés et ajoutés dans une liste qui est passée en argument au constructeur surchargé de `MPresentation`. Le constructeur fait un appel aux méthodes `addAll` (cf. section 1.4.1) pour ajouter tous les éléments. La valeur `null` est acceptée dans le constructeur surchargé. Elle indique qu'il n'y a pas de valeur initiale pour le paramètre concerné. Il est permis par la suite d'ajouter de nouveaux éléments avec les méthodes `add` et `addAll`.

```
1  Macros macros = new Macros();
2  List<Macro> sports = new LinkedList<>();
3
4  MSport mSport = macros.newSport("de l'escalade");
5  sports.add(mSport);
6
7  mSport = macros.newSport("du rugby");
8  sports.add(mSport);
9
10 MPresentation mPresentation = macros.newPresentation(sports);
11
12 System.out.println(mPresentation.build());
```

Listing 1.40: Création d'instances de macro avec le constructeur surchargé

1.5 Conclusion

Nous avons vu dans ce chapitre comment fonctionne ObjectMacro pour générer du texte. Plus spécifiquement, ObjectMacro génère d'abord un modèle objet à partir des macros définies par l'utilisateur. Ce modèle objet est par la suite couplé avec des programmes ou des données pour créer un générateur de texte afin d'obtenir du texte. Nous avons ensuite présenté les différentes étapes permettant de générer du texte à partir de la définition des macros, en passant par les commandes à exécuter pour générer le modèle jusqu'à la génération du texte. Après avoir compris comment fonctionnait ObjectMacro, nous avons montré que les macros sont des patrons de textes avec un nom, des paramètres et un corps. Des options peuvent être définies sur les paramètres d'une macro pour contrôler le rendu du texte du paramètre concerné. Par la suite, nous avons vu que le corps d'une macro peut contenir du texte, des références de paramètre ainsi que des directives d'ObjectMacro telles que l'insertion ou l'indentation de macro. Nous avons présenté en dernier qu'il existe d'autres approches pour simplifier l'ajout d'éléments dans une macro.

CHAPITRE II

LE MODÈLE OBJET

Dans ce chapitre, nous allons présenter le modèle objet plus en détail. Ce chapitre nous permettra de comprendre comment fonctionne le modèle objet pour générer du texte.

Le reste de ce chapitre est organisé comme suit. Dans la section 2.1, nous allons nous pencher sur la génération du modèle objet. Dans la section 2.2, nous allons expliquer comment une macro est représentée sous la forme d'une classe. Dans la section 2.3, nous verrons comment chaque brique d'une instance de macro est construite pour obtenir le texte d'une instance de macro. Dans la section 2.4, nous présenterons la classe abstraite `Macro`, parente de toutes les classes de macro. Pour terminer, dans la section 2.5, nous faisons une synthèse de tous les points abordés dans ce chapitre.

2.1 La génération du modèle objet

Dans cette section, nous allons présenter comment ObjectMacro génère le modèle objet en deux étapes en utilisant un fichier représentant les macros.

2.1.1 La séparation de la bibliothèque ObjectMacro

ObjectMacro est séparé en deux générateurs indépendants. Le premier générateur prend en entrée un fichier de macros pour générer en sortie un fichier représentant les macros que nous appelons la représentation intermédiaire. C'est le premier générateur qui s'assure que les macros décrites dans le fichier soient syntaxiquement, lexicalement et sémantiquement correctes. La représentation intermédiaire permet de faciliter la génération du modèle objet. La figure 2.1 illustre le processus de génération du modèle objet dans le logiciel ObjectMacro.

Le deuxième générateur prend en entrée la représentation intermédiaire pour produire en sortie une bibliothèque de classes permettant de générer du texte. Le but de séparer le logiciel ObjectMacro en deux parties est de faciliter le développement de générateurs de modèles objet dans le langage désiré. Comme nous pouvons voir dans la figure 2.1, plusieurs modèles objets sont obtenus en partant de la même représentation intermédiaire. Chaque générateur prend en entrée cette représentation pour produire un modèle objet différent selon le langage désiré. Présentement seul le générateur de modèles pour le langage Java est développé. Le modèle présenté tout au long de ce mémoire est le modèle objet Java.

2.1.2 La représentation intermédiaire

Pour développer un générateur de modèles respectant les conventions de chaque langage, la représentation intermédiaire est complètement indépendante de n'importe quel langage de programmation. Nous pouvons constater dans le listing 2.2 à la ligne 7, le paramètre `sports_pratiques` est séparé en deux mots. L'application des conventions de nommage n'est pas effectuée dans le générateur de la représentation intermédiaire pour laisser au générateur de modèles effectuer les traitements spécifiques au langage.

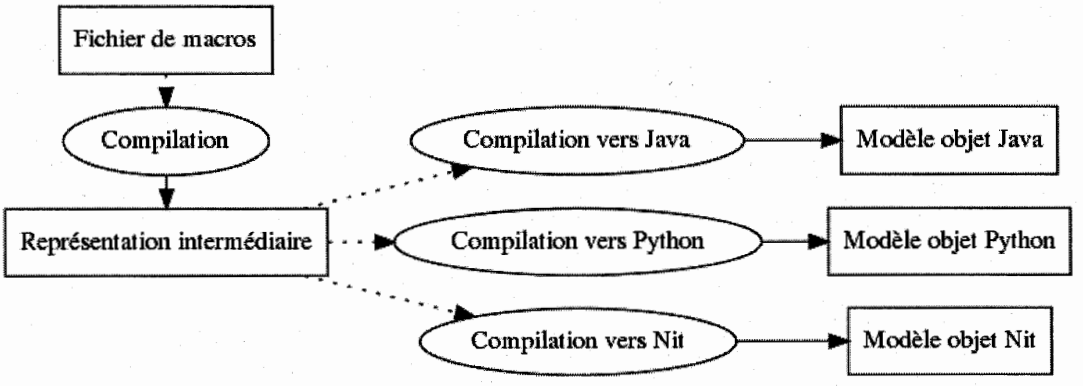

FIGURE 2.1: Processus de génération du modèle objet dans différents langages

```
1  Macro presentation
2      Param
3          metier: String;
4          sports_pratiques: sport, separator="\n";
5  {Body}
6  Bonjour !
7  Je suis {metier}.
8  {sports_pratiques}
9  {End}
10
11 Macro sport
12     Param
13         nom_sport : String;
14 {Body}
15 Je pratique {nom_sport}.
16 {End}
```

Listing 2.1: Macro `presentation` et `sport`

Pour chaque macro définie dans le listing 2.1, un élément avec pour mot clé `Macro` est généré dans la représentation intermédiaire dans le listing 2.2. Cet élément contient toutes les informations concernant la macro et représente une classe à générer. Chaque élément contenu dans la macro est décrit de manière détaillée pour éviter tout calcul supplémentaire dans le générateur de modèles.

```
Macro {
    Name = { 'presentation' }
    Param {
        Name = { 'metier' }
        Type = String  }
    Param {
        Name = { 'sports', 'pratiques' }
        Type {MacroRef{ Name = { 'sport' }}}
        Directive {Name = { 'separator' } eol}}
    MacroBody{
        'Bonjour !'
        eol
        'Je suis '
        ParamInsert {Name = { 'metier' }}
        '.'
        eol
        ParamInsert {Name = { 'sports', 'pratiques' }}}}
...
```

Listing 2.2: La représentation intermédiaire de la macro `presentation` du listing 2.1

2.2 Modélisation de la macro

Dans cette section, nous allons présenter la modélisation objet de la classe représentant une macro.

2.2.1 Les paramètres

Chaque paramètre est encodé sous la forme d'une liste chaînée pour garder l'ordre des instances ajoutées par l'utilisateur. Chaque paramètre possède une méthode privée permettant de calculer le texte du paramètre. Cette méthode commence par `build` suivie du nom du paramètre (exemple : `buildPassions`) comme nous pouvons voir dans la figure 2.2.

Les listes sont des constantes pour empêcher l'utilisateur de modifier cette réfé-

```
1  Macro presentation
2      Param
3          domaine: String;
4          passions: sport, instrument, separator="\n";
5  {Body}
6  {Insert: copyright}
7  Bonjour !
8  Je suis {domaine}.
9  {Indent: "\t - "}
10 {passions}
11 {End}
12 {End}
13
14 Macro sport
15     Param
16         nom_sport: String;
17 {Body}
18 Je pratique {nom_sport}.
19 {End}
20
21 Macro instrument
22     Param
23         nom_instrument: String;
24 {Body}
25 Je joue {nom_instrument}.
26 {End}
27
28 Macro copyright
29 {Body}
30 Copyright 2018 UQAM
31 {End}
```

Listing 2.3: Macro `presentation sport instrument` et `copyright`

rence par inadvertance, ce qui briserait les liens entre les instances de macro.

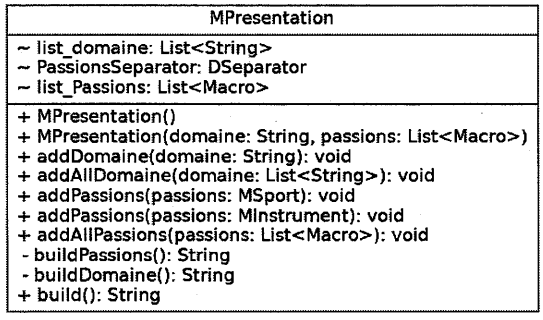

FIGURE 2.2: La classe `MPresentation` correspondant à la macro `presentation` du listing 2.3

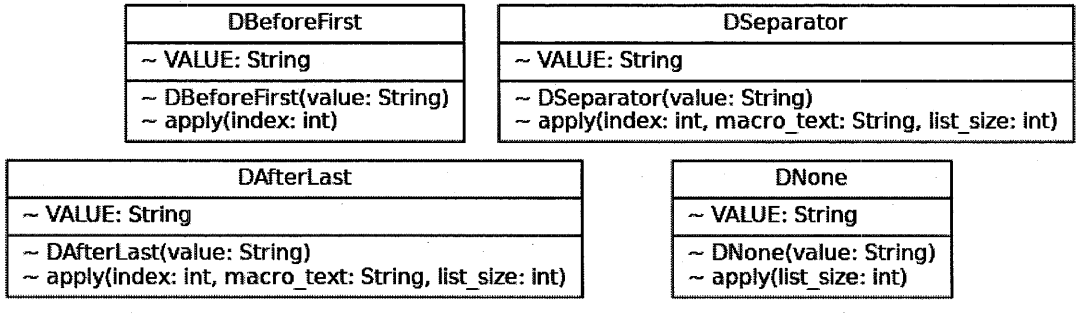

FIGURE 2.3: Les classes instanciées pour les options définies dans ObjectMacro

2.2.1.1 Les options

Pour chaque option définie dans ObjectMacro, une classe dédiée est générée comme nous pouvons voir dans la figure 2.3. Présentement, quatre classes d'options sont implémentées chacune possédant une méthode `apply`. Cette méthode permet d'appliquer le texte de l'option directement sur le texte d'une instance lors de l'appel de la méthode `build` de l'instance.

Pour chaque option définie pour un paramètre, un attribut correspondant est ajoutée à la classe. Comme nous pouvons voir dans la figure 2.2, la classe `MPresentation` possède un attribut `DSeparator` puisque dans le listing 2.3, le paramètre `passions` possède l'option `separator`.

2.2.1.2 L'initialisation des options

Les options sont initialisées paresseusement lors de la construction du texte de la macro, c'est-à-dire, pendant l'exécution de la méthode `build`. L'initialisation d'une option requiert le texte à appliquer sur les paramètres, le texte de l'option doit donc être construit pour initialiser l'attribut de l'option.

Les options peuvent faire référence à des paramètres comme la référence à `metier` dans l'option `none` du paramètre `sports_pratiques` dans le listing 2.4. Les options ne peuvent être initialisées que lors de la construction du texte de la macro dû au fait que l'ensemble des éléments des paramètres n'est connu qu'à l'appel de la méthode `build`.

Comme nous pouvons voir à la ligne 3 dans le listing 2.4, le paramètre `sports_pratiques` est défini avant le paramètre `metier`. L'option `none` possède une référence au paramètre `metier`; le texte du paramètre `metier` doit donc être construit en premier, avant d'initialiser l'option `none` du paramètre

```
1  Macro presentation
2      Param
3          sports_pratiques: sport, none="{metier} n'est pas fait pour le sport !";
4          metier: String, none="Je ne travaille pas.", before_first="Je travaille
               en tant que ";
5  {Body}
6  Bonjour !
7  {metier}.
8  {sports_pratiques}
9  {End}
10
11 Macro sport
12 ...
```

Listing 2.4: Initialisation des options de manière tardive

`sports_pratiques`. Pour construire le paramètre `metier`, les options du paramètre `metier` doivent d'abord être initialisées.

Lors de la construction du texte d'un paramètre, une vérification sur les options est effectuée pour s'assurer qu'elles soient initialisées. Dans le cas où au moins une option n'est pas initialisée, l'initialisation des options du paramètre est lancée de manière récursive. Par conséquent, lors de la construction du paramètre `sports_pratiques`, l'initialisation de ses options est lancée, ce qui cause récursivement le lancement de l'initialisation des options du paramètre `metier`. Cela permet à l'utilisateur de définir des paramètres dans l'ordre qu'il souhaite.

2.2.2 Les méthodes d'ajouts

Pour chaque paramètre défini dans une macro, deux méthodes d'ajouts sont définies. La première méthode permet d'ajouter une seule instance à la fois dans la liste. La deuxième permet d'ajouter une liste contenant plusieurs éléments. Les deux méthodes ajoutent les éléments dans les listes chaînées en s'assurant que les éléments ajoutés ne sont pas `null`. Dans chaque méthode, nous vérifions qu'un

cycle n'est pas introduit (cf. chapitre 3).

Pour chaque macro référencée par un paramètre, une définition pour la méthode d'ajout d'un élément est générée. C'est grâce à la surchage statique en Java que nous pouvons avoir plusieurs définitions d'une même méthode. Nous pouvons constater dans la figure 2.2 qu'il y a deux définitions pour la méthode d'ajout du paramètre `passions`.

La méthode `addAll` s'assure que les éléments contenus dans la liste passée en argument correspondent aux classes des macros référencées dans le paramètre.

2.2.3 La gestion de la valeur `null`

La valeur `null` n'est pas acceptée comme argument dans les méthodes d'ajout `add` et `addAll`. En particuler, pour la méthode `addAll`, la valeur `null` n'est acceptée ni comme argument, ni comme élément de la liste.

2.3 La construction du texte

Dans cette section, nous allons présenter comment le texte d'une instance de macro est construit. Nous verrons les différents choix d'implémentations notamment le cache du texte ainsi que l'importance de rendre les instances de macro immuables après l'exécution de la méthode `build` pour tirer avantage du cache.

2.3.1 La construction du texte d'une instance de macro

La construction du texte d'une instance de macro se fait grâce à la méthode `build`. La méthode `build` calcule le texte d'une macro. Cette méthode construit le texte de la macro par rapport au corps défini dans le fichier de macros en utilisant la classe `StringBuilder` de la bibliothèque standard de Java tel qu'illustré dans le listing 2.5.

```
1  public String build() {
2      ...
3      initDomaineDirectives();
4      initPassionsDirectives();
5
6      StringBuilder macro_text = new StringBuilder();
7      MCopyright m1 = this.getMacros().newCopyright();
8
9      macro_text.append(m1.build());
10     macro_text.append(LINE_SEPARATOR);
11     macro_text.append("Bonjour !");
12     macro_text.append(LINE_SEPARATOR);
13     macro_text.append("Je suis ");
14     macro_text.append(buildDomaine());
15     macro_text.append(".");
16     macro_text.append(LINE_SEPARATOR);
17     StringBuilder text_to_indent = new StringBuilder();
18     StringBuilder indent_text = new StringBuilder();
19     indent_text.append("    ");
20     indent_text.append(" - ");
21     text_to_indent.append(buildPassions());
22     //applyIndent ajoute le texte de l'indentation a chaque ligne de '
           text_to_indent'.
23     macro_text.append(applyIndent(text_to_indent.toString(), indent_text.toString
           ()));
24     ...
25     return macroText.toString();
26 }
```

Listing 2.5: La méthode `build` de la macro `presentation` du listing 2.3

2.3.1.1 La construction d'un paramètre

La référence d'un paramètre est encodée par l'appel de la méthode de construction du texte du paramètre référencé (cf. section 2.2.1) comme nous pouvons le constater dans le listing 2.5 à la ligne 21. Cette méthode parcourt tous les éléments de la liste pour calculer leur texte et les concatène pour former le texte du paramètre.

```
1  private String buildPassions() {
2      StringBuilder parameter_text = new StringBuilder();
3      List<Macro> macros = this.list_Passions;
4
5      int i = 0;
6      int nb_macros = macros.size();
7      String element_text = null;
8
9      if(this.PassionsSeparator == null) {
10         initPassionsDirectives();
11     }
12
13     for(Macro macro : macros) {
14         element_text = macro.build();
15         // apply ajoute le texte de l'option apres chaque instance sauf a la
               derniere pour l'option separator.
16         element_text = this.PassionsSeparator.apply(i, element_text, nb_macros);
17         parameter_text.append(element_text);
18         i++;
19     }
20
21     return parameter_text.toString();
22 }
```

Listing 2.6: La méthode de construction du paramètre `passions`

Selon l'option, l'application de l'option s'effectue après avoir obtenu le résultat de chaque élément (macro ou chaîne de caractères). Les options sont appliquées sur chaque élément séparément comme nous pouvons voir à la ligne 16 du listing 2.6. Elles ne sont pas appliquées pendant la construction du texte de la macro car les options ne dépendent pas de la définition de la macro mais dépendent de la liste dans laquelle l'instance est ajoutée.

2.3.1.2 La construction d'une macro insérée

L'insertion de macro se traduit par une création d'une nouvelle instance de la macro référencée directement dans la méthode `build`. Dans le listing 2.5 à la

ligne 7, une instance de la macro `copyright` est initialisée pour être ajoutée dans le texte de la macro à la ligne 9.

2.3.1.3 L'indentation du texte

Pour appliquer l'indentation, le texte à indenter doit d'abord être construit. Le texte de l'indentation est ajouté avant chaque ligne du texte à indenter. La méthode `applyIndent` effectue cette opération comme nous pouvons voir à la ligne 23 dans le listing 2.5.

