# Documentation de conception du projet GL

Baptiste Le Duc Mathéo Dupiat Malo Nicolas Ryan El Aroud
Théo Giovinazzi

23 janvier 2025

## Table des matières



## Introduction

En ingénierie logicielle, les besoins des utilisateurs évoluent rapidement et la concurrence impose des délais de livraison serrés, il est alors essentiel de concevoir des logiciels à la fois évolutifs et maintenables. Ces qualités permettent de répondre efficacement aux changements fonctionnels et technologiques tout en minimisant les coûts de développement et de maintenance. Une architecture bien pensée garantit alors une certaine modularité qui facilite l'ajout de nouvelles fonctionnalités, le debug et l'optimisation des performances.

Pour atteindre ces objectifs, il est essentiel de s'appuyer sur de solides pratiques de conception, telles que les principes SOLID. Bien qu'ils ne soient pas tous détaillés ici, ces principes définissent les bases d'un code propre, modulaire et maintenable. Par exemple, en veillant à ce qu'une classe ou un module ait une responsabilité unique (SRP) ou en rendant le code extensible sans avoir à le modifier directement (OCP), il est possible de limiter les dépendances, de réduire les régressions et de mieux anticiper les évolutions.
Dès lors, nous pouvons nous demander, *comment concevoir, dans un temps très restreint, une architecture logicielle bien pensée pour un compilateur du langage Deca ?*

Ce document mettra alors en évidence notre réflexion sur les choix de conception et les pratiques à privilégier pour cela.

# Choix de conception

Le patron interprète fournit dans le squelette initial nous a semblé être la solution parfaite pour implémenter les différentes fonctionnalités de notre compilateur. En effet, celui-ci nous a permis d'implémenter l'étape de vérification contextuelle (B) et l'étape (C) au sein même des noeuds de l'arbre à travers des méthodes abstraites `verify...()` pour la vérification contextuelle et `codeGenInst()` pour la génération de code.

Cependant, bien que le squelette de code fournissait tout le nécessaire pour implémenter l'étape B sans nécessiter de nouveaux objets, il en allait autrement pour l'étape de génération de code, qui demandait des adaptations plus importantes.

## Génération de Code

### `DVAl`

Pour la génération de code, il a été nécessaire de concevoir un mécanisme permettant de localiser précisément l'opérande produit par la génération de code d'un nœud (dans un registre, un immédiat, la pile, etc.). Pour cela, nous avons adopté l'utilisation des objets de type `DVal`, qui, conformément à la spécification, permettent de représenter de manière abstraite les registres, les immédiats, les labels et les adresses.

À la fin de chaque méthode `codeGenInst(Decacompiler compiler)` on met à jour la localisation de l'objet à l'aide de `setDVal`. Ce `DVal` pouvant être dans le cas d'un int ou un float, un immédiat. Cependant, pour garantir une manipulation uniforme des données dans des registres lors des différents appels à `codeGenInst()` nous avons dû définir une méthode abstraite nommée `codeGenToGPRegister()` pour tous les types de `DVal`

Cette méthode, implémentée de manière spécifique pour chaque sous-classe de `DVal`, permet de s'assurer que, quelle que soit la nature initiale de l'opérande (immédiat, adresse, etc.), elle sera toujours placée dans un registre général après l'appel à codeGenToGPRegister(). Ce mécanisme repose sur les principes de la programmation orientée objet et garantit une uniformité dans le traitement des opérandes, facilitant ainsi l'écriture et la maintenance du code généré.

Possibilité d'amélioration : Avec plus de temps, nous aurions pu définir des méthodes similaires à `codeGenBool()` (utilisée pour la génération de code des opérations booléennes), telles que `codeGenRegister()`, `codeGenAddr()` ou `codeGenDVal()`. Cela aurait permis au nœud parent de choisir directement où placer le résultat (registre, adresse, etc.) en appelant la méthode appropriée.

## Gestion des labels

La machine abstraite `IMA` impose que les labels aient des noms uniques pour permettre des branchements corrects et éviter toute ambiguïté. Pour répondre à cette exigence, nous avons mis en place un compteur statique, utilisé pour générer des noms de labels uniques en y ajoutant un suffixe incrémental.

