UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR
STÉPHANE VELOU BLÉ

TESTS D'INTÉGRATION DES CLASSES DANS LES SYSTÈMES ORIENTÉS OBJET : UNE ÉVALUATION EXPÉRIMENTALE

MAI 2005

TESTS D'INTÉGRATION DES CLASSES DANS LES SYSTÈMES ORIENTÉS OBJET : UNE ÉVALUATION EXPÉRIMENTALE

Sommaire

Dans les systèmes orientés objet, les objets interagissent pour implémenter le comportement. Une des principales difficultés rencontrées lors du processus de test d'intégration de tels systèmes est la minimisation des bouchons de test et la détermination d'un ordre effectif d'intégration des classes. La présence de cycles de dépendances entre les classes complique encore plus ces tâches. Nous présentons, dans ce mémoire, une nouvelle stratégie d'intégration des classes basée sur un nouveau Modèle de Dépendances entre les Classes (***MDC***). Le modèle MDC prend en compte toutes les interactions entre les classes. Afin de valider notre approche et de la comparer aux stratégies existantes, nous avons mené une étude expérimentale sur plusieurs applications réelles Java. Les résultats obtenus démontrent que la nouvelle stratégie réduit considérablement le nombre de bouchons de test.

OBJECT-ORIENTED INTEGRATION TESTING:

AN EXPERIMENTAL EVALUATION

Abstract

Objects interact in order to implement behavior. One important problem when integrating and testing object-oriented software is to reduce the number of required test stubs and to determine an effective class integration order. The strong connectivity between classes complicates this task. We present a new class integration testing strategy based on a new Class Dependency Model (CDM). The CDM model takes into account the interactions between classes. In order to validate our approach and to compare it to some of the existing object-oriented integration strategies, we conducted an experimental study on several real-world Java programs. The obtained results show that the strategy we propose reduce considerably the number of required test stubs.

DÉDICACES

Je dédie ce mémoire

À toute ma famille, en particulier à mon père Blé Youty Charles,

pour tout le soutien et les encouragements qu'ils m'ont apporté.

À mes chères grand mère Yênet et Nimpou pour leur amour inépuisable.

À tous mes amis qui de près ou de loin ont su m'encourager et me supporter tout au long de cette période très difficile.

À mes directeurs de recherche Linda Badri et Mourad Badri qui n'ont à aucun instant baissé les bras.

REMERCIEMENTS

Je remercie tout d'abord Dieu Tout-Puissant pour sa miséricorde, son amour pour tous ces enfants.

Je remercie particulièrement Linda Badri et Mourad Badri pour leur immense générosité, leur encadrement ainsi que leur disponibilité.

Je remercie tous les membres du Département de mathématiques et informatique pour avoir contribué de près ou de loin à la réalisation de ce mémoire .

Pour finir, j'aimerai remercier toute ma famille et tous mes amis pour leur support durant toute cette période .

TABLE DES MATIÈRES

LISTE DES FIGURES

TABLE DES TABLEAUX

CHAPITRE

1

INTRODUCTION AU TEST DE LOGICIELS

1.1 Introduction

La place du logiciel est de plus en plus importante dans notre vie quotidienne. Les domaines de son application sont très variés : domaine médical, financier, militaire, académique, etc. Il existe plusieurs approches pour le développement de logiciels.

Nous nous intéressons, dans ce document, à l'approche orientée objet. Le développement orienté objet est un processus de modélisation de problèmes ayant pour objectif, l'abstraction en vue de réduire la complexité et la modularité dans le but de faciliter les modifications et la réutilisation .

Le paradigme objet a introduit de nouveaux concepts qui sont, entre autres, les classes, l'encapsulation, le polymorphisme et l'héritage. L'apport de l'approche orientée objet est bénéfique dans tout le cycle de développement du logiciel, de la phase d'analyse à la phase d'implémentation en passant par la phase de conception. Dans la dernière décennie, la programmation orientée objet est devenue l'une des principales technologies à faire face aux exigences toujours croissantes relatives aux fonctionnalités et la qualité des logiciels. En ce qui concerne la qualité du logiciel, certaines croyances selon [Bei90], ne trouvaient pas nécessaire l'élaboration de tests spécifiques liés à la méthodologie orientée objet. Par ailleurs, plusieurs études ont démontré la nécessité du test dans le contexte des applications orientées objet et surtout le développement de techniques de test spécifiques à la méthodologie orientée objet. Robert V. Binder dans [Bin94], indique que bien qu'il y ait des similitudes avec les tests conventionnels, le test orienté objet présente des différences significatives. Ces différences sont dues en grande partie aux concepts introduits par le paradigme orienté objet. Ce sont, entre autres, les concepts d'héritage, d'encapsulation et de polymorphisme dont Binder [Bin94], débat de leur impact sur le test.

1.2 Le test

Le test constitue un élément important dans la vie du logiciel. Pour atteindre un certaine niveau de qualité, le test du logiciel est nécessaire et indissociable du cycle de développement. La sensibilité des domaines d'application et la complexité des

applications ainsi que leur envergure font que le test occupe une place de plus en plus grande dans le cycle de vie du logiciel. Selon Beize [Bei90], l'activité de test occupe quasiment 40 à 70 % de l'effort de développement.

Le test et la qualité du logiciel sont étroitement liés. Le test représente une activité importante de l'assurance qualité. Le test fait partie du processus de vérification [Lab97]. C'est un processus qui permet de trouver les erreurs des programmeurs et des défauts de fonctionnalité ou d'implémentation. Il permet aussi d'avoir confiance au fait que l'implémentation correspond aux spécifications, bien qu'il ne puisse pas assurer l'exactitude du programme. Il consiste à exécuter le programme avec des données en entrée (proches des données réelles) et d'observer les résultats pour en déduire l'existence d'erreurs. En pratique, il est impossible de satisfaire tout le domaine de données [Lab97], ce qui reviendrait à un test exhaustif. Le testeur doit alors choisir un sous-ensemble du domaine de test (en entrée) qui est bien adapté afin d'indiquer de vrais, mais d'inconnus, défauts du logiciel [Lab97].

Le processus entier du test repose sur le concept de cas de test. Un cas de test est l'ensemble des données (données en entrées et résultats escomptés). Il inclut aussi les pré-conditions ainsi que les post-conditions.

La sélection du sous-ensemble du domaine de test (et par conséquent du cas de test) est guidée par des critères de test basés soit sur la structure du programme (test structurel) ou sur les fonctionnalités du programme (test fonctionnel) . Cela permet de dégager deux grandes approches de test, l'approche fonctionnelle et l'approche structurelle.

1.3 Approches fonctionnelles

Le test fonctionnel se base sur les spécifications (en opposition au code source) pour développer les cas de test. Il est aussi appelé test de « boîte noire ». Ici, les spécifications sont utilisées pour produire les cas de test (tout comme pour développer le programme). Les spécifications peuvent donc jouer deux grands rôles dans le test de logiciels. D'une part, elles fournissent les informations nécessaires pour vérifier les résultats du programme. D'autre part, elles fournissent les informations pour la sélection des cas de test et la mesure adéquate du test. Le test fonctionnel offre beaucoup d'avantages au test de logiciels. L'un de ces avantages est l'indépendance du test vis-à-vis de l'implémentation. Cet avantage permet aux ingénieurs de test d'utiliser les mêmes cas de test lors des tests de régression (modification du système, ajout de nouvelles fonctionnalités, etc.). L'avantage principal du test fonctionnel réside dans la possibilité de produire des cas de test tôt dans le processus de développement du logiciel. Cela peut se faire avant même que l'implémentation soit complétée. Les cas de test peuvent être produits en parallèle avec l'implémentation. Cependant, le test fonctionnel ne peut (et ne doit) pas exprimer tous les détails de l'implémentation. L'analyse du code est donc nécessaire.

1.4 Le test structurel

Le test structurel, dit aussi test de « boîte blanche », se base sur l'analyse du code source pour dériver les cas de test. Le but du test structurel est de trouver les erreurs dans le code en suivant les chemins possibles du programme. En partant d'une donnée spécifique en entrée, à un résultat en sortie. Il existe deux variantes dans le test de boîte blanche. Ce sont l' « *interface-based testing* » et la « *method-based testing* ». « L'interface-based testing » est souvent (injustement) considéré comme un test fonctionnel car il repose sur le développement de cas de test basé sur des composantes visibles des classes [Bin99], à savoir des exceptions et des messages (implémentés). En utilisant le test structurel, l'ingénieur de test peut garantir que tous les chemins indépendants dans un module sont exécutés au moins une fois; examiner toutes les décisions logiques des deux possibilités (vrai ou faux); exécuter toutes les structures de données internes pour assurer leur validité.

1.5 Les niveaux de test

Selon [Har92], pour aboutir à un test complet et approprié, le processus de test devrait suivre la séquence suivante à savoir le test unitaire, le test d'intégration et le test de système.

1.5.1 Le test unitaire

Le test unitaire focalise l'effort de vérification sur la plus petite unité du logiciel. L'unité du logiciel est la plus petite partie ou composante du logiciel pouvant être testée indépendamment [Lab97]. Selon [Har92], l'approche structurelle est la plus appropriée au test unitaire.

1.5.2 Le test d'intégration

Le test d'intégration a pour but la découverte d'erreurs dues à l'interaction entre les différentes unités du logiciel. Ce niveau de test permet de tester les unités lors de leur intégration au système. L'approche utilisée ici peut aussi bien être le test fonctionnel comme le test structurel ou une combinaison des deux approches [McG94].

1.5.3 Le test de système

Le test de système est l'étape finale du processus de test. Il examine tout le système informatique. Les composantes du logiciel, le support informatique ainsi que toutes les interfaces possibles. Il doit inclure les tests de recouvrement, de sécurité, de surcharge et de performance de système

1.6 Le test dans le contexte orienté objet

Le paradigme orienté objet est une nouvelle approche de développement de logiciel [Lab97]. Il réfère à une manière particulière de structurer le programme dont les buts principaux sont [Lab97]:

- haut niveau d'abstraction
- faciliter la réutilisation.

Pour atteindre ces objectifs, les programmes orientés objet utilisent la structuration suivante:

- modularisation : les données et les opérations sont cachées derrière l'interface des objets. La seule façon d'interagir avec l'objet est à travers son interface.
- classification : les objets de même classe partagent le même comportement.
- partage flexible du comportement (flexibilité) : les objets de différentes classes peuvent aussi partager les mêmes comportements à travers l'héritage et le polymorphisme.
- interprétation : le comportement réel qui se produit pour un appel particulier de fonction dépend de l'objet de réception, par un mécanisme de surcharge et de la liaison dynamique.

1.7 Test conventionnel versus test orienté objet

Le test est un passage nécessaire dans le processus de développement du logiciel. Cette étape du processus de développement permet de s'assurer de la qualité du logiciel. Il est utilisé pour prouver que l'implémentation rencontre les spécifications. Que ce soit le test conventionnel ou le test orienté objet, le but visé est le même. Cependant, la manière de l'appliquer, ainsi que l'environnement d'application et les concepts sur lesquels ils s'appliquent sont très différents. Cette différence est due à certains concepts tel que :

- ***L'héritage***

L'héritage signifie que des propriétés (c.-à-d., des attributs et des opérations) définies pour une classe d'objets sont automatiquement définies pour toutes ses sous-classes. L'avantage majeur de l'héritage réside dans la réutilisation. Certaines personnes peuvent penser qu'il est inutile de tester des méthodes héritées (car déjà testées dans la super-classe). Mais, selon [Lab97], cela est une erreur parce que les invocations d'une méthode et ses résultats dépendent de l'état de l'objet. Donc, le test de méthodes héritées devrait être une règle plutôt qu'une exception [Bin99]. Surtout parce que l'ensemble des états peut différer de la super-classe à la classe fille, car, les attributs peuvent être différents.

- ***L'encapsulation***

L'encapsulation est définie comme la modélisation et le stockage dans un objet des attributs et des opérations que l'objet est capable d'exécuter. L'encapsulation n'est pas une source de problème mais peut cependant être un obstacle au test. Le test exige qu'on se rende compte de l'état concret et abstrait de l'objet. Or, l'encapsulation pose le problème d'observabilité. La seule manière d'observer l'état d'un objet est par l'intermédiaire de ses méthodes. Le problème se pose lorsque certains attributs ne peuvent être atteints à travers des méthodes.

- ***Le polymorphisme***

Le polymorphisme est la capacité de prendre plus d'une forme: un attribut peut avoir plus d'un ensemble de valeurs, et une opération peut être mise en application par plus d'une méthode. Le polymorphisme et la ligature dynamique posent le problème de l'indécidabilité. Il est très difficile voir même impossible de déterminer de façon statique, quelle méthode sera appelée dans un cas donné de test.

Dans le test orienté objet, le sous programme ne peut être considéré comme l'élément unitaire (l'entité) du test. Cet élément étant le plus petit composant pouvant être testé de manière isolée. L'objet est donc l'entité du test orienté objet.

1.8 Test orienté objet

Nous appelons test orienté objet, le test spécifique aux applications orientées objet. En opposition avec le test conventionnel qui lui s'applique aux programmes orientés procédure. Le test orienté objet prend tout son sens avec les concepts introduits par le paradigme orienté objet. Le test orienté objet n'est pas différent du test conventionnel dans sa chronologie. Cependant, chaque étape du test doit s'ajuster aux changements apportés par les objets.

1.9 Test de classe ou test unitaire

Dans le paradigme orienté objet, il n'est pas question de tester individuellement les opérations en tant qu'entité. Selon [Bin94], il y a un consensus qui fait de la classe l'entité naturelle du cas de test. Une méthode n'existe que par rapport à la classe à laquelle elle est rattachée. On ne peut pas tester la méthode sans l'appliquer à un objet. Chaque objet possède un état; le contexte dans lequel une méthode est exécutée n'est pas seulement défini par ses paramètres mais principalement par l'objet auquel elle est appliquée. La classe étant l'élément de base des stratégies de test, son comportement doit être pris en compte [Lab97]. Pour ce faire, le comportement est décomposé en états. C'est-à-dire, les valeurs stockées dans chacun des attributs qui constituent l'ensemble des données est le test basé-état. Plusieurs auteurs traitant du sujet sont cités dans [Lab97]. Le test basé-état dérive les cas de test en modélisant les classes comme des

machines à états. Les méthodes passent par des transitions d'états. Les transitions permises sont définies par le modèle d'états [Bin94]. L'application du test basé-état n'est pas sans difficulté. Pour plus de détails voir : [Bin94, Bad95]. Donc, le test de classe en orienté-objet est l'équivalent du test unitaire dans le test conventionnel. Le test conventionnel se focalise sur les détails algorithmiques des modules et des flux de données à travers les interfaces des modules alors que le test des classes en orienté objet est guidé par l'encapsulation des méthodes dans les classes et le comportement des objets liés à leurs différents états.

1.10 Test d'intégration

Le test d'intégration consiste à tester les classes pendant qu'elles sont intégrées dans le système. Selon [Bin94], l'intégration des classes pour créer un système d'application doit être étroitement attachée à l'approche globale de développement. À cause de sa structure de contrôle non hiérarchisée, les stratégies « top-down » et « bottom-up » du test conventionnel ne peuvent pas être directement appliquées au test orienté objet [Har92]. Selon Binder [Bin94], le test d'intégration orienté objet a deux stratégies de base à savoir : le « *thread-based* » et le « *use-based* ».

- Le *Thread-based*, intègre un ensemble de classes qui répondent toutes aux mêmes stimuli. Cet ensemble de classes est utilisé pour créer un *thread.*
- Le *Uses-based*, pour un cas d'utilisation donné, les classes activées sont testées.

Une fois ce groupe testé, un autre groupe dépendant de celui-ci est testé. Il existe d'autres méthodes telles que le «*cluster testing*» qui teste les groupes de classes couplées, selon [Har92], c'est une étape du test d'intégration.

1.11 Test de système

Le test de système orienté objet est très similaire au test conventionnel. Il permet de tester le système une fois que tous les composants sont intégrés au système. Le plus souvent, l'approche fonctionnelle est la plus indiquée pour ce type de test.

Le reste du mémoire est organisé comme suit : le chapitre 2 présente les stratégies d'intégration. Le chapitre 3 aborde la nouvelle stratégie d'intégration que nous préconisons. Le chapitre 4 porte sur une évaluation des différentes stratégie abordées tout au long de ce mémoire. Nous terminerons par une conclusion au chapitre 5.

CHAPITRE 2

LES STRATÉGIES D'INTÉGRATION

2.1 Introduction

Dans les systèmes orientés objet, les objets interagissent pour implémenter le comportement. Une des principales difficultés rencontrées durant le processus d'intégration des classes de tels systèmes réside dans la détermination d'un ordre efficace de l'intégration des classes. Cette difficulté découle essentiellement des diverses dépendances qui existent entre les classes. Ces dépendances constituent une des principales caractéristiques des systèmes orientés objet [Lab00]. La présence de cycles dans ces dépendances, engendrés par les différentes relations qui peuvent exister entres les classes ainsi que les interactions qu'elles peuvent avoir entre elles, ne fait que compliquer le problème [Bad04]. Ces interactions sont inhérentes aux systèmes orientés

objet. L'objectif principal des tests d'intégration étant de révéler les fautes qui entravent le bon fonctionnement des interactions entre les différentes classes d'un système. Il est donc tout à fait pertinent d'en tenir compte [Bad04]. Le problème fondamental, en fait, consiste à définir un ordre efficace d'intégration des classes [Mil02]. Il s'agit, donc, de déterminer un ordre de séquencement de l'intégration des classes prenant en compte toutes les dépendances, en particulier les interactions, en minimisant le nombre de bouchons de test [Bad04].

Durant le processus d'intégration des classes, plusieurs critères doivent être pris en compte (orientation de l'intégration, optimisation du processus, complexité des classes, sollicitation des classes, nombre de bouchon de test, etc.). La complexité des interactions et des dépendances entre les classes peut rendre cette tâche très difficile. Plusieurs stratégies ont été proposées dans la littérature pour résoudre les divers problèmes liés aux tests d'intégration des classes dans les systèmes orientés objet [Tai99, Bri02, Tra00, etc.]. Ces stratégies ont plusieurs points en commun; elles sont guidées par deux objectifs principaux [Hah01]. Le premier porte sur la minimisation de l'effort de test (minimisation du nombre de bouchons de test, de leur complexité, de l'effort de leur conception, des étapes de l'intégration). Le second consiste à déterminer un ordre d'intégration des différentes composantes. Elles présentent, cependant, quelques différences. Elles diffèrent essentiellement dans la manière de détecter et d'éliminer les cycles de dépendances [Hah01]. Par ailleurs, certaines stratégies se préoccupent plus de l'ordonnancement de l'intégration des classes en délaissant le nombre de bouchons de

test [Tai99]. Nous présentons, dans la section suivante, quelques critères et définitions utiles à la compréhension des stratégies d'intégration orientées objet.

2.2 Dépendances

Le diagramme de la figure 1 nous permet d'illustrer le concept de dépendance entre les classes.

Figure 1: Diagramme de dépendance

La classe « H » dépend de la classe « I » et la classe « I » dépend de la classe «J» . La relation de dépendance que nous avons notée « *dep* » dans la figure 1, peut être aussi bien interprétée comme une relation d'utilisation ou comme une relation d'héritage. Autrement dit, la classe « H » utilise ou hérite de un ou plusieurs services de la classe « I » et la classe « I » utilise ou hérite de un ou plusieurs services de la classe «J».