2.3.2 Le cache des macros

Après l'exécution de la méthode `build`, le texte construit d'une instance est stocké dans un cache de texte. Lorsque la méthode `build` est appelée une seconde fois, le texte contenu dans le cache est retourné. Cela évite de calculer à plusieurs reprises le texte d'une instance de macro.

Le listing 2.7 montre le code de la méthode `build` avec la mise en place du cache.

```
1  public String build() {
2      CacheBuilder cache_builder = this.cacheBuilder;
3      if(cache_builder == null) {
4          cache_builder = new CacheBuilder();
5      }
6      else { return cache_builder.getExpansion(); }
7      this.cacheBuilder = cache_builder;
8      /* ... */
9      cache_builder.setExpansion(macro_text.toString());
10     return macro_text.toString();
11 }
```

Listing 2.7: La méthode `build` de la macro `presentation` du listing 2.3 concentré sur le cache du texte

	Sans cache	Avec cache
Figure 2.4	4 appels	4 appels
Figure 2.5	12 appels	8 appels
Figure 2.6	28 appels	12 appels
...	...	...

TABLE 2.1: Tableau contenant le total des appels de construction du texte avec et sans cache selon le cas de figure

2.3.2.1 Importance du cache

La figure 2.4 illustre un graphe où la méthode `build` d'une macro (`d` dans le cas présent) est appelée à plus d'une reprise. Dans le graphe, chaque nœud représente une macro et les arcs représentent l'appel de la construction du texte d'une instance de macro liée. Un parcours en profondeur à partir de l'instance `a` est effectué pour construire le texte de l'instance `a`. La méthode `build` de `a` fait appel à la méthode `build` de `b` correspondant à l'arc 1. Pour concision, nous nommons « appel 1 » cet appel relié à l'arc 1.

L'appel 1 cause l'appel 2. Au retour de ces deux appels, la méthode `build` de l'instance `c` est lancée ce qui correspond à l'appel 3 qui cause l'appel 4. Au total, il y a donc quatre appels. La présence ou non d'un cache ne change rien au nombre d'appels dans ce cas-ci. Nous indiquons le nombre d'appels dans le tableau 2.1.

Notons qu'en présence du cache, la construction du texte de l'instance `d` n'est effectuée qu'une seule fois car lors du deuxième appel le texte préconstruit et stocké dans le cache est retourné. Sans le cache, la construction du texte de l'instance `d` est effectuée à deux reprises, une pour chaque appel.

La figure 2.5 représente un graphe contenant des nœuds supplémentaires par rap-

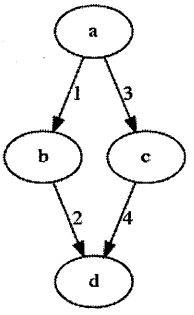

FIGURE 2.4: Graphe orientée avec quatre nœuds montrant la séquence de construction d'une macro

port à la figure 2.4. Ces nœuds sont structurés similairement au graphe précédent pour causer une construction multiple du nœud g. Le nouveau sous-graphe est attaché au nœud d.

En absence du cache, le texte de l'instance d doit être construit à deux reprises. Les appels 5, 6, 7 et 8 sont donc effectués à deux reprises. Au total, en incluant les appels 1, 2, 3 et 4, nous avons $4+2*4=12$ appels. En présence du cache, le texte de l'instance d n'est construit qu'à une seule reprise. Par conséquent, les appels 5, 6, 7 et 8 sont effectués qu'une seule fois pour un total de $4+4=8$ appels. Nous transcrivons ces nombres d'appels dans le tableau 2.1.

La figure 2.6 représente un graphe contenant des nœuds supplémentaires par rapport à la figure 2.5 pour causer une construction multiple du nœud j. Le nouveau sous-graphe est rattaché au nœud g.

En absence du cache, le texte de l'instance d doit être construit à deux reprises. Les appels 5, 6, 7 et 8 sont donc effectués à deux reprises causant quatre constructions du texte de l'instance g. Les appels 9, 10, 11 et 12 sont donc effectués à quatre reprises. Au total, nous avons $4+2*4+4*4=28$ appels. En présence du cache, le texte des instances d et g n'est construit qu'à une seule reprise pour un total

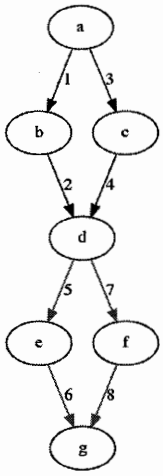

FIGURE 2.5: Graphe orientée avec sept nœuds montrant la séquence de construction d'une macro

de $4+4+4=12$ appels. Nous transcrivons ces nombres d'appels dans le tableau 2.1.

Nous pouvons réécrire le nombre d'appels de la figure 2.6 comme ceci :

— Sans cache

$$4+2*4+4*4=4(2^0+2^1+2^2)$$

— Avec cache

$$4+4+4=3*4$$

Nous pouvons considérer les sous-graphes ajoutés dans les figures 2.5 et 2.6 comme des niveaux. En considérant que la figure 2.4 n'a qu'un seul niveau, les deux autres figures en ont respectivement deux et trois. Pour un graphe similaire de n niveaux, nous pouvons calculer le nombre d'appels en utilisant la formule des suites géométriques (Clapham et Nicholson, 2009b) :

— Sans cache

$$4\sum_{i=0}^{n-1} 2^i = 4(\frac{1-2^n}{1-2}) = 4 * (2^n - 1)$$

— Avec cache

$$n * 4$$

Nous pouvons en déduire que la complexité de l'algorithme est :

— Sans cache

$$\mathcal{O}(4 * (2^n - 1)) = \mathcal{O}(2^n)$$

— Avec cache

$$\mathcal{O}(4n) = \mathcal{O}(n)$$

Sans le cache, la complexité serait exponentielle. Avec le cache, la complexité est linéaire.

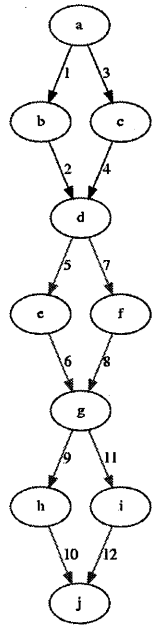

FIGURE 2.6: Graphe orientée avec dix nœuds montrant la séquence de construction d'une macro

2.3.3 Immuabilité des macros

Après la construction du texte d'une instance de macro, cette instance devient immuable. Lorsque le cache d'une instance d'une macro est rempli, l'ajout de nouveaux éléments à cette instance n'est alors pas autorisé par le modèle. Cette instance devient immuable pour empêcher que le texte de l'instance de la macro soit différent du texte contenu dans le cache.

Dans le cas où nous voudrions permettre d'ajouter des éléments après l'appel de la méthode `build`, l'invalidation du cache serait nécessaire pour avoir le texte du cache équivalent au texte de l'instance. L'invalidation du cache d'une instance de macro devrait être effectuée sur le cache de chaque instance de macro dépendante. Au vu de la complexité de la mise en oeuvre de l'invalidation du cache, nous avons fait le choix d'interdire l'ajout d'éléments après l'appel de la méthode `build`.

Le listing 2.8 est utilisé pour le listing 2.9. À la ligne 5 du listing 2.9, une exception est soulevée pour signaler qu'il est impossible d'ajouter de nouveaux éléments dans l'instance après une première construction du texte.

2.4 La classe parent `Macro`

Chaque classe de macro générée hérite d'une classe parent `Macro` pour bénéficier du polymorphisme lors de l'ajout de plusieurs éléments avec `addAll` dans une macro et lors de la construction d'un paramètre. Cette classe est une classe abstraite contenant toutes les méthodes et les attributs communs à chaque macro comme

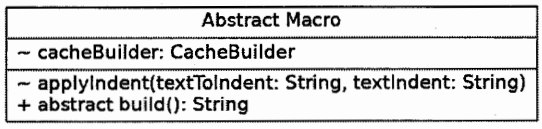

FIGURE 2.7: La classe parent `Macro`

```
1 Macro presentation
2     Param
3         metier: String;
4         passions: String, separator=", ";
5 {Begin}
6 Bonjour !
7 Je suis {metier}.
8 {passions}
9 {End}
```

Listing 2.8: Macro `presentation`

```
1 ...
2 mPresentation.addPassions("Je pratique du rugby !");
3 mPresentation.addPassions("Je joue de la guitare !");
4 mPresentation.build();
5 mPresentation.addPassions("Je fais de la boxe"); // Exception soulevée
6 ...
```

Listing 2.9: Ajout d'éléments après la construction de la macro

nous pouvons le constater dans la figure 2.7.

Les paramètres peuvent référencer plusieurs macros, l'attribut du paramètre est une liste qui ne doit contenir que des enfants de `Macro`.

Chaque enfant de `Macro` définit la méthode abstraite `build` pour permettre la construction du texte spécifique à la macro. Cela permet de construire le texte d'un paramètre sans connaître son type statique (cf. section 2.3.1.1).

C'est la classe `Macro` qui fournit le cache à l'aide de l'attribut `cacheBuilder` (voir figure 2.7). La classe `Macro` nous permet également d'appliquer le patron de conception `Visiteur` (Gamma *et al.*, 1993) qui permet de vérifier le type dynamique d'une instance de macro (cf. section 4.2.4.2).

2.5 Conclusion

Dans ce chapitre, nous avons expliqué la génération du modèle ainsi que le contenu de ce modèle objet. Le logiciel est séparé en deux générateurs connectés par un fichier représentant les macros. Nous avons présenté comment une classe représentant une macro est spécifiée dans le modèle objet généré. Nous avons montré qu'une classe de macro possède des attributs pour les paramètres et d'autres attributs pour les options. Les options sont initialisées paresseusement lors de la construction du texte de l'instance parce que les listes des paramètres ne sont pas complètes avant l'appel de la méthode `build`. Nous avons montré comment la méthode `build` est définie. Pour éviter des calculs répétitifs du texte d'une même instance de macro, nous avons fait le choix de mettre en place un cache. Par conséquent, nous avons également fait le choix de rendre les instances de macros immuables après l'appel du `build`. Pour terminer, nous avons montré que la classe parent `Macro` est utile pour bénéficier du polymorphisme.

CHAPITRE III

LA DÉTECTION DE CYCLES

Dans ce chapitre, nous allons expliquer les cycles qui peuvent être rencontrés dans les définitions de macro et dans leur usage dynamique. Nous présenterons, également, les algorithmes qui ont été mis en place pour détecter ces cycles et rapporter une erreur statique ou dynamique, selon le cas, au développeur.

Le reste de ce chapitre est construit comme suit. Dans la section 3.1, nous donnons une notation que nous utiliserons tout le long de ce chapitre. Dans la section 3.2, nous allons présenter la détection de cycles dans la définition des paramètres d'une macro. À la section 3.3, nous allons présenter la détection de cycles dans un contexte dynamique, c'est-à-dire, dans l'utilisation du modèle d'instanciation. Nous expliquerons également trois algorithmes permettant de détecter les cycles dans ce contexte. Nous présenterons, par la suite, les expérimentations et les mesures de performance effectuées sur deux de ces algorithmes pour montrer que l'algorithme naïf correpond le mieux pour répondre à notre problématique. Pour terminer, à la section 3.4, nous ferons une synthèse des points abordés dans ce chapitre.

3.1 Préambule

Une notation est utilisée tout le long de ce chapitre pour simplifier la compréhension des algorithmes. Dans un graphe dirigé, nous notons `v` le nœud parent auquel est ajouté un nœud. Le nœud enfant est noté `w`. L'arc de `v` à `w` est noté (`v`,`w`). Dans le cas d'ObjectMacro, cela s'apparente à noter la macro parent `v` et la macro enfant `w`.

Un ordre topologique d'un graphe orienté acyclique désigne un ordonnancement linéaire des nœuds tel que pour tout arc allant de `v` à `w`, le nœud `v` est ordonné avant le nœud `w`.

3.2 La détection de cycle statique

Dans cette section, nous allons présenter comment un cycle peut être rencontré dans la définition des paramètres d'une macro. Nous montrerons comment l'algorithme de Tarjan est utilisé pour détecter les cycles dans ce contexte.

3.2.1 Illustration du problème

```
1  Macro facture
2      Param
3          nom_client: String, before_first="La commande {numero_commande}
               appartient au client : ";
4
5          numero_commande: String, before_first="{numero_client}";
6
7          numero_client: String, before_first="Le client {nom_client} a pour numéro
               : ";
8  {Body}
9  ...
10 {End}
```

Listing 3.1: Référence cyclique entre les paramètres

Un paramètre peut faire référence à un autre paramètre à travers le texte des options. Comme nous pouvons constater dans le listing 3.1, le paramètre `nom_client` fait référence au paramètre `numero_commande` à travers son option `before_first`. `numero_commande` fait référence au paramètre `numero_client` qui fait lui même référence au `nom_client`. Nous remarquons qu'il y a un cycle de dépendances entre ces paramètres. Le texte du paramètre `nom_client` ne peut pas être construit sans le texte du paramètre `numero_commande`. Le texte du paramètre `numero_commande` ne peut pas être construit sans le texte du paramètre `numero_client` et le texte du paramètre `numero_client` ne peut pas être construit sans le texte de `nom_client`. Aucun des trois paramètres ne peut donc être construit. Le cycle doit être détecté avant la génération du modèle, plus précisément, avant la génération de la représentation intermédiaire (cf. section 2.1.2) et une erreur doit être soulevée.

3.2.2 Présentation Algorithme de Tarjan

L'algorithme de Tarjan (Tarjan, 1971) implémenté dans la bibliothèque SableCC est utilisée pour détecter ces cycles dans un graphe orienté statique.

Le but de cet algorithme est d'identifier les composantes fortement connexes dans un graphe. Une composante fortement connexe est un sous-graphe tel que : pour tout couple `(v, w)` de nœuds dans ce sous-graphe, il existe un chemin de `v` à `w` et inversement. Dans cet algorithme, un parcours en profondeur est effectué dans un graphe depuis chacun des nœuds. Chaque nœud parcouru pendant la recherche en profondeur est ajouté à un ensemble `P`. Pendant le parcours à chaque nœud, deux numéros sont assignés à chaque nœud dans l'ordre où ils sont explorés. Le premier numéro, correspondant à `num` dans l'algorithme 1, représente le numéro du nœud. Le deuxième numéro correspond au nœud le plus proche dans un sous arbre ou au successeur direct du nœud concerné.

Un nœud est considéré comme le représentant d'une composante fortement

Algorithm 1: Algorithme de détection des cycles de Tarjan

Entrées: G #graphe à parcourir

Sorties : *partition* #Liste des composantes fortement connexes

```
DÉBUT detectCycle
    num ← 0;
    P ← []; partition ← [];
    POUR CHAQUE v ∈ G FAIRE
        SI v.num = null ALORS
            parcours(v);
        FIN
    FIN
    Function parcours(node)
        node.num ← num; node.minNum ← num; num ← num + 1;
        P.add(node); node.inP = true;
        POUR CHAQUE w ∈ node.children FAIRE
            SI w.num = null OU w.inP ALORS
                SI w.num = null ALORS
                    parcours(w);
                FIN
                node.minNum ← min(node.minNum, w.minNum);
            FIN
        FIN
        SI node.minNum = node.num ALORS
            M ← []
            FAIRE
                w ← P.pop(); M.add(w);
            TANT QUE w ≠ node;
            partition.add(M);
        FIN
    End
FIN
```

connexe lorsque le premier numéro est égal au deuxième numéro. Tous les sommets accessibles de ce nœud sont alors ajoutés à un ensemble et forme une composante fortement connexe. L'ensemble `partition` dans l'algorithme 1 est l'ensemble des composantes fortement connexes.

3.2.3 Application du l'algorithme de Tarjan

Après avoir collecté les informations sur les macros, l'algorithme de Tarjan est exécuté. Pour déterminer la présence d'un cycle, une boucle est effectuée sur tous les paramètres. Si un paramètre n'est pas le représentant d'une composante fortement connexe, cela signifie que le paramètre est impliqué dans un cycle. Une exception est alors soulevée signalant à l'utilisateur qu'un cycle est détecté.

3.3 Détection dynamique de cycles

Dans cette section, nous allons présenter la détection de cycle dans un contexte dynamique. Nous présenterons notre choix d'utiliser l'algorithme naïf à travers un exposé des expérimentations et des mesures de performances sur deux des algorithmes présentés.

3.3.1 Illustration du problème

La détection dynamique de cycles est une détection de cycles lors de la construction du graphe orienté représentant les dépendances entre les macros. La détection de cycles s'effectue lorsque l'utilisateur ajoute des instances de macros via les méthodes d'ajouts (cf. section 2.2.2).

Dans le listing 3.2, nous observons que la macro `div` fait référence à elle-même dans la définition du paramètre `parts`. Cette référence cyclique est autorisée par ObjectMacro. Dans le listing 3.3, nous pouvons remarquer qu'une instance de `MDiv` fait référence à une autre instance de `MDiv`. Lors de la construction du texte

```
1  Macro html
2      Param
3          parts : text, div;
4  {Body}
5  <html>
6      <title> Mon petit doc HTML </title>
7      <body>
8  {Indent: "           "}
9  {parts}
10 {End}
11     </body>
12 </html>
13 {End}
14
15 Macro text
16     Param
17         content : String;
18 {Body}
19 {content}
20 {End}
21
22 Macro div
23     Param
24         parts : text, div;
25 {Body}
26 <div>{parts}</div>
27 {End}
```

Listing 3.2: Illustration du problème de cycle

de `div1`, la construction du texte de l'instance `div2` est exécutée. Étant donné que `div1` et `div2` n'ont pas la même référence, ce sont deux méthodes `build` différentes qui sont exécutées.

En revanche, par inadvertance, l'utilisateur peut introduire un cycle en ajoutant une instance dans la même instance comme dans le listing 3.4. Étant donné que `div1` et `div2` partagent la même référence, c'est la même méthode `build` qui est exécutée pour construire `div1` et `div2`. Le cycle doit être détecté et une erreur

```
1  ...
2  MDiv div1 = m.newDiv();
3  MDiv div2 = m.newDiv();
4  div1.addParts(div2);
5  ...
```

Listing 3.3: Création d'une dépendance entre deux instances d'une même classe de macro

```
1  ...
2  mHtml.addParts(m.newText("Du texte"));
3  mHtml.addParts(div1);
4  div2 = div1;
5  ...
6  div1.addParts(div2);
```

Listing 3.4: Introduction d'un cycle

doit être soulevée.

