SIGL

www.sigl.epita.fr

Intégration - AGL
Groupe 4

Lisbonne

Dossier d'architecture technique de l'AGL et de ses passerelles

(DATAP)

Nb de Pages : 21	Version : 1.1	Date : 08/07/2015	Visa
Référence	EPITA_SIGL_LISBONNE_DATAP		
Auteurs	Thibaut HEYBERGER Clément LAFFAGE Guillaume PANGAUD Ludovic MEYER Valentin BAUDRY	08/07/2015	
Validateur	Thibaut HEYBERGER	08/07/2015	
Vérificateur	Ludovic MEYER	08/07/2015	
Approbateur	Clément LAFFAGE	08/07/2015	
Destinataire	Hervé Durand, Frédéric Fourdrinier		

Remarques de l'auteur :
Ce document est le dossier d'architecture technique de l'AGL et de ses passerelles, permettant par la suite sa réalisation

Historique du document:

N° version	Date	Auteur	Description
1.0	28/05/2015	Thibaut HEYBERGER Clément LAFFAGE Guillaume PANGAUD Ludovic MEYER Valentin BAUDRY	Version de référence
1.1	08/07/2015	Thibaut HEYBERGER Clément LAFFAGE Guillaume PANGAUD Ludovic MEYER Valentin BAUDRY	Mise à jour de la version de référence (précision des passerelles, génération de code métier, faisabilité)

Table des Matières

Aucune entrée de table d'illustration n'a été trouvée.

1 INTRODUCTION

1.1 Objet du document

L'objet de ce document est de :

- Identifier les composants de l'AGL
- Spécifier l'AGL d'entreprise
- Bâtir le référentiel de l'AGL
- Définir l'intégration des OGL et des méthodes en respectant les contraintes
- Envisager les passerelles, les paramétrages, la standardisation, la nomenclature et les règles
- Initialiser la démarche méthodologique cible de l'AGL

1.2 Objectifs du document

Les objectifs de ces documents sont de :

- Décrire l'implémentation de l'intégration des différents OGL, des passerelles et du référentiel de l'AGL
- Constituer une base solide pour la réalisation de l'AGL

2 SPECIFICATION DE L'AGL

2.1 Identification des composants

2.1.1 OGLs

- Dia, version 0.97.2
- IntelliJ IDEA Ultimate 14.1.2
- Junit, version 4.12 (plugin dans IntelliJ)
- Cucumber for Java, version 1.2.2 (plugin dans IntelliJ)

2.1.2 Outils de liaisons

2.1.2.1 De la conception à la réalisation

2.1.2.1.1 Code logiciel

Dia2Code, v 0.97.2, est l'outil qui transformera un diagramme de classe de conception conçu sous Dia en code source Java. C'est l'outil officiel de génération de code source et promu par Dia.

2.1.2.1.2 Base de données

Parse::Dia::SQL, v 0.23, est l'outil qui transformera un diagramme « database » conçu sous Dia en script SQL. C'est l'outil officiel de génération de scripts de création d'une base de données (Data Definition Language, DDL) et promu par Dia.

2.1.2.1.3 Flowchart To Code

Flowchart To Code est un outil réalisé par l'éditeur AthTek permettant de générer du code source à partir d'un d'un flowchart. Il sera utilisé pour générer la structure du code métier (structure des conditions et des boucles par exemple) de l'AGL

2.2 Contraintes

2.2.1 Techniques

2.2.1.1 Système d'exploitation

Le choix du système d'exploitation sur lequel tournera l'AGL se limite à Windows, Linux/Unix et Mac OS X étant donné que les OGL sont tous compatibles sur ces OS. Mac OS X est éliminé car Parse::Dia::SQL n'est disponible que sur Windows et Linux/Unix. L'OS Windows est choisi car il s'agit de l'OS le plus répandu en termes de part de marché et l'installation des différents outils se fait de manière plus simple pour un non-informaticien. La version 8.1 de Windows sera utilisée.

2.2.1.2 Conception logicielle

JUnit imposant le langage Java, ce langage étant très largement répandue – classé 2 au classement TOIBE 2013 – il est choisi comme langage de programmation pour la phase de réalisation de l'AGL.

L'outil de conception, Dia, est uniquement scriptable en python. Ce langage, en version 2.7 sera utilisé pour orchestrer les phases de conception et de réalisation de l'AGL.

Un pattern de conception est retenu pour l'architecture du logiciel, il s'agit du MVC, Model-View-Controller. Ce pattern, architecturé en trois couches élémentaires permet d'organiser le logiciel, de séparer les traitements, les données et les vues. Maîtrisé par les étudiants EPITA, il est compréhensible par un non informaticien ; une très large et accessible documentation étant disponible.

Avec une documentation très complète, une utilisation très répandue et son intégration dédiée dans l'IDE IntelliJ IDEA, le Framework Spring dans sa version 4.1.6 est retenu. Le Framework Hibernate dans sa version 5.0.2 sera utilisé afin de fournir une persistance des données et une génération de code performante.

L'outil intégré « Swing GUI Designer » de l'IDE Intellij IDEA sera utilisé pour la conception graphique.