2.3 L'ordonnancement

L'ordonnancement concerne l'ordre dans lequel vont être intégrées les différentes classes du système. Il est fortement lié aux relations de dépendances entre les classes. Si nous nous référons une fois de plus au diagramme de la figure 1, dans lequel

la classe « H » dépend de la classe « I » et la classe « I » dépend de la classe « J », un ordonnancement possible est : l'intégration de la classe J, ensuite l'intégration de la classe I et, pour finir l'intégration, de la classe H.

2.4 Les cycles de dépendance

Il n'est pas toujours facile de déterminer un ordonnancement comme se fut le cas à la section précédente. En présence de cycles de dépendances, il est impossible d'intégrer adéquatement les classes impliquées dans le cycle. La dépendance cyclique comme l'illustre la figure 2, est la cause de cette difficulté. Cette difficulté peut être surmontée par l'utilisation de bouchons de test. Nous aborderons les bouchons de test à la section suivante.

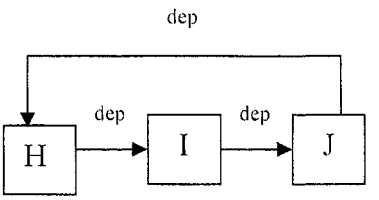


Figure 2: Diagramme: cycle de dépendance 1

2.5 Création de bouchons de test

Lors de la création de bouchons de test, les liens de dépendances vis-à-vis de la classe que l'on choisit comme bouchon de test sont cassés afin de rendre le modèle acyclique. Ce qui permettra un ordonnancement sans difficulté. Selon Tai et al. [Tai 99],

si une classe est intégrée avant une autre dont elle dépend, il y a création de bouchon de test. Cependant, la création de bouchons de test, parfois nécessaire, ne devrait se faire qu'en présence de cycle de dépendances [Hah01]. En effet, la création de bouchons de test en l'absence de cycles de dépendances est inutile [Bad04]. Les bouchons de test doivent donc, dans le cadre de toute stratégie d'intégration, faire l'objet d'une attention particulière. Un des objectifs importants de telles stratégies doit, à notre avis, être la réduction du nombre de bouchons de test, d'une part, et la minimisation de leur effort de conception, d'autre part. La figure 3 illustre la création du bouchon de test de la classe «J». Le lien [I, J] est cassé et le modèle devient donc acyclique. Il sera alors simple de faire un ordonnancement.

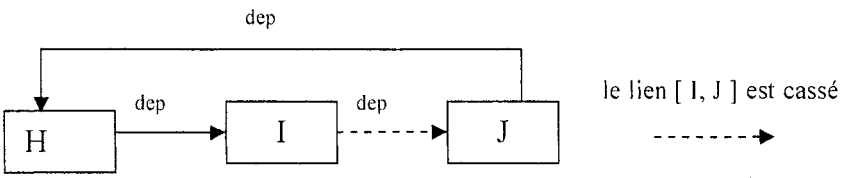


Figure 3: Illustration de la création de bouchon de test

2.6 Les bouchons de test

Un bouchon de test pourrait par exemple une classe non encore intégrée dont on simule le comportement afin d'intégrer une autre classe qui l'utilise. L'ordonnancement du modèle de la figure 2 pourrait se faire en utilisant un bouchon de test. Nous pouvons choisir de façon arbitraire de créer un bouchon de la classe «J» . L'ordonnance dans ce cas serait, l'intégration de la classe «I» (en utilisant une simulation de la classe «J»),

ensuite l'intégration de la classe «H» et pour finir, l'intégration de la classe «J». Il existe deux types de bouchons de test à savoir, les bouchons spécifiques et les bouchons réels. L'ordonnancement du modèle de la figure 3 requiert lui aussi la création de bouchons de test. Cependant, contrairement au modèle de la figure 2, qui ne contient qu'un seul cycle de dépendance qui est le cycle { H, I, J, H }, la figure 4 comprend deux cycles de dépendance à savoir , le cycle { H, I, J, H } et le cycle { I, J, I }. Si nous choisissons de façon arbitraire de créer un bouchon de la classe «I», nous serions amenés à faire un choix entre l'utilisation d'un ***bouchon de test spécifique*** et l'utilisation d'un ***bouchon test réel*** de la classe «I» .

La figure 5 illustre la création du bouchon de test de la classe «I». Les liens de dépendance [J, I] et [H, I] sont cassés afin de rendre le modèle acyclique.

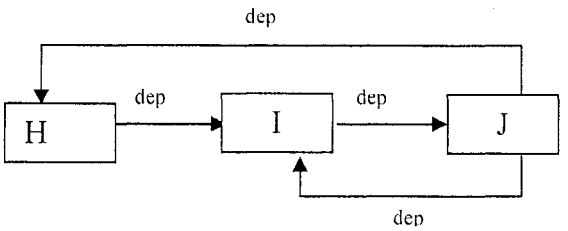


Figure 4: Diagramme: cycle de dépendance 2

2.6.1 Bouchons réels

Le bouchon réel consiste en la simulation du comportement de la classe entière. Donc, si nous choisissons d'utiliser le bouchon réel de la classe «I», lors de

l'ordonnancement du modèle de la figure 4, nous devrions concevoir un composant (bouchon) qui simule le comportement de la classe «I». Lors de l'intégration des classes du modèle, le bouchon réel de la classe «I» pourrait être utilisé pour intégrer aussi bien la classe «J» que la classe «H». L'ordonnancement serait : l'intégration de la classe «H» avec le bouchon réel de la classe «I», ensuite l'intégration de la classe «J» avec le bouchon réel de la classe «I» et pour terminer l'intégration de la classe «I». Ceci pour dire qu'un bouchon réel peut être utilisé pour intégrer toutes les classes qui dépendent de la classe dont il simule le comportement.

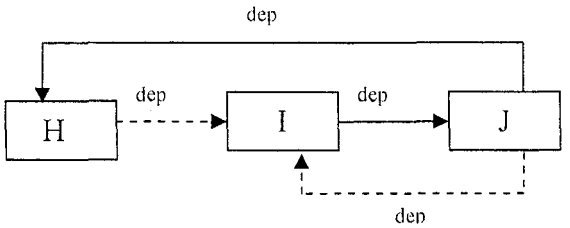


Figure 5: La création du bouchon de test de la classe «I»

2.6.2 Bouchons spécifiques

Le bouchon spécifique permet de simuler un seul service de la classe utilisée comme bouchon. La figure 5 présente deux bouchons spécifiques pour la classe «I». Ce sont les bouchons spécifiques [J, I] et [H, I]. Contrairement au bouchon réel, le bouchon spécifique ne peut être utilisé qu'avec une seule classe. Le ***bouchon spécifique [J, I]*** sera utilisé uniquement lors de l'intégration de la classe «J», alors que le ***bouchon spécifique [H, I]*** sera utilisé uniquement lors de l'intégration de la classe «H».

Dans cette configuration, le testeur a le choix entre la gestion du nombre de bouchons de test et la complexité des bouchons de test, si on part du principe qu'il est plus simple de concevoir un bouchon spécifique qu'un bouchon réel. La minimisation des bouchons de test et la complexité des bouchons de test sont des critères très importants dans le test d'intégration. Nous aborderons ces deux points dans les sections suivantes.

2.7 Minimisation des bouchons de test

La minimisation des bouchons de test dans le cadre d'une stratégie donnée est étroitement liée à la manière de détecter et de briser les dépendances, c'est-à-dire rendre le modèle acyclique. Ceci en considérant que la stratégie englobe la démarche et le modèle utilisé pour le test d'intégration. Dans cette section, nous essayerons de montrer qu'il peut y avoir des nombres différents de bouchons de test en fonction des choix que nous ferons dans la démarche de l'intégration. Le plus souvent, ces choix sont éclairés par des stratégies d'intégration. Nous n'aborderons pas ici les stratégies d'intégration. Nous mettrons tout simplement en relief le nombre de bouchons selon le choix des liens à casser pour rendre le modèle acyclique. Nous utiliserons l'expression « sortir du cycle de dépendance» pour exprimer l'annulation de la dépendance cyclique entre les classes du modèle ou un groupe de classes du modèle au cas où il existerait plusieurs cycles de dépendance dans le modèle. Nous baserons notre analyse sur le modèle de la figure 3. Elle se fera en trois démarches.

2.7.1 Démarche 1

Dans la démarche 1, nous choisissons d'utiliser la classe «J» comme bouchon de test. La figure 6 illustre le lien cassé (le lien [I, J]) pour sortir des cycles de dépendances. Ce choix donnera un bouchon réel ou un bouchon spécifique.

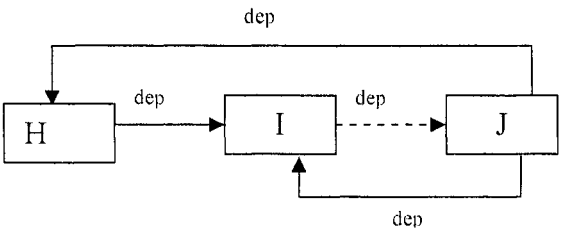


Figure 6: Diagramme: démarche 1

2.7.2 Démarche 2

Dans la démarche 2, nous choisissons d'utiliser la classe «I» comme bouchon de test. La figure 7 illustre les liens cassés (le lien [J, I] et le lien [H, I]) pour sortir des cycles de dépendances. Ce choix donnera un bouchon réel ou deux bouchons spécifiques.

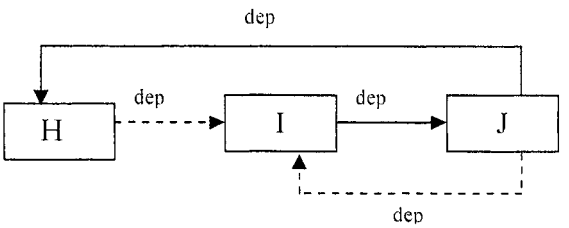


Figure 7: Diagramme: démarche 2

2.7.3 Démarche 3

Dans la démarche 3, nous choisissons dans un premier temps d'utiliser la classe «H» comme bouchon de test. La figure 8-a illustre le lien cassé à cette étape (le lien [J, H]). Cette étape nous permet de sortir d'un seul cycle de dépendance, soit le cycle { H, I, J, H }. Le modèle n'étant pas complètement acyclique après cette étape, il nous faut encore une autre étape. À l'étape deux, nous avons le choix entre casser le lien [I, J] ou le lien [J, I]. Après cette deuxième étape, le modèle est complètement acyclique. Cette démarche donnera deux bouchons réels ou deux bouchons spécifiques. La minimisation des bouchons de test est importante dans la mesure ou elle est étroitement liée au temps et à l'effort alloués au test.

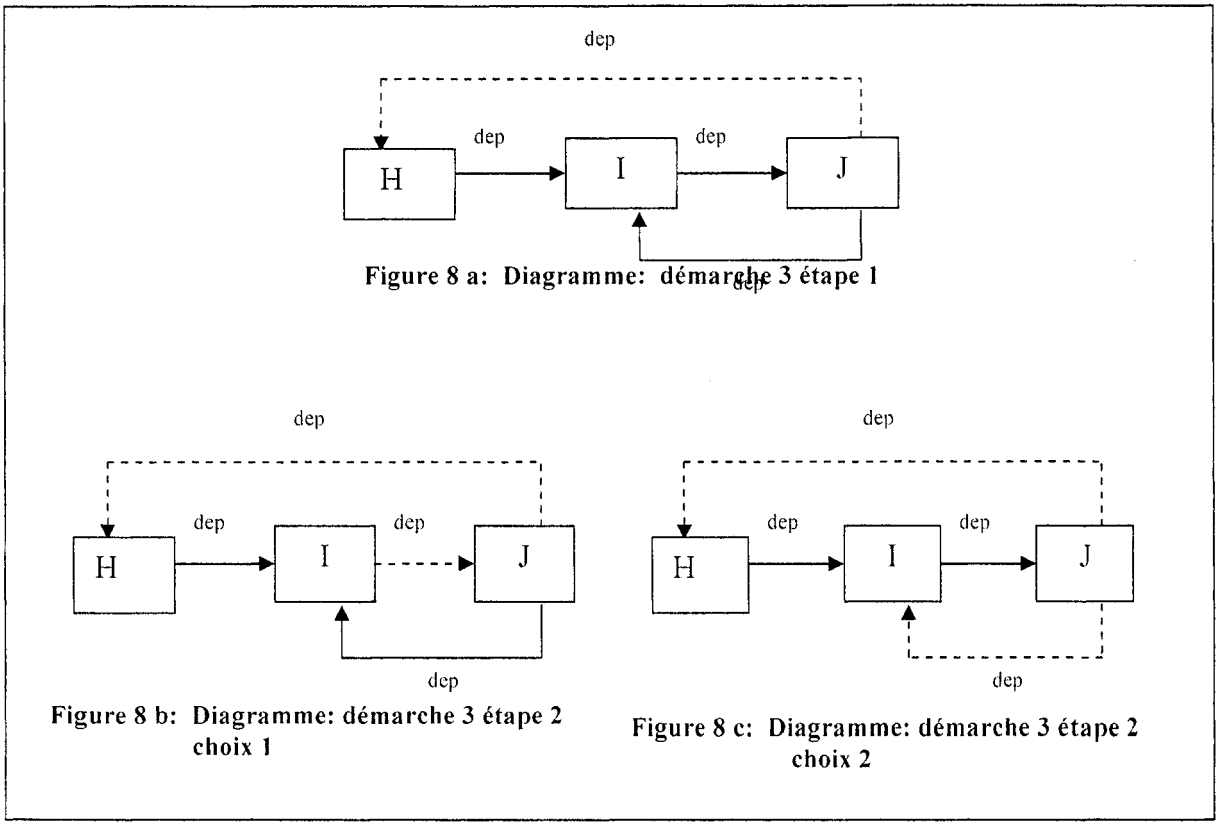


Figure 8 a: Diagramme: démarche 3 étape 1

Figure 8 b: Diagramme: démarche 3 étape 2 choix 1

Figure 8 c: Diagramme: démarche 3 étape 2 choix 2

Figure 8 : Diagramme démarche 3

2.8 Complexité des bouchons de test

La complexité des bouchons de test est liée selon certains auteurs à la nature du lien de dépendance qui existe entre les classes [Tai99, Bri02, Le Tra00, Kun95, etc.]. La complexité des bouchons de test, selon [Hah01], dépend de la densité des interactions entre les classes et de leur nature.

2.9 Le diagramme de relations entre les objets (ORD)

La modélisation des multiples dépendances et interactions entre les classes d'un système est à la base du processus d'intégration. Nous pensons que la prise en compte de ces différents aspects importants est essentielle. Ils peuvent, en effet, avoir un impact important sur le processus d'intégration à travers, en particulier, le nombre de bouchons de test, leur complexité ainsi que l'effort et le coût liés à leur création [Bad05]. Le diagramme de relations entre les objets (ORD) qui est à la base de plusieurs stratégies d'intégration [Tai99, Bri02], est plutôt basé sur les relations d'héritage, d'association et d'agrégation entre les classes comme illustré à la figure 9.

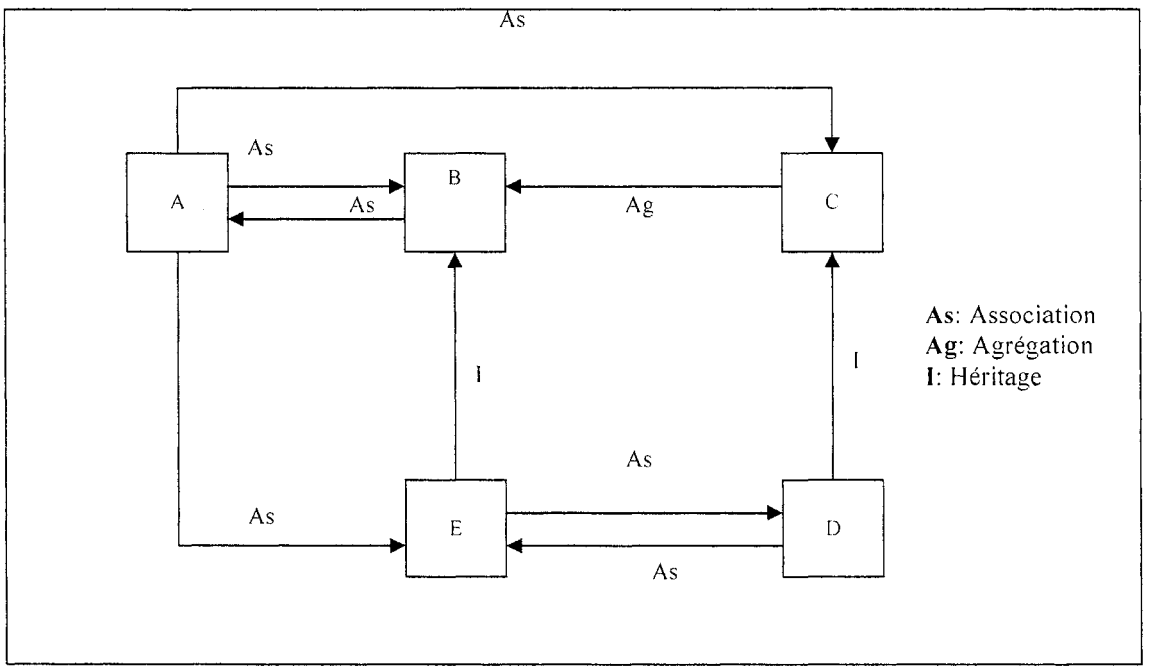


Figure 9: Diagramme des Relations entre Objets (ORD).

Le problème avec ce type de modèle est qu'il ne contient pas suffisamment de détails sur la nature et la complexité des interactions entre les classes (échanges de messages entre classes, méthodes appelées, méthodes appelantes). En effet, les modèles utilisés par ces stratégies ne donnent aucune information précise quant aux méthodes de classes effectivement impliquées dans les interactions. Aucune information n'est disponible sur les interactions qui peuvent exister entre les méthodes des classes que ça soit au niveau intra classe ou inter classes [Bad05]. Cette faiblesse pourrait, comme mentionné dans [Mil02], conduire à un grand nombre de cycles de dépendances qui ne sont pas effectifs. En effet, un cycle de dépendances qui existe entre un groupe de classes du modèle conceptuel (modèle de haut niveau d'abstraction) peut ne pas être un cycle effectif [Bad 05]. La non prise en compte des interactions entre les classes pourrait conduire à la conception de bouchons de test qui peuvent être évités dans plusieurs cas. Ceci peut, dans certains cas, engendrer des efforts et coûts importants relatifs à la conception des bouchons de test.

2.10 Les principales stratégies d'intégration

L'importance des problèmes liés au test d'intégration a toujours suscité l'intérêt des chercheurs et des industriels. Les travaux publiés sur le sujet sont nombreux. Nous nous limiterons, dans cette section, à une brève présentation des principales stratégies d'intégration des classes proposées dans la littérature [Bad04, Bad05, Tra00, Tai99, Bri02 et Hah01]. Nous nous concentrerons sur les approches basées sur les graphes. La

figure 9 illustre un exemple de diagramme ORD, qui est le modèle sur lequel se base ces stratégies d'intégration. Les différents nœuds de la figure 9 représentent les classes. Les relations entre ces classes, représentées par des arcs, sont de trois types: l'association, l'agrégation et l'héritage [Taï99].