3.3.2 Détection hâtive du cycle (`Fail-Fast`)

Pour faciliter le débogage pour l'utilisateur, la détection de cycles est effectuée lorsque l'utilisateur crée une dépendance entre les macros, c'est-à-dire, lorsqu'il ajoute un élément dans une macro comme nous pouvons constater dans le listing 3.5. Cela nous permet de garder une politique de `Fail-Fast` (Shore, 2004) puisque l'erreur est détectée à la source du problème. Dans le listing 3.4, une exception est soulevée à l'appel de la méthode `addParts` à la ligne 6.

3.3.3 L'algorithme naïf

La version naïve de la détection incrémentale consiste à trouver un chemin à partir de `w` vers `v` lorsque l'arc (`v`, `w`) est ajouté. Cela nous permet de déterminer si `v` est un enfant de `w`. Nous considérons qu'il n'y a pas de cycle avant l'ajout. Une

```
Exception in thread "main" back.cycle.macro.ObjectMacroException:
*** OBJECT MACRO ERROR ***

An instance of "Div" is a cyclic reference to the same instance.

    at back.example.macro.ObjectMacroException.cyclicReference(
        ObjectMacroException.java:59)
    /* ... */
    at back.example.macro.Div.addParts(MDiv.java:40)
    at back.example.Main.main(Main.java:32)
```

Listing 3.5: Exception soulevée lors de l'ajout de la macro

recherche en profondeur est effectuée à partir de `w`. Si `v` est rencontré, cela signifie que `v` est un enfant de `w` et un cycle est donc présent.

À chaque visite d'un nœud, le nœud est ajouté dans un ensemble. Cet ensemble englobe tous les nœuds visités. La recherche en profondeur est alors limitée aux nœuds qui n'ont pas encore été traversés. Lors de la visite d'un nœud, l'algorithme s'assure que le nœud ne se trouve pas dans l'ensemble. À chaque nouvelle recherche (i.e. lors d'un nouvel appel d'un `add` ou `addAll`), l'ensemble est réinitialisé pour ne pas fausser la détection du cycle car l'introduction d'un nouveau nœud peut produire un cycle dans les nœuds qui ont déjà été visités.

3.3.3.1 Complexité quadratique de l'algorithme naïf

Le pire cas pour l'algorithme naïf est un graphe devenant de plus en plus profond à chaque appel de la méthode `add`. Un tel graphe force la traversée complète du graphe pour détecter un cycle.

La figure 3.1 illustre plusieurs graphes permettant de montrer la complexité de l'algorithme naïf. Dans chacun des graphes, chaque nœud représente une macro et chaque arc correspond à un lien entre deux macros créé après l'appel de la

Algorithm 2: Algorithme naïf

Entrées: v, w;

$macros\_parcourus$; # les macros visités

DÉBUT *detectCycle*

SI $w \in macros\_parcourus$ **ALORS**

stop;

FINSI

SI $w = v$ **ALORS**

afficher("Cycle détecté");

stop;

FINSI

$macros\_parcourus.add(w)$;

POUR CHAQUE $child \in w.children$ **FAIRE**

$detectCycle(v, child)$;

FIN

FIN

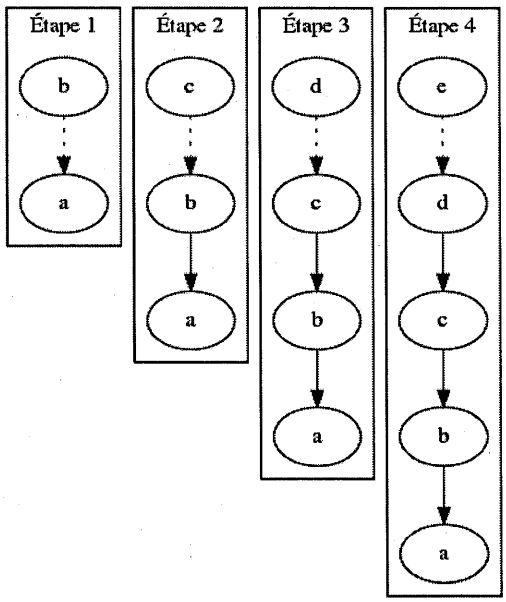

FIGURE 3.1: Pire cas pour l'algorithme naïf

méthode `add`. Les arcs en pointillés représentent les liens nouveaux. À chaque étape de la figure 3.1, une recherche en profondeur à partir de l'enfant est effectuée pour essayer de retrouver le parent. Par exemple, dans l'étape 4, une recherche en profondeur est effectuée à partir de `d` pour essayer de retrouver `e`. Comme nous pouvons le voir dans chaque étape, une traversée complète du graphe est nécessaire.

Dans l'étape 1, aucun arc est visité puisque `a` ne possède pas d'enfant. Dans l'étape 2, une seule visite de l'arc (`b`,`a`) est effectuée. Dans l'étape 3, deux arcs sont visités. Dans la dernièrer étape, trois arcs sont visités. Le nombre total d'arcs visités lors de la construction de ce graphe est de $0 + 1 + 2 + 3 = 6$ arcs visités.

Pour la construction d'un graphe similaire où `n` méthodes `add` sont appelées, nous pouvons calculer le nombre total de visites d'arc en utilisant la formule des suites arithmétiques (Clapham et Nicholson, 2009a) :

$$0+1+2+3+...+n = 0+\sum_{r=1}^{n} r = \frac{n*(n+1)}{2}$$

Nous pouvons en déduire que la complexité de l'algorithme naïf est :

$$\mathcal{O}(\frac{n*(n+1)}{2}) = \mathcal{O}(n^2)$$

La complexité est quadratique pour l'algorithme naïf.

3.3.4 L'algorithme de Haeupler, Kavitha, Mathew, Sen et Tarjan *HKMST*

L'algorithme *HKMST* (Haeupler Kavitha Mathew Sen Tarjan) (Haeupler *et al.*, 2012) permet de faire la détection incrémentale des cycles dans une complexité $\mathcal{O}(m^{\frac{3}{2}} \log n)$ en temps avec `m` et `n` correspondant respectivement au nombre d'arcs et au nombre de nœuds.

L'algorithme consiste à traverser tous les nœuds compatibles jusqu'à ce qu'un cycle soit détecté, soit jusqu'à ce qu'il n'y ait plus de nœud compatible à traverser ou qu'il y ait plus de paire de nœuds à traverser. Une paire de nœuds `v` et `w` est dit compatible si `v < w` dans l'ordre topologique. Dans la figure 3.2, le nœud `A` est compatible avec le nœud `E` puisque `A` est inférieur à `E` dans l'ordre topologique. Lorsque l'arc (`v`, `w`) est ajouté, la zone entre `v` et `w` dans l'ordre topologique est la région affectée. Cet algorithme effectue deux recherches, une recherche vers l'avant et une recherche en arrière. Deux ensembles sont définis : `F` les nœuds visités par la recherche avant, `B` les nœuds visités par la recherche arrière. Chaque nœud est défini par trois états : non-visité, *forward* (visité par la recherche avant) et *backward* (visité par la recherche arrière). Les deux recherches s'effectuent simultanément en traversant chacun un arc où les nœuds de destination sont compatibles. Le cycle est détecté lorsque la recherche avant traverse un nœud visité par la recherche

arrière et inversement.

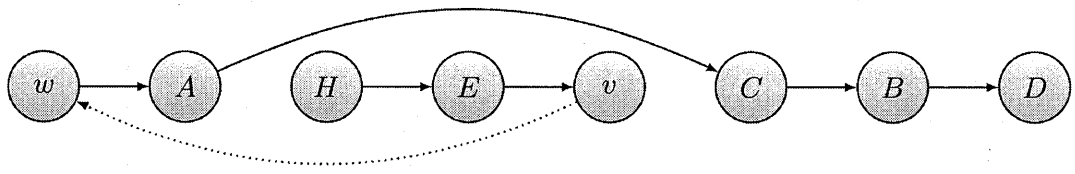

FIGURE 3.2: Exemple de l'algorithme HKMST Source : (Sigurðsson, 2016)

Comme nous pouvons remarquer dans la figure 3.2, la recherche vers l'avant traverse `w` et `A`, la recherche arrière quant à elle traverse `v` et `E`. Les recherches s'arrêtent car `H` et `C` ne sont pas compatibles puisque `C` est supérieur à `H` dans l'ordre topologique.

Pour restaurer l'ordre topologique, le nœud le plus petit noté `t` est déterminé par rapport à l'ordre topologique. Ce nœud est dans l'ensemble des nœuds non visités par la recherche avant incluant `v`. Si `t = v`, tous les nœuds de `F` sont alors ajoutés après `t` donc après `v`. Autrement, tous les nœuds inférieurs à `t` parmi `F` dans l'ordre topologique sont ajoutés avant `t`. Ensuite, tous les nœuds supérieurs à `t` parmi `B` sont ajoutés avant les nœuds de `F` ajoutés précédemment.

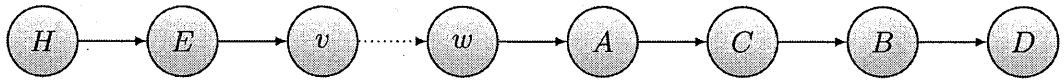

FIGURE 3.3: Exemple de l'algorithme HKMST Source : (Sigurðsson, 2016)

Comme nous pouvons remarquer dans la figure 3.3, l'ordre est restauré. Les nœuds dans `F` inférieurs à `v` sont insérés après `v` dans la figure 3.3 ; `w` et `A` sont donc ajoutés.

L'algorithme HKMST (Haeupler *et al.*, 2012), pour les graphes clairsemés, requiert une structure de données spécifique et complexe afin de répondre au problème d'ordonnancement des nœuds. Cette structure de données permet de garder

un ordre topologique des nœuds dans un temps constant amorti.

3.3.5 L'algorithme de Bender, Fineman, Gilbert et Tarjan *BFGT*

L'algorithme *BFGT* est une amélioration de l'algorithme *HKMST* en terme de temps et de simplicité d'implémentation. Cet algorithme a pour complexité en temps $O(min(m^{\frac{1}{2}}, n^{\frac{2}{3}}))$ avec `m` et `n` correspondant respectivement au nombre d'arcs et au nombre de nœuds. Nous notons par `m` et `n` le nombre d'arcs et le nombre L'algorithme *BFGT* (Bender *et al.*, 2016) présenté dans cette section est l'algorithme pour les graphes clairsemés effectuant deux traversées. Pour cela, l'algorithme maintient un ordre pseudo-topologique de nombres. Un ordre pseudo-topologique d'un graphe orienté acyclique désigne un ordonnancement des nœuds tel que pour tout arc allant de `v` à `w`, le nœud `v` est ordonné avant ou au même rang que le nœud `w`.

L'ordonnancement des nombres a pour avantage de ne pas effectuer les deux traversées si l'ordre topologique est maintenu. La répartition des nombres dans les nœuds permet de classer les nœuds par niveau.

Dans le cas où l'ordre topologique n'est pas conforme, la première traversée est effectuée. Elle consiste à faire une recherche en arrière dans les nœuds du même niveau. En somme, pour un arc (`v`,`w`), la recherche en arrière a pour but de déterminer si `w` est retrouvé à partir de `v`. S'il est retrouvé, un cycle existe alors et l'arc n'est pas ajouté dans le graphe. En revanche, si la recherche traverse un nombre défini d'arcs, la recherche s'arrête et le niveau du nœud est augmenté. Le fait d'arrêter la recherche limite le temps d'exécution. Si cette recherche est arrêtée, les nœuds traversés sont alors maintenus dans un ensemble.

Dans le cas où le niveau d'un nœud est augmenté, une recherche vers l'avant, parmi les nœuds où le niveau a été augmenté, est effectuée pour augmenter le

niveau des autres nœuds pour garder un ordre pseudo-topologique.

Lors de la recherche en avant, si un des nœuds de l'ensemble des nœuds traversés par la recherche arrière est visité, cela signifie qu'un cycle est formé. Cela revient à chercher un chemin entre les prédecesseurs de v et v pour détecter un cycle. Dans le cas contraire, des nouveaux niveaux sont assignés aux nœuds pour remettre en place l'ordre pseudo-topologique.

3.3.6 Expérimentations

Dans cette section, nous allons présenter les deux expérimentations effectuées permettant de confimer le choix de l'algorithme à utiliser. En utilisant les mesures de performance, nous avons pu déduire que l'algorithme naïf est l'algorithme qui correspond au mieux à notre problème.

Pour les expérimentations, seulement l'algorithme *BFGT* et l'algorithme naïf sont implémentés. Il existe des versions pour les graphes denses pour l'algorithme *HKMST* et *BFGT*. Il n'est pas nécessaire dans notre cas d'implémenter l'algorithme pour un graphe dense car dans ObjectMacro, il y a peu de chance qu'un tel graphe soit construit. En effet, un graphe dense signifie que le nombre d'arcs approche son nombre maximal possible dans un graphe. Dans le cas d'ObjectMacro, cela voudrait dire que toutes les instances de macro se référencent toutes entre elles.

3.3.6.1 Cas en cercle

La première expérimentation consiste à comparer les performances du pire cas de l'algorithme naïf et les performances de l'algorithme *BFGT* pour le même cas. La figure 3.4 représente un graphe similaire au pire cas présenté dans la section 3.3.3.1 auquel nous ajoutons un arc pour créer un cycle. Nous utilisons plusieurs graphes

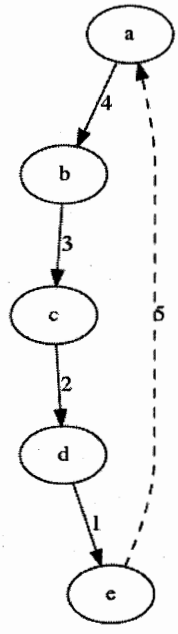

FIGURE 3.4: Pire cas pour l'algorithme naïf

similaires avec un nombre de nœuds différents pour obtenir des temps d'exécution et tracer une courbe pour chaque algorithme. Ces courbes nous permettent par la suite de comparer les performances de l'algorithme naïf et l'algorithme *BFGT*.

3.3.6.2 Cas aléatoire

Le cas aléatoire a été implémenté dans le but de tester les capacités des algorithmes. La génération du cas aléatoire consiste à itérer sur chacun des nœuds et ajouter un nœud enfant aléatoire parmi tous les nœuds créés auparavant.

La figure 3.5 représente la deuxième expérimentation où les arcs sont ajoutés de manière aléatoire. Comme nous pouvons voir dans la figure 3.5, chacun des nœuds possède un enfant pour éviter d'avoir un nœud isolé étant donné que l'algorithme *BFGT* se base sur un graphe dirigé ne contenant aucun nœud isolé. Pour avoir des mesures cohérentes, nous nous assurons que les algorithmes travaillent sur le

même graphe.

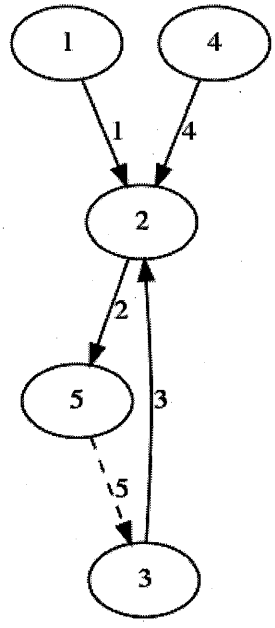

FIGURE 3.5: Cas aléatoire

3.3.7 Mesures de performance

Pour mesurer les performances, nous avons créé un générateur permettant de créer un arbre pour les deux expérimentations précédentes. Avant le lancement des tests, un échauffement de la machine virtuelle de Java est effectué pour avoir des mesures de performances significatives. L'échauffement de la machine virtuelle est un lancement préliminaire des tests permettant à son moteur d'exécution de compiler et d'optimiser le code (compilation *juste-à-temps*).

3.3.7.1 Résultats du cas en cercle

Pour le cas en cercle, nous pouvons remarquer dans la figure 3.6 que l'algorithme naïf augmente de manière quadratique au fur et à mesure que la profondeur s'agrandit. À partir de 1000 nœuds, le temps d'exécution augmente considérable-

ment. Tandis que l'algorithme de *BFGT* ne croit pas du tout au fur et à mesure que la profondeur du graphe augmente. Il est alors intéressant d'utiliser l'algorithme *BFGT* dans le cas où le profondeur excède les 800 nœuds. En revanche, pour l'utilisation des macros, l'implémentation d'une complexité en terme d'algorithme et en terme de structure de données n'est pas nécessaire. En effet, les probabilités qu'une macro avec le nombre d'enfants atteignant une profondeur supérieure à 10 sont extrêmement faibles. Comme nous pouvons le voir dans la figure 3.6, lorsque le cycle de 10 nœuds de profondeur est formé, il n'y a aucun gain réel de performance entre l'algorithme naïf et l'algorithme *BFGT*. C'est pourquoi, l'algorithme naïf est l'algorithme qui convient le mieux pour répondre à notre problématique.

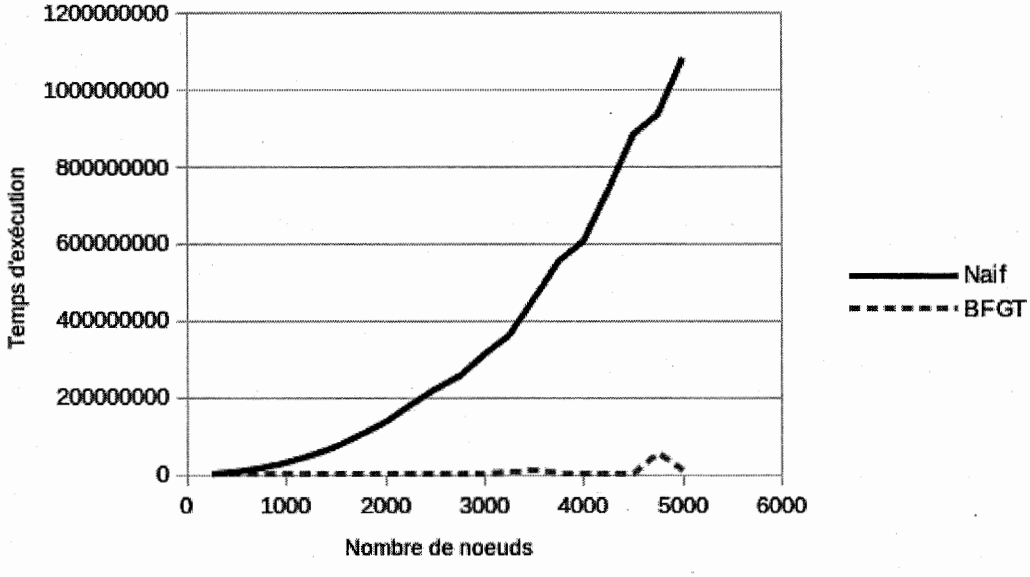

FIGURE 3.6: Graphique des expérimentations pour le cas en cercle

3.3.7.2 Résultats du cas aléatoire

Dans la figure 3.7, les deux algorithmes sont assez proches en terme de temps d'exécution. Étant donné que le moment où le cycle est formé n'est seulement connu qu'à l'exécution, il est difficile de déterminer quel ajout d'arc est à l'origine

du problème. C'est pourquoi, pour certains nombres de nœuds, le temps d'exécution est plutôt proche. Ce graphe permet de justifier que mettre en place la complexité des structures et l'implémentation de l'algorithme *BFGT* n'est pas utile pour notre cas d'étude. À 2700 nœuds, nous pouvons supposer que l'algorithme naïf a eu son pire cas. Cela semble donc indiquer que la recherche ne devient quadratique que lorsqu'un long cycle se forme aléatoirement. Dans le cas contraire, les deux algorithmes ont un temps d'exécution assez similaire.

