Rapport TER :

Etude qualitative et Reengineering d'un framework

Encadré par Monsieur :

Christophe DONY.

Réalisé par :

Ahmad Fraidoun FAQIRI.
Imane HAFFANE.
Marylène PHAROSE.
Younes BENZAKI.

REMERCIEMENTS :

Nous tenons à remercier Monsieur Christophe DONY pour son encadrement, sa disponibilité et ses conseils précieux, nous permettant ainsi de cerner notre projet dès le début de notre mission.

Nous remercions également notre Rapporteur Monsieur Abdelhak-Djamel SERIAI pour le temps qu'il va consacrer à la lecture de ce Rapport.

Enfin, Nous remercions l'ensemble des jurys ainsi que le corps enseignants de la Faculté des Sciences de Montpellier II.

SOMMAIRE :

E. Amélioration de la classe Entity :
    1. Refactoring des méthodes Collide.
    2. Refactoring des méthodes Move.
F. Gestion des Exceptions :
    3. Problème rencontré.
    4. Solution apportée.
G. Constatation des améliorations.

Section IV : Conclusion et Synthèse du TER.

Bibliographie.

Annexe :

a. Tableau de figures
b. Règles de violation relatives à l'analyse qualitative.
c. Les fiches de Lecture du Livre : « Object-Oriented Reengineering Patterns ».

Section I : Introduction

A. Objectifs du Projet :

Le but de notre projet est l'analyse et l'amélioration de la qualité logicielle d'un programme. Ce programme contient un framework, nommé « bounding-ball », dédié à la réalisation de jeux d'arcades dans lesquels des objets mobiles divers peuvent se rencontrer entre eux, ou rencontrer des contrôleurs animés par un utilisateur.

Le programme que nous avons récupéré inclut également trois extensions, i.e. trois jeux qui sont : "casse-brique", "space invaders", "tetris" ou "pack man".
L'ensemble a été réalisé précédemment par un autre groupe de TER.

Notre projet consiste en premier lieu à évaluer ce programme, sa qualité logicielle, sa réutilisabilité et en second lieu en une restructuration basée sur l'évaluation utilisant les techniques de l'ingénierie inverse, du "refactoring" et du "reengineering".
L'évaluation doit utiliser différents outils et techniques : une analyse manuelle du code (lecture, exécution, constatation des performances) et une analyse automatisée utilisant des outils dédiés au calcul de métriques. Le choix, l'installation et l'utilisation d'outils appropriés de calcul de métriques fait également partie du problème qui nous a été posé. L'analyse manuelle quant à elle, doit également être basée sur l'étude du livre « Object-Oriented Reengineering Patterns », qui fournit un ensemble de schémas d'amélioration et de restructuration de logiciels écrits avec des langages à objets.
La phase d'évaluation doit conduire à la présentation d'un ensemble de propositions d'amélioration.

Notre projet consiste ensuite en la réalisation des améliorations proposées. Il s'agit donc de faire une nouvelle version du framework qui soit mieux structurée, mieux documentée, plus simple à réutiliser et à étendre. Les jeux que nous avons récupérés doivent être réécrits en conséquence, et éventuellement nous pouvions en réaliser un nouveau.

Nous avons eu comme objectifs alors, de :

- Comprendre le code.
- L'utiliser pour créer une nouvelle application.
- Le tester avec des outils de calculs afin de détecter ses qualités et ses défauts.
- et enfin, l'améliorer en s'appuyant sur les concepts de refactoring et de reengineering.

B. Comment s'organiser pour atteindre ces Objectifs ?

1) Pour connaitre et comprendre le processus que doit suivre un tel projet de réingénierie logicielle, il fallait étudier et faire les fiches de lecture du Livre « Object-Oriented Reengineering Patterns » fourni par notre tuteur que vous trouverez en Annexe.

2) Dans un premier temps, nous étions amenés à analyser différentes techniques de réingénierie à travers les documents et les livres que nous avons étudiés.

3) Dans un deuxième temps, nous avons pris contact avec le code source et les diagrammes de classes du Framework, en essayant de surmonter la complexité du système et le manque de commentaires et de documentation.

4) Plusieurs tests à travers différents jeux réalisés précédemment, étaient nécessaires pour comprendre le fonctionnement.

5) La création de notre propre jeu était une façon pratique pour mettre la main sur l'essentiel.

6) Pour améliorer l'existant, il fallait faire des tests pour détecter rapidement les entités potentiellement problématiques. Il s'agit là du calcul métrique que nous avons effectué avec plusieurs outils.

7) Une revue du code a été impérative afin de comparer les résultats des métriques et de s'assurer de la conformité du diagnostique établi.

8) Appliquer quelques Schémas de conception afin de restructurer le Framework était une première étape d'amélioration.

9) Nous étions amenés ensuite à détecter et réduire le code dupliqué qui empêche toujours les modifications futures vu que chaque morceau dupliqué doit être mis à jour partout ce qui engendre des bogues en cas d'oubli.

10) Au long de notre étude du code source nous avons remarqué que les exceptions n'étaient en aucun cas traitées, d'où notre idée de prendre en charge ceci.

11) L'amélioration de la Java doc était à réaliser.

12) Enfin, nous avons pris le soin de comparer l'ancien et le nouveau Framework et donc les résultats obtenus avant et après modifications à l'aide des mêmes outils utilisés précédemment.

C. Intérêt du Projet?

Créer un logiciel pour la première fois est un travail personnel basé sur le besoin des utilisateurs d'un système ou d'une entreprise donnée. Cela nécessite une créativité et un savoir faire, bref une implémentation qui suit notre propre raisonnement fondé sur notre connaissance et notre capacité à répondre à toutes les attentes de notre client initial.

Au fil du temps, ce même logiciel aura à s'adapter aux changements survenus : ceux liés aux nouveaux besoins du marché, ou alors ceux liés aux demandes d'amélioration de fonctionnalités de la part des utilisateurs pour plus de facilité d'utilisation.

L'ajout de nouvelles fonctionnalités ne devrait alors poser aucun problème et le coût de maintenance devrait rester raisonnable. Sans ces deux critères le remplacement de ce logiciel sera envisagé.

C'est pour cela que le refactoring des logiciels existants constitue l'activité pour laquelle plusieurs entreprises optent.

Il s'agit en fait d'une maintenance préventive et adaptative, son but essentiel sera de réduire les coûts de maintenance tout en améliorant la qualité de service au sens performance et fiabilité. Le comportement du logiciel ne changera pas, autrement dit, rajouter de nouvelles fonctionnalités ne fera pas partie de cette activité.

La réingénierie de ce logiciel doit pouvoir s'appliquer aussi à l'intégration de nouvelles contraintes ou alors l'ajout de fonctionnalités.

Notre projet portera alors sur ces activités. Notre point de départ est le Framework « Bouding-balls » et nous commençons par l'ingénierie inverse qui représente la technique qui permet de déterminer l'utilité et le fonctionnement de l'application à travers la compréhension d'un code mal documenté pour notre cas.

D. Outils, méthodes et répartition des taches :

1. Outils de travail :

Le Framework que nous avons à notre disposition a été implémenté en Java. Le choix d'utiliser l'IDE Eclipse a été basé sur le fait qu'il regroupe un ensemble d'outils pour le développement de logiciels et en particulier ceux dédiés au refactoring. On prend comme exemples: le renommage d'identifiant, le changement de la localisation de méthode, la généralisation ou alors la spécialisation d'une classe ou d'une méthode, l'encapsulation des données et plusieurs autres fonctionnalités intéressantes.

2. Méthode de travail :

Une réunion a été organisée chaque semaine afin d'avancer le projet et de compléter l'écriture du rapport en parallèle. Les problèmes rencontrés étaient discutés et d'autres objectifs étaient fixés pour la semaine d'après.

Les réunions organisées avec notre tuteur nous ont permis de se mettre sur la bonne voie et de corriger nos erreurs.

Un contact régulier par mail nous a permis d'avoir un suivi afin de garantir l'avancement des parties de chaque membre du groupe.

3. Répartitions des taches :

Taches	Younes Benzaki	Faqiri Ahmad Fraidoon	Haffane Imane	Pharose Maryléne
Résumé des chapitres	*	*	*	*
Création du jeu	**	**		
Lecture du code	*	*	*	*
analyse quantitative (Metrics)	*	**	*	*
Bilan analyse quantitative	**	*	*	*
Analyse qualitative		*		
Bilan analyse qualitative		*		
Découpage de GameEngine	**	**	*	*
Refactoring de CollisionDetection	**	*	*	*
Réécriture de GetEntityAt	**	*	*	*
Application du pattern Stategy	*	**		
Gestion des exceptions			**	*
Réécriture de la Javadoc			*	**
Refactoring de GamePackage	*	*	*	*
Ecriture du Rapport	**	**	***	**

Section II. Analyse du Framework

A. Analyse de l'architecture générale du Framework

Afin d'avoir une vision globale du fondement de ce framework, nous avons généré les différents diagrammes UML afin de comprendre sa structure d'une part et d'étudier l'interaction entre ses composants et ses différentes couches d'autres parts.

Dans cette partie, nous allons vous présenter l'ensemble des packages de notre application tout en mettant en évidence quelques classes que nous avons jugées importantes.

'Bounding-ball' est un framework composé des six packages que nous retrouvons dans la figure suivante :

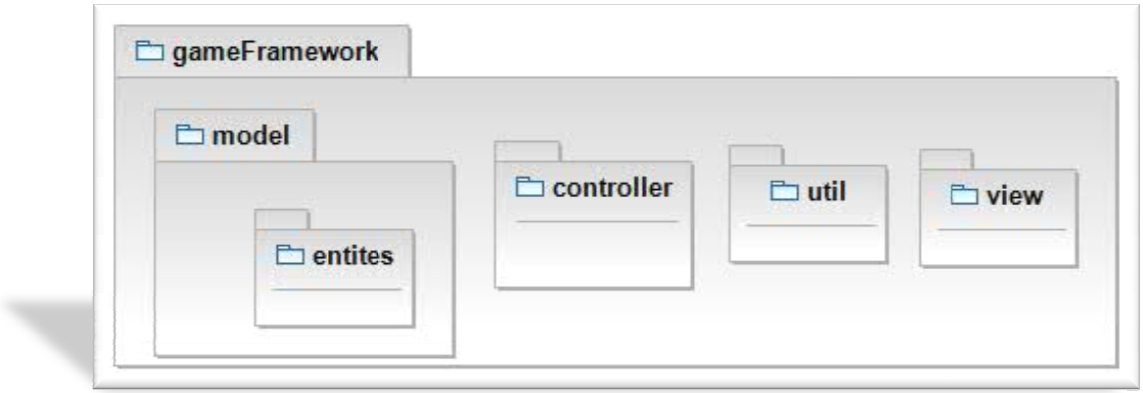


Figure II.A.1 : diagramme de package

1) Package gameFramework

Ce package sert de conteneur du reste des packages du Framework. Il ne contient par contre que la classe Main du Framework.

2) Package model

Ce package représente la couche métier ou modèle du Framework et contient la classe 'GameEngine' considérée comme le cœur du Framework.

La classe GameEngine

Cette classe étant la classe principale du Framework, elle prend en charge plusieurs responsabilités telles que le chargement et le déchargement des différents jeux sur différents niveaux ainsi que la construction et l'exécution des parties de jeux. En outre elle se charge de

la gestion et de la détection de collisions entre les différentes entités des jeux. Ce qui en fait la classe la plus volumineuse du Framework.

La classe GameSubject

Cette classe sert comme un conteneur de tous les observateurs du Framework. Elle sert principalement à déclencher des événements suite à un changement d'état du Framework pour notifier les observateurs des jeux afin qu'ils puissent mettre à jour leurs états internes.

La classe GameManager

Cette classe ne contenant que des méthodes abstraites sert comme interface entre le cœur du Framework et les classes des différents jeux. Elle sert principalement à initialiser les niveaux du jeu et à vérifier son état à tout instant. Chaque jeu doit implémenter les méthodes de cette classe afin de permettre au Framework de contrôler leur flot d'exécution.

La classe Timer

Cette classe représente l'horloge du jeu et émet à des intervalles de temps réguliers un signal à chacun de ses observateurs pour qu'ils puissent se mettre à jour.

L'interface TimerListener

Cette interface doit être implémentée par tous les objets désirant avoir la possibilité de recevoir les événements du Timer.

La classe Player

Cette classe représente la notion de joueur et toutes les informations lui concernant telles que le score, les points de vie, etc.

3) Package controller

Ce package représente la couche contrôleur du Framework. Il joue un rôle d'intermédiaire entre les couches métier et la présentation du Framework.

Et voici le diagramme de classes du package Model :

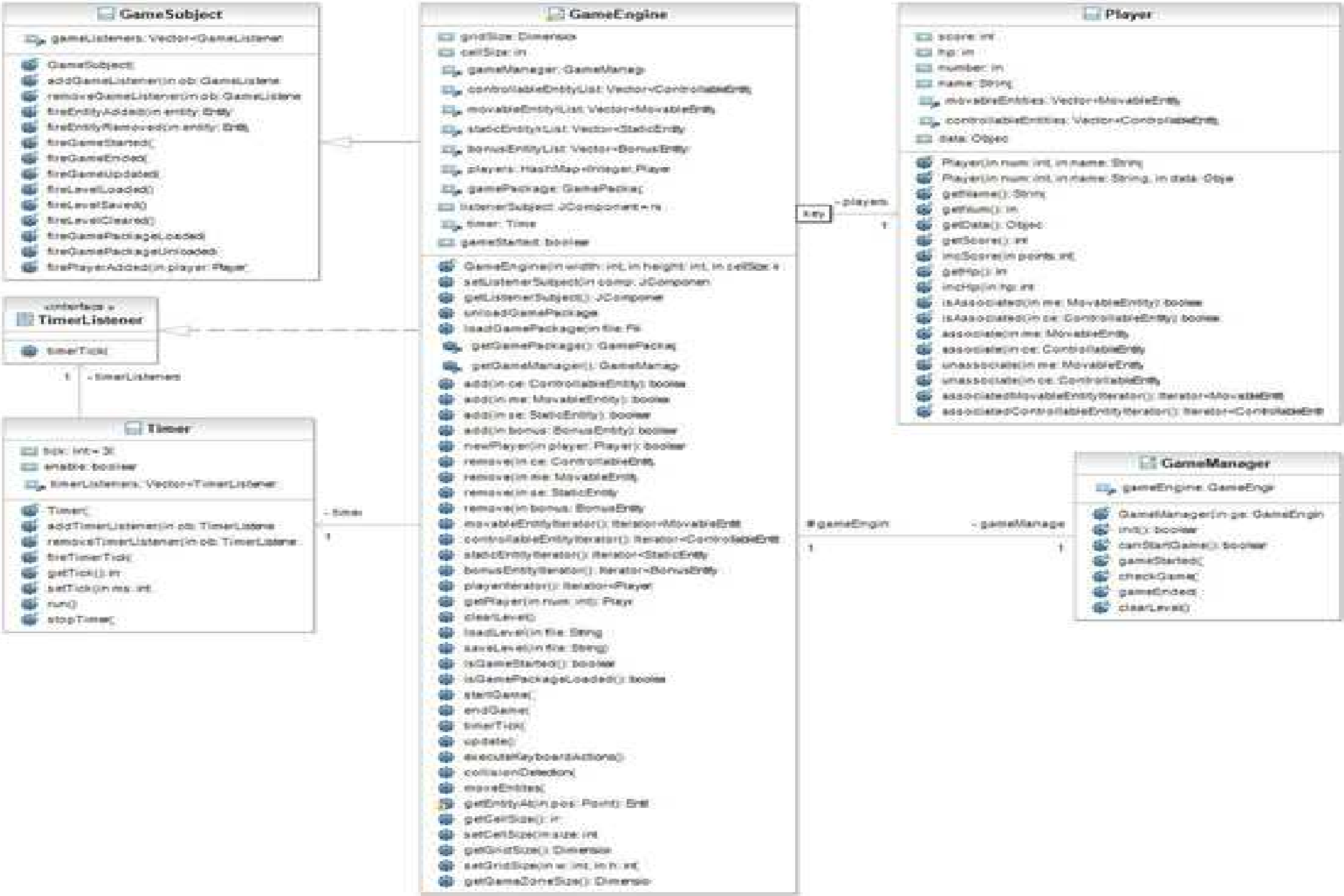


Figure II.A.2 : diagramme de classe du package Model

L'interface GameListener

Cette interface sert à intercepter les événements du moteur de jeu et donc fournit la spécification des méthodes qu'une classe observatrice doit implémenter pour avoir la possibilité de recevoir ces événements.

4) Package model.entities

Ce package regroupe toutes les classes permettant de définir les différents types d'entités manipulables du Framework. Il fait aussi partie de la couche métier du Framework.

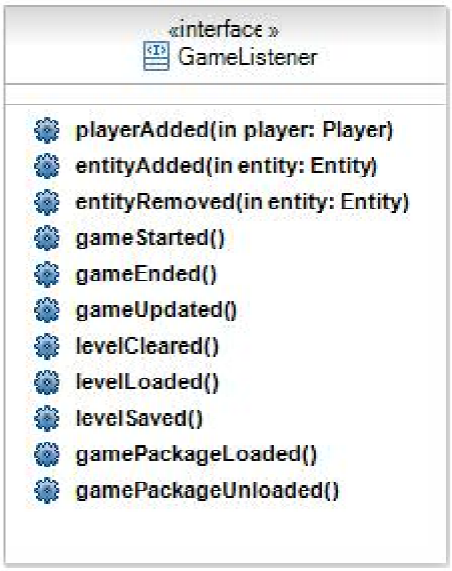


Figure II.A.3 : diagramme de classe du package model.controller

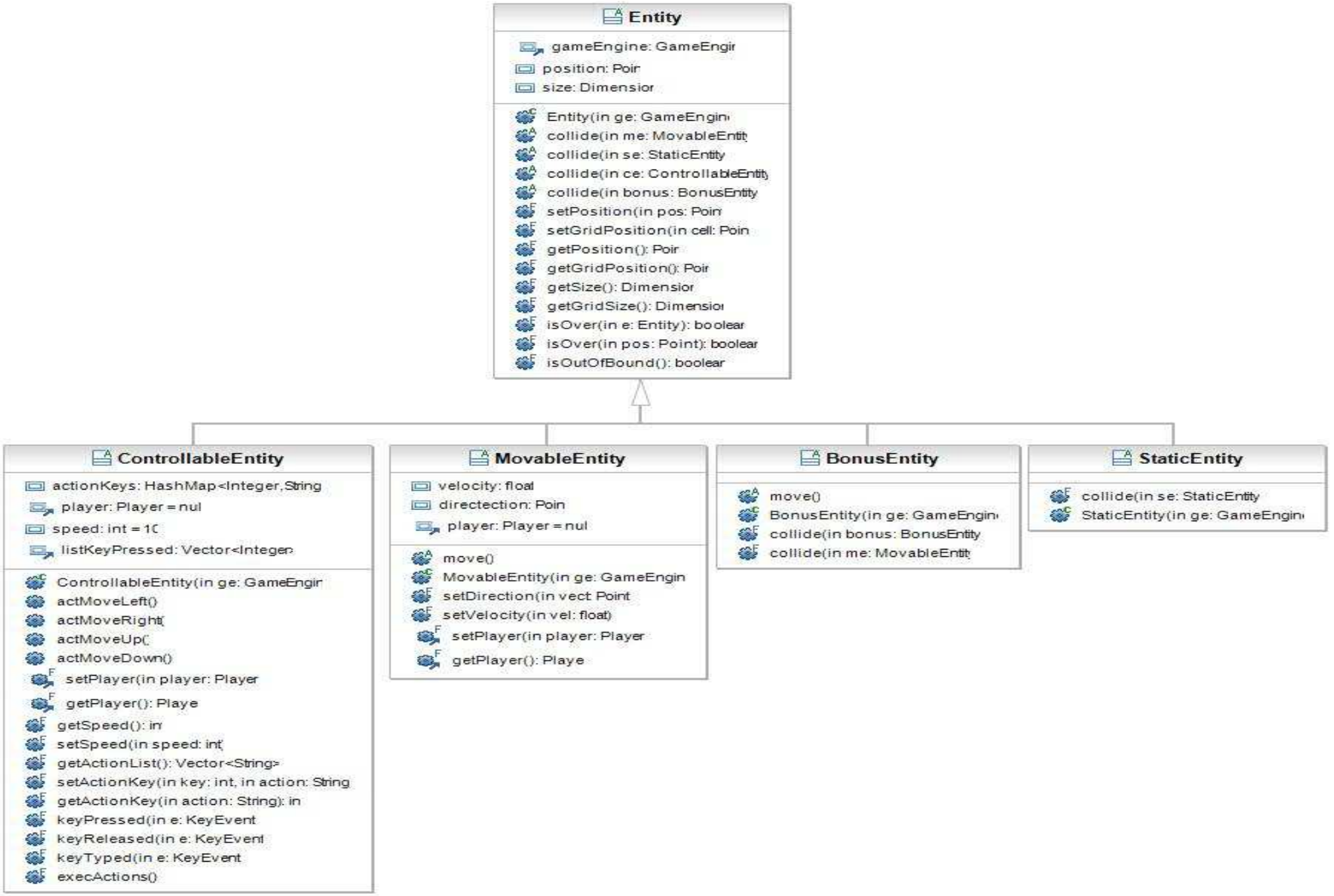


Figure II.A.4 : diagramme de classe du package model.entities

La classe Entity

Cette classe abstraite représente la notion d'une entité abstraite et regroupe toutes les fonctionnalités communes aux quatre sous-types d'entités du Framework.

La classe ControllableEntity

Cette classe abstraite représente les entités contrôlables par l'utilisateur car ce sont les seules entités à pouvoir recevoir les événements du clavier. Elles sont capables d'entrer en collision avec tout autre type d'entité.

La classe MovableEntity

Cette classe abstraite représente les entités ayant la capacité de se déplacer automatiquement à chaque tour d'horloge. Elles peuvent entrer en collision avec tout autre type d'entité.

La classe BonusEntity

Cette classe abstraite représente la notion du bonus à savoir des entités qui peuvent se déplacer mais ne pouvant entrer en collision qu'avec des entités contrôlables et statiques.

La classe StaticEntity

Cette classe abstraite permet de représenter les entités statiques donc immobiles du jeu. Elles peuvent entrer en collision avec tout type d'entités sauf eux-mêmes et reçoivent le contrôle uniquement en cas de collision.

5) Package view

Ce package représente la couche présentation du Framework et regroupe toutes les classes permettant de gérer l'interface graphique du Framework.

La classe ToolBar

Cette classe permet de créer un panneau en haut du Framework représentant la barre d'outils.

La classe PlayerPanel

Cette classe permet de créer un panneau par joueur afin d'afficher son score et son niveau de vie.

La classe GEntity

Cette classe abstraite permet d'associer une représentation visuelle aux différents types d'entités du moteur de jeu. Le nom de la classe graphique d'une entité doit se nommer 'G'+nom de la classe de l'entité.

La classe MainFrame

Cette classe est responsable de la création de la fenêtre principale du Framework en instanciant les différents panneaux d'affichage cités en haut.

6) Package util

Ce package contient deux classes utilitaire «GameEngine» et « JarLoader» servant respectivement pour le chargement des instances des classes graphiques correspondant aux instances des classes métiers et au chargement des fichiers jar des jeux développés à l'aide du Framework.

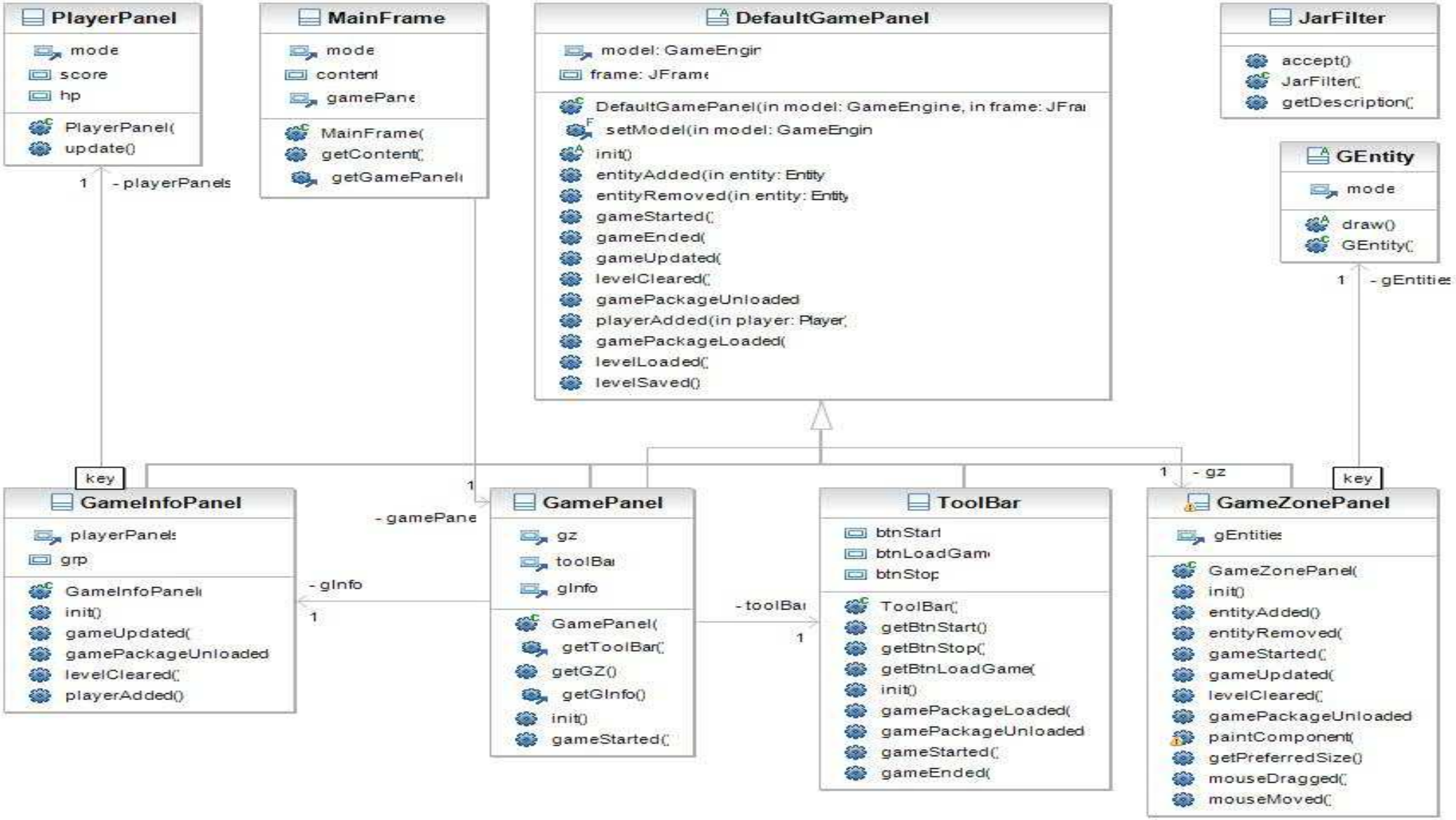


Figure II.A.5 : diagramme de classe du package view

B. Appliquer le Framework : création de notre propre Jeu

a) Présentation du jeu :

Afin de bien comprendre le Framework et détecter ses points faibles et forts, nous avons développé le jeu de Serpent.

Nous l'avons développé en manipulant tous les types d'entités disponibles.

b) Principe du jeu :

Le but du jeu est de manger toutes les balles qui se déplacent aléatoirement dans un temps limité. Dans le début de la partie, les balles sont grandes et de couleur bleu et se déplacent à une faible vitesse. Une fois une balle bleue touchée, elle se divise en deux balles de couleurs jaunes avec une vitesse plus grande que celle des balles bleues.

Finalement, quand on touche une balle jaune elle se divise en deux balles vertes de tailles moindres et ayant une plus grande vitesse. Pour gagner, il faut manger toutes les balles vertes dans un temps limité.

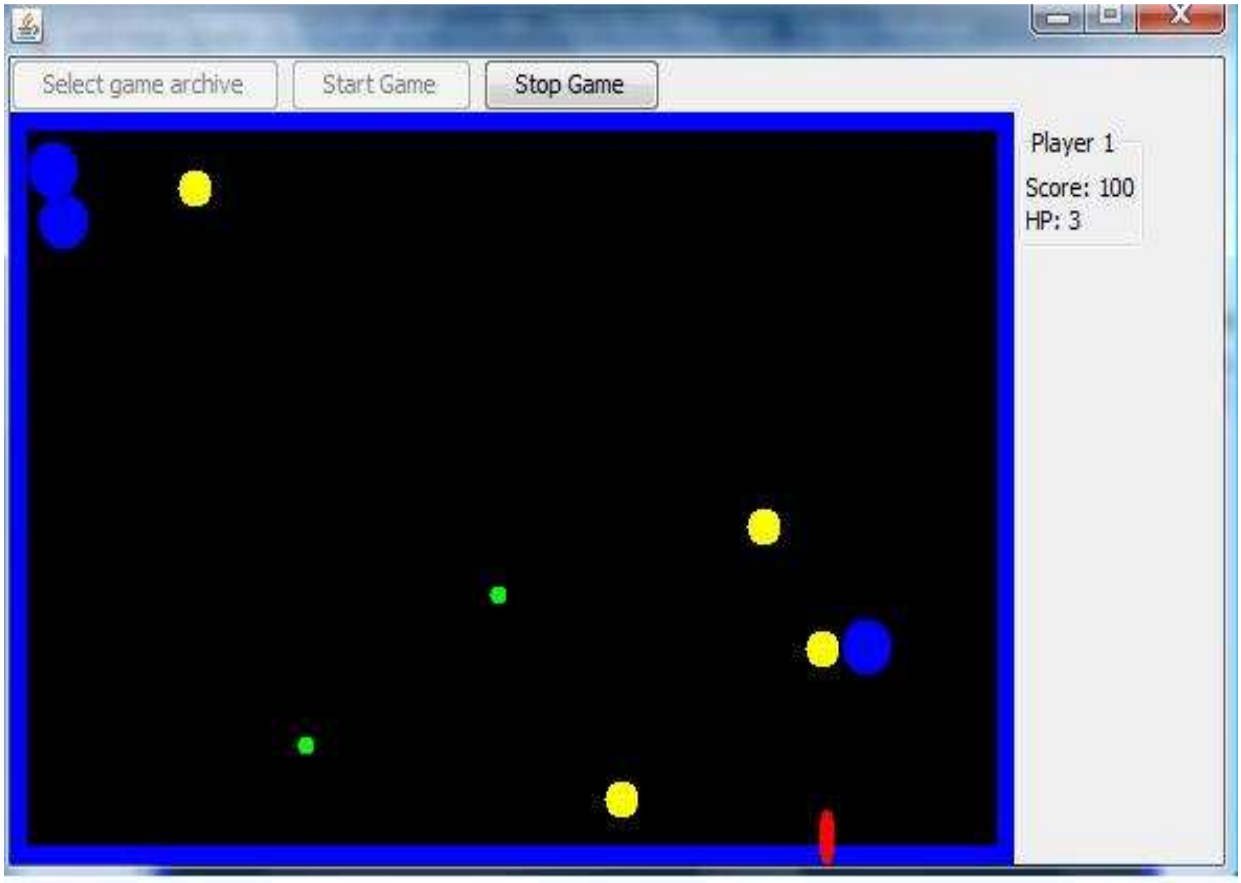


Figure II.B.1 : illustration du jeu Dividing Ball.

c) Développement du jeu :

Le développement du jeu s'est fait en se basant sur les points de paramétrage du Framework, à savoir : les entités et le gestionnaire de jeu.