2.10.1 La stratégie de Taï et Daniels

La stratégie de Taï et Daniels est organisée principalement en deux étapes [Tai99]. Dans un premier temps, la stratégie consiste à assigner les nombres *niveau majeur* en se basant sur les relations d'agrégation et d'héritage (tout en ignorant les relations d'association). Ensuite, au sein de chaque *niveau majeur* déterminé, la stratégie consiste à assigner aux classes les nombres *niveau mineur* en se basant sur les relations d'association. Les nombres *niveau majeur* et *niveau mineur* permettent de déterminer l'ordre d'intégration des classes du modèle. Le nombre *niveau majeur* permet la hiérarchisation des classes en fonction des relations d'héritage et d'agrégation. Le nombre *niveau mineur* permet de déterminer un ordre à l'intérieur de chaque *niveau majeur*.

Les classes auront, donc, un couple de nombres *niveau majeur* (I) et *niveau mineur* (J). C'est à partir de ces couples (I, J) que l'ordre d'intégration de chaque classe du diagramme est déterminé. Les algorithmes utilisés pour affecter les nombres *majeur* et *mineur* sont décrits dans [Tai99]. Les cycles de dépendances, dans le cadre de cette stratégie, sont éliminés au sein de chaque niveau majeur. Ce processus est basé sur une

fonction de pondération. La pondération des nœuds est déterminée selon le modèle suivant: Poids (d_i) = Nombre de Liens Entrants (nœuds de départ du lien d_i) + Nombre de Liens Sortants (nœuds d'arrivée du lien d_i). Pour éliminer un cycle de dépendances, la relation ayant le poids le plus élevé est retirée. Dans cette stratégie, les associations qui traversent des niveaux majeurs dans le sens descendant sont systématiquement retirées [Tai99]. Cela permet d'éviter qu'une classe soit intégrée avant une autre dont elle dépend. Cette stratégie ne prend pas vraiment en compte le nombre de bouchons de test. Il arrive même que, dans certains cas, elle aboutisse à la création de bouchons de test en l'absence de cycles de dépendances [Bri02].

2.10.2 La stratégie de Le Traon et al.

L'approche adoptée par Le Traon et al. [Tra00] utilise l'algorithme de Tarjan [Tar72] pour trouver les composants fortement couplés (SCC). Il a été adapté de telle sorte que chaque cycle de dépendances présent dans le modèle considéré soit déterminé et numéroté selon l'ordre dans lequel il est traversé. Cela donne comme résultat un schéma de classification à partir duquel un ordre partiel d'intégration peut être généré. Pour éliminer les cycles de dépendances, Le Traon et al. définissent un nouveau type de dépendance dénotée « *frond* » [Tra00]. Il est défini comme un lien allant d'une classe B à une classe A. la classe A étant l'origine du cycle.

Les auteurs de cette stratégie, dans leur démarche, éliminent les cycles de dépendances au sein de chaque SCC identifié. Ils se basent sur une fonction de pondération qui

permet d'affecter un poids à chacune des classes du SSC considéré. Le poids est défini comme la somme des « *fronds* » entrants plus la somme des « *fronds* » sortants d'une classe. Le cycle est éliminé en retirant les liens entrants de la classe au poids le plus élevé dans le SCC considéré. Leur démarche, pour aboutir à un ordre d'intégration des classes, consiste à appliquer cet algorithme de façon récursive à chaque SCC en choisissant un point de départ (une classe). Le point de départ est choisi de façon arbitraire, ce qui dans un premier temps ne pose pas de problème. Cependant, la notion de *frond* pourrait en souffrir car dépendamment de l'ordre dans lequel une classe est traversée, elle peut être ou non l'origine d'un cycle et posséder donc un *frond*. Cela peut influencer la pondération des classes.

2.10.3 La stratégie de Briand et al.

La stratégie proposée par Briand et al. [Bri02] utilise la même approche que Le Traon et al [Tra00]. Elle est basée sur un appel récursif de l'algorithme de *Tarjan*. Cependant, elle diffère de celle adoptée par Le Traon et al. dans la façon de casser les cycles de dépendances [Bri02]. A l'intérieur de chaque SCC identifié, la démarche adoptée consiste à calculer le poids de chaque relation d'association. La fonction de pondération utilisée permet d'affecter un poids à chaque association du SCC. Le poids de la relation entre deux classes A et B est défini par la somme des liens entrants de la classe A multipliée par la somme des liens sortants de la classe B : poids(A, B) = A_{in} * B_{out}. La relation d'association dont le poids est le plus élevé sera retirée afin d'éliminer

les cycles de dépendances au sein d'un SCC. La stratégie adoptée ne propose aucune solution en cas de parité entre les classes.

2.10.4 La stratégie Triskell

La stratégie Triskell [Hah01] a pour priorité la minimisation des bouchons de test. Elle commence, d'abord, par identifier les bouchons de test et ensuite, elle se penche sur l'ordonnancement de l'intégration [Hah01]. La stratégie Triskell utilise elle aussi l'algorithme de *Tarjan*. À la différence des deux stratégies précédentes, elle n'identifie pas les composants fortement couplés mais plutôt le composant fortement couplé. Elle identifie, en fait, la classe qui appartient simultanément au plus grand nombre de cycles de dépendances. Cette classe sera utilisée pour créer le bouchon de test. Lorsque deux classes participent au même nombre de cycles, la classe dont un lien d'association participe à un plus grand nombre de cycles est utilisée comme bouchon. Pour éliminer les cycles de dépendances, tous les liens entrants (qui participent à un cycle) de la classe retenue sont retirés, peu importe leur nature (héritage, agrégation ou association). La nature du lien est prise en considération seulement quand deux classes participent au même nombre de cycles. Dans ce cas, la stratégie se base sur les liens d'association pour déterminer la classe à utiliser comme bouchon de test. En effet, la classe, dont le lien d'association entrant participe au plus grand nombre de cycles de dépendances, sera celle utilisée comme bouchon de test. Le choix du lien à briser pour

éliminer un cycle de dépendances se fait en fonction du nombre de bouchons de test et non en fonction de la complexité du lien. Une relation d'héritage par exemple pourrait, dans le cadre de cette stratégie, être éliminée pour sortir d'un cycle de dépendances. Le processus d'ordonnancement des étapes de l'intégration est fondé sur l'identification de la profondeur de chaque classe dans le diagramme de dépendances. La profondeur est déterminée par le nombre de classes qu'il y a sur le chemin (le plus long) partant de la classe considérée à la classe source (racine). La classe source est de profondeur égale à 1.

2.11 Conclusion

La stratégie de Le Traon et al. [Tra00] n'est pas complètement déterministe dans la mesure où elle est basée sur certaines décisions arbitraires comme le soulignent Briand et al. dans [Bri02]. Cela est du à la notion de frond introduite par la stratégie. La stratégie de Taï et Daniels ne prend pas vraiment en compte la minimisation de bouchons de test. Elle peut dans certains cas conduire à la création de bouchons de test en l'absence de cycles de dépendances [Bri02]. Par ailleurs, le modèle ORD sur lequel se basent les stratégies ne prend pas vraiment en compte les interactions entre les différents objets, ce qui peut conduire à un manque d'information lors de l'analyse de celles-ci.
C'est dans ce contexte et dans un objectif d'assurance qualité que s'insère ce travail de recherche. Nous présentons, à la section suivante, une nouvelle approche pour les tests

d'intégration des classes prenant en compte les interactions qui existent entre elles. Elle est basée sur le modèle ***MDC*** (***M***odèle de ***D***épendances entre les ***C***lasses) qui dérive du diagramme de classes de conception et des diagrammes de collaboration UML. Il intègre, donc, à la fois des considérations de haut niveau (classes, relations d'héritage entre classes, etc.) et des considérations de conception représentant les multiples interactions qui existent entre les objets (décrites essentiellement dans les diagrammes de collaboration UML). Les relations d'association et d'agrégation entre classes sont, en fait, remplacées par les interactions qui les supportent.

CHAPITRE 3

LA NOUVELLE STRATÉGIE D'INTÉGRATION

3.1 Modèle de Dépendances entre Classes

Le modèle de dépendances entre classes (MDC) sur lequel s'appuie la stratégie que nous proposons tient compte des interactions qui existent entre les différentes classes d'un système. Il dérive du diagramme de classes de conception UML auquel nous intégrons les différentes interactions entre les classes décrites dans les diagrammes de collaboration UML. Les différentes relations d'association et d'agrégation présentes dans le modèle sont remplacées par les interactions entre les classes qui les supportent. Le modèle de dépendances de notre stratégie contient donc deux types de relations de base: la relation d'héritage et la relation d'utilisation. Une classe A utilise une classe B (interagit avec elle) si au moins une des méthodes de la classe A appelle au moins une des méthodes de la classe B. La figure 10 présente un exemple de modèle MDC utilisé

par la nouvelle stratégie. Ce modèle met en évidence les différentes relations que nous considérons dans la nouvelle stratégie, à savoir la relation d'utilisation et la relation d'héritage.

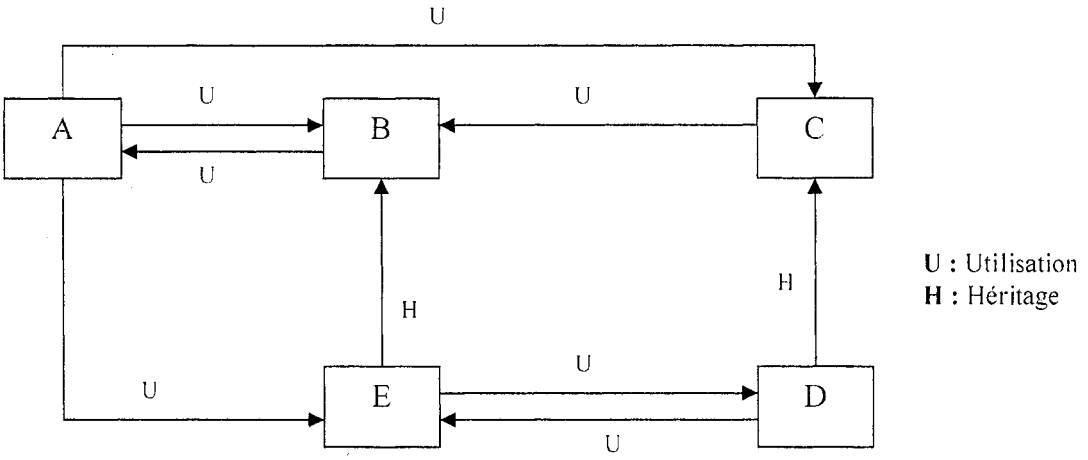


Figure 10: Modèle MDC de la nouvelle Stratégie

L'originalité de l'approche que nous préconisons pour le test d'intégration des classes se situe, d'une part, au niveau de la précision du diagramme de base et, d'autre part, dans la démarche même de la stratégie [Badri 04]. L'apport se situe surtout au niveau de la minimisation du nombre de bouchons de test et de l'effort de leur conception [Badri 05]. La minimisation des bouchons de test est due aussi bien à la précision du diagramme de base utilisé par la stratégie, qu'à la démarche même de la stratégie. La précision du diagramme nous permet, à travers la démarche d'intégration, d'identifier deux types de cycle de dépendances. Les cycles de dépendances effectifs et les cycles de dépendances non effectifs. La figure 11 illustre un exemple de chaque type de cycle. Le premier est un cycle de dépendances effectif et le second est un cycle de dépendances non effectif.

Par ailleurs, la connaissance de la nature des interactions entre les classes du modèle ainsi que de leur complexité est pertinente. Elle nous permet, en cas de parité entre classes, de faire un choix éclairé quant à la classe à utiliser comme bouchon de test. Ces différents aspects seront discutés en détail dans les prochaines sections.

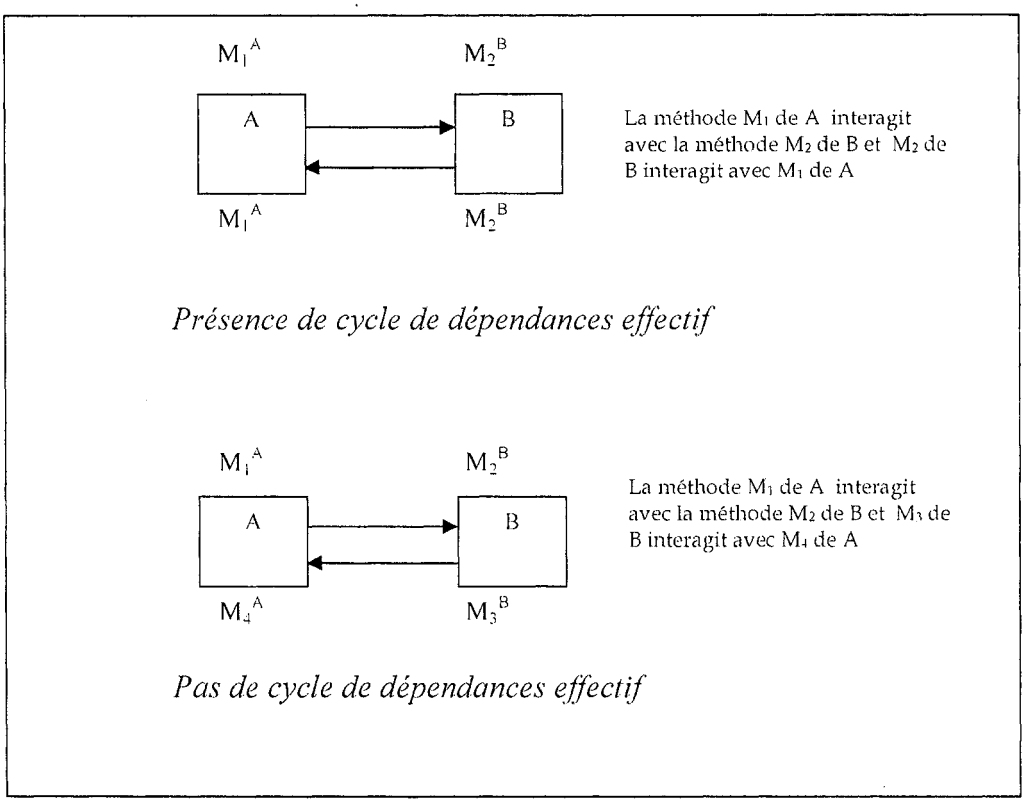


Figure 11: Cycle de dépendances.

La notion de cycle de dépendances non effectif nous a conduit à définir le principe de l'intégration partielle. La figure 12 illustre l'intégration d'un cycle de dépendances non effectif. À travers cette illustration nous présentons la notion de l'intégration partielle,

relative à l'intégration d'une classe sans une ou plusieurs de ses méthodes. Là ou les méthode non intégrées sont celles qui nous permettent de sortir du ou des cycles de dépendance non effectifs. La figure 13 présente l'intégration dans le cas d'un cycle de dépendances effectif.

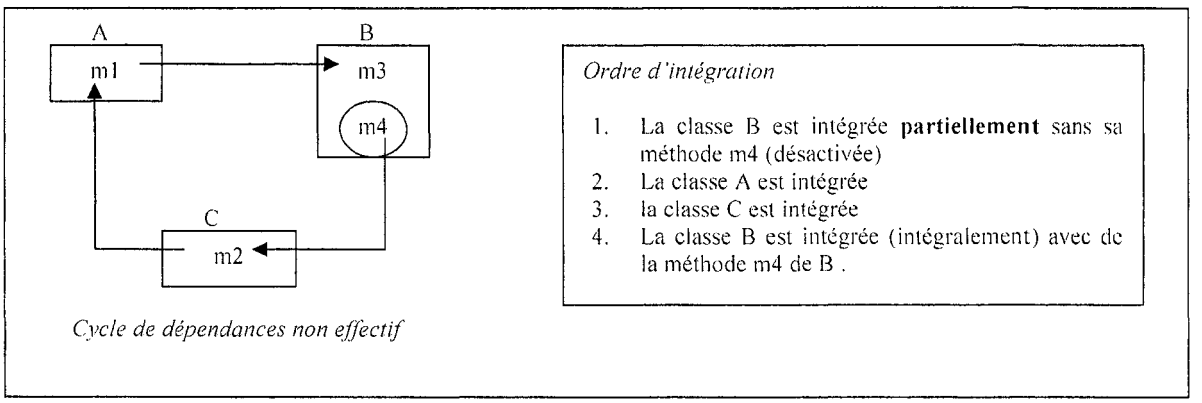


Figure 12: Intégration d'un cycle de dépendances non effectif.

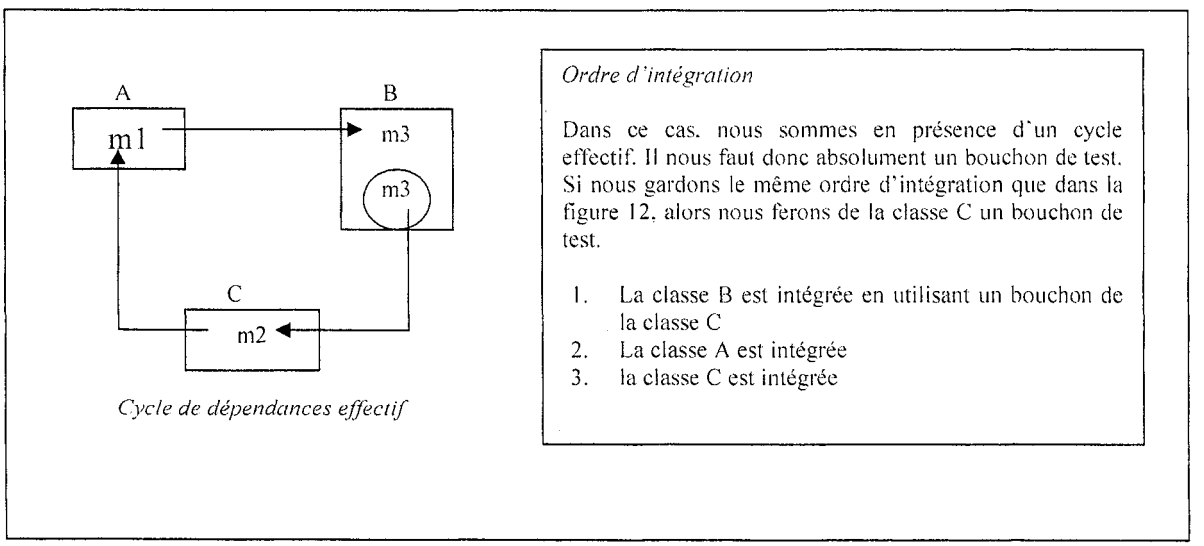


Figure 13: Intégration d'un cycle de dépendances effectif.

3.2 Méthode d'intégration : Principales étapes

La méthode d'intégration que nous proposons est organisée en quatre principales étapes [Badri 04]:

- Affectation d'un nombre niveau majeur,
- Détermination et élimination des cycles de dépendances (effectifs et non effectifs),
- Affectation d'un nombre niveau mineur,
- Ordonnancement des classes.