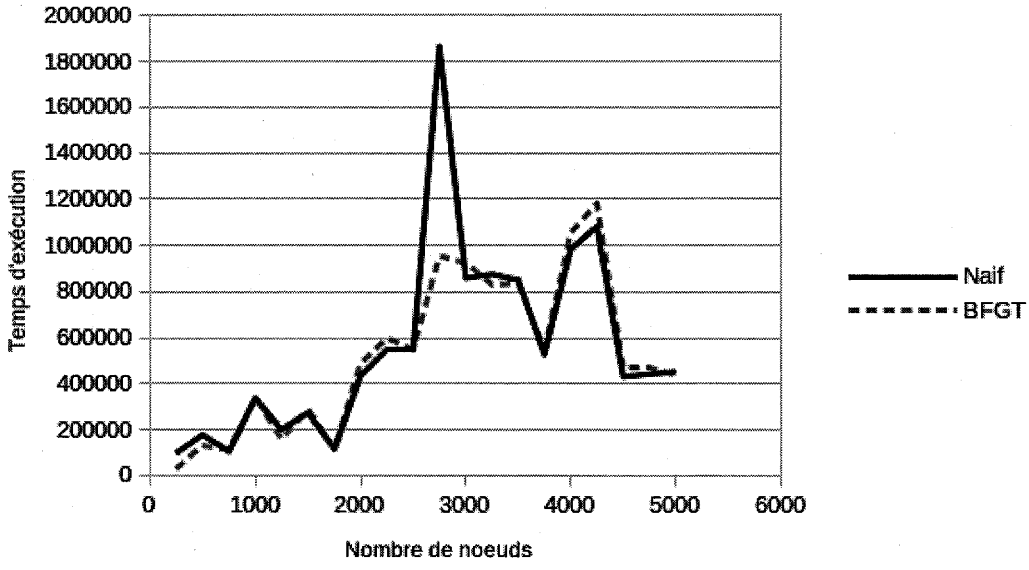

FIGURE 3.7: Graphique des expérimentations pour le cas aléatoire

3.4 Conclusion

Nous avons présenté dans ce chapitre comment détecter les cycles dans deux contextes différents. Nous avons montré qu'un cycle peut être rencontré dans la définition des paramètres. L'algorithme de Tarjan nous permet de détecter ces cycles en découpant un graphe orienté statique en composantes fortement connexes. Nous avons montré qu'un cycle peut apparaître dû à une utilisation maladroite du modèle d'instanciation. Pour répondre à cette problématique, trois

algorithmes ont été présentés mais les expérimentations et les mesures de performance sont effectuées sur deux d'entre eux. Le premier algorithme implémenté est l'algorithme naïf. Cet algorithme consiste à rechercher en profondeur parmi les enfants du nœud ajouté si le nœud parent est retrouvé à chaque ajout d'un arc. Le deuxième algorithme *BFGT* (Bender *et al.*, 2016) est un algorithme qui consiste à faire une ou deux recherches en profondeur pour détecter un cycle lors de la construction du graphe orienté. Malgré la complexité quadratique dans le pire cas de la version naïve, c'est celle-ci qui répond le mieux à notre problématique. Par ailleurs, cette version naïve ne demande que l'ajout d'un simple ensemble, comme structure de données en support à l'algorithme. Cette détection incrémentale des cycles permet, ainsi, de détecter l'origine exacte du problème en conformité avec la stratégie *Fail-Fast* (Shore, 2004).

CHAPITRE IV

LES INTERNES

Dans ce chapitre, nous allons présenter les paramètres contextuels que nous appelons les internes. Un interne est un paramètre dont la ou les valeurs proviennent d'autres macros. Ce chapitre permettra de comprendre comment les internes permettent de factoriser une macro en plusieurs macros. Pour cela, nous présenterons dans ce chapitre la définition, l'utilisation des internes dans une macro et la modélisation des internes dans le modèle objet.

Le reste de ce chapitre est organisé comme suit. Dans la section 4.1, nous verrons la définition des internes et l'utilité des internes. Par la suite, nous présenterons, dans la section 4.2, la modélisation des internes pour permettre à l'utilisateur de générer du texte sans s'occuper de la gestion des internes. Pour terminer, nous ferons une synthèse des différents points abordés dans ce chapitre dans la section 4.3.

4.1 Qu'est ce qu'un interne ?

Dans cette section, nous allons présenter les internes. Plus précisément, nous verrons la définition et l'utilité des internes dans une macro. Lorsque nous évoquons la valeur d'un paramètre ou d'un interne, nous désignons la liste d'éléments.

4.1.1 Définition d'un interne

Le but de mettre en place les internes est de réutiliser des informations d'une macro dans une autre macro sans que l'utilisateur n'ait à les gérer dans le développement de son générateur de textes. Cela permet également de séparer le corps d'une macro en plusieurs autres macros pour rendre le corps plus lisible.

Un interne est un paramètre dont la ou les valeurs proviennent d'autres macros. Une macro qui fait référence à une macro avec un interne doit passer en argument une valeur pour assigner cet interne. Nous pouvons le constater dans le listing 4.1, un argument est fourni à la référence de la macro `sport` dans la définition du paramètre `sports_pratiques` à la ligne 4 et à la référence de la macro `copyright` dans l'insertion à la ligne 6. La valeur peut correspondre à la valeur d'un paramètre ou à la valeur du texte.

Pour définir un interne, il suffit d'ajouter une section avec le mot-clé `Internal` dans la définition de la macro comme nous pouvons voir dans le listing 4.1. Comme les paramètres, ils sont utilisés dans le corps de la macro comme à la ligne 18 ou dans le texte d'une référence de macro ou d'une option. Les internes peuvent être soit de type chaîne de caractères comme le paramètre `nom_pratiquant` dans la macro `sport` soit de type macro.

4.1.2 Différences entre les paramètres et les internes

Une différence majeure des deux types de paramètre est la provenance de la valeur. À la ligne 4 du listing 4.1, le paramètre `nom` de la macro `presentation` est utilisé en tant que valeur pour l'interne `nom_pratiquant` de la macro `sport`. Tandis que la valeur d'un paramètre, dans l'exemple `nom_sport`, est un cumul d'appels des méthodes `add` et `addAll`.

```
1  Macro presentation
2      Param
3          nom: String;
4          sports_pratiques: sport(nom);
5  {Body}
6  {Insert: copyright("2018")}
7  Bonjour tout le monde !
8  Je vous presente {nom}.
9  {sports_pratiques}
10 {End}
11
12 Macro sport
13     Param
14         nom_sport: String;
15     Internal
16         nom_pratiquant: String;
17 {Body}
18 {nom_personne} pratique {nom_sport}.
19 {End}
20
21 Macro copyright
22     Internal
23         date: String;
24 ...
```

Listing 4.1: Définition d'internes

Une autre différence entre les deux types de paramètre est la définition d'options. ObjectMacro n'autorise pas la définition d'options sur les internes. Accepter la définition d'options sur les internes ajouterait une complexité concernant le choix des options à appliquer. Comme nous pouvons voir dans le listing 4.2 à la ligne 12, une option est définie pour le paramètre `nom_pratiquant`. ObjectMacro soulève une exception signalant qu'il n'est pas autorisé de définir des options à un interne. En effet, le paramètre `nom` fournit déjà une option pour l'interne `nom_pratiquant`.

Dans la définition d'un interne, si l'interne est de type macro, il est interdit de fournir des arguments à la référence de la macro. Nous pouvons le constater à

la ligne 18 du listing 4.2 qu'un argument est fourni à la référence de la macro. ObjectMacro refuse cette ligne et soulève une exception car l'argument a déjà été fourni dans le paramètre `sports_pratiques` de la macro `presentation`.

```
1  Macro presentation_centre
2      Param
3          nom: String, before_first="M. ";
4          membres: personnes(sports_communs);
5          sports_communs: sport(nom);
6  ...
7  Macro sport
8      Param
9          nom_sport: String;
10     Internal
11         nom_pratiquant: String;
12         # nom_pratiquant: String, after_last="(un sportif de haut niveau)";
13 ...
14 Macro personne
15     Param
16         prenom: String;
17     Internal
18         sports_pratiques: sport(prenom);
19 ...
```

Listing 4.2: Définition erronée d'internes

4.1.3 Référence de macro dans l'insertion d'une macro

Il est important de noter qu'il n'est pas autorisé dans ObjectMacro d'insérer des macros avec des paramètres. En effet, l'utilisateur n'a pas accès à l'instance créée dans la méthode `build`, il ne peut donc pas faire des appels aux méthodes d'ajouts `add` et `addAll` sur cette instance. Comme vous pouvez le voir dans le listing 4.3 à la ligne 7, ObjectMacro soulève une exception signalant qu'il est interdit d'insérer des macros avec un paramètre.

```
1  Macro presentation
2      Param
3          nom: String, before_first="M. ";
4  {Body}
5  Bonjour tout le monde !
6  Je vous presente {nom}.
7  {Insert: sport(nom)}
8  {End}
9
10 Macro sport
11     Param
12         nom_sport: String;
13     Internal
14         nom_pratiquant: String;
15 ...
```

Listing 4.3: Insertion erronée de macro avec paramètre

4.1.4 Réutilisation d'une même instance de macro

Une même instance de macro peut être utilisée dans plusieurs macros mais le texte de cette instance peut changer en fonction de la macro dans laquelle elle est utilisée. C'est pourquoi, une instance d'une macro est réutilisable, c'est-à-dire, l'instance peut être ajoutée dans plusieurs instances ou dans la même instance. Aucune vérification de doublon ou d'ajout de l'instance courante dans une liste est effectuée. Cela permet à l'utilisateur d'avoir des options d'affichage différentes ou des valeurs d'interne différents pour une même instance. Comme nous pouvons voir dans le listing 4.4, la macro `presentation` possède deux paramètres `sports_pratiques` et `sports_communs` qui font référence à la macro `sport`. Dans le listing 4.5, la même instance de `MSport` est ajoutée dans les deux paramètres. Le résultat est illustré dans la figure 4.6.

Le texte de l'interne `nom_pratiquant` correspond au paramètre `nom` pour toutes les instances ajoutées dans le paramètre `sports_pratiques`. Le texte de l'interne

```
1  Macro presentation
2      Param
3          nom: String;
4          sports_pratiques: sport(nom), separator="\n";
5          sports_communs: sport("Tout le monde");
6  {Body}
7  Bonjour tout le monde !
8  Je vous presente {nom}.
9  {sports_pratiques}
10 {nom}, je vais vous presenter les sports populaires par ici.
11 {sports_communs}
12 {End}
13
14 Macro sport
15     Param
16         nom_sport: String;
17     Internal
18         nom_pratiquant: String;
19 {Body}
20 {nom_pratiquant} pratique {nom_sport} !
21 {End}
```

Listing 4.4: Macro `sport` avec un paramètre contextuel `nom_pratiquant`

```
1  Macros m = new Macros();
2  MPresentation mPresentation = m.newPresentation("Pierre", null);
3  MSport mSport = m.newSport("du rugby");
4  mPresentation.addSportsPratiques(mSport);
5
6  mSport = m.newSport("de l'escalade");
7  mPresentation.addSportsPratiques(mSport);
8  mPresentation.addSportsCommuns(mSport);
9
10 System.out.println(mPresentation.build());
```

Listing 4.5: Générateur de texte basé sur les macros du listing 4.4 avec l'utilisation d'interne

`nom_pratiquant` est « `Tout le monde` » pour toutes les instances ajoutées dans le paramètre `sports_communs`. Comme nous pouvons le constater dans la figure

4.6, le texte d'un interne dépend du contexte dans lequel l'instance est ajoutée.

Le développeur du générateur de textes ne s'occupe pas de gérer l'assignation des internes comme nous pouvons voir dans le listing 4.5. C'est le modèle qui se charge de faire l'initialisation et l'assignation des internes.

```
Bonjour tout le monde !
Je vous presente Pierre.
Pierre pratique du rugby !
Pierre pratique de l'escalade !
Pierre, je vais vous presenter les sports populaires par ici.
Tout le monde pratique de l'escalade !
```

Listing 4.6: Résultat après exécution du code du listing 4.5

4.2 Modélisation des internes dans le modèle objet

Dans cette section, nous allons présenter la modélisation des internes dans le modèle objet. Pour cela, nous verrons la représentation d'un interne et tous les mécanismes liés à la génération du texte d'un interne cachés à l'utilisateur.

4.2.1 L'encodage des internes

Chaque interne est encodé sous la forme d'un dictionnaire. La clé de ces dictionnaires est un objet `Context` où la référence est utilisée pour identifier la valeur. La valeur du dictionnaire est un objet nommé `Value` contenant les différentes informations du paramètre ou du texte passé en argument de la référence de macro. Le dictionnaire nous permet d'avoir différentes valeurs pour différents contextes dans une même instance de macro la rendant réutilisable (cf. section 4.1.4).

Nous pouvons voir un exemple de représentation d'un interne dans le listing 4.7. Le dictionnaire est initialisé dans le constructeur de la macro.

```
private Map<Context, StringValue> list_NomPratiquant;
```

Listing 4.7: Représentation d'un interne dans le modèle

4.2.2 La représentation des valeurs d'un paramètre

Les valeurs contenues dans le dictionnaire sont des structures de données permettant de faciliter la construction du texte d'un interne. Pour chaque paramètre, un objet `Value` est associé. Le but de cet objet est de garder une référence sur la liste du paramètre, le contexte et les options associés. Cet objet est utilisé pour l'assignation et la construction du texte des internes.

La classe `Value` possède des attributs pour les options d'affichages et un attribut pour le contexte associés au paramètre modélisé dans la figure 4.1. Pour chaque type de paramètre, une classe enfant est définie ; `MacroValue` et `StringValue`. Chacune des classes possède une référence sur la liste du paramètre associé. Chacune de ces classes prend en argument de son constructeur la liste à laquelle elle fait référence et le contexte du paramètre comme le témoigne la figure 4.1.

La méthode `build` dans `MacroValue` et dans `StringValue` a la même fonction que la construction des paramètres (cf. section 2.2.1) à la différence que les options ne sont pas initialisées dans cette méthode. Les options de l'objet `Value` sont initialisées lors de la construction du texte de la macro. L'assignation des options de cet objet s'effectue à travers les méthodes définies dans la classe parent `Value`.

En se basant sur la macro `presentation` de la définition du listing 4.4, un objet `Context` et un objet `MacroValue` sont associés au paramètre `sports_pratiques` comme nous pouvons voir dans le listing 4.8. L'initialisation de l'objet `MacroValue` s'effectue dans le constructeur de la macro `presentation` avec en argument la liste des éléments du paramètre `sports_pratiques` ainsi que son contexte

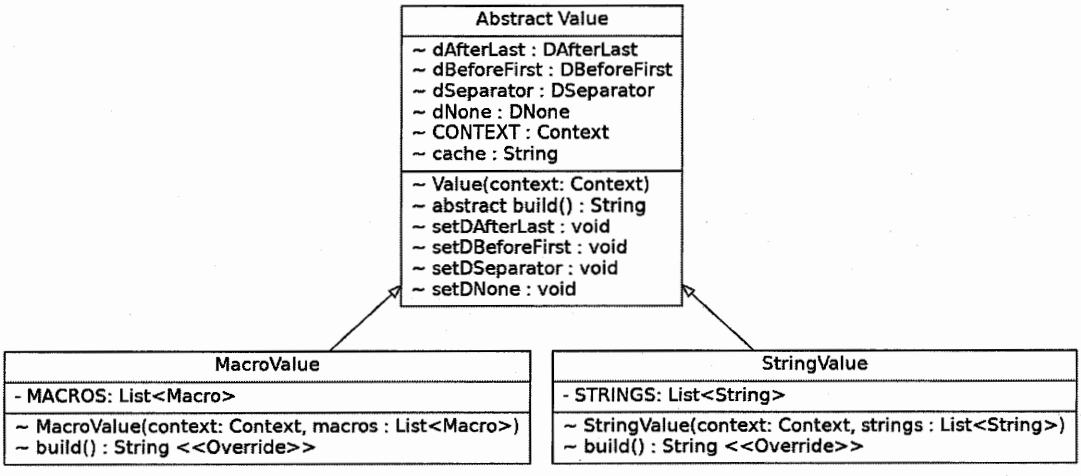

FIGURE 4.1: Diagramme de classes des valeurs d'un paramètre

`SportsPratiquesContext`. Un exemple de construction de l'objet `MacroValue` est illustré dans le listing 4.9.

```
final List<String> list_SportsPratiques;
final Context SportsPratiquesContext = new Context();
final StringValue SportsPratiquesValue;
```

Listing 4.8: Attributs associés au paramètre `sports_pratiques`

```
this.SportsPratiquesValue = new MacroValue(this.list_SportsPratiques, this.
    SportsPratiquesContext);
```

Listing 4.9: Initialisation de la valeur d'un paramètre contextuel

4.2.3 L'ajout de nouvelles valeurs à un interne

L'ajout d'une nouvelle valeur à un interne s'effectue avec la méthode, appelée `set` suivie du nom du paramètre, définie dans la classe de la macro dans laquelle est définie l'interne. Cette méthode ajoute dans le dictionnaire un objet `StringValue` ou `MacroValue`, selon le type de l'interne, et en clé le contexte du paramètre faisant référence à la macro comme nous pouvons constater dans le listing 4.10.