Afin de satisfaire le bon fonctionnement des tests, les dépendances Maven ci-dessous seront ajoutées :

- Info.cukes:cucumber-java
- Com.sun.maven:maven-junit-plugin
- Info.cukes:cucumber-junit

2.2.1.3 Base de données

Le choix de la base de données se fait en fonction des critères suivants :

MySQL	PostgreSQL
ACID embryonnaire avec InnoDB	ACID compliant
Libre & Propriétaire	Libre
Documentation en français complète et exhaustive	

Le SGBDR PostgreSQL dans sa version 9.4.2 est sélectionné.

2.2.2 Méthodologiques

La méthode OOSE est imposée pour créer les séquences de modèles. Cette méthode est composée de quatre modèles principaux :

- Le « requirements model » qui définit les cas d'utilisation et les objets qui seront utilisés dans le système à développer
- L' « analysis model » qui définit les classes d'analyse du système : les entités, les contrôleurs et les interfaces. Ces classes définissent la structure logique du système et ne prennent pas en compte l'environnement d'implémentation
- Le « design model » qui définit les classes de conception du système. Les classes d'analyses sont adaptées à l'environnement d'implémentation.
- L' « implementation model » qui est le code source du système accompagné des commentaires

Pour implémenter les trois premiers modèles de la méthode OOSE, il est possible d'utiliser le formalisme UML. Or, l'outil DIA permet de faire de la modélisation UML. Les types de diagrammes utilisés pour implémenter ces trois modèles sont les suivants :

- Diagrammes de cas d'utilisation : ils permettent d'identifier toutes les fonctionnalités du logiciel à développer
- Diagrammes de classes d'analyse : ils sont à élaborer à partir des diagrammes de cas d'utilisation
- Diagrammes de classes de conception : ils adaptent les classes d'analyse à l'environnement d'exécution. Ils vont permettre de générer le squelette du code source du logiciel à développer

Les diagrammes de type « database » seront aussi utilisés. Ils permettent de modéliser la structure de base de données du logiciel à développer et ils permettront de générer le script de création de cette base de données (DDL).

2.3 Valeurs ajoutées

2.3.1 Traçabilité

Cette valeur ajoutée implique une traçabilité des exigences du produit à travers toutes les phases de l'AGL. Il doit aussi être possible de revenir à une phase précédente grâce à des liens entre les éléments de deux phases successives. Ainsi, si une modification est faite durant une des phases de développement, les autres phases doivent être notifiées de cette modification. Le référentiel a pour objectif de remplir le rôle de garantir la traçabilité des informations entre les différentes phases de l'AGL (cf. §4).
Cette valeur ajoutée peut être évaluée selon les critères suivants :

- Le niveau de granularité des informations gérées en traçabilité par le référentiel
- La capacité du référentiel de répercuter les changements faits durant une phase de développement dans les autres phases
- La capacité à tout moment du développement de savoir sur quelle exigence on est en train de travailler
- La capacité de savoir lors de la phase de conception à quel cas d'utilisation correspond un diagramme de classe d'analyse
- La capacité de savoir lors de la phase de conception à quelle classe d'analyse correspond une classe de conception
- La capacité de savoir lors de la phase de réalisation à quel diagramme de classe de conception correspond une classe dans le code source
- La capacité de savoir lors de la phase de test à quelle méthode du code source correspond un test unitaire
- La capacité de savoir lors de la phase de test à quel cas d'utilisation correspond un test fonctionnel

2.3.2 Open-Source

Cette valeur ajoutée peut être évaluée selon le critère suivant :

- L'accessibilité de tous les éléments de l'architecture de l'AGL aux étudiants de la majeure SIGL

2.3.3 Génération automatique de code

Cette valeur ajoutée implique que l'AGL puisse générer automatiquement du code, l'objectif étant d'atteindre au minimum 90% de code généré. Pour évaluer l'atteinte de cet objectif, il faut évaluer la proportion des différents types de code générables par l'AGL parmi l'ensemble du code de l'application finale produite.

Etant donné que l'AGL doit produit une application de gestion, il est indispensable de doter l'application d'une base de données. Ainsi, une telle application possèdera les différents types de code suivants :

- Une documentation (code source et base de données)
- Un squelette du code source
- Un script de création de la base de données (schéma)
- Des DAO (Data Access Object)
- Des opérations CRUD de modification de la base de données (Create, Read, Update, Delete)
- Une IHM (Interface Homme Machine)
- Du code métier (corps des méthodes)

On part du principe que les traitements sur les données dans une telle application sont assez limités et que les opérations à effectuer vont surtout consister en des opérations d'accès et de modification de la base de données (Create, Read et Update). Dans ce contexte, on établit le postulat suivant pour estimer la proportion des différents types de code parmi l'ensemble du code de l'application finale produite :

- Une documentation (code source et base de données) : 20%
- Un squelette du code source : 10%
- Un script de création de la base de données (schéma) : 10%

- Des DAO (Data Access Object) : 5%
- Des opérations CRUD de modification de la base de données (Create, Read, Update, Delete) : 20%
- Une IHM (Interface Homme Machine) : 20%
- Du code métier (corps des fonctions) : 15%

Ce postulat sera utilisé pour justifier l'estimation du pourcentage de génération automatique de code de l'AGL en fonction de ce que les OGL peuvent générer et des pertes d'information à la suite des différentes phases de développement.