La zone de jeu utilise trois types que nous avons créés par extension des types d'entités du Framework:

- Static Entity : nous nous sommes servis de ce type d'entités pour délimiter la zone de jeu.
- Movable Entity : ce type nous a servi dans la création des balles bleues, jaunes et vertes.
- Controllable Entity : ce type permet au joueur de contrôler une entité.

d) Constatation des qualités et défauts possibles:

A l'issu de cette étape, nous avons constaté les qualités et défauts possibles:

Qualités :

- Application du pattern MVC ce qui a permis de séparer l'interface graphique du modèle.
- Points d'extensions du Framework explicites : Ces points sont représentés par des classes abstraites à étendre par les utilisateurs du Framework.
- Graphe d'héritage pas trop profond.
- Code assez compréhensible : En effet, la nomenclature est claire et explicite.

Défauts :

Pendant le développement du jeu nous avons constaté plusieurs problèmes. Nous citons les plus importants :

- Difficulté de la compréhension de la classe GameEngine.
- Duplication de code dans plusieurs classes.
- Peu de commentaires dans le code.
- Difficulté pour faire évoluer des parties du Framework par manque d'utilisation de patrons de conception.

C. Analyse quantitative du framework :

Le but de cette partie sera de détecter les composants suspects du logiciel qui doivent être éventuellement refondus. Nous nous reposons sur plusieurs indicateurs afin de réaliser cette tache:

- Mesure des dimensions du framework.
- Mesure des risques
- Mesure de la cohérence

L'analyse quantitative de notre Framework permettra alors d'aller de la critique vers le refactoring en passant par la revue de code, pour s'assurer ainsi des interprétations que nous avons associées aux résultats obtenus. Pour atteindre notre objectif, il a fallu mesurer les propriétés de notre Framework à l'aide des métriques. Nous avons alors fait appel à des outils de mesures dont nous avons choisis les plus riches et les plus précis :

a) Le plugin Eclipse Metrics

Nous avons utilisé le plugin Metrics disponible sur *http://metrics.sourceforge.net* pour calculer les différentes métriques de notre projet.
En plus des mesures classiques, ce plugin open source met en évidence les valeurs critiques des mesures effectuées et a comme particularité d'intégrer un analyseur de dépendance cyclique.
Le deuxième outil que nous avons utilisé est :

b) Le plugin CodePro Analytix

CodePro Analytix est un logiciel Java de la société Instanciation permettant de faire des tests de qualité logicielle. Il s'intègre de façon transparente dans tous les environnements de développement basé sur le Framework Eclipse. CodePro nous permet de créer un code avec une qualité supérieure grâce à des fonctionnalités tel que la vérification automatique de celui-ci ainsi que l'identification des ses failles de sécurité. Il permet également de faire des tests unitaires automatisés et fournit même des recommandations concrètes pour fixer les problèmes détectés.

Nous allons maintenant définir et calculer les différentes métriques à l'aide de nos outils de mesures. Nous allons regrouper chaque indicateur au sein des trois catégories de mesures citées en introduction.

1) *Les mesures des dimensions du Framework*

Les dimensions du logiciel sont les mesures les plus simples à obtenir, même si, comme pour beaucoup de logiciels, les dimensions varient d'une technologie à une autre.

a) *Les dimensions globales*

Ces dimensions s'intéressent au logiciel dans sa globalité et en offre une vision macroscopique. Il faut noter que vu cette portée générale, ils ne sont pas de très bons indicateurs pour détecter les zones du logiciels à inclure dans le refactoring sauf pour des valeurs extrêmes sans rapport avec la nature du Framework.

Nombre de packages, d'interfaces et de classes :

Number of Packages (NOP), Interfaces (NOI) and Classes (NOC)

Définition : Il s'agit du nombre de packages, d'interfaces et de classes du Framework dans l'élément que nous avons sélectionné.

Résultats :

Package	Nombre de Package	Nombre d'Interfaces	Nombre de classes
gameFramework	1	0	1
gameFramework.controller	1	1	0
gameFramework.model	1	1	5
gameFramework.model.entites	1	0	5
gameFramework.util	1	0	2
gameFramework.view	1	0	9
Total	**6**	**2**	**22**

Tableau II.C .1 : Nombre de packages, d'interfaces et de classes.

Interprétation :

Un package est une entité logicielle regroupant un ensemble de classes et/ou interfaces du même thème ce qui permet de raisonner sur l'architecture du logiciel en termes de services. Nous pouvons le considérer comme un indicateur de regroupement des composants du logiciel en entités organisationnelles. Cependant, cette métrique ne donne aucune information sur la pertinence de ce regroupement et donc elle sert plutôt à détecter des valeurs extrêmes (petites et grandes) injustifiables par rapport à la nature du logiciel.

Le nombre de classes dans un package est un bon indicateur du niveau d'importance et de la richesse d'un package ou plutôt d'un domaine fonctionnel d'une application. Ce nombre nous permet de faire la différence entre le nombre de classes implémentées et le nombre de classes modélisées lors de la phase de conception. Ainsi une forte déviation peut indiquer une mauvaise synchronisation entre la conception et la réalisation.

Dans notre Framework le nombre de classes modélisées correspond exactement à celui calculé.

Nombre de lignes de code **Line of code (LOC)**

Définition : Cette métrique représente le nombre total de lignes de code. Les lignes blanches et les commentaires ne sont pas comptabilisés.

Résultats :

Package	Total
gameFramework	21
gameFramework.controller	16
gameFramework.model	540
gameFramework.model.entites	251
gameFramework.util	145
gameFramework.view	393
Total	**1366**
Moyenne	**227**

Tableau II.C .2 : Nombre de lignes de codes.

Interprétation :

Le comptage du nombre de lignes de code est la métrique la plus élémentaire en analyse quantitative. Cependant, ce critère de mesure est loin d'être clair et informatif et donc ne peut pas être pris comme une mesure universelle compte tenu des problèmes qu'il pose quant à la façon de comptabiliser les lignes de code, la puissance d'une ligne de code source qui varie selon les différents langages et le comptage des lignes de code dans des logiciels qui intègrent différent technologies.

Cependant, cette métrique pourrait être un bon indicateur dans le cas de comparaison des méthodes implémentant un algorithme bien connu ou pour la détection d'éventuels problèmes dans une partie bien délimitée du code source. De plus, elle peut servir comme référence pour l'étude de l'évolution du logiciel entre ses différentes versions, néanmoins elle ne peut être prise comme une mesure universelle, à la différence des mesures physiques du système international.

b) *Les dimensions d'une classe ou d'une interface*

Ces mesures, souvent trop synthétiques, sont principalement utilisées comme base pour les mesures de risques. Nous avons pris le soin de regrouper toutes les mesures liées aux différentes classes du Framework dans un seul tableau qui se trouve à la fin de la section de l'analyse quantitative.

Nombre de lignes de code des méthodes Method Lines Of Code (MLOC)

Définition :

Cette métrique représente le nombre total de lignes de codes dans les méthodes d'une classe précise sans compter les lignes blanches et les commentaires.

Résultats:

Package	Nom de la Méthode (Max)	LOC des méthodes(Max)
gameFramework	Main	13
gameFramework.controller	/*interface*/	0
gameFramework.model	collisionDetection	66
gameFramework.model.entites	isOver	28
gameFramework.util	loadJar	34
gameFramework.view	paintComponent	18
Total		159
Moyenne		26.5

Tableau II.C .3 : Nombre de lignes de code des méthodes.

Interprétation: Cette métrique est un bon indicateur pour l'étude de l'évolution du logiciel, elle nous aidera à bien distinguer les méthodes qui ont grandi en taille au cours des différentes versions et qui auraient éventuellement besoin d'une restructuration. Dans notre cas, nous avons calculé quelques méthodes critiques qui prennent la valeur maximale du nombre de lignes de code par package.

Somme de la complexité cyclomatique Weighted Methodes Per Class (WMC)

Définition :

Cette métrique représente la somme de la complexité cyclomatique de McCabe pour l'ensemble des méthodes de la classe.

Résultats:

Voir le tableau des métriques du niveau de classe

Interprétation :

Le somme de la complexité cyclomatique pour la classe gameEngine est de l'ordre de 99 et la moyenne générale de cette même métrique pour l'ensemble des classes du framework est égale à 13. Ce qui justifie la complexité de la classe gameEngine au niveau du Framework.

2) *Les mesures de risques :*

Les mesures de risques regroupent un ensemble d'indicateurs permettant d'avoir une vision globale sur le niveau de risque associé aux différents éléments d'un logiciel. Cependant, elles doivent être utilisées avec précaution, car elles sont sujettes à une forte incertitude.

Le couplage:

Le couplage s'intéresse au nombre de relations qu'entretient une entité, en l'occurrence une classe ou un package, vis-à-vis de l'extérieur.

Il existe deux types de couplages selon la nature des relations d'une entité avec son extérieur.

a) Couplage afférent :

Définition :

Le nombre de classes extérieur à un package qui dépend d'une classe quelconque de ce paquetage.

Résultats :

Package	Total
gameFramework	0
gameFramework.controller	3
gameFramework.model	15
gameFramework.model.entites	9
gameFramework.util	2
gameFramework.view	2
Moyenne	**5.2**

Tableau II.C .4 : couplage afférent.

Interprétation :

Cette métrique est un bon indicateur du niveau de risque lié à la modification d'une entité. Plus une entité a un couplage afférent fort, et plus sa modification risque d'avoir des effets de bord sur les entités utilisant ses services.

Le tableau ci-dessus montre une dépendance élevée des autres packages avec les packages «model» et «model.entities». Ces deux derniers représentent d'ailleurs le corps du framework. Par conséquent une modification de ces packages peut avoir plus d'effets que le reste du framework. Nous pouvons aussi penser à décomposer un package en plusieurs afin de réduire cette dépendance.

b) Couplage efférent

Définition :

Le nombre de classes dans un package qui dépend d'une classe quelconque extérieur à ce package.

Résultats :

Package	Total
gameFramework	1
gameFramework.controller	1
gameFramework.model	3
gameFramework.model.entites	5
gameFramework.util	2
gameFramework.view	8
Moyenne	**3.3**

Tableau II.C .5 : couplage efférent.

Interprétation :

Cette métrique permet d'associer un risque lié aux modifications de l'environnement de l'entité étudiée. Plus une entité a un couplage efférent fort, plus elle dépend de tiers pour remplir son service, ce qui augmente le risque d'effets de bord induits par les modifications de ces entités extérieures.

Nous pouvons le constater clairement dans le cas du package graphique «view» qui possède le couplage efférent le plus élevé comme n'importe quel autre package graphique.

Complexité cyclomatique (McCabe) McCabe Cyclomatic Complexity (VG)

Définition :

La complexité cyclomatique est la mesure de la complexité d'un programme. Elle est conçue de telle manière à être la plus indépendante possible des langages sur lesquels elle est appliquée. Elle facilite ainsi la construction de références pour estimer la complexité d'un logiciel.

Le calcul de la complexité cyclomatique repose sur une représentation du programme à mesurer sous forme d'un graphe dont chaque nœud représente une instruction du programme et chaque arc représente la séquence de ces instructions. Les instructions de type structure de contrôle (boucle, conditions) ont plusieurs arcs tandis que les autres instructions ne peuvent avoir au maximum qu'un arc entrant et un arc sortant.

Une fois le graphe obtenu, le calcul de la complexité cyclomatique CC se fait par la formule CC = E-N+p où E représente le nombre d'arcs du graphe, N: le nombre de nœuds du graphe et p: la somme des points d'entrée et de sortie dans le programme.

Résultats**:** Voir le tableau des métriques du niveau de classe : le tableau récapitulatif.

Interprétation : La complexité cyclomatique est le plus souvent calculée au niveau de chaque méthode d'une classe mais pour avoir une vision globale sur une classe, nous pouvons la calculer en prenant la moyenne et l'écart-type de la complexité des méthodes de la classe.

Plus la complexité cyclomatique est basse, moins le programme est compliqué.

Le tableau ci-dessous (d'après le livre Refactoring des applications JAVA) donne une estimation du niveau de risque associé à la complexité cyclomatique.

Complexité cyclomatique	Niveau de risque
1-10	Programme simple, sans véritable risque
11-20	Programme modérément complexe et risqué
21-50	Programme complexe et hautement risqué
>50	Programme non testable et extrêmement risqué

Tableau II.C .6 : Estimation du niveau de risque associé à la complexité cyclomatique.

Le tableau des métriques au niveau des classes donne une vision de la complexité cyclomatique des classes du framework. Nous pouvons constater qu'aucune classe ne représente de vrais risques de complexité. Par contre au niveau des méthodes, la méthode 'collisionDetection' de la classe GameEngine a une complexité de 25 ce qui constitue une valeur critique pour un programme simple comme notre framework.

Une solution possible à un tel problème est de décomposer la méthode en plusieurs sous-méthodes, de manière à répartir les structures de contrôles sur celles-ci et à réduire la complexité cyclomatique de la méthode originale.

Profondeur du graphe d'héritage — Depth of inheritance tree (DIT)

Définition: Il s'agit de la distance entre la classe observée et son ancêtre le plus éloigné. Dans le cadre du langage Java, cette profondeur est généralement calculée à partir de la super classe java.lang.Object.

Résultats : Voir le tableau récapitulatif.

Interprétation :

Cette métrique est très importante puisqu'elle permet de déterminer pour chaque classe le poids de son héritage à partir de la classe « java.lang.Object » sachant que les graphes très profonds peuvent créer un phénomène de dégénérescence, les descendants n'ayant plus aucun rapport avec leurs ancêtres.

En générale, hormis les classes d'interfaces graphiques qui dépendent de la hiérarchie des composants swing, cette métrique ne dépasse pas la valeur 2, ce que nous constatons d'ailleurs dans le framework.

Profondeur d'imbrication des blocs de code — Nested Block Depth (NBD)

Définition**:** Cette métrique est calculée au sein des méthodes en s'intéressant aux niveaux d'imbrication de leurs structures de contrôles.

Résultats : **V**oir le tableau récapitulatif.

Interprétation : Cette métrique est très utile pour mettre en évidence les méthodes dont le code est rendu illisible à cause d'un nombre important d'imbrications au niveau des structures de contrôles. Pourtant elle n'offre pas la même vision que la complexité cyclomatique puisqu'elle ne s'intéresse qu'à l'imbrication la plus profonde de la méthode et ne donne pas une vue d'ensemble de cette dernière.

En ce qui concerne notre Framework la plupart des méthodes ont des profondeurs d'imbrication de blocs de code raisonnables et ne posent aucun souci particulier à part la méthode collisionDetection qui a une profondeur d'imbrication de 4.

3) Les mesures de cohérence

Les mesures de cohérence analysent les structures internes des éléments ou leurs relations avec le reste du logiciel.

Cohésion d'une classe — Lack of cohesion methods (LCOM)

Définition : Cette métrique sert d'indicateur de niveau d'équilibre ou de déséquilibre entre les attributs d'une classe et les méthodes de la classe qui y accède. Elle est calculée en déterminant le nombre moyen de méthodes qui accèdent à un attribut c'est-à-dire la somme obtenue par l'addition pour chaque attribut A du nombre de méthodes m(A) y accédant divisée par le nombre d'attributs de la classe, puis en soustrayant à ce résultat le nombre total de méthodes et en divisant le tout par 1 moins le nombre total de méthodes.

Résultats **:** Voir le tableau récapitulatif.

Interprétation : Cette métrique doit être la plus proche de zéro représentant ainsi une bonne cohésion au sein de la classe étudiée. En règle générale plus le nombre est petit plus la classe est cohérente alors qu'un nombre proche de 1 indique un manque de cohésion et peut suggérer la décomposition de la classe en plusieurs classes ayant une meilleure cohésion.
La classe «GameEngine» dans notre Framework a une cohésion de 0.845 ce qui suggère une décomposition de celle-ci en plusieurs classes.
Cette métrique peut être trompeuse au cas où on a trop de méthodes getters et setters dans notre classe. Cela augmente le nombre de méthodes accédant aux attributs et par conséquent on obtient une cohésion faible pour la classe étudiée alors que sa sémantique est parfaitement cohérente. Donc une revue de code est nécessaire pour juger de l'exactitude de cette métrique avant de procéder à la décomposition d'une classe.

Indice de spécialisation — Specialization Index (SIX)

Définition : L'indice de spécialisation d'une classe sert d'indicateur pour évaluer la part de réutilisation issue de l'héritage au sein de la classe. Il est calculé en multipliant le nombre de méthodes surchargées (NORM) par la profondeur du graphe d'héritage (DIT) et en divisant le résultat par le nombre total de méthodes (NOM) de la classe observée ce qui donne la formule (NORM * DIT) / NOM.

Résultats : Voir le tableau récapitulatif.

Interprétation : Un indice de spécialisation assez élevé indique souvent un problème de conception au niveau de l'utilisation du mécanisme d'héritage puisque la majorité des méthodes héritées sont surchargées. Cette métrique peut être trompeuse au cas où par exemple une classe non abstraite hérite d'une classe abstraite alors dans ce cas il est tout à fait normal d'avoir un indice fort puisque le nombre de méthodes surchargées est incrémenté dans la classe concrète.

L'instabilité du package — Package instability (RMI)

Définition : Cette métrique est calculée en divisant le nombre de classes du package observé (dépendant de classes extérieurs) (couplage efférent CE) par la somme de ce nombre et du nombre de classes extérieures dépendant des classes du package (couplage afférent CA) ce qui donne la formule CE / (CE + CA).

Résultats :

Package	Total
gameFramework	1
gameFramework.controller	0.25
gameFramework.model	0.17
gameFramework.model.entites	0.36
gameFramework.util	0.5
gameFramework.view	0.8
Moyenne	**0.51**

Tableau II.C .7 : Instabilité du package

Interprétation : La dépendance vis-à-vis d'entités extérieures est une source d'instabilité du fait des effets de bord potentiels issus de la modification de ces dernières. Donc un indice d'instabilité fort indique

une fragilité du package par rapport à son environnement extérieur. D'après le tableau ci-dessus, le package graphique «view» a l'indice d'instabilité le plus élevé puisque comme n'importe quel autre package graphique, il ne fournit aucun service aux autres packages mais repose sur le contenu des packages contenant les services métiers de l'application.

Taux d'abstraction — Abstractness (RMA)

Définition **:** Le taux d'abstraction d'un package est calculé en divisant le nombre de classes abstraites et d'interfaces d'un package par son nombre total de classes.

Résultats :

Package	Taux (%)
gameFramework	0 %
gameFramework.controller	100 %
gameFramework.model	33.3 %
gameFramework.model.entites	100 %
gameFramework.util	0 %
gameFramework.view	22.2 %
Total	**42.6 %**

Tableau II.C .8 : Taux d'abstraction

Interprétation : Cette métrique donne le pourcentage de classes abstraites par package, ce qui peut être vu comme un indicateur du niveau de réutilisabilité et d'extensibilité des services de package.

Puisque notre framework est destiné à créer de nouveaux jeux , il n'y a rien d'étonnant de voir des packages avec des taux d'abstraction de 100%, ce qui offre une meilleure extensibilité.

L'indice de spécialisation d'une classe permet d'évaluer la part de réutilisation issue de l'héritage au sein de la classe observée. Un indice fort indique souvent un problème de conception et une mauvaise utilisation du mécanisme d'héritage puisque la majorité de méthodes héritées sont surchargées. Un indice de spécialisation très proche ou égale à zéro est souvent un indicateur d'une bonne utilisation de l'héritage. La plupart des indices de spécialisation sont de zéro ce qui montre une bonne pratique des mécanismes d'héritage.

Paquetage	Nom	Type	LOC	NOF	NOM	NSM	NSF	NORM	NSC	DIT	VG	LCOM	SIX	WMC
gameFramework	Main	CC	21	0	0	1	1	0	0	1	5	0	0	5
Controller	gameListener	I	16	0	0	0	0	0	0	0	0	0	0	0
Model	gameEngine	CC	343	12	40	0	0	0	0	2	2.475	0.845	0	99
	gameManager	AC	13	1	7	0	0	0	3	1	1	0	0	7
	gameSubject	CC	62	1	14	0	0	0	1	1	1.143	0	0	16
	Player	CC	73	7	17	0	0	0	0	1	1.118	0.821	0	19
	Timer	CC	45	3	8	0	0	0	0	2	1.5	0.714	0	12
	TimerListener	I	4	0	0	0	0	0	0	0	0	0	0	0
Model.entities	BonusEntity	AC	14	0	4	0	0	0	1	2	1	0	0	4
	ControllableEntity	AC	117	4	16	0	0	0	3	2	1.625	0.769	0	26
	Entity	AC	83	3	14	0	0	0	4	1	1.643	0.458	0	23
	MovableEntity	AC	27	3	6	0	0	0	4	2	1	0.75	0	6
	StaticEntity	AC	10	0	2	0	0	0	2	2	1	0	0	2
util	gamePackage	CC	79	6	3	0	0	0	0	1	5.333	0.833	0	16
	JarLoader	CC	66	0	0	2	0	0	0	1	5.5	0	0	11
view	DefaultGamePanel	AC	43	2	14	0	0	0	4	5	1	0.5	0	14
	GameInfoPanel	CC	54	2	6	0	0	4	0	6	1 .333	0.25	4	8
	GamePanel	CC	42	3	6	0	0	1	0	6	1.5	1	1	9
	GameZonePanel	CC	80	1	12	0	0	7	0	6	1.333	0	3.5	16

	GEntity	AC	10	1	2	0	0	0	11	1	1	0	0	2
	MainFrame	CC	32	3	3	0	0	0	0	6	1.667	0.833	0	5
	PlayerPanel	CC	28	3	2	0	0	0	0	5	1	0	0	2
	ToolBar	CC	104	3	11	0	0	4	0	4	1.273	0.5	1.333	14

Tableau II.C .9 : Tableau récapitulatif

Avec les abréviations:

Lines of Code (LOC):	**Nombre de lignes de code** Le nombre total de lignes de code
Number Of Attributes (NOF)	**Nombre d'attributs :** Le nombre d'attributs dans l'entité à étudier
Number of Methods (NOM)	**Nombre de méthodes :** Le nombre de méthodes dans l'entité à étudier
Number of Static Methods (NSM)	**Nombre de méthodes statique :** Le nombre de méthodes statiques dans l'entité à étudier
Number of Static Attributs (NSF)	**Nombre d'attributs statique :** Le nombre d'attributs statiques dans l'entité à étudier
Number of Overridden Methods (NORM)	**Nombre de méthodes surchargées :** Le nombre de méthodes redéfinies dans l'entité à étudier
Number of children (NSC)	**Nombre de sous-classes:** le nombre de sous-classes directes d'une classe.
Depth of Inheritance Tree (DIT):	**Dépendance dans l'arbre d'héritage :** Distance jusqu'à la classe Object dans la hiérarchie d'héritage.
cCabe Cyclomatic Complexity (VG)	**La complexité cyclomatique d'une méthode :** le nombre de chemins possibles à l'intérieur d'une méthode
Lack of Cohesion of Methods (LCOM)	**Une mesure de la cohésion d'une classe**
Specialization Index (SIX)	**Moyenne de l'index de spécialisation.**
Weighted Methods per Class (WMC)	La somme de la complexité cyclomatique de McCabe pour toutes les méthodes de la classe.

4) Synthèse de l'analyse quantitative

A l'issu de l'analyse quantitative et du calcul des métriques, nous avons détecté les problèmes suivants :

- La classe GameEngine détient beaucoup de responsabilités.
- La méthode de détection de collision est trop complexe et la valeur de sa complexité cyclomatique est critique.
- Mauvaise répartition des responsabilités
- Gestion des exceptions.
- Classe GamePackage contenant un constructeur avec un grand nombre de paramètres.
- Duplication de code dans la méthode GetEntityAt.
- Manque de souplesse au niveau de la conception dans quelques parties du Framework.
- Aucune gestion des éventuelles exceptions.

D- Analyse qualitative du framework :

L'analyse qualitative du Framework nous permet de vérifier la qualité du code du Framework en le comparant avec les bonnes règles et conventions de programmation. A l'inverse de l'analyse quantitative qui porte sur la sémantique en étudiant la hiérarchie et la structuration des logiciels, l'analyse qualitative permet d'étudier de plus près la qualité syntaxique du code source d'un logiciel en le testant et en le comparant avec les normes pré établies de la programmation.

Dans le cadre de notre projet, nous avons utilisé l'outil « CodePro Analytix » pour détecter les violations des normes de programmation dans le Framework. Cet outil nous permet de générer et de regrouper les violations constatées dans différents groupes selon leurs niveaux de sévérités, leurs auteurs, les ressources à savoir les classes où elles se sont produites ainsi que leurs catégories.

Le tableau ci-dessous donne un résumé du nombre de violations constatées dans le cadre du Framework regroupé par niveau de sévérité.

Niveau de sévérité	Nombre de violation
Haut	7
Moyenne	608
Bas	63

Tableau II.D.1 : Nombre de violation par sévérité

Le tableau suivant donne un autre résumé par règle de violation. Une explication de chacune de ces règles, est donnée en annexe du rapport.

Règle de violation	Nombre de violation
Abstract Specialization	1
Always Override toString	7
Boolean Method Naming Convention	6
Caught Exceptions	1
Close Where Created	1
Constants in Comparison	17
Dangling Else	8
Debugging Code	27
Declare As Interface	27
Declare Default Constructors	16
Define Load Factor	4
Disallow Sleep Inside While	1
Empty Catch Clause	1
Empty Method	17
Explicit "this" Usage	190
Field Javadoc Conventions	7
File Comment	23
Import Order	15
Local Declarations	28
Method Invocation in Loop Condition	6
Method Javadoc Conventions	55
Missing Block	44
No Spelling Dictionary	1
Non-protected Constructor in	8
Numeric Literals	13
Obsolete Modifier Usage	12
Overloaded Methods	14
Package Naming Convention	1
Package Prefix Naming Convention	6
Prefer Interfaces To Reflection	1
Questionable Assignment	1
Questionable Name	2
Reusable Immutables	2
String Concatenation	4
String Literals	18
Too Many Violations	4
Type Javadoc Conventions	25
Use Compound Assignment	10
Use equals() Rather Than ==	1
Variable Declared Within a Loop	10
Variable Should Be Final	43

Tableau II.D.2 : Nombre de violation par règle

Synthèse de l'analyse qualitative

L'analyse qualitative du Framework nous a permis de mettre en évidence et de corriger les points du Framework qui violent les bonnes règles syntaxiques et parfois sémantiques de la programmation. Bien que ces violations n'aient aucun impact direct sur l'exécution du Framework, les corriger permettra d'avoir un code de bonne qualité respectant ainsi toutes les normes de la programmation.

Parmi les résultats obtenus, on peut distinguer dans un premier temps qu'il y a un vrai manque de documentation au niveau du Framework car il y a en total 87 violations qui se portent sur la documentation Javadoc des champs, des méthodes et des classes du Framework. Soit des parties du Framework n'ont pas été commenté du tout ou alors, la documentation existante ne respecte pas le standard de la documentation Javadoc.

Nous pouvons aussi constater que la violation la plus commise dans le cadre de ce Framework est l'usage explicite du mot clé 'this' là ou on peut les supprimer. Ca représente 190 violations c'est-à-dire 28% de toutes les violations du Framework.

En dernier point de synthèse, nous pouvons citer le non respect des conventions de nommage dans le cadre du nommage des packages et certains champs et variables.

En conclusion, nous listons ci-dessous les points les plus importants qui nous permettront d'améliorer la qualité du Framework :

- Compléter la documentation existante.
- Renommer les packages, les champs et les méthodes de classes ne respectant pas les conventions de nommage.
- Supprimer tous les codes de débogages

Section III. Les améliorations proposées

A- Refactoring de la méthode CollisionDetection

Le calcul métrique que nous avons effectué précédemment nous a permis de faire quelques constatations : Il s'agit d'une complexité cyclomatique conséquente de la méthode « CollisionDetection ». Nous avons détecté aussi que la Profondeur d'imbrication des blocs de code a pour valeur 4, ce qui montre que cette méthode est complexe. C'est pour cela que nous avons procédé à son refactoring, qui nous a d'ailleurs poussé à faire des modifications au niveau de la classe « GameEngine » dont nous expliquerons le procédé plus tard.

La solution que nous avons adoptée fut construite en deux temps. Dans un premier temps, nous avons extrait juste la méthode « DetectionCollision » hors de la classe GameEngine vers la classe « Collision ». Cette extraction nous a permis de réduire la complexité cyclomatique dans la méthode CollisionDetection qui fait juste une délégation à « Collision ».

La deuxième phase de la solution était d'améliorer la méthode « CollisionDetection » de la classe « Collision ». En effet, la première étape n'a pas réduit la complexité cyclomatique dans la classe « Collision ». L'amélioration que nous avons adoptée fut l'utilisation de la généricité paramétrique.

L'utilisation de celle-ci dans la classe « Collision » nous a conduit à créer de nouvelles classes ayant des responsabilités bien définies dont nous expliquons les rôles :

- EntityVector : Elle permet de représenter la notion d'ensemble d'entités. Cette classe générique est un sous type de Vector. Elle reçoit les événements du moteur de jeu. La gestion des entités, à savoir, l'ajout et la suppression, est gérée par cette classe.
- NonStaticEntityVector : Nous avons crée cette classe qui spécialise la classe « EntityVector » pour pouvoir gérer les MovableEntity et les ControllableEntity. En effet, ces deux types d'entités contiennent des attributs Player. Donc il faut gérer l'ensemble de joueurs de ces entités.
- PlayerCollection : Cette classe spécialise la classe HashMap paramétrée par des entiers et des joueurs. Nous l'avons créé pour que la classe GameEngine ne se préoccupe pas de la gestion des joueurs.
- Collision : Cette classe fut créée suite à l'application du pattern « délégation ». C'est une classe générique instanciable par des sous classes d'Entity.

A l'issu de cette étape, nous avons réduit la complexité cyclomatique à 1 dans la méthode « CollisionDetection » de la classe « GameEngine ».

Voici un exemple où la classe « GameEngine » délègue la détection de collision à la classe «Collision»

```
CD = new Collision <MovableEntity, ControllableEntity>
                              (MovableEntityList, controllableEntityList);
CD.collideOther ();
```

Comme vous pouvez le constater, la méthode CollisionDetection ne fait que passer des ensembles d'entités et c'est la classe Collision qui se charge du calcul.

Nous allons expliquer, dans le paragraphe suivant, les classes que nous avons dû créer pour pouvoir faire le refactoring de la méthode CollisionDetection. Ces classes nous ont permis aussi de faire le refactoring de la classe « GameEngine ».