Le contexte peut prendre la valeur `null` dans la méthode `set` pour l'assignation des internes pour une macro insérée (cf. section 4.2.4.1).

```
1  void setNomPratiquant(
2          Context context,
3          StringValue value) {
4
5      if(value == null){
6          throw new RuntimeException("value cannot be null here");
7      }
8
9      this.list_NomPratiquant.put(context, value);
10 }
```

Listing 4.10: Ajout de nouvelles valeurs contextuelles

4.2.4 L'assignation des internes

Nous allons voir dans cette section comment sont assignés les internes lors d'une référence de macro dans une insertion de macro et dans la définition d'un paramètre pour chaque instance.

4.2.4.1 L'assignation des internes d'une insertion de macro

Dans le listing 4.11, la macro `prefixe` est insérée dans la macro `salutation` avec le paramètre `nom` comme argument de la référence. Dans la construction du texte de la macro `salutation`, une insertion de macro se traduit par l'instanciation d'un objet macro comme nous pouvons constater dans le listing 4.12 à la ligne 6. Nous considérons que l'instance ne possède qu'un seul contexte et une seule valeur peut être assignée à l'interne `nom` de la macro `prefixe`. Le contexte prend la valeur `null` pour assigner une valeur par défaut dans le dictionnaire de l'interne `nom` à la ligne 7. À la construction du texte de l'instance nouvellement créée, la valeur `null` en tant que contexte est utilisée pour obtenir la valeur de l'interne à la ligne 8.

```
1  Macro salutation
2      Param
3          nom: String;
4  {Body}
5  Bonjour {Insert: prefixe(nom)} !
6  {End}
7
8  Macro prefixe
9      Internal
10         nom: String;
11 {Body}
12 M. {nom}
13 {End}
```

Listing 4.11: Exemple de macro contenant une insertion de macro

```
1  public String build(){
2      /* ... */
3      StringBuilder macro_text = new StringBuilder();
4      macro_text.append("Bonjour ");
5
6      MPrefixe m1 = this.getMacros().newPrefixe();
7      m1.setNom(null, getNom());
8      macro_text.append(m1.build(null));
9      macro_text.append(" !");
10     cache_builder.setExpansion(macro_text.toString());
11     return macro_text.toString();
12 }
```

Listing 4.12: Ajout de nouvelles valeurs contextuelles pour une macro insérée

4.2.4.2 L'assignation des internes dans la définition d'un paramètre non contextuel

L'assignation des internes s'effectue lors de la construction du texte de la macro. Après la vérification du contenu du cache, la méthode `build` assigne les internes des macros référencées dans la définition des paramètres.

Pour assigner les internes, un parcours de la liste de chaque paramètre de type

macro est effectué. A chaque itération, une vérification du type est effectuée pour connaître le type de la macro pour assigner de manière statique les internes. Pour chaque paramètre, une méthode `init` suivie du nom de l'interne et de `Internals` est définie comme nous pouvons voir dans le listing 4.13. En se basant sur la macro `presentation` du listing 4.4, la méthode `initSportsPratiquesInternals` effectue un parcours de la liste du paramètre `sports_pratiques` pour assigner les internes de ces éléments.

```
1  private void initSportsPratiquesInternals(Context context) {
2      for(Macro macro : this.list_SportsPratiques) {
3          macro.apply(new InternalsInitializer("SportsPratiques"){
4
5              @Override
6              void setSport(MSport mSport){
7                  mSport.setNomPratiquant(SportsPratiquesContext, getNom());
8              }
9          });
10     }
11 }
```

Listing 4.13: Assignation des internes avec le patron de conception `Visiteur`

Le patron de conception `Visiteur` (ou *Walker* en anglais) (Gamma *et al.*, 1993) est utilisé pour assigner les internes. Il nous permet d'adapter le comportement de l'assignation des internes en fonction du type de l'instance.

Une méthode abstraite est définie, dans notre cas nous la nommons `apply`, dans la macro parent `Macro`. Cette méthode est redéfinie chez tous les enfants de `Macro`. La méthode `apply` permet de passer la référence de l'instance courante à la méthode d'assignation des internes.

Pour chaque enfant de `Macro`, une méthode d'assignation des internes est définie dans la classe `InternalsInitializer`. Dans notre exemple du listing 4.4, deux méthodes sont définies ; une pour la macro `presentation` et une autre pour la

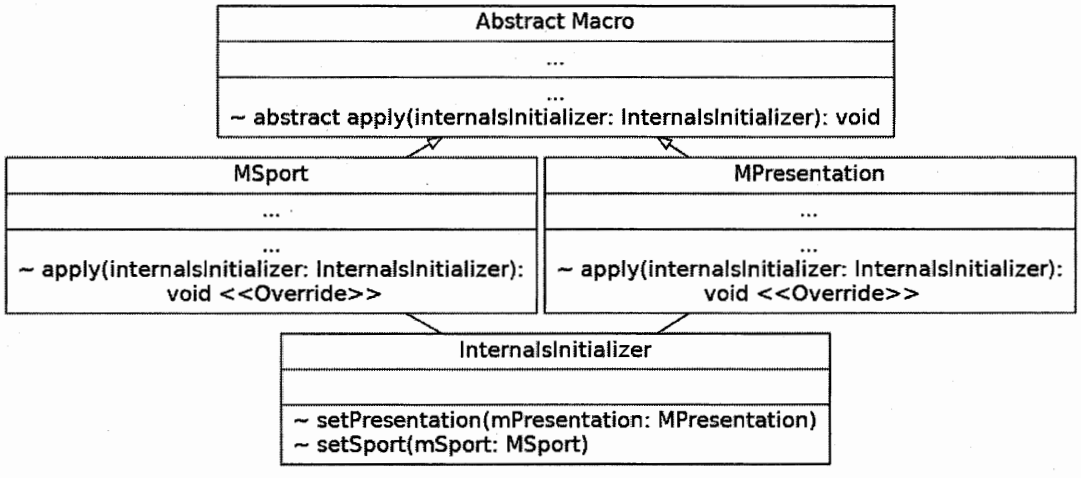

FIGURE 4.2: Modèle objet des macros 4.4 avec l'accent sur le patron `Visiteur`

macro `sport` comme nous pouvons constater dans la figure 4.2.

Dans le cas de la macro `sport`, la méthode `apply` dans `MSport` passe en paramètre l'instance courante pour faire appel à la méthode `setSport` comme nous pouvons voir dans le listing 4.14. La méthode `setSport` fait, par la suite, appel à la méthode `setNomPratiquant`.

```
1 @Override
2 void apply(
3         InternalsInitializer internalsInitializer) {
4     //setSport assigne les internes de la class MSport
5     internalsInitializer.setSport(this);
6 }
```

Listing 4.14: Application de l'initialiseur d'internes

Dans la méthode `init`, une classe anonyme héritant de `InternalsInitializer` est créée à la ligne 3 du listing 4.13. La classe anonyme redéfinit toutes les méthodes d'assignation des internes des macros référencées. La méthode `setSport` est redéfinie car le paramètre `sports_pratiques` fait référence à la macro `sport`. Lorsque le type du receveur est `MSport`, la méthode `apply` définie dans la classe

`MSport` est exécutée.

Lorsque le type du receveur n'est pas `MSport`, par exemple `MPresentation`, la méthode exécutée est celle définie dans la classe `InternalsInitializer` puisque dans le listing 4.13 la classe anonyme ne redéfinit pas `setPresentation`. Cette méthode soulève une exception indiquant qu'une instance de type `MPresentation` n'était pas attendu dans le paramètre `sports_pratiques`. Le patron de conception `Visiteur` nous permet de vérifier le type d'une instance dans une liste en plus d'assigner les internes.

4.2.5 La transformation du texte en objet `StringValue` pour de l'assignation d'interne

Pour assigner une valeur à un interne à partir du texte passé en argument, un objet de type `StringValue` doit être initialisé. Dans la redéfinition de la méthode d'assignation des internes, la chaîne de caractères est construite comme nous pouvons constater dans le listing 4.15. Cette chaîne de caractères est ajoutée dans une liste et ensuite passée dans le constructeur de `StringValue` à la ligne 9 du listing 4.15. Le contexte a pour valeur `null` car les chaînes de caractères n'ont pas besoin de contexte pour être construites.

4.2.6 La construction du texte d'une interne

Toutes les macros avec au moins un interne ne possèdent pas une méthode `build` publique. Pour construire le texte d'une macro avec un interne, il faut connaître le contexte dans lequel l'instance est utilisée. C'est pourquoi, ces macros possèdent une méthode `build` nécessitant un objet `Context` pour construire le texte comme nous pouvons voir dans le listing 4.16.

La construction du texte du paramètre `nom_pratiquant` nécessite un objet `Context` pour obtenir la valeur correspondante. Après avoir obtenu cette valeur, la

```
for(Macro macro : this.list_SportsCommuns) {
    macro.apply(new InternalsInitializer("SportsCommuns"){

        @Override
        void setSport(MSport mSport){

            StringBuilder text_argument = new StringBuilder();
            text_argument.append("Tout le monde");
            StringValue value2 = new StringValue(new ArrayList<>(Collections.
                singletonList(text_argument.toString())), null);
            mSport.setNomPratiquant(SportsPratiquesContext, value2);
        }
    });
}
```

Listing 4.15: Création d'un objet `StringValue` pour l'assignation de l'interne `nom_pratiquant`

```
String build(Context context) {
    /* ... */
    initNomSportDirectives();

    StringBuilder macro_text = new StringBuilder();

    macro_text.append(buildNomPratiquant(context));
    macro_text.append(" fait ");
    macro_text.append(buildNomSport());
    macro_text.append(" !");
    cache_builder.setExpansion(macro_text.toString());
    return macro_text.toString();
}
```

Listing 4.16: La méthode `build` avec un contexte de la macro `sport` du listing 4.4

méthode `build` est exécutée et retourne le résultat de la construction du paramètre référé dans l'objet `Value` (voir listing 4.17). Selon le contexte fourni, la construction du texte de l'interne `nom_pratiquant` peut correspondre à la construction du paramètre `nom` ou au littéral « `Tout le monde` ».

```
1 private String buildNomPratiquant(Context context) {
2
3     StringValue stringValue = this.list_NomPratiquant.get(context);
4     return stringValue.build();
5 }
```

Listing 4.17: Construction de l'interne `nom_pratiquant` de la macro `sport` du listing 4.4

4.2.7 La visibilité de la méthode `build`

La figure 4.3 illustre un diagramme de classes minimisé contenant seulement les méthodes de construction du texte de macro. Cette figure nous montre comment sont déclarées ces méthodes en fonction des paramètres définis dans les macros.

Pour construire le texte d'une macro, il suffit d'avoir une méthode `build` définie dans chaque classe d'une macro. Les classes de macro avec un interne comme la classe `MSport` ne possèdent pas une méthode `build` publique. L'utilisateur ne peut pas construire le texte de ces macros puisqu'il n'a pas accès à l'objet `Context`. Tandis que la macro `presentation` ne possède pas d'interne; aucun contexte n'est alors à fournir, la classe `MPresentation` possède alors une méthode `build` publique.

4.2.8 La redéfinition de la méthode `build` avec contexte

Lors de la construction d'un paramètre de type macro, le type statique d'une instance de macro est `Macro` (cf. section 2.3.1.1). Selon le type dynamique d'une instance de macro, il se peut que la construction du texte de l'instance nécessite un contexte. Pour garder au maximum les bénéfices du polymorphisme lors de la construction des paramètres, la classe parent `Macro` possède une méthode `build` abstraite qui est visible seulement par les classes dans le même paquetage comme

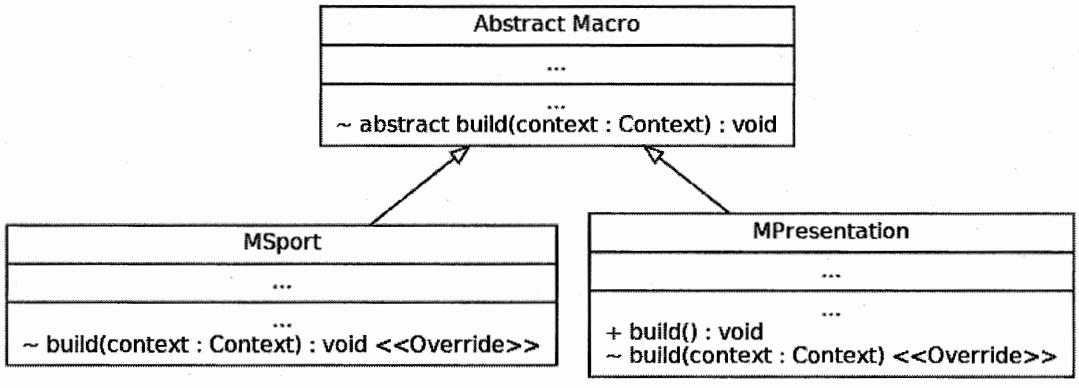

FIGURE 4.3: Diagramme de classes concentrée sur les méthodes `build` des macros du listing 4.4

nous pouvons le voir dans la figure 4.3.

Pour chacune des macros ne nécessitant pas d'objet `Contexte` à la construction comme la macro `presentation`, la méthode `build` avec un `Context` est redéfinie. Cette méthode retourne seulement le contenu d'un appel à la méthode `build` sans contexte comme nous pouvons voir dans le listing 4.18.

```
1 public String build() {
2     /* ... */
3 }
4
5 @Override
6 String build(Context context) {
7     return build();
8 }
```

Listing 4.18: Redéfinition de la méthode abstraite sans contexte

Dans le listing 4.19, `sports_pratiques` possède deux références de macros. La méthode permettant de construire ce paramètre connaît seulement le type statique pour construire le paramètre `sports_pratiques` (cf. section 2.4). À la ligne 4 du listing 4.20, le contexte est fourni en argument à la méthode `build`. En effet, la

```
1  Macro presentation
2      Param
3          nom: String;
4          sports_pratiques: sport(nom), sport_national, separator="\n";
5  ...
6  Macro sport
7      Param
8          nom_sport: String;
9          nb_joueur: String;
10     Internal
11         nom_pratiquant: String;
12 ...
13 Macro sport_national
14     Param
15         nom_sport: String;
16 ...
```

Listing 4.19: Exemple pour montrer les appels polymorphiques de la construction du texte

```
1      private String buildSportsPratiques() {
2          ...
3          for(Macro macro : macros) {
4              element_text = macro.build(this.SportsPratiquesContext);
5              ...
6          }
7          ...
8      }
```

Listing 4.20: Constructeur du texte du paramètre `sports_pratiques` de la macro `presentation` du listing 4.19

macro `sport` nécessite un objet `Contexte` pour construire son texte. En revanche, la construction du texte de la macro `sport_national` ne nécessite pas d'objet `Context` puisque la macro ne possède pas d'interne.

4.3 Conclusion

Nous avons présenté dans ce chapitre comment définir et utiliser les internes. Nous avons montré que les internes sont référencés de la même manière que les paramètres. Nous avons expliqué qu'un interne nous permet de réutiliser la valeur d'un paramètre dans d'autres macros pour factoriser le corps d'une macro. Nous avons exposé la modélisation des internes dans le modèle d'instanciation. Un interne est représenté par un dictionnaire pour permettre à une instance d'être réutilisable et d'avoir du texte différent suivant le contexte. La valeur d'un paramètre est représentée par une classe `Value` (`StringValue` ou `MacroValue` selon le type) pour être passée à une macro. Cet objet permet de garder une référence des listes du paramètre à construire et du contexte dans lequel le paramètre doit être construit. L'assignation des internes pour les paramètres contextuels se traduit par l'utilisation du patron de conception `Visiteur`. Ce patron de conception nous permet d'adapter le comportement de l'assignation d'un interne selon le contexte et de vérifier le type d'une instance. Le texte d'un interne se construit en exécutant la méthode `build` de la classe `Value`. Nous avons par la suite montré que l'utilisateur n'a pas accès à la méthode `build` pour les macros avec un interne puisque l'utilisateur n'a pas connaissance de l'objet `Context`. Nous avons également montré que les macros sans internes possèdent deux définitions de la méthode `build` ; une publique et une de visibilité paquetaque.

CHAPITRE V

LES VERSIONS

Dans ce chapitre, nous allons présenter les versions de macro. Une même macro peut avoir plusieurs versions adjacentes. Ce chapitre permettra de comprendre comment les versions de macro permettent de générer du texte différent avec un seul modèle objet. Nous présenterons la définition de versions de macro, l'utilisation des versions de macro ainsi que l'implémentation des versions dans le modèle objet.

Le reste de ce chapitre est organisé comme suit. Dans la section 5.1, nous verrons la définition des versions de macro et l'utilisation des versions de macro. Dans la section 5.2, nous présenterons la hiérarchie de classes des macros pour obtenir une sortie différente sans que l'utilisateur n'ait à modifier drastiquement son générateur de textes. Pour terminer, à la section 5.3, nous ferons une synthèse des différents points abordés dans ce chapitre.

5.1 Que sont les versions de macro ?

Nous allons présenter dans cette section ce que sont les versions de macro. Plus précisément, nous verrons l'utilité et la définition de versions d'une même macro.

5.1.1 Définition d'une macro versionnée

Le but est de générer du texte différent sans que l'utilisateur ait à adapter son générateur de textes. Pour cela, une macro peut avoir plusieurs versions. Nous appelons ces macros, dans ce chapitre, les macros versionnées.

```
1  Version fr, eng
2
3  Macro salutation
4      Version
5          fr;
6      Param
7          nom: String;
8  {Body}
9  Bonjour {nom} !
10 {End}
11
12 Macro salutation
13     Version
14         eng;
15     Param
16         nom: String;
17 {Body}
18 Hello {nom} !
19 {End}
```

Listing 5.1: Définition de différentes versions de la macro `salutation`

Des étiquettes de version sont définies grâce au mot clé `Version` comme nous pouvons voir à la ligne 1 du listing 5.1. Deux étiquettes de version sont définies ; la version `fr` et la version `eng`. Nous pouvons voir que la macro `salutation` possède deux versions ; une première version pour l'étiquette `fr` et une autre version pour l'étiquette `eng`. Une erreur est soulevée par ObjecMacro lorsque deux versions d'une macro sont définies pour une même étiquette.

Entre chaque version d'une macro, il est obligatoire d'avoir les paramètres équi-

valents tant pour le type et que pour le nom.