2.3.4 Compréhensibilité par un non informaticien

Cette valeur ajoutée peut être évaluée selon les critères suivants :

- Les types de diagrammes UML utilisés dans la phase de conception de l'AGL
  - Par exemple, un diagramme de cas d'utilisation est plus facilement compréhensible par un non informaticien car il ne requière pas de compétences techniques spécifiques et permettra une meilleure compréhension du besoin
- Le niveau de granularité des diagrammes
  - Par exemple, un diagramme de cas d'utilisation global est plus facilement compréhensible qu'un diagramme de cas d'utilisation plus détaillé
- Les flux et types de données échangés entre chaque phase et entre chaque OGL et le référentiel
  - Ces flux sont-ils bien identifiés ?
  - Les informations échangées sont-elles de nature complexes ?

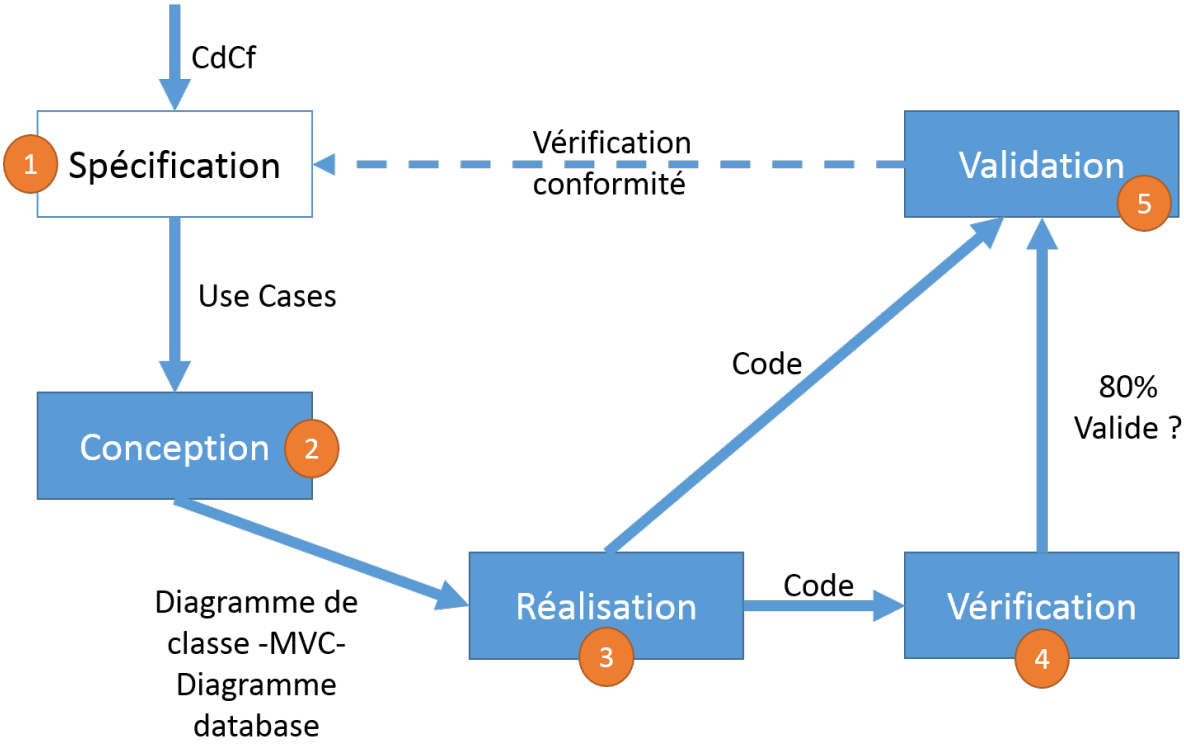
Cette valeur ajoutée sera surtout optimisée lors de la phase « Réaliser » du projet dans le cadre du dossier méthodologique de l'AGL.

3 ARCHITECTURE DE L'AGL

3.1 Architecture logique

L'architecture logique permet de mettre en avant les liens entre les phases de développement de l'**AGL**.