B- Découpage et refactoring de la classe GameEngine

Se reposant sur les patterns de restructuration lus dans le livre, il nous a paru judicieux de découper la classe « GameEngine ». En effet, cette classe se qualifie en une « god Class ». Ce terme est adopté du fait que la classe « GameEngine » détient beaucoup de responsabilités.

Le schéma UML de la classe « GameEngine » avant le refactoring était le suivant :

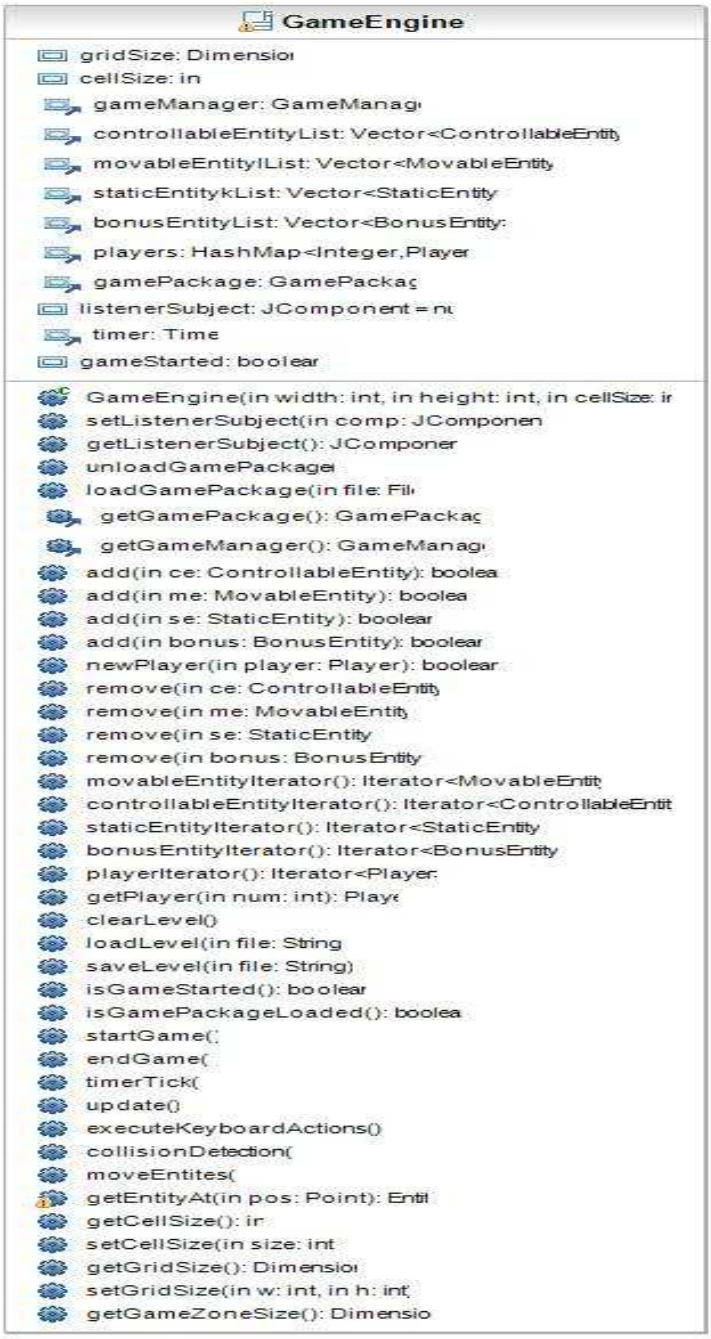
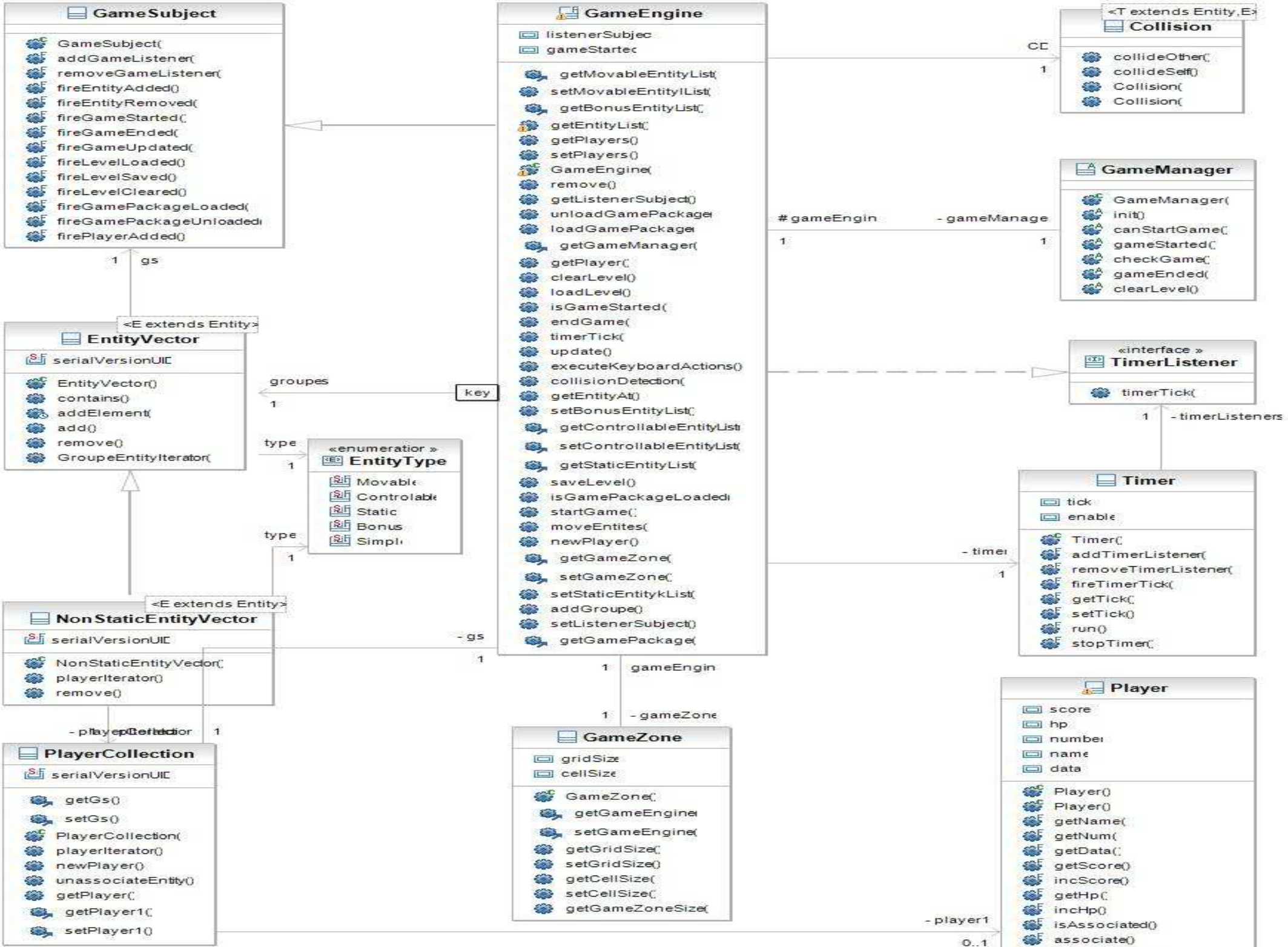


Figure III.B.1 : Diagramme UML de la classe GameEngine avant le refactoring.

Extraction de la partie Zone de jeu

Pour surmonter ce problème, nous avons créé la classe « GameZone ». Cette classe aura pour responsabilités de contenir toutes les informations concernant la zone de jeu, ainsi que toutes les méthodes de calcul des cordonnées de cette dernière. L'adoption de cette solution a réduit significativement *l'effet de navigation du code* décrit dans le livre et nous a permis d'éviter à la classe GameEngine et aux autres classes d'implanter les comportements de calcul des coordonnées.

A l'issu de l'étape de la réécriture de la méthode de détection des collisions, l'extraction des données ainsi que les comportements concernant la zone de jeu, nous avons obtenu le diagramme de classe suivant : Figure III.B.2 : Diagramme UML de classe du Package Model après refactoring


GameSubject
GameSubject(
addGameListener(
removeGameListener(
fireEntityAdded()
fireEntityRemoved(
fireGameStarted(
fireGameEnded(
fireGameUpdated(
fireLevelLoaded()
fireLevelSaved()
fireLevelCleared()
fireGamePackageLoaded(
fireGamePackageUnloaded
firePlayerAdded()
1
gs
GameEngine
listenerSubjec
gameStartec
getMovableEntityList(
setMovableEntityList(
getBonusEntityList(
getEntityList(
getPlayers()
setPlayers()
GameEngine(
remove()
getListenerSubject()
unloadGamePackage
loadGamePackage
getGameManager(
getPlayer(
clearLevel()
loadLevel()
isGameStarted(
endGame(
timerTick(
update()
executeKeyboardActions()
collisionDetection(
getEntityAt()
setBonusEntityList(
getControllableEntityList
setControllableEntityList(
getStaticEntityList(
saveLevel()
isGamePackageLoaded
startGame(
moveEntites(
newPlayer()
getGameZone(
setGameZone(
setStaticEntitykList(
addGroupe()
setListenerSubject()
getGamePackage(
CC
1
<T extends Entity,E>
Collision
collideOther(
collideSelf()
Collision(
Collision(
gameEngin
1
- gameManage
1
GameManager
GameManager(
init()
canStartGame(
gameStarted(
checkGame(
gameEnded(
clearLevel()
<E extends Entity>
EntityVector
serialVersionUID
EntityVector()
contains()
addElement(
add()
remove()
GroupeEntityIterator(
groupes
1
key
type
1
«enumeration »
EntityType
Movable
Controlable
Static
Bonus
Simpl
type
1
«interface »
TimerListener
timerTick(
1
- timerListeners
Timer
tick
enable
Timer(
addTimerListener(
removeTimerListener(
fireTimerTick(
getTick(
setTick()
run()
stopTimer(
- timer
1
<E extends Entity>
NonStaticEntityVector
serialVersionUID
NonStaticEntityVector(
playerIterator()
remove()
- gs
1
1
gameEngin
1
- gameZone
- player1
1
PlayerCollection
serialVersionUID
getGs()
setGs()
PlayerCollection(
playerIterator()
newPlayer()
unassociateEntity()
getPlayer(
getPlayer1(
setPlayer1()
GameZone
gridSize
cellSize
GameZone(
getGameEngine
setGameEngine(
getGridSize(
setGridSize()
getCellSize(
setCellSize(
getGameZoneSize(
Player
score
hp
number
name
data
Player()
Player()
getName(
getNum(
getData(
getScore()
incScore()
getHp(
incHp()
isAssociated()
associate()
0..1

C- Réécriture de la méthode GetEntityAt :

Pendant notre lecture manuelle du code, nous avons détecté l'annotation TODO de la javadoc sur la méthode « getEntityAt » de la classe GameEngine demandant d'optimiser l'algorithme de recherche.

Comme les patrons de restructuration le conseillent, il ne faut pas se fier à la documentation du système étudié. C'est pour cela qu'on a revu complètement le code de cette méthode et nous avons détecté une duplication de code à cause des différents types statiques des entités.

Voici le code de cette méthode avant de procéder au refactoring :

```
public Entity getEntityAt(Point pos) {
      Iterator it;
      Entity e = null;

      it = this.movableEntityIterator();

      while (it.hasNext()) {
            e = (Entity) it.next();
            if (e.isOver(pos))
                  return e;
      }

      it = this.staticEntityIterator();

      while (it.hasNext()) {
                  e = (Entity) it.next();
                  if (e.isOver(pos))
                        return e;
      }
```

```
		it = this.controllableEntityIterator();

		while (it.hasNext()) {
				e = (Entity) it.next();
				if (e.isOver(pos))
					return e;
		}

		it = this.bonusEntityIterator();

		while (it.hasNext()) {
				e = (Entity) it.next();
				if (e.isOver(pos))
					return e;
		}

		return null;
	}
```

Pour surpasser le problème, nous avons eu recours à la généricité paramétrique : nous avons rajouté un attribut de type HashMap nommé « Groupes » qui contient nos groupes d'entités. On parcourt alors cette collection et par polymorphisme on fait appel aux bonnes méthodes.

La méthode GetEntityAt devient (après refactoring):

```
public Entity getEntityAt(Point pos){

		Iterator it;

		Entity e = null;

	for(GroupeEntity groupe : groupes.values()){

		it = groupe.GroupeEntityIterator();

		while (it.hasNext()) {

			e = (Entity) it.next();

			if (e.isOver(pos))

			return e;

	}

	}

	return null;

	}
```

D- Refactoring de la classe GamePackage :

Au cours de notre analyse de la classe « GamePackage », nous avons détecté que le constructeur de cette classe prenait sept paramètres. Quatre de ces paramètres sont des vecteurs qui représentent les entités du jeu chargé dans le moteur de jeu.

Pour diminuer le nombre de paramètres, nous avons regroupé ces vecteurs dans une seule collection. En procédant de la sorte, le nombre de paramètres a diminué de trois.

Déplacement de UnloadGamePackage et LoadGamePackage

Etape 1

Le refactoring de la classe GamePackage consiste à rajouter les services de chargement et de déchargement des jeux contenus dans les fichiers JARs dans le moteur de jeu. En effet, ces deux services étaient regroupés dans la classe GameEngine.

Voici ci-dessous le diagramme de classe « GamePackage » avant le refactoring.

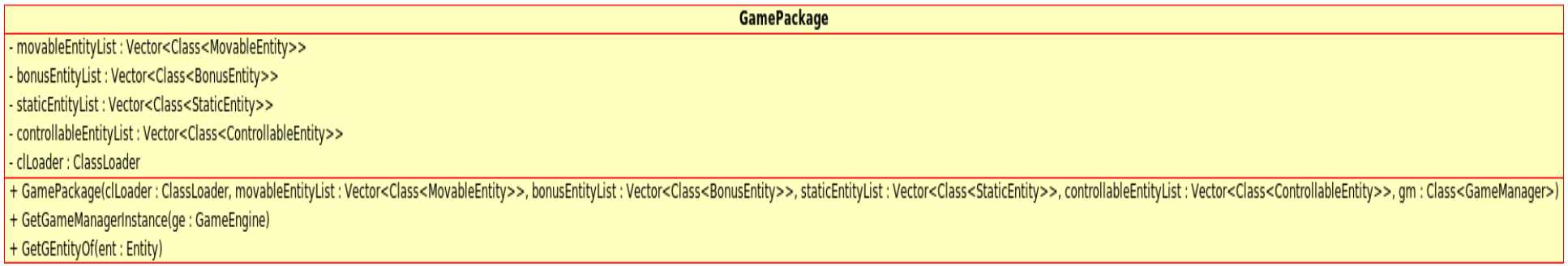


Figure III.D .1 : Diagramme de la classe GamePackage avant Refactoring

La solution consistait à passer à rajouter en paramètres des deux méthodes une référence de GameEngine. Le moteur de jeu passé en paramètres obtiendra tous les entités et les informations concernant le jeu et les joueurs dans en faisant appel à une instance de « GamePackage ».

Voici le listening des deux methodes stockées dans la classe GameEngine.

```
public void unloadGamePackage() {
            if ((this.gameManager == null) || (this.isGameStarted()))
                  return;

            this.controllableEntityList.clear();
            this.movableEntityList.clear();
            this.bonusEntityList.clear();
            this.staticEntityList.clear();
            this.playerCollection.clear();

            this.gameManager = null;
            this.gamePackage = null;

            this.fireGamePackageUnloaded();
      }
```

Et le code de LoadGamePackage

```
public void loadGamePackage(File file) {
            if (this.isGameStarted())
                  return;

            try {
                  this.unloadGamePackage();
                  this.gamePackage =
JarLoader.loadJar(file.getAbsolutePath());

                  // récupère une instance du GameManager
                  this.gameManager =
this.gamePackage.getGameManagerInstance(this);
                  this.gameManager.init();
            }
            catch (FileNotFoundException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }

            this.fireGamePackageLoaded();
      }
```

Après l'application de l'amélioration, le code des deux opérations dans « GameEngine » devient

```
      public void  LoadGamePackage (File JarFile){

            GP.LoadGamePackage (JarFile,  this) ;
```

```
}
        public void UnLoadGamePackage (){
                this.GP.UnLoadGamePackage (this);
}
```

À la fin de cette étape, le diagramme de la classe « GamePackage » devient

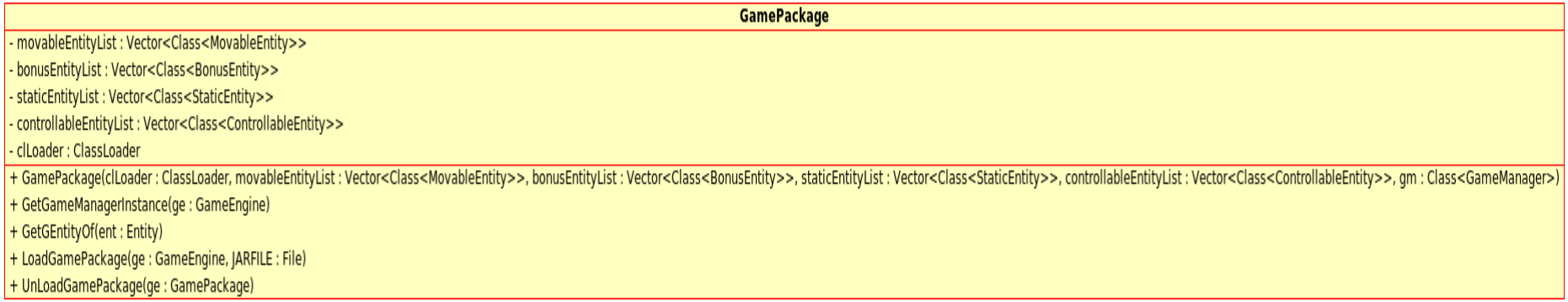


Figure III.D .2 : Diagramme de la classe GamePackage après Refactoring

Cette amélioration a pour conséquences la réduction de la taille et de la complexité de la classe « GameEngine », ainsi qu'une répartition meilleure des taches dans le Framework.

Etape 2:

La deuxième étape de l'amélioration de la classe GamePackage, a été de réduire le nombre de paramètres du constructeur. En effet, le constructeur de la classe GamePackage prend sept paramètres.

```
public GamePackage(ClassLoader clLoader, Vector<Class<MovableEntity>> movableEntitylList,
                   Vector<Class<BonusEntity>> bonusEntityList, Vector<Class<StaticEntity>>
staticEntityList,
                   Vector<Class<ControllableEntity>> controllableEntityList, Class<GameManager> gm)
```

La solution que nous avons adoptée a été de regrouper les vecteurs de classes dans une meta-collection. Cette solution a eu pour effet la réduction du nombre de paramètres à quatre.

E- Amélioration de la classe Entity

Dans notre procédé d'amélioration du Framework, nous avons détecté un manque de souplesse dans la conception de la classe « Entity ». Ce manque se traduit par la surcharge de la méthode collide et par la non-utilisation du patron de conception « Stratégie ».

Afin de surmonter ce problème, et rajouter plus de souplesse au Design du Framework nous avons dû appliquer le Patron stratégie sur la méthode Collide et Move.

a) Refactoring des méthodes Collide

Utilisation du patron de conception Stratégie

Afin de factoriser les quatre méthodes de collisions collide () dans la classe Entity qui diffèrent seulement selon leurs paramètre d'entrée, nous avons pensé à les extraire de la classe Entity et à créer une interface collidable comprenant ces quatre méthodes. Cela nous a permis de ne garder dans la classe Entity qu'une seule méthode collide () paramétrée par la super classe des quatre types d'entités à savoir la classe Entity.

Ainsi cette méthode transmettra chaque demande de collision aux classes de jeux implémentant l'interface collidable. En plus la classe DefaultCollidable implémente toutes les méthodes de l'interface collidable à vide permettant de factoriser le comportement de certaines entités de jeux qui n'entrent pas en collision avec d'autres types d'entités.

Figure III.E.1: la classe Entity avant factorisation comportant quatre méthodes collide ()

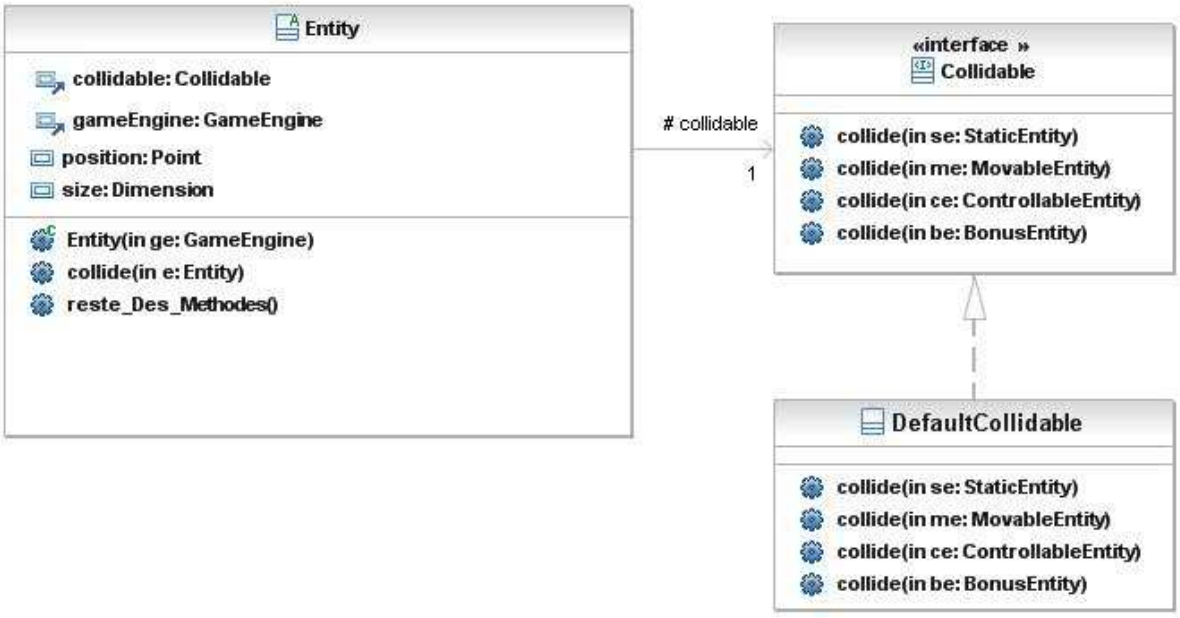


Figure III.E.2 : la classe Entity après factorisation comportant une seule méthode collide ()

Cette méthode d'application s'inspire du patron de conception stratégie qui extrait d'une classe tous ses comportements algorithmiques et les déplacent en dehors de la classe permettant l'extensibilité des comportements d'une classe

b) Refactoring des méthodes Move

Le pattern Stratégie peut être aussi utilisé pour factoriser le code lié aux mouvements des différentes entités pouvant se déplacer. Pour cela, nous pouvons créer une interface ne comportant que la méthode move () et chaque type d'entité pouvant se déplacer va implémenter cette interface.
Cela nous permet d'extraire les algorithmes de mouvement des classes d'entités en les regroupant dans une famille de classes qui implémente différents types de mouvements.
Ainsi les différentes classes de mouvement peuvent être réutilisées dans différents jeux et donc nous pouvons étendre et créer facilement de nouveaux types de mouvement.
Cela nous permet aussi d'avoir simultanément plusieurs comportements de mouvement pour la même entité mobile que nous regroupons dans une collection et nous pouvons changer dynamiquement de comportement lors de l'exécution.

GESTION DES EXCEPTIONS :

Lors de notre revue de code, et lors de l'analyse qualitative, nous avons détecté que les exceptions ne sont en aucun cas traitées dans le Framework. Cela a pour conséquence une certaine faiblesse et vulnérabilité envers les pannes et les éventuelles erreurs au moment de l'exécution.

F. CONSTATATION DES AMELIORATIONS :

Nous allons réappliquer les métriques afin de s'assurer des améliorations apportées.

Nombre de Ligne de code :

Package	Avant amélioration	Après amélioration
GameFramework	21	21
Controller	16	16
Model	540	624
Entities	251	377
View	393	395
Util	145	149
Total	1366	1582

Tableau III.E.1 : Nombre de lignes de code après améliorations

Interprétation :

Après la restructuration du Framework, nous avons remarqué une augmentation du nombre totale de lignes de code. Cette augmentation est causée par l'ajout des classes après l'application du patron « delegate ».

Afferent coupling

Package	Avant amélioration	Après amélioration
GameFramework	0	0
Controller	0	0
Model	14	10
Entities	13	8
View	2	2
Util	9	9

Tableau III.E.2 : Couplage afférent après améliorations

Interprétation :

Une baisse remarquable du couplage afférent dans le package Model a été constatée après l'application des patrons de conception. Cela se traduit par un découplage meilleur des composants du Framework.

Number Of classes

Package	Avant amélioration	Après amélioration
GameFramework	1	1
Controller	0	0
Model	5	10
Entities	5	7
View	2	9
Util	9	2

Tableau III.E.3 : Nombre de classes après améliorations

Interprétation :

Nous constatons une augmentation du nombre total des classes du Framework. Cette augmentation est causée par l'application des patrons de restructuration, spécialement le patron «découpage de la god class » et aussi les patrons de conception comme delegate et stratégie.

Instability

Package	Avant amélioration	Après amélioration
GameFramework	1	1
Controller	0.25	0.25
Model	0.17	0.193
Entities	0.36	0.147
View	0.8	0.5
Util	0.615	0.615

Tableau III.E.4 : Instabilité du package après améliorations

Interprétation :

Nous constatons une meilleure stabilité dans les différents packages du Framework. Cette instabilité permettra une évolution et un paramétrage aisé et simple du Framework.

Nested Block Depth:

Comme nous l'avons signalé dans la partie analyse quantitative, cette métrique est déterminée au niveau des méthodes. Elle détermine le niveau d'imbrication des structures de contrôles au sein d'une méthode.

Après les modifications que nous avons appliquées, nous avons recalculé cette métrique sur les méthodes que nous avons changées.

Nom de la méthode	Valeur avant modification	Valeur Après modification
Collision Detection	4	1
GetEntityAt	2	1
Collide	2	2

Tableau III.E.6 : Profondeur d'imbrication des blocs de code après améliorations

Comme vous pouvez le constater, le problème majeur était la méthode « CollisionDetection » qui avait une valeur de 4 pour l'imbrication de code. Maintenant, la complexité de la méthode est moindre et les valeurs de cette métrique pour les autres méthodes que nous avons modifiées sont désormais raisonnables.

Lack of Cohesion of Methods: LOCM

Cette métrique sert d'indicateur du niveau d'équilibre ou de déséquilibre entre les attributs d'une classe et les méthodes de la classe qui y accède.

Après la modification du Framework, nous avons recalculé les valeurs LCOM sur tous les packages, et nous avons obtenu les résultats suivants :

Nom du package	Valeur avant modification	Valeur après modification
GameFramework	0	0
Model	0.476	0.5
Model.Entities	0.396	0.411
View	0.398	0.398
Util	0.417	0.417
Controller	0	0

Figure III.E.7 : Cohésion d'une classe après améliorations

Nous constatons une légère hausse de la métrique, cela est dû principalement à l'ajout des accesseurs en lecture et écriture. En effet, ces méthodes sont comptabilisées comme des méthodes accédant aux attributs.

McCabe Cyclomatic Complexity : VG

Nous avons recalculé la complexité cyclomatique des méthodes que nous avons modifiées. Nous les représentons dans le tableau suivant :

Nom du package	Valeur avant modification	Valeur après modification
CollisionDetection	25	1
GetEntityAt	9	1

Figure III.E.8 : Complexité cyclomatique après améliorations

La forte réduction de complexité des deux méthodes est due principalement à l'application du patron de conception « delegate » ainsi qu'à la généricité paramétrique. Cette amélioration a pour conséquence la réduction de la complexité cyclomatique moyenne de tous le Framework.

Taux d'abstraction

Le taux d'abstraction permet de connaitre le taux de réutilisation et de flexibilité du Framework. Lors de notre étude du Framework, nous avons détecté que le package entities contenait des classes qui jouaient le rôle de point de paramétrage. La classe GameManager contenue dans le package model jouait aussi ce rôle.

C'est pour cela que le package Entities a gardé le taux maximum d'abstraction. Cependant, le package Model se voit diminuer son taux d'abstraction. Cela est dû à notre modification qui fait en sorte que le développeur de jeu n'a pas à se soucier de l'implémentation du Framework, et doive juste étendre le GameManager du package Model.

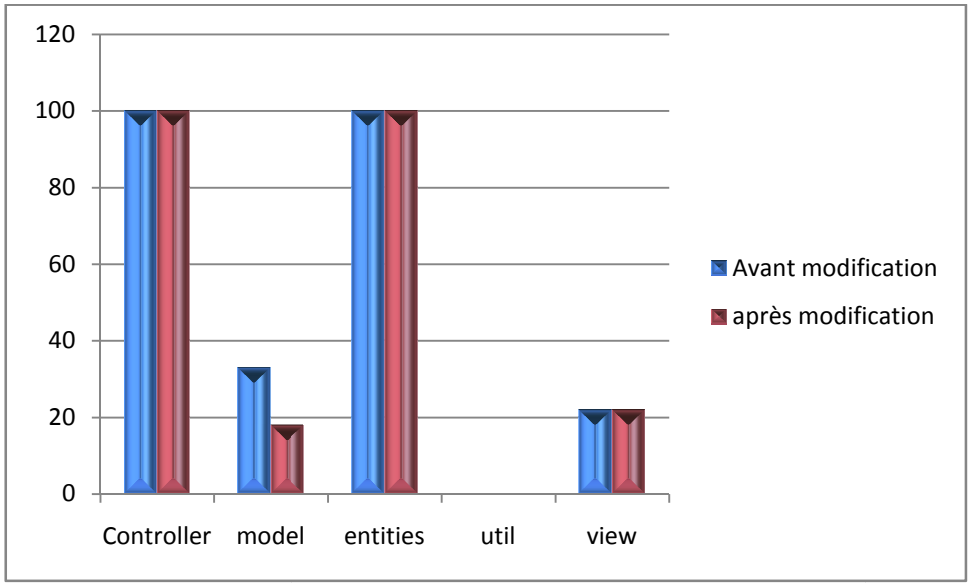


Figure III.E.8 : Taux d'abstraction après améliorations

Number of Methods (NOM):

Les packages model et Entities ont connu une hausse importante du nombre de méthodes. Cette hausse est d'environ 10 méthodes par package. Cela s'explique par le changement du design du Framework. En effet, l'application des patrons de conception et de réingénierie nous permet d'avoir un code réutilisable mais l'inconvénient est l'augmentation de la quantité du code.

Section IV. Conclusion et Synthèse du Projet

Dans ce projet, nous étions amenés à analyser le Framework « Bouding-balls » et le restructurer afin de l'améliorer.

Analyse :

Les différentes techniques que nous avons utilisées au long de ce projet nous ont permis de comprendre et améliorer notre Framework et ainsi d'atteindre notre objectif final :

Une étude de l'architecture générale du framework nous a facilité la lecture du code. Tous les deux nous ont permis de créer notre propre jeu.