5.1.2 Définition d'une macro à version unique

Une macro sans mot clé `Version` est une macro à version unique qui représente toutes les versions telle qu'illustré dans le listing 5.2. Une macro avec toutes les étiquettes, telle qu'illustrée dans le listing 5.3, est une macro à version unique.

```
1  Version fr, eng
2
3  Macro salutation
4      Param
5          nom: String;
6  {Body}
7  Bonjour {nom} !
8  {End}
```

Listing 5.2: Macro à version unique sans le mot-clé `Version`

```
1  Version fr, eng
2
3  Macro salutation
4      Version
5          fr, eng;
6      Param
7          nom: String;
8  {Body}
9  Bonjour {nom} !
10 {End}
```

Listing 5.3: Macro à version unique avec le mot-clé `Version`

La version de la macro `salutation` dans le listing 5.3 est équivalente à la version de la macro `salutation` dans le listing 5.2. Par conséquent, la déclaration de ces deux versions entraîne une erreur sémantique spécifiant à l'utilisateur que la macro `salutation` est déjà définie pour les deux versions.

5.1.3 La communication entre macros

Pour qu'une macro puisse communiquer avec une autre macro, il faut obligatoirement qu'elles aient au moins une version en commun.

Dans le listing 5.4, la macro `presentation` possède deux versions identiques. C'est pourquoi, une seule version est définie pour les deux étiquettes. La macro `sport` ne peut être pas utilisée par la macro `presentation` puisque `sport` ne possède qu'une seule version pour l'étiquette `eng`. Une erreur sémantique est soulevée spécifiant à l'utilisateur que la macro `sport` est inconnue pour la version `fr` et `qc`.

```
1  Version fr, qc, eng
2
3  Macro presentation
4      Version
5          fr, qc;
6      Param
7          nom: String;
8          sports_pratiques: sport(nom), separator="\t";
9  {Body}
10 Bonjour !
11 Je vous presente {nom} !
12 {sports_pratiques}
13 {End}
14
15 Macro sport
16     Version
17         eng;
18     Param
19         nom_sport: String;
20     Internal
21         nom_pratiquant: String;
22 {Body}
23 {nom_pratiquant} practices {nom_sport} !
24 {End}
```

Listing 5.4: Communication erronée entre macros

5.1.4 Différences entre les versions d'une macro

Les internes et les options sont gérés dans le modèle sans qu'il y ait d'impact sur le générateur de texte de l'utilisateur. C'est pourquoi, d'une version de macro à une autre, les internes peuvent être différents. Par conséquent, les arguments des références de macro doivent également être différents. Les options peuvent également être différentes d'une version de macro à une autre. Bien évidemment, la différence la plus notable entre les versions d'une macro est le corps de la macro.

Dans le listing 5.5, les macros de la version `fr` faisant référence à la macro `sport` doivent avoir un argument à la référence. Tandis que pour la version `eng` de `sport`, les références à cette macro n'ont pas besoin d'argument. Par conséquent, les macros à version unique ne peuvent pas faire appel à la macro `sport` car chaque version de cette macro possède des internes différents. Dans le listing 5.5, une erreur sémantique est soulevée à la ligne 26.

5.1.5 L'utilisation des macros versionnées

Le but des macros versionnées est de générer du texte différent selon la version choisie tout en minimisant les modifications du générateur de texte. Le listing 5.8 illustre le résultat après l'exécution du générateur de texte dans la version `fr`. Le listing 5.9 montre le résultat de l'exécution du code du générateur avec la version `eng`.

La seule différence entre les deux générateurs est le choix de la version lors de l'instanciation de l'usine à objets, nommée `Macros`, comme nous pouvons voir dans le listing 5.6 et le listing 5.7. L'usine nous permet de faire le choix entre les différentes étiquettes de version déclarées dans le fichier de macros. L'énumération `Versions` contient toutes les étiquettes déclarées dans le fichier de macros. Un objet `Versions` est envoyé au constructeur de l'usine pour choisir la version

```
1  Version fr, eng
2
3  Macro sport
4      Version
5          fr;
6      Param
7          nom_sport: String;
8      Internal
9          nom_pratiquant: String;
10 {Body}
11 {nom_pratiquant} pratique {nom_sport} !
12 {End}
13
14 Macro sport
15     Version
16         eng;
17     Param
18         nom_sport: String;
19 {Body}
20 He practices {nom_sport} !
21 {End}
22
23 Macro presentation
24     Param
25         nom: String;
26         sports_pratiques: sports, separator="\n";
27 {Body}
28 Bonjour !
29 Je vous presente {nom} !
30 {sports_pratiques}
31 {End}
```

Listing 5.5: Référence erronée de macro

des macros à créer. Les instances de macro créées par une usine ne peuvent pas communiquer avec des instances créées par une autre usine.

```
1 Macros m = new Macros(Versions.FR);
2
3 MSalutation mSalutation = m.newSalutation("Pierre");
4 System.out.println(mSalutation.build());
```

Listing 5.6: Utilisation du modèle avec la version `fr` des macros définies dans le listing 5.1

```
1 Macros m = new Macros(Versions.ENG);
2
3 MSalutation mSalutation = m.newSalutation("Pierre");
4 System.out.println(mSalutation.build());
```

Listing 5.7: Utilisation du modèle avec la version `eng` des macros définies dans le listing 5.1

```
Bonjour Pierre !
```

Listing 5.8: Résultat après exécution du code du listing 5.6

```
Hello Pierre  !
```

Listing 5.9: Résultat après exécution du code du listing 5.7

```
1 Macros m = new Macros();
2
3 MSalutation mSalutation = m.newSalutation("Pierre");
4 System.out.println(mSalutation.build());
```

Listing 5.10: Utilisation du modèle avec la version par défaut

5.1.6 Le choix par défaut de la version

Lorsque l'utilisateur ne fournit pas d'objet `Versions` au constructeur de `Macros`, une version est choisie par défaut. Cette version correspond à la première étiquette déclarée à l'instruction `Version`. Dans le listing 5.1, la version choisie par défaut

est la version `fr`.

5.2 Implémentation dans le modèle objet

Dans cette section, nous allons voir comment les versions sont représentées dans le modèle objet. Pour cela, nous allons présenter la hiérarchie des classes des macros versionnées pour générer du texte sans que l'utilisateur ait la connaissance des classes représentant les versions. A noter que les classes des macros à version unique ne sont pas différentes des classes présentées dans le chapitre 2.

5.2.1 La représentation objet des macros versionnées

Le diagramme de classe de la figure 5.1 est obtenu en prenant les définitions de macros dans l'annexe A. La macro `presentation` est une macro versionnée possédant deux versions ; une pour l'étiquette de version `fr` et une autre pour l'étiquette `eng`.

ObjectMacro génère une classe abstraite représentant la macro versionnée et une classe concrète pour chaque version de la macro versionnée. La classe abstraite est la classe qui est utilisée dans le générateur par l'utilisateur. Pour que l'utilisateur n'ait pas accès aux classes concrètes, elles sont de visibilité paquetage.

Pour les macros de l'annexe A, la classe abstraite `MPresentation` est générée ainsi que deux classes concrètes héritant de `MPresentation` modélisées dans la figure 5.1.

5.2.1.1 La répartition des paramètres

Les paramètres ne doivent absolument pas être différents entre les versions de macro. C'est pourquoi, les listes des paramètres se retrouvent dans la classe abstraite `MPresentation` puisqu'elles sont communes à chaque version comme nous

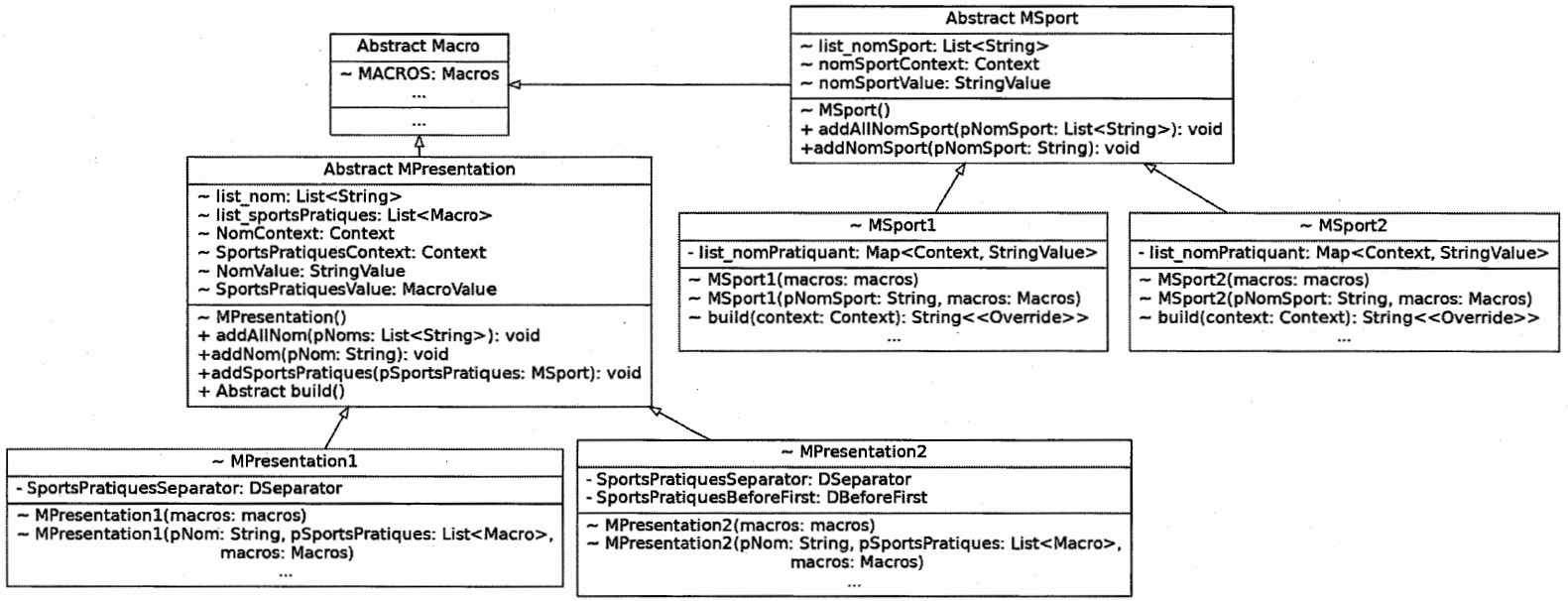

FIGURE 5.1: Hiérachie de classes des macros versionnées

pouvons voir dans la figure 5.1.

Étant donné que les internes et les options sont spécifiques à chaque version de macro (cf. section 5.1.4), les classes concrètes possèdent les dictionnaires des internes. Les classes `MSport1` et `MSport2` possèdent le dictionnaire correspondant à l'interne `nom_pratiquant`. Nous pouvons remarquer que la classe concrète `MPresentation1` possède un attribut pour l'option `separator` et la classe `MPresentation2` possède deux attributs pour les options liées au paramètre `nom`.

5.2.1.2 La répartition des méthodes

Toutes les méthodes en lien avec les internes, pour la construction du texte et l'assignation, sont dans les classes concrètes puisque les internes peuvent être différents d'une version de macro à une autre.

Les options s'appliquent lors de la construction du texte d'un paramètre. Étant donné que les options peuvent être différents d'une version à une autre (cf. section 2.3.1.1), les méthodes de construction du texte d'un paramètre sont définies dans les classes concrètes.

La classe abstraite `MPresentation` expose les méthodes dont a besoin l'utilisateur pour générer le texte, c'est-à-dire, la méthode `build` et les méthodes d'ajouts d'éléments comme nous pouvons le voir dans la figure 5.2.

La méthode `build` est abstraite dans la classe `MPresentation` pour que les classes concrètes la redéfinissent pour avoir le corps spécifique à la version. Tandis que dans la classe abstraite `MSport`, la méthode `build` n'est pas définie pour ne pas être exposée. En effet, la macro `sport` possède un interne (cf. section 4.2.7). C'est dans les classes concrètes de la macro `sport` que les méthodes `build` avec l'objet `Context` sont redéfinies telles qu'illustrées dans la figure 5.1.

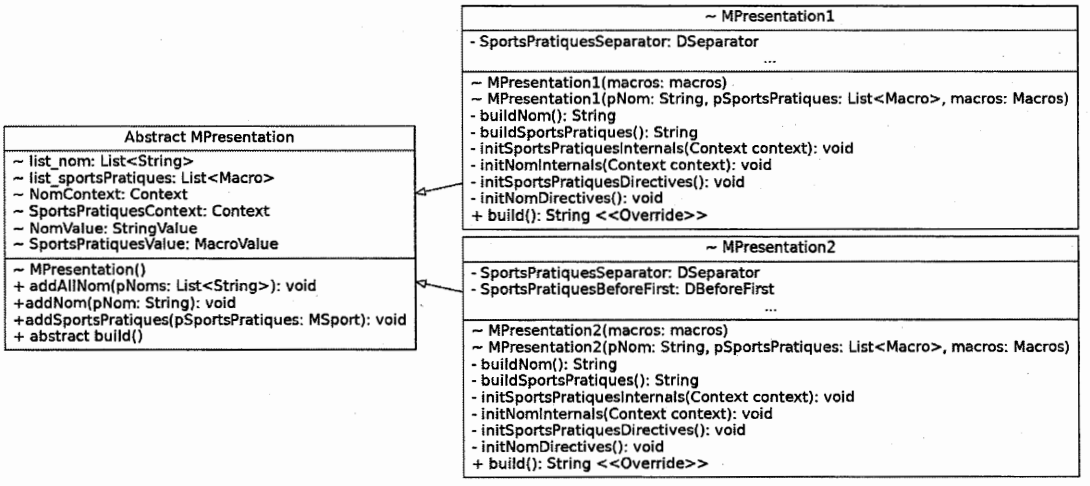

FIGURE 5.2: La répartition des méthodes dans les classes concrètes et la classe abstraite pour la macro `presentation`

5.2.1.3 Communication entre instances de macro

Chaque constructeur de chaque classe concrète prend en paramètre l'usine à objets comme nous pouvons le remarquer dans le diagramme de classes 5.1. L'instance de la macro possède une référence de l'usine à objets qui l'a créée. Lors de l'appel à la méthode `add` ou `addAll`, une comparaison des deux références est effectuée pour s'assurer que les deux pointent sur la même instance de `Macros`. Cela empêche la communication entre deux instances provenant d'usine différente.

5.2.1.4 La vérification de types

La figure 5.2 illustre la répartition des méthodes notamment les méthodes d'ajouts. Comme nous pouvons remarquer dans la figure 5.2, les méthodes d'ajouts sont définies dans la classe abstraite pour être exposées de manière publique. La méthode d'ajout `addAll` nécessite une vérification de type pour s'assurer que les instances dans la liste correspondent aux macros référencées dans le paramètre. Des méthodes de vérification du type dynamique sont définies en utilisant le patron de

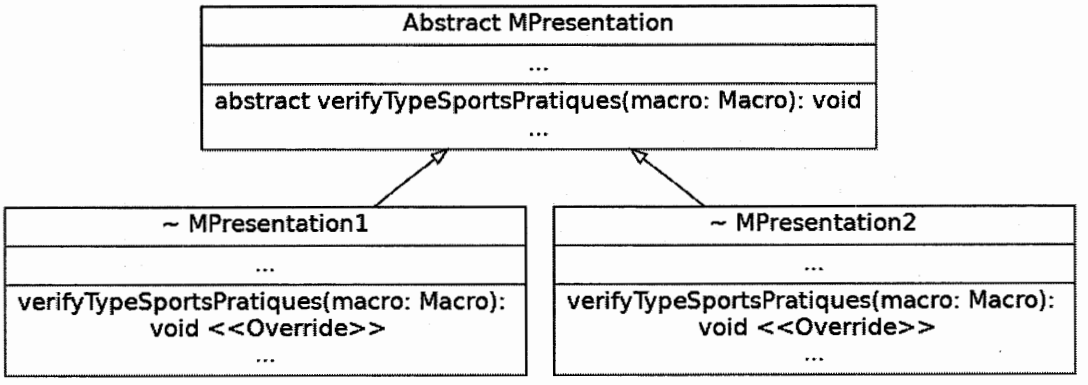

FIGURE 5.3: Diagramme de classes concentrées sur les méthodes de vérification du type d'une instance

conception `Visiteur` (Gamma *et al.*, 1993) (cf. section 4.2.4.2).

Avec les versions, la méthode `addAll` doit accepter des instances des classes concrètes de macro de la même version. `MPresentation1` pour la version `fr` n'accepte que les macros de type `MSport1` qui représente la version `fr`. Tandis que `MPresentation2` n'accepte que des instances de `MSport2`. La vérification des types est spécifique à chaque classe concrète. C'est pourquoi, une méthode abstraite `verifyTypeSportsPratiques` est définie dans `MPresentation` pour que les classes concrètes redéfinissent le comportement spécifique selon la version. La figure 5.3 modélise les méthodes de vérification du type d'une instance pour la macro `presentation` et ses versions.

