**PLAN**

1. **PRESENTATION DU PROJET**
2. **OUTIL D’ANALYSE UTILISE**
3. **ANALYSE DU CODE**
4. **INTERPRETATION**
5. **METRIQUES APPLIQUEES**

* **LINES OF CODE**
* **COMPLEXITE CYCLOMATIQUE**
* **METHOD LEVEL**

1. **PRESENTATION DU PROJET**

Notre projet consiste à écrire un programme pour la gestion des fruits d’un super marché. Dans la démarche de notre analyse nous avons considéré que le Super Marché est composé d’un ensemble de rayon dans les quelles sont ranger un type spécifique de fruit identifiable par l’étiquette du rayon. Nous avons utilisé au total quatre (04) classes à savoir :

* **Classe DriverSuperMarcher**

Classe permettant de diriger comme son nom l’indique toute activité de notre super Marcher ;

* **Classe SuperMarcher**

D’après notre règle de gestion, la classe Supermarché est incontournable car celui-ci a une importance capitale elle est le système d’information que l’on aimera implémenter (Modélisé) ;

* **Classe Rayon**

La classe Rayon a été choisie car selon le modèle réel considéré tous le super marché est organisé en rayon dans lesquels sont disposés différents fruits ;

* **Classe Fruit**

Cette classe est la classe des objets manipulés dans le super marché avec ses attributs : Nom, type, prix, poids, couleur, durée de vie.

1. **OUTILS D’ANALYSE UTILISE : JArchitect**



JArchitect est principalement dédié à l’analyse de code en langage java. JArchitect est l’outil d’analyse le plus exhaustif qui analyse :

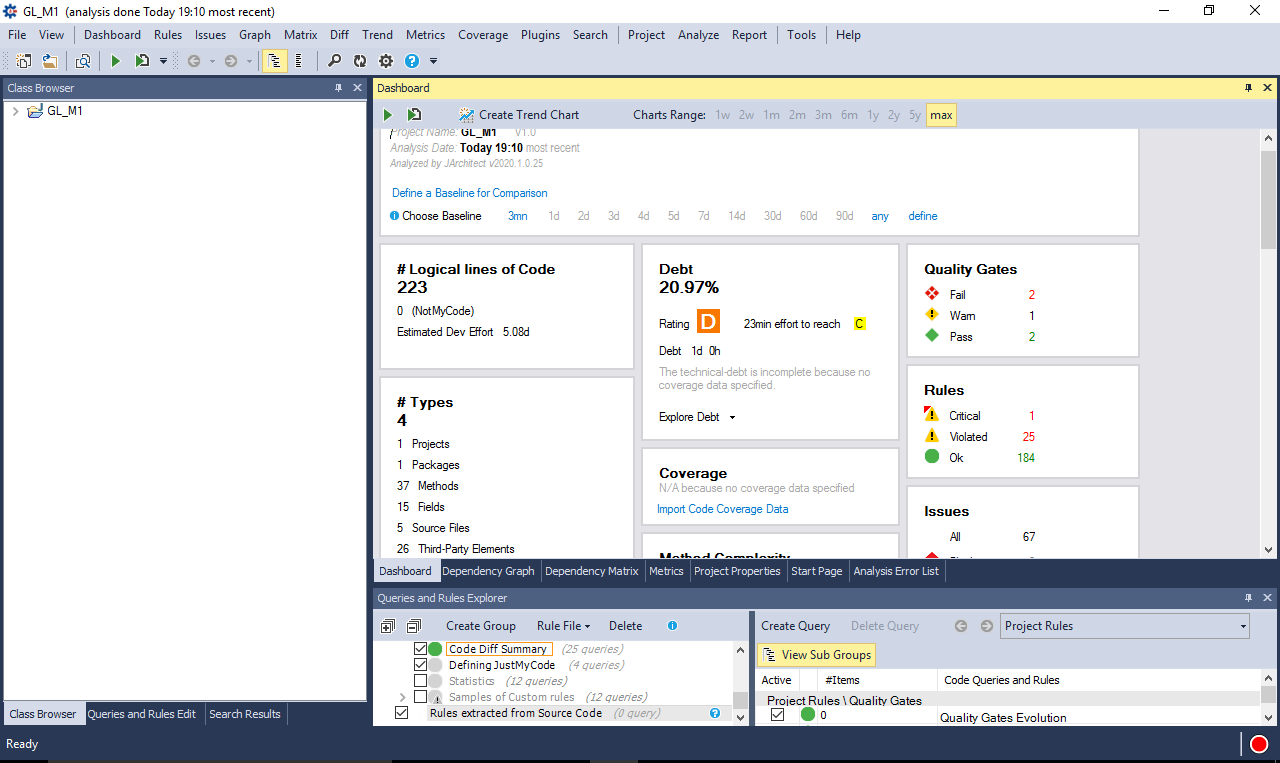
* Hiérarchie d’appel ;
* Consommation de mémoire ;
* Complexité de code ;
* Couplage fonctionnel ;
* Défauts architecturaux dans la mise en œuvre.

