

Table des Matières

ABBREVIATIONS	8
LISTE DES FIGURES.....	9
LISTE DES TABLEAUX.....	10
I. INTRODUCTION GÉNÉRALE.....	11
A. CONTEXTE	11
B. PROBLÉMATIQUE	12
C. TRAVAUX CONNEXES.....	13
D. ORGANISATION DE LA THÈSE	16
II. EVOLUTION DE LOGICIEL	18
1. INTRODUCTION.....	18
2. ÉVOLUTION VERSUS MAINTENANCE	18
3. TAXONOMIE DE LA MAINTENANCE ET DE L'ÉVOLUTION DES LOGICIELS.	20
a) <i>Caractéristiques de la maintenance de logiciel</i>	20
b) <i>Réutilisation des modèles orientés</i>	21
c) <i>Itératif Vs. Incrémentale</i>	22
d) <i>Evolution de logiciel</i>	24
e) <i>Maintenance de logiciel</i>	26
f) <i>Normes de maintenance de logiciel</i>	28
g) <i>Modèles et processus d'évolution des logiciels</i>	29
a) <i>Modèles itératifs</i>	30
b) <i>Modèles de changement en mini-cycles</i>	30
4. GESTION DE L'ÉVOLUTION DE LOGICIEL	33
a) <i>Gestion de configuration logicielle</i>	33
b) <i>Réingénierie</i>	34
c) <i>Rétro ingénierie</i>	36
d) <i>Analyse d'impact</i>	37
5. CRITÈRES D'ÉVALUATION DE LA QUALITÉ DU CODE SOURCE D'UN LOGICIEL	37
6. LES MÉTRIQUES DE LOGICIEL	38
a) <i>Différents types de métriques</i>	39
b) <i>Quelques métriques de conception orientée objet</i>	39
7. CONCLUSION.....	42
III. IMPACT DE CHANGEMENT.....	43
1. INTRODUCTION.....	43

Tables des Matières

2.	DÉFINITION DE L'IMPACT DE CHANGEMENT	43
3.	PROCESSUS DU CHANGEMENT	43
4.	DÉFINITIONS DE L'ANALYSE DE L'IMPACT DE CHANGEMENT	44
5.	INTÉRÊTS DE L'ANALYSE DE L'IMPACT DE CHANGEMENT.....	45
6.	TECHNIQUES D'ANALYSE D'IMPACT	46
a)	<i>Analyse de traçabilité</i>	46
b)	<i>Analyse de dépendance</i>	47
c)	<i>Analyse de l'effet d'entraînement</i>	47
d)	<i>Propagation du changement</i>	48
7.	EFFET DE VAGUE''RIPPLE EFFECT ''	48
8.	APPROCHES D'ANALYSES D'IMPACT DE CHANGEMENT	49
9.	EVALUATION ET PROPAGATION D'IMPACT	51
10.	OUTILS D'ANALYSE D'IMPACT	52
a)	<i>Chianti</i>	52
b)	<i>FaultTracer</i>	53
c)	<i>ImpactMiner</i>	54
d)	<i>JRipples</i>	55
e)	<i>Lattix</i>	56
f)	<i>JArchitect</i>	56
11.	CONCLUSION	57
IV.	LA TECHNIQUE DE PLAN D'EXPÉRIENCES "DESIGN OF EXPERIMENTS" (DOE)	58
1.	INTRODUCTION	58
2.	DÉFINITION.....	58
3.	ESPACE EXPÉRIMENTAL.....	59
4.	MODÉLISATION MATHÉMATIQUE	62
5.	DOMAINE EXPÉRIMENTAL	63
6.	LES PLANS D'EXPÉRIENCES.....	64
a.	<i>Plans factoriels complets à deux niveaux</i>	65
b)	<i>Plans à plusieurs niveaux</i>	68
c)	<i>Plans pour surfaces de réponse</i>	69
d)	<i>Plans de mélanges</i>	72
e)	<i>Plans Booléens</i>	72
f)	<i>Plans pour simulations numériques</i>	73
7.	CONCLUSION.....	73
V.	EXPÉRIMENTATION ET RÉSULTATS	75
1)	INTRODUCTION.....	75
2)	MÉTRIQUES DE COUPLAGE	75

Tables des Matières

a) <i>Coupling Between Objects (CBO)</i>	76
b) <i>Response For a Class (RFC)</i>	76
c) <i>Message Passing Coupling (MPC)</i>	76
3) MÉTHODOLOGIE DE RECHERCHE.....	77
4) CONTEXTE DE L'ÉTUDE	78
5) MODÉLISATION D'ÉQUATIONS MATHÉMATIQUES	79
6) VALIDATION EMPIRIQUE	80
7) PLAN D'EXPÉRIENCES	81
8) OUTIL PROTOTYPE.....	82
9) AJUSTEMENT DES PARAMÈTRES POUR LA DoE	83
10) APPLICATION DES PARAMÈTRES DoE AJUSTÉS	85
11) EXÉCUTION DE LA MÉTHODE DoE	87
12) TEST DE ROBUSTESSE DU MODÈLE	89
13) ANALYSE DE DONNÉES	90
14) CONCLUSION	92
VI. CONCLUSIONS ET PERSPECTIVES.....	94
A. CONCLUSIONS	94
B. PERSPECTIVES	96
RÉFÉRENCES	97
ANNEXES.....	107

Abbreviations

SDLC: Software Developement Life Cycle

SMLC: Software Maintannace Life Cycle

DoE: Design of Experiments

CR : Change Request

CM : Change Management

SCM : Software Change Management

MR : Modification Request

EIC : Estimation de l'Impact de Changement

CBO: Coupling Between Objects

RFC: Response For a Class

MPC: Message Passing Coupling

Liste des Figures

FIGURE 1 - MODELE SDLC TRADITIONNEL(ASIEDU & GU, 1998)	20
FIGURE 2 - LE MODELE A SOLUTION RAPIDE(HARRISON & COOK, 1990).....	22
FIGURE 3 - LE MODELE ITERATIF D'AMELIORATION(HARRISON & COOK, 1990).....	23
FIGURE 4 - IMPACT DES TYPES D'EVOLUTION ET DE MAINTENANCE DES LOGICIELS.....	27
FIGURE 5 - PROCESSUS DE MAINTENANCE ITERATIVE ISO/CEI 14764 (MENS, 2008)	28
FIGURE 6 - MODELE EN CASCADE DE DEVELOPPEMENT DE LOGICIEL.....	31
FIGURE 7 - MODELES DE CHANGEMENT EN MINI-CYCLES.....	31
FIGURE 8 - LE MODELE PAR ETAPES (V. T. RAJLICH & BENNETT, 2000).....	33
FIGURE 9 - LE PROCESSUS DE REINGENIERIE (SOMMERVILLE, 2011).....	35
FIGURE 10 - OUTIL DE RETRO INGENIERIE IDA PRO.(CONTI, DEAN, SINDA, & SANGSTER, 2008).....	37
FIGURE 11 - PROCESSUS DE MESURE.....	38
FIGURE 12 - LES SEPT PHASES DU PROCESSUS DE MAINTENANCE (CANFORA & CIMITILE, 2001).....	44
FIGURE 13 - CAPTURE D'ECRAN DU VOLET "AFFECTING CHANGES VIEWER" DE CHIANTI.....	53
FIGURE 14 - VUE D'ENSEMBLE DU FRONT-END DE FAULTTRACER.....	54
FIGURE 15 - VUE DES RESULTATS D'IMPACTMINER.....	55
FIGURE 16 - JRIPPLES VIEW (ENVIRONNEMENT ECLIPSE).....	55
FIGURE 17 - VUE D'ENSEMBLE DU FRONT-END DE LATTIX.....	56
FIGURE 18 - VUE D'ENSEMBLE DU FRONT-END DE JARCHITECT.....	57
FIGURE 19 - NOTION D'ESPACE EXPERIMENTAL (JACQUES GOUPY, 1995).....	60
FIGURE 20 – REPRESENTATION DE FACTEUR	60
FIGURE 21 - DEFINITION DES POINTS EXPERIMENTAUX	61
FIGURE 22 - DOMAINE D'ETUDE DEFINI PAR L'EXPERIMENTATEUR.....	61
FIGURE 23 - SCHEMA REPRESENTANT LES FACTEURS ET LES REPONSES	62
FIGURE 24 - PLAN D'EXPERIMENTATION	81
FIGURE 25 - INTERFACE DE L'OUTIL DE CALCUL DE METRIQUES "SACI"	82
FIGURE 26 - EXTRAIT DE LA TABLE DES METRIQUES DE COUPLAGE DE LA VERSION1 DE "JFREECHART"	83
FIGURE 27 - REPRESENTATION GRAPHIQUE DU TEST DE ROBUSTESSE DU MODELE PROPOSE	91
FIGURE 28 - REPRESENTATION TRIDIMENSIONNELLE DE L'INFLUENCE DU MPC ET DU CBO SUR L'EIC.....	91
FIGURE 29 - REPRESENTATION ISOMETRIQUE DE L'INFLUENCE DU MPC ET DU CBO SUR L'EIC.....	92
FIGURE 30 - ENVIRONNEMENT DE L'OUTIL "SACI"	107
FIGURE 31 - VOLET "FICHIER" DE L'OUTIL "SACI"	108
FIGURE 32 - VOLET "STRUCTURE" DE L'OUTIL "SACI"	108
FIGURE 33 - VOLET "EXTRACTION" DE L'OUTIL "SACI"	109
FIGURE 34 - VOLET "CALCUL METRIQUE" DE L'OUTIL "SACI"	109

Liste des Tableaux

TABLEAU 1 - PROTOCOLE EXPERIMENTAL	86
TABLEAU 2 - REPONSES OBSERVEES	86
TABLEAU 3 - COEFFICIENTS β_i DU MODELE D'IMPACT DE CHANGEMENT (FACTEURS CBO, RFC ET MPC)	87
TABLEAU 4 - COEFFICIENTS β_i DU MODELE D'IMPACT DE CHANGEMENT (FACTEURS CBO, MPC).....	88
TABLEAU 5 - OUTILS D'ANALYSE D'IMPACT DE CHANGEMENT (GALBO, 2017)	110

I. Introduction générale

A. Contexte

La maintenance ou l'évolution est une étape incontournable dans le cycle de vie d'un logiciel. Plusieurs travaux ont, depuis de nombreuses années, mis en évidence l'importance de la maintenance et/ou évolution de logiciel et ont dressé des taxinomies, (Kitchenham et al., 1999). Les plus anciens ont permis de considérer trois sortes de maintenance (Swanson, 1976) : *la maintenance corrective* consistant à réaliser des opérations de changement dans le logiciel afin de supprimer des erreurs ou fautes pouvant être originales ou résultant d'effets de bord de changements induits par le processus de maintenance; *la maintenance adaptative* consistant à adapter le logiciel aux changements de son environnement tels que le changement des infrastructures de déploiement; *la maintenance perfective* consiste en des changements visant à améliorer le logiciel en y introduisant notamment de nouvelles fonctionnalités ou en améliorant ses performances, etc. A ces trois sortes de maintenance s'est ajoutée la maintenance dite *préventive* (Chapin, Hale, Khan, Ramil, & Tan, 2001) dont le but est d'améliorer l'aptitude du logiciel à évoluer. En d'autres termes cette dernière sorte de maintenance permet l'amélioration du critère de maintenabilité ou d'évolutivité du logiciel sans que cela se traduise par des modifications de ses fonctionnalités ou de ses performances.

Dans nos travaux, nous ne faisons pas de différence entre maintenance et évolution bien que ces deux concepts ont été sujets à de nombreuses comparaisons. La maintenance est souvent vue comme un processus d'ingénierie dans lequel les travaux de recherche académiques ont consisté à en délimiter et caractériser les étapes, les activités à mener, les acteurs et les ressources à mettre en œuvre,...etc. En d'autres termes le concept de maintenance est le résultat d'une sorte de morphisme entre le monde industriel et l'activité de développement de logiciels. Le terme évolution de logiciel est le résultat de travaux (Belady & Lehman, 1976; Meir M Lehman, 1980; Meir M Lehman, Ramil, Wernick, Perry, & Turski, 1997)) ayant pour but de comprendre le phénomène de changement affectant les artefacts logiciels. Les lois d'évolution de Lehmann (Meir M Lehman et al., 1997) par exemple tentent

d'expliquer la manière dont les logiciels évoluent jusqu'à devenir obsolètes. Ces lois sont le résultat d'études empiriques menées pendant plusieurs années sur des logiciels de taille réelle. Dans le même ordre d'idée, les auteurs dans (V. T. Rajlich & Bennett, 2000) ont dressé dans un cycle de vie du logiciel conçu non comme un moyen de développer le logiciel mais comme une explication de la manière dont les logiciels sont produits, évoluent, sont maintenus pour finir par être retirés des systèmes d'information.

En ce qui nous concerne, nous utiliserons de façons indifférentes les termes maintenance ou évolution. Ce qui nous intéresse particulièrement est la notion de changement et au-delà la maîtrise du processus de changement et plus particulièrement l'analyse de l'impact de ces changements.

Il est clair que l'une des préoccupations majeures des acteurs du développement est de maîtriser le processus du changement. En effet, un changement non maîtrisé peut induire rapidement à un glissement en termes de coût et de délais pouvant conduire à des échecs de projets de maintenance/évolution de logiciels. Cela est d'autant plus critique que plusieurs études ont, depuis longtemps, montré l'influence de la maintenance sur le coût global d'une application informatique (Ferens, 1999). Par ailleurs, selon la norme ISO 250010 Model (ISO/ IEC CD 25010 2007) (José, Mauricio, & Rodríguez, 2014), la maintenabilité ou évolutivité constitue une des principales composantes de la qualité d'un logiciel.

B. Problématique

Dans cette thèse, nous nous intéressons à l'estimation de l'impact du changement du logiciel mais d'un point de vue quantitative. En d'autres termes nous essayons de produire une modélisation à base d'équations mathématiques nous permettant d'estimer, en quelque sorte, le coût du changement d'un logiciel donné. En effet, une modification dans un objectif d'évolution d'un logiciel se traduit par des changements opérés dans le code source dans le but d'améliorer ou corriger son fonctionnement. Ces changements englobent toute modification apportée à l'un des éléments du logiciel (variable, méthode ou classe). Par exemple, la suppression d'une

variable, la modification de la portée d'une méthode, ou le déplacement du lien entre une classe et son parent, etc. Un changement peut avoir des effets considérables et inattendus sur le reste du système. Le danger encouru lors d'une modification se manifeste par des effets de bord indésirables. Pour éviter ces situations, il serait intéressant d'estimer l'impact de changement avant même d'opérer le changement. La question qui se pose à ce stade est le choix des connaissances à utiliser pour réaliser cette estimation. Selon Chidamber et Kemerer (Chidamber & Kemerer, 1994), un code est caractérisé par un certain nombre de mesures qui reflètent sa structure. Ceci constitue à notre avis une base intéressante permettant d'estimer l'effet de changement. Pour cela, nous proposons dans nos travaux d'étudier la faisabilité de l'estimation de l'impact de changement à travers des métriques. Autrement dit, nous proposons de modéliser le lien qui existe entre l'impact de changement et les métriques dites de couplage.

C. Travaux connexes

Le changement est une caractéristique cruciale et inévitable du logiciel tout au long de son cycle de vie. Ceci est dû aux efforts de maintenance et/ou d'évolution nécessaires concernant les différents artefacts logiciels conduisant inévitablement à des opérations de changement les affectant. Depuis plus de quatre décennies, de nombreux travaux ont été consacrés à ce domaine de l'ingénierie logicielle, donnant lieu à de nombreuses définitions et taxonomies (Meir M. Lehman & Ramil, 2002). Les travaux dans ce domaine peuvent être globalement classés selon différentes approches : ceux qui concernent les artefacts touchés par le changement; d'autres qui s'intéressent aux métriques mesurant les attributs de tels artefacts ; et certaines études empiriques qui traitent principalement les données historiques des logiciels. Ci-après, nous considérons quatre approches basées respectivement sur la représentation graphique du logiciel, l'utilisation de métriques logicielles, l'utilisation de modèles, et l'utilisation de données historiques.

Les approches concernant la représentation logicielle à base de graphes consistent à extraire un graphe basé sur les artefacts logiciels afin d'illustrer la propagation de

l'impact du changement. Dans (Abdi & Dinedane, 2015), un graphe de dépendance du système a été extrait. L'idée sous-jacente est de mesurer l'accessibilité des différents nœuds du graphe. Il s'agit du calcul des fermetures transitives des relations représentées par les arêtes du graphe. Considérons que cette fermeture détermine la force de la dépendance entre les nœuds en tant que composants liés ou fortement lié. Par conséquent, cette fermeture peut informer sur la propagation de l'impact du changement. Ce dernier est souvent calculé à l'aide d'un algorithme largement appliqué, appelé algorithme de Warshall (Warshall S., 1962). Dans (Bouneffa & Ahmad, 2013), le logiciel est représenté par un modèle basé sur un système de réécriture de graphes où les composants logiciels sont liés par des relations significatives. Dans ce contexte (Musco, Monperrus, & Preux, 2014) proposent un modèle générique de graphes de dépendance logicielle qui synthétise les graphes où la distribution des degrés est proche de celle empirique observée dans les systèmes logiciels réels. Ce modèle fournit de nouvelles informations sur les règles fondamentales potentielles de l'évolution du logiciel. Dans (Ahmad, Basson, & Bouneffa, 2017), les auteurs proposent une approche de modélisation globale pour interroger les interdépendances structurelles et qualitatives d'un logiciel lors de l'intégration de plusieurs modifications. Dans (Bhattacharya, Iliofotou, Neamtiu, & Faloutsos, 2012), les auteurs construisent des graphes qui capturent les structures logicielles, puis exploitent les avancées récentes en matière d'analyse de la topologie des graphes pour mieux comprendre l'évolution du logiciel. Dans (Abdeen, Bali, Sahraoui, & Dufour, 2015), les indicateurs d'impact de changement sont tirés des graphiques d'impact de changement antérieurs, extraits de l'historique des versions, ainsi que de leurs associations avec différents facteurs déterminants de la propagation du changement.

Les approches basées sur des métriques traitent généralement des concepts orientés objet et concernent principalement des métriques connues comme une estimation précise de l'effort de maintenance. Dans (Saraiva, de França, Soares, Filho, & de Souza, 2015), les valeurs des métriques d'analyse montrent une forte corrélation entre les métriques de couplage et la mesure de l'effort de maintenance. Dans (Almugrin,

Albattah, & Melton, 2016), les auteurs ont établi qu'un nombre moins élevé de révisions de packages (REVISIONS) et un nombre plus petit de lignes de codes révisées (RLOC) au cours de l'historique de maintenance des packages indiquent un effort moindre de maintenance des packages.

Les approches basées sur un modèle visent principalement à prédire l'impact du changement au niveau conceptuel. Dans (M. Ajmal Chaumon, Hind Kabaili, Rudolf K. Keller, & François Lustman, 2002), les divers liens entre les classes d'un système sont pris en compte pour calculer ou estimer l'impact du changement. L'impact est donc exprimé par une combinaison de ces liens. De la même manière (Müller & Rumpe, 2014) proposent une approche de l'analyse d'impact de changement dans laquelle des règles d'impact explicites capturent les conséquences de la modification des diagrammes de classes UML sur les autres artefacts. Les diagrammes de classes UML analysés décrivent généralement deux versions du système en cours de développement. Les différences sont automatiquement identifiées à l'aide de la différenciation des modèles. La formulation explicite des conséquences des modifications des règles d'impact permet de créer une liste de contrôle avec des indications précises concernant les étapes de développement nécessaires à la gestion de l'évolution.

Les approches empiriques visent à utiliser les avantages de l'application de méthodes et techniques d'autres domaines au génie logiciel et illustrent comment, malgré les difficultés, l'évolution du logiciel peut être étudiée empiriquement. Dans (Hegedüs, Kádár, Ferenc, & Gyimóthy, 2018), les auteurs ont divisé les éléments de code source en un groupe contenant les éléments refactorisés et un groupe d'éléments non refactorisés. Les auteurs ont analysé les caractéristiques des éléments dans ces groupes en utilisant une analyse de corrélation, le test U de Mann – Whitney et des mesures de la taille de l'effet. Dans (Anwer, Adbellatif, Alshayeb, & Anjum, 2017), les auteurs étudient l'impact du couplage afférent (Ca), du couplage efférent (Ce) et du couplage entre objets (CBO) sur la prévision des fautes par corrélation bivariée. Le résultat de cette étude est un modèle de prédiction utilisant ces métriques pour prédire les erreurs.

Notre travail se situe dans la catégorie qui s'intéresse à l'utilisation de l'estimation de l'impact de changement comme indicateur principal de l'estimation de l'effort de maintenance en exploitant les mesures ou métriques du logiciel. Nous nous intéressons particulièrement aux systèmes à objets. Nous appliquons une technique dite de plan d'expériences, généralement utilisée en sciences expérimentales, dans le but de quantifier l'effort de maintenance qui serait une mesure de l'impact du changement, sous la forme d'une équation mathématique dont les paramètres sont les valeurs de ces métriques.

D. Organisation de la thèse

Cette thèse est organisée de la manière suivante :

Le chapitre 2 traite de l'évolution de logiciel, à savoir les lois de l'évolution, la définition de la maintenance en exposant ses différents types, ses activités, ses problèmes et ses techniques. Ensuite, il définit ce qu'est une métrique dans le domaine de génie logiciel, puis montre les différents types de métriques, et dresse des exemples de métriques orientées objets les plus connues dans la littérature.

Le chapitre 3 présente une vue d'ensemble sur l'impact de changement. Il commence par présenter les différents travaux faits autour de l'analyse d'impact du changement, les différentes approches adoptées ainsi que les modèles utilisés.

La présentation de la technique de plan d'expériences "Design of Experiments" (DoE), qui est utilisée pour la construction du modèle de représentation d'un logiciel est effectuée dans le chapitre 4.

Le Chapitre 5 aborde la problématique de l'estimation de l'impact de changement en utilisant la technique de plan d'expériences DoE. On y présente les étapes qui permettent de modéliser l'interaction d'un ensemble de métriques, en occurrence *CBO*, *RFC* et *MPC*, afin d'évaluer leur contribution globale à l'analyse de l'impact du changement dans un système logiciel donné. Les détails de l'expérimentation sont présentés et les résultats obtenus sont discutés.

Enfin, le Chapitre 6 conclut cette thèse, rappelle les principaux résultats obtenus et expose les perspectives majeures pour les travaux futurs.

II. Evolution de logiciel

1. Introduction

L'évolution ou bien la maintenance des logiciels constituent la phase la plus longue du cycle de vie d'un logiciel (Swanson, 1976). Il s'agit de la période entre le déploiement de la première version et le retrait du produit. Pour certains systèmes logiciels critiques, cela peut s'étendre sur plusieurs décennies. Les logiciels libres sont également un bon exemple de la longévité de la phase de maintenance (Tu, 2000). Évidemment, sur une période aussi longue, un produit doit subir de nombreuses modifications (des défauts sont signalés, de nouvelles fonctionnalités sont demandées, de nouvelles technologies doivent être prises en charge, etc.) qui engendrent des changements. De par sa définition, un changement est une caractéristique essentielle du développement de logiciels, car ces derniers doivent s'adapter en permanence à leur environnement qui est en continuel changement. Lehman a fait le constat que les systèmes logiciels doivent pouvoir évoluer ou risquer une fin prématurée (Meir M Lehman et al., 1997).

2. Évolution versus maintenance

Le concept d'évolution des logiciels a été introduit par Mark Halpern en 1965 pour décrire les caractéristiques de croissance des logiciels (Halpern, 1965). Belady et Lehman se sont intéressées à ce concept et ont publié un ensemble de principes déterminant l'évolution des systèmes logiciels (Belady & Lehman, 1976). R. G. Canning a comparé la maintenance logicielle à un «iceberg» pour souligner le fait que les développeurs de logiciels et le personnel de maintenance font face à un grand nombre de problèmes (Canning, 1972). Swanson en 1976 a introduit le terme "maintenance" en classant les activités de maintenance en trois catégories de base : *corrective*, *adaptive* et *perfective* (Swanson, 1976). IBM dans le début des années 1970 désigna par "ingénieurs de maintenance" ou "mainteneurs" celui qui fait des modifications intentionnelles dans un code qu'il n'avait pas développé. La principale raison d'utiliser du personnel non chargé du développement pour la maintenance a été

dans le but de libérer les ingénieurs de développement de la tâche du support technique.