Nous avons analysé le Framework et nous avons réalisé les tests et les calculs métriques à l'aide des logiciels « Metric » et « CodePro Analytix » qui nous ont tous les deux permis de mettre la main sur les points critiques qui nécessitaient une étude approfondie afin de corriger les défauts.

Amélioration proposées:

Suite à l'analyse que nous avons effectuée, nous avons proposé l'ensemble des améliorations suivantes :

a) Refactoring de quelques classes considérées comme critiques.

b) Réécriture de quelques méthodes.

c) Gestion des exceptions

d) Amélioration de la Java.

Nous avons fait appel à des schémas de conception et plusieurs autres techniques pour atteindre notre but.

Validation de l'amélioration :

Enfin, Nous avons vérifié que les résultats que nous avons obtenus répondent bien à nos attentes initiales.

Ce qui nous a permis de valider nos améliorations et ainsi donner plus de réutilisabilité et de modularité au framework, facilitant ainsi sa maintenance et augmentant ainsi sa qualité.

Nous avons par ailleurs réalisé des fiches de lecture détaillées relatives au livre « Object-Oriented Reengineering Patterns ».
Ces fiches de lecture sont présentées en annexe.

Il faut noter que le travail effectué sera peut être la base d'un deuxième voir un nième refactoring possible.

L'utilisation du pattern State ou d'autres patterns ainsi que la réalisation des Tests unitaires pourront être une suite possible au travail réalisé.

Section V : Bibliographie :

Ouvrages:

Le livre " Object-Oriented Reengineering Patterns".
Le livre "Refactoring des application Java/J2EE".

Liens:

http://rainet.enic.fr/unit/a43/s11/frame_a43_s11.htm
http://rainet.enic.fr/unit/A43/s12/frame_a43_s12.htm

Plan Annexe :

1) Tableau de figures.

2) Règles de violation (calculés dans l'analyse qualitative).

3) Les fiches de Lecture de l'ensemble des chapitres du Livre : Object-Oriented Reengineering Patterns.

Tableau de figures.

1) Règles de violation

Spécialisation abstraite **Abstract Specialisation**

Les classes abstraites ne devraient pas être des sous-classes de classes concrètes. Exception faite pour les sous-classes de la classe java.lang.Object.

Redéfinir toujours toString () **Always Override toString()**

Chaque classe devrait redéfinir la méthode toString (). Exception faite pour les classes non instanciables et les classes ne dérivant pas de la classe Object dans leurs hiérarchies d'héritage.

Convention de nommage des méthodes booléennes **Boolean Method Naming Convention**

Le nom des méthodes booléennes doit commencer par «is», «can», «has» ou «have». Les méthodes non booléennes ne devraient pas commencer par «is».

Les exceptions interceptées ou Catchées **Caught exceptions**

Quelques exceptions ne devraient pas être interceptées (catchée). Cette règle recherche des clauses catch qui interceptent des classes d'exceptions non permises car soit trop générales comme Throwable ou Exception ou alors non marqués comme Error ou RuntimeException.

Fermer les descripteurs ouverts **Close where created**

Les descripteurs de fichier ouverts, en lecture, écriture, les sockets et les streams doivent être fermés dans la méthode où ils étaient ouverts. Donc les instances des sous-classes de java.io.InputStream, java.io.OutputStream, java.io.Reader, java.io.Writer, et java.net.Socket doivent être fermées dans la méthode où ils sont ouverts de manière à éviter des erreurs.

Les constantes dans les comparaisons **Constants in comparaison**

Les constantes doivent apparaitre sur le même côté soit à gauche, soit à droite des expressions de comparaisons tout au long du code. Il est conseillé de les mettre plutôt à gauche des expressions de comparaison pour que le compilateur détecte s'il y a une erreur due à l'usage à tort de l'opérateur d'affectation (=) ou qu'il s'agit bel et bien de la comparaison d'égalité (==).

Les blocs dans else **Dangling Else**

Des blocs {} doivent être utilisés pour éviter l'ambiguïté des clauses Else. Donc les clauses Else doivent toujours être précédées d'un bloc pour éviter de se perdre entre les clauses Else.

Code de débogage **Debugging Code**

Le code de débogage ne doit pas être laissé dans le code final de production. En effet cette règle cherche les endroits du code où on écrit soit sur System.err, soit sur System.out ou alors lorsque les méthodes Throwable.printStackTrace (), Thread.dumpStack (), Runtime.freeMemory (), Runtime.totalMemory (), Runtime.traceMethodCalls () ou Runtime.traceInstructions () sont invoquées.

Déclarer comme interface **Declare As Interface**

Les variables de certains types doivent être déclarées en utilisant une interface. Cette règle recherche les déclarations de champs de classe, des variables locales et des méthodes dont le type doit avoir été déclaré comme une interface, mais qui a été déclaré comme une classe qui implémente l'interface.

Déclarer le constructeur par défaut **Declare default constructor**

Toutes les classes doivent implémenter le constructeur par défaut.

Rejeter les sleeps dans les boucles **Disallow sleep inside while**

La méthode sleep () ne doit pas être invoquée au sein des boucles while car cela met en place de longue boucles d'attente, ce qui est inefficace. Par contre il est conseillé d'utiliser les appels wait () et notify() pour bloquer le fil jusqu'à ce qu'il soit possible de procéder à l'exécution.

Clause de catch vide **Empty catch clause**

Les clauses catch ne doivent pas être vides. Cette règle cherche les endroits où une exception est interceptée (catchée) mais rien n'est fait.

Méthode vide **Empty method**

Les méthodes dont le corps est vide ne doivent jamais être utilisées. Ceci se produit souvent lorsque quelqu'un a oublié d'implémenter la méthode.

Usage explicite de 'this' **Explicit 'this' usage**

Les variables d'instances doivent, ou non, être accessible en utilisant 'this'. Cette règle peut être configurée afin de permettre l'usage explicite de 'this' pour accéder aux variables d'instances et déclencher une violation au cas où on pouvait les accéder sans ambiguïté en enlevant le 'this'.

Convention de Javadoc pour les champs **Field Javadoc conventions**

Chaque champ d'une classe doit avoir un commentaire Javadoc associé.

Commentaire de fichier **File Comment**

Chaque fichier compilable doit avoir un commentaire de fichier tout au début du fichier même avant les imports ou la déclaration de package.

Ordre des importations **Import Order**

La déclaration des importations doit être ordonnée systématiquement. Cet ordre peut être configuré.

Déclaration des variables locales **Local Declarations**

Cette règle vérifie que les déclarations de variables locales suivent bien un certain style de codage prédéfini par l'utilisateur tel que forcer la déclaration de toutes les variables au début du bloc englobant.

Appel des méthodes au sein de la condition de boucle **Method invocation in loop condition**

Les méthodes ne doivent pas être invoquées dans la condition d'une boucle. Cette règle recherche les emplacements où une méthode est invoquée dans la partie de la condition d'une boucle. A moins que la méthode renvoie une valeur différente à chaque fois qu'elle est appelée. Mettre cette méthode dans la condition de la boucle va forcer autant d'appels que de tours de cette boucle. Il est plutôt conseillé de l'invoquer une seule fois juste avant l'entrée de la boucle.

Convention de Javadoc pour les méthodes **Method Javadoc convention**

Chaque méthode doit avoir un commentaire Javadoc associé. Cette règle vérifie que chaque commentaire Javadoc comprend une balise @param pour chaque paramètre, une balise @return si la méthode a un type de retour non void et une balise de @throws pour chaque exception explicitement déclarée.

Bloc manquant **Missing block**

Une déclaration simple sans bloc {} ne doit jamais être utilisée là où un bloc est autorisé surtout pour les déclarations concernées par un if, do, for ou while.

Absence de dictionnaire d'orthographe **No Spelling dictionnary**

Cette règle est plutôt interne au fonctionnement de CodePro pour vérifier si le dictionnaire d'orthographe a été configuré pour l'application à tester.

Constructeur de classe abstraite non protégé **Non protected constructor in abstract type**

Les constructeurs des classes abstraites doivent être protégés. Les constructeurs d'une classe abstraite ne pouvant être appelés qu'à partir d'une sous-classe, il est préférable alors de leur donner une visibilité protected masquant les accès depuis l'extérieur.

Littéraux numériques **Numeric Litteral**

Des littéraux numériques ne doivent pas apparaître dans le code. A part les littéraux 0, 1 et -1, tous les autres littéraux doivent être définis dans la liste des littéraux acceptables par l'utilisateur.

Usage des modificateurs obsolètes **Obsolet Modifier Usage**

Les modificateurs obsolètes ne doivent pas être utilisés. Il existe un certain nombre de modificateurs qui sont valides selon la spécification du langage java mais que Sun déconseille fortement l'utilisation. Comme dans l'exemple d'une déclaration de visibilité 'public' pour les méthodes d'une interface, on n'aura pas besoin de le préciser puisque toutes les méthodes d'une interface sont publiques implicitement.

Méthodes surchargées **Overloaded Methods**

La surcharge de noms de méthodes peut être source de confusion et d'erreur. Les méthodes surchargées sont celles qui ont le même nom et le même nombre de paramètres, mais pas les mêmes types de paramètres, ce qui peut conduire à des confusions car il n'est pas toujours évident de connaitre quelle méthode va être sélectionnée au moment de l'exécution.

Convention de nommage des packages **Package Naming Convention**

Le nom des packages doit être conforme à la norme définie.

Convention de nommage des préfixes des packages **Package Prefix Naming Convention**

Le nom des packages doit commencer soit par un nom de domaine de plus haut niveau, soit par deux lettres représentant le code du pays développeur.

Préférer l'interface à la réflexion (invocation de méthodes) **Prefer interface to reflection**

Les interfaces doivent être privilégiées à la réflexion c'est-à-dire à l'appel explicite des méthodes par la méthode invoke (). Tout ce qui est accompli par l'appel de la méthode invoke () peut être fait de manière plus simple par une utilisation judicieuse des interfaces.

Affectation discutable **Questionnable Assignement**

Des affectations ambigües peuvent conduire à des erreurs sémantiques. Il s'agit par exemple des affectations des variables qui contrôlent le flot d'exécution d'une boucle for ou bien des affectations directes aux paramètres d'une méthode au sein de la méthode elle-même. Bien que ces types d'affectation puissent s'avérer parfois utiles, ils ont souvent des effets secondaires non intentionnés.

Nom discutable **Questionnable Name**

Des noms ambigües peuvent conduire à du code source incompréhensible. Cette règle recherche tous les noms des variables, des types, des champs et des méthodes de classes qui ne respectent pas le format pré établie de la longueur des noms et le style de nommage.

Réutilisation des objets non modifiables **Reusable immutables**

Les objets non modifiables pendant l'exécution doivent être déclarés en tant que constantes statiques afin de réduire la quantité des objets à nettoyer par le ramasse miette.

Concaténation des chaines de caractères **String concatenation**

Les chaînes de caractères ne doivent pas être concaténées car selon le système local de chaque pays le format et l'ordre de concaténation des chaînes de caractères peuvent changer. Au lieu de cela, la classe MessageFormat devrait être utilisée pour la construction des chaînes sur la base d'un modèle qui peut être localisé et adapté pour chaque pays.

Littéraux de types chaînes de caractères **String litterals**

Des littéraux de types chaîne de caractères ne doivent pas apparaître dans le code. Cette règle recherche toutes les chaînes qui ne sont pas incluses dans une liste de chaines acceptable prédéfinie par l'utilisateur. Le principal objectif de cette règle est d'aider à l'internationalisation des logiciels car ceci nous obligera à déplacer les chaînes dans des fichiers de propriétés externes qui peuvent être traduits et de ne jamais les mettre en dur dans l'application.

Trop de violation **Too Many Violation**

Cette règle de violation se produit quand pour une classe ou une entité donnée, le nombre de même type de violation dépasse une certaine limite. Dans le cadre de CodePro, c'est de l'ordre de 20 violations de même type.

Convention de Javadoc pour les types **Type Javadoc Conventions**

Chaque type de l'application doit avoir un commentaire Javadoc associé. Cette règle vérifie optionnellement l'existence d'un tag @author suivi du nom de l'auteur du type et du tag @version suivi de la version du type.

Privilégier les affectations élargies **Use Compound Assignement**

Les affectations avec les opérateurs d'affectations élargies (exemple i += 1) sont plus compactes et rendent le code plus facile à lire. Cette règle recherche donc toutes les affectations simples qui peuvent être transformées en affectations élargies.

Privilégier equal () à == **Use equal() rather than ==**

La valeur des chaînes de caractères ne doit pas être comparée à l'aide des opérateurs (==) ou (!=). Il vaut mieux utiliser la méthode de comparaison equal () des objets.

Variable déclarée dans une boucle **Variable declared within a loop**

Les variables déclarées et initialisées au sein des boucles sont souvent sources de différents problèmes et indiquent un problème de conception avec la logique d'un programme.

Variable doit être final **Variable should be final**

Les variables qui sont affectées une seule fois durant toute l'exécution du programme doivent être déclarées « final » car cela communique des informations supplémentaires sur la manière dont la variable est censée être utilisée. Cette règle recherche tous les champs de la classe qui ont été initialisé une seule fois soit lors de l'initialisation, soit dans le constructeur mais pas ailleurs.

Fiches de Lecture du livre

Fiche de lecture du Chapitre 1 :

L'héritage est un mécanisme permettant de créer une nouvelle classe à partir d'une classe existante en lui proférant ses propriétés et ses méthodes.
De la même manière, l'héritage des logiciels permet de créer un logiciel à partir d'un autre déjà existant.

Le fait d'hériter un logiciel signifie que nous ne voulons tout de même pas le jeter mais plutôt le réutiliser, ce qui implique qu'il faudrait se préparer à s'adapter aux changements et donc il faut que le logiciel que nous avons l'habitude d'utiliser soit adaptable aussi.
Heureusement, plusieurs logiciels peuvent être améliorés et mis à jour ou simplement remplacés par d'autres quand ils ne répondent plus aux besoins de nos utilisateurs.
Le but de la réingénierie est de diminuer la complexité du système de l'héritage suffisamment pour continuer à utiliser et adapter le logiciel avec un coût acceptable.

Les raisons de la réingénierie des logiciels peuvent varier significativement. Par exemple :

-Vouloir rendre le logiciel plus performant: nettoyer le code après avoir pensé à la performance.
-Vouloir utiliser le logiciel sur une nouvelle plate forme donc il sera envisageable de séparer le code et les appels système…

Quelles sont les besoins qui nous poussent à faire de la réingénierie :

-Manque de document.
-Manque de tests couvrant tous les cas d'utilisations et les scénarios possibles.
-Le code de base n'est pas synchronisé avec la documentation fournie, personne ne comprend comment ça marche réellement.
-Compréhension limitée du système en sa généralité.
-Trop de temps pour faire un minimum de changements.
-Trop de bugs : des bugs qui surgissent juste après avoir corrigé les premiers, ce qui prouve qu'il y a trop de dépendances et c'est un point critique dans le logiciel que nous sommes entrain d'utiliser.
-Le temps nécessaire pour une recompilation est très grand ralentissant ainsi notre capacité à effectuer des modifications, cela veut dire que l'organisation du système est beaucoup trop complexe pour que le compilateur fasse le travail efficacement.
-Difficulté de gestion du produit quand il s'agit de plusieurs clients : Difficulté à l'adapter à chacun de ces clients.
-Code dupliqué (c'est un code qui doit être changé): de longues méthodes, de très grandes classes, de longs paramètres…

Qu'y a-t-il de spécial avec les objets ?

Il faut savoir que la personne qui voudra utiliser un code orienté objet remarquera qu'il sera difficile de trouver les objets car en réalité l'architecture d'une application orientée objet veut qu'ils soient cachés.

Les techniques décrites dans le livre aideront à résoudre le problème de la compréhension du système orienté objet.

Le cycle de vie de la réingénierie :

a)L'ingénierie inverse :

L'ingénierie inverse est une technique permettant de déterminer l'utilité et le fonctionnement d'un programme. Cette technique nous permet de passer d'un code compréhensif par la machine et pas par l'homme à un code lisible par l'homme et plus par la machine.
Elle consiste aussi à analyser un produit autant qu'objet fini pour en reproduire ses spécifications et identifier ses composants.
Cette technique peut être utilisée pour un grand nombre de raisons :

– Analyser et comprendre un code inconnu (pour déterminer un algorithme par exemple),
– Récupérer du code source perdu dans le cas où nous ne possédons plus que l'exécutable,
– Déterminer l'existence de virus ou de code parasite dans un programme,
– Déterminer des failles de sécurité, ou des erreurs (bugs),
– Déterminer les protections logicielles,
– Comprendre le fonctionnement d'un logiciel insuffisamment documenté, pour, par exemple, faire un logiciel qui inter-opère avec le logiciel tracé.

Ceux-ci ne sont que des exemples d'applications, on peut en trouver bien d'autres. Leurs nombre fait de l'ingénierie inverse une pratique très courante.

b) le refactoring :

Le refactoring est une activité de la réingénierie qui permet de modifier le code afin d'améliorer sa qualité sans altérer son comportement.

La réingénierie Patterns :

Les design patterns sont des modèles génériques qui expriment dans un contexte d'architecture orientée objet une problématique récurrente. A ce titre ils correspondent, dans le domaine de la conception et de la programmation orientée objet, à l'atteinte d'une certaine maturité : le catalogue de ces modèles, dont chacun correspond à la résolution d'une problématique récurrente et caractéristique du domaine, est d'une certaine façon le résultat de la capitalisation d'expérience de la communauté des développeurs orienté objet.
La réingénierie Patterns entraine plus que du refactoring de code, elle va jusqu'à décrire le processus avec la détection des symptômes au début et le refactoring du code pour arriver à une nouvelle solution à la fin.

2ème partie : l'ingénierie inverse - Fiche de Lecture du Chapitre 2

2.1. Réglage de l'orientation

Problème: Comment établir un sens commun du but dans une équipe ?
Solution: Etablir les priorités clés pour le projet et identifier les principaux chemins qui aideront l'équipe à rester sur la voie.
Discussion :
Les maximes sont des règles de conduite qui peuvent aider à mener un projet attiré dans plusieurs directions. Tous les patterns de ce chapitre peuvent être lus comme des maximes.
Cependant, un projet peut complètement dérailler si nous approuvons les mauvaises maximes, ou les bonnes mais au mauvais moment.

2.2. Désignation d'un chef

Problème : Comment maintenir la vision architecturale au cours d'un projet complexe ?

Solution:

Il faudra désigner une personne spécifique dont la responsabilité en tant que chef sera d'assurer le maintien de la vision architecturale.
Discussion :
Dans tout système, l'architecture a tendance à se dégrader au fil du temps et à devenir moins appropriée aux nouvelles exigences. Le défi d'un projet de réingénierie est de développer une nouvelle vision architecturale qui permettra au système de perdurer et d'évoluer au cours des années. Une ébauche d'architecture nous aidera à résoudre les problèmes mais pas les symptômes.

2.3. Discutions en table ronde

Problème : Comment garder l'équipe synchronisée ?

Solution:

Il faudra organiser brièvement et régulièrement des réunions en table ronde.
Discussion :
Les réunions doivent être brèves et prises au moins une fois par semaine, mais tout le monde est prié d'y contribuer. Chacun doit être capable de dire ce qu'il a fait depuis la dernière réunion, ce qu'il a appris, quels problèmes a-t-il rencontré et ce qu'il a prévu de faire avant la prochaine réunion. Il est conseillé de tenir un journal de bord et d'y relater les décisions prises et les tâches qui doivent être accomplies dans un certain délai.

2.4. Le plus important d'abord

Problème : Sur quels problèmes se concentrer d'abord ?

Solution:

Il faudra commencer par travailler sur les aspects les plus importants pour notre client.
Discussion :
Il est difficile d'évaluer ce qui est important pour le client.
Il faudra donc :

- Essayer de comprendre le modèle d'affaire du client.
- Essayer de comprendre quel but mesurable le client veut obtenir (exemple : temps d'exécution).
- Essayer de comprendre si le but principal est de protéger un atout existant ou plutôt d'ajouter de la valeur en termes de nouvelles particularités et capacités.
- Examiner les changements dans le journal de bord et déterminer où il y a le plus d'activités historiques dans le système.
- Si le client est incapable de situer les priorités, rassembler toutes les exigences et faire une estimation de l'effort requis pour chaque tâche identifiable.
- Découper les tâches s'étendant sur une plage de temps trop longue en sous-tâches.

2.5. Régler les problèmes, pas les symptômes

Problème : Comment pouvons-nous aborder tous les problèmes rapportés ?

Solution:

Il faudra vivre la source du problème, plutôt que les requêtes particulières de notre preneur de paris.

Discussion :

Une difficulté commune durant un effort de réingénierie est de décider s'il faut emballer, redécouper, ou réécrire un composant d'héritage.

Il faudra donc :

Régler les problèmes par les symptômes, ce qui nous apprend à nous concentrer sur la source du problème et non sa manifestation. Par exemple :

- Si le code d'un composant d'héritage est stable et que les problèmes apparaissent principalement avec les changements de client, alors le problème est probablement dû à l'interface du composant d'héritage, plutôt que son implémentation, peu importe à quel point le code est mauvais. Il faudra appliquer La Bonne Interface Actuelle p. 207.
- Si le composant d'héritage est en grande partie sans défaut mais inapte pour les changements du système, alors il devrait certainement être « refactoré » pour limiter l'effet des changements futurs. Appliquez Le Partage de la classe mère p. 263.
- Si le composant d'héritage souffre de trop de défauts, Il faudra faire un Pont vers la Nouvelle Ville p. 203.

2.6. Si ça ne casse pas, ne réparez pas

Problème :

Quelles parties d'un système d'héritage devons-nous « ré ingénier » et que devons-nous laisser tel quel ?

Solution:

Il faudra réparer uniquement les parties cassées – celles qui ne peuvent plus être adaptées pour planifier des modifications.

Discussion :

Dans un projet de réingénierie, les parties cassées sont celles qui mettent l'héritage en péril comme:

- Les composants qui nécessitent d'être fréquemment adaptés pour arriver à de nouvelles exigences, mais qui sont difficiles à modifier en raison d'une grande complexité des changements dans le design,
- Les composants précieux, mais qui traditionnellement contiennent de nombreux défauts.

2.7. Faire simple

Problème : Quel niveau de flexibilité devons-nous essayer de construire dans le nouveau système?

Solution :

Nous préférons une solution adéquate mais simple à une solution potentiellement plus générale mais complexe.
Discussion : « Aller au plus simple qui fonctionnera » est une maxime de la Programmation extrême qui s'applique à tout effort de réingénierie. « Inclure les Utilisateurs p. 185 et Construire la Confiance p. 189 ».
Si nous laissons les choses simplement, cela ira plus vite, nous aurons des feedbacks rapides, et nous retrouverons nos erreurs plus facilement. Ainsi nous pourrons passer à l'étape suivante.

Fiche de Lecture du Chapitre 3 : Premier contacte

Dans ce chapitre, le livre traite les problèmes rencontrés lors du premier contact avec un système informatique à étudier. La première question qui vient à l'esprit lorsque nous sommes face à un système complexe et nouveau, c'est : « Par où commencer ?».

Les Problèmes souvent rencontrés lors du 1er contact avec le système sont :
La complexité du système.
L'état du système en question.
Le manque de documentation ou l'absence complète de cette dernière.
Les qualifications des membres du groupe de réingénierie.
Au vue de ses obstacles, ce chapitre propose une méthodologie et un ensemble de patterns qui permettent de faire face à cette situation. Cette méthodologie suggère de commencer par des interviews et des discussions avec les personnes qui ont contribué au développement du logiciel ou qui ont été chargé de sa maintenance. Ces discussions permettront de faire une évaluation sur les facteurs qui contribueront au succès de la faisabilité ou non de la réingénierie du système étudié ainsi que les risques rencontrés (exemple : manque de tests, documentation non à jour….).
Les patterns permettant d'avoir un bon 1er contact avec le logiciel en réingénierie :

3.1 Discuter avec les personnes chargées de la maintenance du système

But du pattern :
Il faudra se renseigner sur le contexte historique et politique de notre projet par des discussions avec les personnes maintenant le système.
Problème :
Comment obtenir une bonne perspective sur le contexte historique et politique du système à « ré-ingénierer».
Difficultés du problème :
-La documentation, si elle est présente, contient typiquement des décisions au sujet de la solution, pas au sujet des facteurs qui ont influencé cette solution.
-Les personnes travaillant avec le système peuvent nous donner de fausses informations. Ils peuvent parfois le faire délibérément s'ils sont responsables de la partie du logiciel causant des problèmes.

Comment surmonter ces problèmes :

-Nous allons devoir demander des renseignements au prés de l'équipe de maintenance du logiciel. Bien qu'il se peut que l'équipe ne connaisse pas tout sur le contexte historique du système. Mais il y'a de fortes chances qu'elle sache l'état actuelle du système.

Solution:

Il va falloir discuter et demander des renseignements au prés du personnel chargé de la maintenance du système. Ainsi que les gens qui ont été étroitement associés à l'ancien système, ils sont bien conscients de l'histoire de ce dernier.

Astuces :

Afin de ne pas tomber dans le piège de récolte de fausses informations. Ce pattern propose des questions pertinentes qu'il faut poser à l'équipe de maintenance, par exemple :

Quel est le bug le plus facile que vous avez dû réparer pendant le dernier mois ? Quel est le plus compliqué ?combien de temps ça vous a pris pour les réparer ?

Comment l'équipe de maintenance récupère les rapports de bugs ?comment se déroule le processus de maintenance ?il y'a-t-il une sorte de base de données permettant d'enregistrer ces rapports ?

Qui sont les gens qui ont été dans l'équipe de développement et qui sont en maintenance ?

Comment trouvez-vous la qualité du code et celle de la documentation ?

Résultats de l'application du pattern:

Obtention efficace d'informations.

Obtention de la « confiance » des mainteneurs du système. Ceci pourra être bénéfique pour les étapes ultérieures du projet de réingénierie.

Parfois les informations obtenues peuvent être insuffisantes pour nous donner une vision globale sur le système. Ce qui nous obligera à chercher des informations complémentaires en se basant sur d'autres patterns comme « Lire tout le code en une heure, et écrémer la documentation ».

3.2Lire tout le code en une heure

But du pattern :

Evaluer l'état d'un système logiciel par le biais d'une revue de code.

Problème :

Comment avoir une première impression de la qualité du code source ?

Difficultés du problème :

La qualité du code varie en fonction des personnes qui ont été impliquées dans le développement et la maintenance.

Le système étant grand, il y'a plusieurs données à analyser. C'est pour cela qu'il est dur d'avoir une évaluation précise.

Le système étant nouveau pour nous. Nous ne pouvons discerner ce qui est pertinent de ce qu'il ne l'est pas.

Comment surmonter ces problèmes :

Nous avons une expérience raisonnable dans le langage utilisé et nous pouvons reconnaitre la qualité du code.

Notre processus de reengineering a un but clairement défini, et nous avons une idée de la qualité de code nécessaire pour atteindre ce but.

Solution:

Il faudra prendre un peu de temps pour étudier le code de manière ininterrompue en prenant des notes, à la fin de la séance de lecture et rédiger un bref rapport qui inclut :

Les indications qui permettent d'affirmer que le reengineering est faisable ou pas.
Les entités qui semblent importantes.
Les parties de code suspicieuses.
Les parties qui doivent être examinées d'avantage.

Astuces :

Il sera judicieux de :
Se renseigner si l'équipe de développement utilisait des styles de codage et des conventions. Cela permet d'accélérer le processus de lecture du code.
Mettre en place des « check-lists » ce qui permettra de se focaliser sur des points précis lors de la lecture du code.
Autres éléments à examiner :
Test fonctionnel et test unitaire : Ils donnent de nombreuse information sur le fonctionnement de certaines fonctionnalités.
Les classes abstraites donnent des indications sur les intentions du design.
Les classes les plus hautes dans la hiérarchie introduisent la définition du domaine, les sous classes introduisent les variations.
Les occurrences de singleton représentent une constante pour l'exécution du système.
Les commentaires sont révélateurs des intentions du design.

Des concessions :
Pour un début efficace il est conseillé de :
Lire le code en se forçant à limiter le temps de lecture et à examiner tout le code en s'appuyant sur son expertise pour retenir ce qui est important.
Se familiariser au jargon des développeurs : Acquérir le jargon des développeurs aide à mieux comprendre le logiciel et à mieux communiquer avec l'équipe de développement.
Résultat de l'application du pattern
Ce pattern permet d'avoir une première abstraction du système, cependant celle-ci ne couvre pas tout les concepts de ce dernier. Par conséquent il faut compléter cette vision par d'autres représentations en utilisant d'autres patterns.

3.3 Ecrémer la documentation

But du pattern :
Evaluer la pertinence de la documentation en la lisant dans un temps limité.

Problème :
Comment identifier les parties de la documentation qui pourront nous aider.

Difficultés du problème :
La documentation dont nous disposons est souvent conçue à l'intention des utilisateurs ou de l'équipe de développement, elle n'est pas toujours pertinente pour faire du reengineering. De plus la documentation n'est souvent pas mise à jour et décalée par rapport à l'état du système.
Nous ne savons pas comment le processus de reengineering va se dérouler, nous ignorons également quelles sont les documents qui sont pertinents.

Comment surmonter ces problèmes :
Une forme de la documentation est disponible : au moins il ya une description qui a pour but d'aider les gens concernés par le système.
Le processus de reengineering a un but clairement défini, nous pouvons donc sélectionner les parties de la documentation qui concernent les parties ciblées.

Solution:

Il faudra :

Préparer une liste récapitulant les aspects intéressants du système pour le projet de réingénierie. Et, pour chacun de ses aspects, essayer de faire la correspondance avec la documentation correspondante tout en prenant en compte sa date de mise à jour. Enfin caractériser dans un court rapport nos résultats en incluant :

Une justification de l'utilité de documentation.

Une liste des parties de la documentation (des spécifications, d'importantes contraintes, les schémas de conception, des manuels utilisateurs et l'opérateur) qui seront utiles et pourquoi.

Astuces :

Nous seront amenés à :

Essayer d'extraire les informations utiles rapidement, en s'appuyant sur les points suivants :

La table des matières donne une vision globale du contenu de la documentation.

En regardant les numéros des versions et les dates, nous pouvons avoir une idée globale sur le « jusqu'à quelle date la documentation resterait à jour »

En s'appuyant sur les schémas fournis avec la documentation. Ils fournissent une bonne source d'informations.

Les compromis :

La documentation fournie n'est pas écrite pour un but de réingénierie. Donc elle n'a souvent pas un niveau d'abstraction assez élevé pour fournir des informations pertinentes.

Il faut se concentrer sur les parties qui paraissent pertinentes et donc orienter sa lecture permettra de trouver plus facilement des informations et minimisera la perte de temps.

La documentation peut contenir des incohérences, il faut donc éviter de prendre d'importantes décisions en s'appuyant sur la documentation.

Résultat de l'application du pattern :

Ce pattern permet de savoir si la documentation fournie avec le système sera utile ou non. Il permet de savoir quelle partie de la documentation est utile.