Pour organiser les labels prédéfinis, tels que les labels d'erreur intégrés en dur dès le démarrage du compilateur, nous avons conçu une classe d'énumération appelée LabelManager. Cette classe centralise la gestion des labels et offre également la possibilité de créer dynamiquement des labels uniques pendant la génération de code des constructeurs de classes. Pour garantir une architecture cohérente et unifiée, toutes les classes utilitaires, y compris LabelManager, interagissent avec le compilateur, qui joue le rôle d'intermédiaire centralisé.

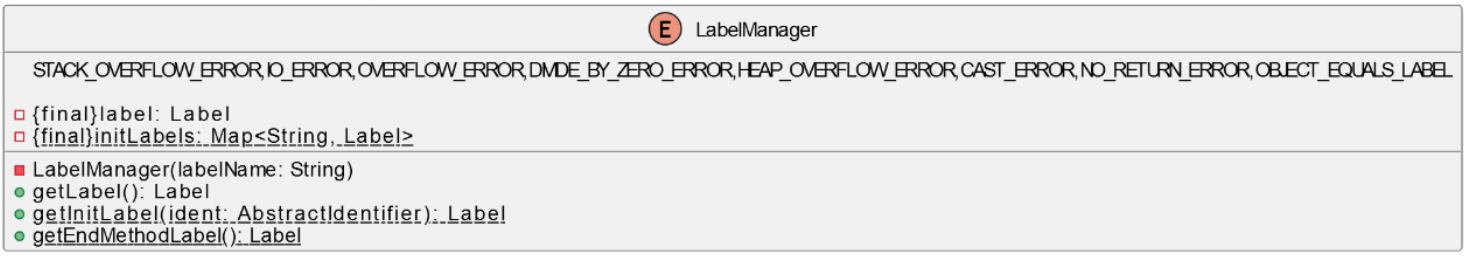


Fig. 1 : Schéma UML de la classe LabelManager

## Gestion des erreurs

La classe ErrorManager centralise la gestion des erreurs dans notre compilateur Deca. Elle génère automatiquement le code d'assemblage pour différents types d'erreurs, comme le dépassement de pile, les divisions par zéro, ou les erreurs d'entrée/sortie. Chaque type d'erreur est associé à un label unique défini dans la classe LabelManager, garantissant une identification précise des gestionnaires d'erreurs.

Le fonctionnement général repose sur une méthode générique generateError, qui insère dans le code d'assemblage les instructions nécessaires : affichage d'un message d'erreur, retour à la ligne, et arrêt du programme. Les erreurs spécifiques, comme les erreurs de débordement ou de cast, sont ensuite générées via des méthodes dédiées, permettant une organisation claire et une réutilisabilité du code.

Enfin, la méthode generateAllErrors permet d'initialiser en une seule étape l'ensemble des gestionnaires d'erreurs au démarrage du compilateur.

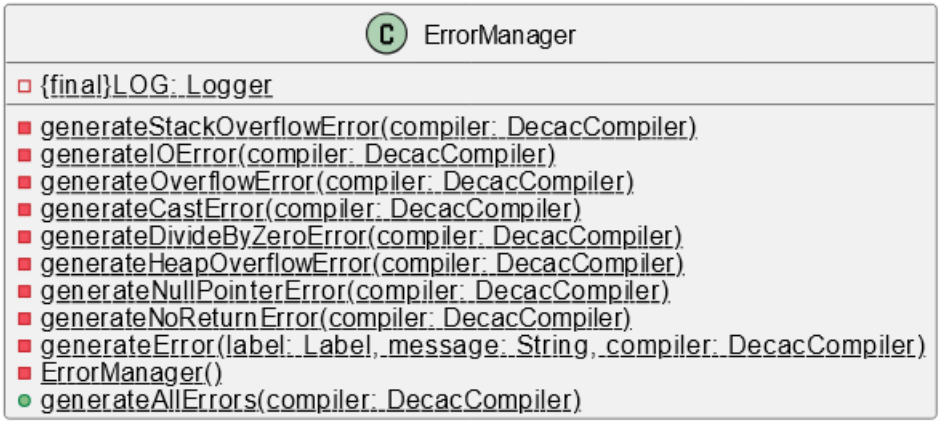


Fig. 2 : Schéma UML de la classe ErrorManager

## Gestion de la pile et des registres

La classe StackManager gère la pile et les registres pour les programmes IMA. Elle centralise :
- Allocation des registres : Suivi dynamique des registres disponibles et utilisés, avec des méthodes pour les allouer et libérer.
- Gestion des offsets : Maintien des offsets pour les registres globaux (GB), locaux (LB), et de pile (SP), essentiels pour l'allocation des variables.