Bennett et Rajlich (Bennett & Rajlich, 2000) ont étudié le terme "évolution de logiciel" et ont découvert qu'il n'y a pas un consensus sur la définition de ce terme . Les praticiens ne font pas de distinction entre les expressions "évolution de logiciel" et "maintenance de logiciel" bien qu'il existe des différences sémantiques entre les deux termes :

- Le concept de maintenance logicielle signifie : éviter que le logiciel tombe en panne en ajoutant des fonctionnalités souhaitées par la correction de bugs.
- Le concept d'évolution de logiciel signifie un changement continu d'un état inférieur, simple ou mauvais vers un état supérieur ou meilleur (Arthur, 1988).

Bennett et Xu (Bennett & Xu, 2003) ont fait la distinction suivante entre les deux termes :

- Toutes les activités de soutien réalisées après la livraison des logiciels appartiennent à la catégorie de la maintenance.
- Toutes les activités menées pour modifier les exigences relèvent de la catégorie de l'évolution.

En général, la maintenance et l'évolution sont généralement différenciées comme suit :

- La maintenance des systèmes logiciels signifie principalement la correction des bugs des logiciels tout en préservant leurs fonctionnalités. Les tâches de maintenance sont planifiées mais aussi des actions de correction imprévues sont également nécessaires. D'une manière générale, la maintenance consiste à maintenir un logiciel installé, en marche, sans modifier sa conception (Parnas, 1994).
- L'évolution des logiciels implique la création de nouvelles conceptions basées sur celles déjà existantes afin de prendre en charge de nouvelles fonctionnalités, par exemple le logiciel s'exécutera mieux, permettre le fonctionnement du logiciel sur un autre système d'exploitation,...etc. Au fil du temps, les mainteneurs développent plus de connaissances sur le système. Cela permet au logiciel d'évoluer de plusieurs façons. Mehdi Jazayeri a observé que : « ce qui

évolue n'est pas le logiciel, mais notre connaissance d'un type particulier de logiciel » (Jazayeri, 2005). Ce qui signifie qu'avec le temps, non seulement de nouvelles fonctionnalités émergent, mais aussi les utilisateurs deviennent plus avertis.

Enfin, il faut préciser que dans la maintenance matérielle, un système ou un composant est rétabli dans son état d'origine. Tandis que, dans la maintenance logicielle, un logiciel est transformé d'un état initial erroné vers un état cible corrigé (Stachour & Collier-Brown, 2009). La maintenance du logiciel comprend toutes les activités associées au processus de changement de logiciel afin de :

- corriger les bugs; et / ou
- améliorer la conception du système afin que les modifications futures du logiciel soient moins coûteuses.

3. Taxonomie de la maintenance et de l'évolution des Logiciels.

Un cycle de développement logiciel traditionnel comprend deux phases distinctes, à savoir : **Développement et Maintenance**

La partie maintenance correspond aux deux tiers de la durée de vie du logiciel (Barry, 1981)

Les pourcentages indiquant les coûts relatifs à chaque étape sont illustrés dans la Figure 1:

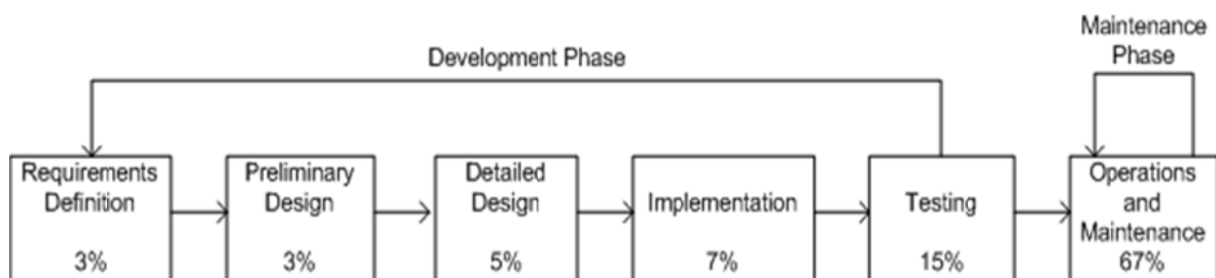


Figure 1 - Modèle SDLC traditionnel (Asiedu & Gu, 1998)

a) Caractéristiques de la maintenance de logiciel

- **Contraintes d'un système existant** : La maintenance est effectuée sur un système opérationnel. Par conséquent, toutes les modifications

doivent être compatibles avec les contraintes de l'architecture, de la conception et du code existants.

- **Délai plus court** : Une activité de maintenance peut durer de quelques heures à quelques mois, alors que le développement d'un logiciel peut durer une ou plusieurs années.
- **Données d'essai disponibles** : Dans le développement de logiciels, les scénarios de test sont conçus à partir de zéro, tandis que la maintenance logicielle peut sélectionner un sous-ensemble de ces scénarios de test et les exécuter comme tests de régression.

La maintenance logicielle devrait avoir son propre modèle de cycle de vie de la maintenance logicielle (SMLC) car elle implique de nombreuses activités uniques.

b) Réutilisation des modèles orientés

On obtient une nouvelle version d'un ancien système en modifiant un ou plusieurs composants de l'ancien système et en ajoutant éventuellement de nouveaux composants.

Sur la base de ce concept, trois modèles de processus de maintenance ont été proposés par Basili (Basili & Rombach, 1990):

- **Modèle à fixation rapide** : Dans ce modèle, les modifications nécessaires sont rapidement apportées au code, puis à la documentation qui l'accompagne.
- **Modèle itératif d'amélioration** : Dans ce modèle, les premières modifications sont apportées aux documents de niveau supérieur. Finalement, les changements se propagent jusqu'au niveau du code.
- **Modèle de réutilisation complète** : Dans ce modèle, un nouveau système est construit à partir de composants de l'ancien système et d'autres disponibles dans le référentiel.

Dans le modèle à fixation rapide, comme l'illustre la Figure 2 le code source est modifié pour résoudre le problème ; les modifications nécessaires sont apportées aux

documents pertinents ; et le nouveau code est recompilé pour produire une nouvelle version.

Souvent, les changements apportés au code source sont effectués sans enquête préalable, comme l'analyse de l'impact des changements, les effets de vague et les tests de régression.

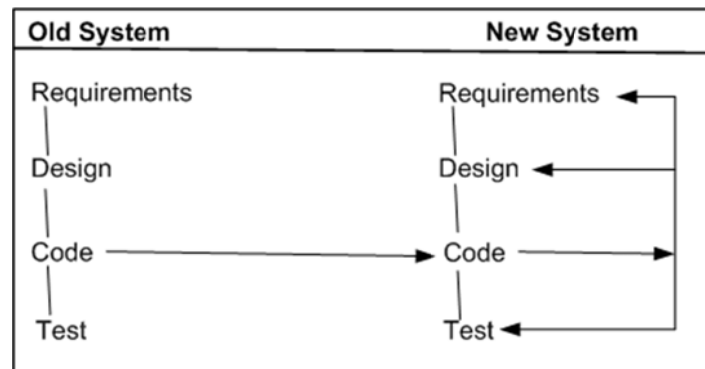


Figure 2 - Le modèle à solution rapide(Harrison & Cook, 1990)

c) Itératif Vs. Incrémentale

Les termes itération et incrément sont largement utilisés lorsqu'on parle de développement itératif et incrémental. Cependant, ils ne sont pas synonymes dans le domaine du génie logiciel.

L'itération implique qu'un processus est fondamentalement cyclique, ce qui signifie que les activités du processus sont exécutées de manière répétée et structurée.

Le développement itératif est basé sur des stratégies d'ordonnancement dans lesquelles du temps est réservé à l'amélioration et à la révision de parties du système en développement.

Le développement progressif est basé sur des stratégies de mise en scène et d'ordonnancement dans lesquelles des parties du système sont développées à des moments et/ou à des rythmes différents, et intégrées à mesure qu'elles sont achevées. Le modèle itératif d'amélioration, expliqué à la Figure 3, montre comment les changements passent des documents de haut niveau aux documents de bas niveau.

Le modèle fonctionne comme suit :

Il commence par les artefacts du système existant, à savoir les exigences, la conception, le code, les tests et les documents d'analyse.

Il révisé les documents de plus haut niveau affectés par les changements et diffuse les changements vers le bas à travers les documents de niveau inférieur.

Le modèle permet aux responsables de la maintenance de reconcevoir le système, sur la base de l'analyse du système existant.

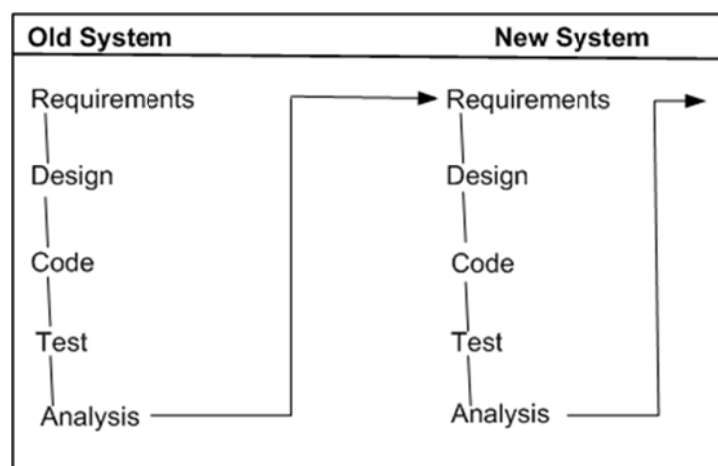


Figure 3 - Le modèle itératif d'amélioration(Harrison & Cook, 1990)

La principale hypothèse de ce modèle est la disponibilité d'un dépôt d'artefacts décrivant les versions antérieures des systèmes.

Dans le modèle de réutilisation intégrale, la réutilisation est explicite et les activités suivantes sont réalisées :

- identifier les composants de l'ancien système qui sont susceptibles d'être réutilisés
- comprendre les composants du système identifiés.
- modifier les composants de l'ancien système pour prendre en charge les nouvelles exigences.
- intégrer les composants modifiés pour former le système nouvellement développé



d) Evolution de logiciel

Bien que l'expression "évolution de logiciel" ait déjà été utilisée par d'autres chercheurs, les travaux fondamentaux dans le domaine de l'évolution des logiciels ont été réalisés par Lehman et ses collaborateurs. Sur la base d'études empiriques (Belady & Lehman, 1976; Meir M Lehman, Perry, & Ramil, 1998), leurs observations sont présentées comme des lois de l'évolution de logiciels.

Lehman (Meir M Lehman, 1980) présente ces lois en distinguant trois catégories de logiciels :

- logiciel de type S qui est écrit en fonction d'une spécification exacte de ce que ce logiciel peut faire ;
- logiciel de type P qui est écrit pour implémenter certaines procédures qui déterminent complètement ce que le logiciel peut faire.
- logiciel de type E qui a un comportement lié à l'environnement dans lequel il s'exécute. De ce fait, il doit s'adapter aux différentes exigences de cet environnement.

En d'autres termes Lehman fait une distinction entre les logiciels de bases comme les systèmes d'exploitation et les logiciels d'applications dépendant du contexte ou environnement.

Les lois ne concernent que la catégorie de logiciels de type E "logiciels d'application". Les "lois" elles-mêmes sont "passées" de trois en 1974 à huit en 1997 (Meir M Lehman et al., 1997; Madhavji, Lehman, Perry, & Ramil, 2006; Yu & Mishra, 2013). Ces lois sont :

1. **Changement continu** : À moins qu'un logiciel ne soit modifié en permanence pour satisfaire les besoins des utilisateurs, il devient de moins en moins utile ;
2. **Complexité grandissante** : À moins que du travail supplémentaire ne soit fait pour réduire explicitement la complexité du logiciel, le système deviendra de plus en plus complexe en raison des modifications liées à la maintenance.
3. **Autorégulation** : L'évolution des grands programmes est un processus autorégulateur. Les attributs comme la taille, le temps entre versions et le

nombre d'erreurs signalées sont approximativement invariants pour chaque version du programme.

4. **Conservation de la stabilité organisationnelle** : L'effort dédié à faire évoluer un logiciel demeure constant au cours du temps.
5. **Conservation de la familiarité** : À mesure qu'un système évolue, tous les types de personnel, à savoir: les développeurs et les utilisateurs, par exemple, doivent acquérir le niveau de compréhension souhaité du contenu et du comportement du logiciel pour parvenir à une évolution satisfaisante.
6. **Croissance continue** : Au fil du temps, le contenu fonctionnel d'un système est continuellement augmenté pour répondre aux besoins des utilisateurs.
7. **Dégradation de la qualité** : À moins que la conception du système ne soit minutieusement ajustée et adaptée aux nouveaux environnements opérationnels, les qualités du système seront perçues comme déclinant au cours de son cycle de vie.
8. **Principe du feed-back** : tout changement dans l'environnement extérieur envoie un signal au sein du logiciel, dont les évolutions constituent la réponse renvoyée à l'extérieur. Cette réponse peut générer elle-même des changements, engendrant ainsi une boucle de rétroaction.

Partant du constat sur les différences entre l'évolution du système d'exploitation Unix développé par les laboratoires Bell et les systèmes étudiés par Lehman et ses collaborateurs, réalisé par Pirzada (Pirzada, 1988) et complété par (Tu, 2000) et qui montre que des différences dans le développement de logiciels FOSS " Free Open Software Source" et industriels pourraient conduire à des différences dans le modèle d'évolution. Robles et al. (Robles, Amor, Gonzalez-Barahona, & Herraiz, 2005) précisent que les lois de Lehman 3, 4 et 5 ne s'appliquent pas aux logiciels libres FOSS tels que Linux du fait de la particularité de leurs développements qui suit un paradigme d'évolution différent à celui du processus de développement de logiciel étudié par Lehman. Ce processus porte principalement sur les propriétés exclusives développées par une équipe de développeurs au sein d'une entreprise.

e) Maintenance de logiciel

Après chaque livraison de logiciel des dysfonctionnements apparaissent au fur et à mesure de l'utilisation de ce dernier car les processus d'élimination des erreurs et de contrôle de la qualité ne sont pas parfaits (R. J. Martin & Osborne, 1985). Par conséquent, la maintenance est nécessaire pour éliminer les bugs. Swanson (Swanson, 1976) définissait initialement trois catégories de l'activité de maintenance logicielle, à savoir corrective, adaptative et perfective. Ces définitions ont ensuite été intégrées au *"standard of software engineering–software life cycle processes–Maintenance"* (Standard, 2006). Une quatrième catégorie appelée maintenance préventive est venue pour compléter ce classement. On note ici qu'il existe certains développeurs et chercheurs qui considèrent la maintenance préventive comme un sous-ensemble de la maintenance perfective.

La classification des activités de maintenance effectuée par Swanson est fondée sur l'intention, car les activités de maintenance reflètent l'intention du développeur d'effectuer des opérations de maintenance spécifiques sur le logiciel. On note que dans la classification basée sur l'intention de l'activité de la maintenance, l'intention dépend des motivations du changement.

Une autre technique de classification des modifications apportées aux logiciels consiste à classer simplement les modifications en termes d'activités réalisées (Kitchenham et al., 1999) :

Corrections : s'il y a des écarts entre les valeurs attendues et les valeurs réelles suite à l'exécution d'un logiciel, alors certaines actions sont réalisées pour éliminer ou bien réduire ces écarts.

Améliorations : un nombre d'activités sont effectuées pour mettre en œuvre une modification du système, modifiant ainsi le comportement ou la mise en œuvre du système. Cette catégorie d'activités est ensuite affinée en trois sous-catégories d'améliorations :

- ✓ celles qui modifient les besoins existants,
- ✓ celles qui créent de nouveaux besoins,
- ✓ celles qui modifient la mise en œuvre sans changer les besoins.

Dans (Chapin et al., 2001) les auteurs ont étendu la typologie de Swanson à une classification prouvée de 12 différents types de maintenance logicielle: *formation, consultative, évaluative, réformatrice, évolutive, préventive, performante, adaptative, réductive, corrective et enrichissante*. Cette classification de la maintenance de logiciel a été réalisée conformément aux trois objectifs suivants :

1. Il est plus enrichissant de classer les tâches de maintenance en fonction de preuves objectives pouvant être vérifiées à l'aide d'observations et/ou de comparaisons du logiciel avant et après modifications. Cela ne nécessite pas d'accéder aux connaissances du personnel qui a initialement développé le système.
2. La granularité de la classification proposée peut être adaptée pour refléter avec précision la combinaison réelle d'activités observée dans la pratique de la maintenance et de l'évolution des logiciels.
3. Les groupes de classification sont indépendants de la plate-forme matérielle, du choix du système d'exploitation, de la méthodologie de conception, du langage d'implémentation, des pratiques de l'organisation et de la disponibilité du personnel qui a effectué le développement d'origine.

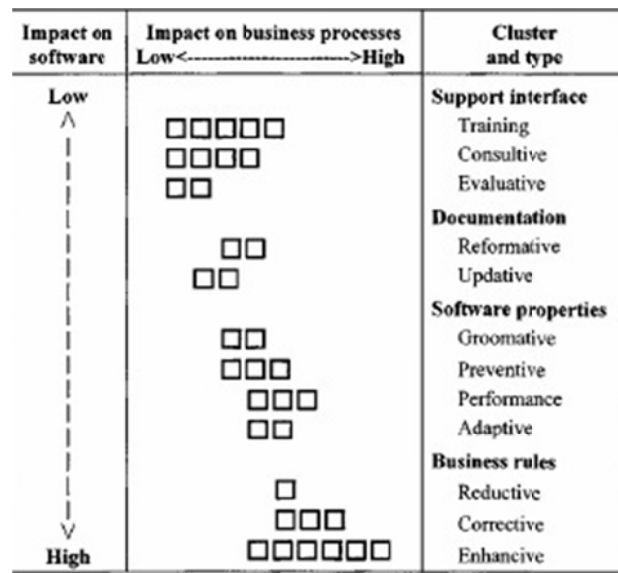


Figure 4 - Impact des types d'évolution et de maintenance des logiciels

f) Normes de maintenance de logiciel

IEEE " Institute of Electrical and Electronics Engineers" et ISO "International Organization for Standardization " ont tous deux abordé les processus de maintenance.

IEEE/EIA 1219 organise le processus de maintenance en sept phases :

L'identification des problèmes, l'analyse, la conception, la mise en œuvre, l'essai du système, l'essai d'acceptation et la livraison.

La norme ISO/IEC 14764 décrit la maintenance comme un processus itératif de gestion et d'exécution des activités de maintenance logicielle.

Les activités qui composent le processus de maintenance sont :

Mise en œuvre des processus, analyse des problèmes et des modifications, mise en œuvre des modifications, examen/acceptation de la maintenance, migration et retrait.

Chacune de ces activités de maintenance est composée de tâches. Chaque tâche décrit une action spécifique avec des entrées et des sorties.

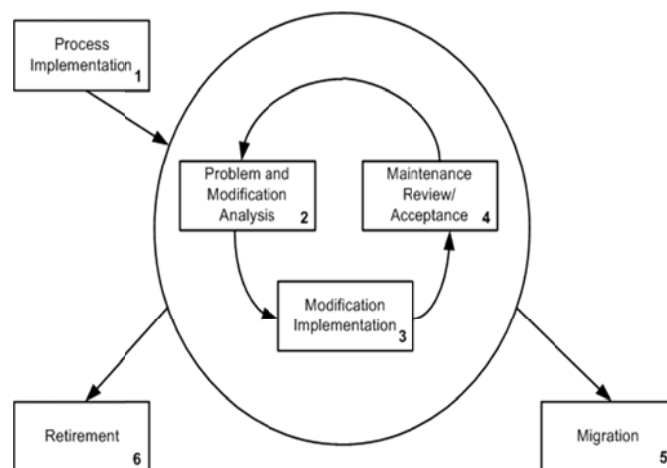


Figure 5 - Processus de maintenance itérative ISO/CEI 14764 (Mens, 2008)

Un processus bien défini pour la maintenance logicielle peut être observé et mesuré, et donc amélioré. En outre, l'adoption de processus permet plus rapidement d'acquérir une expérience personnelle de la diffusion des pratiques de travail efficaces. La maintenance logicielle centrée sur les processus est davantage une activité d'ingénierie, avec des contraintes prévisibles de temps et d'effort. Par conséquent, les normes de maintenance logicielle ont été formulées par les normes ISO et IEEE. Le

document de la norme de maintenance ISO s'appelle ISO / IEC 14764 (Standard, 2006) et fait partie du document standard ISO / IEC 12207 (Standardization, 2008) pour les processus de cycle de vie. Le document de maintenance standard IEEE s'appelle IEEE / EIA 1219 (Mamone, 1994). Les deux normes décrivent les processus de gestion et d'exécution des activités de maintenance. La norme IEEE / EIA 1219 organise le processus de maintenance en sept phases : *identification du problème, analyse, conception, mise en œuvre, test du système, test de réception et livraison*. En résumé, la norme identifie les différentes phases et la séquence de leurs exécutions. Ensuite, pour chaque phase, la norme identifie les produits livrables d'entrée et de sortie, les processus d'appui et les activités connexes, ainsi qu'un ensemble de paramètres d'évaluation. Les deux normes, à savoir ISO / IEC 14764 et IEEE / EIA 1219, utilisent la même terminologie pour décrire la maintenance logicielle, avec une légère différence dans leur description. Un processus itératif a été décrit dans la norme ISO / CEI 14764 pour gérer et exécuter des activités de maintenance. Les activités composant le processus de maintenance sont :

- mise en œuvre du processus,
- analyse des problèmes et des modifications,
- modification et mise en œuvre,
- examen de maintenance/acceptation,
- migration,
- retrait.

Chacune des activités susmentionnées est composée de tâches décrites avec des entrées, des sorties et des actions spécifiques.

g) Modèles et processus d'évolution des logiciels

Il existe beaucoup de confusion entre les termes "maintenance de logiciel" et "évolution de logiciel". Cela est dû au fait de considérer la maintenance comme activité inclut dans le développement logiciel. Si on considère le modèle de classification en cascade (voir Figure 6) (Balaji & Murugaiyan, 2012), la phase finale du modèle en cascade est nommée maintenance, ce qui implique que la maintenance

logicielle fait partie du développement de logiciel. Il convient de citer Schneidewind (Schneidewind, 1987) qui précise que : "La conception traditionnelle du cycle de vie des logiciels a porté préjudice à la maintenance en la décrivant uniquement comme une étape unique à la fin du cycle". Par conséquent, la maintenance logicielle doit avoir son propre modèle de cycle de vie de maintenance logicielle "Software Maintenance Life Cycle" (SMLC) (Chapin, 1988). Un certain nombre de modèles SMLC avec quelques variantes sont disponibles dans la littérature (Kung & Hsu, 1998). Trois caractéristiques communes des modèles orientés code seulement SMLC sont :

- a) comprendre le code,
- b) modifier le code,
- c) revalider le code.

D'autres modèles considèrent le développement de logiciels comme des processus itératifs et basés sur l'idée de mini-cycle de changement (Bennett & Rajlich, 2000; R. C. Martin, 2005; Yau, Collofello, & MacGregor, 1993), comme expliqué ci-après :

a) Modèles itératifs.

Les modèles itératifs partagent les idées selon lesquelles un ensemble complet de besoins pour un système ne peut pas être complètement compris ou que les développeurs ne savent pas comment construire le système complet. Par conséquent, les systèmes sont construits dans des versions, chacune étant un raffinement des besoins de la version précédente.

Une construction est affinée en prenant en compte les commentaires des utilisateurs (Gilb & Finzi, 1988). On peut noter que les activités de maintenance et d'évolution n'existent pas en tant que phases distinctes. Au contraire, ils sont étroitement liés.

b) Modèles de changement en mini-cycles.