3.4 Interview During Demo:

But du pattern:

Obtenir une première impression sur les fonctionnalités les plus appréciées du logiciel, en assistant et en interviewant les gens faisant la démonstration du logiciel.

Difficulté du problème :

Des scénarios d'utilisation typiques varient beaucoup selon les utilisateurs

Le système est grand, et il y'a beaucoup de données à inspecter. Donc il est difficile de mettre au point une évaluation précise.

Nous ne sommes pas familiers avec le système. Nous ne pouvons juger ce qui est intéressant et pertinent de ce qui ne l'est pas.

Comment surmonter ces problèmes :

Nous pouvons exploiter le fait que le système fonctionne, et aussi la présence d'utilisateurs pouvant démontrer l'utilisation du système.

Solution:

Il faudra :

Observer le système en fonctionnement, en faisant une démo et en interrogeant les personnes qui font la démonstration.

Après la démo, il faudra essayer de produire un rapport traitant les points suivants:

- Quelques cas typiques de scénarios d'utilisation.
- Les fonctionnalités principales offertes par le système, et si elles sont appréciées ou non.
- Les composants système et leurs responsabilités.

Astuces :

Il sera nécessaire de :
Sélectionner de manière efficace les gens qui feront la démonstration, nous pouvons choisir des gens faisant parti des catégories suivantes :
- Un utilisateur final du système, nous permettra d'avoir une vision externe de ce dernier.
- Un « manager » nous donnera une idée sur la grandeur du domaine que le système occupe dans l'entreprise.
- Un « help desk » nous donnera l'ensemble des problèmes que les utilisateurs rencontrent, et comment il fait pour surmonter ces bugs
- un développeur du système ou quelqu'un de la maintenance pourra nous donner des informations concernant ce qui se passe dans le système pendant l'exécution de ce dernier.

Compromis :

Il faudra :
Se concentrer sur les fonctionnalités importantes du système.
Savoir extraire les informations pertinentes. Cela est un peu difficile parce qu'une démonstration offre peu d'informations et donc il faut savoir quoi prendre.
Résultat de l'application du pattern :
Le résultat d'application du pattern est d'avoir différents aperçus sur le système (son utilisation externe, fonctionnement interne, son importance dans l'entreprise…).

Do a Mock Installation

But du pattern :
Vérifier si nous avons les moyens nécessaires pour compiler et installer le système sur notre machine.

Problèmes :
Le système est nouveau pour nous, et nous n'avons pas d'idée claire sur les fichiers indispensables pour installer le système.
Il est possible que le système se base sur des librairies, Framework, et patches…et il se peut aussi que nous n'avons pas les bonnes versions de ces derniers.
Le système est complexe. De ce fait, il sera difficile de reconnaitre la configuration optimale.

Comment surmonter ces problèmes :
Nous avons accès au code source et aux outils de compilations (make files, compilateurs….)
Nous avons la possibilité d'installer le système dans un environnement similaire à celui où il s'exécutait.

Solution:
Il faudra
Essayez d'installer le système dans un environnement propre au cours d'une durée limitée de temps (au plus un jour) ainsi qu'exécuter l'autotest si le système comprend un.

Fiche de lecture du chapitre 4 «Compréhension initiale »

Analyser les données persistantes

But du pattern :

Identifier les principaux objets de valeurs qui doivent être stockés dans une base de données système.

Problème :

Quelles structures d'objets représentent les données de valeurs pour le système?

Sources du problème

Les données vitales sont stockées sur des périphériques de stockage externe (i.e. système de fichier, base de données) qui sont rarement nettoyés et donc contiennent des données obsolètes ou inutiles.
Un grand écart entre la représentation des structures de données sur les supports de stockages externes et leurs représentations lors du chargement en mémoire centrale.
Il se peut qu'une grande partie des données de l'ancien système ne soit pas pertinente pour le nouveau système et donc inutile de l'étudier pour notre projet de réingénierie.

Comment surmonter ces problèmes :

L'ancien logiciel emploie une certaine forme de base de données pour stocker ses données persistantes donc du coup il existe forcément un schéma offrant une description statique de ses données.
Appliquer nos connaissances en matière de transformation des schémas de base de données vers des structures de données du langage d'implémentation pour reconstruire le diagramme de classe à partir du schéma de base de données existants.
Dégager les parties essentielles de la base de données qui ont un rapport direct avec les objectives de notre projet de réingénierie.

Solution

Il sera utile d'analyser le schéma de base de données existant pour en tirer les données qui ont une certaine utilité et un certain rapport avec notre projet de réingénierie afin de créer par la suite un diagramme de classe représentant les entités stockées pour bien documenter le schéma pour le reste de notre équipe de réingénierie.
Pour réaliser cette solution, il faut passer par toutes les étapes de la transformation d'un schéma de base de données relationnelle en un diagramme de classes UML.

Compromis

Positives

Amélioration de la communication au sein de l'équipe de réingénierie et surtout l'équipe de maintenance en assurant l'existence d'un schéma unique de base de données.
Mise au point des données de valeurs pour le projet de réingénierie

Négatives

Les bases de données ont une portée limitée et ne représente qu'une partie du système complet à étudier. Par conséquent, nous ne pouvons pas compter sur ce motif de conception pour avoir une vue entière du système.

- La base de données de l'ancien système contient beaucoup de données inutiles à notre projet de réingénierie donc à nous de les distinguer et de ne tirer que ceux qui sont en rapport avec les exigences du nouveau système.
- Cela requiert une expertise en base de données pour la transformation du schéma de base de données en représentions objet.

- Le diagramme de classe extrait est très orienté vers les données et ne comporte pas les comportements du système donc ca ne représente que la moitié de l'image du système étudié.

Difficultés

Le schéma de base de données lui-même n'est pas toujours la meilleure source d'information pour reconstruire un diagramme de classes. Ceci évolue au fil du temps pour s'adapter aux différentes questions d'optimisation. Par conséquent, il est très important de raffiner le diagramme de classe initial par l'analyse des données et des requêtes SQL embarquées de l'application.

Reproduire le schéma de conception

But du pattern :

Raffiner progressivement le modèle de conception face au code source en vérifiant toutes les hypothèses faites sur ce modèle par rapport au code source.

Problème

Comment extraire les schémas de conception à partir du code source?

Sources du problème

- Il existe de nombreux modèles et schémas de conception et d'innombrables façons de les représenter dans un langage de programmation.
- Une grande partie du code source n'a rien à voir avec la conception mais plutôt avec l'implémentation tel que les interfaces graphiques, les connections avec les bases de données, etc.

Comment surmonter ces problèmes :

- Nous avons une vue et compréhension approximative des fonctionnalités du système obtenu lors des étapes précédentes donc nous devons avoir une première idée des questions de conceptions à aborder.
- Nous avons des connaissances en matière de conception de sorte que nous puissions imaginer comment concevoir le même problème par nous-mêmes.
- Nous sommes déjà familiers avec la structure du code source (obtenu lors de l'étape « Lire tous le code en une heure » donc nous pouvons trouver notre chemin.

Solution :

- Il faudra utiliser notre expertise en développement pour concevoir un diagramme de classes hypothétique représentant approximativement le schéma de conception initial. Ensuite nous

seront amenés à affiner ce schéma en vérifiant si les noms dans le diagramme de classes apparaissent dans le code source et par conséquent modifier notre diagramme. Nous allons répéter ce processus jusqu'à la stabilité du diagramme de classes.

Compromis

Positives

- Spéculer le schéma de conception initiale est une bonne technique pour les grandes applications orientées objets avec des centaines de classes pour lesquelles l'approche bas-haut devient quasiment impraticable.
- Cette technique est un investissement rentable vue ce qu'elle nous rapporte en terme de compréhension du système initial.

Négatives

- Il faudrait avoir une bonne connaissance des idiomes, des schémas de conceptions, des algorithmes et des diverses techniques pour avoir une idée de ce que fait le code source.
- Beaucoup de temps est mis en jeu avant d'obtenir une représentation satisfaisante.

Difficultés

- Maintien de la cohérence : garder à jour le diagramme de classe lors de l'avancement de la phase de retro-ingénierie. Sinon tous nos efforts seront perdus. D'où la nécessité d'avoir des règles de nommage identiques à celles du code source pour les entités de nos diagrammes de classes.

Etude et analyse des entités exceptionnels

But du pattern :

Identifier les problèmes de conception en effectuant des mesures et en étudiant les valeurs critiques et exceptionnelles des entités logicielles.

Problème

Comment décerner rapidement les éventuels problèmes conséquents de conception des logiciels ?

Sources du problème

Difficile de discerner les problèmes d'un schéma de conception puisqu'il faut toujours évaluer la qualité d'un schéma en termes de ce qu'il essaye de résoudre comme problème et donc c'est difficile d'avoir une idée sur la qualité d'un logiciel en étudiant simplement son schéma.
Nécessité de connaitre la structure interne d'un schéma pour confirmer qu'une partie de son code représente des problèmes de conception.
Le système est grand, donc une évaluation détaillée de la qualité de la conception de chaque morceau du code n'est pas possible.

Comment surmonter ces problèmes :

Nous disposons des outils de mesure de qualité pour obtenir rapidement des mesures sur les entités du système ainsi que d'autres outils pour naviguer au sein du code source afin de détecter manuellement les entités qui représentent des problèmes.

Solution

Relever des mesures sur les différentes entités du système tel que la hiérarchie de l'héritage, les paquetages, les classes, les méthodes et les analyser pour rechercher des entités de valeur critiques. Vérifier par la suite si ces anomalies relèvent des problèmes de conception.

Conseils

Quel outil utiliser? Il existe de nombreux outils commerciaux et gratuits qui permettent d'effectuer des métriques. Les principaux critères à prendre en compte pour effectuer notre choix d'outil sont la simplicité de l'utilisation, de l'apprentissage et son intégration dans nos outils de développement.
- Quelles sont les métriques à collecter? En règle générale, il est préférable dans un premier temps de ne prendre en compte que les métriques les plus simples puisque ceux qui sont compliqués demandent plus de temps pour leur analyse et donnent rarement de meilleurs résultats.
- Quels seuils appliquer? Pour des raisons de fiabilité, il est conseillé d'appliquer les seuils standards.
- Comment interpréter les résultats? Examiner simultanément les résultats de différentes métriques pour la même entité et éventuellement en faire des recoupements pour s'assurer d'une anomalie dans le code.
- Comment identifier rapidement les anomalies? Bien qu'il soit possible d'identifier les valeurs critiques d'un système sous un format de représentation tabulaire, cette démarche est assez fastidieuse et source d'erreurs. Il est donc préférable d'utiliser des outils qui proposent d'autres format de visualisation tel que les histogrammes, les nuages de points, etc. facilitant l'analyse rapide des points critiques.
- Comparer les résultats des métriques avec le code: Les métriques constituent une aide précieuse pour détecter rapidement les entités potentiellement problématique, mais un examen manuel du code est nécessaire par la suite pour confirmer la présence d'une anomalie.

Compromis

Positives

- Bien S'adapter : Les métriques sont facilement applicables sur les grands systèmes car seulement 20% de toutes les entités nécessitent un peu plus d'investigation. Lorsque différentes métriques sont combinées correctement, de préférence avec des outils de visualisation, nous pouvons rapidement discerner les parties du système qui représentent des problèmes de conception.
- La vue d'ensemble du système obtenue avec le support des outils de visualisations nous propose des vues graphiques des métriques nous permettant ainsi de voir rapidement les entités qui comportent des anomalies.

Négatives

- Résultats inexacts : certaines entités donnent des résultats qui dénoncent des anomalies qui n'en sont pas véritablement.

- Priorités des anomalies : la phase d'identification des problèmes est plutôt facile mais la vraie difficulté consiste à évaluer la gravité des problèmes par rapport à notre projet de réingénierie.

Difficultés

- *Les données sont fastidieuses à interpréter:* évaluer la qualité d'une partie du code implique la production de plusieurs métriques .L'interprétation et la comparaison de ces données est en final très fastidieux. Il est alors judicieux d'utiliser des outils de visualisation pour examiner simultanément les différents résultats.
- *Ces données Nécessitent une expertise* et une connaissance dans les domaines de collecte et d'interprétation des métriques.

Fiche de lecture du Chapitre 5 : Capture détaillée de Modèle :

Dans un premier temps, la connaissance du système a été faite à l'aide des modèles de premier contact, la priorité principale maintenant sera de créer un modèle plus détaillé et plus approfondi des parties du système qui nous aidera dans la réingénierie.
La plupart des modèles concernés par la Capture détaillée de modèle nécessitent la connaissance de plusieurs techniques, l'utilisation d'outils et un effort considérable par rapport à ce que nous avons vu jusqu'ici.

La question que nous devons se poser c'est comment filtrer les détails importants de ceux qui nous ne sont d'aucune utilité ? Il nous parait évident aussi qu'au fur et à mesure que nous lisons le code, nous nous demandons souvent comment et pourquoi certaines décisions de conception ont été prises ?

Par conséquent une fois le raisonnement de conception d'une situation quelconque a été découvert, il sera important de l'enregistrer afin d'aider nos successeurs à comprendre plus rapidement plutôt qu'être forcé de réinventer la roue.

La vue d'ensemble

Les modèles que nous allons étudier dans ce chapitre sont là pour aider à exposer la conception du système logiciel et pour garder une trace de notre compréhension.
Lier le Code et les Questions est considéré comme la relation la plus fondamentale de ces modèles et la plus facile à s'appliquer. Lorsque nous travaillons sur le code source, il est essentiel de laisser des traces de commentaires, des questions, des hypothèses et des actions possibles à exécuter.
Il est important de comprendre que l'intention de ce modèle n'est pas d'améliorer la base du code lui-même, mais seulement améliorer notre compréhension. Parfois nous nous poseront la question aussi si certaines décisions de conception ont été vraiment nécessaires et vraiment justifiées.
Il est quand même nécessaire de dire qu'avec une comparaison des différentes versions de la base du code nous serons capables d'Apprendre du Passé.

5.1 Lier le Code aux Questions :

Problème.

Comment garder une trace de ce que nous avons compris d'un morceau de code par exemple ?

Solution :

Annoter le code et enregistrer ce que nous avons compris réellement près de l'élément de code auquel il se réfère.

Conseils:

-Les annotations peuvent être des questions, des hypothèses, des listes, ou simplement des observations sur le code que nous voulons enregistrer pour la référence future.
- Des conventions d'utilisation des annotations seront à mettre en œuvre pour une meilleure lisibilité.
-Une fois la solution à une question est trouvée, il est primordial de mettre à jour son annotation.

Différences.

- Synchronisation des annotations avec le code : au fur et à mesure que le code est modifié, les annotations le seront aussi, celles-ci peuvent être supprimées s'elles ne sont plus importantes.
- Amélioration de la communication entre les membres de l'équipe.

Difficultés.

Il faudra :
- Rester le plus court et précis possible dans les commentaires afin de présenter juste le détail et le plus important.
- Les programmeurs ont du mal à écrire des commentaires.

5.2 Faire du refactoring pour comprendre :

Problème :

- Comment pouvons-nous comprendre un morceau de code que nous n'avons pas écrit?
- Ce problème est difficile car le code ne reflète pas nos propres idées.

Solution :

Rebâtir itérativement le code en s'assurant que la structure du code reflète ce que le système fait en réalité. Les tests d'exécution après chaque changement sont indispensables afin de vérifier si nos changements signifient quelque chose.

Conseils

Le but principal ici est de comprendre le système et non pas améliorer le code.
Les changements que nous apportons au code devraient donc être traités comme "des expériences" pour tester notre compréhension du code. En conséquence, il faut toujours garder une copie du code original avant de commencer. Il est possible que nous puissions changer le code et arriver à l'améliorer, sauf qu'à ce stade il est probable que nous puissions causer un désordre dans un autre niveau, plusieurs tests seront les bienvenus afin de s'assurer que nous n'avons violé aucune règle.

Les directives suivantes pourront nous aider à surmonter le cap :

- Rebaptiser des attributs pour transmettre des rôles. Il s'agit surtout des attributs qui ont des noms énigmatiques (pour cela il faut voir au niveau des accesseurs de ces attributs).
- Rebaptiser des méthodes afin de transmettre leurs intentions. Pour récupérer l'intention d'une méthode qui n'a pas d'intention de révéler le nom, il faut examiner toutes les invocations et déduire la responsabilité de la méthode ensuite mettre à jour toutes les invocations et recompilez le système.
- Rebaptiser des classes pour transmettre le but. Pour capturer le but de classe ayant un nom peu clair, il faut examinez les clients de la classe en examinant qui invoque ses opérations d'or, qui en crée les autorités. Ensuite, il faut rebaptiser la classe selon son but, mettre à jour toutes les références et recompilez le système.
- Enlever le code dupliqué pour cela il faut essayer le refactoring dans un seul emplacement
- Remplacez des branches de condition par des méthodes.
- Faire un refactoring de méthode pour des corps très long de méthode en remplaçant les blocs par des méthodes.

Différences :

-Validation progressive vu que la compréhension ne surgit pas tout à coup, il s'agit bien d'un processus itératif.

Difficultés:

-Le risque de présenter des erreurs : Moins nous changeons le code, petite sont nos chances d'avoir des erreurs.
- maintenance d'Outils : il existe des outils divers pour faire du refactoring.
-Acceptation de Changements : Beaucoup de sociétés ont la culture de la propriété du code, donc nous considérons souvent l'amélioration du code de quelqu'un d'autre comme une insulte.
-Quand arrêter. Il est souvent difficile d'arrêter de changer le code quand nous identifions des problèmes. Il faut se rappeler que le but primaire est de juste comprendre le système. Une fois ce but réalisé, il est temps de s'arrêter.

5. 3 Pas vers l'Exécution :

Problème.

Comment découvrir comment les objets sont instanciés à la durée d'exploitation et comment ils collaborent ?
Ce problème est difficile parce que :
Le code source expose la hiérarchie de classe, pas les objets instanciables au moment de l'exécution et comment ils agissent réciproquement.
En présence de polymorphisme. Qu'un objet utilise une certaine interface qu'un autre objet fournit, ne signifie pas que l'ancien est en réalité un client du dernier.
La lecture du code ne nous donnera pas le scénario concret qui peut avoir lieu.

Solution:

Exécuter chacun des scénarios et utiliser le débogueur pour parcourir le code.
Observer les objets qui collaborent et comment ils sont instanciés. Ensuite, généraliser ces observations et enregistrez nos connaissances en référence.

Conseils.

Changer l'état interne des objets peut nous permettre de voir comment des chemins d'exécution alternatifs sont déclenchés.

Différences.

- Complexité de poignées. À une petite échelle il est possible de déduire des collaborations d'objet afin d'analyser le code source. Les outils de coupe peuvent par exemple nous dire quelles déclarations du code source sont affectées par une variable donnée. Pour des systèmes grands et complexes, le nombre de possibilités et d'interactions est trop grand. Donc, la seule façon raisonnable d'apprendre comment les objets collaborent est d'étudier les traces d'exécution.
- L'importance des scénarios : il faut faire de son mieux pour choisir des scénarios représentatifs. Malheureusement, ce choix nous mène à un retour à la case de départ des fois car la seule façon d'être sûr que nous avons un jeu représentatif de scénarios est de vérifier s'ils couvrent toutes les collaborations d'objet possibles.

Difficultés.

Dépendance des Outils. Il faut utiliser les bons moyens pour faire l'exécution. Le programme de mise au point devrait nous permettre de changer l'état interne d'un objet, ou reprendre même une méthode actuellement sur la pile d'exécution.

5.4 Chercher les Contrats :

Déduire l'utilisation appropriée d'une interface de classe en étudiant la voie que les clients utilisent actuellement.

Problème.

Comment savoir ce qu'une classe attend de ses classes clientes pour fonctionner comme prévu.
Ce problème est difficile parce que :

- Les relations de client/fournisseur et des contrats sont seulement implicites dans le code.
- La dactylo et les règles de revue forcent souvent des programmeurs à mettre en péril l'interface du fournisseur. De plus, l'encapsulation construit (par exemple, des déclarations publiques/privées) sont fréquemment employées improprement pour faire face aux questions de mise en œuvre. Par exemple, la base de données et des toolkits d'interface utilisateur exigent souvent la présence d'accesseurs publics des méthodes.

Solution

Généraliser les observations en forme de contrats, c'est-à-dire, les déclarations explicites de ce qu'une classe s'attend de ses clients.

Allusions

Le but ici est de comprendre comment les classes collaborent. Bien que les contrats soient seulement implicites dans le code le plus fréquemment utilisé, il y aura des allusions dans le code entre les différentes classes. Ces allusions peuvent se manifester comme des idiomes particuliers à la langue de programmation dans l'utilisation, des conventions dans l'utilisation par l'équipe de développement, ou des modèles de conception même communs.

Différences.

- Les contrats auxquels les clients et les sous-classes adhèrent ne sont pas nécessairement ceux que la classe soutient en réalité.
- L'examen du code source ressemble à l'extraction en concentrant toute notre attention sur des utilisations idiomatiques, nous serons capables de réduire le facteur sonore largement.

5. 5 Apprendre du Passé.

Problème.

Comment pouvons-nous découvrir pourquoi le système est conçu de la façon dont il est actuellement ?

Ce problème est difficile parce que :

- Les leçons apprises pendant un processus de développement sont rarement enregistrées dans la documentation. En outre, les perceptions des développeurs et la mémoire des décisions de conception ont tendance à se déformer après un laps de temps. Donc, nous ne pouvons compter que sur le code source.

Solution.

Utiliser la métrique ou l'outil de gestion de configuration pour trouver des entités où la fonctionnalité a été enlevée.

But :

Le but est d'obtenir une compassion de comment et pourquoi le système s'est développé à son état actuel.

Difficultés

Quand trop de changements ont été appliqués sur le même morceau de code, il devient difficile de reconstruire le processus de changement.

Raisonnement.

Beaucoup de systèmes orientés objet ont surgi via une combinaison d'interactivité et de développement progressif (voir [Boo94] [GR95] [JGJ97] [Ree96]).

Il est valable de reconstruire ce processus d'étude parce qu'il aide à comprendre le raisonnement inclus dans la conception du système. Une façon de reconstruire le processus d'étude est de récupérer ses étapes primitives.

Utilisations Connues.

Il faudra utiliser une représentation visuelle tridimensionnelle pour examiner l'histoire de sortie logicielle d'un système [JGR99].

Quel Avenir ?

Maintenant que nous avons découvert quelques parties stables dans la conception, nous voulons à présent probablement les réutiliser.

Fiche de lecture du Chapitre 6 : Les tests : Notre assurance vie !

Ce chapitre évoque la nécessité des tests et plus précisément des tests systématiques, la création d'une base de tests et d'un framework test, des tests réutilisables et aussi de la façon dont les tests aident à la compréhension du système.

6.1. Ecrire des tests pour permettre l'évolution

But du pattern : Protéger notre investissement dans l'ancien code en imposant un programme de test systématique.

Problème : Comment minimiser les risques d'un projet de réingénierie ?

Le risque :

- d'échouer en simplifiant l'ancien système,
- de rendre le système plus complexe,
- de casser des particularités qui fonctionnent habituellement,
- de dépenser trop d'énergie sur les mauvaises tâches,
- d'échouer à héberger les changements futurs ?

Ce problème est difficile parce que :

- L'impact des changements ne peut pas toujours être prévu car les parties du système peuvent être mal comprises et peuvent cacher des dépendances.
- Tout changement dans un ancien système pourra être déstabilisant. Ceci est dû aux aspects de dépendances pour lesquels il n'y a pas de documentation.

Cependant, résoudre ce problème est faisable parce que :

- Nous avons un système courant, donc Nous pouvons déterminer ce qui fonctionne ou pas.
- Nous savons quelles parties du système sont stables et celles sujettes au changement.

Solution :

Il sera temps d'introduire un processus de test basé sur les tests qui sont automatisés, répétitifs et mémorisés.

Conseils : Les tests bien conçus sont :

- automatisés (i.e. doivent s'effectuer sans l'intervention humaine),
- persistants (i.e. doivent être mémorisés pour être automatisables),
- récurrents (i.e. doivent pouvoir être répétés après que chaque changement soit implémenté),
- associatifs (i.e. peuvent être associés à des composants logiciels individuels afin d'identifier quelle partie du système est testée),
- indépendants (i.e. doivent minimiser leurs dépendances avec d'autres tests).

Compromis

Avantages :

- Les tests augmentent notre confiance envers le système et améliorent notre aptitude à changer la fonctionnalité, le design et même l'architecture du système.
- Ils renseignent sur la façon dont les artéfacts du système doivent être utilisés. En contraste avec la documentation écrite, effectuer des tests est toujours une description à jour du système.
- Il est important de noter que le fait de vendre des tests aux clients concernés par la sécurité et la stabilité n'est généralement pas un problème : Assurer une longue vie au système est aussi un bon argument.
- Ils fournissent le climat nécessaire pour permettre une future évolution du système.
- Il existe des frameworks de test pour tout langage principal orienté objet comme Smalltalk, Java, C++ et même Perl.

Inconvénients :

- Les tests ne sont pas gratuits. Des ressources doivent être allouées pour les écrire.
- Ils démontrent seulement la présence d'erreurs. Il est impossible de tester tous les aspects d'un ancien système.
- De plus, nous penserons que notre système fonctionne bien parce que tous les tests se déroulent bien, mais ça pourrait ne pas être le cas.

Difficultés :

Il est difficile de :

- Choisir une approche simple qui s'adapte à notre processus de développement.
- Tester d'anciens systèmes. En effet, les systèmes ont tendance à être vastes et sans documentation. Parfois, tester une partie du système requiert une procédure d'initialisation grande et complexe qui pourrait sembler prohibitif.
- Les développeurs sont réticents à utiliser les tests qu'ils trouvent d'ailleurs ennuyeux.

- Pour unir les tests, SUnit et ses nombreuses variantes sont simples, gratuites et disponibles pour Smalltalk, C++, Java et d'autres langages.

Raisonnement : Les tests représentent la confiance en un système parce qu'ils spécifient comment les parties du système fonctionnent de façon vérifiable, et parce qu'ils peuvent être exécutés à tout moment pour vérifier si le système est toujours cohérent.
Une gamme de tests d'exécution fournit une confiance au niveau du système, elle représente l'information que nous avons l'intention de reproduire et vérifie la synchronisation avec l'application.
Il faut préciser que l'écriture des tests augmente la productivité, parce que les bogues sont trouvés plus tôt dans le processus de développement.

Les patterns liés

Il est sûr que l'écriture des tests (pour Permettre l'Evolution) est une condition pour Toujours Avoir une Version Fonctionnelle p.199. Seulement avec un programme de tests complet, est ce qu'il serait possible de Faire Migrer les Schémas de façon Incrémentielle ? p.191.

6.2. Augmentez notre base de tests de façon incrémentielle

But du Pattern :

Equilibrer les coûts et les bénéfices des tests en introduisant de façon incrémentielle uniquement les tests dont nous avons besoin à un moment précis.

Problème : Quand faut il commencer à introduire des tests ? Quand devons-nous s'arrêter ?

Ce problème est difficile parce que :

- Dans un projet de réingénierie, nous ne pouvons pas nous permettre de passer trop de temps à écrire des tests.
- Les anciens systèmes ont tendance à être énormes, donc tout tester est impossible.
- Les anciens systèmes ont tendance à être peu documentés et peu compris.
- Les développeurs authentiques pourraient abandonner, et ceux chargés de l'entretien du système ont seulement des droits limités sur les travaux internes de celui-ci.

Cependant, résoudre ce problème est faisable parce que :

- Nous savons où les parties fragiles du système ou les parties que nous aimerions changer se situent.
- Nous pouvons convaincre les programmeurs qu'ils peuvent bénéficier des tests.

Solution : Introduire les tests de façon incrémentielle pour les parties du système sur lesquelles nous travaillons.

Conseils :

- Quand nous faisons de la réingénierie, il sera primordiale d'introduire des tests pour les nouvelles particularités, les parties de l'héritage qui pourraient être touchées et tous les bogues que nous identifions durant ce procédé.

Il faudra :

- Garder une sauvegarde de l'ancien système.
- Commencer par écrire des tests pour les parties du système qui ont les artéfacts les plus importants. Essayer d'Enregistrer les Lois d'Affaires comme des Tests.
- Si nous avons l'historique des bogues ou problèmes résolus, il faudra appliquer et tester les anciens bogues p.316 comme point de départ.
- Si nous avons une documentation satisfaisante et des développeurs authentiques sous la main, il faudra examiner et tester les particularités Floues p.316.
- Appliquer et tester l'interface et pas l'implémentation. Commencer par tester les grosses abstractions puis raffinez les tests si le temps le permet.
- Faire une boîte noire de tests (sous-systèmes, classes, méthodes) qui promettent un changement d'implémentation dans le futur.

Compromis

Avantages :

- Nous économisons du temps en développant seulement les tests dont nous avons besoin.
- Nous accumulons une base des tests les plus critiques.

- Nous avançons avec confiance.
- Nous rationalisons le développement à venir et maintenons les activités.

Inconvénients :

- Nous pourrions mal supposer quels sont les aspects les plus critiques.
- Les tests peuvent nous donner une fausse confiance.car des bogues non testés peuvent encore se tapir dans le système.

Difficultés :

- Mettre à jour le contexte convenable pour les tests.
- Identifier les limites des composants à tester est juste difficile. Décider quelles parties tester et comment les tester requiert une bonne compréhension du système et de la façon dont nous avons prévu de le ré-ingénier.

Raisonnement : Une stratégie de tests incrémentielle permet de ré-ingénier avant que tous les tests ne soient mis en place. En nous concentrant uniquement sur ceux qui concernent les parties du système en cours de modification, nous permettons le changement avec un investissement minimum en tests tout en agrandissant notre base de tests.

Les patterns liés : Utiliser un Framework Test pour organiser nos tests. Tester l'Interface, Pas l'Implémentation fournit une stratégie pour développer des tests de niveau arbitraire. Rapporter les Lois d'Affaires comme Tests fournit une autre stratégie pour tester des composants qui implémentent la logique des affaires. Ecrire des Tests pour Composants nous aide à préparer une base de tests pendant que nous inversons encore l'ingénierie du système.

6.3. Utiliser un Framework Test

But : Encourager les développeurs à écrire et utiliser des tests de régression en fournissant un framework qui facilitera le développement, l'organisation et l'exécution des tests.

Problème : Comment encourageons-nous l'équipe à adopter les tests systématiques ?

Ce problème est difficile parce que :

- Les tests sont ennuyeux à écrire.
- Ils requièrent une donnée test considérable pour s'accumuler et diminuer.
- Il peut être difficile de distinguer la défaillance des tests et les erreurs inattendues.

Cependant, résoudre ce problème est faisable parce que :

- La majorité des tests suit le même pattern basique : créer quelques données tests, accomplir quelques actions, voir si les résultats satisfassent nos attentes, nettoyer la donnée test.
- Une très petite infrastructure est nécessaire pour exécuter les tests et rapporter les échecs et erreurs.

Solution : Utiliser un framework test qui permettra aux suites de tests d'être composées de cas de tests individuels.