Possibilité d'amélioration : Pour répondre au principe de *Single Responsibility*, il aurait été judicieux de

séparer la logique de gestion des registres de la pile. Cela aurait pu être possible en introduisant une nouvelle classe.*RegisterManager*.


FIG. 3 : Schéma UML de la classe StackManager

## Gestion de `Object`

Pour généraliser la gestion de `Object`, présent dans tous les programmes Deca, nous l'avons directement intégré à l'environnement des types. Sa génération de code étant fixe, une classe dédiée a été créée avec une

méthode de génération en dur.

Possibilité d'amélioration : Introduire un nœud `Object` dans l'arbre syntaxique après l'étape de parsing aurait permis une généralisation complète, en le traitant comme un nœud standard, intégré au reste du compilateur.

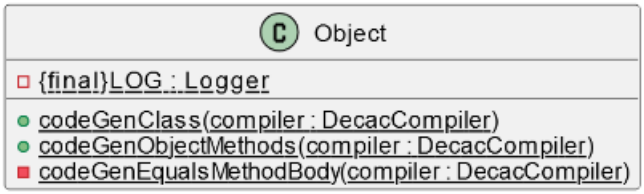


FIG. 4 : Schéma UML de la classe Object

## Création des tables de méthodes

La classe MethodTable centralise la gestion des tables des méthodes pour les classes du langage Deca. Elle initialise ces tables en allouant des espaces pour toutes les méthodes, y compris celles héritées, tout en générant des labels uniques pour chaque méthode.

Lors de la construction, elle prend en compte l'héritage en parcourant les classes parentes pour inclure les méthodes héritées ou les redéfinitions. La méthode codeGenTable génère ensuite le code d'assemblage pour construire la table en mémoire, en enregistrant les pointeurs vers les tables parentes et les adresses des méthodes.

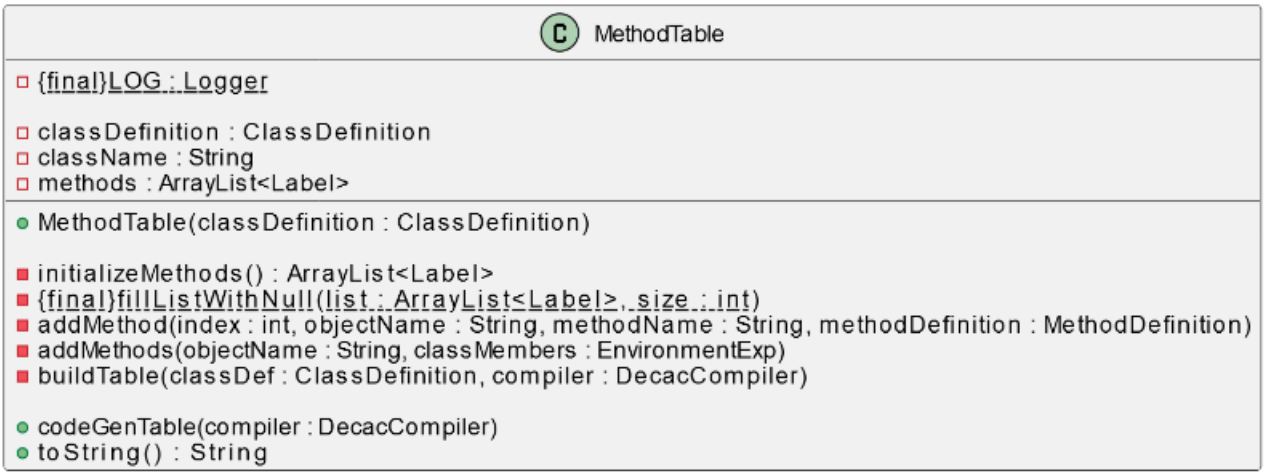


FIG. 5 : Schéma UML de la classe MethodTable

## Génération de code pour les constructeurs de classes

La classe Constructor gère la génération de code pour les constructeurs des classes Deca. Elle s'assure que tous les champs de la classe sont correctement initialisés, qu'il s'agisse des champs de la classe elle-même ou de ceux hérités de ses superclasses. Elle permet, entre autres :

- **L'initialisation des champs**

Les champs peuvent être initialisés explicitement via leur valeur définie ou implicitement à zéro si aucune valeur n'est spécifiée via les méthodes `initializeAllFieldsToZero` et `initializeFieldExplicitly`.