D'abord proposé par Yau et al. (Yau et al., 1993) à la fin des années 1970, ces modèles ont été récemment revus par Bennet et al. (Bennett & Xu, 2003) et Mens (Mens, 2008) parmi d'autres. Ces modèles comportent cinq phases

principales: demande de changement, analyse et planification du changement, mise en œuvre du changement, vérification et validation, et mise à jour de la documentation (voir Figure 7).

Dans ce modèle de processus, plusieurs activités importantes ont été identifiées, telles que la compréhension du programme, l'analyse d'impact, le refactoring et la propagation du changement.

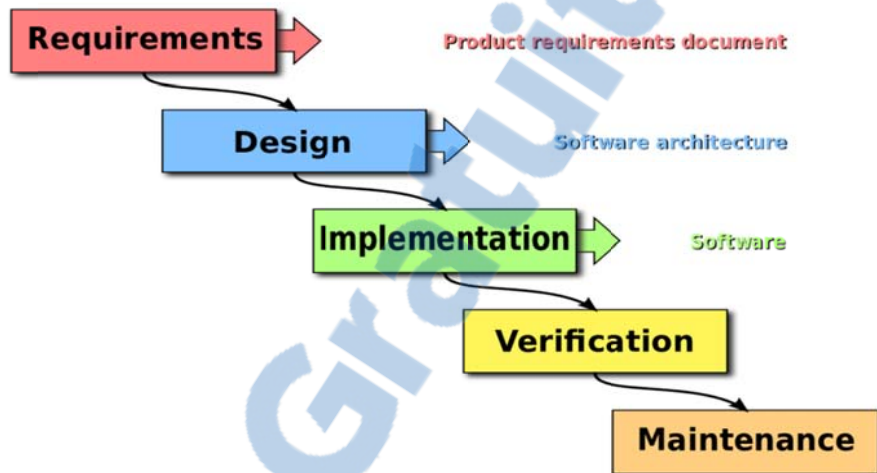


Figure 6 - Modèle en cascade de développement de logiciel

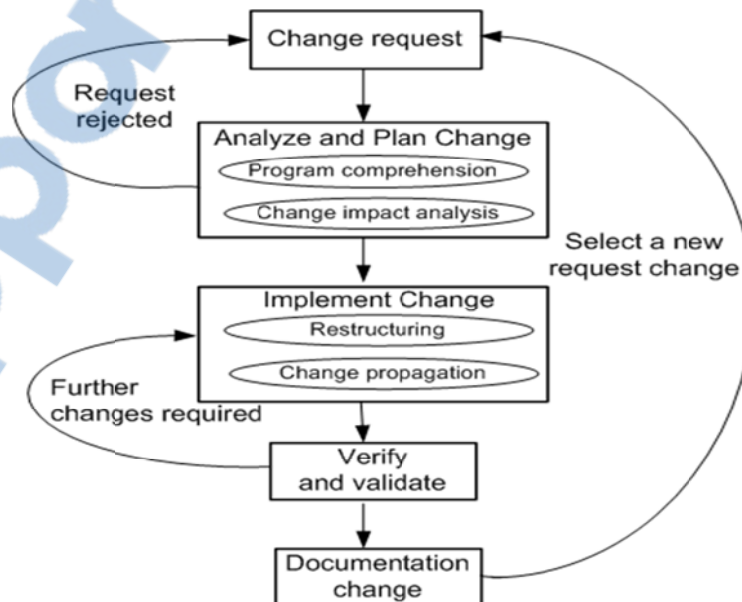


Figure 7 - Modèles de Changement en mini-cycles

Rajlich et Bennett ont proposé un autre modèle d'évolution de logiciel, dénommé *modèle par étapes de maintenance et d'évolution* (V. T. Rajlich & Bennett, 2000). Le modèle est de nature descriptive et son objectif principal est d'améliorer la compréhension de l'évolution de logiciels à longue durée. Le modèle considère quatre étapes séquentielles distinctes tout au long de la durée de vie d'un logiciel, décrites comme suit :

1. Développement initial : Lorsque la version initiale du système est produite, une connaissance détaillée du système est récente. Avant la livraison du système, il subit de nombreuses modifications. Finalement, une architecture de système émerge et se stabilise rapidement.

2. Évolution : Après la stabilité initiale, il est facile d'apporter des modifications simples au système. Les changements importants impliquent des coûts et des risques plus élevés. Dans la période qui a immédiatement suivi la livraison initiale, la connaissance du système est encore presque fraîche dans l'esprit des développeurs. Il est possible que l'équipe de développement n'existe pas dans son ensemble, car de nombreux développeurs originaux ont assumé de nouvelles responsabilités au sein de l'organisation et certains ont peut-être quitté le poste.

3. Entretien : Lorsque les connaissances sur le système ont considérablement diminué, les développeurs se concentrent principalement sur les tâches de maintenance, telles que la correction des bogues, alors que les modifications architecturales sont rarement effectuées. Les développeurs ne considèrent pas le système comme un atout essentiel. À ce stade, les effets des changements sont très difficiles à prévoir. De plus, les coûts et les risques liés aux modifications sont très importants.

4. Phase out : Lorsque la maintenance même minimale d'un système n'est pas une option, le système entre dans sa toute dernière étape. L'organisation décide de remplacer le système pour les raisons suivantes : (i) il est trop coûteux de maintenir le système; ou (ii) une nouvelle solution est disponible. Par conséquent, l'organisation développe une stratégie de sortie pour passer du système actuel à un nouveau système. Le passage d'un système existant difficile à entretenir à une solution moderne pose des

problèmes particuliers en ce qui concerne l'encapsulation et la migration des données. Une fois que le nouveau système fonctionne de manière satisfaisante, parfois en parallèle avec l'ancien système, celui-ci est finalement complètement arrêté.

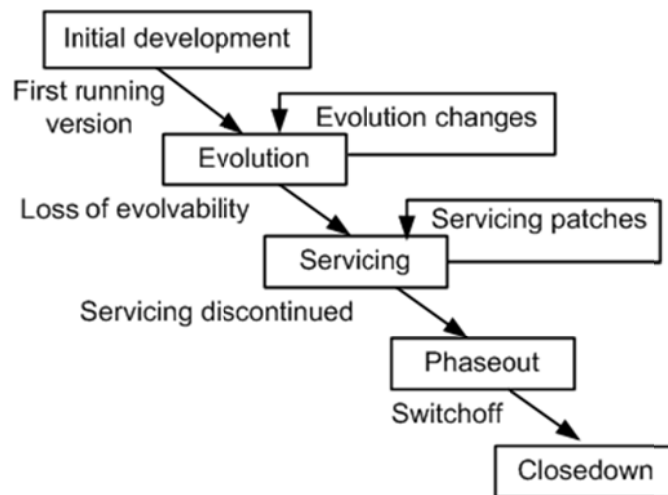


Figure 8 - Le modèle par étapes (V. T. Rajlich & Bennett, 2000)

4. Gestion de l'évolution de logiciel

a) Gestion de configuration logicielle

La gestion de configuration "Change Management" (CM) est la discipline de la gestion des modifications dans les grands systèmes. Le but de CM est de gérer et contrôler les différentes extensions, adaptations et corrections appliquées à un système tout au long de son cycle de vie. La CM gère aussi le contrôle de tous les produits/éléments de configuration et les modifications apportées à ces éléments. La gestion de configuration logicielle (SCM) est la gestion de la configuration appliquée aux systèmes logiciels. La SCM est le moyen par lequel le processus d'évolution de logiciel est géré. une norme SCM a été définie dans la norme IEEE 1042 (Electrical & Engineers, 1988) comme suit : "La gestion de configuration logicielle (SCM) est la discipline de gestion et de contrôle des modifications de l'évolution des systèmes logiciels". La SCM fournit un cadre pour gérer les modifications de manière contrôlée. Le SCM a pour objectif de réduire les erreurs de communication entre le personnel travaillant sur différents aspects du projet de logiciel en fournissant un référentiel

centralisé d'informations sur le projet et un ensemble de procédures convenues pour faire face aux changements. Cela garantit que le logiciel publié n'est pas contaminé par des modifications non contrôlées ou non approuvées. Les premiers outils SCM avaient des capacités limitées en termes de fonctionnalités et d'applicabilité. Cependant, les systèmes SCM modernes offrent des fonctionnalités avancées grâce auxquelles de nombreux artefacts différents sont gérés. Par exemple, les systèmes SCM modernes aident leurs utilisateurs à créer un programme exécutable à partir de ses fichiers source versionnés. De plus, il doit être possible de régénérer les anciennes versions du logiciel. En général, un système SCM comporte quatre éléments différents, chaque élément adressant un besoin utilisateur distinct comme suit (Narayanaswamy & Scacchi, 1987) (Leblang, 1995):

- *Identification des configurations logicielles* : Cela inclut les définitions des différents artefacts, leurs lignes de base, ainsi que les modifications apportées aux artefacts.
- *Contrôle des configurations logicielles* : Cet élément concerne le contrôle de la manière dont les artefacts ou les configurations sont modifiés avec le support technique et administratif nécessaire.
- *Audit des configurations logicielles* : Cet élément consiste à rendre visible pour la direction le statut actuel du système logiciel au cours de son cycle de vie et à déterminer si les références de base répondent ou non à leurs exigences.
- *Statut de configuration du logiciel* : Cet élément consiste à fournir un historique administratif de la manière dont le système logiciel a été modifié, en enregistrant les activités requises par les trois autres éléments "Software change Management" SCM.

b) Réingénierie

Hongji et Martin (Yang & Ward, 2003) ont défini l'évolution logicielle comme "... le processus de réingénierie logicielle continue ". La réingénierie implique un cycle unique consistant à prendre un logiciel existant et à en générer un logiciel de nouveau, alors que l'évolution peut durer une éternité. En d'autres termes, dans une large mesure, l'évolution des logiciels peut être considérée comme une réingénierie

logicielle répétée. La réingénierie est effectuée pour transformer un système existant "plus petit ou plus simple " en un nouveau logiciel «meilleur». La réingénierie est définie comme " l'examen et la modification d'un système sujet pour le reconstituer sous une nouvelle forme et la mise en œuvre ultérieure de la nouvelle forme" (Shi & Olsson, 2006).

Par conséquent, la réingénierie comprend une sorte d'activités d'ingénierie inverse permettant de concevoir une vue abstraite d'un système donné. La nouvelle vue abstraite est restructurée et des activités d'ingénierie aval sont exécutées pour mettre en œuvre le système dans sa nouvelle forme.

Lors de l'ingénierie inverse sur un grand système, les outils et les méthodologies ne sont généralement pas stables. Par conséquent, un paradigme organisationnel de haut niveau permet la répétition de processus afin que les ingénieurs de maintenance en apprennent davantage sur le système. Benedusi et al. (Benedusi, Cimitile, & De Carlini, 1992) ont proposé un paradigme reproductible, appelé Objectifs/Modèles/Outils, qui décrit l'ingénierie inverse en trois étapes successives, à savoir Objectifs, Modèles et Outils (Sommerville, 2011).

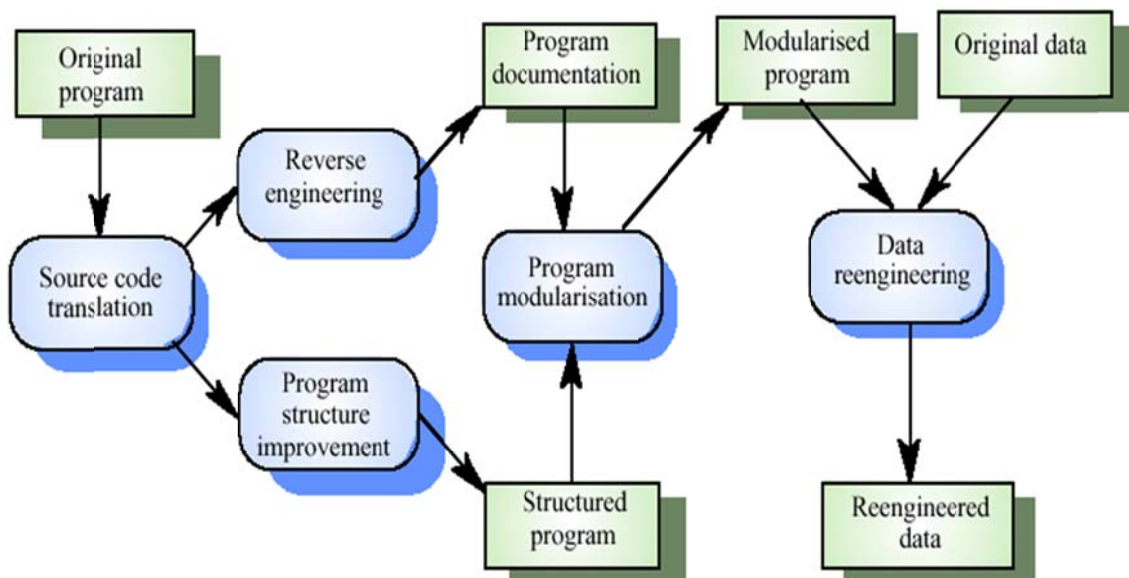


Figure 9 - Le processus de réingénierie (Sommerville, 2011)

c) Rétro ingénierie

La rétro-ingénierie d'un logiciel est un processus d'analyse de la version opérationnelle de ce composant qui vise à en reconstruire les spécifications techniques et fonctionnelles. La Rétro-ingénierie a comme but la re-documentation, la conversion, la maintenance ou l'évolution d'anciennes applications. (Henrard, Roland, Englebert, Hick, & Hainaut, 1998)

Il est démontré qu'au cours de l'évolution du logiciel, la description la plus fiable et la plus précise du comportement d'un système logiciel est son code source (Tonella, 2005). En fait, les diagrammes de conception sont souvent dépassées ou absentes. Un dépôt d'informations si précieux ne répondra peut-être pas directement à toutes les questions sur le système. La rétro ingénierie permet d'extraire des vues de plus haut niveau du système, qui résument certains aspects pertinents du calcul effectué par les énoncés du logiciel. Les diagrammes inversés prennent en charge la compréhension du logiciel, ainsi que la restructuration et la traçabilité.

Une problématique importante des techniques de rétro ingénierie est quelle doit prendre en compte c'est le fait de la facilité d'utilisation (Tonella, 2005). Puisque les vues récupérées sont pour les humains et non pour les ordinateurs, ils doivent être compatibles avec les capacités cognitives des êtres humains. Plusieurs approches peuvent être adoptées pour soutenir la visualisation est le mode de navigation qui permet l'utilisation des informations issues de la rétro-ingénierie. Elles vont de la possibilité de se concentrer sur une partie du système, à la possibilité de se concentrer sur l'ensemble du système, d'agrandir/réduire ou de faire un zoom avant/arrière, ou à la disponibilité d'une vue d'ensemble de carte de navigation complétée et par vue détaillée.

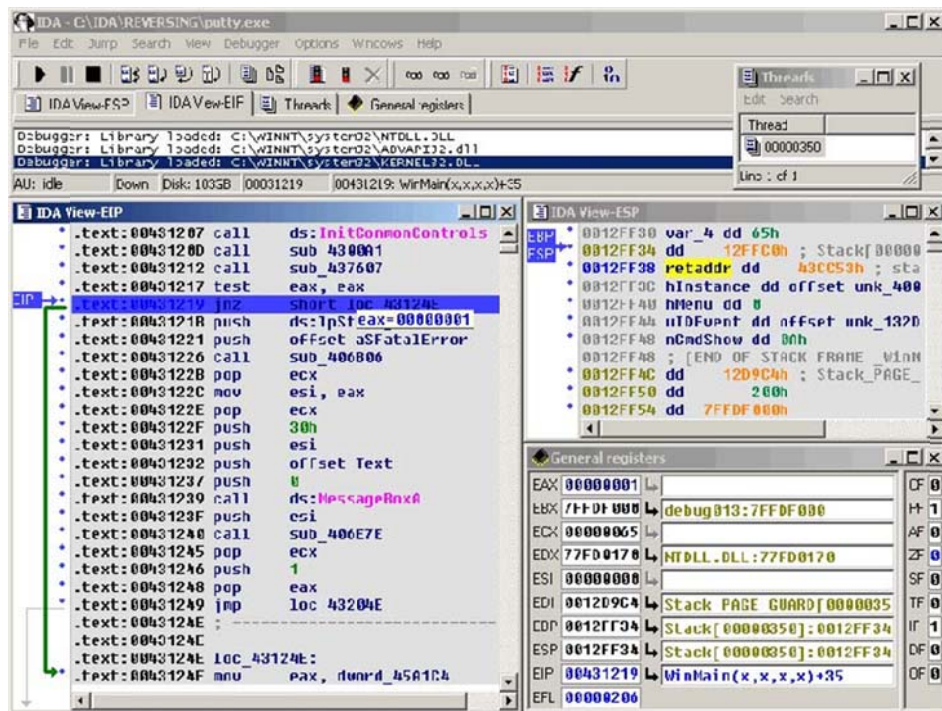


Figure 10 - Outil de rétro ingénierie IDA Pro.(Conti, Dean, Sinda, & Sangster, 2008)

d) Analyse d'impact

L'analyse d'impact consiste à identifier les parties du logiciel susceptibles d'être affectées si une modification proposée du système est effectuée. Les résultats de l'analyse d'impact peuvent être utilisés lors de la planification des modifications, de leur réalisation et du suivi de leurs effets afin de localiser la source des nouvelles fautes. Une analyse d'impact de changement s'impose afin de :

- (i) Déterminer tous les composants du système qui doivent être changé en conséquent du changement initial
- (ii) Estimé les ressources nécessaire pour implémenté le changement.

Ce résultat permet au gestionnaire de prendre une décision sur la meilleure solution à mettre en œuvre ou annulé la requête du changement

5. Critères d'évaluation de la qualité du code source d'un logiciel

Les critères fréquemment utilisés pour caractériser et évaluer la qualité d'un code source sont les métriques. Avant de décrire les métriques, il y a lieu d'expliquer

le concept de "qualité logicielle ". En effet, la qualité du code, aussi appelée qualité structurelle du logiciel, est différente de la qualité d'un logiciel, aussi appelée qualité fonctionnelle d'un logiciel, dans le sens où la première concerne la manière avec laquelle une fonctionnalité est implémentée et la seconde le résultat final. Dans son ouvrage, " Metrics and Models in Software Quality Engineering ",(Kan, 2002) définit la qualité fonctionnelle d'un logiciel comme la "conformité aux exigences". On pourrait dire qu'un logiciel est de qualité s'il a un nombre faible de bogues fonctionnels. La qualité structurelle d'un logiciel est donc la manière avec laquelle ces fonctionnalités sont implémentées et inclut entre autres la robustesse, la maintenabilité, la lisibilité ou encore l'évolutivité.

6. Les Métriques de logiciel

Les métriques représentent une échelle quantitative et des méthodes, qui peuvent être employées pour déterminer la valeur que prend une propriété pour un produit logiciel spécifique. Avant d'être utilisée une métrique suit le processus de mesure (Miloud Dahane, 2011), commence par la création de la méthode de mesure et se termine par son utilisation.

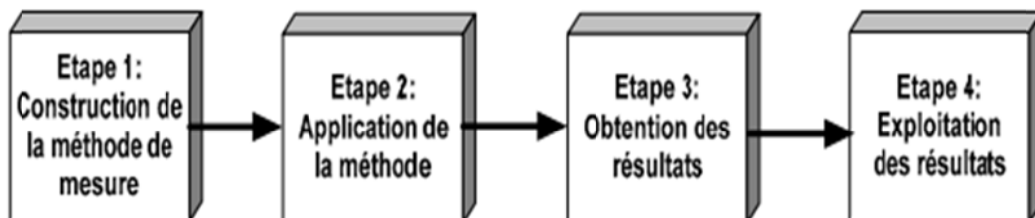


Figure 11 - Processus de mesure

- ✓ Etape 1 : Pour pouvoir mesuré, la construction de méthode de mesure est nécessaire.
- ✓ Etape 2 : Les règles de méthode de mesure sont appliquées soit sur le logiciel soit sur la partie de logiciel a mesuré.
- ✓ Etape 3 : Afin d'obtenir un résultat il faut appliquée les règles de méthode de mesure. Ce résultat est mesuré

- ✓ Etape 4: Les résultats fournis par la méthode de mesure sont exploités dans des modèles qualitatifs ou quantitatifs

a) Différents types de métriques

Plusieurs classifications des métriques peuvent être utilisées. Elles peuvent se faire par :

1) Métriques de produit

Les métriques concernant le logiciel sont celles du produit. Elles décrivent une propriété du logiciel. Exemples : nombre de fonctions, d'opérateurs par fonction, complexité cyclomatique, etc.

2) Métriques de processus

Elles décrivent une propriété du processus relatif au logiciel (développement, maintenance, tests, etc.). Exemples : effort de développement, productivité, etc.

3) Métriques du code

- a) lignes de code, nombres d'opérandes, d'opérateurs, etc.
- b) complexité cyclomatique (nombre de chemins à emprunter).
- c) taux de commentaires

4) Métriques de spécification (conception)

- a) cohésion et couplage des modules
- b) taille et fréquence de communication de données entre les différents modules

5) Métriques du processus de gestion

- a) mesure de la capacité à estimer
- b) mesures liées à la documentation (taille, modularité, ...)

6) Métriques du processus qualité

Nombres de revues, d'inspections, etc.

b) Quelques métriques de conception orientée objet

Les métriques de conception permettent d'évaluer la structure, les relations et les fonctionnalités des composants d'un système logiciel. Un logiciel peut être

analysé à deux niveaux : au niveau système, les caractéristiques externes telles que la hiérarchie des classes et les relations entre elles sont évaluées; au niveau classe, les caractéristiques internes qui sont évaluées sont les méthodes, les attributs et tous types d'interactions. Parmi les aspects les plus importants dans une conception orientée objet, on cite la cohésion, le couplage et l'héritage.

1) Métrique d'héritage

La métrique d'héritage mesure la position de classe dans l'arbre d'héritage le nombre de superclasse, des méthodes héritées et le nombre de sous-classe. Ces métrique ont été définie par :

1. Weighted Methods per Class (WMC)

Somme pondérée des complexités des méthodes d'une classe. Si toutes les méthodes d'une classe sont de complexité 1, elle est égale au nombre de méthodes

$$WMC(c) = \sum_{m \in M_I(c)} comp(m)$$
$$WMC(c) = |M_I(c)| \text{ si } \forall m \in M_I(c), comp(m) = 1 \quad (1)$$

$M_I(c)$: ensemble des méthodes implantées dans la classe c

C : ensemble de toutes les classes du système

2. Depth of Inheritance Tree (DIT)

Profondeur de la classe dans l'arbre d'héritage. En cas d'héritage multiple, DIT vaudra la profondeur maximale depuis la classe racine jusqu'à la classe mesurée

$$DIT(c) = \begin{cases} 0, & \text{if } parents(c) = \emptyset \\ 1 + \max\{DIT(c') : c' \in parents(c)\} & \text{sinon} \end{cases} \quad (2)$$

C : ensemble de toutes les classes du système

3. Number Of Children (NOC)

Nombre de descendants immédiat de la classe dans la hiérarchie de classe.

$$NOC(c) = |Children(c)| \quad (3)$$

C : ensemble de toutes les classes du système

Parents(c) : ensemble des superclasses directes de c

Children(c) : ensemble des sous-classes directes de c

2) Métrique de cohésion

Elle représente le degré avec lequel les méthodes d'une classe interagissent,

1. Lack of COhesion in Methods (LCOM1)

LCOM1 compte le nombre de paires de méthodes qui n'accèdent pas aux mêmes attributs

$$LCOM1(c) = |P| \quad (4)$$

$$P = |[m_1, m_2]: m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \emptyset|$$

C : ensemble de toutes les classes du système

$M_I(c)$: ensemble des méthodes implantées dans la classe c

$AR(m)$: ensemble des attributs référencés par une méthode m

2. Lack of COhesion in Methods (LCOM2)

LCOM2 compte le nombre de paires de méthodes qui n'accèdent pas aux mêmes attributs diminué du nombre de paires de méthodes qui accèdent aux mêmes attributs (= 0 si la différence est négative)

$$LCOM2 = \begin{cases} |P| - |Q| & \text{si } |P| > |Q| \\ 0 & \text{sinon} \end{cases} \quad (5)$$

$$Q = |[m_1, m_2]: m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \emptyset|$$

C : ensemble de toutes les classes du système

$M_I(c)$: ensemble des méthodes implantées dans la classe c

$AR(m)$: ensemble des attributs référencés par une méthode m

3) Métrique de couplage

Le couplage indique le niveau d'interaction entre deux ou plusieurs composants logiciels. Deux composants sont dits couplés s'ils échangent de l'information. On parle de couplage fort ou couplage serré si les composants échangent beaucoup d'information. On parle de couplage faible, couplage léger ou couplage souple si les composants échangent peu d'information.(McGraw-Hill, 1992)

7. Conclusion

L'évolution est un aspect important dans le cycle de vie d'un logiciel. Diverses techniques comme l'analyse de l'impact de changements sont utilisées pour garantir le bon fonctionnement du système et assurer son évolution. L'étude de l'impact de changement est une activité fondamentale dans la gestion du changement car elle peut servir à la planification des changements, à les mettre en place et à prévoir ou détecter leurs effets sur le reste du système.