Etapes : Un ensemble de frameworks test comme JUnit et SUnit ainsi que différents packages commerciaux de tests sont compatibles pour la plupart des langages de programmation. Sinon, nous serons amenés à préparer notre framework suivant les principes suivants :

- L'utilisateur doit fournir des cas de tests (qui initialisent la donnée test) et faire des affirmations sur les résultats.
- Le framework test doit emballer presque tous les cas tests afin de distinguer des échecs d'affirmation et des erreurs inattendues.
- Le framework doit fournir seulement un feedback minimum si les tests réussissent.
- Les échecs d'affirmation doivent indiquer précisément quel test a échoué.
- Les erreurs doivent résulter en feedback plus détaillé.
- Le framework doit permettre aux tests d'être composés comme des suites de tests.

Compromis

Avantages : Un framework test simplifie la formulation des tests et encourage les programmeurs à écrire des tests et à les utiliser.

Inconvénients : Les tests requièrent engagement, discipline et entretien.

Raisonnement : Un framework test facilite l'organisation et l'exécution des tests. Hiérarchiquement, organiser des tests facilite l'exécution des tests qui concernent la partie du système sur laquelle nous travaillons.

Connaître l'utilité : Les frameworks test existent pour un large choix de langages incluant Ada, ANT, C, C++, Delphi, .Net (tous les langages), Eiffel, Forte 4GL, GemStone/S, Jade, JUnit Java, JavaScript, le langage k (ksql, from kbd), Objective C, Open Road (CA), Oracle, PalmUnit, Perl, PhpUnit, PowerBuilder, Python, Rebol, 'Ruby, Smalltalk, Visual Objects et UVisual Basic.

6.4. Tester l'Interface, Pas l'Implémentation / Test de la Boîte Noire

But : Accumuler des tests réutilisables qui se concentrent sur le comportement externe plutôt que sur les détails d'implémentation.

Problème : Comment pouvons-nous développer des tests qui non seulement protègent notre héritage logiciel, mais aussi continueront à être précieux quand notre système changera ?

Ce problème est difficile parce que :

- Les anciens systèmes ont beaucoup de particularités qui pourront continuer de fonctionner alors que le système évolue.
- Nous ne pouvons pas nous permettre de passer trop de temps à écrire des tests en ré-ingéniant le système.
- Nous ne voulons pas perdre de l'énergie à développer des tests qui pourront être changés quand le système changera.

Cependant, résoudre ce problème est faisable parce que :

- L'interface des composants du système nous dit ce qui doit être testé.
- Les interfaces ont tendance à être plus stables que les implémentations.

Solution : Développer des tests de boîte noire qui entraînent l'interface publique de nos composants.

Conseils :

Il faut :

- Etre sûr de tester les limites (i.e. valeurs minimums et maximums pour les paramètres des méthodes).
- Utiliser une stratégie de haut en bas pour développer les tests de la boîte noire s'il y a beaucoup de composants subtils pour lesquels nous n'avons initialement pas le temps de développer des tests.
- Utiliser une stratégie de bas en haut si nous sommes entrain de remplacer une fonctionnalité dans une partie très ciblée de l'ancien système.

Compromis :

Avantages :

- Les tests des interfaces publiques sont plus aptes à être réutilisés si l'implémentation change.
- Les tests de la boîte noire peuvent souvent être utilisés pour tester de multiples implémentations de la même interface.
- Il est relativement facile de développer des tests basés sur l'interface d'un composant.
- Se concentrer sur le comportement externe réduit considérablement l'écriture des tests possibles tout en couvrant encore les aspects essentiels d'un système.

Inconvénients :

- Les tests de la boîte noire n'empruntent pas forcément tous les chemins possibles du programme. Nous devons utiliser un outil de couverture séparé pour vérifier que nos tests couvrent tout le code.
- Si l'interface d'un composant change, nous devons adapter les tests encore une fois.

Difficultés : Parfois, la classe ne fournit pas la bonne interface pour supporter les tests de la boîte noire. Ajouter des accesseurs pour surveiller l'état d'un objet peut être une solution simple, mais elle affaiblit l'encapsulation généralement.
Raisonnement : L'interface d'un composant est une conséquence directe de ses collaborations sur d'autres composants. Néanmoins, les tests de la boîte noire ont une bonne chance d'effectuer les interactions les plus importantes du système.
Depuis que les interfaces ont tendance à être plus stables que les implémentations, les tests de la boîte noire ont une bonne chance de survivre à la majorité des changements du système, et protègent notre investissement au développement des tests.
Connaître l'utilité : Les tests de la boîte noire sont une stratégie de test standard.
Les patterns liés : Enregistrer les Lois d'Affaires représente une stratégie différente pour développer les tests qui se concentrent sur l'exercice des lois d'affaires, et ceci concerne les composants implémentant la logique des affaires. Pour les autres, nous aurons à tester l'interface, Pas l'Implémentation.
Les tests de la boîte blanche forment une autre technique standard pour tester les algorithmes dans lesquels des cas tests sont générés pour couvrir tout chemin possible à travers un algorithme.

6.5. Enregistrer les Lois d'Affaires comme Tests

But : Garder le système synchronisé avec les lois d'affaires qu'il implémente en encodant les lois comme des tests explicitement.
Problème : Comment garder les lois d'affaires actuelles, la documentation sur ces lois d'affaires et le système d'implémentation synchronisés alors que les trois changent ?
Ce problème est difficile parce que :

- La documentation écrite devient vite dépassée et ne nous assure pas que notre système implémente vraiment la description des lois d'affaires que nous avons.
- Les lois d'affaires ont tendance à être implicites dans le code. Il ne sera pas évident de savoir quelles pièces du logiciel sont responsables pour calculer une loi d'affaire donnée.
- Le roulement de développeurs introduit un risque élevé pour nos affaires en ayant de plus en plus de personnes sachant de moins en moins de choses sur le système.
- La plupart du temps, seul un programmeur ou utilisateur connaît les lois spécifiques et cette personne pourrait s'en aller du jour au lendemain.
- Les lois d'affaires sont propices aux changements en raison de facteurs externes comme l'introduction d'une nouvelle loi, donc il est important de les représenter explicitement.

Cependant, résoudre ce problème est faisable parce que :

- La plupart des lois d'affaires sont bien exprimées par un ensemble d'exemples canoniques, chacun nécessite certaines actions bien définies à réaliser et des résultats clairs et observables.

Solution : Ecrire des tests exécutables qui enregistrent les lois d'affaires comme des cas tests, des actions, et qui testent les résultats. Quand les tests cassent, nous savons que les choses sont loin d'être synchronisées.
Conseils : Les développeurs devraient écrire des tests associés à une fonctionnalité spécifique ou bout de code. Les utilisateurs devraient aussi avoir à écrire des tests complets sous forme de cas d'utilisation qui lient plusieurs unités de test.
Compromis :
Avantages :

- Les règles deviennent explicites, réduisant ainsi la dépendance sur la mémoire humaine.
- Nous avons besoin d'enregistrer de toutes les façons les règles d'affaires avant de pouvoir ré-ingénier le système d'héritage.
- Enregistrer les règles d'affaires comme des tests permettant l'évolution : quand de nouvelles particularités doivent être ajoutées, nous devons vérifier que les règles d'affaires existantes

sont toujours implémentées correctement en effectuant les tests de régression. D'autre part, quand les lois d'affaires changent nous aurons à mettre à jour les tests correspondants pour refléter les changements.

Inconvénients :

- Les tests peuvent seulement encoder des scénarios concrets, non pas la logique des règles d'affaires elles-mêmes.
- Quand la logique des affaires traite un vaste choix de cas, il ne serait pas pratique de tous les tester.

Difficultés :

- Enregistrer les règles d'affaires ne signifie pas les extraire. Extraire les règles d'affaires du code avec la technologie courante est une idée de rêve.
- Enregistrer les règles d'affaires peut être difficile pour un système dont les développeurs d'origine et les utilisateurs ont tous déserté.

Raisonnement : En documentant les règles d'affaires comme des tests, nous garantissons que la description des règles d'affaires sera synchronisée avec l'implémentation.

Les patterns liés : Quand l'ingénierie inverse d'un ancien système est notre but, nous serons amenés à écrire des tests pour comprendre. Durant ce processus, il sera naturel d'enregistrer les règles d'affaires comme tests. De cette façon, nous pouvons préparer notre base de tests pendant que nous agrandissons notre base de tests de façon incrémentielle.

6.6. Ecrire nos tests pour comprendre :

But : Enregistrer la compréhension d'une partie du code sous forme de tests exécutables et préparer le terrain pour des changements futurs.

Problème : Comment développer la compréhension d'une partie d'un ancien système qui ne contient ni des tests ni une documentation exacte et précise ?

Solution : Encoder nos hypothèses et conclusions en tests exécutables.

Ce problème est difficile parce que :

- Le code est toujours difficile à comprendre.
- Nous aimerions faire des hypothèses sur ce que le code fait réellement et les valider.
- Nous aimerions spécifier précisément le comportement du système.
- Nous aimerions enregistrer notre compréhension pour la communiquer mais nous ne voulons pas perdre notre temps à écrire des documents qui seront obsolètes aussitôt que nous commencerons à changer le code.

Cependant, résoudre ce problème est faisable parce que :

- La partie de code est relativement petite.
- Nous avons la possibilité de spécifier les tests et de les valider.

Compromis :

Avantages :

- Les tests nous aident à confirmer notre compréhension.
- Ils peuvent fournir une spécification précise de certains aspects du système.
- Ils peuvent être appliqués pour gagner différents niveaux de compréhension. Par exemple, les tests de la boîte noire peuvent nous aider à affiner notre compréhension des rôles et collaborations, tandis que les tests de la boîte blanche peuvent nous aider à comprendre l'implémentation d'une logique complexe.
- Les tests que nous développons aideront à permettre un futur effort de réingénierie.
- Ils nous obligent à être précis au sujet de la création et de l'utilisation des objets à tester.

Inconvénient :

Ecrire des tests demande du temps.

Difficultés :

- Obtenir un contexte bien défini dans lequel nous pouvons tester les objets est difficile, spécialement si les objets à tester ne représentent pas des abstractions spécifiques.
- Les systèmes courants sont connus pour être difficiles à tester, donc les tests peuvent rater des aspects importants.

Raisonnement : En écrivant des tests automatisés, nous testons les parties du système que nous voulons comprendre tout en enregistrant notre compréhension et en préparant le terrain pour un futur effort de réingénierie.

Les patterns liés : Avant d'écrire des tests, nous aurons tendance à vouloir « Refactorer » pour comprendre p.127. Quand nous sommes entrain d'écrire nos propres tests, il faudrait être sûr de bien lire le Code, ainsi que les Questions p.121.

Fiche de lecture du Chapitre 7 : Stratégies de Migration

Ce chapitre traite le problème de migration vers le nouveau système. En effet, sans une bonne tactique de transition, les risques de refus et d'échec du projet de réingénierie sont élevés.
C'est pour cela que ce chapitre offre un ensemble de patterns qui permettront de minimiser les risques d'échec du projet. Ainsi la migration du système se ferait en « douceur ».
Un ensemble de compromis s'imposent lors de la reconstruction du système, nous énumérons les plus importants :
- L'introduction de plusieurs changements à la fois dans le système peut troubler les utilisateurs.
- Les utilisateurs exigent des solutions prêtes. Ils ne veulent pas se distraire par des solutions incomplètes.
- Les données enregistrées doivent rester intactes.

Pour augmenter les chances de la réussite du projet de réingénierie, il faut migrer le système de façon incrémentielle. Pour cela, il faut envisager l'utilisation d'autres patterns, comme « impliquer les utilisateurs ».

7.1 Impliquer les utilisateurs :

But du pattern : Maximiser les chances du non rejet par les utilisateurs, en les impliquant dans toutes les étapes du projet.

Problème :

- Comment pouvons-nous être sûrs que les utilisateurs accepteront le système restructuré?

Difficultés du problème :

- Bien que certains problèmes existent, l'ancien système fonctionne. Et les utilisateurs ont le savoir faire pour contourner la plupart des problèmes rencontrés.
- Les utilisateurs détestent d'apprendre quelque chose de nouveau, sauf si ça leur facilitera la vie.
- Les perceptions des utilisateurs de ce qu'est nécessaire pour améliorer le système ont tendance à changer.
- Les utilisateurs peuvent avoir des difficultés à comprendre un document de conception.

Ce problème est surmontable parce que :

- Les utilisateurs essayeront d'utiliser les nouvelles solutions, s'ils remarquent que leurs besoins sont sérieusement pris en compte.
- Les utilisateurs pourront donner des commentaires si nous leur donnons quelque chose d'utile à utiliser.

Solution :

Impliquer directement les utilisateurs dans le développement, et leur fournir un soutien pendant leurs utilisations du nouveau système.

Etapes :

-Essayer de connaitre les exigences prioritaires des utilisateurs.
-Découper les demandes par priorité et commencer par ce qu'est le plus utile (voir le pattern Most Valuable First).
-Créer un environnement qui favorisera le contact entre les utilisateurs et les développeurs.
-Mettre en place des procédures simples pour fournir les résultats intermédiaires et essayer d'obtenir des commentaires.

Avantages et inconvénients :

- Valider de façon continue les exigences des utilisateurs fera monter les chances de succès du projet.
- Si les utilisateurs sentent que les changements apportés au système en cours de restructuration sont utiles, ils essayeront de donner des commentaires utiles.
- En faisant impliquer les utilisateurs dans le projet, ils auront une facilité avec le nouveau système. Et donc une session de formation ne sera pas nécessaire.
- Les développeurs pourront se sentir embêter en essayant de donner du soutien aux utilisateurs, ce qui pourra les déconcentrer.
- En impliquant les utilisateurs dans le projet, il se peut que leurs exigences augmentent. Ce qui causera une pression supplémentaire sur l'équipe.

Résumé du pattern :

Essayer d'avoir un minimum de contact avec les utilisateurs est nécessaire. Cela permet de savoir si le système répond aux attentes des utilisateurs. Et aussi d'augmenter les chances de réussite.

7.2 gagner la confiance des utilisateurs :

But du pattern :

-Améliorer les chances de succès du projet en montrant régulièrement des résultats.

Problème:

-Comment surpasser le doute et le scepticisme des utilisateurs dû au projet.

Difficultés du problème :

- Les projets de développement de logiciels ont souvent des problèmes de dépassement des contraintes à la fois du budget et du temps. Ils ne répondent pas, souvent, aux attentes complètes des utilisateurs.
- Il est souvent difficile de convaincre les utilisateurs que le système en cours de restructure peut être sauvé.

Solutions :

-Créer une atmosphère positive en montrant des résultats positifs au plutôt et continuer à le faire régulièrement.

Avantages et inconvénients :

- les utilisateurs et les développeurs peuvent constater les progrès.
- Garder une sorte de synchronisation entre les développeurs et les utilisateurs.
- Les utilisateurs peuvent se voir obligés de fournir plus d'efforts pour comprendre le fonctionnement du nouveau système et en même temps travailler avec l'ancienne version.
- Si nous montrons des résultats positifs très tôt, il est possible que les attentes et les exigences des utilisateurs augmentent.

Pattern 7.3 : adopter un système de migration incrémental :

But du pattern :

-Eviter la complexité et les risques de big-bang de réingénierie en déployant des nouvelles fonctionnalités de façon régulière et continue.

Problème :

-Quand est ce qu'il faut commencer à déployer le nouveau système.

Difficultés du problème :

-Les projets sont souvent planifiés et rendu dans un temps précis, non pas comme les projets de réingénierie où les parties traitées sont rendues au fur et à mesure.

-Les utilisateurs vont résister à l'adoption d'un nouveau système qui est radicalement différent de ce qu'ils ont utilisés, surtout si cela ne fonctionne pas sans faille dès le début.

-Déployer le nouveau système en retard implique plus d'attente au niveau de l'obtention des commentaires des utilisateurs.

-Nous ne pouvons pas déployer un système incomplet car les utilisateurs ne voudront pas perdre de temps avec des solutions incomplètes.

Solution :

-Déployer une première mise à jour de l'ancien système, et commencer ensuite à migrer progressivement vers le système cible.

Etapes :

- Décomposer le système en parties.
- traiter une partie à la fois.
- Mettre en place des tests pour la partie abordée et des tests pour les parties qui en dépendent
- Remplacer l'ancien composant par la nouvelle version.
- Itérer jusqu'au traitement de toutes les parties.

Avantages et inconvénients :

- Obtenir les commentaires des utilisateurs très tôt pour créer une sorte de confiance.
- Nous constatons alors immédiatement qu'il y a des parties qui se « cassent » lors du remplacement.
- Les utilisateurs apprennent au fur et à mesure les nouvelles parties du système.
- Il faudra travailler plus pour maintenir le système en marche pendant le changement des parties.

7.3 Prototype the Target Solution

But du pattern :

Evaluer le risque de migration vers le système cible en construisant un prototype.

Difficulté du problème :

Il est risqué d'appliquer des changements radicaux sur un système qui marche.

Il peut être difficile de prévoir l'impact des modifications de conception sur les fonctionnalités existantes.

Solution :

Élaborer un prototype du nouveau concept et l'évaluer tout en respectant les nouveaux besoins.

Etapes :

Identifier les risques techniques du projet de réingénierie comme :

Le choix de la nouvelle architecture.

Migration des données enregistrées dans l'ancien système vers le nouveau.

Décider de mettre en place un prototype exploratoire qui permettra d'évaluer la faisabilité d'une technique ou d'une option ou bien un prototype révolutionnaire selon les besoins du projet.

Avantages et inconvénients :

- Un prototype peut être construit rapidement, puisque nous ne sommes pas obligés d'implémenter toutes les fonctionnalités.
- Nous pourrions alors se servir de l'ancien système pour avoir un prototype qui marche.
- Les utilisateurs ne veulent pas perdre du temps avec un prototype jetable.
- Nous seront tentés par l'idée de continuer le développement du prototype jetable.

Pattern 7.5 : Toujours avoir une version fonctionnelle du logiciel.

But du pattern :

Accroitre la confiance en reconstruisant régulièrement le système.

Problème :

Comment convaincre le client que nous sommes sur la bonne voie.

Difficulté du problème :

Il est peut être difficile de faire une démonstration d'un système logiciel en cours de développement, ou de discuter des problèmes avec les utilisateurs car il n'y a souvent pas une version stable de ce dernier.

7.6 : Effectuer des tests de régression après chaque amélioration.

But du pattern :

Accroitre la confiance en s'assurant que les composants fonctionnels sur l'ancien système, continueront à l'être sur le nouveau.

Problème :

Comment être sûr que les modifications que nous venons d'apporter au système ne le cassera pas.

Difficulté du problème :

Dans un système complexe, les plus petits changements pourront avoir des effets inattendus.

Solution :

Effectuer des tests de régression à chaque fois que nous obtenons une version stable du système en restructuration.

Avantages et inconvénients :

- écrire les tests sans relâche.
- En effectuant des tests, nous avons toujours une « bonne » version du projet que nous pouvons d'ailleurs montrer aux clients. Ceci permettra d'accroitre la confiance de ces derniers.

Pattern 7.7 : Construire un pont vers la nouvelle ville.

But du pattern :

Faire migrer les données et les connaissances stockées dans l'ancien système vers le nouveau, en fonctionnant des deux versions du logiciel en même temps en les connectant avec un « pont ».

Problème :

Comment migrer progressivement les données de l'ancien système vers son remplaçant en faisant fonctionner les deux en même temps.

Difficultés du problème :

Il se peut que certains composants doivent être remplacés au lieu d'être réparés.
Les données en cours de transitions doivent survivre à l'opération de migration et doivent rester cohérentes.

Solution :

Construire un « pont » de données qui permettra de transférer, de façon progressive, les données stockées dans l'ancien système vers son remplaçant.

Etapes :

-Identifier dans l'ancien système et son remplaçant les modules qui traitent les mêmes entités de données.
-Implémenter un « pont de données » qui sera responsable de la réorientation des requêtes selon la disponibilité des données demandées dans le nouveau composant. Le pont prendra en charge les éventuelles conversions nécessaires.
-Supprimer le pont quand toutes les données seront sur le nouveau système.

Avantages et inconvénients :

-Nous pouvons utiliser le nouveau système même si la migration des données n'est pas encore achevée.
-La construction du pont peut être difficile et couteuse si les données ne sont pas simples.

Pattern 7.8 : Présenter à l'utilisateur l'interface adéquate.

But du pattern : envelopper l'ancien système pour exporter ses abstractions, même s'elles ne sont pas reflétées dans l'implémentation.

Problème :

Comment le système cible devrait accéder aux services de l'ancien système durant le processus de migration.

Difficultés du problème :

Le nouveau système n'est pas tout à fait prêt. Donc ce dernier doit utiliser les services de l'ancienne version.

L'ancien système ne dispose pas des interfaces dont le nouveau système aura besoin.

Solution :

Identifier les abstractions qui sont intéressantes pour le nouveau système, et envelopper ces parties pour émuler les nouvelles abstractions.

Avantages et inconvénients :

Il est facile de construire les services du nouveau système si nous isolons les bonnes abstractions.

La nouvelle interface peut ne pas être stable. Ce qui pourra être réticent pour les développeurs de l'utiliser.

Pattern 7.9 : distinguer les interfaces publiques des interfaces publiées.

But du pattern : Faciliter le développement en parallèle, en distinguant les interfaces publiées qui sont instables, et les interfaces publiques qui sont stables.

Problème : Comment permettre la migration depuis les anciennes interfaces vers les nouvelles, alors que les nouvelles sont encore en cours de développement.

Solution :

Distinguer les interfaces publiques des composants qui sont à la disposition du reste du système, et les interfaces publiées des composants qui sont disponibles dans un sous-système, mais ne sont pas encore prêtes à être utilisées.

Astuce :

Il serait astucieux de mettre les nouvelles interfaces en mode protégé « protected » en attendant la fin de leur cycle de développement.

Avantages et inconvénients :

Les clients des interfaces publiées sont conscients qu'ils peuvent être changés.

Identifier une interface comme publiée est une question de convention, donc il n'y a pas de règle générale pour la distinguer.

7.10 Déprécier « deprecate » les interfaces obsoletes.

But du pattern :

Donner le temps au client de l'interface de réagir face au changement de statut (de publique à déprécié) de l'interface que ce dernier utilise.

Problème :

Comment rendre une interface obsolète sans rendre ses clients invalides.

Difficulté du problème :

Changer le statut d'une interface peut endommager les clients de cette dernière.

Garder une interface obsolète dans le code rendra la maintenance plus dure.

Solution :

Marquer les interfaces obsolètes par un marqueur « deprecated » pour notifier ses clients qu'il est probable que l'interface sera supprimer dans le futur.

Etapes :

Déterminer l'interface à rendre obsolète.
Ecrire la nouvelle interface adjacente et déprécier l'ancienne. Cette dépréciation devra faire en sorte qu'elle recommande la nouvelle interface.
Evaluer dans quelle mesure l'interface obsolète continue à être utilisée, et si elle peut être retirée de façon permanente. Envisager la suppression dans une future version.

Avantages et inconvénients :

- Les clients n'ont pas à s'adapter aux changements immédiatement.
- Les clients sont libres de considérer ou non la dépréciation.

Pattern 7.11 : conserver la familiarité du système :

Migrer vers une nouvelle solution d'une manière incrémentale.
Introduire un nombre constant relativement petit de changement après chaque itération

Pattern 7.12 : utiliser un profil de code après et avant toute optimisation.

But du pattern :

Eviter de faire des efforts inutiles d'optimisations dans un projet de réingénierie en localisant les goulots d'étranglement.

Problème :

Quand faut-il réécrire clairement une partie inefficace du code.

Difficulté du problème :

Il est difficile de prédire l'impact en termes de performance en réécrivant une partie du code. Il se peut d'ailleurs que nous perdions beaucoup de temps en s'appuyant sur une simple supposition.
Un code optimisé est souvent plus complexe qu'un code « naïve ».
Ce problème est surmontable parce qu':
Il existe des outils qui peuvent nous indiquer où se trouve les problèmes de performances dans le code.

Solutions :

Lorsque nous sommes tentés par optimiser une partie du code, il faudra utiliser d'abord un profileur pour déterminer si elle est effectivement un goulot d'étranglement.
Il ne faudrait pas optimiser une partie du code sauf si le profiler précise que ceci fera un changement considérable.
Si nous sommes décidés à effectuer l'optimisation, il faut effectuer des « Benchmarks » qui montreront les gains de performances.

Avantages :

- Nous ne perdrons pas de temps en optimisant quelque chose qui ne fera pas une différence de performance globale.

Inconvénient :

- Il est possible qu'après avoir appliqué le pattern, certains bouts naïfs du code survivront.

Fiche de lecture du Chapitre 8 : Détecter la duplication du code

Des statistiques montrent que généralement entre 8 et 12% des logiciels industriels se composent de code dupliqué. Bien que le taux de duplication semble faible, il est très difficile de parvenir à des taux beaucoup plus élevés. Ainsi un taux de 10 à 20% est vraiment très grave pour une application.

Il est très important d'identifier le code dupliqué pour les raisons suivants :

- Le code dupliqué nous empêche d'introduire des modifications car chaque morceau de fonctionnalité dupliqué devra être modifié par conséquent et en cas d'oubli, des bogues sont susceptibles de se produire dans d'autres endroits.

- Le code dupliqué répète et disperse la logique d'un système plutôt que de les regrouper dans des objets identifiables (classes, méthodes, paquetages). Cela nuit à la compréhension et la modification du système.

La duplication du code se produit pour plusieurs raisons :

- Chaque programmeur préfère utiliser le code existant pour implémenter une fonctionnalité similaire à ce qui a été déjà fait auparavant. Copier-coller et ensuite modifier légèrement l'ancien code pour réaliser une nouvelle fonctionnalité n'est pas très grave si le nouveau et l'ancien code se trouvent dans des applications différentes mais conduisent à une duplication de code au cas où ils se trouvent dans la même application.
- Parfois le code est copié, collé et modifié entre différentes applications ou dans des versions différentes de la même application. Nous aurons forcément des problèmes de code dupliqué quand plusieurs versions doivent être fusionnés ou maintenues en même temps.

8.1 : Comparer mécaniquement le code

But :

Découvrir le code dupliqué en comparant ligne par ligne tous les fichiers de codes sources

Problème

Comment découvrir les parties dupliqués du code source de l'application?

Sources du problème

- Nous pensons que le code a été dupliqué mais à priori nous n'avons aucune preuve où cela se produit.

-Parcourir le code en entier n'est pas efficace pour découvrir les duplications, nous ne les découvrirons que par hasard.

-Les programmeurs pourraient ne pas simplement copier-coller le code mais aussi modifier les variables et changer un peu la forme des programmes.

Comment surmonter ces problèmes :

La plupart du code dupliqué peut être détecté par des procédures mécaniques.

Solution :

Comparer textuellement chaque ligne du code source de l'application avec le reste des lignes du code.

Etapes :

-Normalisez les lignes du code en supprimant les commentaires, les tabulations et les blancs.

-Supprimez les lignes ne contenant que des éléments de code pas très intéressant à la comparaison (par exemple seulement des else ou «}»)

-Comparez chaque ligne avec toutes les autres lignes. Nous pourrions procéder à une méthode de hachage pour réduire la complexité de cette opération :

Prétraitement : Calculer la valeur de hachage pour chaque ligne.

Comparaison effectif du code : il faudra comparer toutes les lignes qui se trouvent sur le même intervalle de valeur de hachage.

Compromis

Positives

- Cette approche est simple et donne de bons résultats en n'exigeant que des ressources modestes
- Cette approche est indépendante du langage utilisé dans le sens où le développement se fait par un simple analyseur lexical et non un vrai parseur (pour réalisation de ce pattern).
- De simples données statistiques et pourcentages sont facilement calculés et peuvent nous aider à acquérir plus de crédibilité et de force lors des discussions sur l'allocation des ressources ou éventuellement l'embauche de nouvelles personnes.

Négatives

- Un code qui a été largement modifié après la copie peut être difficilement identifié en tant que code dupliqué.

- Les systèmes contenant trop de code dupliqué génèrent par conséquent trop de données difficiles à analyser de manière efficace par la suite.

8.2 : Visualiser graphiquement le code par des tracés de points

But :
Comprendre la nature de la duplication du code en étudiant la forme graphique (motif) des tracés de points représentant le code.

Problème
Comment avoir un aperçu sur la portée et la nature de la duplication du code dans une application?

Sources du problème
Connaître l'emplacement du code dupliqué dans un système ne nous aide pas forcément à comprendre sa nature ou ce qui devait être fait à son égard.

Comment surmonter ces problèmes :
Une image vaut mille mots

Solution
Visualiser le code comme une matrice dans laquelle les deux axes représentent deux fichiers de code source (peut être le même fichier) et ainsi des points apparaissent sur la matrice là où des lignes de code ont été dupliqué.

Etapes :
Si nous voulons analyser deux fichiers A et B il faut:
-Normaliser le contenu des deux fichiers afin d'éliminer le bruit de fond sur les images (les espaces blancs, etc.)
-Chaque axe de la matrice représente des éléments (par exemple les lignes de code) des fichiers normalisés
-Représenter un appariement (match) entre deux éléments par un point dans la matrice.
-Interpréter les images obtenues : une diagonale représente du code dupliqué entre les deux fichiers.
Pour analyser la duplication au sein d'un même fichier source, il faudrait tracer les éléments de ce même fichier sur les deux axes de la matrice.

Compromis

Positives

- Cette approche est indépendante du langage puisque la phase de normalisation du code dépend de la syntaxe du langage
- Cette approche fonctionne bien pour la rétro-ingénierie vu la grande quantité de code. Ceci à pour cause, les tracés qui attirent l'attention sur les parties du code qui doivent être étudiées de plus prés.
- L'idée est simple mais avec des résultats intéressants. Une version simple de cette approche peut être programmée dans quelques jours

Négatives

- Cette approche de tracés de points représente seulement des comparaisons par paires de fichiers. Elle ne peut pas nous aider à identifier tous les cas de duplication des éléments du système en entier.
- Bien que l'approche puisse facilement être étendue à des fichiers multiples sur chaque axe, la comparaison reste toujours deux à deux.

Difficultés
Un outil simple de visualisation des tracés de point peut être incompatible dans un système à grand échelle et donc optimiser un tel outil peut compromettre à la simplicité de l'approche elle-même
L'interprétation des données peut être plus subtile que cela semble à première vue puisqu'en comparant de multiples fichiers la diagonal peut représenter plus de duplication que ce qui l'en est vraiment dans le système car nous sommes entrain d'essayer de comparer des fragments dupliqués entre eux-mêmes sur plusieurs fichiers.

La taille de l'écran limite la quantité de l'information qui peut être visualisées. Des succès ont été acquis avec les méthodes de visualisation dites «mural» mais la difficulté de la mise en place de ces méthodes ne vaut pas l'effort supplémentaire.

Fiche de Lecture du Chapitre 9 : Redistribuer les responsabilités :

Ce chapitre nous montre comment redistribuer les responsabilités des « god classes » et comment les découper. Il traite aussi la problématique des classes conteneurs et comment éviter la navigation du code.

Pattern 1 : Rapprocher les comportements des données.

But du pattern : Renforcer l'encapsulation en déplaçant les comportements vers les classes contenant les données et en les supprimant dans les clients indirects.