3.2.1 Affectation d'un nombre niveau majeur

Le principe de cette étape consiste à diviser le diagramme initial en plusieurs niveaux, en prenant la relation d'héritage comme critère de séparation. Cette démarche nous permet d'isoler les relations d'héritage lors d'une première suppression des cycles de dépendance. C'est une adaptation de la stratégie de Taï et Daniels [Tai 99]. La stratégie de Taï et Daniels se base sur les relations d'héritage et d'agrégation pour affecter le nombre *niveau majeur*. La figure 14 illustre l'affectation des nombres *niveau majeur* au modèle de la figure 10.

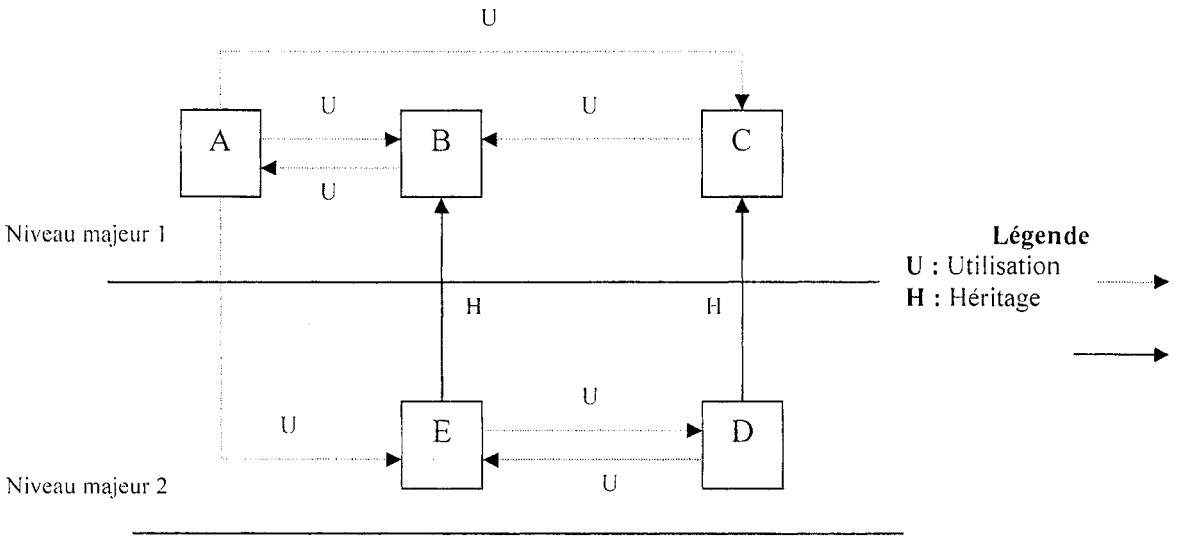


Figure 14: Affectation du niveau majeur en fonction uniquement de l'héritage.

3.2.2 Détermination et élimination des cycles de dépendances

Comme dans toutes les autres stratégies, cette étape consiste à détecter et à éliminer les cycles de dépendances présents dans le modèle. Dans la nouvelle stratégie, nous nous attaquons, dans un premier temps, aux cycles de dépendances à l'intérieur de chaque niveau majeur (comme le fait la stratégie de Taï et Daniels). Pour ce faire, nous calculons le poids de chacune des classes du niveau majeur considéré. Le poids des classes nous permet de déterminer les bouchons de test. Le poids de chaque classe est évalué en fonction de ses liens sortants impliqués dans le cycle: *poids de la classe C = somme de ses liens sortants impliqués dans un ou des cycles de dépendances*. La classe qui à le poids le plus élevé est choisie comme bouchon de test. Après le choix du

bouchon, tous ses liens entrants qui interviennent dans le SCC sont retirés du diagramme. Ensuite, nous calculons à nouveau le poids des classes et ainsi de suite jusqu'à ce qu'il n'y ait plus de cycles de dépendances au sein du *niveau majeur*. L'élimination des cycles de dépendances est basée sur un processus descendant. Il commence au niveau le plus élevé et descend jusqu'au niveau le plus bas.

En cas de parité:

Dans le cas de parité entre classes à un niveau N, l'algorithme de la démarche se reporte au niveau N+1. Nous procédons, dans ce cas, à une analyse des relations que possèdent les classes de parité égale du niveau N avec les classes du niveau N+1. Plusieurs cas sont considérés:

- Si l'une des classes du niveau N utilise une classe du niveau N+1, celle-ci est prise comme bouchon de test pour éliminer le cycle. La classe appelante est toujours intégrée après la classe appelée. Dans l'exemple illustré par la figure 15, il y a parité entre les classes B et H, toutes les deux de niveau 1. Étant donné que la classe B utilise la classe D qui est de niveau 2, la classe B est alors prise comme bouchon de test.

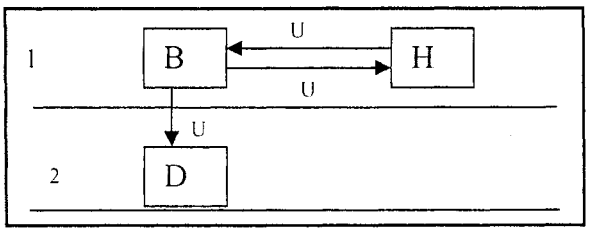


Figure 15: Parité entre deux classes, l'une utilise une classe de niveau n+1.

- Si une classe donnée est impliquée dans un cycle de dépendances qui n'implique aucune autre classe de même parité, celle-ci est alors utilisée comme bouchon de test. Ceci permettra de sortir du cycle de dépendances du niveau N, d'une part, et du cycle inter niveaux engendré par les relations de la classe avec les autres classes du modèle, d'autre part. Dans l'exemple de la figure 16, Il y a parité entre les classes B et A. Étant donné que A et B sont impliquées dans un SCC inter niveaux, alors nous appliquons la fonction de pondération pour sortir des cycles de dépendances.

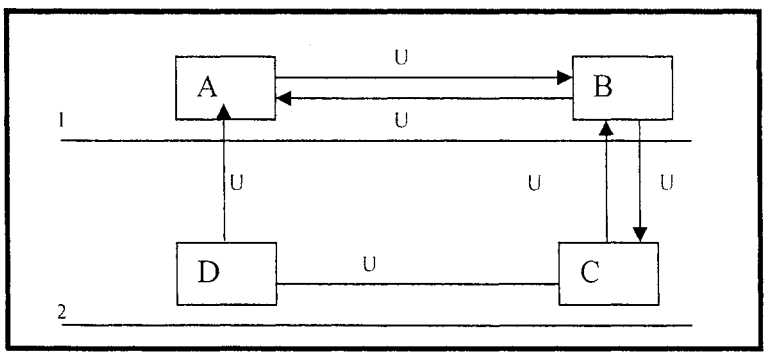


Figure 16: Parité entre deux classes, l'une est impliquée dans un cycle inter niveaux.

- Dans le cas où plusieurs classes de parité égale seraient impliquées dans un cycle inter niveaux, la démarche consiste à appliquer au SCC en question la fonction de pondération définie précédemment pour sortir du cycle. Nous partons du principe, dans ce cas, que si une classe est utilisée comme bouchon de test dans un niveau N et est impliquée dans un cycle entre deux niveaux, elle est plus susceptible d'être utilisée comme bouchon de test. L'exemple de la figure 9, illustre le cas ou il y a parité entre les classes B et A. Étant donné que A et B sont impliquées dans un SCC inter niveaux, alors nous appliquons la fonction de pondération pour sortir des cycles de dépendances.

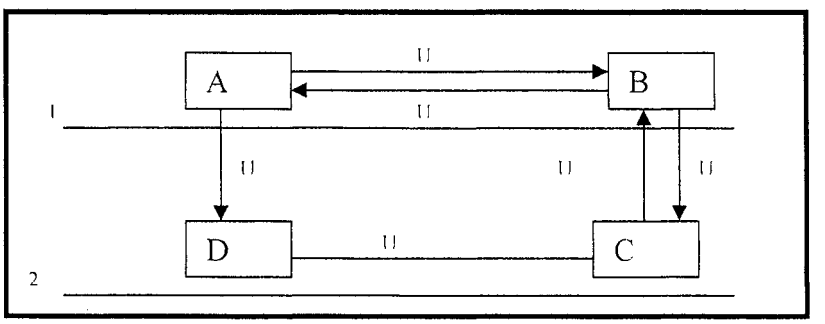


Figure 17: Parité entre deux classes impliquées dans un cycle inter niveaux.

- Si la parité persiste, la démarche consiste à utiliser les différents détails contenus dans le modèle initial quant aux interactions entre classes. Ceci permet de mettre en évidence et de prendre en compte la complexité des

interactions entre les classes impliquées dans le cycle et de faire, par conséquent, un choix judicieux du bouchon le plus complexe.

3.2.3 Affectation d'un nombre niveau mineur

Il s'agit, à cette étape, d'affecter à chaque classe du modèle une valeur déterminée en fonction des relations de dépendance qu'elle possède avec les autres classes du modèle. Si une classe B dépend (hérite ou utilise) d'une classe A, alors à la classe B sera affecté le nombre *niveau mineur* de la classe A augmenté de 1. Ce processus d'affectation se fait d'abord, comme dans le cas de la stratégie proposée par Taï et Daniels [Tai 99], au sein de chaque *niveau majeur*, sans tenir compte des dépendances inter *niveaux majeur*. À ce stade du processus d'intégration, les valeurs affectées aux classes sont provisoires. Ce n'est que par la suite, en tenant compte des relations inter *niveaux majeurs*, que nous affectons les valeurs définitives correspondant au *niveau mineur* de chaque classe du modèle. Si nous nous référons à la figure 18, nous pouvons voir que les classes A, B, D et C ont respectivement pour niveau mineur provisoire (au sein de leur niveau majeur), $P_a = 1$, $P_b = 2$, $P_d = 1$ et $P_c = 1$. En tenant compte des relations inter niveaux majeurs, les *niveaux mineurs* définitifs sont (dans le même ordre que précédemment), $D_a = 2$, $D_b = 3$, $D_d = 1$ et $D_c = 4$.

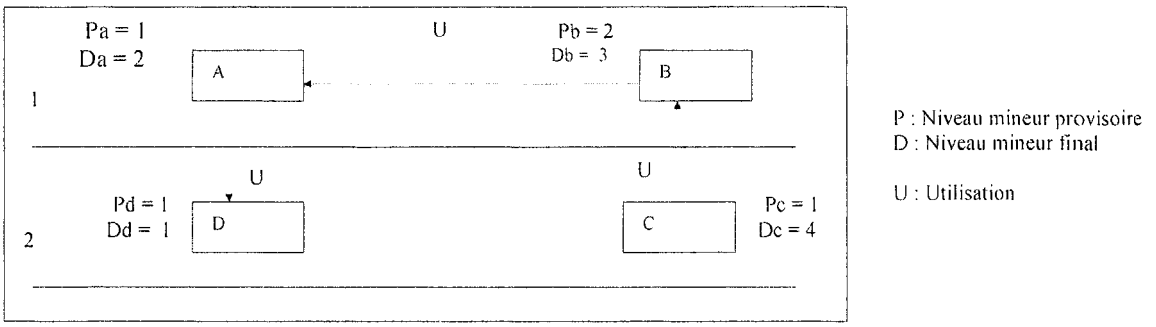


Figure 18: Processus d'affectation du nombre *niveau mineur* aux classes.

3.2.3 Processus d'ordonnancement des classes

Le processus d'ordonnancement consiste à déterminer l'ordre d'intégration des classes. Il est basé sur les couples (*niveau majeur*, *niveau mineur*) déterminés précédemment. Les classes dont le niveau mineur est plus faible sont intégrées avant celles dont le *niveau mineur* est élevé. En cas de parité du *niveau mineur*, le niveau majeur est utilisé pour permettre d'effectuer une sélection finale. Si on se rapporte à la figure 19, la classe B est intégrée avant la classe A, car le niveau mineur de B (1) est inférieur à celui de A (2)

Classe A : Niveau majeur =1
Niveau mineur =2

Classe B : Niveau majeur =2
Niveau mineur =1

Figure 19: Processus d'ordonnancement des classes.

3.2.5 Algorithme principal de la stratégie

Le processus d'intégration se déroule en trois principales étapes: l'affectation de nombres *niveau majeur*, l'affectation de nombres *niveau mineur* et l'ordonnancement des classes.

AffectationNombreNiveauMajeur (MDC)

Diviser le diagramme en plusieurs niveaux, en prenant l'héritage pour seul critère de séparation.

ÉliminationCyclesDépendances()

- *TraiterCycleDependances (niveau majeur)*
    - *DétecterCyclesDépendances()*
    - *CalculPonderation ()*
    - *ChoixBouchonTest ()*
        - *TraitementParité*
    - *RetraitLiensEntrantsBouchon (bouchon)*

Au sein de chaque *niveau majeur*, nous cassons les cycles de dépendances en calculant le poids de chacune des classes du *niveau majeur*. La classe qui a le poids le plus élevé est choisie comme bouchon de test. Après le choix du bouchon, tous les liens entrants du bouchon qui interviennent dans le SCC sont retirés du diagramme. Ensuite, nous calculons à nouveau le poids des classes tant qu'il y a des cycles de dépendances au sein du *niveau majeur*.

AffectationNombreNiveauMineur ( )

Cette affectation se fait d'abord au sein de chaque *niveau majeur* (d'un nombre niveau mineur provisoire), sans tenir compte des dépendances inter niveaux. Ensuite, en tenant compte des relations inter niveau, nous affectons le nombre niveau mineur définitif.

OrdonnancementClasses ( )

L'ordonnancement des classes se fait en tenant compte de leur couple (niveau majeur, niveau mineur). Les classes avec un *niveau mineur* plus bas sont intégrées avant celles dont le *niveau mineur* est élevé. En cas de parité du niveau mineur, le niveau majeur est utilisé.

CHAPITRE 4

ÉVALUATION COMPARATIVE DES STRATÉGIES D'INTÉGRATION

4.1 Introduction

Nous présentons, dans cette section, une évaluation comparative des différentes stratégies d'intégration présentées dans les chapitres 2 et 3. La comparaison s'effectuera principalement en termes du nombre de bouchons de test engendrés. Nous ferons trois évaluations. La première et la deuxième évaluation seront effectuées sur des modèles théoriques. Dans la troisième évaluation, nous utiliserons des cas pratiques d'applications.

L'évaluation 1 s'effectuera sur le modèle illustré par la figure 20. La stratégie de Le Traon et al. [Tra00], celle de Taï et Daniels [Taï99] ainsi que celle de Briand et al. [Bri01], ont été traitées dans [Bri01], pour ce modèle. Nous reprenons donc les mêmes

résultats que [Bri01]. Le modèle de la figure 20, sera légèrement modifié, à la Figure 19 (nous remplaçons dans le diagramme les relations d'association et d'agrégation par la relation d'utilisation) pour appliquer la stratégie que nous préconisons. L'évaluation 2 s'effectuera sur le modèle illustré par la figure 24, dans lequel nous insérons les interactions entre les méthodes d'une classe à l'autre. Nous ne touchons pas aux interactions des méthodes au sein d'une même classe.

Nous avons complété cette comparaison avec l'évaluation 3. Cette évaluation c'est faite sur des projets réels Java.

4.2 Évaluation 1

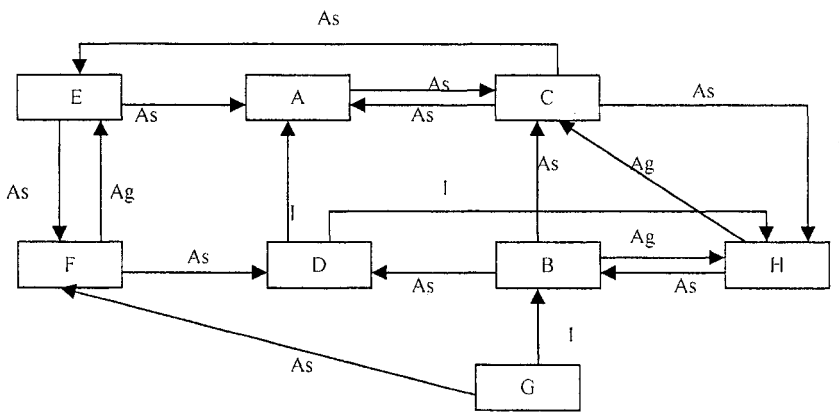


Figure 20: ORD évaluation 1.

4.2.1 Application de la stratégie de Le Traon et al.

Cette stratégie se base sur l'algorithme de Tarjan [Tar72], pour déterminer les composant fortement couplés. Le choix du point de départ dans cette stratégie est arbitraire. Le point de départ choisi dans cet exemple est la classe G. Le résultat obtenu est illustré par la figure 21. Chaque lien cassé dans le modèle engendre la création de bouchons de test.

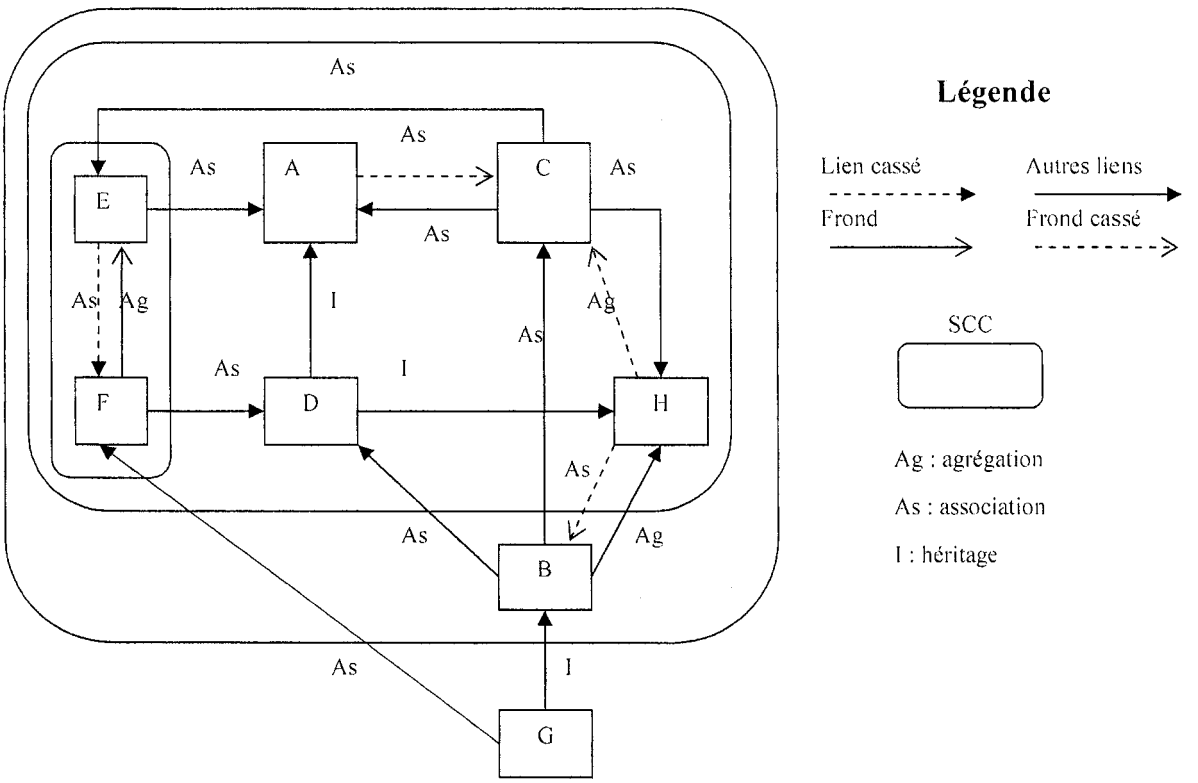
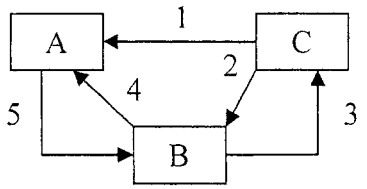


Figure 21: Résultat final de la stratégie de Le Traon et al.