III. Impact de changement

1. Introduction

Un changement, même élémentaire, peut avoir des conséquences inattendues sur le fonctionnement global du système. Par exemple, modifier la visibilité d'un attribut peut avoir des résultats non attendus sur le fonctionnement du système, ces conséquences sont plus néfastes si l'on n'arrive pas à déterminer les éléments qui sont affectés par le changement réalisé. Une fois ces éléments détectés, ils doivent être modifiés par les mainteneurs afin de garantir le bon fonctionnement du système. Ce besoin d'identifier les éléments affectés par un changement constitue l'objectif de la tâche d'analyse de l'impact de changement.

2. Définition de l'impact de changement

Un impact, dans son sens le plus large, est l'effet qu'une chose a sur une autre. L'impact est souvent la conséquence d'un changement (Ryder & Tip, 2001). L'analyse d'impact de changement est la détermination de la portée d'une modification, i.e., l'évaluation des éléments d'un système qui seront affectés par cette modification. Dans un logiciel, la modification d'une partie d'un code peut avoir des répercussions sur d'autres parties, obligeant le programmeur à modifier ou re-tester ces parties. On parle pour cela de propagation d'une modification (ripple effect). Deux aspects d'un changement peuvent être considérés, l'aspect quantitatif, i.e., connaître le nombre d'éléments touchés et l'aspect qualitatif, à savoir comment un élément a été affecté (Abdi 2007).

3. Processus du changement

Une demande de changement active un processus pour modifier un système logiciel afin d'assurer sa maintenance. Le processus de maintenance est lancé en effectuant une analyse d'impact. L'analyse d'impact consiste essentiellement à identifier les composants impactés par la demande de changement (Mens, 2008; Queille, Voidrot, Wilde, & Munro, 1994; Truver & Munro, 1991). L'analyse d'impact permet de comprendre et de mettre en œuvre les changements dans le système. Les



effets potentiels des modifications proposées sont visibles en effectuant une analyse d'impact. En d'autres termes afin de faciliter des changements dans un environnement, il est nécessaire de connaître tous les facteurs influençant un changement donné, et les répercussions de ce dernier sur le reste du système. L'analyse de l'impact tente d'apporter des solutions à ce problème.

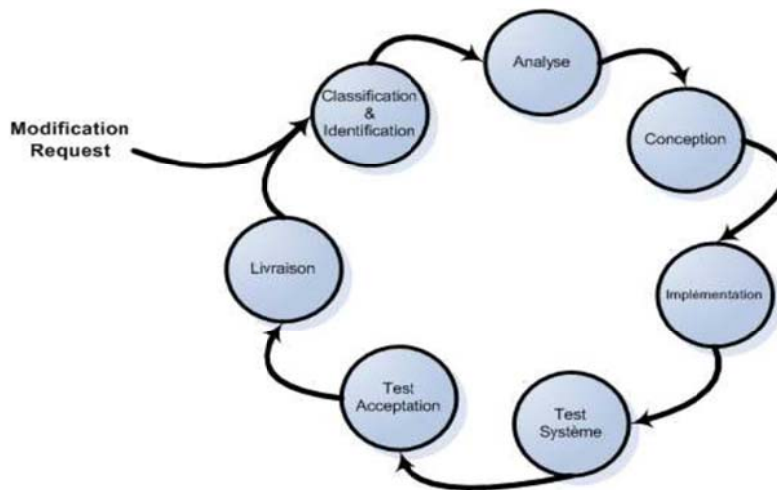


Figure 12 - Les sept phases du processus de maintenance (Canfora & Cimitile, 2001)

Dans leur travail, Bohner et Arnold (Bohner & Arnold, 1996) décomposent l'analyse de l'impact de changement en six étapes :

1. Gérer les changements du logiciel,
2. Comprendre les changements,
3. Identifier les impacts,
4. Spécifier et concevoir les changements,
5. Implanter les changements,
6. Re-tester le logiciel affecté.

4. Définitions de l'analyse de l'impact de changement

Il existe plusieurs manières d'analyser l'impact de changement. On peut lister les suivantes :

- Turver et Munro (Turver & Munro, 1994) définissent l'analyse d'impact en tant qu'évaluation de l'impact d'une modification du code source d'un module sur les autres modules du système. Il détermine l'étendue d'un changement et fournit une mesure de sa complexité.
- Queille et al. (Queille et al., 1994) définissent l'analyse d'impact comme la tâche d'évaluer les effets de la modification d'un système logiciel.
- Bohner et Arnold (Bohner & Arnold, 1996) définissent l'analyse d'impact comme l'identification des conséquences potentielles d'une modification ou la découverte des entités si on accomplit un changement.

Les trois définitions mettent l'accent sur l'estimation des impacts potentiels, ce qui est crucial pour les tâches de maintenance, car ce qui est réellement changé ne sera pas complètement connu avant la fin du changement de logiciel.

5. Intérêts de l'analyse de l'impact de changement

Avant d'exécuter une demande de modification, l'impact des modifications est analysé pour les raisons suivantes (Ajila, 1995; Lehnert, 2011).

- Estimer le coût de la réalisation de la demande de changement. L'exécution d'une demande de modification implique des coûts. Si une demande de modification peut potentiellement avoir un impact sur de grandes parties disjointes du système, réexaminer la demande. Un changement important entraînera probablement des coûts plus élevés et rendra le système incohérent. Par conséquent, si l'ampleur du changement est importante, rejeter la demande en faveur d'un changement plus sûr.
- Déterminer si certaines parties critiques du système vont être affectées par le changement demandé. Si tel est le cas, davantage de ressources doivent être allouées pour exécuter la demande de changement.
- Enregistrer l'historique des informations relatives aux modifications pour une évaluation future des modifications. En prenant en compte les commentaires des utilisateurs finaux, la qualité globale des modifications peut être évaluée ultérieurement.

- Comprendre comment les éléments de changement sont liés à la structure du logiciel. Cela permettra aux ingénieurs de maintenance de mieux comprendre la demande de changement en cours.
- Déterminer les parties du logiciel qui doivent être soumises à un test de régression après qu'une modification ait été effectuée.

6. Techniques d'analyse d'impact

L'implémentation des demandes de changement affecte tous les types d'artefacts, incluant le code source, les exigences, la documentation de conception et les scénarios de test, etc. Par conséquent, les informations de traçabilité de l'analyse peuvent être utilisées pour effectuer une analyse d'impact (De Lucia, Fasano, & Oliveto, 2008). Gotel et Finkelstein (Gotel & Finkelstein, 1994) définissent la traçabilité comme la capacité de décrire et de suivre la vie d'un artefact dans les deux sens. Bohner et Arnold (Bohner & Arnold, 1996) définissent la traçabilité comme la capacité de tracer des artefacts logiciels générés et modifiés au cours du cycle de vie du logiciel. Ainsi, la traçabilité aide les développeurs de logiciels à comprendre les relations entre tous les artefacts logiciels.

Après avoir identifié les documents de haut niveau concernant la fonctionnalité à modifier, en utilisant le concept de traçabilité, le mainteneur localise les entités à modifier. Ces entités dépendent soit de la conception, soit du code source.

a) Analyse de traçabilité

Dans cette approche, les artefacts, tels que les exigences, la conception, le code et les cas de test liés à la fonctionnalité à modifier, sont identifiés. Un modèle d'associations entre artefacts, tel que chaque artefact est lié à d'autres artefacts, est construit. Cela aide à localiser les parties correspondantes de la conception, du code et des scénarios de test à gérer.

Il existe deux grands types de traçabilité: (i) la traçabilité horizontale (externe); et (ii) la traçabilité verticale (interne). La traçabilité des artefacts entre différents modèles est appelée traçabilité externe, tandis que la traçabilité interne fait référence au traçage des artefacts dépendants dans le même modèle. La traçabilité interne se

concentre principalement sur les artefacts de code source (Li, Sun, Leung, & Zhang, 2013). Dans ce contexte, les trois techniques classiques d'analyse d'impact, basées sur la dépendance de programme, sont celles basées sur un graphe d'appel, la découpe de programme statique et la découpe dynamique du programme "program slicing". Par exemple, l'insertion de modifications dans le code et l'exécution d'une découpe en aval à partir des points modifiés génèrent un ensemble d'instructions susceptibles d'être affectées par les modifications.

b) Analyse de dépendance

L'analyse de dépendance tente d'évaluer les effets d'un changement sur les dépendances sémantiques entre les entités de programme. Ceci est réalisé en identifiant les dépendances syntaxiques susceptibles de signaler la présence de telles dépendances sémantiques (Podgurski & Clarke, 1990). Les deux techniques d'analyse d'impact basées sur la dépendance sont (Harrold & Malloy, 1993): l'analyse par graphe d'appel et l'analyse par graphe de dépendance. L'analyse de dépendance peut concerner soit l'analyse de code source ou bien la dépendance entre les artefacts de conception.

Les deux notions suivantes sont aussi utilisées pour la compréhension de l'analyse d'impact:

c) Analyse de l'effet d'entraînement

L'analyse des effets d'ondulation met l'accent sur les répercussions du traçage dans le code source lorsque le code est modifié. Elle mesure l'impact d'un changement apporté à un module particulier sur le reste du programme (Black, 2008). L'impact peut être exprimé en termes de problèmes créés pour le reste du programme en raison du changement. L'analyse de l'effet d'entraînement peut fournir des informations sur les changements en cours et leur lieu. La mesure de l'effet d'entraînement peut fournir des connaissances sur le système dans son ensemble à travers son évolution (i) le montant de l'augmentation ou de la diminution de sa complexité depuis la version précédente; (ii) les niveaux de complexité des différentes parties d'un système par rapport à d'autres parties du système; et (iii) l'effet qu'un nouveau module a sur la complexité d'un système dans son ensemble.

d) Propagation du changement

Les activités de propagation des modifications garantissent qu'une modification apportée à un composant est propagée certainement dans tout le système (N. Ibrahim, Kadir, & Deris, 2008). Si une modification n'est pas propagée correctement, le projet risque d'introduire de nouveaux défauts. (Perry & Evangelist, 1987).

7. Effet de vague "Ripple effect "

L'effet de vague "*Ripple effect* " signifie qu'une modification d'une seule variable peut nécessiter la modification de plusieurs parties du système logiciel. Le concept d'effet de vague est pertinent dans l'évolution du logiciel car il concerne les changements et leurs effets (Black, 2006). L'analyse de l'effet de vague révèle où les changements se produisent. La mesure des effets de vague peut fournir les informations sur un système logiciel en cours d'évolution ainsi que sur sa complexité dans le cas de l'ajout d'un nouveau module.

La mise en œuvre d'une demande de modification peut comporter plusieurs étapes. À chaque étape, un module spécifique identifié lors de l'analyse d'impact, est visité et refactorisé. Si un composant visité est modifié, il peut ne plus interagir correctement avec d'autres composants. Par conséquent, des modifications supplémentaires doivent être apportées aux composants voisins, ce qui peut déclencher d'autres modifications dans tout le système logiciel. Ce processus est nommé "activité de propagation du changement" (V. Rajlich, 1997). L'activité de propagation des modifications garantit qu'une modification apportée à un composant est propagée correctement dans tout le système. Les ingénieurs de maintenance ne parviennent pas à propager correctement et complètement les modifications apportées à un système logiciel pour un certain nombre de raisons: manque d'expérience, dépendances inattendues entre les blocs de code et compréhension erronée des détails de conception. En conséquence, les projets risquent de générer de nouveaux défauts d'interface si les modifications ne sont pas correctement propagées (Perry & Evangelist, 1987).

8. Approches d'analyses d'impact de changement

Plusieurs études ont été menées sous différentes formes à propos de l'analyse d'impact.

Turver et Munro (Turver & Munro, 1994) ont proposé l'idée d'un graphe de propagation par ondulation pour la modélisation des entités de documentation et de leurs liens avec le code source.

Lindvall *et al.* (Lindvall & Sandahl, 1998) montrent un traçage à travers les phases avec des dépendances intra-niveaux et inter-niveaux. Les auteurs discutent également d'une technique d'analyse d'impact basée sur les données de traçabilité d'un système orienté objet.

Antoniol *et al.* proposent une approche pour prédire la taille des changements des systèmes Orientés Objets en évolution, basée sur l'analyse des classes impactées par la requête du changement (Antoniol, Canfora, & De Lucia, 1999). Ils prédisent la taille du changement en termes de nombre de lignes de code (LOC) ajoutées et modifiées. L'approche a été évaluée empiriquement par l'analyse de la relation entre le nombre de LOCs ajoutées/modifiées et le nombre de classes ajoutées/modifiées sur 31 versions d'un système. L'analyse des codes sources a montré que le LOCs a augmenté considérablement de la première version à la dernière version. En outre, le nombre de LOCs supprimées est généralement faible, comparé au nombre de LOCs ajoutées et modifiées, et le nombre de LOCs ajoutées est plus grand que le nombre de LOCs modifiées.

Une analyse d'impact pour les architectures logicielles a été étudiée par Zhao *et al.* (Zhao, Yang, Xiang, & Xu, 2002). Sur la base d'un découpage architectural et la technique de hachage, les auteurs proposent dans cette étude une approche pour effectuer une analyse de l'impact du changement au niveau architectural des systèmes logiciels.

Dans (Cantave, 2003), l'analyse d'impact a été faite avec l'objectif de réduire les coûts et la durée des tests de régression. Aucun modèle n'a été défini dans cette

étude. L'analyse a été faite à partir du graphe de dépendance extrait de la base de données (repository).

Ibrahim *et al.* (S. Ibrahim, Idris, Munro, & Deraman, 2005) ont développé un outil de traçabilité, appelé Catia, permettant de suivre les effets de changement. Bianchi *et al.* (Bianchi, Fasolino, & Visaggio, 2000) ont expérimenté de nombreux exemples de lien de traçabilité à l'aide de l'outil ANALYST (Fasolino & Visaggio, 1999); leur objectif était d'évaluer: (i) l'efficacité de ces liens de traçabilité pour supporter l'analyse d'impact dans un environnement orienté objet; et (ii) l'exactitude du processus de maintenance. Un processus de changement a été modélisé comme une séquence d'instanciations par Rajlich "graph rewriting" (V. Rajlich, 1997). Une instanciation représente un moment du processus de changement. Dans cette technique, un programme est représenté par un graphe dirigé, où les nœuds représentent des entités (par exemple, des classes).

Chaumon et Kabaili se sont intéressés à un des aspects de la maintenance qui est la changeabilité (M Ajmal Chaumon, Hind Kabaili, Rudolf K Keller, & François Lustman, 2002). Leur approche consiste à calculer l'impact du changement apporté aux classes du système en utilisant un modèle d'impact. Ce modèle consiste à spécifier un ensemble de changements pouvant intervenir dans une classe, et pour chaque changement son impact sur les classes directement connectées à la classe changée. Par ailleurs, ce modèle d'impact a été étendu par Kabaili pour tenir compte de l'effet de propagation et des tests de régression afin d'obtenir une meilleure évaluation de la changeabilité du système (Kabaili, Keller, & Lustman, 2005). Pour déterminer les classes à re-tester, elle a utilisé une approche inspirée du concept du firewall et basée sur les relations de dépendances entre les classes.

Dans (Abdi, Lounis, & Sahraoui, 2006), les auteurs ont essayé de vérifier empiriquement les relations entre les décisions de conception orientée objet et trois attributs de qualité, à savoir, l'impact de changement, la propension d'erreur (fault-proneness) et la réutilisabilité. Plusieurs algorithmes d'apprentissage automatique ont été sélectionnés et exécutés sur des systèmes de taille moyenne.

Berg *et al.* (Van Den Berg, Conejero, & Hernández, 2007) ont proposé un framework d'analyse d'impact des artefacts logiciels au cours de plusieurs phases de développement logiciel.

Dans (Keller & Demeyer, 2012) une technique d'analyse d'impact du changement présente les relations entre les éléments du modèle UML. Le type de changement (par exemple, ajouter, supprimer, modifier) est utilisé pour réduire le nombre de relations à tracer. Par exemple, pour déterminer l'impact de la création d'un élément de modèle UML, toutes les relations, mais uniquement les relations de liens de cet élément sont suivies. Grâce à l'exploitation du méta-modèle UML et du type de changement, ils présentent un petit ensemble générique de règles d'analyse d'impact.

Dans (Chaturvedi, 2014) l'auteur propose une approche qui est prototypée comme un service Web qui aide à sélectionner les cas de test pertinents pour construire une liste de tests réduite à partir de l'ancienne liste de tests. La réduction de l'effort de test de régression du service Web a été également estimée.

9. Evaluation et propagation d'impact

L'analyse d'impact permet d'une part, l'évaluation de l'impact afin d'avoir une meilleure estimation des ressources, de l'effort et du temps associés à une activité de maintenance (Abdi 2007). En identifiant l'impact potentiel d'une modification, on réduit le risque de s'embarquer dans des changements coûteux et imprévisibles. Et d'autre part, la mesure de la *propagation du changement* sur la base de l'évaluation de la complexité d'un code et la conduite des tests de régression. Les tests de régression sont une importante activité de maintenance et d'assurance qualité, visant à assurer que le comportement d'un programme continu, après modification, de satisfaire aux spécifications initiales. Si chaque modification d'une partie d'un code a un impact sur un grand nombre d'autres parties, cela indique une forte dépendance entre ces parties, un fort couplage et par conséquent une grande complexité (Dinedane, 2008).

10. Outils d'analyse d'impact

Pour aider les développeurs à mieux cerner le concept de l'analyse d'impact, il existe un certain nombre d'outils offrant divers services pour faciliter la mesure de l'impact. Nous identifions ici un certain nombre d'outils et de plug-ins d'analyse d'impact actuellement disponibles dans le milieu universitaire et dans le secteur économique :

a) Chianti

Chianti (Ren, Shah, Tip, Ryder, & Chesley, 2004) est un outil d'analyse d'impact de changement disponible sous la forme d'un plug-in Eclipse qui identifie un ensemble de changements atomiques représentant la différence entre deux versions d'un programme donné. À partir de cet ensemble de modifications, un ensemble de tests de régression potentiellement affectés est identifié.

La méthode à la base de Chianti est capable d'effectuer une analyse statique ou dynamique, mais l'analyse dynamique est utilisée pour illustrer les exemples. La première étape de la méthode consiste à compiler un ensemble de modifications atomiques, classées par type de modification, telles que «classe ajoutée», «méthodes modifiées» ou «champs supprimés». Ensuite, des graphes d'appel sont construits pour chaque test de régression, et l'ensemble des modifications atomiques est mis en corrélation avec leurs équivalents dans les graphes d'appel construits. Enfin, pour chaque test de régression, l'ensemble des modifications atomiques pertinentes est identifié et mis en corrélation.

Cet outil est utile pour les développeurs qui souhaitent analyser l'impact au niveau du code, en fonction de la finesse de l'analyse dont le développeur a besoin. Une interface utilisateur est fournie qui montre les tests de régression affectés et les modifications atomiques pertinentes associées. En cliquant sur un élément dans la fenêtre de résultats, l'utilisateur sera dirigé vers l'emplacement associé dans le code source.

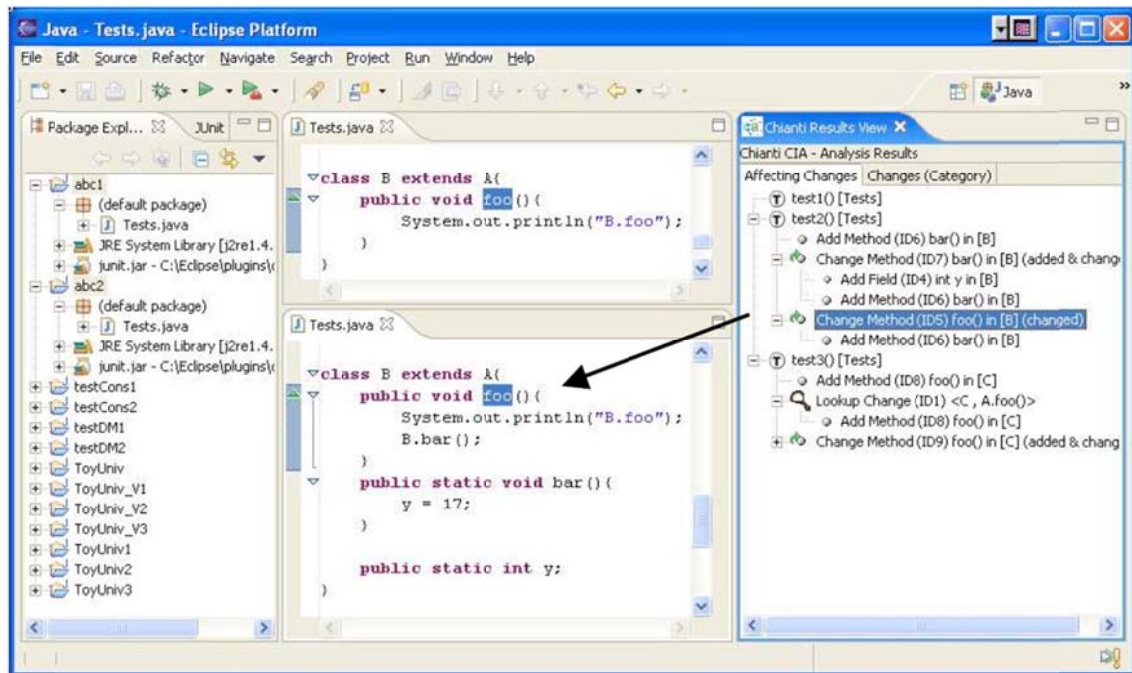


Figure 13 - Capture d'écran du volet "Affecting changes viewer" de Chianti

b) FaultTracer

FaultTracer (Zhang, Kim, & Khurshid, 2012) est un outil d'analyse d'impact des changements open-source implémenté sous forme de plug-in pour Eclipse. À l'aide de l'AST (Java Abstract Syntax Tree), cet outil compare deux versions données d'un programme et, après avoir extrait les modifications, identifie les tests de régression potentiellement affectés. FaultTracer est similaire à Chianti en termes de buts et de techniques sous-jacentes, mais il prétend dépasser de plus de 50% les performances de l'heuristique de classement de Chianti.