3.1.1 Schéma relation architecture logique

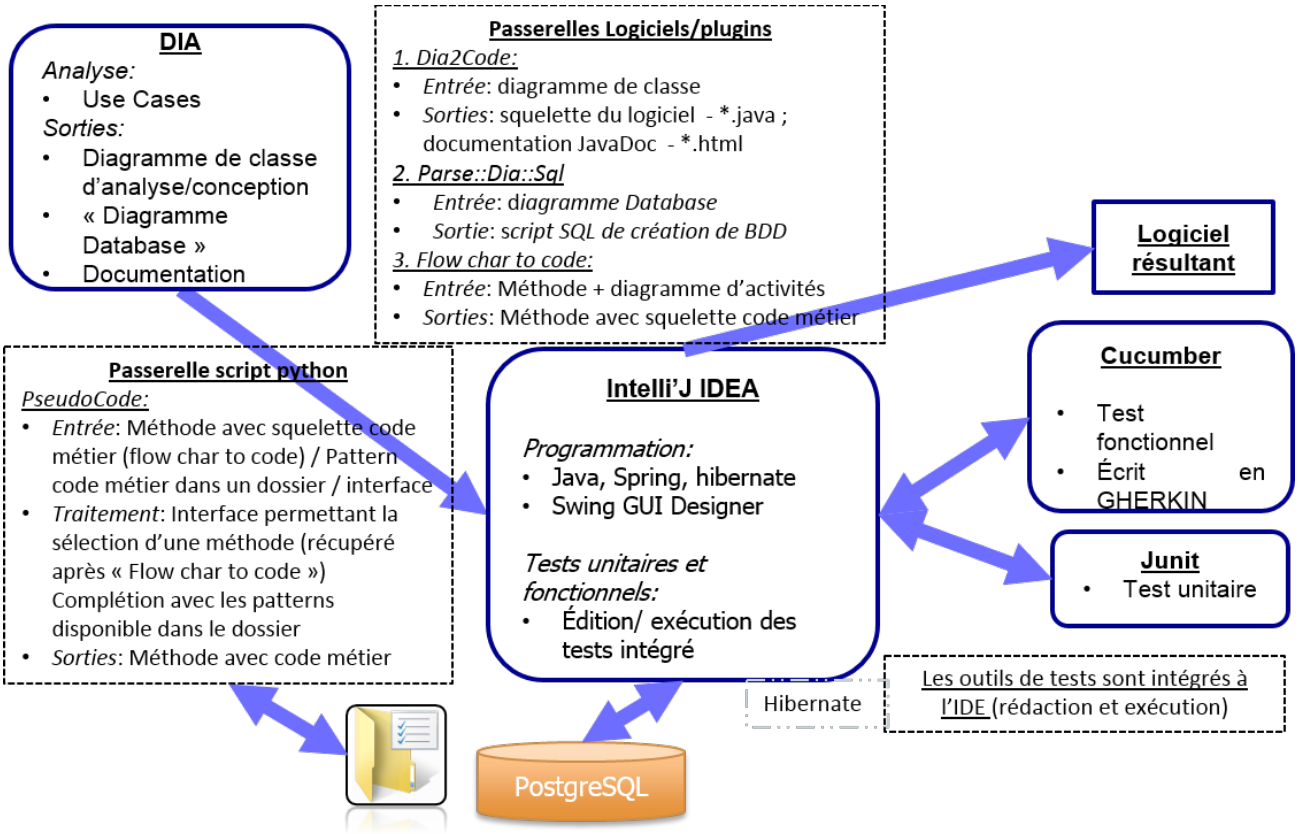


3.1.2 Type d'information transitoire

1. Traduction du **CdCF** en **cas d'utilisation** par l'utilisateur
2. Définition des diagrammes de classes d'analyse et de conception en respectant le modèle **MVC** (squelette de l'**AGL**)
3. Création du code source du logiciel à partir des diagrammes de classes de conception identifiés dans la phase précédente : création du squelette du code source des classes du logiciel
4. Vérification du code source via les tests unitaires
5. Validation de la conformité du code par rapport à la spécification exprimée lors de 80% de la validation des tests unitaires associés à la fonctionnalité testée

3.2 Architecture physique

3.2.1 Schéma de l'architecture physique



3.2.2 Description chronologique

Les étapes de l'AGL, en fonction du temps, sont :

- Conception : Nous réalisons les diagrammes de cas d'utilisation, les diagrammes de classes d'analyse et de conception et de la base de données au format « -.dia ». Les champs « commentaire » des classes, méthodes, attributs et paramètres des diagrammes de classes d'analyse et de conception seront complétés.
  - L'outil « Dia2Code » converti les diagrammes de classe de conception en code java, au format « *.java ». Il génère également de la documentation JavaDoc dans le squelette du code source à partir des commentaires insérés dans les diagrammes de classe d'analyse et de conception.
  - L'outil « Parse::Dia::SQL » converti le digramme de conception « Database » en script SQL de création de la base de données au format « *.SQL ».
  - L'outil Flowchart To Code va intéragir avec lutilisateur. Ce dernier va choisir une méthode, implémenter la structure métier puis enregistrer ce code généré dans la classe java.
- Réalisation : IntelliJ IDEA ouvre le code généré. Création automatique de la base de données, des données de connexion –dont les DAOs–. Programmation des algorithmes.
  - Test Unitaire avec JUnit, un test par méthode
- Test Fonctionnel : Rédaction des scénarios de tests en Gherkin pour une utilisation de Cucumber. L'Editeur est IntelliJ IDEA.

- Exécution

Les outils de tests étant intégrés dans l'outil de réalisation, aucun développement n'est nécessaire pour automatiser le passage d'une étape à une autre.

Les outils de conception et de réalisation et les outils de liaisons seront orchestrés par un script développé en python.

3.3 Plateforme d'exécution

Elle est basée sur du python car DIA contient de multiples plugins permettant un scriptage en python.