L'ordre d'intégration obtenu en appliquant la stratégie de Le Traon et al. [Tra00] est le suivant:

1. A est testée en utilisant le bouchon spécifique représenté par le lien (A, C)
2. H est testée en utilisant le bouchon spécifique représenté par le lien (H, B) et le bouchon spécifique représenté par le lien (H, C)
3. D est testée en utilisant A et H
4. E est testée en utilisant A et le bouchon spécifique représenté par le lien (E, F)
5. F est testée en utilisant E et D
6. C est testée en utilisant A, H et E
7. B est testée en utilisant D, C, H
8. G est testée en dernier lieu en utilisant F et B.

L'application de la stratégie de Le Traon et al. engendre, tel qu'illustré précédemment, un nombre de bouchons de test réels égal à 3. Ce sont les classes: C, B et F. Le nombre de bouchons spécifiques est de 4. Par ailleurs, l'algorithme qui supporte la stratégie de Le Traon et al. n'est pas déterministe. En effet, son rendement dépend d'un certain nombre de décisions arbitraires comme mentionné dans [Bri01]. Il y a, en fait, deux niveaux de non-déterminisme [Bri01]. D'abord, le résultat dépend de la classe choisie initialement. Cette classe constitue le point de départ de l'algorithme de Tarjan [Tar72].



Si A est le point de départ, les liens 1, 2 et 4 sont des *fronds*. Nous avons alors deux bouchons.
Si B est le point de départ, les liens 5 et 2 sont des *fronds*. Nous avons alors un bouchon.

Figure 22: Exemple de détermination des fronds.

Les dépendances peuvent être classées comme *frond* ou non, selon la façon dont le diagramme est traversé. Si nous nous référons à la figure 22, lorsque A est choisi comme point de départ, les liens 1, 2 et 4 sont des *fronds*. Nous avons dans ce cas deux bouchons. Dans le cas où c'est B qui est choisi comme point de départ, les liens 5 et 2 sont des *fronds*. Nous avons dans ce cas un bouchon. La pondération, se basant seulement sur des dépendances de *frond*, peut nous conduire à des nombres différents de bouchons de test pour un même modèle. Par ailleurs, l'algorithme utilisé n'indique pas ce qu'il faut faire lorsque les classes ont le même poids.

4.2.2 Application de la stratégie de Briand et al.

La stratégie de Briand et al. [Br01] se base, elle aussi, sur l'algorithme de Tarjan [Tar72]. Le premier appel à cet algorithme permet d'identifier le premier niveau SCC. La figure 23 présente le résultat (partiel) de cette première étape. Le SCC déterminé est: {F, E, C, A, D, B, H}. Ceci implique l'ordre de test partiel suivant :

1. SCC {F, E, C, A, D, B, H} est testé;

2. G est testée en utilisant le SCC {F, E, C, A, D, B, H}.

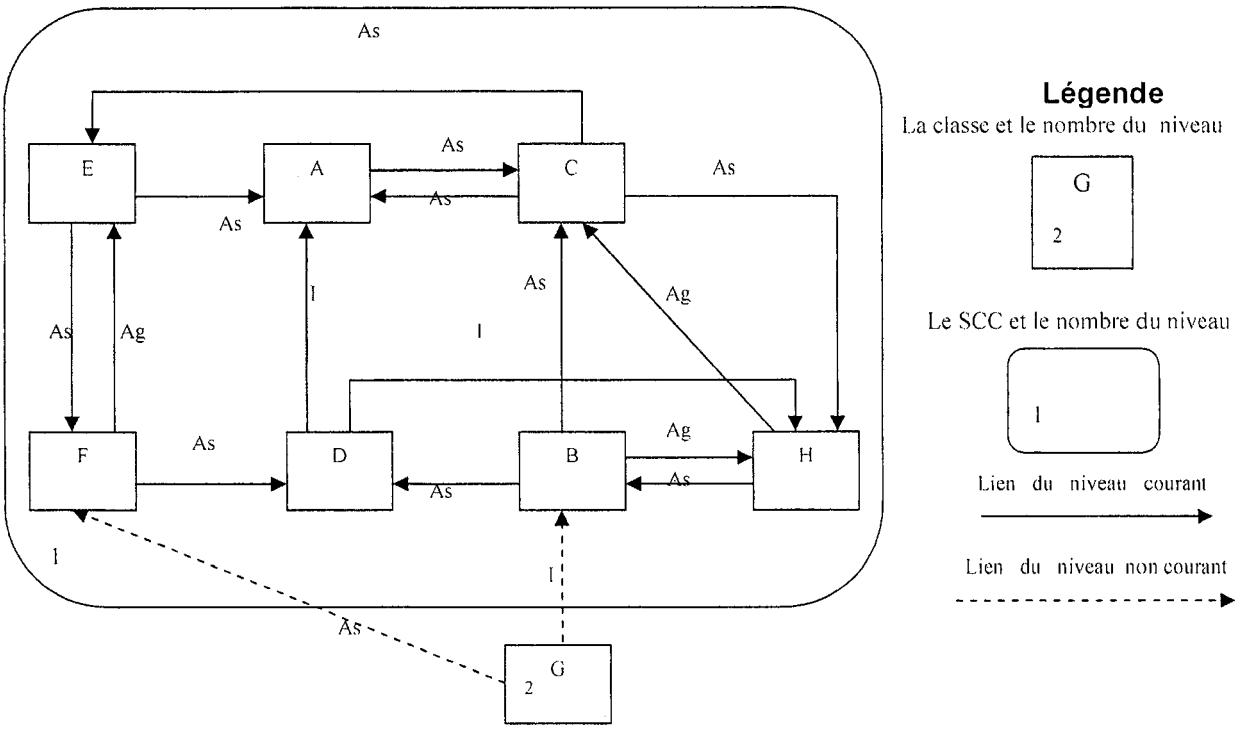


Figure 23: Premiers niveaux de SCC de la stratégie de Briand et al.

Le calcul, selon l'approche adoptée par Briand et al. [Bri01], du poids de chaque association au sein du SCC {F, E, C, A, D, B, H } montre que les associations (A, C) et (H, B) ont le même poids : $W(H,B) = H_{in} * B_{out} = 3 * 3 = 9$ et $W(A,C) = A_{in} * C_{out} = 3 * 3 = 9$. Ces associations possèdent les poids les plus élevés. L'association (A, C) est supprimée pour éliminer le cycle de dépendances. Selon la démarche de cette approche, il s'agit ensuite d'appliquer, lors de la prochaine étape, l'algorithme de *Trajan* de façon récursive au SCC {F, E, C, A, D, B, H}. Les différentes étapes de cette stratégie sont présentées dans [Bri01]. La figure 24 représente le résultat final.

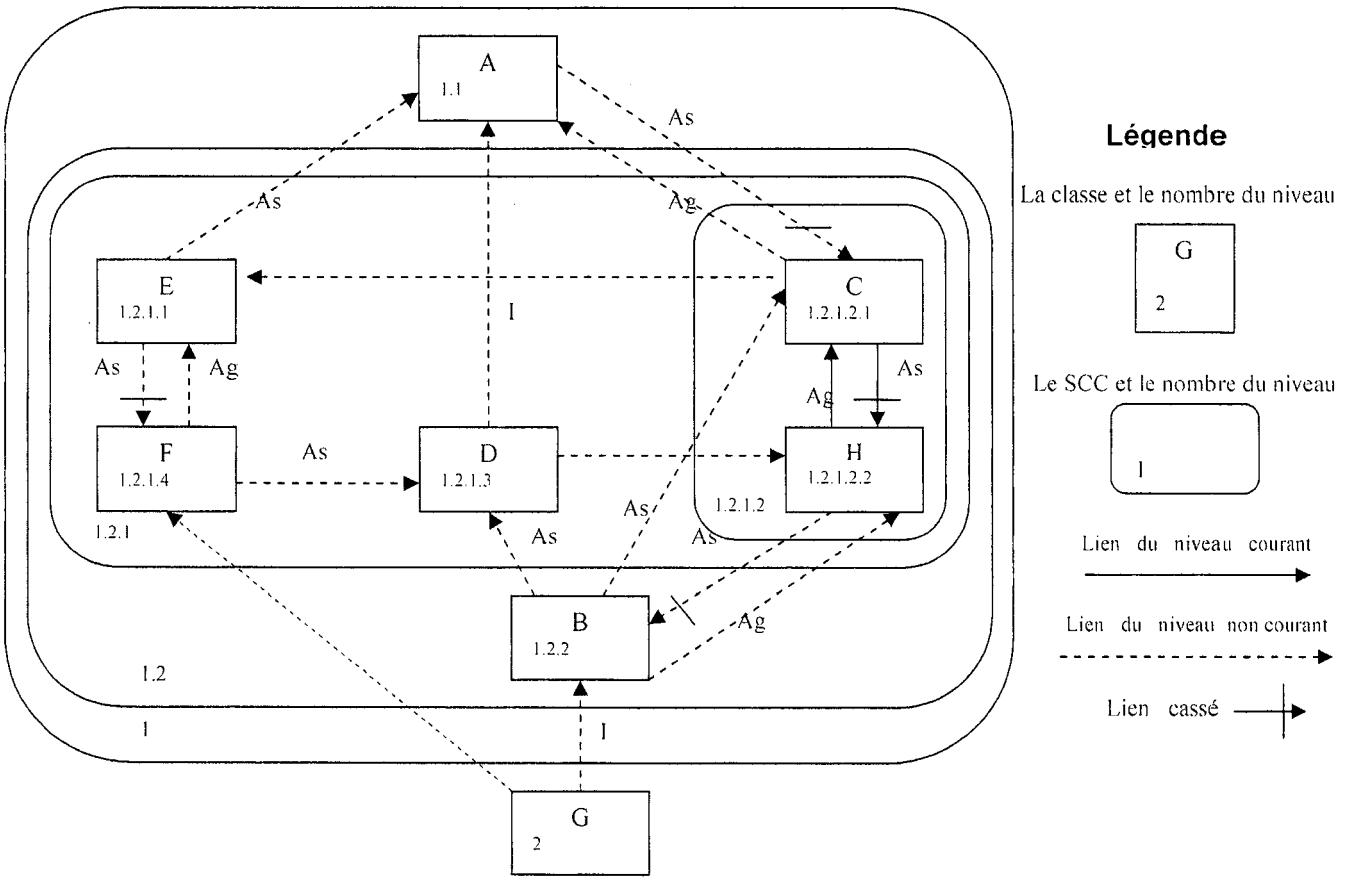


Figure 24: Résultat final de la stratégie de Briand et al.

L'ordre de test final est le suivant :

1. Pour le SCC {F, E, C, A, D, B, H},

   1.1 A est testée en utilisant le bouchon spécifique (A,C),

   1.2 Pour le SCC {F, E, C, D, B, H},

      1.2.1 Pour le SCC {D, F, E, C, H},

         1.2.1.1 E est testée en utilisant A et le bouchon (E,F),

1.2.1.2 Pour le SCC {C, H},

1.2.1.2.1 C est testée en utilisant A et le bouchon spécifique (H,C),

1.2.1.2.2 H est testée en utilisant C et le bouchon spécifique (H,B),

1.2.1.3 D est testée en utilisant A et H,

1.2.1.4 F est testée en utilisant E et D,

1.2.2 B est testée en utilisant C, D et H,

2. G est testée en utilisant B et F.

L'application de cette stratégie engendre un nombre de bouchons réels égal à 4. Ce sont les classes : B, C, F et H. Le nombre de bouchons spécifiques engendrés est aussi égal à 4.

4.2.3 Application de la stratégie Triskell

L'application de la stratégie Triskell [Hah01] au diagramme de dépendances illustré par la figure 12 donne les résultats présentés dans les différents tableaux suivants. Le tableau 1 présente les résultats de l'application de la fonction de recherche des cycles dans lesquels est impliquée chacune des classes. Le tableau 2 présente la profondeur de chaque classe ainsi que son ordre d'intégration. Nous remarquons, dans le tableau 1, la présence de poids équivalents pour les classes C et H. Nous appliquons, alors, le deuxième critère de la stratégie Triskell pour les départager. La classe H sera choisie. Ses liens entrants seront retirés pour éliminer le plus grand nombre de cycles de

dépendances. Sachant qu'il existe encore des cycles dans le graphe, nous appliquons alors de nouveau le même algorithme jusqu'à ce qu'il n'y ait plus de cycles.

Classe	Cycles impliquant la classe	Nombre de cycles impliquant la classe
A	(E,A,C), (A,C), (A,C,H,B,D)	3
B	(D,B,H), (A,C,H,B,D), (B H)	3
C	(E,A,C), (A,C), (A,C,H,B,D), (C,H), (E,F,D,H,C)	5
D	(D,B,H), (A,C,H,B,D), (E,F,D,H,C)	3
E	(E,A,C), (E,F), (E,F,D,H,C)	2
F	(E,F), (E,F,D,H,C)	1
G	-	0
H	(E,F,D,H,C), (D,B,H), (A,C,H,B,D), (C,H), (B H)	5

Tableau 1: Cycles impliquant chacune des classes.

Classe	Profondeur	Ordre de test
A	4	1
B	2	7
C	3	5
D	3	3
E	3	4
F	2	6
G	1	8
H	4	2

Tableau 2: Ordre de test et profondeur des classes.

L'application de la stratégie Triskell [Hah01] sur le modèle de la figure 20 engendre trois bouchons de test réels (C, B, F) et quatre bouchons de test spécifiques.

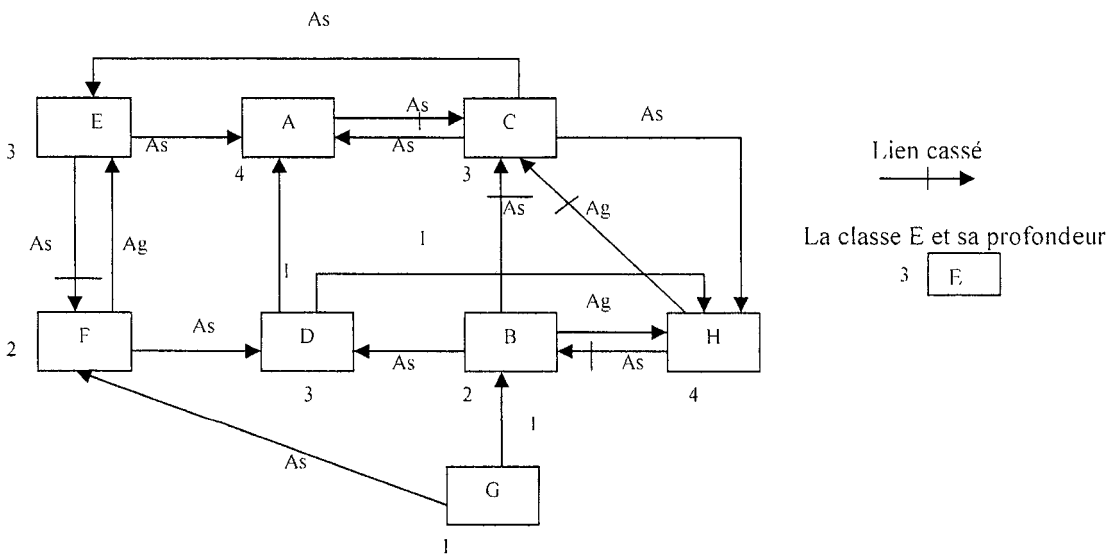


Figure 25: Résultat final de l'application de l'algorithme de la stratégie Triskell.

4.2.4 Application de la stratégie de Taï et Daniels

Le résultat de l'application de la stratégie de Taï et Daniels [Taï99] sur l'exemple considéré est donné par la figure 26.

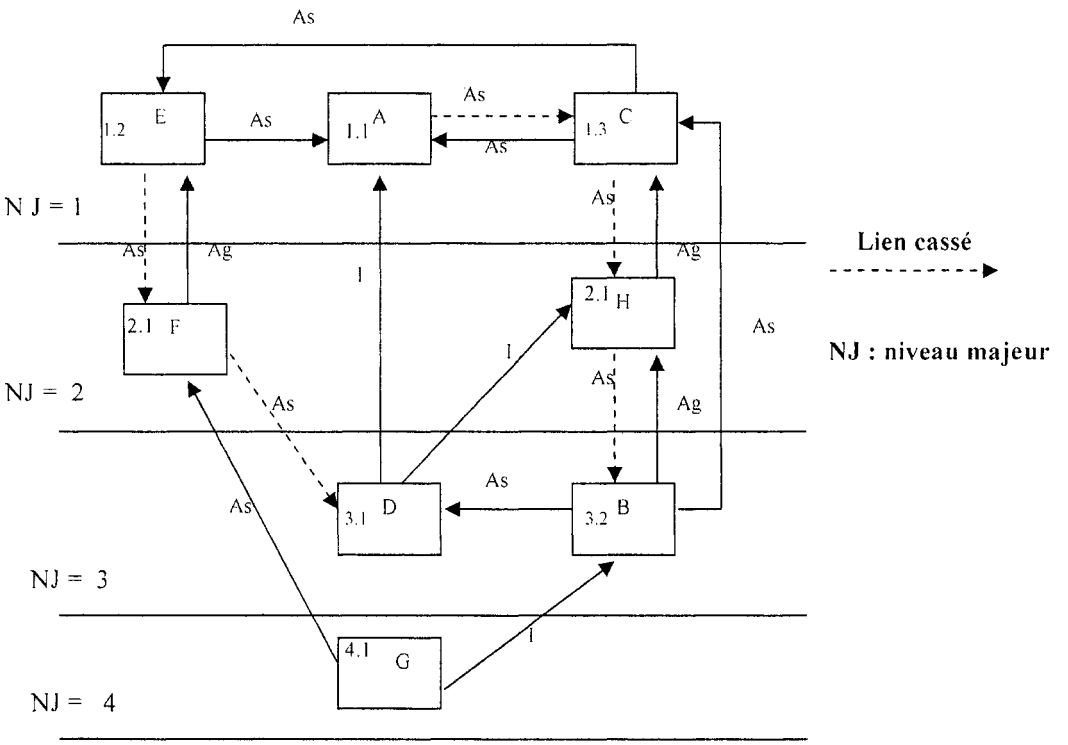


Figure 26: Résultat final de l'application de l'algorithme Taï et Daniels.

La figure 26 donne l'ordre d'intégration suivant:

1. A est intégrée en utilisant le bouchon (C, A),
2. E est intégrée en utilisant le bouchon (F, E) et la classe A,
3. C est intégrée en utilisant le bouchon (H,C) et les classes E et A,

4. F est intégrée en utilisant le bouchon (D,F) et la classe E,
5. H est intégrée en utilisant le bouchon (B,H) et la classe C,
6. D est intégrée en utilisant les classes A et H,
7. B est intégrée en utilisant les classes D, H et C,
8. G est intégrée en utilisant le bouchon (H,C) et les classes F et B.