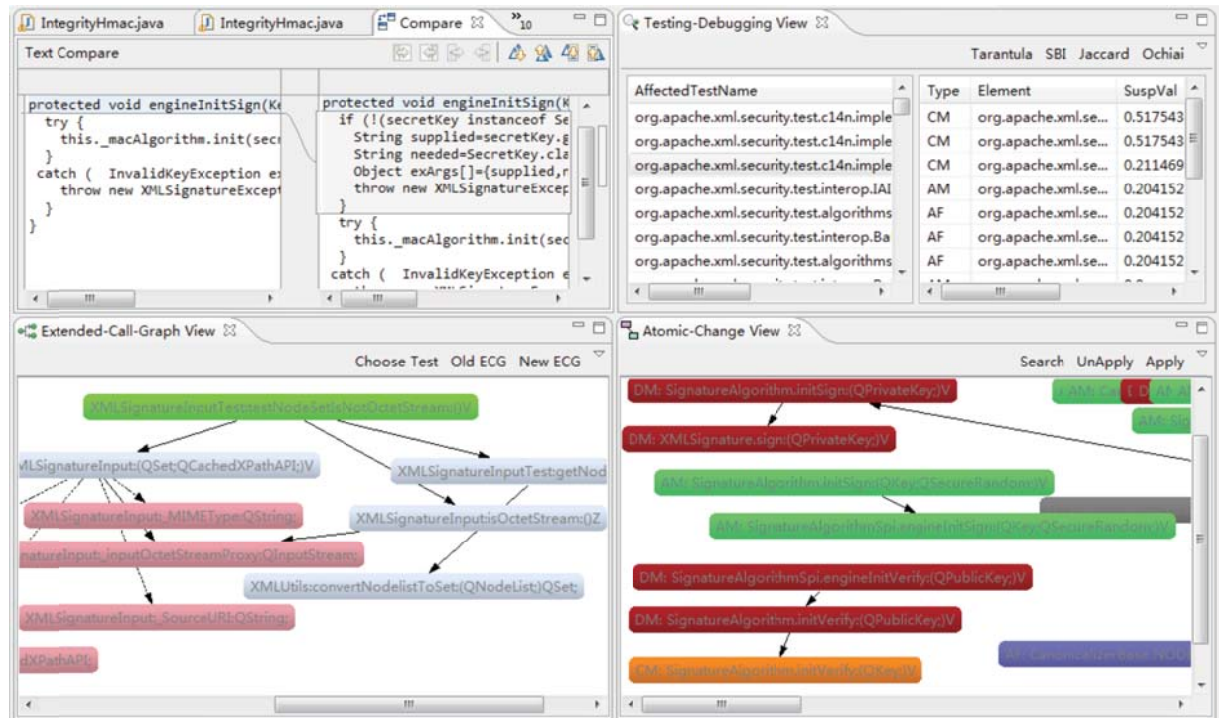


Figure 14 - Vue d'ensemble du front-end de FaultTracer

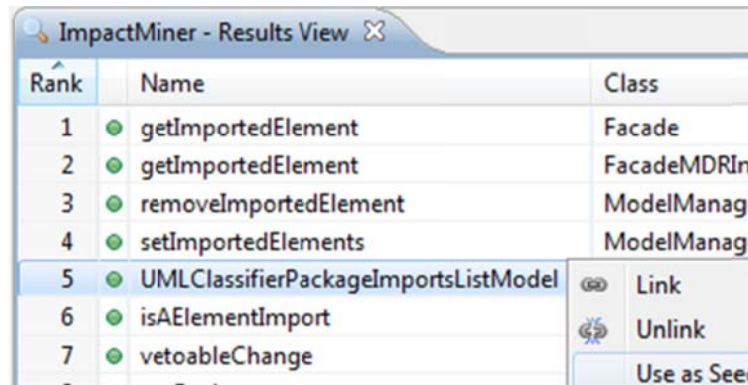
FaultTracer est composé de trois vues: la vue de changement atomique, la vue de graphe d'appels étendue et la vue de test-débogage. La vue de changement atomique permet aux développeurs de visualiser et d'interagir avec les différences entre deux versions sélectionnées. Dans le graphe d'appel étendu (ECG), les développeurs peuvent voir les ECG de chaque test, ce qui peut faciliter la compréhension des modifications qui en découlent. La vue testing-débogage affiche la sortie finale des algorithmes de FaultTracer, y compris les tests affectés, et un classement des modifications atomiques pertinentes pour chaque test.

Le développeur doit disposer de deux versions différentes d'un projet dans le même espace de travail pour pouvoir exécuter FaultTracer. Une suite de tests de régression doit également être fournie.

c) ImpactMiner

ImpactMiner (Dit et al., 2014) est un outil d'analyse d'impact de changement utilisant l'extraction de référentiels de code source et proposant trois techniques d'analyse: extraction d'informations (IR), analyse dynamique (Dyn) et analyse historique (Hist). Implémenté en tant que plug-in Eclipse, ImpactMiner peut analyser

des programmes Java et, en utilisant l'intégration à subversion, notifier au développeur d'autres méthodes qui sont généralement modifiées en liaison avec les modifications actuellement identifiées.



Rank	Name	Class
1	getImportedElement	Facade
2	getImportedElement	FacadeMDRIn
3	removeImportedElement	ModelManag
4	setImportedElements	ModelManag
5	UMLClassifierPackageImportsListModel	
6	isAElementImport	
7	vetoableChange	

Link
Unlink
Use as Seen

Figure 15 - Vue des résultats d'ImpactMiner

d) JRipples

JRipples (Buckner, Buchta, Petrenko, & Rajlich, 2005) est un plugin Eclipse qui vise à aider les développeurs avec les tâches de propagation de changement et de changement incrémentiel. Constitué de trois composants (un analyseur syntaxique, une base de données avec des règles d'organisation et une interface utilisateur), il analyse les dépendances au sein d'un projet Java et renvoie un ensemble de résultats de classes potentiellement affectées que le développeur peut réviser. Le développeur passe ensuite en revue les résultats, un par un, et les marque comme «impactés» ou «visités».

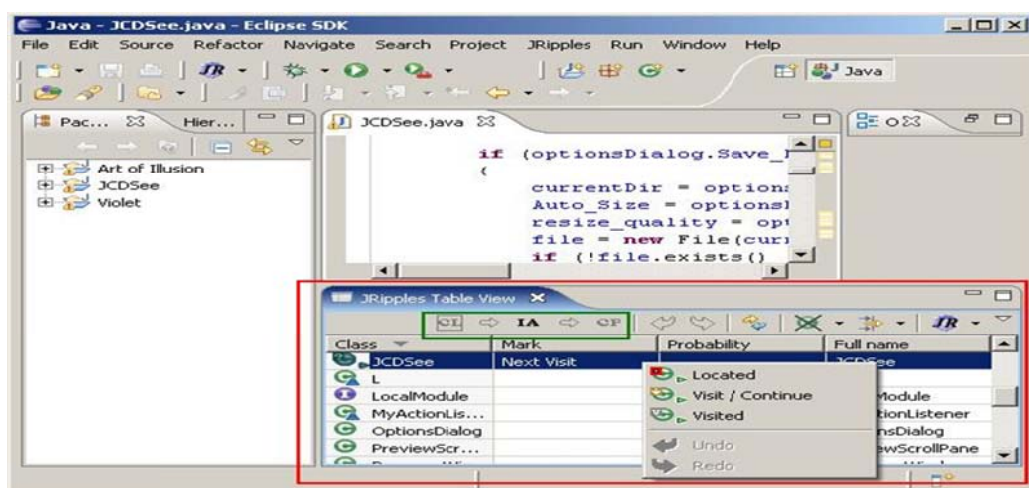


Figure 16 - JRipples View (Environnement Eclipse)

e) Lattix

Lattix (Galbo, 2017) est un outil commercial qui peut être utilisé par les architectes, les développeurs, les gestionnaires et les ingénieurs d'assurance qualité pour mieux comprendre l'architecture de projet et l'impact des modifications tout au long du cycle de vie du logiciel. Une solution de système d'entreprise est proposée qui permet aux développeurs d'effectuer une analyse d'impact et de comprendre l'impact potentiel des modifications sur le reste du système.

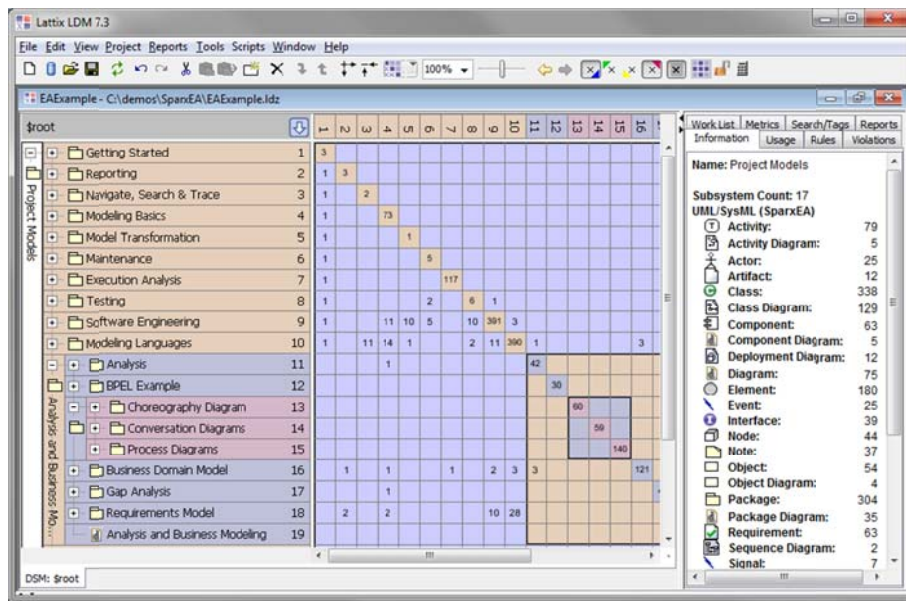


Figure 17 - Vue d'ensemble du front-end de Lattix

En plus, Lattix prend en charge une variété de langages autres que Java, C et C++.

f) JArchitect

JArchitect (Galbo, 2017) est un outil commercial d'analyse statique pour les programmes Java. Il offre un large éventail de fonctionnalités, y compris des métriques de qualité de code, une surveillance des tendances, des fonctions de diagramme et un graphique interactif de dépendance. Les développeurs peuvent sélectionner une entité dans leur projet, et JArchitect affichera le graphique de dépendance montrant les projets, packages, méthodes ou champs potentiellement affectés. D'autres types de graphiques peuvent également être générés, afin de comprendre les relations entre les éléments, tels qu'un graphique de couplage, un graphique de trajectoire, un graphique de toutes les trajectoires et un graphique de cycle.

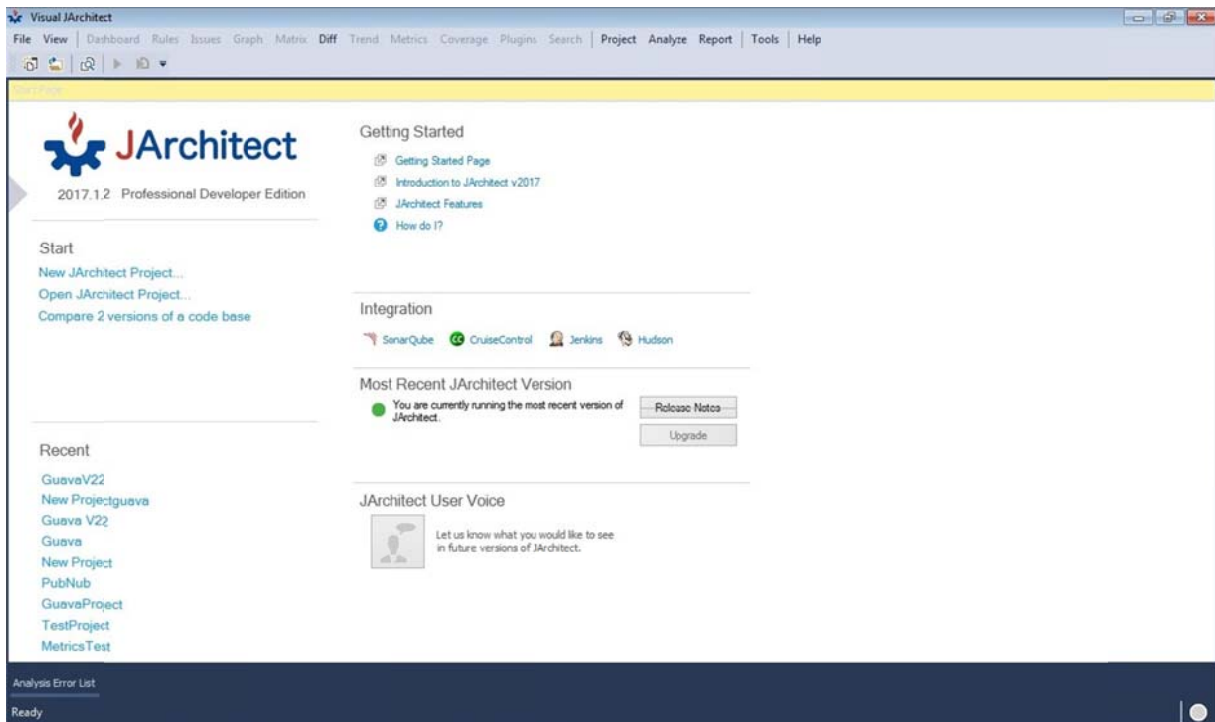


Figure 18 - Vue d'ensemble du front-end de JArchitect

11.Conclusion

L'étude de l'impact de changement est une activité fondamentale dans le génie logiciel car elle peut servir à planifier des changements, à les mettre en place et à prévoir ou détecter leurs effets sur le système et essayer de les réduire.

Le facteur principal qui rend la réalisation du changement difficile dans un logiciel orientée objet est les dépendances entre les classes. Un changement impacte la structure, les performances, les fonctionnalités et le comportement du système. La sévérité de cet impact peut dépendre du degré d'interdépendance entre les composants du système.

Le chapitre suivant présente la technique utilisée, inspirée des sciences expérimentales, pour estimer l'impact de changement.

IV. La technique de plan d'expériences "*Design of Experiments*" (DoE)

1. Introduction

De nos jours, il est courant de trouver de nombreux chercheurs utilisant les statistiques. Ces techniques aident à comprendre l'importance des paramètres afin de définir un modèle qui est une abstraction du système réel et représente ses propriétés et ses caractéristiques. Ce modèle peut être utilisé pour calculer, analyser et prédire le fonctionnement du système réel à moindre coût et pour certains cas qui sont impossibles à réaliser.

Parmi ces techniques, la technique de plan d'expériences "*Design of Experiments*" (DoE), est un ensemble homogène d'outils et de méthodes algébro-statistiques visant à établir et analyser les relations existant entre les grandeurs étudiées (réponses) et leurs sources de variations supposées (facteurs). Elle a pour but la détermination de modèles mathématiques approchés des réponses exprimées en fonction des facteurs. Ces modélisations sont déduites des valeurs obtenues à l'issue de séries de simulations : les plans d'expériences. La définition de ces derniers détermine la qualité mesurable des modèles.

2. Définition

La technique de plan d'expériences "*Design of Experiments*" (DoE) a été introduite dans les années 1920 par Sir Ronald A. Fisher dans le domaine de la recherche agricole (Montgomery, 1995). Depuis, de nombreuses publications sur les aspects de la DoE, tels que Wu et Hamada (Wu & Hamada, 2000) et Box et al. (Box, Hunter, & Hunter, 2005) ont été produites.

La DoE fournit une approche systématique pour réaliser les expériences et obtenir une relation entre les paramètres du processus d'entrée et les réponses de sortie. En utilisant cette technique, le nombre d'expériences pour un problème particulier peut être minimisé, mais les influences ou les dépendances des paramètres d'entrée sur la sortie peuvent être établies de manière satisfaisante (Davim & Aveiro, 2016).

Autrement dit, la technique de plan d'expériences cherche à déterminer et à établir les liens existant entre deux types de variables (Vivier, 2002) :

La réponse : grandeur physique étudiée ;

Les facteurs : grandeurs physiques modifiables par l'expérimentateur, influant sur les variations de la réponse.

Plus précisément, elle vise aussi bien à comprendre les relations liant la réponse avec les facteurs, que les facteurs entre eux.

Pour cela, la solution proposée consiste dans tous les cas à établir **un modèle**, exprimant la réponse en fonction des facteurs.

Pour élaborer ce modèle, il faut appréhender ces variations. Celles-ci sont déduites des résultats de séries d'expériences, c'est-à-dire de plusieurs configurations pour lesquelles les valeurs des facteurs sont différentes, correspondant à la réalisation d'une série de "N" expériences.

La réalisation d'un plan revient à déterminer la valeur de la fonction réponse pour ces "N" configurations. Une relation de modélisation en est alors déduite (Vivier, 2002). Elle permet de définir le lien qui existe entre une grandeur d'intérêt "**y**" et des variables x_i sous la forme d'une fonction de type (Jacques Goupy, 1995) :

$$y = f(x_1, \dots, x_n) \quad (6)$$

La compréhension de la DoE s'appuie sur deux notions essentielles, celle *d'espace expérimental* et celle de *modélisation mathématique* des grandeurs étudiées.

3. Espace expérimental

Un expérimentateur qui lance une étude s'intéresse à une grandeur qu'il mesure à chaque essai. Cette grandeur s'appelle la **réponse**, c'est la grandeur d'intérêt. La valeur de cette grandeur dépend de plusieurs variables. Au lieu du terme "variable" on utilisera le mot **facteur**. La réponse dépend donc de un ou de plusieurs facteurs. Le premier facteur peut être représenté par un axe gradué et

orienté, la valeur donnée à un facteur pour réaliser un essai est appelée **niveau**. Lorsqu'on étudie l'influence d'un facteur, en général, on limite ses variations entre deux bornes. La borne inférieure est le niveau bas. La borne supérieure est le niveau haut.

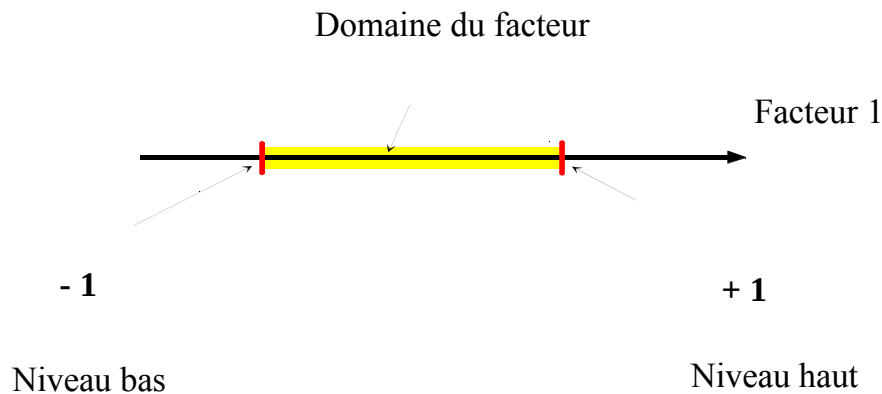


Figure 19 - Notion d'espace expérimental (Jacques Goupy, 1995)

L'ensemble de toutes les valeurs que peut prendre le facteur entre le niveau bas et le niveau haut, s'appelle le domaine de variation du facteur ou plus simplement le domaine du facteur. On a l'habitude de noter le niveau bas par -1 et le niveau haut par $+1$.

S'il y a un second facteur, il est représenté, lui aussi, par un axe gradué et orienté. On définit, comme pour le premier facteur, son niveau haut, son niveau bas et son domaine de variation. Ce second axe est disposé orthogonalement au premier. On obtient ainsi un repère cartésien qui définit un espace euclidien à deux dimensions. Cet espace est appelé l'espace expérimental :

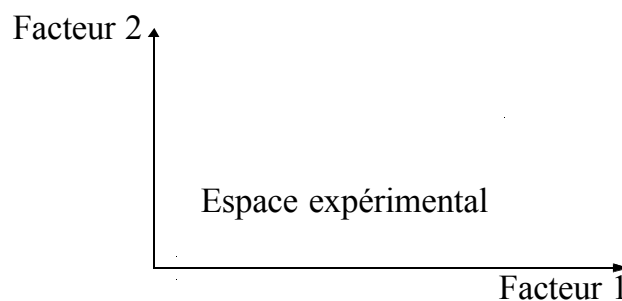


Figure 20 – Représentation de facteur

Le niveau x_1 du facteur 1 et le niveau x_2 du facteur 2 peuvent être considérés

comme les coordonnées d'un point de l'espace expérimental. Une expérience donnée est alors représentée par un point dans ce système d'axes. Un plan d'expériences est représenté par un ensemble de points expérimentaux.

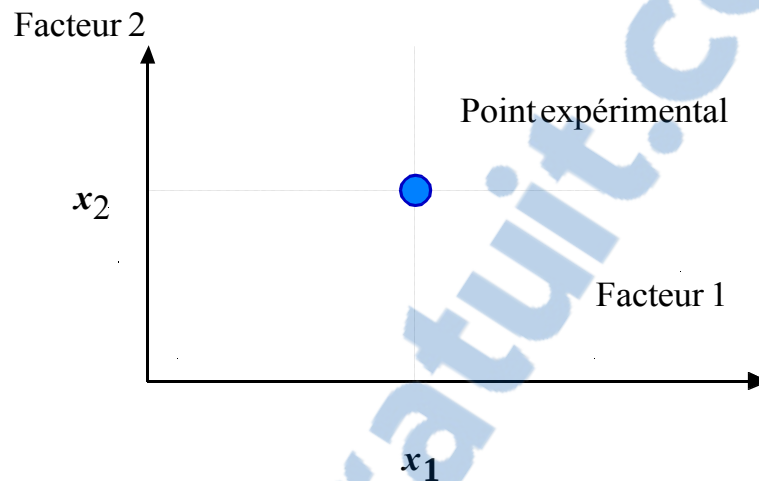


Figure 21 - Définition des points expérimentaux

Le regroupement des domaines des facteurs définit le "domaine d'étude". Ce domaine d'étude est la zone de l'espace expérimental choisie par l'expérimentateur pour faire ses essais. Une étude, c'est-à-dire plusieurs expériences bien définies, est représentée par des points répartis dans le domaine d'étude.

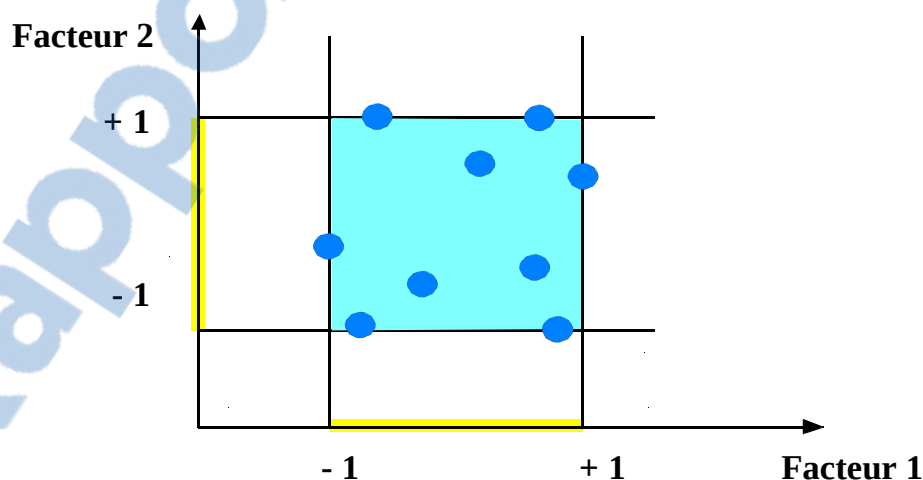


Figure 22 - Domaine d'étude défini par l'expérimentateur

4. Modélisation mathématique

Un phénomène est supposé être caractérisé par des *variables d'intérêt* regroupées dans un vecteur $\mathbf{y}_{réel}$ de dimension 1. Par analogie à un contexte de régression, la variable $\mathbf{y}_{réel}$ sera appelée *réponse*, des *variables d'environnement* le plus souvent mal connues, regroupées dans une quantité $\mathbf{v}_{env}$. Les variables d'environnement sont essentiellement les variables descriptives. Les *variables de contrôle* peuvent être fixées par l'utilisateur pour atteindre divers objectifs ; elles sont regroupées dans la quantité $\mathbf{u}_{cont}$ que l'on suppose, pour simplifier la dimension finie. Elles sont déterminées en phase d'appréciation/développement (Franco, 2008).