Problème :

Les classes conteneurs de données se distinguent par leur manque de méthodes de comportement. De ce fait la modification de leur représentation interne a une influence sur les classes auxquelles elles sont liées.

Rendre les conteneurs de données, de vraies classes offrant des services.

Solution : Déplacer les comportements définis dans les clients indirects du conteneur vers ce dernier.

Etapes :

Repérer le comportement qui opère sur les données du conteneur de données.

Créer un comportement correspondant dans la classe conteneur.

Dans le client, invoquer le comportement du nouveau conteneur avec les bons paramètres.

Nettoyer le code du client en appelant le comportement demandé quand c'est nécessaire.

Avantages:

Les classes conteneurs ont maintenant de vraies responsabilités.

Les clients sont moins sensibles au changement de l'implémentation interne des classes conteneurs.

Réduction de la quantité du code dupliqué.

Pattern 2 : Eliminer la navigation dans le code.

But du pattern : Réduire les dépendances entre les classes ainsi que l'impact de la modification d'une classe sur ses clients.

Nous reconnaissons l'effet de la navigation de code quand :

La modification d'une classe implique la modification des clients directs, et aussi des clients indirects.

Il y a une abondance des attributs publics et des accesseurs.

Pour pouvoir surmonter le problème de la navigation de code, ce patron nous propose le pseudo-algorithme suivant :

Identifier le code suspect.

Appliquer le pattern « Rapprocher les comportements des données » en éliminant les faux accesseurs.
Répéter si nécessaire.

Avantages :

- Les dépendances entre les classes sont moindres voire éliminées. Et donc peu de clients sont affectés par les changements.

Pattern 3 : Découper la « god classe ».

But du Pattern : Découper une classe détenant plusieurs responsabilités en plusieurs petites classes s'entraidant afin de remplir le rôle de la god classe.

Problème :

Les god classes maintiennent plusieurs responsabilités, ce qui rend difficile l'évolution du système.
La compréhension du système devient moins difficile.
Apporter des changements dans la « god classe » se répercute sur tout le système.

Solution :

Redistribuer les responsabilités de la « god classe » entre des classes afin de s'entraider.
Ce patron de réingénierie propose des indicateurs permettant de détecter une « god classe »:
La taille de la classe est conséquente.
Apporter des changements au système se traduit par un changement d'une classe.
La réutilisation de la classe est presque impossible, du fait qu'elle détient plusieurs taches.
La « god classe » demande beaucoup de temps pour la compilation, même pour des petits changements.
Exigence de beaucoup de place mémoire.
Une fois la « god classe » détectée, nous pourrions appliquer le pseudo-algorithme suivant afin de la découper :

- Regrouper les variables d'instances, liées entres elles et utilisées dans la « god classe » dans un conteneur de données.
- Identifier toutes les classes utilisées comme des conteneurs de données par la « god classe », et leur appliquer le patron : Rapprocher les comportements des données. En procédant de la sorte, les conteneurs de données offriront des services à la « god classe », et les méthodes de cette dernière ne feront que déléguer le comportement aux conteneurs de données.
- Après des applications itératives des étapes (1) et (2), la « god classe »deviendra juste une façade contenant une grande méthode servant à l'initialisation. Nous pourrons alors créer une classe qui sert juste à initialisation.

Avantages :

Le contrôle de l'application n'est plus centralisé dans une seule entité.
Le temps de la compilation devient moindre car nous réduisons les dépendances du système.
Certaines parties du système deviennent plus facile à maintenir et à comprendre.

Défauts :

Augmentation du nombre de classes du système.

Fiche de Lecture Chapitre 10 : Passer du conditionnel au polymorphisme

Après avoir copié le code, l'un des signes les plus étonnants de responsabilités égarées en génie logiciel orienté objet est l'existence de plusieurs méthodes consistant presqu'entièrement en une (ou des) déclaration de cas qui testent le type de certains arguments.

Bien que la déclaration de cas ne soit pas intrinsèquement mauvaise, dans du code orienté objet, elles sont fréquemment un signe que l'objet faisant le test assume les responsabilités qui devraient plutôt être distribuées aux objets qui sont testés. Les conditions importantes surgissent naturellement avec le temps, tout comme le code copié le fait. Puisque le logiciel est adapté pour traiter de nouveaux cas, ces cas surgissent en tant que conditions dans le code. Le problème avec ces conditions importantes est qu'elles peuvent rendre le code encore plus fragile à long terme.

Avantages :

Puisque les besoins changent avec le temps, les classes dans un système logiciel devront être adaptées pour traiter des cas nouveaux, particuliers.

- Ajouter de nouvelles classes ou sous-classes à un système encombre l'espace de nom.
- Le moyen le plus rapide d'adapter une partie du logiciel qui fonctionne pour traiter un nouveau besoin, est souvent d'ajouter un test conditionnel pour le cas particulier à certains endroits du code.
- Avec le temps, un simple design a tendance à devenir encombré avec plusieurs tests conditionnels pour des cas particuliers.
- Les déclarations de cas rassemblent toutes les variantes en un seul endroit au lieu de répartir les différents cas dans différentes classes. Cependant, elles amènent à concevoir ce qui est moins flexible si la déclaration de cas apparaît dans plus d'un endroit.
- Dans certains langages de programmation, la déclaration de cas est un idiome plus conventionnel pour implémenter le comportement variable que le polymorphisme.

Les conditions importantes sont souvent un signe que le comportement implémenté par les clients devrait probablement être déplacé vers les classes du fournisseur. Habituellement, une nouvelle méthode sera introduite dans la hiérarchie du fournisseur, et les cas individuels de la déclaration conditionnelle bougeront chacun vers l'une des classes du fournisseur.

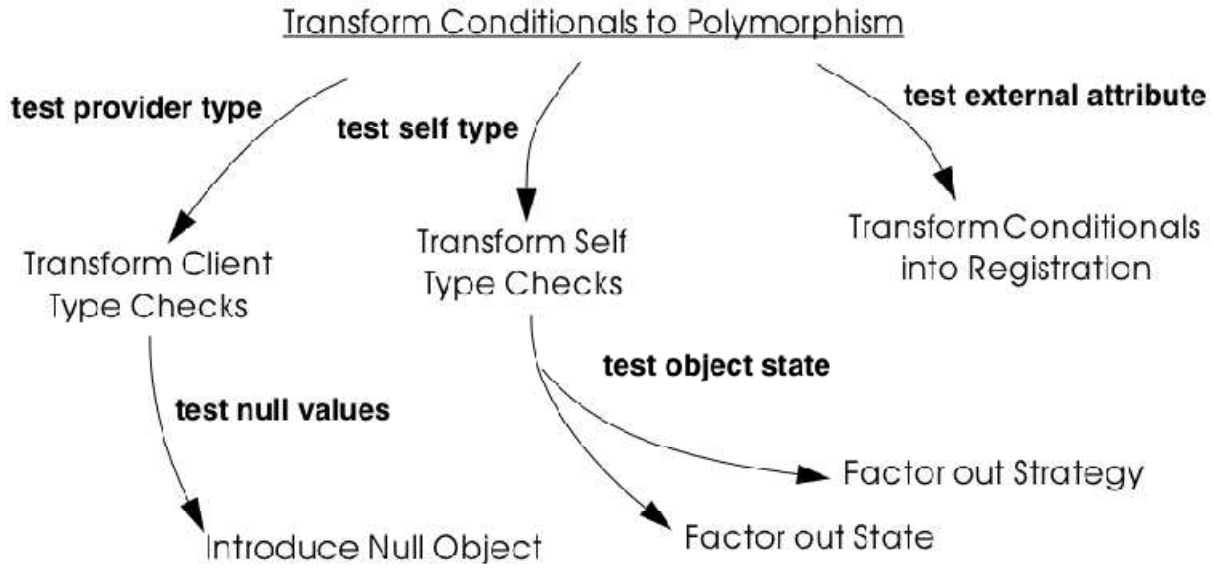
Bien que le symptôme soit reconnaissable à la lecture, les détails techniques et la solution préférée pourraient différer considérablement. En particulier, quand la hiérarchie de fournisseur existe déjà, et que les conditions vérifient explicitement la classe de l'instance du fournisseur, le refactoring est relativement simple. Mais souvent, la hiérarchie du fournisseur n'existe pas, et les conditions testent les attributs qui présentent implicitement le type d'information. De plus, les conditions pourraient ne pas se produire seulement dans des clients externes, mais également dans la hiérarchie du fournisseur elle-même.

Vue d'ensemble

Transformer le conditionnel en polymorphisme est un pattern qui décrit comment redistribuer les responsabilités pour éliminer ces importantes conditions, et par conséquent réduire les rapports entre les classes, et améliorer la flexibilité face aux futurs changements.

Ce pattern consiste en six patterns qui abordent les problèmes les plus communs qui se produisent quand les conditions sont utilisées pour simuler le polymorphisme. Transformer ses propres vérifications de type (Transform Self Type Checks) et transformer les vérifications de type du client (Transform Client Type Checks) abordent les cas les plus typiques qui surgissent quand des vérifications de type explicites sont accomplies. Transformer les conditions en déclarations (Transform Conditionals into Registration) se produit moins fréquemment. Nous incluons aussi Facteur hors déclaration (Factor out State), Facteur hors stratégie (Factor out Strategy) et Introduire l'objet null (Introduce Null Object), pas dans le but de copier trois design patterns établis (State, Strategy et Null Object) mais plutôt pour montrer comment ces design patterns peuvent s'appliquer dans un contexte de réingénierie pour éliminer les conditions de vérification de type.

Cette figure résume les relations et les différences entre les patterns.



- Transform Self Type Checks élimine les conditions sur l'information de type dans une classe du fournisseur en introduisant des sous-classes pour chaque cas de type. Le code conditionnel est remplacé par une seule méthode polymorphique dans une instance de l'une des nouvelles sous-classes.
- Transform Client Type Checks transforme les conditions sur l'information de type dans une classe client en introduisant une nouvelle méthode pour chacune des méthodes des classes du fournisseur. La condition est remplacée par un seul appel polymorphique dans la nouvelle méthode.

- Factor out State traite un cas particulier de Transform Self Type Checks dans lequel l'information de type qui est testée pourrait changer dynamiquement. Un objet State est introduit dans la classe du fournisseur pour présenter l'état de changement, et la condition est remplacée par un appel à une méthode du nouvel objet State.
- Factor our Strategy est un autre cas particulier de Transform Sefl Type Checks dans lequel les algorithmes pour traiter les différents cas du fournisseur sont factorisés en introduisant une nouvelle stratégie. La différence clé avec Factor out State est que l'algorithme plutôt que l'état devrait varier dynamiquement.
- Introduce Null Object aborde le cas particulier de Transform Client Type Checks dans lequel le test accomplit des vérifications que le fournisseur défini ou non. La condition est éliminée par l'introduction d'un objet Null qui implémente le comportement par défaut approprié.
- Transform Conditionals into Registration aborde la situation dans laquelle la condition est responsable du coût d'un outil externe basé sur certains attributs d'un objet à traiter. La solution est d'introduire un service de recherche là où les outils sont déclarés comme des plug-ins. La condition est ensuite remplacée par une simple recherche du plug-in déclaré. La solution est ensuite entièrement dynamique parce que les nouveaux plug-ins peuvent être ajoutés ou supprimés sans aucun changement dans les outils des utilisateurs.

10.1 Transform Self Type Checks

But : Améliorer l'extensibilité d'une classe en remplaçant une déclaration complexe de condition par un appel à une méthode d'accroche implémentée par la sous-classe.

Problème : Une classe est dure à modifier ou à étendre parce qu'elle regroupe de multiples comportements possibles dans des déclarations complexes de conditions qui testent certains attributs représentant le « type » courant de l'objet.

Ce problème est difficile parce que :

- Conceptuellement, des extensions simples requièrent beaucoup de changements dans le code des conditions.
- « Sous-classer » est ensuite impossible sans dupliquer et adapter les méthodes contenant le code des conditions.
- Ajouter un nouveau comportement résulte toujours des changements du même ensemble de méthodes et en ajout un nouveau cas dans le code des conditions.

Cependant, résoudre ce problème est faisable parce que :

- Faire soi-même des vérifications de type simule le polymorphisme. Le code des conditions nous dit quelles sous-classes nous devrions avoir à la place.

Solution **:** Identifier les méthodes avec des branches complexes de conditions. Dans chaque cas, remplacer le code des conditions par un appel à une nouvelle méthode d'accroche. Identifier ou introduire des sous-classes correspondant aux cas des conditions. Dans chacune de ces sous-classes, implémenter la méthode d'accroche avec le code correspondant à ce cas dans la déclaration de cas d'origine.

Détection

La plupart du temps, le discernement du type nous sautera aux yeux pendant que nous travaillons sur le code, donc cela signifie que nous n'aurons pas vraiment besoin de détecter où les vérifications sont faites. Cependant, il peut être intéressant d'avoir de simples techniques pour contrôler si des parties inconnues du système souffrent de pratiques similaires. Ceci pourra être une source précieuse d'information pour évaluer l'état du système.

- Chercher les méthodes longues avec des structures de décision complexes sur certains attributs immuables de l'objet qui présentent l'information du type. En particulier, chercher les attributs qui sont regroupés dans le constructeur et qui ne changent jamais.
- Les attributs qui sont utilisés pour représenter l'information du type prennent typiquement les valeurs de certains types énumérés, ou de certains ensembles finis de valeurs constantes. Cherchez les définitions de constantes dont le nom représente des entités ou concepts pour lesquels nous s'attendions à ce qu'ils soient associés à des classes (comme RetiredEmployee ou PendingOrder). Les conditions compareront seulement la valeur d'un attribut fixé à l'une de ces valeurs constantes.
- Spécialement, il faut regarder les classes où des méthodes multiples sont reliées aux mêmes attributs. C'est un autre signe commun que l'attribut est utilisé pour simuler un type.
- Puisque les méthodes contenant des déclarations de cas ont tendance à être longues, utiliser un outil qui trie les méthodes par lignes de code ou visualise les classes et les méthodes en fonction de leur taille nous aidera. Alternativement, il faudra chercher les classes et les méthodes avec un grand nombre de déclarations de conditions.
- Pour les langages comme C++ ou Java où il est commun de mémoriser l'implémentation d'une classe dans un fichier séparé, il est simple de chercher et de compter le taux de mots clés conditionnels (if, else, case, etc.). Par exemple, dans le système UNIX,

  grep 'switch' `find . --name "*.cxx" --print`

  Énumère tous les fichiers dans un arbre de répertoire avec l'extension .cxx qui contiennent un switch. D'autres outils de traitement de texte comme agrep offrent des possibilités d'affecter de meilleures requêtes de granularité. Les langages de traitement de texte comme Perl pourraient convenir mieux pour évaluer certains types de requêtes, spécialement celles qui couvrent plusieurs lignes.

Etapes **:**

1. Identifier la classe à transformer et les différentes classes conceptuelles qu'elle implémente. Un type d'énumération ou ensemble de constantes documentera probablement ceci bien.
2. Introduire une nouvelle sous-classe pour chaque comportement implémenté comme le montre la figure ci-dessous. Modifier les clients pour instancier la nouvelle sous-classe plutôt que la classe d'origine. Effectuer les tests.
3. Identifier toutes les méthodes de la classe d'origine qui implémentent un comportement variable au moyen de déclarations de conditions. Si les conditions sont entourées par d'autres déclarations, les déplacer pour séparer les méthodes d'accroche protégées. Quand chaque condition occupe l'une de ses méthodes, effectuer les tests.
4. Itérativement, déplacer les cas des conditions vers les sous-classes correspondantes, tout en effectuant les tests périodiquement.
5. Les méthodes qui contiennent du code conditionnel devraient maintenant toutes être vides. Les remplacer par des méthodes abstraites et effectuer les tests.
6. Alternativement, s'il y a des comportements par défaut appropriés, les implémenter à la racine de la nouvelle hiérarchie.
7. Si la logique nécessitait de décider quelle sous-classe à instancier est non triviale, considérer qu'encapsuler cette logique est une méthode facteur de la nouvelle racine de la hiérarchie. Mettre à jour les clients pour utiliser la nouvelle méthode facteur et effectuer les tests.

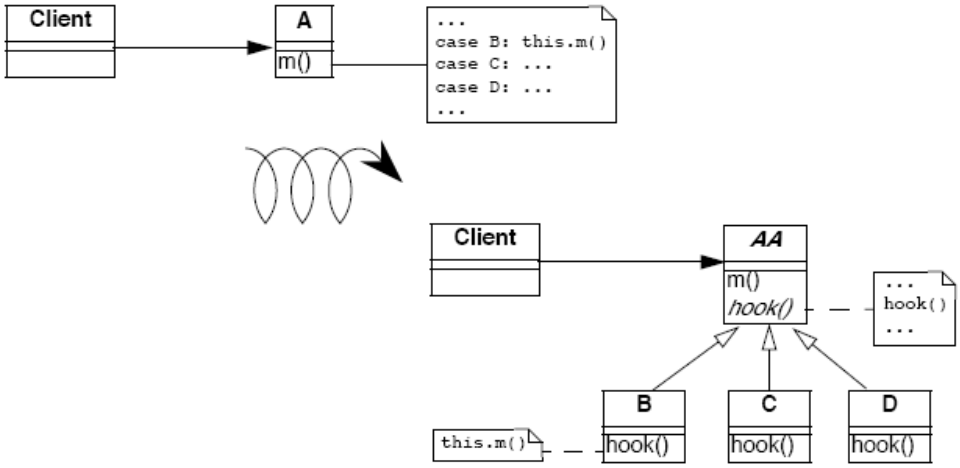


Figure 10.2 : Transformation d'une vérification de type explicite en appels de méthode polymorphiques.

Compromis

Avantages :

- De nouveaux comportements ne peuvent pas être ajoutés de manière incrémentielle, sans avoir à changer un ensemble de méthodes d'une seule classe contenant tout le comportement. Un comportement spécifique peut maintenant être compris indépendamment des autres variantes.
- Un nouveau comportement représente sa donnée indépendamment des autres, et ainsi, minimise l'interférence possible et augmente la compréhension des comportements séparés.
- Tous les comportements partagent maintenant une interface commune, et ainsi, améliorent leur lecture.

Inconvénients :

- Tous les comportements sont maintenant dispersés en abstractions multiples mais liées, donc obtenir un aperçu du comportement pourrait être plus difficile. Cependant, les concepts sont liés d'ailleurs, le fait de partager l'interface représentée par la classe abstraite réduit ensuite le problème.
- Un grand nombre de classes rend le design plus complexe, et potentiellement plus difficile à comprendre. Si les déclarations de conditions d'origine sont simples, ça ne vaut pas la peine d'accomplir cette transformation.
- Des vérifications de type explicites ne sont pas toujours un problème et nous pouvons parfois les tolérer. Créer de nouvelles classes augmente le nombre d'abstractions dans les applications et peut encombrer les espaces de noms. C'est pour cela que les vérifications de type explicites pourraient être une alternative à la création de nouvelles classes quand :
    - L'ensemble sur lequel se fait la sélection de méthode est fixé et n'évoluera pas à l'avenir.
    - La vérification de type est faite seulement dans peu d'endroits.

Difficultés :

- Puisque les sous-classes requises n'existent pas encore, il peut être difficile de dire quand les conditions sont utilisées pour simuler plusieurs types.
- Cependant, des instances de la classe transformée sont créées au départ, maintenant, des instances de sous-classes différentes doivent être créées. Si l'instanciation se produit dans le

code du client, ce code doit maintenant être adapté pour instancier la bonne classe. Des objets de facteur ou méthodes pourraient être nécessaires pour cacher cette complexité au client.

- Si nous n'avons pas accès au code source des clients, il devrait être difficile ou impossible d'appliquer ce pattern puisque nous ne serons pas capables de changer les appels aux constructeurs.
- Si les déclarations de cas testent plus d'un attribut, il devrait être nécessaire de maintenir une hiérarchie plus complexe, probablement nécessitant de l'héritage multiple. Il faudra penser à diviser la classe en différentes parties, chacune avec sa propre hiérarchie.
- Quand la classe contenant les conditions d'origine ne peut être « sous-classée », Transform Self Type Checks peut être composé avec délégation. L'idée est d'exploiter le polymorphisme sur une autre hiérarchie en déplaçant une partie de l'état et du comportement de la classe d'origine vers une classe séparée que la méthode déléguera, comme montré à la figure ci-dessous :

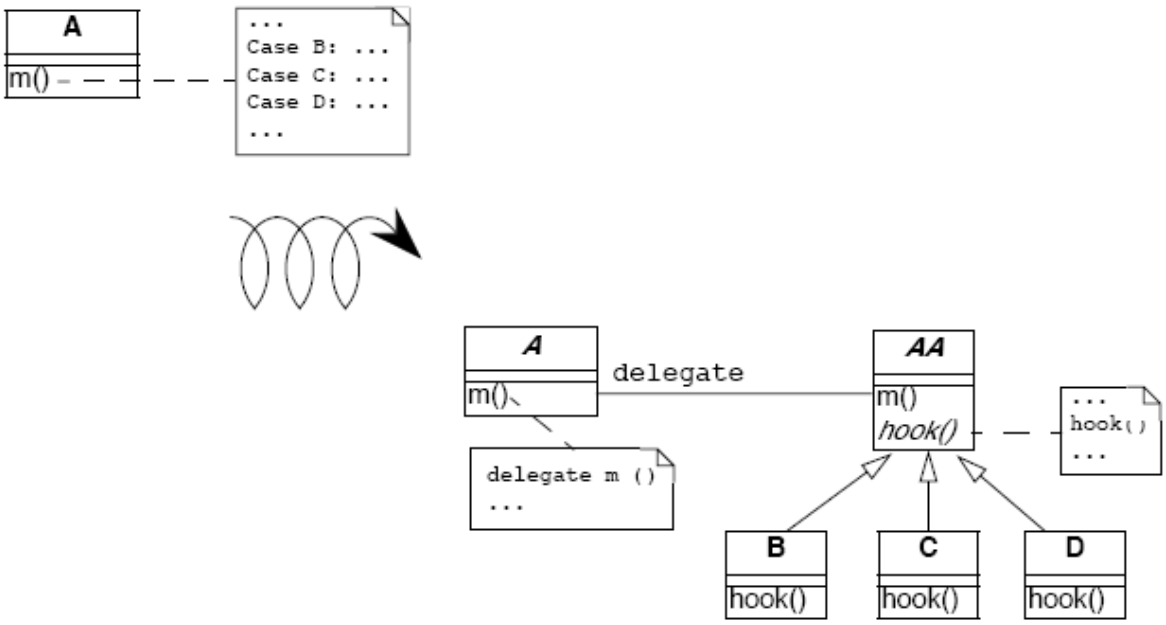


Figure 10.3 : Réunit une simple délégation et Transform Self Type Checks quand la classe ne peut être sous-classée.

Quand l'ancienne solution est la solution :

Il y a certaines situations dans lesquelles les vérifications de type explicites devraient néanmoins être la bonne solution :

- Le code des conditions devrait être produit à partir d'un outil particulier. Les analyseurs syntaxiques et parseurs, par exemple, devraient être automatiquement produits pour contenir le genre de code conditionnel que nous essayons d'éviter. Dans ces cas, les classes produites ne devraient jamais être étendues manuellement, mais simplement produites à nouveau à partir des spécifications modifiées.

Raisonnement

Les classes qui se déguisent en multiples types de données rendent un design plus difficile à comprendre et à étendre. L'utilisation de vérifications explicites de type mène à de longues méthodes qui mélangent plusieurs comportements différents. Introduire un nouveau comportement

nécessite que des changements de toutes ces méthodes soient faits au lieu de spécifier simplement une nouvelle classe représentant le nouveau comportement.

En transformant de telles classes en hiérarchies qui représentent explicitement les multiples types de données, nous améliorons la cohésion en réunissant tout le code concernant un seul type de données, nous éliminons une certaine quantité de code dupliqué (i.e. les tests conditionnels), et nous rendons notre design plus transparent, et par conséquent nous obtiendrons un système plus facile à entretenir.

Patterns liés

Dans Transform Self Type Checks, la condition pour transformer l'information du type des tests qui est représentée comme un attribut de la classe elle-même.

Si la condition teste l'état mutable de l'objet hôte, il faudra examiner au lieu d'appliquer Factor out State, ou peut-être Factor out Strategy.

Si la condition se produit dans un client plutôt que dans la classe fournisseur, il est nécessaire d'envisager d'appliquer Transform Client Type Checks.

Si le code conditionnel teste certains attributs de type d'un second objet pour permettre de sélectionner un troisième objet porteur, il faut examiner au lieu d'appliquer Transform Conditionals into Registration.

10.2 Transform Client Type Checks

But **:** Réduire le rapport client/fournisseur en transformant le code conditionnel qui teste le type du fournisseur dans un appel polymorphique à une nouvelle méthode de fournisseur.

Problème **:** Comment réduire le rapport entre clients et fournisseurs de services, où les clients vérifient explicitement le type des fournisseurs et ont la responsabilité de composer le code des fournisseurs ?

Ce problème est difficile parce que :

- Ajouter une nouvelle sous-classe à la hiérarchie du fournisseur nécessite d'effectuer des changements dans plusieurs clients, spécialement où les tests ont lieu.
- Clients et fournisseurs auront tendance à être solidement associés, puisque les clients accomplissent des actions qui devraient être la responsabilité des fournisseurs.

Cependant, résoudre ce problème est faisable parce que :

- Les conditions nous disent à quelles classes nous devrions changer le comportement.

Solution **:** Introduire une nouvelle méthode dans la hiérarchie du fournisseur. Implémenter la nouvelle méthode dans chaque sous-classe de la hiérarchie du fournisseur en déplaçant le cas correspondant des clients dans cette classe. Remplacer la condition entière par un appel simple à la nouvelle méthode.

Détection

Appliquer essentiellement les mêmes techniques décrites dans Transform Self Type Checks pour détecter les déclarations de cas, mais chercher des conditions qui testent le type d'un fournisseur de service séparé qui implémente déjà une hiérarchie. Nous devrions aussi chercher des déclarations de cas se trouvant dans différents clients dans la même hiérarchie de fournisseur.

- Java : Chercher des applications de l'opérateur instanceof, avec l'adhésion des tests d'un objet dans une classe connue, spécifique. Bien que les classes en Java ne soient pas des objets comme en Smalltalk, chaque classe qui est chargée dans la machine virtuelle est représentée par une seule instance de java.lang.Class. Il est ensuite possible de déterminer si deux objets, x et y appartiennent à la même classe en exécutant le test :

x.getClass() = y.getClass()

Alternativement, l'hadhésion de la classe devrait être testée en comparant les noms de classes :

x.getClass().getName().equals(y.getClass().getName())

Etapes :

1. Identifier les clients accomplissant des vérifications de type explicites.
2. Ajouter une nouvelle méthode vide à la racine de la hiérarchie du fournisseur représentant l'action exécutée dans le code conditionnel (voir Figure 10.6)
3. Itérativement, déplacer un cas de la condition vers certaines classes du fournisseur, en le remplaçant par un appel à cette méthode. Après chaque déplacement, les tests de régression devraient s'effectuer.
4. Quand toutes les méthodes ont été déplacées, chaque cas de la condition consiste en un appel à la nouvelle méthode, donc ceci remplace la condition entière par un seul appel à la nouvelle méthode.
5. Il faudra envisager de faire la méthode abstraite dans la racine du fournisseur. Alternativement implémenter le comportement par défaut adéquat ici.

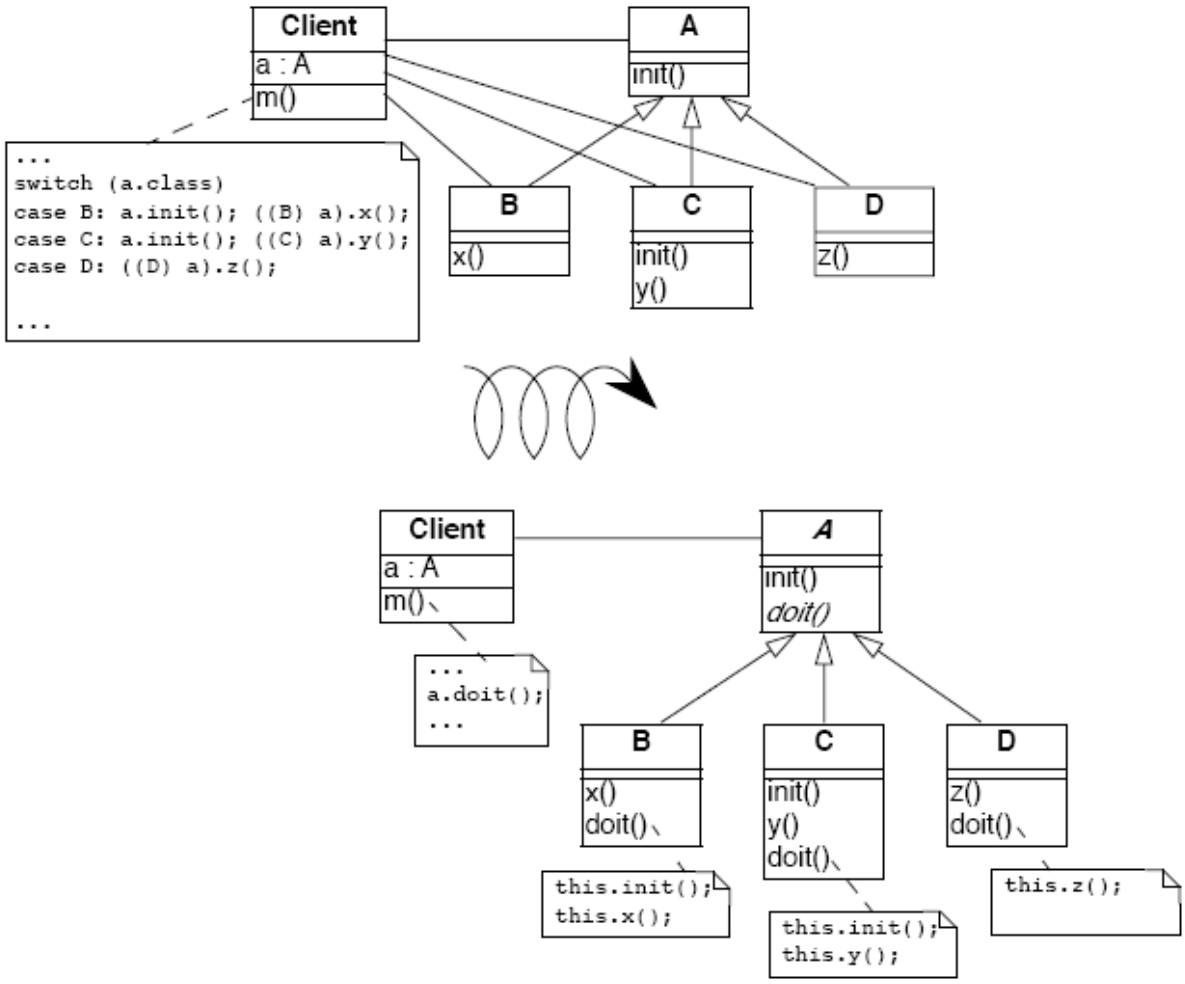


Figure 10.6 : Transformation d'une vérification explicite de type utilisée pour déterminer quelles méthodes d'un client devraient être invoquées à travers des appels de méthodes polymorphiques.