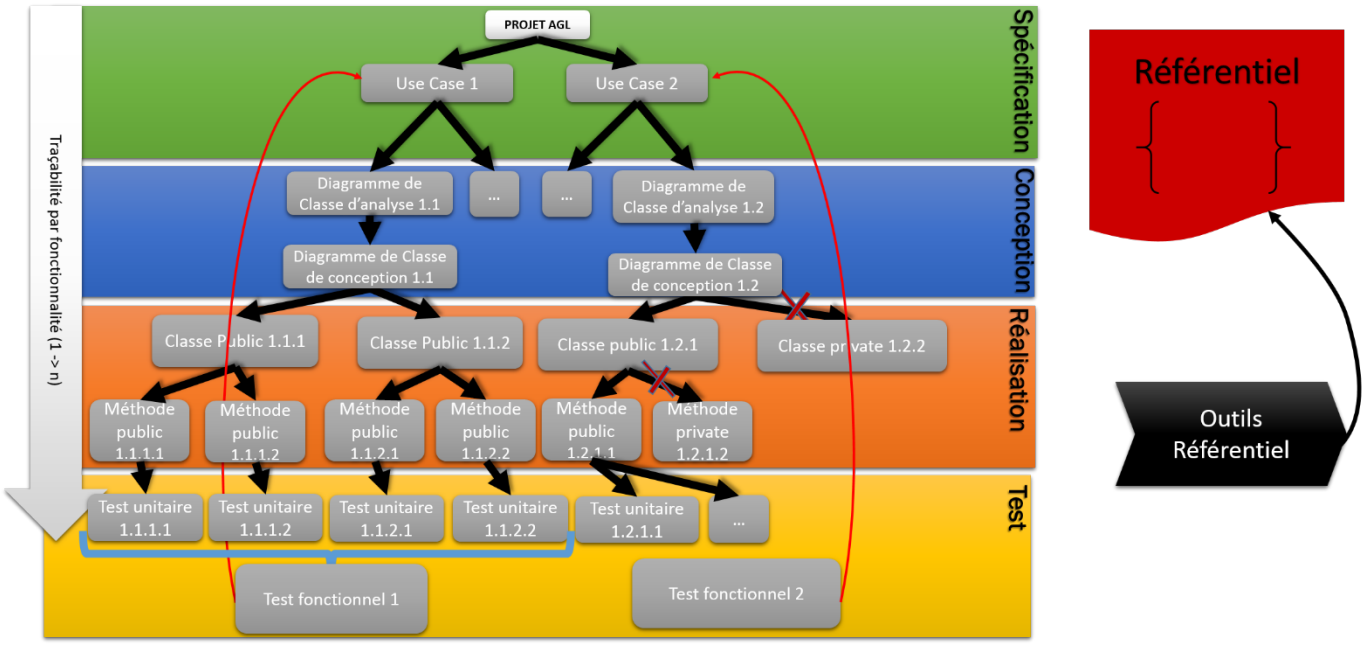
Les principes de cette plateforme sont les suivants :

- Il s'agit d'une IHM
- Elle permet de gérer plusieurs projets avec la capacité de :
  - Sauvegarder l'état d'avancement d'un projet
  - Réaliser une reprise sur activité
- Elle permet la gestion des passerelles
  - En particulier les plugins dia2code et Parse::Dia::Sql

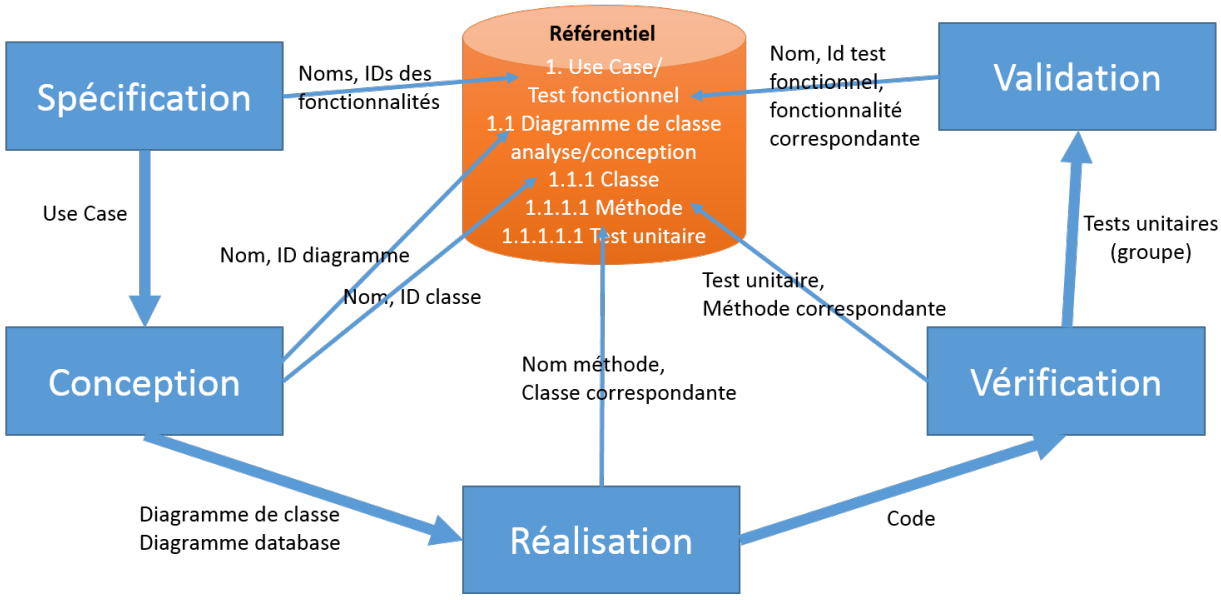
Cette plateforme d'exécution intégrera le référentiel (cf. *§4)*.