Ces variables sont liées selon le schéma suivant (Figure 23), de type boîte noire

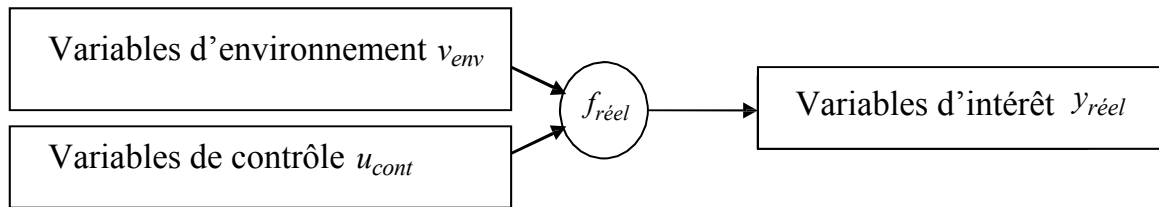


Figure 23 - Schéma représentant les facteurs et les réponses

Sa traduction fonctionnelle en supposant une loi d'évolution est la suivante (Franco, 2008):

$$\mathbf{y} = \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)$$

Cette fonction est générale, il est souvent de coutume d'opter pour un développement limité de Taylor-Mac Laurin (une approximation). La fonction prend la structure d'un polynôme de degré plus ou moins élevé (Jacques Goupy & Lee, 2006):

$$y = \beta_0 + \sum \beta_i x_i + \sum \beta_{ii} x_i^2 + \sum \beta_{ij} x_i x_{ij} \quad (7)$$

où :

- y : la variable dépendante à modéliser (la réponse) ;
- β_0 : constante ;
- $\beta_i, \beta_{ii}, \beta_{ij}$: les coefficients du modèle ;

➤ x_i, x_j : Les variables dépendantes dans le domaine d'expérience.

L'intérêt de modéliser la réponse par un polynôme est de pouvoir calculer ensuite toutes les réponses du domaine d'étude sans être obligé de faire les expériences.

5. Domaine expérimental

Le domaine expérimental est un espace d -dimensionnel défini par une borne inférieure et une borne supérieure pour chaque variable. Les influences des facteurs peuvent être déterminées en imposant à ceux-ci des variations pouvant entraîner des modifications du comportement du phénomène étudié. Un domaine de variation est défini pour chaque facteur à l'intérieur duquel il peut prendre différentes valeurs (ou niveaux). Désormais, le domaine expérimental est ramené, à l'aide d'une transformation linéaire, à l'hyper cube unité $[-1,+1]$.

Une **expérience** (ou point) est défini par un ensemble de conditions opératoires correspondant mathématiquement à un vecteur x de longueur d .

Le **plan d'expériences** (ou plan d'apprentissage) est un objet mathématique qui représente, sous forme codée ou normée, l'ensemble des n expériences à réaliser. En expérimentation numérique, cela correspond aux n vecteurs sur lesquels on évalue le modèle. Un plan d'expériences est représenté par une matrice constituée de n lignes (correspondant aux n expériences) et d colonnes (correspondant aux d facteurs).

La **réponse** est la quantité y obtenue pour un plan spécifié, *i.e.* lorsque l'on fait varier les facteurs étudiés.

La donnée du plan X constitué de n expériences et du vecteur y permet de construire une réponse approchée. On notera alors e le vecteur des résidus, *i.e.* les écarts entre les observations y et les valeurs approchées.

Chaque point expérimental permet d'obtenir une valeur de la réponse. Cette réponse est modélisée par un polynôme dont les coefficients sont les inconnues qu'il faut déterminer. A la fin du plan d'expériences, on a un système de n équations (s'il y a

n essais) à p inconnues (s'il y a p coefficients dans le modèle choisi a priori). Ce système s'écrit d'une manière simple en notation matricielle (Telford, 2007):

$$\mathbf{y} = \mathbf{a} \mathbf{X} + \mathbf{e} \quad (8)$$

Avec :

- $\mathbf{y}$ vecteur des réponses,
- $\mathbf{X}$ matrice de calcul, qui dépend des points expérimentaux,
- $\mathbf{a}$ vecteur des coefficients,
- $\mathbf{e}$ vecteur des résidus.

Il est à noter que ce type de système ne peut pas toujours être résolu du fait que le nombre d'équations est inférieur au nombre d'inconnues. En effet, il y a n équations et $n + p$ inconnues. On peut opérer autrement si l'on utilise une méthode de régression qui est basée en général sur le critère d'optimisation des moindres carrés. On aura ainsi les estimations des coefficients notées : $\hat{\mathbf{a}}$

$$\text{où : } \hat{\mathbf{a}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} \quad (9)$$

Remarque : la matrice $\mathbf{X}'$ est la matrice transposée de $\mathbf{X}$.

Cette opération est assurée par divers logiciels qui réalisent ce calcul et qui fournissent les valeurs des coefficients.

Dans la plus part du temps, les deux matrices qui interviennent dans la théorie des plans d'expérience sont :

- la matrice d'information : $\mathbf{X}'\mathbf{X}$,
- la matrice de dispersion : $(\mathbf{X}'\mathbf{X})^{-1}$

6. Les plans d'expériences

Parmi les plans d'expériences, les plus courants sont les plans factoriels, les plans de Box-Behnken et composites centrés qui sont faciles à générer. Nous

présenterons aussi les plans de Doehlert qui répondent particulièrement bien à la question du remplissage de l'espace. Nous faisons une brève description de ces plans tout en étudiant les différentes propriétés de chacun.

a. Plans factoriels complets à deux niveaux

Ces plans possèdent un nombre de niveaux limité à deux pour chaque facteur. Toutes les combinaisons de niveaux sont effectuées au cours de l'expérimentation. Ces plans peuvent être utilisés indistinctement pour les variables continues et pour les variables discrètes

a. Plan à deux facteurs

Pour deux facteurs, le domaine d'étude est un carré. Le modèle mathématique postulé est un modèle du premier degré. Les meilleurs emplacements des points expérimentaux sont les sommets du domaine d'étude lorsque le modèle postulé est du premier degré.

b. Plans factoriels à K facteurs à 2 niveaux

On peut augmenter le nombre de facteurs. L'espace expérimental possède autant de dimensions qu'il y a de facteurs.

Un plan comportant k facteurs à deux niveaux est noté 2^k .

- Le k en exposant signifie qu'il y a k facteurs étudiés.
- Le 2 indique le nombre de niveaux par facteur.

On remarquera que cette notation indique également le nombre d'essais à réaliser. Ce nombre devient rapidement très important. Pour seulement 7 facteurs, il faudrait exécuter 128 essais. Pour diminuer le nombre des essais en conservant la possibilité d'étudier tous les facteurs, les plans factoriels fractionnaires à deux niveaux ont été introduits.

c. Plans factoriels fractionnaires à deux niveaux 2^{k-Q}

Les plans factoriels fractionnaires sont des plans factoriels qui permettent d'étudier tous les facteurs mais dont le nombre d'essais est réduit par rapport aux plans

factoriels complets. Un plan factoriel fractionnaire à 2 fois moins, ou 4 fois moins d'essais que le factoriel complet correspondant.

A la fin d'un plan factoriel fractionnaire, on a un système de n équations à p coefficients inconnus avec p plus grand que n . On ne sait pas résoudre un tel système. Comme on ne peut pas augmenter le nombre d'équations, il faut diminuer le nombre d'inconnues. On y arrive en utilisant un artifice : on regroupe les coefficients de telle manière qu'il y ait n inconnues. On résout donc un système de n équations à n groupes de coefficients. On appelle ces groupes de coefficients des contrastes ou des alaises et on dit que les coefficients sont alaisées dans les contrastes.

Notation des plans factoriels fractionnaires :

Pour k facteurs prenant deux niveaux le plan complet est noté 2^k .

Le plan fractionnaire, moitié du plan complet possède $1/2 \cdot 2^k$ ou 2^{k-1} essais. On peut donner une signification à chaque caractère de cette notation :

- Le k signifie qu'il y a k facteurs étudiés.
- Le 2 signifie que chaque facteur prend deux niveaux.
- Le 1 signifie que le nombre d'essais du plan a été divisé par 2.

Un plan 2^{k-q} permet d'étudier k facteurs prenant chacun deux niveaux. Le plan complet a été divisé par 2^q .

d. Autres plans à deux niveaux

Les plans factoriels complets et fractionnaires sont basés sur des modèles mathématiques du premier degré. Ils couvrent la plupart des besoins des expérimentateurs lors d'une étude de raffinement. Ce sont eux qui sont employés dans la majorité des cas. D'autres plans à deux niveaux, basés également sur un modèle mathématique du premier degré, ont été mis au point pour répondre à des situations particulières. Nous examinerons les plans de Koshal, les plans de Rechtschaffner, les plans de Plackett et Burman, les Tables de Taguchi et les plans supersaturés.

e. Plans de Koshal

Les plans de Koshal (Koshal, 1933) sont des plans qui permettent de déterminer uniquement les effets principaux des facteurs. On ne peut pas évaluer les interactions.

f. Plans de Rechtschaffner

Les plans de Rechtschaffner (Qu, 2007) sont des plans factoriels fractionnaires simplifiés qui permettent de déterminer les effets des facteurs et les interactions d'ordre deux. Toutes les autres interactions sont supposées nulles avant même l'expérimentation.

g. Plans de Plackett et Burmann

Les matrices de calcul des plans de Plackett et Burman (Vanaja & Shobha Rani, 2007) sont des matrices d'Hadamard, c'est-à-dire des matrices ayant 4, 8, 12, 16, 20, 24, 28, 32 et 36 lignes etc. Elles permettent donc des expérimentations ayant un nombre d'essais intermédiaire de celui des plans factoriels qui eux ont seulement $2k$ lignes (4, 8, 16, 32, etc.).

Les plans de Plackett et Burman sont le plus souvent saturés. Le modèle mathématique est alors un modèle sans interaction analogue à celui des plans de Koshal.

h. Tables de Taguchi

Les Tables de Taguchi (Pillet, 2000) sont des plans de Plackett et Burman dans lesquels on a remplacé +1 par 1 et -1 par 2. Les noms des plans ont également été traduits. Le plan 23 est la Table L8 et le plan 24 est la Table L16. Le plan de 12 essais, appelé Table L12, est un plan de Plackett et Burman. A l'origine, ces plans étaient utilisés avec un modèle sans interaction. Aujourd'hui, certaines personnes leur appliquent les résultats et les principes de la théorie classique.

i. Plans sursaturés

Un plan saturé (Lin, 1995) est un plan qui comporte autant d'essais que de coefficients à déterminer dans le modèle mathématique. Les plans de Rechtschaffner, les plans de Plackett et Burman et les tables de Taguchi sont souvent des plans saturés. Un plan sursaturé est un plan qui comporte moins d'essais que de coefficients à

déterminer dans le modèle mathématique. A ce titre les plans factoriels fractionnaires peuvent être considérés comme sursaturés. Mais, il existe des plans encore plus sursaturés que les plans factoriels fractionnaires (Booth & Cox, 1962). Ce sont des plans dont les facteurs principaux sont aliasés entre eux. Ces plans sont utiles lorsqu'il y a beaucoup de facteurs à examiner et lorsqu'on est sûr que peu d'entre eux sont influents sur la réponse. Certains plans proposent l'étude de 66 facteurs en 12 essais ou de 272 facteurs en 24 essais.

La théorie des alias est applicable à ces plans puisqu'on regroupe les coefficients dans des contrastes. Cependant l'interprétation n'est pas toujours facile.

b) Plans à plusieurs niveaux

Les plans à deux niveaux sont très utilisés parce qu'ils sont économiques en nombre d'essais. Mais il n'y a aucune raison de ne pas considérer des plans ayant des facteurs prenant plus de deux niveaux (Jacques Goupy & Lee, 2006). Il faut donner à chaque facteur le nombre de niveaux nécessaires aux exigences de l'étude. Il existe, là aussi, des plans complets et des plans fractionnaires qui permettent de réduire le nombre des essais malgré l'augmentation du nombre de niveaux.

L'interprétation des résultats dépend de la nature des variables. Si les facteurs sont continus, on pourra utiliser un modèle du premier ou du deuxième degré et exécuter les calculs avec la relation en vue de la modélisation. Pour ces variables, il existe des plans optimisés selon les différents critères d'optimalité. On les trouvera décrits dans les paragraphes consacrés aux plans factoriels et aux plans pour surfaces de réponse. Si les variables sont discrètes, il faut employer l'analyse de la variance pour interpréter les résultats. Pour ces variables, il existe des plans particuliers tels que les carrés latins, les carrés gréco-latins, les plans de Youden et les plans à niveaux mixtes.

a. Plans complets à trois niveaux

S'il y a deux facteurs prenant chacun trois niveaux, il faut exécuter 9 essais. On note ce plan 32.

S'il y a trois facteurs prenant chacun trois niveaux (plan 33), il faut exécuter 27 essais. Ce qui commence à faire beaucoup. C'est la raison pour laquelle on a introduit

les plans fractionnaires correspondants qui portent le nom de carrés latins (Jacques Goupy & Lee, 2006).

b. Carrés latins

Les carrés latins sont des plans pour étudier 3 facteurs prenant chacun 3 niveaux. On réalise 9 essais au lieu de 27 pour le plan complet. Ce sont des plans fractionnaire 33-1. La disposition des points expérimentaux est telle que tous les niveaux sont représentés et qu'il n'y a pas de répétition. Ces plans sont souvent utilisés pour les variables discrètes et le modèle mathématique est souvent un modèle sans interaction (Montgomery, 1995).

c. Carrés gréco-latins

Les carrés gréco-latins sont des plans de 16 essais permettant d'étudier 4 facteurs ayant chacun 4 niveaux. Ce sont des plans 44-2. L'emplacement des points expérimentaux est tel que la matrice de calcul est orthogonale. Ces plans sont utilisés pour les variables discrètes et le modèle mathématique est un modèle avec ou sans interaction d'ordre 2 (Telford, 2007).

d. Carrés de Youden

Youden (Youden, 1951) a développé des plans pour deux variables discrètes prenant plus de quatre niveaux. Le principe est analogue à celui des carrés latins. On réduit le nombre des essais en retirant des points au plan complet. Ces plans sont souvent utilisés pour les variables discrètes. Ils ont servi également de base à l'établissement de plans pour simulations numériques.

e. Plans à niveaux mixtes

Il y a autant de facteurs que l'on veut et chaque facteur prend le nombre de niveaux nécessaires à la bonne exécution de l'étude. Par exemple, un facteur peut prendre trois niveaux, un autre quatre niveaux et un troisième six. Ces plans sont très utilisés pour les variables discrètes.

c) Plans pour surfaces de réponse

Les plans du second degré ou plans pour surfaces de réponse (Khoder, 2011) permettent d'établir des modèles mathématiques du second degré. Ils sont utilisés pour

les variables continues. Ces plans sont utiles à chaque fois que l'on se trouve près d'un maximum ou d'un minimum.

a. Plans composites

Un plan composite est constitué de trois parties :

- 1) Un plan factoriel dont les facteurs prennent deux niveaux.
- 2) Au moins un point expérimental situé au centre du domaine d'étude.
- 3) Des points axiaux. Ces points expérimentaux sont situés sur les axes de chacun des facteurs.

Les plans composites prennent facilement la suite d'un premier plan factoriel dont les résultats sont insuffisamment expliqués par un modèle du premier degré. Il suffit d'effectuer les expériences correspondant aux points en étoile et de faire les calculs sur l'ensemble de toutes les expériences. Les plans composites sont parfaitement adaptés à une acquisition progressive des résultats.

b. Plans de Doehlert

La caractéristique principale des plans de Doehlert (Bezerra, Santelli, Oliveira, Villar, & Escaleira, 2008) est d'avoir une répartition uniforme des points expérimentaux dans l'espace expérimental.. Tous les points sont à la même distance du centre du domaine d'étude et sont situés sur le cercle trigonométrique. Ils forment un hexagone régulier.

c. Plans de Box-Behnken

Les points expérimentaux sont au milieu des arêtes de chacun des côtés du cube. Ce plan comporte douze essais auxquels on peut ajouter un (ou plusieurs) point central.

Les plans de Box-Behnken (Ferreira et al., 2007) répondent à un critère d'optimisation particulier : l'erreur de prévision des réponses est la même pour tous les points d'une sphère (ou une hyper sphère) centrée à l'origine du domaine expérimental. C'est le critère d'isovariance par rotation. Le plus connu des plans de Box-Behnken est celui qui permet d'étudier trois facteurs.

d. Plans hybrides

Les plans hybrides ont été mis au point par Roquemore (Roquemore, 1976). Leur objectif est d'essayer d'approcher deux critères d'optimalité, celui d'orthogonalité et celui d'isovariance par rotation. L'orthogonalité garantit la meilleure précision possible sur les coefficients du modèle et l'isovariance par rotation conduit à des erreurs de prévisions identiques à une même distance du centre du domaine. Si l'expérimentateur recherche ces deux propriétés, il doit penser à utiliser un plan hybride.

Les plans hybrides se désignent de la manière suivante : on indique le nombre de facteurs, puis le nombre de points expérimentaux dont un seul point central, et une lettre pour distinguer deux plans ayant le même nombre de facteurs et le même nombre de points expérimentaux.

e. Plans de Mozzo

Les plans de Mozzo (Mozzo, 1990) présentent deux avantages : Ils sont séquentiels et le nombre de niveaux d'étude est restreint. On peut commencer par étudier deux facteurs en trois essais. Puis, si l'on désire étudier un troisième facteur, il suffit de réaliser trois essais supplémentaires. Douze essais permettent d'étudier quatre facteurs.

Les plans de Mozzo ne permettent pas tous d'établir un modèle du second degré. Il faut alors avoir recours aux plans quadratiques gigognes de Mozzo.

f. Plans de Rechtschaffner pour le second degré

Les plans de Rechtschaffner (Qu, 2007) permettant d'établir un modèle du second degré sont des plans saturés. S'il y a k facteurs, il faut effectuer un nombre n d'essais.

g. Plans de D-optimaux

Les contraintes expérimentales ne permettent pas toujours d'être dans les conditions idéales des plans d'expériences précédemment décrits. Par exemple, les réglages de l'appareil ne permettent pas d'atteindre les niveaux préconisés par la théorie ou des combinaisons de niveaux peuvent se révéler dangereuses : réaction

explosive pour les chimistes, concentration toxique pour les médecins, etc. Dans cette situation, il est extrêmement commode d'utiliser les plans D-optimaux (de Aguiar, Bourguignon, Khots, Massart, & Phan-Than-Luu, 1995). Le choix de l'emplacement des points expérimentaux nécessite alors un logiciel de plans d'expériences. Il suffit de préciser le nombre d'expériences que l'on désire effectuer et le modèle a priori. Le logiciel choisi alors, grâce à un algorithme d'échange, le plan le mieux adapté à l'étude.

h. Plans non conventionnels et leur éventuelle réparation

Il peut arriver que l'on soit en possession d'une série de résultats expérimentaux qui n'ont pas été obtenus selon un plan d'expériences. Dans ce cas, il a été montré (Jacques Goupy, 1996) que l'on peut utiliser ces résultats moyennant certaines précautions. Si la position des essais ne s'éloigne pas trop des plans classiques, les erreurs sur les coefficients du modèle sont faibles et les réponses prédites sont, dans la plupart des cas, tout à fait acceptables.

d) Plans de mélanges

Les facteurs d'étude des plans de mélanges (Cornell, 2011) sont les proportions des constituants du mélange. Or, ces constituants ne sont pas indépendants les uns des autres. La somme des proportions d'un mélange est toujours égale à 100%. Le pourcentage du dernier constituant est imposé par la somme des pourcentages des premiers composés. C'est la raison pour laquelle les plans de mélanges sont traités à part.

Les plans de mélanges sont aussi caractérisés par de nombreuses contraintes qui peuvent peser sur le choix des proportions des constituants (Dagnelie, 2012). Par exemple, la concentration d'un produit doit être au moins de x pour-cent ou cette concentration ne peut excéder une valeur donnée. En fonction de ces contraintes la planification de l'étude est modifiée et elle doit être adaptée à chaque cas.

e) Plans Booléens

Il n'est pas toujours possible d'exploiter les résultats d'un plan d'expériences avec les outils classiques : régression multilinéaire pour les variables continues ou analyse de la variance pour les variables discrètes. Si les facteurs d'étude sont des

variables booléennes, c'est-à-dire ne prenant que deux valeurs, il faut trouver la fonction booléenne, ne prenant elle-même que deux valeurs, en fonction des valeurs des variables.

Les plans booléens (J Goupy, 2000) ressemblent aux plans factoriels à deux niveaux mais l'interprétation mathématique est complètement différente puisque la nature des variables est différente. Chaque essai permet de connaître la valeur d'un minterm (ou d'un maxterm). L'ensemble de tous les minterms donne la forme canonique de la fonction cherchée. La forme canonique est ensuite simplifiée grâce à l'une des méthodes classiques de simplification des fonctions booléennes (Méthode de Veitch, Harvard ou Quine)(Umezawa, 1967).

f) Plans pour simulations numériques

Lorsque les expérimentations sont coûteuses, il arrive de plus en plus fréquemment que l'on étudie d'abord les phénomènes à l'aide de simulation numérique (Gunasegaram, Farnsworth, & Nguyen, 2009). Les calculs sont souvent complexes et nécessitent des temps de calcul très longs. Il est alors avantageux de faire des simulations de la même manière que les essais des plans d'expériences. Cependant, deux particularités des calculs numériques doivent être prises en compte : la première est l'absence d'erreur expérimentale, et la seconde est la complexité des modèles à priori permettant l'interprétation des résultats.

7. Conclusion

Ce chapitre a porté sur un aspect fondamental de la technique de plan d'expériences "*Design of Experiments*" (DoE) : la modélisation. Cela comprend les principales techniques mathématiques utilisées pour exprimer les variations des réponses en fonction des valeurs des facteurs.

La présentation des principaux plans d'expériences : respectivement les plans de Box-Behnken, composites centrés, des plans de Doehlert,...etc. est réalisée. Elle englobe une brève description des plans, suivie d'une étude des différentes propriétés de chacun.

On a pu voir que l'utilisation des notations matricielles dans la modélisation, repose sur le fait d'une adaptation afin de gérer les modèles polynomiaux d'ordre supérieur à 1. Le chapitre suivant discute l'adaptation de ces concepts dans la gestion de l'évolution des logiciels.

V. Expérimentation et Résultats

1) Introduction

Les approches empiriques visent à utiliser les avantages de l'application de méthodes et techniques d'autres domaines au domaine du génie logiciel en illustrant comment malgré les difficultés de l'adaptation l'évolution du logiciel peut être étudiée empiriquement. Ces approches portent principalement sur l'utilisation de l'estimation de l'impact des modifications comme principal indicateur de l'estimation de l'effort de maintenance en exploitant des métriques de logiciel. Nous étudions plus spécifiquement les systèmes orientés objet et appliquons la technique de plan d'expériences "*Design of Experiments*" (DoE), généralement utilisée dans les sciences expérimentales, afin de quantifier l'effort de maintenance qui constituerait une mesure de l'impact du changement, sous la forme d'une équation dont les paramètres seraient les valeurs de ces métriques (Miloud Dahane, Abdi, Bouneffa, Ahmad, & Basson, 2019).

2) Métriques de couplage

Les métriques logicielles peuvent fournir une grande variété d'informations concernant une application logicielle. Les mesures de différents aspects du logiciel peuvent aider à en assurer le succès. Dans le contexte des travaux en cours, nous préférons utiliser des métriques de couplage (Almugrin et al., 2016) pour estimer l'impact du changement dans une application logicielle. Nous choisissons bien les métriques de couplage car elles peuvent être extraites directement du code source. Cela peut fournir des mesures significatives pour les attributs liés à l'analyse d'impact du changement (Elmidaoui, Cheikhi, & Idri, 2016; Nuez-Varela, Prez-Gonzalez, Martinez-Perez, & Soubervielle-Montalvo, 2017).

Le couplage représente la mesure du degré de dépendance entre deux artefacts logiciels (Tempero & Ralph, 2018). Nous considérons cette mesure entre les classes au niveau du code source du logiciel. Si le degré de couplage d'une classe C est supérieur à une autre classe C' , la classe C est plus sensible aux changements qui affectent C' ;

par conséquent, on peut en déduire que les classes les plus étroitement couplées peuvent avoir une plus grande capacité à gérer l'impact du changement entre elles. Cet attribut peut qualifier la dépendance entre les classes et les rendre plus significatives pour être modifiées ou déboguées. Il est donc important de maintenir soigneusement ces classes.