5.2.2 L'usine à objets (patron de conception *Factory*)

Le patron de conception `Fabrique` (ou *Factory* en anglais) (Gamma *et al.*, 1993) est utilisé pour créer des instances de macro. La classe `Macros` représente l'usine permettant de créer des objets de la macro `presentation` et de `sport`. Le diagramme de classes de la figure 5.4 se base sur les définitions de macros dans l'annexe A. Pour chaque macro définie, deux méthodes sont définies dans l'usine

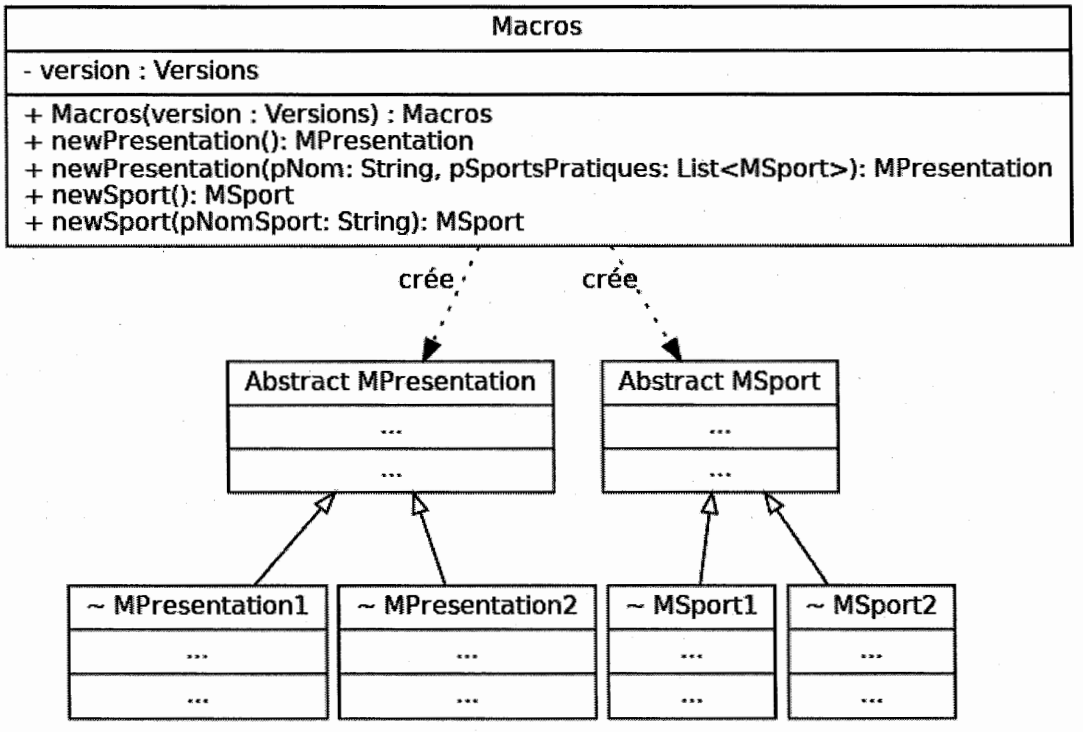

FIGURE 5.4: Patron de conception *Factory*

à objets. Une méthode ne contient pas d'argument et fait appel au constructeur par défaut de la macro illustré dans le listing 5.11 et une autre avec des arguments faisant appel au constructeur surchargé.

L'utilisateur ne connaît que le type statique (avec l'annexe A, `MPresentation` ou `MSport`) pour construire l'instance et générer le texte de cette instance. Le générateur de textes de l'utilisateur devient générique peu importe la version qu'il souhaite utiliser.

```
1  public MPresentation newPresentation(){
2      MPresentation mPresentation;
3
4      switch(this.version) {
5          case FR :
6              mPresentation = new MPresentation1(this);
7              break;
8          case ENG :
9              mPresentation = new MPresentation2(this);
10             break;
11         default :
12             throw new RuntimeException("unknown version");
13     }
14
15     return mPresentation;
16 }
```

Listing 5.11: Méthode de création d'un objet `MPresentation`

5.3 Conclusion

Dans ce chapitre, nous avons présenté comment l'utilisateur peut définir différentes versions de macros sans changer drastiquement le code de son générateur de textes. Nous avons présenté la définition des macros versionnées et des macros à version unique. Nous avons montré que le corps, les internes, les options et les arguments des références peuvent être différents entre chaque version d'une même macro.

Nous avons présenté, par la suite, la modélisation des macros versionnées. La représentation d'une macro versionnée est une classe abstraite dont les enfants représentent chacune une version de la macro versionnée. Pour terminer, nous avons présenté le patron de conception *Factory* pour créer des instances de macro sans exposer à l'utilisateur le mécanisme de création des instances des différentes versions.

CHAPITRE VI

ETAT DE L'ART

Dans ce chapitre, nous allons présenter trois logiciels reliés à la génération de textes. Ce chapitre nous permettra de voir leurs différences avec ObjectMacro ainsi que leurs avantages et inconvénients.

Le reste de ce chapitre est construit comme suit. Dans la section 6.1, nous présenterons la première version d'ObjectMacro pour comprendre les différences et les avantages qu'apportent la deuxième version d'ObjectMacro. Dans la section 6.2, nous présenterons le logiciel StringTemplate développé par Terrence Parr. Dans la section 6.3, nous présenterons le logiciel Velocity faisant parti des projets Apache. Pour terminer, dans la section 6.4, nous ferons une synthèse des différents points abordés.

6.1 ObjectMacro 1

Dans cette section, nous allons présenter la première version d'ObjectMacro (Gagnon, 2015). Pour cela, nous ferons une comparaison de la première version d'ObjectMacro et la deuxième version d'ObjectMacro pour comprendre l'intérêt d'avoir amélioré ObjectMacro.

6.1.1 Description d'ObjectMacro 1

La première version d'ObjectMacro est une première tentative pour mettre en place un générateur de textes. ObjectMacro prend en entrée un fichier de macros pour générer en sortie un modèle objet qui sera utilisé pour générer du texte. C'est à l'utilisateur de coupler le modèle avec des données pour générer du texte.

La compilation du fichier de macros pour obtenir un modèle de génération du texte est retenu dans ObjectMacro 2. La syntaxe de la définition des macros a été remaniée pour rendre le corps de la macro plus lisible. Une refonte complète du modèle objet a été effectuée dans ObjectMacro 2 pour donner plus de flexibilité à l'utilisateur.

6.1.2 La syntaxe

L'objectif principal de la refonte de la syntaxe est d'améliorer la lisibilité du corps de la macro. L'ajout des commandes `{Body}` et `{End}` apporte un gain en terme de lisibilité en séparant explicitement le corps et la définition des paramètres de la macro.

À la ligne 4 du listing 6.2, la commande `$expand:$` est une référence de la macro `sport`. Ces références correspondent dans la deuxième version aux paramètres de type macro. En enlevant toutes les références et les définitions d'options, le corps devient un peu plus lisible.

À la ligne 5 du listing 6.2, la macro `sport` est une macro interne puisqu'elle est définie à l'intérieur de la macro `presentation`. Les macros internes permettent de réutiliser les paramètres de la macro parent ; le paramètre `nom` de la macro `presentation` est réutilisée dans la macro `sport` à la ligne 6 dans le listing 6.2.

Avec les macros internes, une hiérarchie des macros est mise en place. Les macros

```
1  Macro presentation
2      Param
3          nom: String;
4          passions: sports(nom), separator="\n";
5  {Body}
6  Bonjour tout le monde !
7  Je vous presente {nom}.
8  {passions}
9  {End}
10
11 Macro sports
12     Param
13         nom_sport: String;
14     Internal
15         nom_pratiquant: String;
16 {Body}
17 {nom_pratiquant} pratique {nom_sport}.
18 {End}
```

Listing 6.1: Macro `presentation` et `sport` en ObjectMacro 2

```
1  $macro: presentation(nom) $
2  Bonjour tout le monde !
3  Je vous presente $nom.
4  $expand: sport, separator="\n"$
5  $macro: sport(nom_sport)$
6  $nom pratique $nom_sport
7  $end: sport$
8  $end: presentation$
```

Listing 6.2: Macro `presentation` et `sport` en ObjectMacro 1

internes ne peuvent être référencées que par leur parent direct. Pour enlever cette hiérarchie, toutes les macros sont déclarées au même niveau, c'est-à-dire, il n'est plus possible de créer des macros à l'intérieur d'autres macros illustrées dans le listing 6.1. La définition d'internes à une macro permet de réutiliser la valeur d'un paramètre dans une autre macro (cf. chapitre 4).

Ces modifications apportent des gains en terme de lisibilité malgré que le nombre de lignes dans le fichier de macros augmente considérablement.

6.1.3 Le modèle objet ObjectMacro 1

```
1  Macros m = new Macros();
2
3  MPresentation mPresentation = m.newPresentation();
4  mPresentation.addNom("Pierre");
5  MSport mSport = m.newSport("du rugby");
6  mPresentation.addPassions(mSport);
7  mSport = m.newSport("de l'escalade");
8  mPresentation.addPassions(mSport);
9
10 System.out.println(mPresentation.build());
```

Listing 6.3: Utilisation du modèle d'instanciation du modèle actuel

```
1  MPresentation mPresentation = new MPresentation("Pierre");
2  mPresentation.newSport("de l'escalade");
3  mPresentation.newSport("du rugby");
4
5  System.out.println(mPresentation.toString());
```

Listing 6.4: Utilisation du modèle d'instanciation de la première version

Le modèle objet a été complètement reconçu. Le but de cette reconception est de donner à l'utilisateur plus de flexibilité au niveau des créations d'instances de macro. Pour cela, la représentation des instances d'une macro est passée d'un arbre à un graphe acyclique, c'est-à-dire, une instance de macros peut avoir plusieurs parents (cf. section 4.1.4). L'instance de macro n'a pas besoin d'une instance de macro parent pour être créée comme nous pouvons le voir à la ligne 5 du listing 6.3. Cela permet d'avoir une flexibilité dans l'utilisation des instances de macro en terme de liaisons des macros. Nous pouvons constater dans le listing 6.4 à la ligne 2 que la création de l'instance `MSport` est créée à travers l'instance `MPresentation`

pour qu'elle soit rattaché à `MPresentation`.

Dans la première version d'ObjectMacro, les instances de macro sont ajoutées par le biais des méthodes `new` comme à la ligne 2 du listing 6.4. À la création de l'instance `MSport` dans le listing 6.4, l'instance est ajoutée en même temps dans l'instance `MPresentation`. Il est impossible de créer une instance de `MSport` et l'ajouter par la suite dans une instance de `MPresentation` avec le modèle du listing 6.4. Dans le modèle du listing 6.3, l'ajout et la création sont séparés. L'ajout d'éléments s'effectue grâce aux méthodes d'ajouts `add` et `addAll` permettant à l'utilisateur d'effectuer ces ajouts comme il le souhaite.

Le prix à payer pour gagner en flexibilité est la perte de sûreté statique. Dans la deuxième version d'ObjectMacro, la structure de données générée est plus flexible que la structure de données obtenue avec la première version. En effet, la représentation des instances de la première version est une arboresence permettant de garder un modèle statiquement robuste. Tandis que la représentation des instances dans la deuxième version est un graphe acyclique où des détections dynamiques telles que la détection de cycles (cf. section 3.3) sont obligatoires pour garder un modèle robuste. Le modèle objet généré avec la deuxième version d'ObjectMacro est moins robuste statiquement au bénéfice du gain en flexibilité.

6.2 StringTemplate

Dans cette section, nous allons présenter le moteur de patrons StringTemplate.

6.2.1 Philosophie de StringTemplate

StringTemplate est un moteur de patrons (ou *templates* en anglais) permettant la génération de textes, de codes source ou de pages web. Terrence Parr présente dans son article (Parr, 2004) comment renforcer la séparation entre le modèle et

la vue. Terrence Parr développe StringTemplate avec cette idée de séparation de la logique métier et la logique de l'affichage du texte. Le but de StringTemplate est de donner la possibilité de séparer le développement du générateur de textes du développement des *templates*.

StringTemplate est un logiciel qui s'imbrique dans une architecture qui ne nécessite pas de dépendance mis à part *ANTLR* (Parr et Quong, 1995) pour les analyses syntaxiques et lexicales des *templates*.

6.2.2 Le langage offert par StringTemplate

Les *templates* de StringTemplate sont analysés avec *ANTLR* (Parr et Quong, 1995). Dans les *templates*, seulement quatre opérations sont possibles :

- Les références de paramètres.
- Les références d'autres *templates*.
- Les inclusions conditionnelles de *templates*.
- Les applications de *templates* sur les attributs.

```
1  // Fichier presentationST.stg
2  presentation(personne, sports) ::= <<
3  Bonjour tout le monde !
4  Je vous présente <personne.prenom> <personne.nom>.
5  <if(length(sports))>
6  <sports:{s | <s:sport(personne.prenom)> !}; separator="\n">
7  <else>Je ne pratique aucun sport.<endif>
8  >>
9
10 sport(sport, personne) ::= "<personne> pratique <sport> !"
```

Listing 6.5: Template `presentation`

Contrairement à ObjectMacro, StringTemplate permet dans ces *templates* les instructions conditionnelles comme nous pouvons voir à la ligne 5 du listing 6.5.

StringTemplate permet d'accéder aux attributs d'un objet comme nous pouvons le voir à la ligne 4 du listing 6.5 dans le cas où les attributs existent. Autrement, un espace vide est affiché. StringTemplate permet de formater le texte du paramètre avec un *template*. À la ligne 6, pour chaque élément contenu dans `sports`, le texte du *template* `sport` est appliqué à l'élément `<s>` où `<s>` est l'élément courant.

6.2.3 La génération de textes

La génération de textes dans StringTemplate consiste à charger les *templates* dans le code du générateur de textes, ajouter des éléments dans le *template* et appeler une méthode pour construire le texte du *template*. Dans notre étude, nous examinons StringTemplate avec le langage Java. StringTemplate est utilisable dans plusieurs autres langages.

Pour obtenir un objet représentant le *template*, il faut charger le fichier qui contient le *template* ou le répertoire contenant les *templates*. Comme nous pouvons voir à la ligne 1 du listing 6.6, le fichier de groupe de *templates* `presentationST.stg` est chargé contenant le *template* `presentation`.

À partir de l'objet représentant le fichier de groupe, la méthode `getInstanceOf` initialise un objet de type `ST` représentant le *template*. À l'appel de la méthode `getInstanceOf`, le *template* demandé est analysé syntaxiquement, lexicalement et sémantiquement à la ligne 2 du listing 6.6. L'objet est utilisé pour ajouter des éléments au *template* concerné ainsi que construire le texte du *template*. Pour ajouter des valeurs à ce *template*, il faut faire appel à la méthode `add` comme à la ligne 3 du listing 6.6. Les valeurs sont ajoutées dans un dictionnaire.

Un paramètre est référencé à la ligne 4 du listing 6.5, une recherche dans le dictionnaire est effectuée pour obtenir la valeur liée au paramètre `personne`. Dans le listing 6.5 à la ligne 4, nous souhaitons accéder à un attribut du paramètre

`personne`. Cet accès est traduit dans le code Java par un appel à l'accesseur de l'attribut et un appel à `toString` sur l'attribut pour obtenir le texte.

```
1 STGroup group = new STGroupFile("templates/presentationST.stg");
2 ST presentationST = group.getInstanceOf("presentation");
3 presentationST.add("sports", sports);
4 Personne personne = new Personne("Londubas", "Pierre");
5 presentationST.add("personne", personne);
6
7 System.out.println(presentationST.render());
```

Listing 6.6: Générateur de textes

```
Bonjour tout le monde !
Je vous présente Pierre Londubas.
Pierre pratique du rugby !
Pierre pratique de l'escalade !
```

Listing 6.7: Résultat après exécution du listing 6.6

C'est avec un appel à la méthode `render` que le texte est généré. Le texte est illustré dans le listing 6.7. Les valeurs envoyées dans le dictionnaire sont évaluées pareusseusement en utilisant la méthode `toString` de chaque paramètre lors de l'appel de la méthode `render`.

6.2.4 L'héritage avec StringTemplate

StringTemplate propose l'héritage des groupes de *templates*. L'héritage permet de redéfinir un ou plusieurs *templates* dans un groupe de *templates*. Nous pouvons voir dans le listing 6.8, trois *templates* sont définis dans le fichier de groupe `Java1_4.stg`. Dans le fichier de groupe `Java1_5.stg`, le fichier `Java1_4.stg` est importé pour que les *templates* soient réutilisés ou redéfinis comme nous pouvons le voir dans le listing 6.9.

Dans l'exemple du listing 6.9, le *template* `constants` est redéfini. Le listing

6.10 génère deux classes avec les deux groupes de *templates* `Java1_4.stg` et `Java1_5.stg`. Avec la méthode `getCode`, nous pouvons remarquer que le code est commun pour les deux groupes. Le résultat de l'exécution du listing 6.10 est illustré dans le listing 6.11.

```
1 class(name, members) ::= <<
2 class <name> {
3     <members>
4 }
5 >>
6 constants(typename, names) ::= "<names:{n | <constant(n,i)>}; separator={<\n>}>"
7 constant(n) ::= "public static final int <typename>_<n>=<i>;"
```

Listing 6.8: Fichier de groupe représentant une classe en Java version 4 nommée `Java1_4.stg`

```
1 import "Java1_4.stg"
2
3 /** Override constants de Java1_4.stg */
4 constants(typename, names) ::= <<
5 public enum <typename> { <names; separator=", "> }
6 >>
```

Listing 6.9: Fichier de groupe représentant une classe en Java version 5 nommée `Java1_5.stg`

6.2.5 Comparaison avec ObjectMacro

Dans cette section, nous ferons un comparatif avec ObjecMacro.

6.2.5.1 La syntaxe

Le langage offert par StringTemplate permet les structures de contrôle comme les conditions et les boucles. Par ailleurs, le corps d'un *template* contient des références, des options ainsi que des utilisations de *templates* sur des paramètres rendant alors le corps d'un *template* moins lisible.

```
1  public void main(String[] args) {
2      STGroup java1_4 = new STGroupFile("Java1_4.stg");
3      STGroup java1_5 = new STGroupFile("Java1_5.stg");
4      System.out.println( getCode(java1_4) );
5      System.out.println( getCode(java1_5) );
6  }
7
8  public String getCode(STGroup java) {
9      ST cl = java.getInstanceOf("class"); // création de la classe
10     cl.add("name", "MaClasse");
11     ST consts = java.getInstanceOf("constants"); // chargement du template '
           constants'
12     consts.add("typename", "MyEnum");
13     consts.add("names", new String[] {"A","B"});
14     cl.add("members", consts); // ajout des constantes en tant que membre de la
           classe
15     return cl.render();
16 }
```

Listing 6.10: Génération des classes Java du listing 6.8 et du listing 6.9

```
// Enumeration avec Java 4
class MaClasse {
    public static final MyEnum_A=1;
    public static final MyEnum_B=2;
}
// Enumeration avec Java 5
class MaClasse {
    public enum MyEnum { A, B }
}
```

Listing 6.11: Résultat après exécution du listing 6.10

Contrairement à StringTemplate, ObjectMacro propose un langage déclaratif permettant de focaliser le contenu des macros sur l'affichage. C'est pourquoi, le corps d'une macro contient peu de directives et de références. Le corps est dénudé d'options et de références vers des macros pour rendre le corps le plus lisible possible.

6.2.5.2 Le modèle de chargement des *templates*

L'approche utilisée par StringTemplate pour générer du texte est complètement différente de l'approche dans ObjectMacro. Les *templates* sont analysés dynamiquement pendant l'exécution du programme de l'utilisateur dans StringTemplate. La modification d'un *template* n'a aucun impact sur le code du générateur. Dans un contexte dynamique comme le *Web*, c'est intéressant pour un graphiste de modifier un *template* sans impacter le code.