4 REFERENTIEL DE L'AGL

4.1 Sémantique



Architecture logique référentiel



Lors de chaque phase, les éléments nécessaires à la construction du référentiel sont envoyés à la phase N+1 afin de positionner correctement les éléments dans celui-ci.
Par exemple, la phase de conception envoie la liste des classes à la phase de réalisation. Ainsi, dans la

phase de conception, les informations disponibles sont : le nom de la classe ainsi que les méthodes associées. Le positionnement des méthodes *1.1.1.N* dans le référentiel se fera naturellement dans la classe située dans la partie *1.1.1*.

4.2 Architecture technique

La mise en place du référentiel se fera à l'aide du format JSON. Il répond parfaitement à nos attentes car il possède la capacité d'être facilement modifiable et organisé sous la forme de nœuds, ce qui répond parfaitement à notre problématique. Ce type de fichier permet de faire de recherche par niveau qui nous sera utile pour la modification ou la suppression de données.

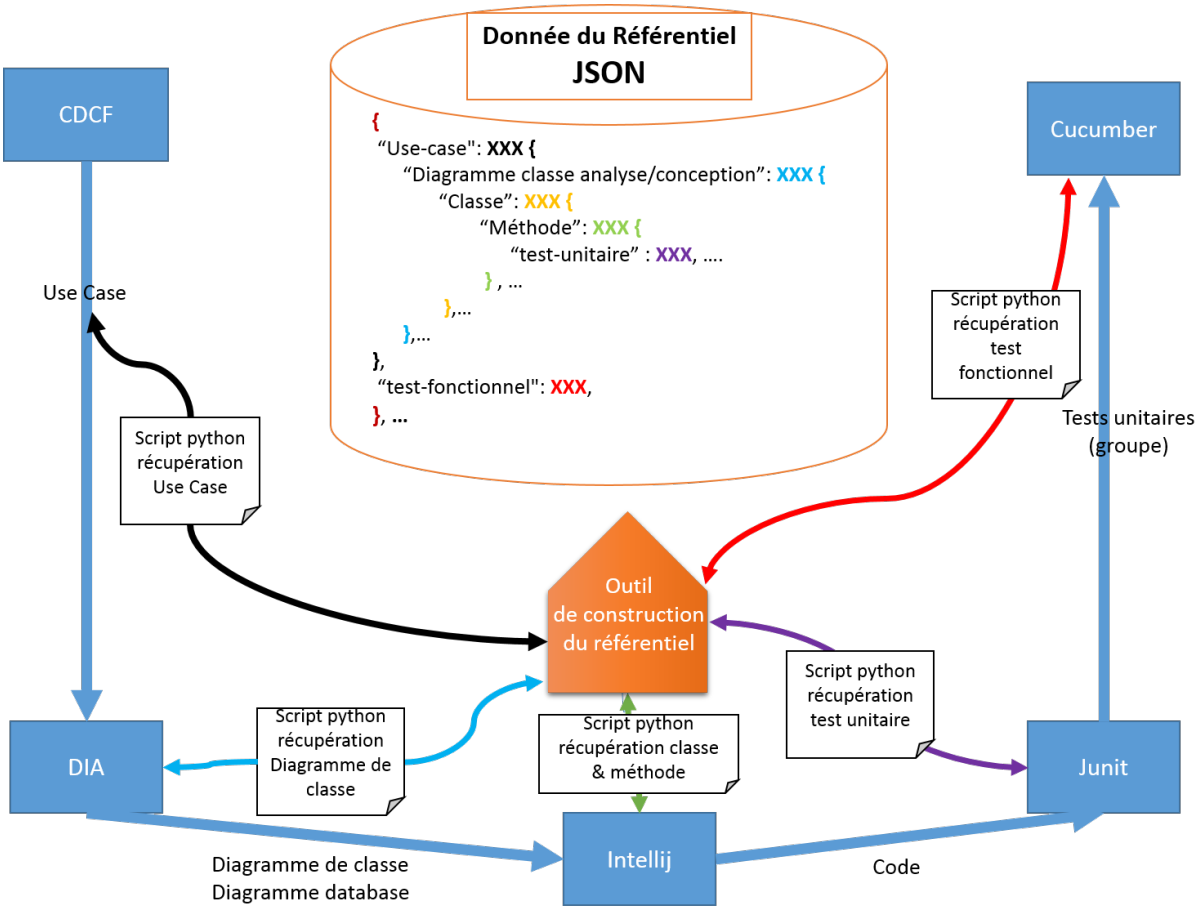
Pour faire la liaison entre le référentiel et les OGL, nous utiliserons un outil qui permettra de récupérer les diverses informations fournies par l'OGL pour mettre à jour le référentiel. Pour le bon respect de l'opération, nous devons respecter les règles de nommage des éléments. Ainsi pour les fonctionnalités seront classé de 1 à N puis pour la création des diagrammes nous reprendrons le chiffre de la fonctionnalité en ajoutant un décimal à ce nombre.