Dans ce qui suit, nous rappelons certaines facettes pour calculer les métriques de couplage entre les classes, afin de mieux comprendre leur rôle lors de la maintenance du logiciel.

a) Coupling Between Objects (CBO)

Couplage entre objets (*CBO*) signifie la mesure (Tempero & Ralph, 2018) entre deux classes A et B telles que A et B soient couplées si les méthodes de la classe A utilisent les méthodes ou les attributs de la classe B :

$$CBO(c) = |\{d \in C - \{c\} : uses(c, d) \vee uses(d, c)\}| \quad (10)$$

Où c est une classe et C est l'ensemble de toutes les classes.

b) Response For a Class (RFC)

La réponse pour une classe (RFC) correspond à la somme du nombre de méthodes et du nombre de méthodes invoquées dans une classe donnée (Tempero & Ralph, 2018). Cette mesure reflète le potentiel de communication entre les classes. Cela peut indiquer que plus le RFC d'une classe est élevé, plus il est significatif pour les modifications, d'où des efforts supplémentaires en termes de testabilité et de débogage.

$$RFC(c) = Nbre(M(c)) + Nbre(Inv(M'(c))) \quad (11)$$

Où $Nbre(M(c))$ représente le nombre de méthodes d'une classe c et $Nbre(Inv(M'(c)))$ représente le nombre de méthodes invoquées d'une classe c .

c) Message Passing Coupling (MPC)

La métrique MPC (Message Passing Coupling) calcule le nombre d'appels de méthode dans une classe (Tempero & Ralph, 2018).

$$MPC(c) = \sum_{m \in M(c)} \sum_{m' \in M(c)} NSI(m, m') \quad (12)$$

Où $NSI(m, m')$ est le nombre d'appels statiques de m' par m (nombre d'appels de méthodes d'une classe c où m' est appelé par un objet de type statique de la classe d et m' appartient à $M(d)$).

3) Méthodologie de recherche

La technique de plan d'expériences "*Design of Experiments*" (DoE) (Heck & Thomas, 2015) est une méthode de conception expérimentale largement utilisée dans les statistiques appliquées. La DoE permet de comprendre et de modéliser des processus complexes qui dépendent d'un grand nombre de variables difficiles à ajuster de manière intuitive. Elle est utilisée depuis plus de deux décennies dans des domaines tels que la biologie, la chimie ou la physique, où le contrôle des hypothèses est généralement effectué par l'expérience. Le principe du DoE est la recherche d'un lien entre une valeur d'intérêt ou une réponse et des variables d'entrée. Sa mise en œuvre repose sur la réalisation d'une série d'expériences permettant chacune d'acquérir de nouvelles connaissances en faisant varier une à plusieurs variables d'entrée afin d'obtenir des résultats permettant de construire un modèle avec un investissement minimal (Dagnelie, 2012).

Nous étudions l'utilisation de cette méthode pour construire un modèle mathématique permettant, sur la base de la variation des valeurs des métriques, de prédire le volume des modifications du code source. En fait, pour chaque scénario de maintenance, nous pouvons connaître les valeurs des métriques qui seront influencées par ces changements. Pour cette étude, nous avons assimilé une expérience à un code source dans lequel les facteurs d'entrée de cette expérience sont un ensemble de métriques de couplage (*CBO*, *RFC* et *MPC*) et la sortie est le pourcentage du code source à modifier. La force de la méthode DoE réside dans la possibilité de créer le modèle mathématique sur la base d'un nombre réduit d'expériences (Jacques Goupy & Lee, 2006). Pour cette raison, nous traitons avec un nombre réduit de versions (12 versions). Cela montre l'utilité de cette approche afin d'éviter des investissements inutiles.

4) Contexte de l'étude

Nous avons modélisé l'interaction d'un ensemble de métriques, en occurrence *CBO*, *RFC* et *MPC*, afin d'évaluer leur contribution globale à l'analyse de l'impact du changement dans un ensemble de classes donné. Une expérience bien effectuée peut répondre aux questions suivantes dans ce contexte:

- Q1. Quelle est l'influence des différentes métriques logicielles dans l'estimation de l'impact du changement ?
- Q2. Parmi les indicateurs les plus significatifs permettant d'estimer l'impact du changement, lequel a le plus de poids ?
- Q3. Peut-on prévoir l'impact du changement en faisant varier ces métriques logicielles ?

La technique de plan d'expériences "*Design of Experiments*" (DoE) adapte les données de réponse aux modèles mathématiques afin de trouver la meilleure réponse à une question posée. Elle peut prédire la réponse optimale à une combinaison de valeurs donnée. Une approche répétitive peut alors aider à acquérir des connaissances plus fines pour optimiser les réponses critiques et trouver la combinaison de valeurs la plus adaptée.

Nous commençons par poser une ou plusieurs questions, puis nous appliquons de manière itérative les différentes variations de *CBO*, *RFC* et *MPC* sur l'impact du changement. Par la suite, cette relation de cause à effet permet d'affiner les données pour tenir compte de l'influence de la variation des métriques logicielles sur l'impact du changement.

Les trois étapes critiques du processus d'acquisition de connaissances sont effectuées de manière itérative (voir Figure 24). Celles-ci sont discutées comme suit :

1. La technique de plan d'expériences "*Design of Experiments*" (DoE) assure une efficacité maximale avec un minimum d'expériences. Le choix de la méthode d'expérimentation facilite l'interprétation progressive des résultats.

2. L'analyse des résultats des expériences est facilitée par le choix initial des expériences. Les outils logiciels de la DoE facilitent la construction des plans d'expériences et leurs calculs. Ils peuvent également appuyer l'analyse des résultats à l'aide des représentations graphiques et améliorer la compréhension des phénomènes.
3. L'acquisition progressive des connaissances est composée de séries d'expériences. Une première série d'expériences conduit à des conclusions intermédiaires; selon ces conclusions intermédiaires, nous continuons avec une nouvelle série d'essais. L'ensemble de séries d'expériences peut ensuite être utilisé pour obtenir un aperçu précis des résultats de l'étude. L'analyste peut accumuler les résultats pertinents et continuer jusqu'à la satisfaction de besoins.

Nous rappelons que les trois étapes ci-dessus du processus d'acquisition de connaissance sont détaillées dans la section 5 du paragraphe précédent.

5) Modélisation d'équations mathématiques

La technique de plan d'expériences "*Design of Experiments*" (DoE) fonctionne sur la base d'une modélisation mathématique des facteurs participants, afin d'illustrer leur interaction pour obtenir un résultat optimal, comme indiqué dans l'équation ci-après.

$$y = f(x_1, \dots, x_n) \quad (13)$$

Nous formalisons la fonction mathématique qui peut indiquer la réponse à l'interaction des facteurs impliqués. Dans ce cas, les facteurs sont les différentes métriques logicielles et la réponse est l'estimation de l'impact du changement. Par exemple, elle peut être formulée comme suit:

$$EIC = f(CBO, RFC, MPC) \quad (14)$$

Où *EIC* indique l'Estimation de l'Impact du Changement. Il en résulte une fonction d'interaction entre *CBO*, *RFC* et *MPC*.

La technique de plan d'expériences DoE prend en charge les calculs de facteurs en interaction. L'équation ci-dessous fournit l'estimation générale de l'impact du changement. Les formules DoE peuvent être résolues à l'aide de divers outils logiciels tels que STATISTICA, R, Minitab, etc. Nous avons utilisé STATISTICA au cours de cette étude pour calculer les valeurs des constantes et des coefficients de l'équation suivante. Il est souvent courant de choisir un développement limité de Taylor-Mac Laurin (une approximation) (Bulut, 2014). Cela conduit à transformer le modèle en une structure polynomiale de degré plus ou moins grand, pour obtenir des connaissances plus précises. Il peut être formulé comme suit:

$$y = \beta_0 + \sum \beta_i x_i + \sum \beta_{ii} x_i^2 + \sum \beta_{ij} x_i x_j \quad (15)$$

Où y est la variable dépendante du modèle (la réponse), β_0 est une constante; $\beta_i, \beta_{ii}, \beta_{ij}$ sont les coefficients du modèle, alors que les x_i, x_j sont les variables dépendantes du champ de l'expérience.

Les coefficients correspondent au degré d'influence de chaque facteur sur la variable dépendante à modéliser. Si nous prenons l'exemple précédent, chaque variable du domaine d'expérience (*CBO*, *RFC* ou *MPC*) a un certain degré d'influence sur l'estimation de l'impact du changement.

6) Validation empirique

Au cours de cette étude, l'objectif principal est de déterminer l'influence de la variation de certains paramètres de logiciel sur l'analyse d'impact de changement dans différentes versions de logiciel. Nous avons donc expérimenté cette approche sur les différentes versions de la bibliothèque open-source "*JFreeChart*", écrite en Java. Cette bibliothèque est principalement utilisée pour générer des graphiques dans les applications Java. Elle est composée de 331 classes et seulement 12 versions nous fournissent une base de test stable.

La technique de plan d'expériences DoE est utilisée comme pratiquement, au cours de cette étude, pour modéliser l'estimation de l'impact du changement pour les différentes valeurs de l'ensemble des métriques du logiciel. Cette méthode présente un

avantage particulier qui est le fait qu'elle nécessite un nombre minimum d'expériences (les différentes versions de logiciels, dans le contexte de notre étude) pour construire un modèle optimal (Tinsson, 2010).

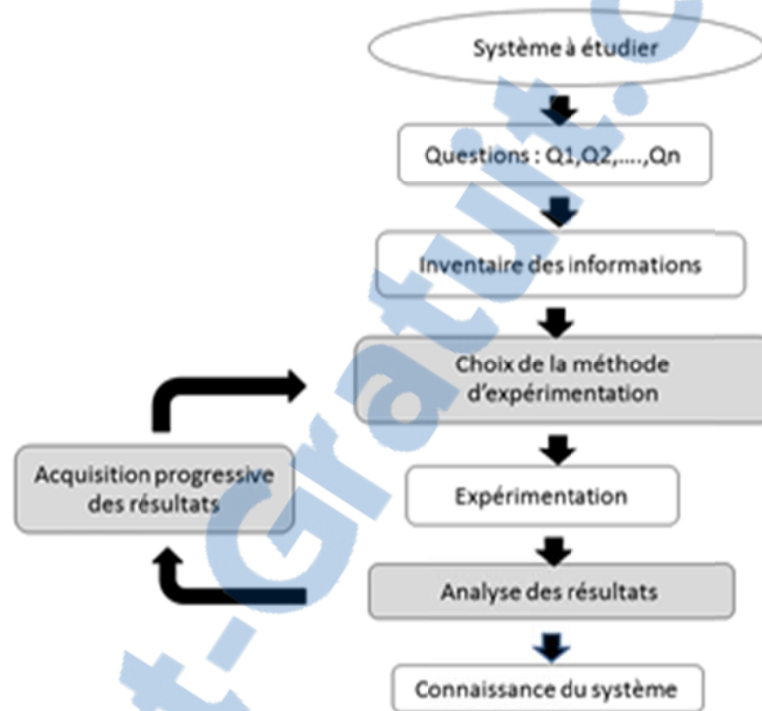


Figure 24 - Plan d'expérimentation

7) Plan d'expériences

Pour cette étude, nous collectons des données à partir du référentiel "*JFreeChart*" implémenté en java. Nous avons traité les données extraites de douze versions de "*JFreeChart*", comme indiqué dans le Tableau 1. Comme requis par la méthode DoE, nous mettons en œuvre un processus consistant d'abord à mesurer les métriques logicielles, puis mesurer le taux de changement de code en considérant les douze versions de "*JFreeChart*". Les différentes valeurs des métriques *CBO*, *RFC* et *MPC* peuvent être extraites directement du code. Nous utilisons le code source disponible des versions de "*JFreeChart*" pour mesurer le volume de modification logicielle. Celles-ci sont mesurées en comparant une version donnée à la suivante, obtenue après la mise à jour du logiciel (Silva, Borel, Lopes, & Murta, 2014).

8) Outil prototype

Les valeurs des différentes métriques logicielles sont extraites de "*JFreeChart*" à l'aide d'un outil prototype que nous avons développé en Java et nommé "SACI" (Système d'Aide de Calcul des Impacts). L'objectif principal de cet outil est de prendre en entrée un code source et de calculer les valeurs des métriques considérées.

Nous calculons les valeurs des métriques de couplage à l'aide de cet outil pour les versions considérées de "*JFreeChart*". La Figure 25 montre un extrait des résultats obtenus.

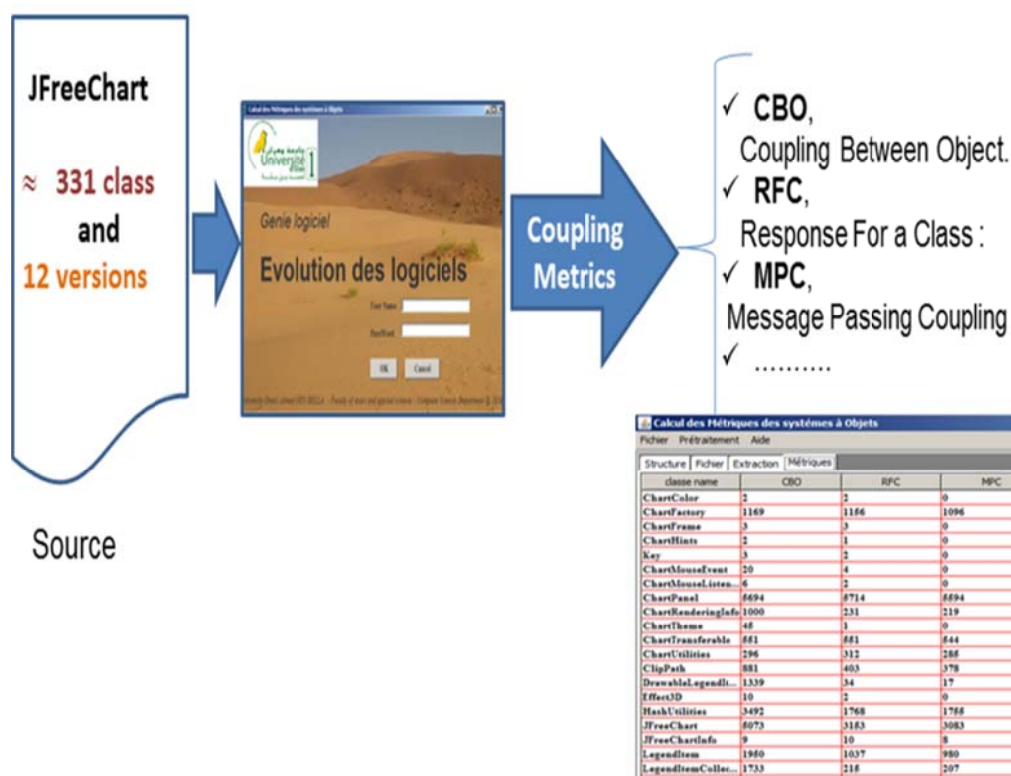


Figure 25 - Interface de l'outil de calcul de métriques "SACI"

Calcul des Métriques des systèmes à Objets			
Fichier Prétraitement Aide			
Structure	Fichier	Extraction	Métriques
classe name	CBO	RFC	MPC
ChartColor	2	2	0
ChartFactory	1169	1156	1096
ChartFrame	3	3	0
ChartHints	2	1	0
Key	3	2	0
ChartMouseEvent	20	4	0
ChartMouseListener...	6	2	0
ChartPanel	5694	5714	5594
ChartRenderingInfo	1000	231	219
ChartTheme	45	1	0
ChartTransferable	551	551	544
ChartUtilities	296	312	285
ClipPath	881	403	378
DrawableLegendIt...	1339	34	17
Effect3D	10	2	0

Figure 26 - Extrait de la table des métriques de couplage de la version1 de "JFreeChart"

9) Ajustement des paramètres pour la DoE

La particularité de métrique de couplage est qu'elle concerne chaque classe du logiciel (Almugrin et al., 2016). Nous devons définir des métriques de couplage représentatives afin d'étudier l'ensemble du logiciel. Nous proposons donc une technique particulière pour garantir un choix valable de la méthode d'expérimentation. La technique proposée permet de déterminer le meilleur représentant pour chaque type de métriques dans l'ensemble des classes.

Considérons un système noté S composé de n classes. Ce système peut être représenté comme suit:

$$S = C_1, \dots, C_i, \dots, C_n \quad (16)$$

Où la mesure d'une classe donnée C_i peut être évaluée par un ensemble de métriques, présentées comme suit:

$$C_1 = m_1^{c1}, \dots, m_i^{c1}, \dots, m_p^{c1} \quad (17)$$

Où, m_p est une métrique particulière appliquée à une classe C .

Nous proposons une fonction $F(S)$ qui recherche la représentation optimale des résultats à partir de tout un ensemble de métriques; il peut être formulé comme suit:

$$F(S) = m_1^S, \dots, m_i^S, \dots, m_p^S \quad (18)$$

Où, la mesure de la métrique m_i^S est la métrique choisie (choix de la méthode d'expérimentation) comme l'exige la technique de plan d'expériences DoE :

$$m_1^S = F(m_1^{c1}, \dots, m_1^{ci}, \dots, m_1^{cn}) \quad (19)$$

Nous décrivons ci-dessous l'importance du choix du meilleur paramètre (fonction F) pour chaque ensemble de métriques:

- Une valeur plus élevée de la métrique CBO, de par sa nature, représente l'effort requis pour le maintien d'une classe donnée (Tiwari & Rathore, 2018). Par conséquent, la représentation prédominante de la fonction F pour la métrique CBO est le maximum de toutes les CBO des classes considérées. L'opérateur d'agrégation MAX est choisi car il représente la pire situation en termes d'impact de changement. L'augmentation de la valeur de CBO signifie un fort impact du changement.
- La métrique RFC reflète le degré de communication entre les classes dans un système logiciel donné (Tiwari & Rathore, 2018). Le cas le plus néfaste est de modifier la classe avec la valeur maximale des liens de communication, car cela entraînerait des modifications dans toutes les classes liées à cette classe. Par conséquent, l'agrégation MAX qui correspond à la valeur la plus élevée de la métrique RFC pour un système donné est le meilleur représentant de ce type de métrique.

- La métrique MPC mesure la communication entre les objets de la classe (Tiwari & Rathore, 2018). Dans le cas où une classe a une méthode invoquée par un grand nombre d'objets, une modification de cette méthode impliquera la modification d'un nombre maximal d'objets. Par conséquent, l'agrégation MAX, qui correspond au nombre le plus élevé d'invocations de méthodes, représente la dépendance la plus élevée entre une classe donnée et les autres classes. C'est donc le meilleur représentant de ces métriques dans le système.

Sur la base du raisonnement donné ci-dessus, nous formulons le choix suivant de la méthode d'expérimentation :

$$CBO^S = \text{Max}(CBO^{c_1}, \dots, CBO^{c_i}, \dots, CBO^{c_n}) \quad (20)$$

$$RFC^S = \text{Max}(RFC^{c_1}, \dots, RFC^{c_i}, \dots, RFC^{c_n}) \quad (21)$$

$$MPC^S = \text{Max}(MPC^{c_1}, \dots, MPC^{c_i}, \dots, MPC^{c_n}) \quad (22)$$

10) Application des paramètres DoE ajustés

Le choix de la méthode d'expérimentation permet le choix d'une valeur unique pour chaque ensemble de représentations de métriques. Nous appliquons cette modélisation, pour chaque représentation de la métrique de couplage pour toutes les classes de la version 1 de "*JFreeChart*", comme suit:

$$\begin{aligned} CBO^{JFreeChart_version1} = \\ \text{Max}(CBO^{\text{Chartcolor}}, \dots, CBO^{\text{Chartrenderinginfo}}, \dots, CBO^{\text{Xycoordinatetype}}) \end{aligned} \quad (23)$$

$$\begin{aligned} RFC^{JFreeChart_version1} = \\ \text{Max}(RFC^{\text{Chartcolor}}, \dots, RFC^{\text{Chartrenderinginfo}}, \dots, RFC^{\text{Xycoordinatetype}}) \end{aligned} \quad (24)$$

$$\begin{aligned} MPC^{JFreeChart_version1} = \\ \text{Max}(MPC^{\text{Chartcolor}}, \dots, MPC^{\text{Chartrenderinginfo}}, \dots, MPC^{\text{Xycoordinatetype}}) \end{aligned} \quad (25)$$

Nous appliquons ces mesures de manière itérative sur les 12 versions de "*JFreeChart*" afin de développer le protocole expérimental, comme indiqué dans le Tableau 1.

Le tableau des "Réponses observées" donne le ratio des lignes modifiées pour une version donnée. Les entrées du tableau "Réponses observées" (Tableau 2) sont obtenues à l'aide de l'outil *WinMerge*, un open source qui compare deux versions successives du code source du logiciel, pour calculer les lignes modifiées du code. La comparaison prend en compte uniquement les lignes exécutables du code source, c'est-à-dire qu'elle exclut les commentaires, les espaces et les lignes vides. Il compare la signature de méthodes de deux versions différentes pour calculer la distance de *Levenshtein* (Sudhish, Jain, & Cao, 2016). La distance de *Levenstein* entre deux mots w_1 et w_2 est le nombre minimal d'opérations (insertions, suppressions ou substitutions) nécessaires pour transformer w_1 en w_2 .

	Version 1	Version 2	Version 3	Version 4	Version 5	Version 6	Version 7	Version 8	Version 9	Version 10	Version 11	Version 12
CBO	8348	8343	8300	7431	8348	8100	6278	8254	5998	8331	8312	8314
RFC	6602	6119	6502	6416	6341	5856	6245	6112	4990	6467	5876	5997
MPC	6372	6371	6245	6115	6243	5436	6224	6554	6345	5994	5973	6876

Tableau 1 - Protocole expérimental

	Version 1	Version 2	Version 3	Version 4	Version 5	Version 6	Version 7	Version 8	Version 9	Version 10	Version 11	Version 12
% Change Impact	0%	1%	3%	8%	5%	20%	7%	4%	30%	7%	6%	20%

Tableau 2 - Réponses observées

11) Exécution de la méthode DoE

Conformément à la méthode DOE, nous disposons d'un ensemble d'expériences, à savoir les valeurs de *CBO*, *RFC* et *MPC* pour chaque version de la librairie "*JFreeChart*" et l'impact estimé du changement, exprimé sous forme de ratio.

Nous calculons les valeurs des coefficients du modèle mathématique de l'équation suivante :

$$y = \beta_0 + \sum \beta_i x_i + \sum \beta_{ii} x_i^2 + \sum \beta_{ij} x_i x_{ij} \quad (26)$$

Nous obtenons les valeurs des coefficients après utilisation du logiciel STATISTICA version 7 :

β_i	β_0	β_1	β_{11}	β_2	β_{22}	β_3	β_{33}	β_{12}	β_{13}	β_{23}
value	0,0704	-0,1847	0,2537	-0,1713	0,3215	-0,0422	0,0298	-0,5330	-0,0897	0,3008
P-value	0,4140	0,4727	0,5618	0,9151	0,1747	0,8785	0,9201	0,2405	0,7900	0,8722

Tableau 3 - Coefficients β_i du modèle d'impact de changement (facteurs *CBO*, *RFC* et *MPC*)

Selon les résultats du Tableau 3, les valeurs p (p-value) des coefficients β_i sont supérieures à la valeur recommandée (qui doit être inférieure à 0,05), cela est dû à la nature du plan d'expériences que nous avons utilisé, qui est en fait un modèle expérimental non conventionnel. Selon (Jacques Goupy & Lee, 2006), l'utilisation d'un tel modèle engendre généralement des erreurs expérimentales. La non signification des coefficients a rendu impossible la construction du modèle mathématique à trois facteurs. Afin de minimiser l'influence des erreurs expérimentales générées par ce type de modèle d'expériences, il est recommandé de minimiser le nombre de facteurs (Jacques Goupy, 1996). Dans le cadre de cette étude, l'alternative consistait à combiner les facteurs deux à deux, toutes les combinaisons ayant été testées, seule la combinaison des facteurs *CBO*, *MPC* donnant des résultats intéressants.