JArchitect est utilisé par des géants comme Samsung, Google, LG et autres. Cela permet de vérifier à quel point l’outil est puissant.

L’analyse de notre projet est décrit la partie suivante.

# **ANALYSE DU CODE SOURCE**

La figue suivante montre de façon global l’analyse de notre code à l’aide de JArchitect.



***Figure 1 : analyse du code***

1. **INTERPRETATION**

Notre projet compte :

* Deux cent vingt-trois (223) lignes de code ;
* Quatre (04) classes ;
* Un (01) package ;
* Trente-sept (37) méthodes ;
* La complexité des méthodes est de 12 au maximun et 2.28 en moyenne.

**METRIQUES APPIQUEES**

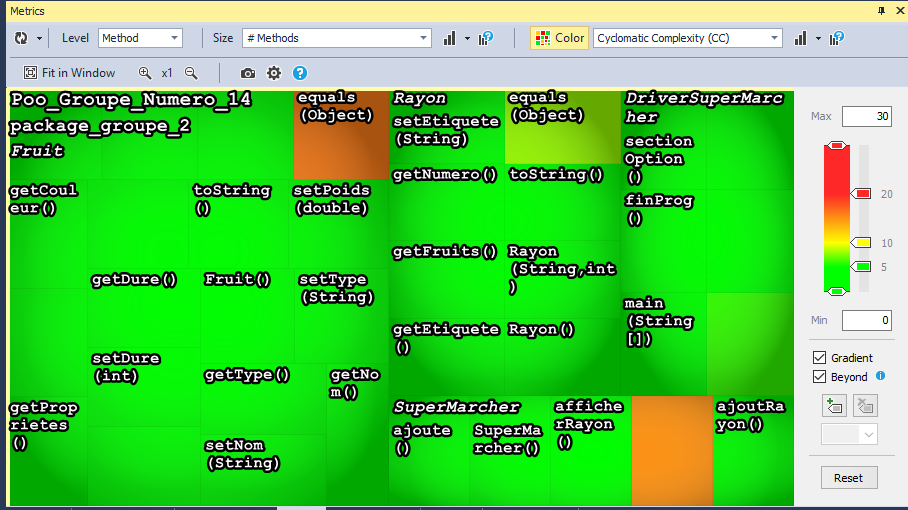
* L**INES OF CODES**



***Figure2 : lines of code***

La figure ci-dessus présente le nombre de ligne de code de chaque méthode. Les méthodes qui sont en vert ont un nombre de ligne de code compris entre 0 et 17, celui des méthodes qui sont très légèrement jaune est compris entre 18 et 19 et pour celui des méthodes en rouge est compris entre 30 et 50 au maximun.

* **COMPLEXITE CYCLOMATIQUE**



***Figure 3 : complexité cyclomatique***

La figure ci-dessus présente la complexité cyclomatique de chaque méthode. Les méthodes en vert ont une complexité cyclomatique compris entre 0 et 7, celle qui est légèrement jaune a une complexité compris entre 9 et 10, celles en orange ont une complexité comprise entre 11 et 15.

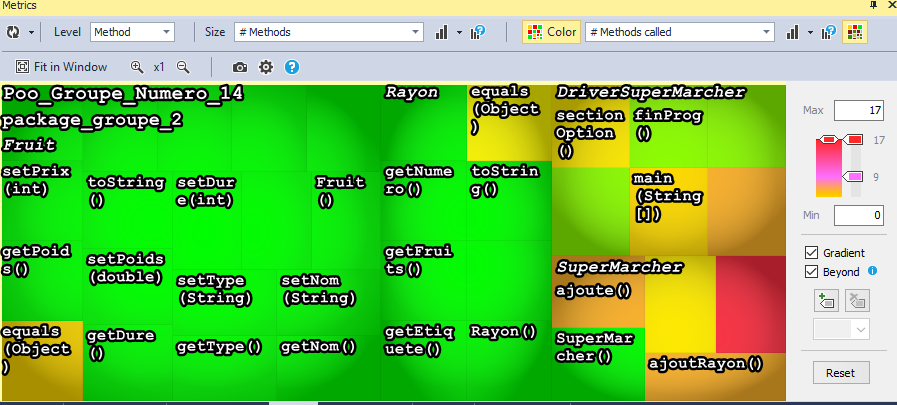
* **METHOD LEVEL**



***Figure 4 : Method level***

La figure en dessus indique le niveau de méthode de chaque méthode. Le niveau de méthode des méthodes en verte est de zéro (00), celui des méthodes en jaune est de 0.6, et celui des méthodes en violet est de trois (03), celui de la méthode en orange est compris entre trois (03) et un (01) et celui de la méthode qui tend vers le rouge est compris entre 3.5 et 4.5.

* **METHODES APPELEES**



***Figure 5 : méthodes appelées***

**CONCLUSION**

Les sont des moyens efficaces pour mesurer la complexité, la qualité et la maintenabilité d’un logiciel. Ces métriques servent également à localiser les modules difficiles à tester et à maintenir. Des actions correctives peuvent alors être enclenchées pour corriger une complexité trop élevée plutôt que de garder des modules susceptibles d’être cher en maintenance.