Nous obtenons donc 5 bouchons C, F, H, D, B et 5 bouchons spécifiques (C, A), (F, E), (H, C), (D, F), (B, H).

4.2.4 Application de la Nouvelle Stratégie

La nouvelle stratégie que nous proposons [Bad04, Bad05] est organisée en plusieurs étapes. Nous illustrons, dans ce qui suit, les résultats de son application sur le modèle de la figure 20. Notons, cependant, que nous avons modifié la figure 20 (pour avoir la figure 27) afin de l'adapter au modèle de la nouvelle Stratégie.

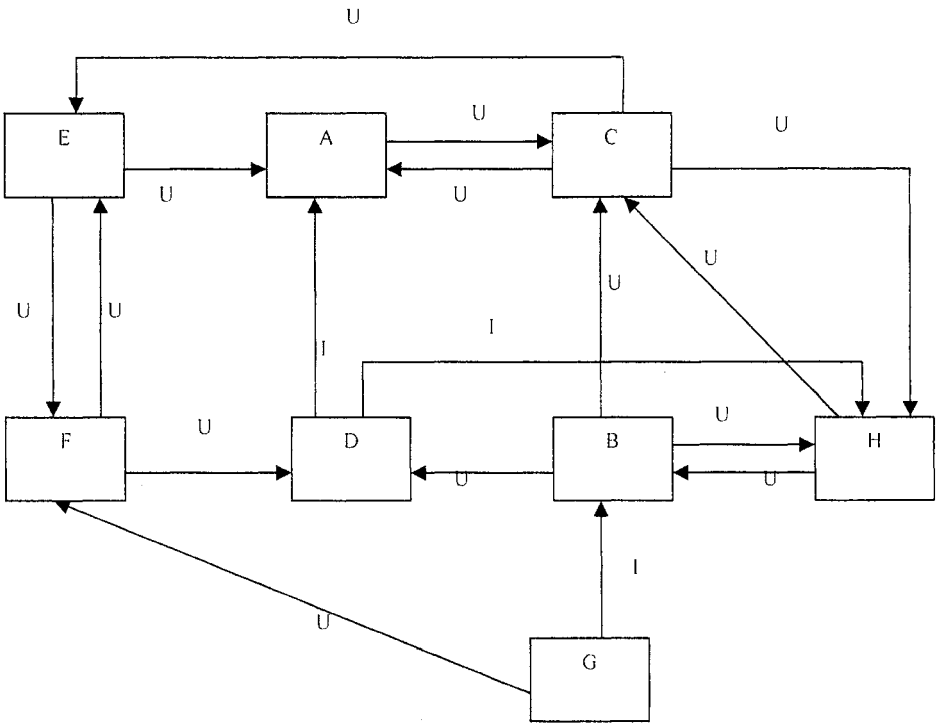


Figure 27: Diagramme modifié pour la nouvelle stratégie.

4.2.4.1 Affectation des nombres niveau majeur

La figure 28 illustre les niveaux déterminés en tenant compte de la relation d'héritage. Cette étape du processus d'intégration est une adaptation de la stratégie présentée dans [Taï99]. Selon cette approche, les classes A, B, C, E, F et H, qui n'héritent d'aucune autre classe, sont dans le niveau 1. Les classes D et G qui héritent respectivement des classes A, H et B sont au niveau 2.

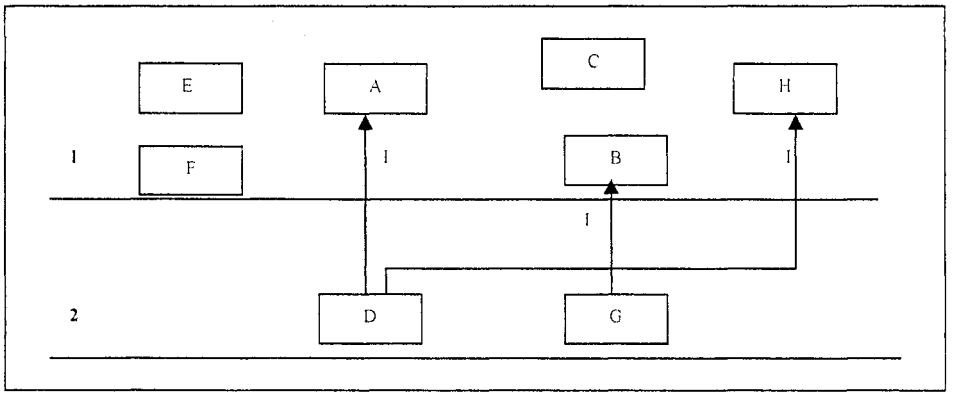


Figure 28: Détermination des niveaux sans tenir compte des relations d'utilisation.

4.2.4.2 Détermination des cycles de dépendances

Le diagramme de la figure 28, représente les différents nombres *niveau majeur* que nous avons obtenus en appliquant la nouvelle stratégie au digramme de la figure 27. Pour obtenir les deux niveaux présentés à la figure 28, nous retirons toutes les relations d'utilisation du modèle de la figure 27, car la détermination des nombres *niveau majeur* se base uniquement sur les relations d'héritage. Après la détermination des nombres *niveau majeur*, nous procédons à l'élimination des cycles de dépendances au sein de chaque *niveau majeur*. Les relations d'utilisation sont alors réintégrées dans le modèle. La figure 29 représente le *niveau majeur* 1. Pour éliminer les cycles de dépendances au sein du niveau majeur 1, il faut casser certains liens du modèle. Pour cela, il faut calculer le poids de chaque classe. La classe qui a le poids le plus élevé est utilisée comme bouchon de test pour sortir des cycles de dépendances dans lesquels elle est impliquée. Le poids est défini pour une classe comme étant le nombre total de ses liens sortants impliqués dans des cycles de dépendances.

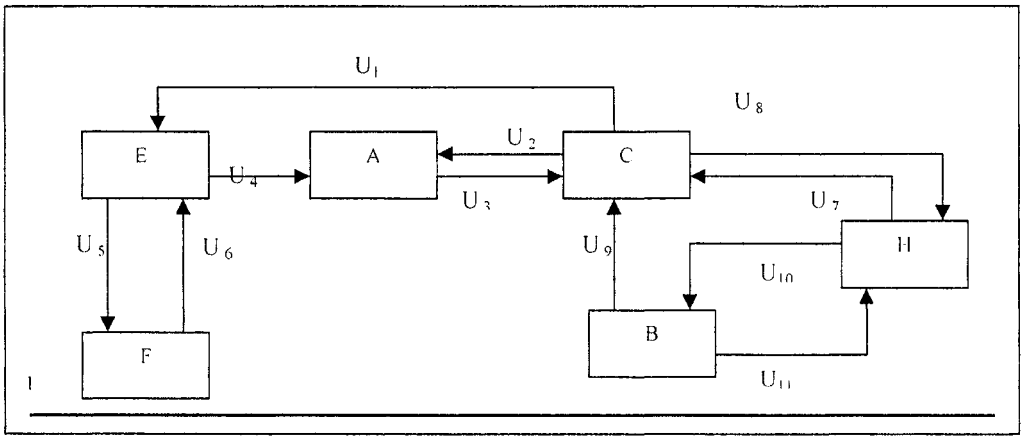


Figure 29: Diagramme du niveau 1 avant l'élimination des cycles de dépendances.

Analyse du niveau 1: Calcul des poids

Dans une première itération, le processus de calcul des poids nous donne, le poids de la classe C (3), comme étant le poids le plus élevé du SCC={A,B,C,E,F,H}. La classe C est donc choisie comme bouchon de test. Les liens U3, U9 et U7 sont alors retirés afin de sortir du cycle de dépendances constitué des classes {A,B,C,H,E}.

L'exemple de calcul de poids pour la classe C : $U_1 + U_2 + U_8 = 3$

Classes	Poids
A	1
B	2
C	3
E	2
F	1
H	2

Tableau 3: Résumé du poids des classes lors de la première itération.

Une seconde itération du processus de calcul du poids des classes est effectuée pour les cycles non éliminés lors de la première itération. Pour briser le cycle entre les classes E et F, nous pouvons utiliser la classe F comme bouchon de test (le poids de la classe E = 1, le poids de la classe F = 1). Sachant que la classe F utilise une classe au niveau 2, elle sera alors utilisée comme bouchon, le lien U5 sera alors retiré. Pour briser le cycle entre les classes B et H, nous considérons le cycle inter niveaux formé par le SCC {B, H, D} (se référer à la figure 10A). Nous calculons le poids des classes afin de briser les cycles de dépendances : $P_B = 2$, $P_D = 1$, $P_H = 1$. La classe B est utilisée comme bouchon de test et le lien U10 est retiré. La figure 22 représente le diagramme du niveau 1 après avoir cassé les cycles de dépendances.

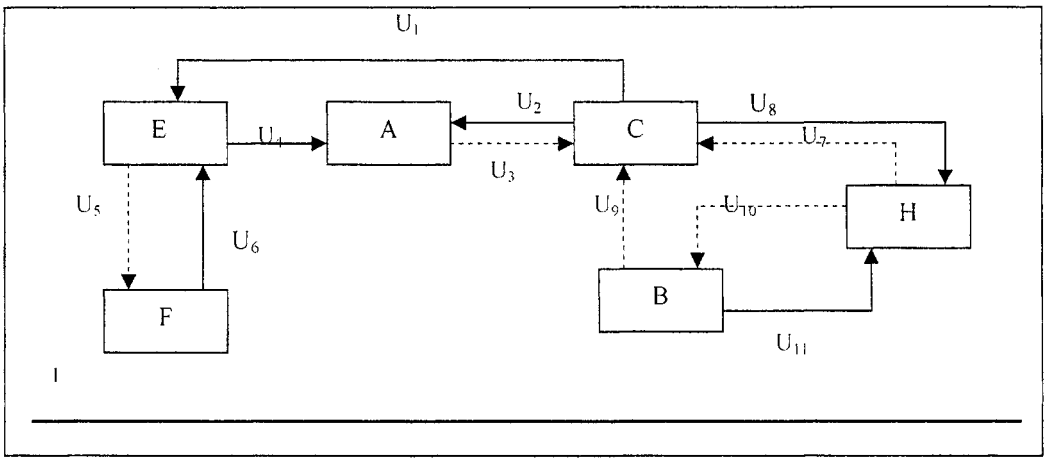


Figure 30: Diagramme du niveau 1 après avoir cassé les cycles de dépendances.

Analyse du niveau 2

Au niveau 2, il n'y a pas de cycle à éliminer.

4.2.4.3 Affectation d'un nombre niveau mineur

Analyse du niveau 1

Une fois tous les cycles correspondant à ce niveau majeur sont brisés, nous pouvons alors affecter les nombres *niveau mineur* provisoires (NminP). Le résultat de cette étape est fournit par la figure 31.

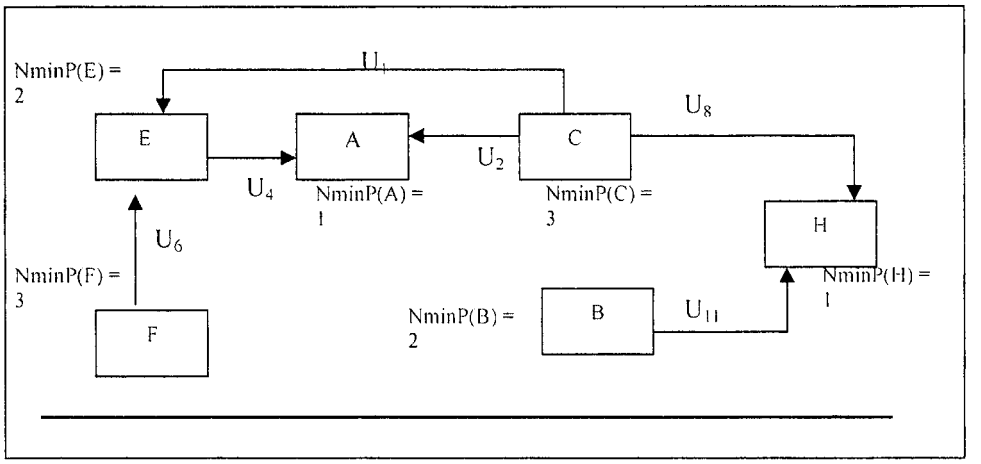


Figure 31: Diagramme du niveau N avec les nombres niveau mineur provisoires.

Analyse du niveau 2

Au niveau majeur 2, il n'y a pas de cycle à briser. Nous pouvons, donc, affecter directement le nombre *niveau mineur provisoire* (NminP). Les deux classes ont alors le même nombre *niveau mineur provisoire* (NminP). Nous procédons, à ce stade, à l'analyse inter niveaux afin d'affecter les nombres *niveau mineur définitifs* (NminD). L'affectation des nombres *niveaux mineurs définitifs* se fait exactement comme pour les *niveaux mineurs provisoires.* La seule différence réside dans la prise en compte des liens inter niveaux. Le résultat de cette étape est fournit par la figure 32.

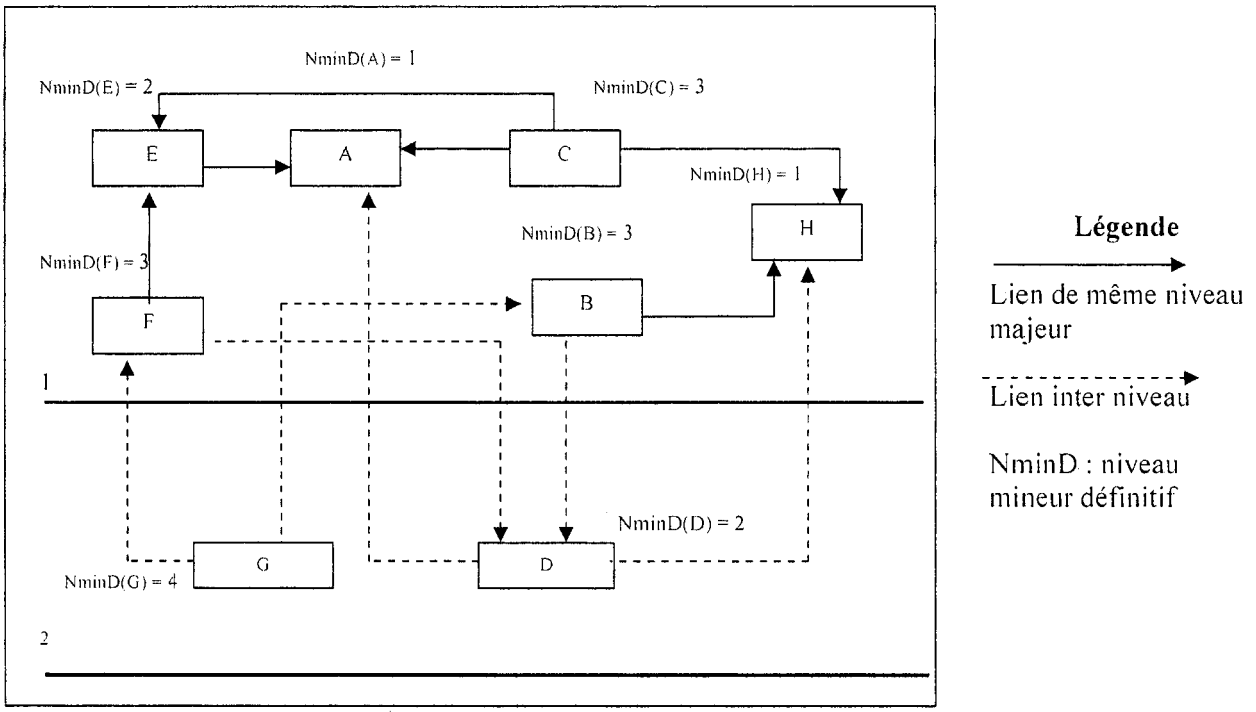


Figure 32: Diagramme contenant les relations inter niveaux avec les nombres niveau mineur définitifs.

4.2.4.4 Processus d'ordonnancement des classes

Nous représentons, dans le tableau 4, les classes du modèle considéré ainsi que leur couple de nombres *niveau majeur* et *niveau mineur*. Les différents couples de valeurs associés aux classes du modèle sont à la base du processus d'ordonnancement.

Classes	Nombres *niveau majeur*	Nombres *niveau mineur*
A	1	1
B	1	3
C	1	**3**
D	2	2
E	1	2
F	1	3
G	2	4
H	1	1

Tableau 4: Nombres niveau majeur et niveau mineur.

L'ordre d'intégration des classes du modèle considéré est le suivant:

1. A est intégrée en utilisant le bouchon de C,
2. H est intégrée en utilisant les bouchons de B et de C,
3. E est intégrée en utilisant le bouchon de F,
4. D est intégrée en utilisant les classes A et H,
5. C est intégrée en utilisant les classes A, E et H,
6. B est intégrée en utilisant les classes C, D et H,
7. F est intégrée en utilisant les classes D et E,
8. G est intégrée en utilisant les classes B et F.

Le nombre de bouchons réels engendrés par l'application de la stratégie que nous proposons est de 1. C'est la classe C qui sera choisie comme bouchon réel de test. Le nombre de bouchons spécifiques est de 2. Ce sont les méthodes m de B et m de F.

4.2.5 Résultat 1

Dans le tableau ci-dessous, nous présentons les résultats de l'application des différentes stratégies au modèle de la figure 20. Dans le modèle présenté à la figure 20, nous avons considéré uniquement des cycles de dépendance effectifs. Cela nous permet de faire une comparaison avec les autres stratégies dans le cas où nous avons uniquement des cycles de dépendance effectifs. Nous pouvons constater dans ce tableau que le nombre de bouchons engendrés par la nouvelle stratégie et celle de Le Traon, est inférieur à celui des trois autres stratégies à savoir, la stratégie Triskell, la stratégie de Briand et celle de Taï et Daniels.

Stratégies / Comparaison	Le traon et al	Triskell	Briand et al	Taï et Daniels	Nouvelle Stratégie
Bouchons réels	B, C, F	C, B, F	B, C, F ,H	C, F, H, D, B	C,B,F
Bouchons spécifiques	(A,C),(E,F) (H,C),(H,B)	(A,C),(E,F) (H,C),(H,B) (B,C)	(A,C),(E,F) (H,C),(H,B)	(C, A), (F, E), (H, C), (D, F), (B, H)	(A,C),(E,F) (H,C),(H,B)

Tableau 5: Résultats récapitulatifs.

4.3 Évaluation 2

Cette évaluation nous permet d'illustrer l'intégration d'un cycle de dépendance non effectif et celle d'un cycle effectif. Dans le modèle de la figure 20, tous les cycles du modèle ont été considérés comme des cycles de dépendance effectifs. Dans le modèle ci-dessous (figure 33), nous introduisons les deux types de cycles de dépendances (cycle effectif et non effectif). Ce modèle nous permet de mettre en évidence les apports de la nouvelle stratégie, d'une part, et de faire une comparaison avec les autres stratégies présentées précédemment, d'autre part.