Autres étapes à considérer

- Il pourrait être bien que des clients multiples accomplissent exactement le même test et prennent les mêmes actions. Dans ce cas, le code dupliqué peut être remplacé par un seul appel de méthode après que l'un des clients ait été transformé. Si des clients accomplissent des tests différents ou prennent différentes actions, alors le pattern doit être appliqué une fois pour chaque condition.
- Si la déclaration de cas ne couvre pas toutes les classes concrètes de la hiérarchie du fournisseur, une nouvelle classe abstraite pourrait nécessiter d'être introduite comme une super-classe commune des classes concernées. La nouvelle méthode sera ensuite introduite seulement pour le sous-arbre approprié. Alternativement, s'il n'est pas possible d'introduire une telle classe étant donné la hiérarchie d'héritage existante, il faudrait envisager d'implémenter la méthode à la racine avec soit une implémentation par défaut vide, soit une qui provoque une exception si elle est appelée pour une classe inappropriée.
- Si les conditions sont « nested », le pattern pourrait nécessiter d'être appliqué récursivement.

Compromis

Avantages :

- La hiérarchie du fournisseur offre un nouveau service polymorphique disponible aussi pour les autres clients.
- Le code des clients est maintenant mieux organisé et n'a plus à traiter des affaires qui sont maintenant sous la responsabilité du fournisseur.
- Tout le code concernant le comportement d'un seul fournisseur est maintenant réuni dans un seul endroit.
- Le fait que la hiérarchie du fournisseur offre une interface uniforme permet aux fournisseurs d'être modifiés sans avoir un impact sur les clients.

Inconvénients :

- Quelquefois, il est commode de voir le code traiter différents cas dans un seul endroit. Transform Client Type Checks redistribue la logique aux classes individuelles du client, avec comme résultat la perte de l'aperçu.

Difficultés :

- Normalement, les instances des classes fournisseurs devraient être créées dans un premier temps donc nous n'avons pas à se préoccuper de cette création, cependant « refactorer » l'interface affectera tous les clients des classes du fournisseur et ne devra pas être entrepris sans considérer les pleines conséquences d'une telle action.

Quand l'ancienne solution est notre solution de base

Les vérifications de type d'un client pourraient néanmoins être la bonne solution quand l'instance du fournisseur n'existe pas encore, ou quand sa classe ne peut être étendue :

- Un objet Abstract Factory pourrait nécessiter le test d'une variable de type dans le but de savoir quelle classe instancier. Par exemple, un facteur pourrait entrer des objets à partir de la représentation d'un fichier texte, et tester certaines variables qui lui diront à quelle classe l'objet entré devrait appartenir.
- Un logiciel qui s'adapte à une librairie non orientée objet, comme une ancienne librairie GUI, pourrait forcer le développeur à simuler l'expédition manuellement. C'est discutable vu qu'il est couteux de développer une façade orientée objet pour la librairie de procédure.

- Si la hiérarchie du fournisseur est bloquée (e.g., parce que le code source n'est pas disponible), alors il ne sera pas possible de transférer le comportement aux classes du fournisseur. Dans ce cas, des classes d'emballage pourraient être définies pour étendre le comportement des classes du fournisseur, mais la complexité (ajoutée) de la définition des emballages pourrait écraser tous les avantages.

Raisonnement **:** "D'habitude, une analyse de cas explicite sur le type d'un objet est une erreur. Le designer devrait utiliser le polymorphisme dans la plupart de ces cas ». En effet, des vérifications explicites de type dans les clients sont un signe de responsabilités mal placées puisqu'elles augmentent les rapports entre clients et fournisseurs. Passer ces responsabilités aux fournisseurs aura les conséquences suivantes :

- Le client et le fournisseur seront reliés plus faiblement puisque le client nécessitera seulement la connaissance de la racine de la hiérarchie du fournisseur au lieu de toutes ses sous-classes concrètes.
- La hiérarchie du fournisseur pourrait évoluer plus gracieusement, avec moins de chances de casser le code du client.
- La taille et la complexité du code du client est réduite. Les collaborations entre clients et fournisseurs deviennent plus abstraites.
- Les attractions implicites dans l'ancien design (i.e., les actions des cas conditionnels) seront rendues explicites comme les méthodes, et seront disponibles aux autres clients.
- La duplication du code pourrait être réduite (si les mêmes conditions se trouvent multipliées).

Patterns liés **:** Dans Transform Client Type Checks, la condition est faite sur l'information de type d'une classe fournisseur. La même situation se produit dans Introduce Null Object où la condition teste sur une valeur nulle avant d'invoquer la méthode. De ce point de vue, Introduce Null Object est une spécialisation de Transform Client Type Checks.

Transform Conditionals into Registration traite le cas particulier dans lequel la condition du client est utilisée pour sélectionner un troisième objet (typiquement une application ou outil externe) pour traiter l'argument.

Replace Conditional with Polymorphism est le refactoring cœur de ce pattern de réingénierie, donc le lecteur pourrait se référer aux étapes décrites en [FBB+99].

10.3 Factor out State

But : Eliminer le code conditionnel complexe sur l'état d'un objet en appliquant le design pattern State.

Problème **:** Comment rendre une classe dont le comportement dépend d'une évaluation complexe de son état courant, plus extensible ?

Ce problème est difficile parce qu' :

- Il y a plusieurs déclarations conditionnelles complexes répandues sur les méthodes de l'objet. Ajouter un nouveau comportement pourrait affecter ces conditions de façons subtiles.
- A chaque fois que de nouveaux états possibles sont introduits, toutes les méthodes qui testent un état doivent être modifiées.

Cependant, résoudre ce problème est faisable parce que :

- Les variables d'instance de l'objet sont typiquement utilisées pour présenter différents états abstraits, chacun ayant son propre comportement. Si nous pouvons identifier ces états abstraits, nous pouvons faire du factoring sur l'état et le comportement à travers un ensemble de classes reliées.

Solution : Appliquer le pattern State, i.e., encapsuler le comportement « state-dépendant »dans des objets séparés, déléguer des appels à ces objets et garder l'état de l'objet cohérent en consultant la bonne instance de ces objets état (voir figure 47).

Comme dans Transform Self Type Checks, transformer le code conditionnel complexe qui teste sur les états quantifiés à travers des appels délégués aux classes d'état. Appliquer le pattern State, en déléguant chaque cas conditionnel à un objet State séparé.

Etapes

1. Identifier l'interface d'un état et le nombre d'états.
   Si nous sommes chanceux, chaque condition partitionnera l'espace de l'état de la même façon, et le nombre d'états sera égal au nombre de cas dans chaque condition. Dans le cas où la condition se recoupe, un partitionnement plus fin sera requis.
   L'interface d'un état dépend de comment l'information de l'état est accédée et mise à jour, et pourrait nécessiter d'être raffinée en sous-étapes.
2. Créer une nouvelle classe abstraite, State, représentant l'interface de l'état.
3. Créer une nouvelle sous-classe de State pour chaque état.
4. Définir des méthodes de l'interface des classes d'état identifiées à l'étape 1 en copiant le code correspondant de la condition dans la nouvelle méthode. Sans oublier de changer l'état de la variable d'instance dans le Contexte pour faire référence à la bonne instance de la classe State. Les méthodes de State ont la responsabilité de changer le Contexte, c'est pourquoi il fait toujours référence à l'instance d'état suivante.
5. Ajouter une nouvelle variable d'instance dans la classe Contexte.
6. Nous obtiendrons une référence de State dans la classe Contexte pour invoquer les transitions d'état des classes State.
7. Il faudra ensuite initialiser l'instance nouvellement créée pour faire référence à une instance par défaut de la classe d'état.
8. Modifier les méthodes de la classe Contexte contenant les tests pour déléguer l'appel à la variable d'instance.

L'étape 4 peut fonctionner en utilisant l'opération Extract Method (Méthode d'extraction) du Refactoring Browser (navigateur de Refactoring). Il faut noter qu'après chaque étape, les tests de régression devraient toujours fonctionner. L'étape critique est la dernière, dans laquelle un comportement est délégué aux nouveaux objets State.

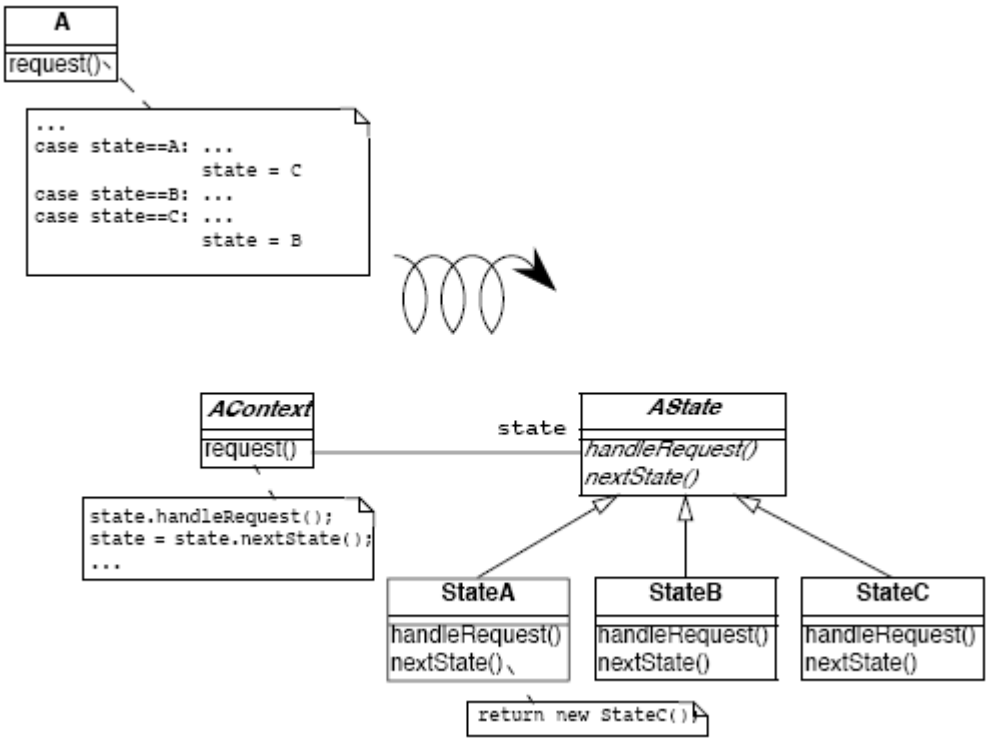


Figure 10.8 : Transformation pour aller d'un pattern state simulé utilisant une condition d'état explicite, vers une situation où le pattern state a été appliqué.

Compromis

Avantages :

- Un impact limité. L'interface publique de la classe d'origine n'a pas à changer. Puisque les instances de State sont accédées par délégation à partir de l'objet d'origine ; les clients ne peuvent pas être affectés. Dans le cas le plus simple, l'application de ce pattern a un impact limité sur les clients.

Inconvénients :

- L'application systématique de ce pattern pourrait mener à une explosion du nombre de classes.
- Ce pattern ne devrait pas être appliqué quand :
  - Il y a trop d'états possibles, ou le nombre d'états n'est pas fixé.
  - Il est difficile de déterminer à partir du code comment les transitions d'état se produisent.

Quand l'ancienne solution est la solution de base

Ce pattern ne doit pas être appliqué à la légère.

- Quand les états sont clairement identifiés et que nous savons qu'ils ne changeront pas, l'ancienne solution a l'avantage de regrouper tout le comportement d'un état par fonctionnalité au lieu de le répandre sur différentes sous-classes.
- Dans certains domaines, tels que les parseurs, le comportement table-driven, codés comme des conditions sur un état, sont mieux compris, et faire du factoring sur les objets d'états pourrait juste rendre le code plus difficile à comprendre, et par conséquent, à maintenir.

Utilisations connues : Le Design Pattern Smalltalk Companion présente une transformation de code étape par étape.

10.4 Factor out Strategy

But : Eliminer le code conditionnel qui sélectionne un algorithme approprié en appliquant le design pattern Strategy.

Problème : Comment rendre une classe dont le comportement dépend du test de la valeur de certaines variables, plus extensible ?

Ce problème est difficile parce que :

- Une nouvelle fonctionnalité ne peut être ajoutée sans modifier toutes les méthodes contenant le code conditionnel.
- Le code conditionnel pourrait être répandu sur plusieurs classes qui prennent des décisions similaires pour savoir quel algorithme appliquer.

Cependant, résoudre ce problème est faisable parce que :

- Les comportements alternatifs sont essentiellement interchangeables.

Solution : Appliquer le pattern Strategy, i.e., encapsuler le comportement algorithmique dépendant dans des objets séparés avec des interfaces polymorphiques et déléguer des appels à ces objets (voir Figure 10.9).

Etapes :

1. Identifier l'interface de la classe Strategy.
2. Créer une nouvelle classe abstraite, Strategy, représentant l'interface des stratégies.
3. Créer une nouvelle sous-classe de Strategy pour chaque algorithme identifié.
4. Définir des méthodes de l'interface identifiée dans l'étape 1 dans chacune des classes de stratégie en copiant le code correspondant au test dans la méthode.
5. Ajouter une nouvelle variable d'instance dans la classe Context pour faire référence à la stratégie courante.

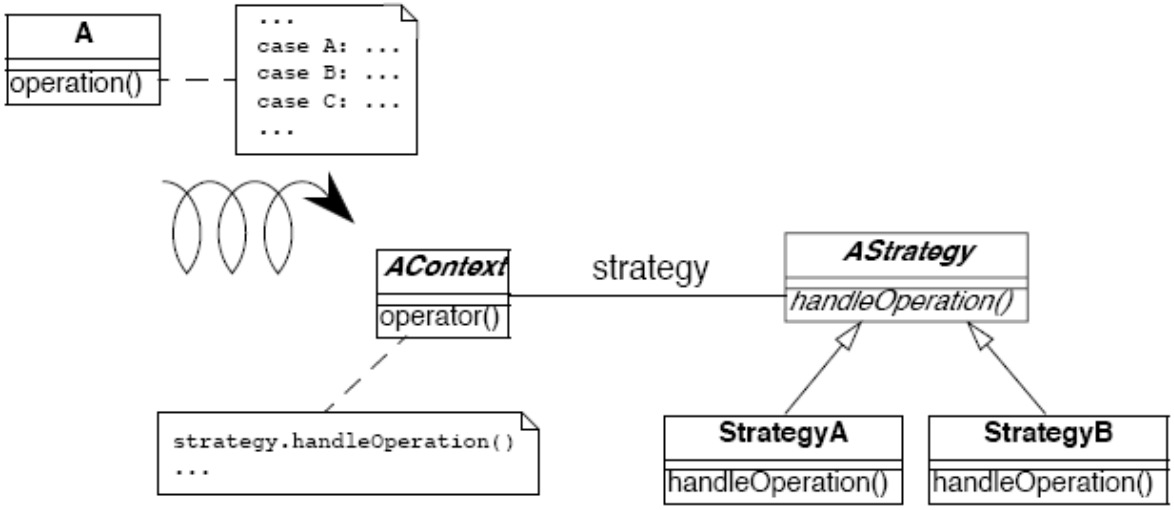


Figure 10.9 : Transformation pour aller d'un pattern state simulé utilisant une condition d'état explicite, vers une situation où le pattern state a été appliqué.

6. Nous sommes amenés à avoir une référence de Strategy vers la classe Context pour assurer un accès à l'information maintenue par le Contexte (Voir les difficultés).
7. Initialiser l'instance nouvellement créée pour faire référence à une instance par défaut de Strategy.
8. Changer les méthodes de la classe Context contenant les tests en éliminant les tests et en déléguant l'appel à la variable d'instance.

L'étape 4 peut fonctionner en utilisant l'opération Extract Method (Méthode d'extraction) du Refactoring Browser (navigateur de Refactoring). On notera qu'après chaque étape, les tests de régression devraient toujours fonctionner. L'étape critique est la dernière, dans laquelle un comportement est délégué aux nouveaux objets Strategy.

Compromis

Avantages :

- Un impact limité. L'interface publique de la classe d'origine n'a pas à changer. Puisque les instances de Strategy sont accédées par délégation à partir de l'objet d'origine, les clients ne peuvent pas être affectés. Dans le cas le plus simple, l'application de ce pattern a un impact limité sur les clients. Cependant, l'interface Context sera réduite parce que tous les algorithmes précédemment implémentés sont maintenant déplacés vers les classes Strategy. Donc nous devons vérifier les invocations de ces méthodes et nous décider à opter pour une base par cas.
- Après avoir appliqué ce pattern, nous devons être capables de créer de nouvelles stratégies sans pour autant modifier l'interface de Context. Ajouter une nouvelle stratégie ne nécessite pas de recompiler la classe Context et ses clients.
- Après avoir appliqué ce pattern, l'interface de la classe Context et les classes Strategy seront plus claires.

Inconvénients :

- L'application systématique de ce pattern pourrait conduire à une explosion. Si
- nous avons 20 algorithmes différents nous pourrions ne pas vouloir avoir 20 nouvelles classes, chacune avec seulement une méthode.
- Explosion d'objet. Les stratégies augmentent le nombre d'instances dans une application.

Difficultés :

- Il y a plusieurs façons de partager une information entre les objets Context et Strategy, et les compromis peuvent être subtils. L'information peut être passée en argument quand la méthode Strategy est invoquée, l'objet Context lui-même peut être passé en argument, ou les objets Strategy peuvent avoir une référence de leur contexte. Si la relation entre le Context et la Strategy est très dynamique, il pourrait alors être préférable de passer cette information comme un argument de méthode.

Exemple : Le Design Pattern Smalltalk Companion présente une transformation de code étape par étape.

Patterns liés : Les symptômes et la structure de Factor out Strategy donnent une comparaison avec Factor out State. La principale différence consiste dans le fait que Factor out State identifie un comportement avec différents états possibles d'objets tandis que Factor out Strategy concerne des algorithmes interchangeables qui sont indépendants d'un objet state. Factor out Strategy permet d'ajouter de nouvelles stratégies sans impact sur les objets Strategy existants.

10.5 Introduce Null Object

But : Eliminer le code qui teste des valeurs nulles en appliquant le design pattern Null Object.

Problème : Comment améliorer la modification et l'extension d'une classe en présence de tests répétés pour des valeurs nulles ?

Ce problème est difficile parce que :

- Les méthodes du client testent toujours si certaines valeurs sont non nulles avant de vraiment invoquer leurs méthodes.
- Ajouter une nouvelle sous-classe à la hiérarchie du client nécessite de tester des valeurs nulles avant d'invoquer certaines méthodes du fournisseur.

Cependant, résoudre ce problème est faisable parce que :

- Le client n'a pas besoin de savoir si le fournisseur représente une valeur nulle.

Solution : Appliquer le pattern Null Object, i.e. encapsuler le comportement null comme une classe séparée de fournisseur, ainsi, la classe du client n'a pas à exécuter un test null.

Détection : Chercher des tests idiomatiques null.

Des tests null pourraient prendre différentes formes, en fonction du langage de programmation et du genre d'entité à tester. En Java, par exemple, une référence d'objet null a la valeur null, tandis qu'en C++, un pointeur d'objet null a la valeur 0.

Etapes : Fowler discute en détails les étapes nécessaires du refactoring [FBB+99].

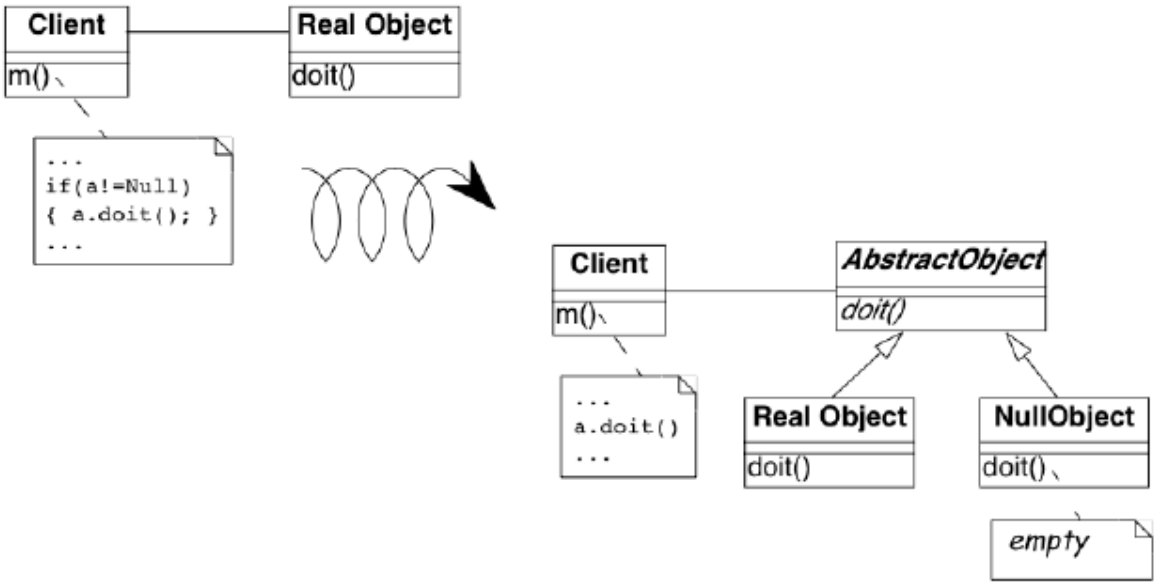


Figure 10.10 : Transformation d'une situation basée sur un test explicite de valeur null en une situation où un Null Object est introduit.

1. Identifier l'interface requise pour le comportement null. (Elle sera normalement identique à celle de l'objet non null.)
2. Créer une nouvelle super classe abstraite comme une super classe de la classe RealObject.
3. Créer une nouvelle sous-classe de la super classe abstraite avec un nom commençant par No ou Null.
4. Définir des méthodes par défaut dans la classe Null Object.
5. Initialiser la variable d'instance ou structure qui est vérifiée pour maintenant avoir au moins une instance de la classe Null Object.
6. Supprimer les tests conditionnels du client.

Si nous voulons encore être capable de tester des valeurs null proprement, nous pourrions introduire une méthode de requête appelée isNull dans les classes RealObject et Null Object, comme décrit par Fowler [FBB+99].

Compromis

Avantages :

- Le code client est plus simple après avoir appliqué le pattern.
- Le pattern est relativement simple à appliquer puisque l'interface du fournisseur n'a pas à être modifiée.

Inconvénients :

- La hiérarchie du fournisseur devient plus complexe.

Difficultés :

- Plusieurs clients pourraient ne pas être d'accord avec le comportement raisonnable par défaut de Null Object. Dans ce cas, plusieurs classes Null Object pourraient nécessiter d'être définies.

Quand l'ancienne solution est la solution de base

- Si les clients ne se mettent pas d'accord sur une interface commune.
- Quand un très petit code utilise la variable directement ou quand le code qui utilise la variable est bien encapsulé dans un seul endroit.

10.6 Transform Conditional into Registration

But: Améliorer la modularité d'un système en remplaçant les conditions dans les clients par un mécanisme d'enregistrement.

Problème : Comment réduire le rapport entre des outils qui fournissent des services et des clients, de façon à ce que l'ajout ou la suppression d'outils ne mène pas à changer le code des clients ?
Ce problème est difficiles parce que :

- Avoir un seul endroit pour chercher tous les types d'outils rend la compréhension du système et l'ajout de nouveaux outils plus faciles.
- Cependant, à chaque fois que nous supprimions un outil, nous devons supprimer un cas dans certaines déclarations conditionnelles, sinon, certaines parties (clients d'un outil) refléteront toujours la présence des outils supprimés, conduisant à des systèmes fragiles. Ensuite, à chaque fois que nous ajoutons un nouvel outil, nous devons ajouter une nouvelle condition dans tous les clients de l'outil.

Cependant, résoudre ce problème est faisable parce que :

- De longues conditions rendent l'identification des différents types d'outils utilisés plus facile.

Solution : Introduire un mécanisme d'enregistrement pour lequel chaque outil est responsable de l'enregistrement lui-même, et transformer les clients d'outil pour vérifier le dépôt d'enregistrement au lieu d'exécuter les conditions.

Etapes

1. Définir une classe décrivant des objets plug-in, i.e., un objet encapsulant l'information nécessaire pour enregistrer un outil. Bien que la structure de l'interface de cette classe dépende du but de l'enregistrement, un plug-in devrait fournir l'information nécessaire pour que le manager d'outil puisse l'identifier, créer une instance de l'outil représenté et invoquer des méthodes. Pour invoquer une méthode d'outil, une méthode ou un mécanisme similaire comme une fermeture de bloc ou une classe interne pourraient être mémorisés dans l'objet plug-in.

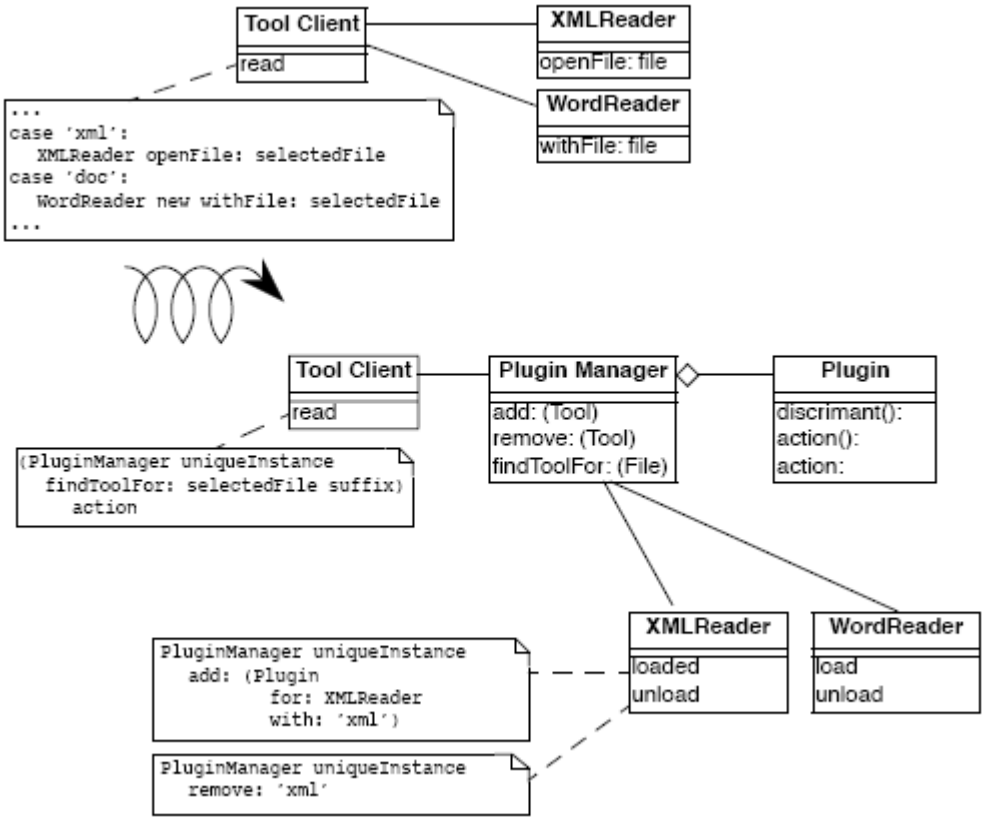


Figure 10.11 : Transformer des conditions en utilisateurs d'outils en introduisant un mécanisme d'enregistrement

2. Définir une classe représentant le plug-in manager, i.e., qui gère les objets du plug-in et qui sera vérifiée par les clients d'un outil pour vérifier la présence d'outils. Cette classe sera certainement un singleton puisque les plug-ins représentant les outils disponibles ne devraient pas être perdus si une nouvelle instance du manager du plug-in est créée.

3. Pour chaque cas de la condition, définir un objet plug-in associé avec l'outil donné. Cet objet plug-in devrait être créé et enregistré automatiquement quand l'outil qu'il représente est chargé, et il ne devrait pas être enregistré si et quand l'outil devient indisponible. Parfois, l'information du client d'outil devrait être passée à l'outil. Le client d'outil courant peut être passé en argument quand l'outil est invoqué.
4. Transformer l'entière expression conditionnelle dans une requête de l'objet manager d'outil. Cette requête devrait retourner un outil associé à la requête et l'invoquer pour accéder à la fonctionnalité souhaitée.
5. Supprimer toutes les actions du client d'outil qui active directement des outils. Ce comportement est maintenant de la responsabilité du manager du plug-in.

Le client ou l'objet du plug-in pourraient avoir la responsabilité d'invoquer un outil. Il est préférable de laisser l'objet du plug-in avoir cette responsabilité parce qu'il a déjà la responsabilité de savoir comment représenter les outils et seulement laisser les clients dire qu'ils ont besoin d'une action d'outil.

Compromis

Avantages:

- En appliquant Transform Conditionals into Registration nous obtiendrons un système qui est à la fois dynamique et flexible. De nouveaux outils peuvent être ajoutés sans impact sur les outils client.
- Les clients d'outil n'ont plus à vérifier si un outil donné est disponible. Le mécanisme d'enregistrement nous assure que l'action peut être exécutée.
- Le protocole d'interaction entre des outils et des clients d'outils est maintenant régularisé.

Inconvénients :

- Nous devons définir deux nouvelles classes, l'une pour l'objet représentant la représentation (plug-in) de l'outil et l'autre pour l'objet gérant les outils enregistrés (manager de plug-in).

Difficultés :

- Pendant que nous transformons une branche de la condition en objet du plug-in, nous devrions définir l'action associée aux outils via l'objet du plug-in. Pour assurer une séparation claire et un enregistrement dynamique complet, cette action devrait être définie dans l'outil et plus dans le client d'un outil. Cependant, comme l'outil pourrait avoir besoin de certaines informations du client d'un autre outil, ce client devrait être passé à l'outil comme paramètre quand l'action est invoquée. Ceci modifie le protocole entre l'outil et le client d'un outil à partir d'un seul appel sur le client d'un outil dans une invocation de méthode dans l'outil avec un paramètre supplémentaire. Ceci implique aussi que dans certains cas, la classe du client d'un outil doit définir de nouvelles méthodes public ou friend pour permettre aux outils d'accéder à la bonne information du client d'un outil.
- Si chaque branche conditionnelle est seulement associée à un seul outil, seule un objet de plug-in est nécessaire. Cependant, si le même outil peut être appelé de différentes façons, nous devons créer plusieurs objets de plug-in.

Quand l'ancienne solution est la solution de base

- S'il y a une seule classe de client d'outil, si tous les outils sont toujours disponibles, et si nous n'ajoutions ou supprimions jamais un outil en état d'exécution, une condition est plus simple.

Patterns liés : Transform Conditionals into Registration et Transform Client Type Checks éliminent des expressions conditionnelles qui décident quelle méthode devrait être invoquée sur quel objet. La différence entre les deux patterns est que Transform Client Type Checks déplace un comportement d'un client vers le fournisseur de service, tandis que Transform Conditionals into Registration traite un comportement qui ne peut être déplacé parce qu'il est implémenté par un outil externe.