Le logiciel possède une dépendance avec *ANTLR* puisqu'il l'utilise à chaque chargement d'un *template*. C'est à l'utilisateur de s'occuper du mécanisme de transformation des *templates* en une représentation objet.

Tandis qu'avec ObjectMacro, le fichier de macros est précompilé pour avoir une modèle objet représentant les macros. L'utilisateur se sert des classes de macro pour générer du texte. Seulement lorsqu'une macro d'ObjectMacro est modifiée, il est nécessaire de recompiler le fichier de macros au complet pour que le modèle reflète les macros décrites.

6.2.5.3 La sûreté statique

À l'ajout d'une valeur dans un *template* dans StringTemplate, aucune vérification statique n'est effectuée. Il suffit que la valeur soit d'un type descendant d'`Object` pour que la valeur ait une implémentation de la méthode `toString`. ObjectMacro, quant à lui, vérifie statiquement les valeurs envoyées aux méthodes `add` pour chaque paramètre puisque la précompilation du fichier de macros permet de connaître le type des paramètres.

Aucune détection de cycle statique ou dynamique n'est effectuée dans StringTemplate. En contraste, ObjectMacro effectue la détection des cycles (cf. chapitre 3).

6.2.5.4 Langages cibles

Étant donné que StringTemplate a une dépendance avec *ANTLR* pour les analyses des *templates*, les langages pouvant être utilisés avec StringTemplate sont restreints aux langages supportés par *ANTLR*. Avec ObjectMacro, il est possible de construire le modèle objet actuel dans un autre langage pour l'utiliser dans cet autre langage. Étant donné que le modèle objet est complètement indépendant, il est possible de créer des macros reflétant le modèle dans le langage ciblé et utiliser le modèle résultant pour créer le générateur de modèles associé.

6.2.5.5 L'héritage des *templates*

Il est possible de comparer les versions dans ObjectMacro (cf. chapitre 5) avec l'héritage des fichiers de groupe de *templates* de StringTemplate. Entre les versions de macro dans ObjectMacro, il n'y a aucun lien tandis qu'il y a des liens de parentés entre les fichiers de groupe de *templates*. Lorsqu'un fichier de groupe dans StringTemplate importe un autre fichier de groupe, le fichier hérite de tous les *templates* définis dans le fichier importé. Tandis qu'avec ObjectMacro, chaque version d'une macro partage les mêmes paramètres et sont au même niveau hiérarchique. Une version ne possède pas de parent et elle est une définition pour une étiquette de version.

6.3 Velocity

Dans cette section, nous allons présenter le moteur de *templates* Velocity (Apache, 2018).

6.3.1 Description de Velocity

Velocity est un moteur de *templates* permettant la génération de textes, de codes sources ou de pages web. Le langage offert par Velocity, défini avec JavaCC, est un langage contenant des références, du texte, des directives ainsi que des structures de contrôle. Les références proviennent d'un objet `VelocityContext` défini dans le code Java permettant de faire le lien entre les données et le *template*.

De plus, Velocity permet de lier les *templates* entre eux en passant par des directives comme l'inclusion d'un *template*.

Velocity renforce la séparation de la logique métier et de la logique consacrée à la vue. Pour cela, Velocity sépare le code Java du texte des *templates*. En effet, toutes les données sont initialisées dans le code Java tandis que les formats et la structure des *templates* sont définis avec le langage proposé par Velocity.

6.3.2 Le langage de Velocity

```
1 Bonjour tout le monde !
2 Je vous présente $prenom $nom.
3 #parse("templates/sports.vm")
```

Listing 6.12: Template `presentation`

```
1 #if($sports.size() == 0)
2 $prenom pratique aucun sport...
3 #else
4 #foreach($sport in $sports)
5 $prenom pratique $sport !
6 #end
7 #end
```

Listing 6.13: *Template* `sports` contenant des directives de contrôle du texte

Les références dans le langage de Velocity sont caractérisées par le symbole $

comme nous pouvons voir dans le code 6.12. Les directives quant à elles sont caractérisées par le symbole #. Le texte de chaque *template* n'est pas interprété par Velocity. En revanche, toutes les directives et les références sont interprétées pour pouvoir ajouter des éléments aux références du *template*.

Dans le *template* `sports`, nous pouvons voir les directives de contrôle du contenu utilisées. La première est le contrôle conditionnel qui, dans ce cas, affiche le texte « `$prenom pratique aucun sport...` » si la référence `sports` ne contient aucun élément. Dans le cas contraire, une boucle est utilisée pour afficher tous les éléments de la référence `sports`.

6.3.3 La génération de textes

Pour générer du texte avec Velocity, différentes étapes sont nécessaires. Dans un premier temps, il faut initialiser le moteur de Velocity comme nous pouvons voir à la ligne 2 du listing 6.14. Par la suite pour travailler avec les *templates*, il faut lier un fichier de *template* avec un objet `Template`. Lors de la liaison du fichier à l'objet à la ligne 4 du listing 6.14, le *template* est analysé syntaxiquement et lexicalement pour obtenir un arbre syntaxique qui sera visité par la suite pour générer le texte.

Pour lier les données au *template*, il faut créer un objet de contexte appelé `VelocityContext` comme nous pouvons voir à la ligne 10 du listing 6.14. Cet objet contient une table de hachage contenant toutes les références utilisées dans les *templates*. L'arbre syntaxique est alors visité lorsque la méthode `merge` est appelée pour écrire le texte dans l'objet `Writer` en se basant sur le contexte fourni en argument (cf. ligne 16 du listing 6.14). Le listing 6.15 illustre le résultat après l'exécution du listing 6.14.

```
1  VelocityEngine velocityEngine = new VelocityEngine();
2  velocityEngine.init();
3
4  Template presentation = velocityEngine.getTemplate("templates/presentation.vm");
5
6  List<String> sports = new LinkedList<String>();
7  sports.add("du rugby");
8  sports.add("de l'escalade");
9
10 VelocityContext context = new VelocityContext();
11 context.put("prenom", "Pierre");
12 context.put("nom", "Londubas");
13 context.put("sports", sports);
14
15 StringWriter writer = new StringWriter();
16 presentation.merge(context, writer);
17
18 System.out.println(writer.toString());
```

Listing 6.14: Génération du texte avec Velocity

```
Bonjour tout le monde !
Je vous présente Pierre Londubas.
Pierre pratique du rugby !
Pierre pratique de l'escalade !
```

Listing 6.15: Résultat après exécution du listing 6.14

6.3.4 Comparaison avec ObjectMacro

Dans cette section, nous ferons une comparaison de Velocity par rapport à ObjectMacro.

6.3.4.1 La syntaxe

Similairement à StringTemplate, le langage offert par Velocity permet d'utiliser des structures de contrôle dans le *template*. Les structures de contrôle rendent le discernement entre le texte du *template* et les instructions plus complexe. Ob-

jectMacro, quant à lui, ne possède aucune structure de contrôle pour focaliser le contenu des macros sur l'affichage. Cela permet de garder le corps d'une macro le plus épuré possible.

6.3.4.2 Le modèle de génération du texte

Les *templates* de Velocity sont analysés lors de l'exécution du programme de l'utilisateur. Ce qui peut être avantageux dans un contexte dynamique où le code du générateur n'a pas besoin d'être modifié notamment dans un environnement *Web*. Contrairement à ObjectMacro, Velocity force l'utilisateur à développer le mécanisme de transformation des *templates* en une représentation objet pour ajouter des valeurs et construire le texte des *templates*.

Comme StringTemplate, Velocity possède une dépendance avec un logiciel permettant d'analyser les *templates*. À chaque exécution du code du générateur, les *templates* sont analysés avec JavaCC et l'arbre syntaxique généré par JavaCC est visité lors de l'évaluation du texte d'un template, c'est-à-dire, à l'appel de la méthode `merge`.

Dans l'exécution d'un programme, il est possible avec Velocity d'avoir plusieurs contextes pour un *template*. C'est à l'utilisateur de connaître le contexte à attacher avec les *templates* qu'il souhaite utiliser. Tandis que dans ObjectMacro, chaque instance de macro correspond à un contexte avec ses valeurs ajoutées grâce aux méthodes `add` et `addAll`. Dans le cas des internes (cf. chapitre 4), les contextes sont gérés directement dans le modèle objet pour éviter toute confusion et gagner en simplicité d'utilisation du modèle.

6.3.4.3 Le typage statique

Aucune vérification statique n'est effectuée lors de l'ajout de valeur dans un objet `VelocityContext`. Le texte d'une référence est construit par l'appel de la méthode `toString` sur l'objet correspondant. C'est pourquoi, il est obligatoire que les objets soient d'un type descendant d'`Object` dans la table de hachage contenue dans l'objet `VelocityContext`. ObjectMacro vérifie statiquement les éléments ajoutés grâce aux méthodes `add` puisque la précompilation des macros permet de connaître le type et le nom des paramètres.

Une seule classe est utilisée pour les instances d'un *template* puisque les *templates* sont chargés dynamiquement rendant impossible la mise en place de type statique des *templates*.

Par ailleurs, aucune détection de cycle statique ou dynamique n'est effectuée dans Velocity. En contraste, ObjectMacro effectue la détection des cycles (cf. chapitre 3).

6.4 Conclusion

Dans ce chapitre, nous avons présenté les différents travaux reliés à la génération de textes. Nous avons choisi de présenter la première version d'ObjectMacro pour montrer le gain en terme de lisibilité du corps des macros ainsi que le gain en flexibilité dans l'utilisation du modèle généré. Nous avons ensuite présenté StringTemplate et Velocity pour montrer leurs différences par rapport à ObjectMacro au niveau de la compilation des *templates* et du typage utilisé par les deux logiciels.

CONCLUSION

Dans ce mémoire, nous avons présenté une refonte en profondeur du générateur de générateurs de textes appelé ObjectMacro[1]. À partir d'un fichier de macros, ObjectMacro crée un modèle robuste et flexible avec lequel le développeur peut générer du texte comme bon lui semble. ObjectMacro offre un langage déclaratif et simple pour la définition des macros. Le corps de la macro est complètement dénudé de toutes références de macros, de définitions ou de structures de contrôle pour garder les macros lisibles.

Contrairement aux autres moteurs de *templates*, le modèle de génération effectue des vérifications statiques à travers les méthodes `add` pour éviter des erreurs à la génération du texte et pour éviter de générer du texte vide que l'utilisateur ne désire pas. De plus, il effectue les vérifications dynamiques telles que la détection incrémentale des cycles ou la vérification des types dynamiques qui sont en conformité avec la politique de *Fail-Fast* (Shore, 2004) pour retrouver facilement l'origine exacte du problème. Cela facilite le débogage du programme d'un utilisateur.

Le modèle objet généré permet à l'utilisateur de créer des instances de macro et de les lier dans l'ordre qu'il souhaite. Tout ce qui concerne l'affichage du texte est directement encodé dans la classe représentant la macro par exemple le corps de la macro est encodé dans la méthode `build`. Toute la complexité liée à la

1. Pour le reste de cette conclusion, ObjectMacro fait référence à la version présentée dans ce mémoire.

construction du texte d'une instance est effectuée par le modèle pour faciliter la génération de textes du point de vue de l'utilisateur.

ObjectMacro détecte deux types de cycles ; les cycles statiques dans la définition des paramètres d'une macro et les cycles dynamiques dans le modèle objet. L'algorithme de Tarjan (Tarjan, 1971) est utilisé pour détecter les cycles dans la définition des paramètres en temps linéaire. Pour détecter les cycles dynamiques, nous avons choisi de détecter ces cycles incrémentalement, c'est-à-dire, lors de l'ajout d'une instance de macro pour rester en conformité avec la politique de *Fail-Fast*. Nous avons expérimenté et comparé l'algorithme naïf et l'algorihtme *BFGT* pour en conclure que l'algorithme naïf est celui qui est plus léger en terme d'implémentation et en terme de structure de données requise. Par rapport à l'usage anticipée d'ObjectMacro, nous avons choisi de ne pas implémenter l'algorithme *BFGT* à cause de sa complexité en terme d'implémentation et de structure de données. En effet, une réelle différence de performance entre les deux algorithmes n'est remarquée que lorsque la profondeur de dépendance d'une macro atteint 200 macros.

Le modèle objet généré par ObjectMacro permet à une instance de macro d'avoir du texte variable selon le contexte dans lequel l'instance est utilisée. Pour permettre cela, nous avons choisi d'introduire la notion d'internes qui peuvent être définis dans les macros. Ces internes contiennent les informations d'une autre macro. L'initialisation et l'assignation des internes sont effectuées par le modèle pour simplifier l'utilisation du modèle. Le patron de conception `Visiteur` est implémenté dans le modèle objet pour assigner les internes d'une instance de macro en fonction du type de l'instance.

Un des objectifs recherchés dans ObjectMacro était de permettre de générer du texte différent en utilisant un seul modèle objet. Pour cela, ObjectMacro permet de

définir plusieurs versions pour une même macro. Le patron de conception *Factory* est implémenté pour exposer seulement les classes abstraites des macros possédant plusieurs versions. Cela permet à l'utilisateur d'utiliser les mêmes classes dans son générateur de textes peu importe la version du texte qu'il choisit de générer.

Nous avons comparé ObjectMacro avec d'autres outils de génération de textes. En particulier, nous avons fait une comparaison avec la première version d'ObjectMacro pour montrer les différences au niveau de la syntaxe et au niveau du modèle objet. La syntaxe de la deuxième version d'ObjectMacro apporte de la lisibilité dans le corps d'une macro. Le modèle de la deuxième version donne d'avantage de flexibilité à l'utilisateur tout en restant robuste. Les approches utilisées dans les moteurs de *templates* tels que StringTemplate et Velocity diffèrent des approches utilisées dans ObjectMacro. En effet, les analyses d'un *template* s'effectuent lors de l'exécution du programme de l'utilisateur. Par conséquent, le modèle de génération de ces moteurs de *templates* n'effectue pas de vérification statique rendant ces modèles moins robustes que le modèle d'ObjectMacro.

Travaux futurs

Nous avons choisi de séparer la génération de modèles objet en deux parties pour faciliter le développement de générateurs de modèles objets pour d'autres langages de programmation. Il serait souhaitable de développer des générateurs de modèles pour d'autres langages de programmation en adaptant les modèles aux particularités de ces langages.

Actuellement, quatre options sont définies pour les paramètres de macros. Pour ajouter plus de possibilités en terme d'affichage des paramètres, des options supplémentaires pourraient être ajoutées. Il serait également intéressant de permettre à l'utilisateur de créer ses propres options. Pour permettre cela, l'utilisateur pourrait définir de nouvelles options et en fournir l'implémentation à l'aide d'un sys-

tème de *plug-in* qui serait ajouté dans ObjectMacro.

ANNEXE A

DÉFINITIONS COMPLÈTES DES MACROS VERSIONNÉES

A.1 Définitions de la macro `presentation` en `fr` et en `eng`

```
1  Version fr, eng
2
3  Macro presentation
4      Version
5          fr;
6      Param
7          nom: String;
8          sports_pratiques: sport(nom), separator="\n";
9  {Body}
10 Bonjour tout le monde !
11 Je vous presente {nom}.
12 {sports_pratiques}
13 {End}
14
15 Macro presentation
16     Version
17         eng;
18     Param
19         nom: String;
20         sports_pratiques: sport(nom), separator="\n", before_first="English
               presentation\n";
21 {Body}
22 Hello everyone !
23 This is {nom}.
24 {sports_pratiques}
25 {End}
```

A.2 Définitions de la macro sport en fr et en eng

```
1  Version fr, eng
2
3  Macro sport
4      Version
5          fr;
6      Param
7          nom_sport: String;
8      Internal
9          nom_pratiquant: String;
10 {Body}
11 {nom_pratiquant} fait {nom_sport} !
12 {End}
13
14 Macro sport
15     Version
16         eng;
17     Param
18         nom_sport: String;
19     Internal
20         nom_pratiquant: String;
21 {Body}
22 {nom_pratiquant} practices {nom_sport} !
23 {End}
```

BIBLIOGRAPHIE

Apache (2018). Velocity. Dernière version : Velocity Tools 3.0. Récupéré de `http://velocity.apache.org/`

Bender, M. A., Fineman, J. T., Gilbert, S. et Tarjan, R. E. (2016). A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms (TALG)*, *12*(2), (p. 1–14).

Clapham, C. et Nicholson, J. (2009a). arithmetic series. Récupéré de `http://www.oxfordreference.com/view/10.1093/acref/9780199235940.001.0001/acref-9780199235940-e-170`

Clapham, C. et Nicholson, J. (2009b). geometric series. Récupéré de `http://www.oxfordreference.com/view/10.1093/acref/9780199235940.001.0001/acref-9780199235940-e-1239`

Gagnon, E. M. (1998). *SableCC, an Object-Oriented Compiler Framework.* (Mémoire de maîtrise). Université McGill, Montréal, Canada.

Gagnon, E. M. (2015). Objectmacro 1. Récupéré de `http://sablecc.org/objectmacro`

Gagnon, E. M. [s.d.]. Sablecc. Récupéré de `http://sablecc.org/`

Gamma, E., Helm, R., Johnson, R. et Vlissides, J. (1993). Design patterns : Abstraction and reuse of object-oriented design. Dans *European Conference on Object-Oriented Programming*, (p. 406–431). Springer.

Haeupler, B., Kavitha, T., Mathew, R., Sen, S. et Tarjan, R. E. (2012). Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms (TALG)*, *8*(1), (p. 1–31).

Palsberg, J. et Jay, C. B. (1998). The essence of the visitor pattern. Dans *compsac*, (p. 1–9). IEEE.

Parr, T. [s.d.]. Stringtemplate. Récupéré de `http://www.stringtemplate.org/`

Parr, T. J. (2004). Enforcing strict model-view separation in template engines. Dans *Proceedings of the 13th international conference on World Wide Web*, (p. 224–233). ACM.

Parr, T. J. et Quong, R. W. (1995). Antlr : A predicated-ll (k) parser generator. *Software : Practice and Experience*, *25*(7), (p. 789–810).

Shore, J. (2004). Fail fast [software debugging]. *IEEE Software*, *21*(5), (p. 21–25).

Sigurðsson, R. L. (2016). *Practical performance of incremental topological sorting and cycle detection algorithms.* (Mémoire de maîtrise). Chalmers University of Technology University of Gothenburg, Gothenburg, Sweden.

Tarjan, R. (1971). Depth-first search and linear graph algorithms. Dans *Switching and Automata Theory, 1971., 12th Annual Symposium on*, (p. 114–121). IEEE.