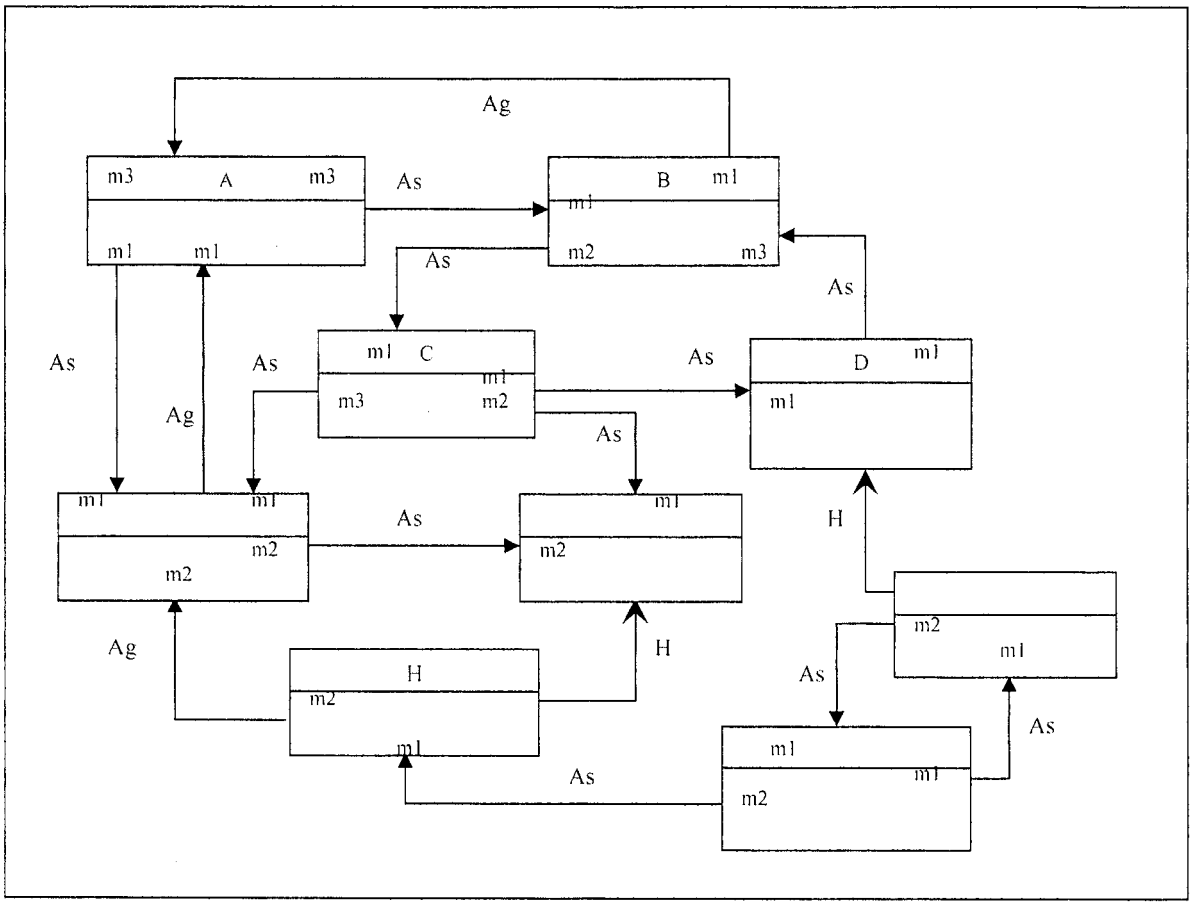


Figure 33 : Diagramme ORD + les méthode sollicitées

4.3.1 Application de la stratégie de Briand et al.

La stratégie de Briand et al [Bri01], appliquée au modèle de la figure 33, donne l'ordre d'intégration suivant :

1. E est intégrée avec aucun bouchon

2.1 D est intégrée avec un bouchon de B

F est intégrée avec un bouchon de A

C est intégrée avec aucun bouchon

H est intégrée avec aucun bouchon

2.1.1 B est intégrée avec un bouchon de A

2.1.2 I est intégrée avec un bouchon de G

G est intégrée avec aucun bouchon

2.2 A est intégrée avec aucun bouchon

Le nombre de bouchons réels est de trois (classes A, B et G). Par ailleurs, le nombre de bouchons spécifiques est de quatre: (F, A), (D, B), (I, G) et (B, A).

4.3.2 Application de la stratégie Triskell

La stratégie Triskell [Hah01], appliquée au modèle de la figure 33, donne les résultats fournis par le tableau 6. Le nombre de bouchon réels est de trois. Ce sont les classes A, C et G. Par ailleurs, le nombre de bouchons spécifiques est de quatre. Ce sont les liens (B,A),(G,A),(B, C) et (I, G) .

classes	profondeur	Ordre de test	Bouchons
A	1	7	aucun
B	3	3	A et C
C	1	8	aucun
D	2	5	aucun
E	5	1	aucun
F	4	2	A
G	1	9	aucun
H	3	4	aucun
I	2	6	G

Tableau 6: Résultats récapitulatifs du résultat de la stratégie Triskell.

4.3.3 Application de la stratégie de Taï et Daniels

La stratégie de Taï et Daniels [Taï99], appliquée au modèle de la figure 33, donne l'ordre d'intégration suivant :

1. B est intégrée avec un bouchon de A et de C
2. E est intégrée avec aucun bouchon
3. I est intégrée avec un bouchon de H et de G
4. D est intégrée avec aucun bouchon
5. F est intégrée avec un bouchon de A
6. A est intégrée avec aucun bouchon
7. C est intégrée avec aucun bouchon
8. G est intégrée avec aucun bouchon

9. H est intégrée avec aucun bouchon

Le nombre de bouchon réels est de quatre. Ce sont les classes A, C,H et G. Par ailleurs, le nombre de bouchons spécifiques est de quatre. Ce sont les liens (B,A),(G,A),(B, C) et (I, G) .

4.3.4 Application de la nouvelle stratégie

La nouvelle stratégie, appliquée au modèle de la figure 33, donne l'ordre d'intégration suivant :

1. E est intégrée (intégralement) avec aucun bouchon
2. B est intégrée (partiellement) avec un bouchon spécifique de A.
3. D est intégrée (intégralement) avec aucun bouchon
4. F est intégrée (intégralement) avec un bouchon spécifique de A
5. A est intégrée (intégralement) avec aucun bouchon
6. C est intégrée (intégralement) avec aucun bouchon
7. B est intégrée (intégralement) avec aucun bouchon
8. H est intégrée (intégralement) avec aucun bouchon
9. G est intégrée (partiellement) avec aucun bouchon
10. I est intégrée (intégralement) avec aucun bouchon
11. G est intégrée (intégralement) avec aucun bouchon

Le nombre de bouchon réels est de un, c'est la classe A. Par ailleurs, le nombre de bouchons spécifiques est de deux, ce sont les liens (B,A),(F,A).

4.3.5 Résultat 2

Dans le tableau ci-dessous, nous présentons les résultats de l'application des différentes stratégies au modèle de la figure 33. Nous pouvons constater dans ce tableau que le nombre de bouchons engendrés par la nouvelle stratégie est nettement inférieur à celui des autres stratégies.

Stratégies / Comparaison	Le traon et al.	Triskel	Briand et al	Taï et Daniels	Nouvelle Stratégie
Bouchons réels	A, C, G	A, C et G	A, C, G	A, C, H , G	A
Bouchons spécifiques	(B,A), (G, A), (B,C), (I, G)	(B,A),(G,A) ,(B, C), (I, G)	(B,A), (G, A), (B,C), (I, G)	(B,A), (G,A), (B, C) , (I, G)	(B,A), (F,A)

Tableau 7: Résultats récapitulatifs des stratégies appliquées à la figure 24

Nous présentons, dans le tableau 8, un récapitulatif des différentes comparaisons effectuées, lors de l'évaluation 1 et 2. Nous comparons, dans ce tableau, les résultats des différentes stratégies appliquées aux modèle de la figure 20 (modèle 1 dans le tableau, à la colonne **Modèle**) et de la figure 33 (modèle 2 dans le tableau, à la colonne **Modèle**)

Stratégie	Modèle	Nombre de bouchons		Remarques
		réels	spécifiques	
Taï et Daniels [Taï99]	1	5	5	Le nombre de bouchons engendrés par cette stratégie est le plus élevé de toutes les stratégies présentées. Cela est du à la conception de bouchons sans la présence de cycles. Le bouchon de la classe D aurait pu être évité lors de l'intégration de la classe F (dans le modèle 1), car il n'y à aucun cycle qui implique simultanément les deux classes.
	2	4	4	
Le Traon et al. [Tra00]	1	3	4	Le nombre de bouchon est semblable pour les stratégies de Briand, Triskell ainsi que celle de Le Traon et al. La seule différence entre les trois stratégies se situe au niveau de la prise en compte de la complexité des bouchons de test (Triskell) et de l'instabilité des résultats.(Le Traon et al).
	2	3	4	
Briand et al. [Bri01]	1	4	4	
	2	3	4	
Triskell [Hah01]	1	3	5	
	2	3	4	
Nouvelle Stratégie	1	3	4	La nouvelle stratégie prend en compte la complexité des bouchons de test. Les détails contenus dans le modèle de dépendances considéré permettent d'avoir une idée précise des multiples interactions entre les classes du système et par conséquent de mieux appréhender la complexité des bouchons de test. Par ailleurs, la stratégie prend aussi en considération la minimisation des bouchons de test. La notion de cycle effectif permet d'éliminer plusieurs cycles de dépendances sans engendrer de bouchons de test. Durant le processus d'intégration, une combinaison peut être effectuée entre les bouchons spécifiques et les bouchons réels. Ceci est du essentiellement à la bonne connaissance des origines des dépendances entre les classes du modèle ainsi que de leur détail.
	2	1	2	

Tableau 8 : Récapitulatif de l'évaluation théorique.

4. 4 Évaluation 3

Les différentes applications que nous utiliserons sont des applications réelles Java. Elles ont en moyenne 30 classes, 150 méthodes et 100 interactions entre classes. Il existe plusieurs critères de comparaison qui permettent d'évaluer les différentes stratégies, les unes par rapport aux autres. Nous pouvons citer entre autres, le nombre de bouchons de test (les bouchons spécifiques et les bouchons réels), le nombre d'étapes d'intégration, la complexité des bouchons de test, etc. Dans ce travail, nous avons mis plutôt l'accent sur le nombre de bouchons. Le nombre de bouchons spécifiques et le nombre de bouchons réels nous permettront d'avoir une très bonne idée de l'effort de test. Nous présenterons dans cette section, les résultats obtenus pour les stratégies évaluées à partir de chacune des applications. Ensuite, nous procéderons à une comparaison de ces différentes stratégies en nous basant sur les résultats obtenus.

4.4.1 Présentation des différentes applications utilisées

Toutes les applications que nous avons utilisées pour faire notre évaluation sont des applications développées en java. Nous avons au total quatre applications : le système ATM, le système Bank, le système Jadvisor et l'application Inquisitor. Nous présentons, ci-dessous, certaines caractéristiques importantes de ces systèmes à savoir, leur nombre de classes, leur nombre de méthodes ainsi que leur nombre d'interactions.

- **Le Système ATM**

Le système ATM est une application qui permet de simuler des transactions bancaires. Elle comporte :

Nombre de classes	21	classes
Nombre de méthodes	84	méthodes
Nombre d'interactions	142	interactions

- **Le Système Bank**

Le système Bank est une application qui permet elle aussi de simuler des transactions bancaires.

Nombre de classes	21	classes
Nombre de méthodes	105	méthodes
Nombre d'interactions	78	interactions

- **Le jeu Inquistor**

C'est un jeu de combat 3D avec plusieurs personnages et paysages. Cette application comporte:

Nombre de classes	38	classes
Nombre de méthodes	194	méthodes
Nombre d'interactions	462	interactions

- **Le Système Jadvisor**

C'est une application permettant de faire la planification des cours pour les étudiants. Cette application comporte:

Nombre de classes	30 classes
Nombre de méthodes	358 méthodes
Nombre d'interactions	253 interactions

4.4.2 Présentation des critères de comparaison

Comparer la nouvelle stratégie avec celles existantes est une nécessité si l'on veut mettre en relief son apport. Plusieurs critères de comparaison nous permettent de faire cette évaluation. Dans ce rapport, nous avons choisit de prendre comme critère, le nombre de bouchons de test réels et le nombre de bouchons de test spécifiques. Selon nous, ces deux critères nous permettent d'avoir une bonne appréciation de l'effort de test.

Les bouchons de test réels

Un bouchon de test réel consiste en la simulation du composant en entier. La conception de ce type de bouchon peut s'avérer complexe car elle nécessite la prise en charge de tous les services fournis par le composant simulé. Une fois conçu, le bouchon

réel peut être utilisé pour l'intégration de n'importe quelle classe du système qui dépend du composant simulé.

L'avantage de tels bouchons de test est la réduction du nombre de bouchons à gérer et du nombre de bouchons à concevoir pour un même composant. Cependant, il peut être complexe à concevoir et peut être source d'erreurs, ce qui peut entacher la qualité du test.

- **Les bouchons de test spécifiques**

Un bouchon de test spécifique consiste en la simulation d'un ou de plusieurs services du composant. La conception de ce type de bouchon est relativement simple comparée à celle d'un bouchon réel. Cependant, elle ne peut être utilisée que pour l'intégration d'un seul composant.

La relative simplicité de sa conception peut être vue comme un avantage important dans la mesure où elle peut permettre d'éviter ou de réduire les sources d'erreurs. L'inconvénient que présentent les bouchons spécifiques, c'est leur multiplicité pour une même classe. Cela peut poser problème lors de la gestion des bouchons de test et dans certains cas être source de redondance, car le même service pourrait être demandé par plusieurs composants. Le problème ici c'est qu'il faut un bouchon spécifique pour l'intégration de chaque composant dépendant du composant simulé. Ce problème pourrait être évité dans la nouvelle stratégie à partir du moment où elle identifie clairement la ou les méthodes impliquées dans la relation de dépendance.

En plus du nombre, la complexité du bouchon de test peut être un indicateur très important de l'effort de test. La démarche utilisée dans le choix des bouchons de test diffère d'une stratégie à une autre. Cette différence dans la démarche peut engendrer une différence de la complexité des bouchons de test d'une stratégie à une autre.

4.4.3 Les stratégies comparées

Nous avons choisi trois différentes stratégies pour faire notre évaluation. Selon Nous, ces trois stratégies sont représentatives des stratégies existantes et dans certaines mesures, elles sont les plus intéressantes. Ce sont: la stratégie proposée par Taï et Daniels, la stratégie proposée par Briand et al, et la stratégie Triskell. Nous comparons ces stratégies entre elle et avec la nouvelle stratégie nommée B3 que nous avons proposée.

4.4.4 Présentation des résultats obtenus pour chaque stratégie

- **Les résultats de l'étude de cas 1 : Application ATM**

Dans le tableau 9, nous présentons le nombre de bouchons spécifiques et le nombre de bouchons réels obtenus par application des différentes stratégies sur le système ATM. Le nombre de bouchons réels est de zéro pour la stratégie B3 et de trois pour les autres stratégies. Le nombre de bouchons spécifiques est de six pour la stratégie de Taï et Daniels et la stratégie Triskell. Avec la stratégie de Briand, nous obtenons cinq bouchons spécifiques. Le nombre de bouchons pour la stratégie B3 est de zéro.

Stratégies / Critères	Triskell	Briand et al.	Taï et Daniels	B3
Nombre de bouchons réels	2	3	2	0
Nombre de bouchons spécifiques	4	3	4	0

Tableau 9 : Les résultats obtenus avec le système ATM

- **Les résultats de l'étude de cas 2 : Application BANK**

Dans le tableau 10, nous présentons le nombre de bouchons spécifiques et le nombre de bouchons réels obtenus par application des différentes stratégies sur le système BANK. Le nombre de bouchons réels est de zéro pour la stratégie B3, de trois pour la stratégie de Briand et de deux pour la stratégie Triskell et celle de Taï et Daniels. Le nombre de bouchons spécifiques est de quatre pour la stratégie de Taï et Daniels et Triskell. Pour la stratégie de Briand, nous obtenons trois bouchons spécifiques. Le nombre de bouchons pour la stratégie B3 est de zéro.

Stratégies / Critères	Triskell	Briand et al.	Taï et Daniels	B3
Nombre de bouchons réels	3	3	3	0
Nombre de bouchons spécifiques	6	5	6	0

Tableau 10 : Les résultats obtenus avec le système BANK

- **Les résultats de l'étude de cas 3 : Application Inquisitor**

Dans le tableau 11, nous présentons le nombre de bouchons spécifiques et le nombre de bouchons réels obtenus par application des différentes stratégies sur le système Inquisitor. Le nombre de bouchons réels est de zéro pour la stratégie B3, de trois pour la stratégie de Briand, de deux pour la stratégie Triskell et de huit pour la stratégie de Taï et Daniels. Le nombre de bouchons spécifiques est de quinze pour la stratégie de Taï de Daniels et de cinq pour la stratégie Triskell et celle de Briand. Le nombre de bouchons pour la stratégie B3 est de zéro.

Stratégies / Critères	Triskell	Briand et al.	Taï et Daniels	B3
Nombre de bouchons réels	2	3	8	0
Nombre de bouchons spécifiques	5	5	15	0

Tableau 11 : Les résultats obtenus avec l'application Inquisitor

- **Les résultats de l'étude de cas 4 : Application Jadvisor**

Dans le tableau 12, nous présentons le nombre de bouchons spécifiques et le nombre de bouchons réels obtenus par application des différentes stratégies sur le système JAdvisor. Le nombre de bouchons réels est de zéro pour la stratégie B3, de deux pour la stratégie de Briand, de un pour la stratégie Triskell et de seize pour la stratégie de Taï et Daniels. Le nombre de bouchons spécifiques est de trente et un pour la stratégie de Taï de Daniels et de cinq pour la stratégie Triskell et celle de Briand. Le nombre de bouchons pour la stratégie B3 est de zéro.

Stratégies / Critères	Triskell	Briand et al.	Taï et Daniels	B3
Nombre de bouchons réels	1	2	16	0
Nombre de bouchons spécifiques	5	5	31	0

Tableau 12 : Les résultats obtenus avec le système JAdvisor

4.4.5 Interprétation et comparaison des résultats obtenus

Les graphiques ci-dessous nous permettent de mieux apprécier le nombre de bouchons obtenus en appliquant chacune des stratégies aux quatre systèmes utilisés.

- **Le nombre de bouchons réels**

La figure 34 illustre le nombre de bouchons réels pour chaque stratégie. Le nombre de bouchons engendrés par la stratégie B3 est de zéro pour toutes les applications utilisées. Alors que le nombre de bouchons varient de un à trois pour la stratégie de Briand et la stratégie Triskell, la stratégie de Taï et Daniels à un nombre de bouchons avec une très grande marge de variation (de deux à seize). Afin de mieux comparer ces différentes stratégies, faisons la moyenne du nombre de bouchons pour chaque stratégie sur le nombre total d'applications utilisées.

Moyenne (S) = nombre de bouchons de (S)/ nombre d'applications

La figure 35 illustre la moyenne (s) de chaque stratégie. À partir de cette moyenne, nous pouvons avoir une idée de l'efficacité de chaque stratégie.