**- Gestion de l'héritage**

Pour les classes héritant d'autres classes, les nouveaux champs des superclasses sont initialisés avant ceux de la classe courante. L'initialisation explicite est ensuite réalisée en appelant le constructeur de la superclasse via une instruction `BSR`.

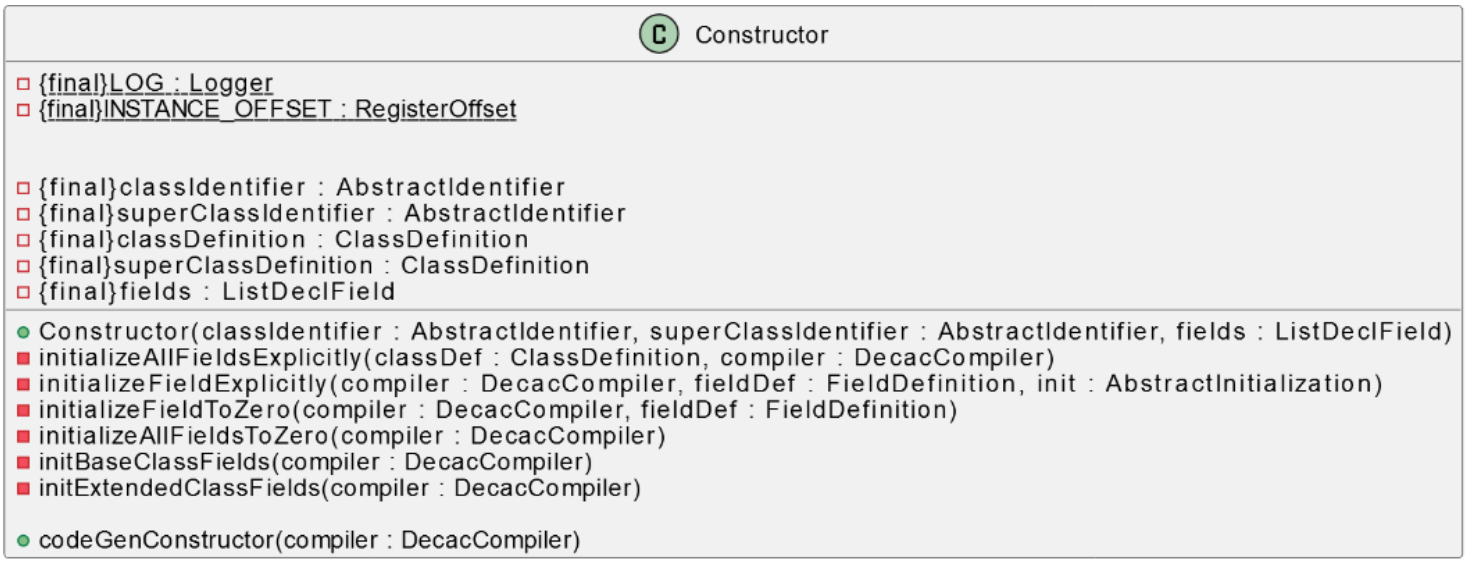


Fig. 6 : Schéma UML de la classe Constructor

# Conclusion

Au cours de ce projet, nous avons constaté qu'il est particulièrement difficile de concilier une architecture idéale avec un développement rapide dans un temps limité. Notre livrable comporte plusieurs aspects qui nécessiteraient un refactoring, réalisable avec davantage de temps. Cependant, nous pensons que cette approche pragmatique n'est pas forcément une mauvaise chose. Dans le métier d'ingénieur, il est courant de privilégier la livraison rapide de fonctionnalités pour répondre aux besoins pressants d'un client.

Le refactoring peut alors être envisagé ultérieurement, lorsque l'architecture devient trop complexe, que les dépendances s'accumulent ou que le code devient difficile à lire. Cette étape peut être réalisée de manière incrémentale, permettant d'améliorer progressivement le logiciel sans perturber les fonctionnalités existantes.

Par ailleurs, réfléchir à l'architecture dès le début d'un projet offre l'avantage de fournir une vue d'ensemble sur la manière dont les différentes briques s'articulent. En pratique, il est presque impossible de tout anticiper. Certains cas imprévus peuvent nécessiter des ajustements ou un refactoring pour adapter l'architecture à la réalité du développement.

Ainsi, ce projet nous a permis de mieux comprendre les compromis entre une conception réfléchie et une implémentation rapide, tout en réfléchissant aux améliorations possible pour optimiser notre architecture actuelle.