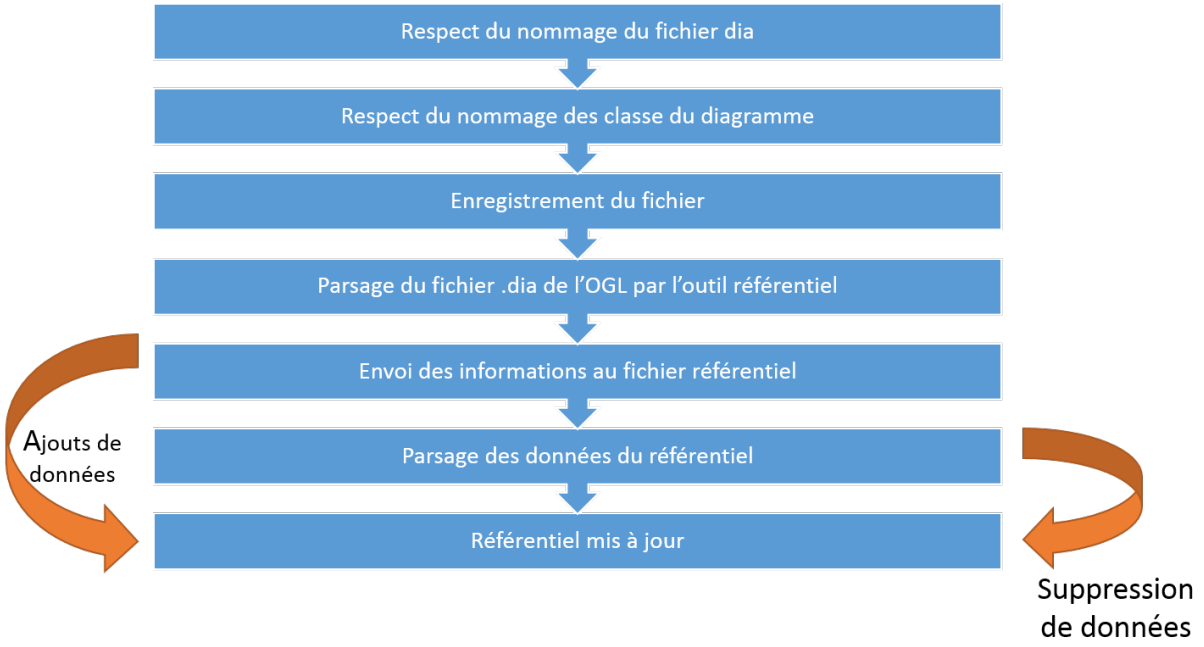
Pour la conception, les fichiers Dia sont construits par le formalisme XML. Il nous suffira donc de parser ce fichier et de récupérer les informations importantes comme les noms de classes puis de comparer les versions avec le référentiel. Si un élément du fichier dia n'est pas référencé dans le logiciel on le rajoute. Si l'élément est déjà présent, on ne fait rien. Si un élément du référentiel n'est plus présent dans le fichier Dia, on considère qu'il n'existe plus. Il est donc supprimé.

Il en va de même pour les fichiers de réalisation. Ils seront parsés par l'outil de référentiel puis ajoutés si nécessaire au bon endroit dans le fichier de réalisation.

Chaque mise à jour du référentiel se fait lorsque que l'on enregistre un fichier qui se rapporte au référentiel.

L'outil de référentiel ainsi que les données du référentiel seront rattachés à la plateforme d'exécution (cf. *§3.3*).





4.3 Aperçu du référentiel

Le référentiel est un outil interfacé avec nos OGLs permettant de façon simple et ergonomique d'accéder aux informations en traçabilité :

Besoin/Exigences	Elément architecture	Classe/fonction	Test	Résultat	Anomalie
1. Identification	1.1 Champ login	1.1.1 Classe login	1.1.1.1 ID1	1.1.1.1.1 **OK**	-
	1.2 Champ MdP	1.2.1 Classe MdP	1.2.1.1 ID2	1.2.1.1.1 **NOK**	1.2.1.1.1.1 Mauvais champs
2. ...					

De plus, le référentiel doit en cas de modification d'une information se mettre à jour et transmettre/propager la modification aux éléments concernés.