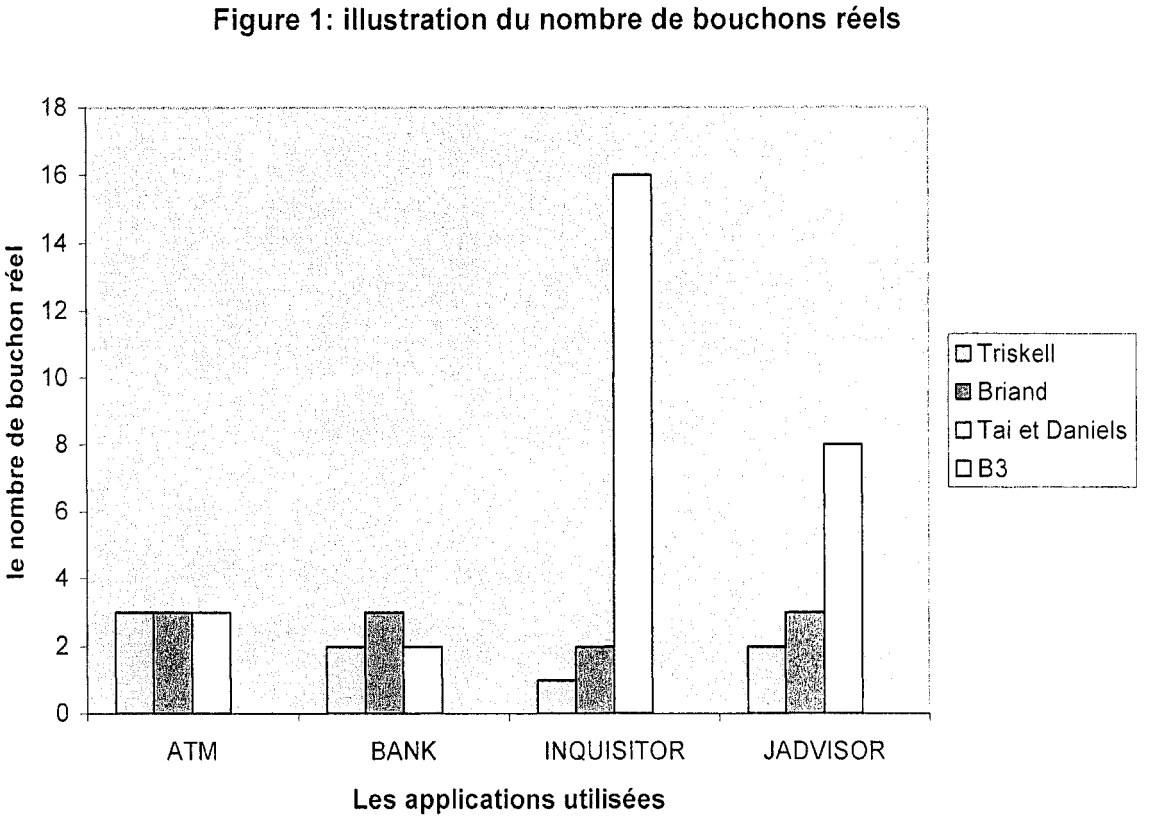


Figure 34: Illustration du nombre de bouchons réels

À la lecture de la figure 35, nous pouvons avancer que la stratégie B3 est de loin la meilleure des quatre stratégies, ensuite viennent dans l'ordre, la stratégie Triskell, la stratégie de Briand et la stratégie de Taï et Daniels de loin la moins efficace.

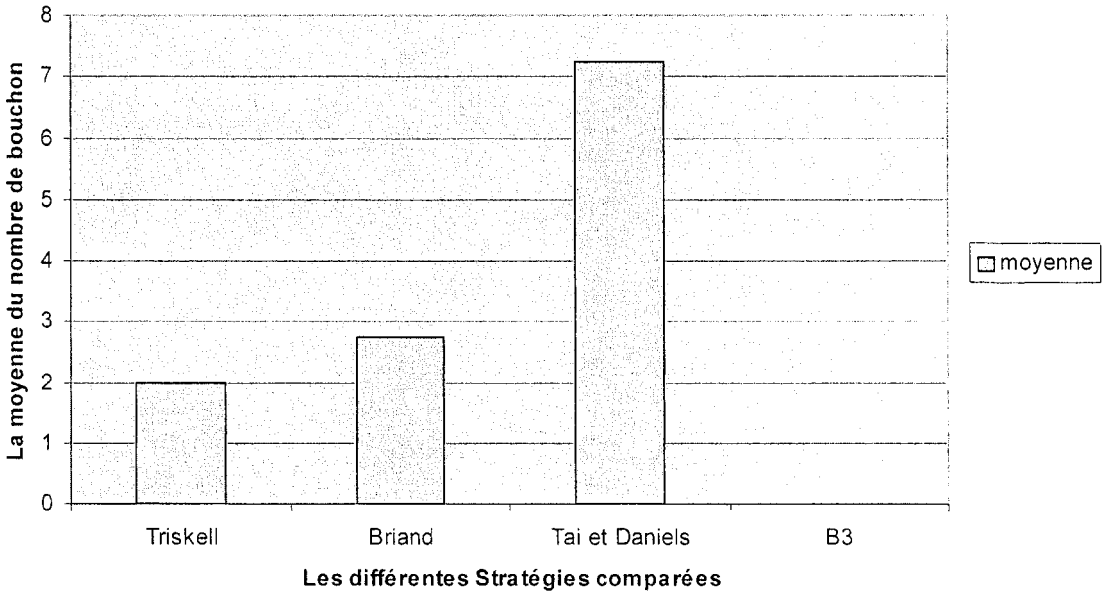


Figure 35: Illustration de la moyenne (S)

- **Le nombre de bouchons spécifiques**

La conception d'un bouchon spécifique est relativement simple comparée à celle d'un bouchon réel. Il est alors plus intéressant d'utiliser des bouchons spécifiques. Cependant, si leur nombre est élevé, la charge de travail pourrait être la même que celle de la conception d'un bouchon réel. Dans le graphique de la figure 36, nous mettons en relief le nombre de bouchons spécifiques pour chaque stratégie.

Figure 3 : Illustration du nombre de bouchons spécifiques

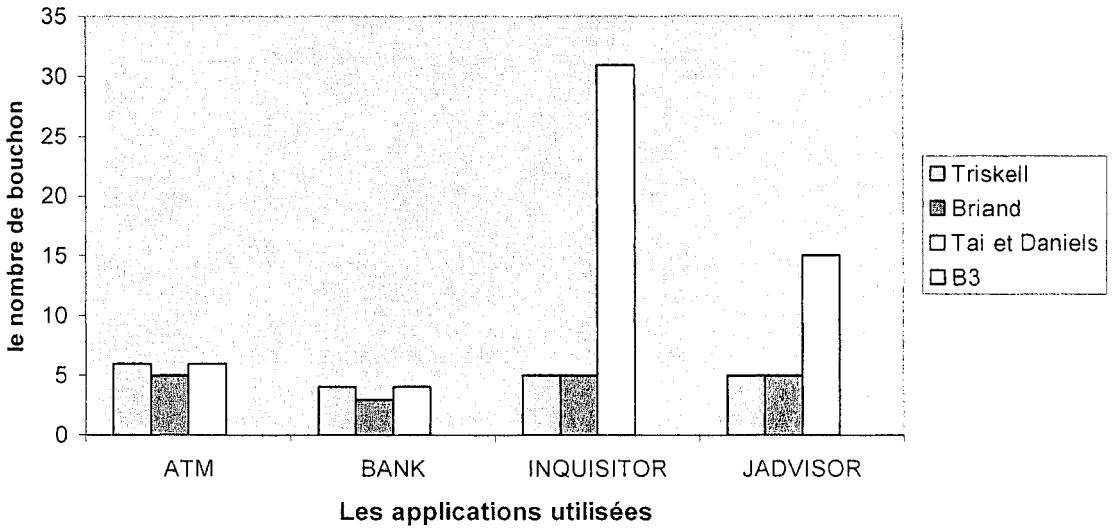


Figure 36: Illustration du nombre de bouchons spécifiques

La figure 37 ci-dessous nous montre, la moyenne du nombre de bouchons spécifiques pour chaque stratégie. Mise à part la stratégie de Taï et Daniels, l'utilisation de bouchons spécifiques serait intéressante dans les autres stratégies.

Figure 4: Illustration de la moyenne

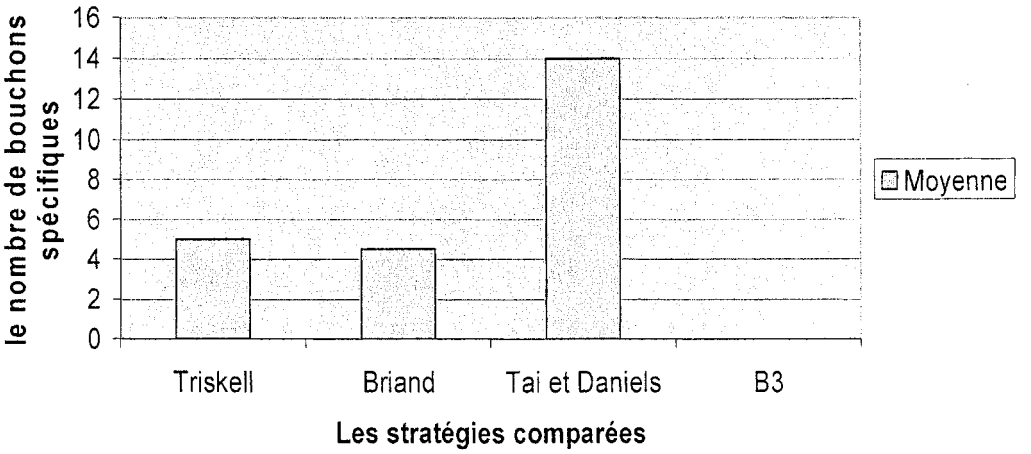


Figure 37: Illustration de la moyenne

Les résultats présentés dans cette section démontre que la stratégie B3 est meilleure que les trois autres stratégies. La moins efficace parmi les stratégies comparées, est la stratégie de Taï et Daniel. Dans les deux premières applications (ATM et BANK), elle se comporte relativement bien par rapport aux stratégies de Briand et et la stratégie Triskell. Dans l'application BANK, elle à même moins de bouchons que la stratégie de Briand.

Le nombre de bouchons est cependant beaucoup plus élevé dans les deux dernières applications (Inquisitor et Jadvisor). Cela est du à l'une des règles qu'utilise Taï et Daniels pour hiérarchiser le ORD à savoir, *lorsqu'une classe d'un niveau majeur N utilise une classe d'un niveau majeur N+1*, la classe du niveau majeur N+1 est utilisée comme bouchon. Lorsque nous analysons les modèles de l'application Inquisitor et de l'application Jadvisor, nous nous aperçevons que des classes de niveau majeur N

utilisent plusieurs classes du niveau majeur N+1. Dans l'application Inquisitor une seule classe du niveau majeur N utilise plusieurs classes du niveau majeur N+1.

L'utilisation de la stratégie B3 n'a engendré aucun bouchon dans toutes les applications. Cela est du aux concepts de ***l'intégration partielle*** et du ***cycle de dépendance non effectif***. À l'analyse des MDC de toutes les applications, nous n'avons trouvé aucun cycle effectif, cela nous évite la création de bouchons.

CHAPITRE 5

CONCLUSION ET TRAVAUX FUTURS

Les stratégies d'intégration orientées objet présentées dans ce mémoire sont, dans l'ensemble, de bonnes stratégies dans la mesure où elles supportent, chacune à sa façon, le processus d'intégration des classes et permettent quelque part de réduire l'effort et les coûts alloués au test d'intégration. Elles permettent, en effet, de planifier le test d'intégration des classes tout en indiquant les différents bouchons de test à simuler afin de mener à bien le test. Cependant, prise individuellement, elles présentent chacune des points forts ainsi que des points faibles.

La stratégie que nous proposons (nommée B3) est une stratégie basée sur un nouveau modèle de dépendances nommé MDC (Modèle de Dépendances entre les Classes) et un algorithme d'intégration permettant de supporter des concepts tels que l'intégration partielle ainsi que le cycle de dépendances effectif et le cycle de dépendances non effectif. La combinaison de la méthodologie d'intégration (algorithme) et du nouveau modèle MDC, permet d'avoir une stratégie relativement plus efficace

aussi bien dans la planification des étapes du test d'intégration que dans l'identification des bouchons de test. L'apport de la nouvelle stratégie a été mis en relief dans le chapitre 4. Elle donne des résultats plus intéressants. En termes du nombre de bouchons de test, la stratégie B3 est de loin la plus efficace. Dans l'évaluation 3 (évaluation faite sur des applications réelles) au chapitre 4, la moyenne de bouchons de test nécessaires pour la stratégie B3 est de 0 bouchon réel et de zéro bouchon spécifique. Alors que la meilleure moyenne de bouchons des autres stratégies est de 2 bouchons réels et de 5 bouchons spécifiques. La plus élevée des moyennes des autres stratégies est de 7 bouchons réels et de 14 bouchons spécifiques. Par ailleurs, ces stratégies font état de bouchons spécifiques. Cependant, les informations contenues dans les modèles considérés (ORD) ne permettent pas d'identifier précisément les dépendances (en termes d'interactions entre classes). Ce manque d'information peut altérer l'efficacité dans la simulation des composants [Mil02]. Par ailleurs, le TGD présenté dans [Tra00] prend en compte plusieurs niveaux d'abstraction. Cela peut permettre d'avoir une bonne idée des interactions entre les classes ou les méthodes. Cependant, la plus part des stratégies qui l'utilisent ne combinent pas les détails apportés par le modèle et leur processus d'intégration. La prise en compte de ces détails pour certaines stratégies, ne peut se faire sans engendrer des coûts supplémentaires lié à leur modification. C'est le cas pour la stratégie Triskell [Hah01].

La complexité des bouchons de test est réduite avec la stratégie B3 car le modèle MDC permet non seulement d'identifier l'origine des dépendances, mais surtout d'avoir une

très bonne idée de la densité des interactions entre les classes. Ces détails apportés par le MDC nous permettent de prendre des décisions éclairées lors du choix des bouchons.

Nous avons développé un outil **(JInt)** en java permettant de supporter la nouvelle stratégie que nous préconisons. Nous avons aussi implémenté les algorithmes d'autres stratégies à savoir, la stratégie de Briand, la stratégie Triskell ainsi que la stratégie de Taï et Daniels. Ces différents outils nous ont permis de faire l'évaluation 3 présentée au chapitre 4. L'outil JInt prend en entrée des modèles de dépendances. Durant notre expérimentation, ces modèles ont été extraits du code des applications considérées à l'aide d'un outil de ré-ingénierie (Together).

RÉFÉRENCES

[Bad05] L. Badri, M. Badri & V.S. Blé, A Method Level Based Approach for OO Integration Testing: An Experimental Study, Proceedings of the Sixth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'2005), IEEE Computer Society Press, Maryland, USA, May 2005, pp.102-109

[Bad04] L. Badri, M. Badri & V.S. Blé, Object-Oriented Integration Testing: A Method Level Approach, Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (SEA'04), MIT, Cambridge, USA, Novembre 2004, pp.324-330

[Bad95] Mourad Badri, Linda Badri and Soumia Layachi, "Vers une stratégie de tests unitaires et d'intégration des classes dans les applications orientées objet", Revue Génie Logiciel, N. 38, 1995.

[Bei90] Beizer, Boris., "Software Testing Techniques. Second Edition.", *Van Nostrand Reinhold*, New York , 1990.

[Ber97] A. Bertolino, P. Inverardi, H. Muccini, A. Rosetti, "An approach to integration testing based on architectural descriptions", Proc. of the Third

IEEE International Conference on Engineering of Complex Computer Systems, Sept 1997, pp.77-84.

[Bin94] Rober V. Binder, "Design for testability in object-oriented systems", *Communication of ACM*, Vol. 37, Sept 1994, pp. 87-100.

[Bin99] Robert V. Binder, "Testing Object-Oriented Systems, Models, Paterns and Tools", *Addison Wesley*, October 1999.

[Bri01] L. Briand, Y. Labiche and Y. Wang, "Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles", Proc. Of the 12th *International Symposium on Software Reliability Engineering (ISSRE'2001)*, Hong Kong, Nov 2001, pp. 287-296.

[Bri02] L. C. Briand, J. Feng and Y. Labiche, "Using Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders", Proc. of the *14th ACM International Conference on Software Engineering and Knowledge Engineering*, Itchier (Italy), July 2002, pp. 43-50.

[Bri03] L. Briand, J. Feng and Y. Labiche, "Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Test Orders", *Software Engineering with Computational Intelligence*, Kluwer, 2003.

[Kun95] D. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima, "Class firewall, test order, and regression testing of object oriented programs", Journal of Object Oriented Programming, Vol 8(2), 1995, pp. 51-65.

[Taï99] Kuo Chung Tai and Fonda J. Daniels, "Interclass Test Order for Object-Oriented Software", *Journal of Object-Oriented Programming*, vol. 12 (4), 1999.

[Hah01] V. Le Hahn, K. Akif, Y. Le Traon and J. M. Jézéquel, "Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies", *15th European Conference for Object-Oriented Programming*, Budapest, June 2001.

[Har92] Mary Jean Harrold, John D. McGregor, and Kevin Fitzpatrick, "Incremental Testing for Object-Oriented Class Structures", *14th International Conference on Software Engineering*, IEEE Computer Society, CA, May 1992, pp. 68-80.

[Jor94] Paul C. Jorgensen and Carl Erikson, "Objet-Oriented Integration Testing", *Communications of the ACM*, V. 37 (9), Sept. 1994, pp. 30-38.

[Lab97] Y. Labiche, "On testing object-oriented programs", *Proc. ECOOP Workshop for Doctoral Students in Object-Oriented Systems*, Jyvaskyla (Finlande), June 1997.

[Lab00] Y. Labiche, P. Thévenod-Fosse, H. Waeselynck and M.H. Durand, "Testing Levels for Object-Oriented Software", *22nd IEEE International Conference on Software Engineering (ICSE'2000)*, Limerick (Ireland), June 2000, pp. 136-145.

[Lar01] G. Larman, "Applying UML and Design Patterns, An introduction to OO analysis and design", Prentice Hall, 2001.

[McG94] John D. McGregor and Tin Korson, "Integrating Objet-Oriented Testing and Development Processes", *Communications of the ACM*, Vol. 37(9), Sept. 1994, pp. 59-77.

[Mil02] A. Milanova, A. Rountev and Barbara G. Ryder, "Constructing Precise Object Relation Diagrams", *International Conference on Software Maintainance*, ICSM'02, October 2002.

[Tar72] R. Tarjan, "Depth-first search and linear graph algorithms", *SIAM J. Comput.*, Vol.1, n 2, June 1972, 146-160, ISSN 1064-8275.

[Tra00] Y. Le Traon, J. M. Jézéquel and P. Morel, "Efficient Object-Oriented Integration And Regression Testing", *IEEE Transactions on Reliability*, vol. 49 (1), March 2000, pp. 12-25.

[Off96] Z. Jin, A. J. Offutt, "Coupling-based integration testing" , Proc. of the Second IEEE International Conference on Engineering of Complex Computer Systems, Oct. 1996, pp. 10-17.