5 FAISABILITE

5.1 Démarche de faisabilité

Afin de calculer la faisabilité de notre AGL, nous allons :

- Lister les tâches à réaliser
- Estimer la charge de travail associée en h*H (heures Hommes) et les risques à prendre en compte
- Prioriser les tâches par urgence et importance
- Planifier et affecter les ressources

5.2 Liste des tâches à réaliser

Architecture :

- Passerelle 1 : Entre Dia et Flowchart to code
- Plateforme d'exécution

Référentiel :

- Script Python 1 : Récupération Use Case
- Script Python 2 : Récupération diagramme de classes
- Script Python 3 : Récupération Classes et méthodes
- Script Python 4 : Récupération tests unitaires
- Script Python 5 : Récupération tests fonctionnels
- Interface et IHM du Référentiel (dans la plateforme d'exécution)
- Interfaçage Base de données

5.3 Estimation de la charge de travail

5.3.1 Charge de travail

Architecture :

Tâche	Charge (h*H)	Description
Passerelle 1 : Entre Dia et Flowchart to code	8	6h codage, 2h Test + débogage
Plateforme d'exécution	45	25h IHM, 10h intégration des passerelles et scripts, 10h Test + débogage

Référentiel :

Tâche	Charge (h*H)	Description
Script Python 1 : Récupération Use Case	6	4h codage, 2h Test + débogage
Script Python 2 : Récupération diagramme de classes	6	4h codage, 2h Test + débogage
Script Python 3 : Récupération Classes et méthodes	6	4h codage, 2h Test + débogage
Script Python 4 : Récupération tests unitaires	6	4h codage, 2h Test + débogage
Script Python 5 : Récupération tests fonctionnels	6	4h codage, 2h Test + débogage
Interface et IHM du Référentiel (dans la plateforme d'exécution)	-	Cf plateforme exécution

Charge totale estimée :

83 h*H

5.3.2 Risques

Risques identifiés :

- Problèmes d'interfaçage des scripts
- Débogage des scripts plus long que prévu
- Retard dans le travail
- Tâches plus complexes que prévues initialement

5.4 Priorisation des tâches

L'importance détermine le poids d'une tâche dans l'AGL final, de 1 à 5.

L'urgence détermine si elle doit être réalisée en priorité en termes de temps, de 1 à 5.

La priorité détermine l'ordre de la tâche. Plusieurs tâches peuvent être concurrentes.

Tâche	Urgence	Importance	Priorité
Passerelle 1 : Entre Dia et Flowchart to code	5	5	**1**
Plateforme d'exécution	2	4	**6**
Script Python 1 : Récupération Use Case	4	4	**2**
Script Python 2 : Récupération diagramme de classes	4	4	**3**
Script Python 3 : Récupération Classes et méthodes	3	4	**4**
Script Python 4 : Récupération tests unitaires	2	3	**5**
Script Python 5 : Récupération tests fonctionnels	2	3	**5**
Interface et IHM du Référentiel (dans la plateforme d'exécution)	-	-	-

5.5 Planification

5.5.1 Ressources

L'équipe est composée de 4 ingénieurs-développeurs :

- Ingénieur 1
- Ingénieur 2
- Ingénieur 3
- Ingénieur 4

5.5.2 Planning

- 6 semaines travaillées d'ici au mois de septembre
- 83 h*H
- 4 personnes

Charge totale estimée :

4,5 h de travail par semaine par personne

➔ AGL réalisable dans les temps !

6 ATTEINTE DES VALEURS AJOUTEES

6.1 Traçabilité

Le référentiel permet une traçabilité par fonctionnalités 1 → n. Le niveau de granularité des éléments gérés en traçabilité est très fin : à l'échelle d'une méthode.

6.2 Open source

Cette valeur ajoutée est respectée car tous les éléments de l'architecture de l'AGL et de ses passerelles sont accessibles aux étudiants de la majeure SIGL.

6.3 Génération automatique de code

Par rapport au postulat établit au *paragraphe 2.3.3*, aux fonctionnalités des différents outils et des données qu'il est possible de faire passer entre chaque outil/phase, il est possible d'estimer le pourcentage de code générable par l'AGL :

Type de code	Proportion (postulat)	Code générable
Documentation	20%	20%
Squelette code source	10%	10%
Script création BD	10%	10%
DAO	5%	5%
Opérations CRUD	20%	13%
IHM	20%	13%
Code métier	15%	9%
Total	100%	80%

La documentation et le squelette du code source sont générés à 100% par dia2code à partir des diagrammes de classe de conception créés à l'aide de Dia. Le script de création de la base de données est généré à 100% par Parse::Dia::SQL à partir des diagrammes « database » créés à l'aide de Dia. Les DAO sont générables à 100% dans IntelliJ grâce à Hibernate. Nous estimons que les opérations CRUD et l'IHM sont uniquement générables à 65% par IntelliJ. Le code métier est généré à 60% grâce à l'outil FlowChart To Code qui permet de générer la structure d'une méthode telle que les conditions et les boucles. Finalement, il est estimé que l'AGL permettra de générer 80% du code.

6.4 Compréhensibilité par un non informaticien

Les diagrammes UML utilisés lors de la phase de conception sont simples à comprendre par un non-informaticien :

- Les diagrammes de cas d'utilisation permettent à l'utilisateur d'identifier toutes les fonctionnalités du logiciel à concevoir
- Les diagrammes de classe d'analyse permettent pour chaque cas d'utilisation d'identifier les entités, les interfaces et les contrôleurs associés
- Les diagrammes de classe de conception permettent d'adapter les classes d'analyse à l'environnement d'implémentation

De plus, les flux entre chaque phase et chaque outil de l'AGL sont bien identifiés (cf. *§3.1* et *§4.1*).