En exploitant les résultats des 12 versions (expériences) comme indiqué dans les Tableaux 1 et 2, c'est-à-dire les exigences de la technique de plan d'expériences DoE, nous obtenons un plan à deux facteurs et nous appliquons l'équation mathématique suivante :

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{11} x_1^2 + \beta_{22} x_2^2 + \beta_{12} x_1 x_2 \quad (27)$$

où y indique l'impact du changement et x_i, x_j sont les variables dépendantes dans le domaine de l'expérience (métriques de couplage à l'échelle de modèle).

Les différents coefficients du modèle β_i sont obtenus par des calculs matriciels, qui figurent dans le Tableau 4 suivant :

β_i	β_0	β_1	β_2	β_{11}	β_{22}	β_{12}
Value	0,08864	1,50561	0,16427	0,44738	-0,29321	-1,70957
p-value	0,022994 *	0,004576 *	0,095781	0,000149 *	0,051806	0,003872 *

* Valeurs statistiquement significatives (inférieures à $p = 0,05$)

Tableau 4 - Coefficients β_i du modèle d'impact de changement (facteurs CBO,MPC)

Nous obtenons les résultats suivants après intégration des valeurs de l'équation précédente :

$$y = 0,0886 + 1,5056x_1 + 0,16427x_2 + 0,4473x_1^2 - 0,29321x_2^2 - 1,7095x_1x_2 \quad (28)$$

L'équation suivante donne les valeurs des coefficients à effet significatif (et ne prend en considération les valeurs non significatives):

$$y = 0,0886 + 1,5056x_1 + 0,4473x_1^2 - 1,7095x_1x_2 \quad (29)$$

Où x_1 correspond à *MPC* à l'échelle générique de modèle et x_2 correspond à *CBO* à l'échelle du modèle générique.

Afin d'avoir le modèle final, nous réalisons une transformation de l'échelle générique à l'échelle finale, cette transformation est obtenue par l'équation suivante (Jacques Goupy & Lee, 2006) :

$$x = \frac{A-A_0}{pas} \quad (30)$$

Où A_0 est la valeur centrale des unités d'origine, A est la variable d'origine et x est la variable codée.

Le modèle d'Estimation d'Impact de Changement (*EIC*) est alors le suivant:

$$\begin{aligned} \mathbf{EIC} = & 0,0886 + 1,5056 \left(\frac{\mathbf{MPC} - 6165}{720} \right) + 0,4473 \left(\frac{\mathbf{MPC} - 6165}{720} \right)^2 \\ & - 1,7095 \left(\frac{\mathbf{MPC} - 6165}{720} \right) \left(\frac{\mathbf{CBO} - 7173}{1175} \right) \end{aligned} \quad (31)$$

Nous améliorons encore le modèle obtenu en effectuant une série de tests, comme indiqué dans ce qui suit.

12) Test de robustesse du modèle

Nous étudions les valeurs du modèle aux points de conception expérimentale en utilisant des techniques telles que la randomisation et la réitération (Watson & Petrie, 2010). En effet, ces plans tendent à positionner les points aux limites du domaine pour prendre en compte la variation aléatoire et offrent une tendance plus fiable vers les réponses en présence d'erreurs de mesure. Pour ce faire, il existe des critères permettant de tester la bonne correspondance entre les observations et les prévisions sur les données utilisées lors de la construction du modèle. C'est le rôle du coefficient de corrélation R^2 . Le modèle est considéré comme robuste lorsque la valeur de ce coefficient tend à 1 (Watson & Petrie, 2010).

Dans ce cas d'expérimentation, on obtient un coefficient de corrélation $R^2 = 0,94$. Cela peut valider l'exactitude du modèle proposé pour estimer l'impact du changement car il répond aux critères de robustesse, comme le montre la Figure 27.

13) Analyse de données

L'estimation de l'impact du changement doit être calculée sur la base de la combinaison des métriques *CBO* et *MPC*. Cela peut aider le décideur à estimer concrètement le pourcentage de code source affecté par chaque scénario proposé, y compris les métriques.

Le modèle d'estimation de l'impact des changements montre une influence de l'effet quadratique du *MPC*.

La technique de plan d'expériences DoE nous offre la possibilité de déterminer les facteurs les plus influents dans l'estimation de l'impact du changement. Cela est possible au moyen de la technique appelée SRM (Surface Response Methodology) (Khoder, 2011).

En examinant la surface de réponse, nous trouvons l'influence du MPC sur l'estimation de l'impact du changement. Ceci est cohérent avec les résultats déduits par le modèle et exprime l'effet quadratique du MPC, qui est en fait cette concentration de couleur dans la région MPC.

Les mêmes affirmations ont déjà été faites par des travaux antérieurs (M Ajmal Chaumun et al., 2002), (Abdi & Dinedane, 2015), (Abdeen et al., 2015). Tous ces travaux ont montré l'importance du couplage en tant que mesure précise pour l'estimation de l'impact du changement.

L'approche proposée souligne le fait qu'elle assure la construction du modèle sur la base d'un nombre réduit d'expériences (versions). Enfin, nous obtenons des résultats similaires, voire meilleurs, que ceux obtenus dans la littérature connexe avec moins d'effort en termes de volume de données à analyser.

Une autre représentation utilisant le principe de l'isoréponse pour détecter visuellement les facteurs les plus influents correspondants à des concentrations de couleurs (Figure 28).

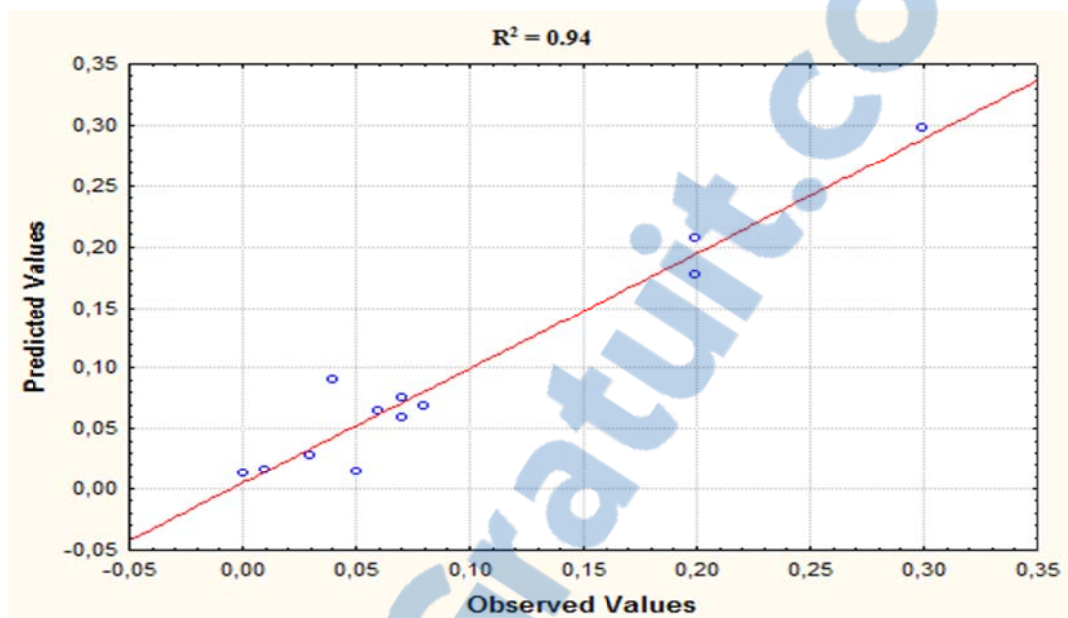


Figure 27 - Représentation graphique du test de robustesse du modèle proposé

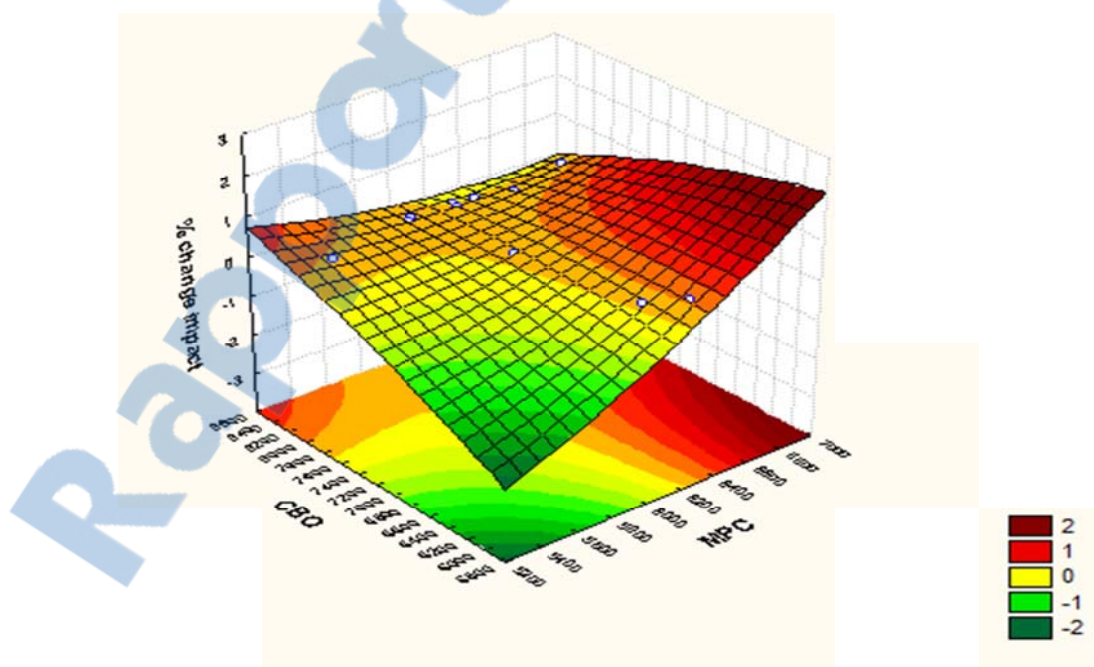


Figure 28 - Représentation tridimensionnelle de l'influence du MPC et du CBO sur l'EIC

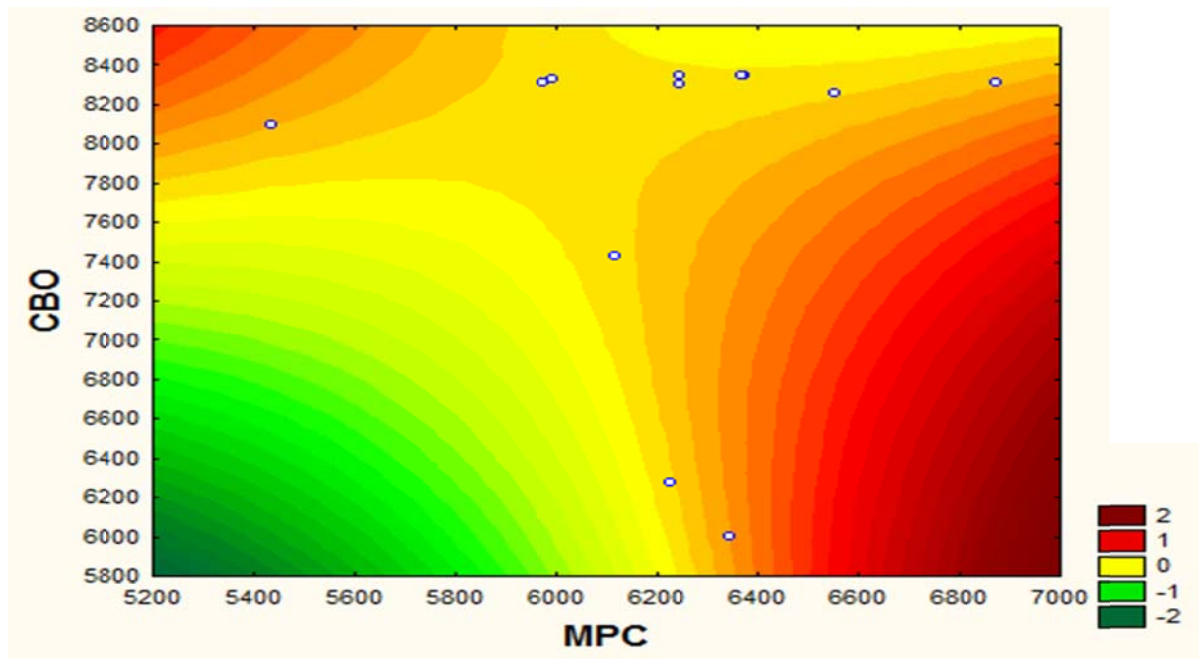


Figure 29 - Représentation isométrique de l'influence du MPC et du CBO sur l'EIC

14) Conclusion

Nous avons analysé l'impact du changement dans les systèmes orientés objet à l'aide d'outils statistiques. Cette analyse est effectuée en utilisant la technique de plan d'expériences "*Design of Experiments*" (DoE) afin de valider l'importance des métriques de couplage et de prévoir l'impact du changement dans les logiciels. Cela a permis de mettre en évidence la corrélation qui existe entre les métriques de couplage et l'impact du changement.

La mise en œuvre de l'approche proposée est réalisée en utilisant trois métriques de couplage bien connues et qui sont largement utilisées dans la littérature : couplage entre objets (CBO), réponse pour la classe (RFC) et couplage de transmission de message (MPC).

L'expérimentation est réalisée sur un nombre bien déterminé de versions d'une librairie open source "*JFreeChart*". Les métriques considérées dans cette expérimentation sont des métriques de couplage : CBO, RFC et MPC. Les résultats de test utilisant la DoE ont validé le choix des métriques CBO et MPC pour analyser leur

influence sur le modèle obtenu pour une meilleure estimation de l'impact du changement.

L'objectif principal de cette étude est de mieux identifier les facteurs significatifs pour la maintenance du logiciel.

VI. Conclusions et perspectives

a. Conclusions

Nous avons abordé dans cette thèse la problématique de l'estimation de l'impact de changement dans un système à objets à travers la technique mathématique de "plan d'expériences", "Design of Experiments" (DoE), cette approche est largement utilisée dans les domaines des sciences expérimentales telles que la biologie, la physique et la chimie.

Nous avons tout d'abord collecté des données à partir du référentiel "JFreeChart" implémenté en java. Nous avons traité les données extraites de douze versions de "JFreeChart", comme requis par la méthode "Design of Experiments" (DoE). Ensuite, nous avons mis en œuvre un processus consistant d'abord à mesurer les métriques logicielles, puis le taux de changement de code en considérant les douze versions de "JFreeChart". Les différentes valeurs des métriques CBO, RFC et MPC ont été extraites directement du code à l'aide de l'outil "SACI" que nous avons développé. Nous avons utilisé le code source disponible des versions de "JFreeChart" pour mesurer le volume des modifications logicielles. Ces dernières sont mesurées en comparant une version donnée à la suivante, obtenue après la mise à jour du logiciel. Ce processus de comparaison est répété jusqu'à épuisement de toutes les versions considérées. Après cela, nous avons procédé à un ajustement vu que la particularité des métriques de couplage est qu'elles concernent chaque classe du logiciel. A cet effet, Nous avons défini des métriques de couplage représentatives afin d'étudier l'ensemble du logiciel. La technique proposée permet de déterminer le meilleur représentant pour chaque type de métriques. Enfin, nous avons procédé à l'application de cette technique pour chaque métrique de couplage à l'ensemble des classes de chacune des versions du logiciel.

Comme nous disposons d'un ensemble d'expériences, à savoir les valeurs de CBO, RFC et MPC pour chaque version de la librairie "JFreeChart", et de l'impact de changement estimé, exprimé sous forme de ratio, nous avons pu calculer les valeurs des coefficients du modèle mathématique obtenu par le biais de la

technique DoE. En exploitant les résultats des 12 versions (expériences), nous obtenons une équation mathématique sous forme d'un plan à deux facteurs.

En dernier lieu, comme l'équation obtenue est constitué de facteurs à l'échelle générique du modèle, nous avons procédé à une transformation de l'échelle générique à l'échelle finale, ce qui nous a permis d'avoir un modèle mathématique final.

Rappelons que selon les conditions de test de robustesse des modèles mathématiques, un modèle est considéré comme robuste si la valeur de son coefficient de corrélation R^2 tend à 1. Dans notre cas, la valeur de R^2 obtenue est de 0,94. Ce qui est en fait une validation du modèle de l'estimation de l'impact du changement obtenu. De plus, cela affirme aussi que le modèle montre l'influence de l'effet quadratique de la métrique MPC sur l'impact de changement.

Ces résultats confirment les résultats trouvés dans travaux antérieurs (Abdeen et al., 2015; M. Ajmal Chaumun et al., 2002), qui affirment l'importance du couplage en tant que mesure pour l'estimation de l'impact du changement.

L'avantage de la technique DoE proposée est qu'elle assure la construction du modèle à partir d'un nombre réduit d'expériences (versions).

Le modèle mathématique obtenu est destiné à jouer le rôle d'indicateur pour les décideurs leur permettant de disposer rapidement d'une mesure de ce que peut coûter le logiciel en termes de modifications et par conséquent d'effort pour le maintenir ou le faire évoluer.

Dans un contexte de plusieurs applicatifs (logiciels), le but des modèles mathématiques est en effet de pouvoir dresser un tableau de bord à destination des décideurs leur permettant de disposer d'indicateurs pour tous les constituants de leur patrimoine applicatif.

Ces indicateurs peuvent donner une orientation en terme d'investigations plus profondes pour étudier la pertinence de la maintenance d'un applicatif donné ou

pour réaliser un arbitrage rapide sur les logiciels qu'il faudra considérer en priorité en matière de maintenance logiciel.

b. Perspectives

Comme perspectives majeures aux travaux de cette thèse, nous pouvons envisager les suivantes :

- ✓ L'enrichissement du modèle obtenu en considérant d'autres métriques de couplage connues dans la littérature du domaine.
- ✓ Exploration de la pertinence d'autres métriques mesurant d'autres propriétés autres que le couplage, comme par exemple la cohésion, et leur influence sur l'estimation de l'impact du changement.
- ✓ Enfin, partant du constat que cette approche doit être appliquée à chaque logiciel, néanmoins elle peut être généralisée à des familles de logiciels définies dans le cadre d'une typologie de logiciels, nous prévoyons de réaliser une classification par familles (typologie) selon un certain nombre de caractéristiques (nombre de classes, domaines d'application, nombre de méthodes par classe,...etc.) des logiciels, afin pouvoir maximiser la réutilisation du modèle obtenu.

Références

- Abdeen, H., Bali, K., Sahraoui, H., & Dufour, B. (2015). Learning dependency-based change impact predictors using independent change histories. *Information and Software Technology*, 67, 220–235.
- Abdi , M. K. (2007). Analyse et Prédiction d'impact de changement dans système à objet. (Thèse de doctorat d'état en Informatique, Université Es-Sénia d'Oran Algérie).
- Abdi, M. K., & Dinedane, Z. M. (2015). Change Impact Identification in Object-Oriented System: Dependence Graph Approach. *International Journal of Education and Management Engineering*, 5(3), 1.
- Abdi, M. K., Lounis, H., & Sahraoui, H. (2006). *Analyzing change impact in object-oriented systems*. Paper presented at the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06).
- Ahmad, A., Basson, H., & Bouneffa, M. (2017). *Analyzing and modeling the structural and qualitative interdependencies of software evolution*. Paper presented at the 7th International Workshop on Computer Science and Engineering, workshop of, WCSE 2017, International Conference on Software Engineering (ICSE).
- Ajila, S. (1995). Software maintenance: an approach to impact analysis of objects change. *Software: Practice and Experience*, 25(10), 1155-1181.
- Almugrin, S., Albattah, W., & Melton, A. (2016). Using indirect coupling metrics to predict package maintainability and testability. *Journal of Systems and Software*, 121, 298-310. doi: <https://doi.org/10.1016/j.jss.2016.02.024>
- Antoniol, G., Canfora, G., & De Lucia, A. (1999). *Estimating the size of changes for evolving object oriented systems: A case study*. Paper presented at the Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403).
- Anwer, S., Adbellatif, A., Alshayeb, M., & Anjum, M. S. (2017, 8-9 March 2017). *Effect of coupling on software faults: An empirical study*. Paper presented at the International Conference on Communication, Computing and Digital Systems (C-CODE).
- Arthur, L. J. (1988). *Software evolution: the software maintenance challenge*: Wiley-Interscience.
- Asiedu, Y., & Gu, P. (1998). Product life cycle cost analysis: state of the art review. *International journal of production research*, 36(4), 883-908.
- Balaji, S., & Murugaiyan, M. S. (2012). Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*, 2(1), 26-30.
- Barry, B. (1981). *Software engineering economics*. New York, 197.

- Basili, V. R., & Rombach, H. D. (1990). Towards a comprehensive framework for reuse: model-based reuse characterization schemes: MARYLAND UNIV COLLEGE PARK INST FOR ADVANCED COMPUTER STUDIES.
- Belady, L. A., & Lehman, M. M. (1976). A model of large program development. *IBM Systems Journal*, 15(3), 225-252. doi: 10.1147/sj.153.0225
- Benedusi, P., Cimitile, A., & De Carlini, U. (1992). Reverse engineering processes, design document production, and structure charts. *Journal of Systems and Software*, 19(3), 225-245.
- Bennett, K. H., & Rajlich, V. T. (2000). *Software maintenance and evolution: a roadmap*. Paper presented at the Proceedings of the Conference on the Future of Software Engineering.
- Bennett, K. H., & Xu, J. (2003). *Software services and software maintenance*. Paper presented at the Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.
- Bezerra, M. A., Santelli, R. E., Oliveira, E. P., Villar, L. S., & Escalera, L. A. (2008). Response surface methodology (RSM) as a tool for optimization in analytical chemistry. *Talanta*, 76(5), 965-977.
- Bhattacharya, P., Iliofotou, M., Neamtiu, I., & Faloutsos, M. (2012, 2-9 June 2012). *Graph-based analysis and prediction for software evolution*. Paper presented at the 34th International Conference on Software Engineering (ICSE).
- Bianchi, A., Fasolino, A. R., & Visaggio, G. (2000). *An exploratory case study of the maintenance effectiveness of traceability models*. Paper presented at the Proceedings IWPC 2000. 8th International workshop on program comprehension.
- Black, S. (2006). The role of ripple effect in software evolution. *Software evolution and feedback*, 249.
- Black, S. (2008). Deriving an approximation algorithm for automatic computation of ripple effect measures. *Information and Software Technology*, 50(7-8), 723-736.
- Bohner, S. A., & Arnold, R. S. (1996). An Introduction to Software Change Impact Analysis. *IEEE Computer society, Press Los Alamitos, California* (in proceedings of the Conference on Software Change Impact Analysis).
- Booth, K. H., & Cox, D. R. (1962). Some systematic supersaturated designs. *Technometrics*, 4(4), 489-495.
- Bouneffa, M., & Ahmad, A. (2013). *The change impact analysis in BPM based software applications: A graph rewriting and ontology based approach*. Paper presented at the International Conference on Enterprise Information Systems.
- Box, G. E., Hunter, J. S., & Hunter, W. G. (2005). *Statistics for experimenters Wiley Series in Probability and Statistics*: Wiley Hoboken, NJ.

- Buckner, J., Buchta, J., Petrenko, M., & Rajlich, V. (2005). *JRipples: A tool for program comprehension during incremental change*. Paper presented at the 13th International Workshop on Program Comprehension (IWPC'05).
- Bulut, S. (2014). Faber polynomial coefficient estimates for a comprehensive subclass of analytic bi-univalent functions. *Comptes Rendus Mathématique*, 352(6), 479-484. doi: <https://doi.org/10.1016/j.crma.2014.04.004>
- Canfora, G., & Cimitile, A. (2001). Software maintenance *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals* (pp. 91-120): World Scientific.
- Canning, R. G. (1972). That maintenance 'iceberg'. *EDP Analyzer*, 10(10), 1-14.
- Cantave, M. H. R. (2003). *Abstractions via un modèle générique d'applications orientées objets*. Master's thesis, Université Laval, Canada.
- Chapin, N. (1988). *Software maintenance life cycle*. Paper presented at the Proceedings. Conference on Software Maintenance, 1988.
- Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., & Tan, W. G. (2001). Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1), 3-30.
- Chaturvedi, A. (2014). Change Impact Analysis Based Regression Testing of Web Services. *arXiv preprint arXiv:1408.1600*.
- Chaumon, M. A., Kabaili, H., Keller, R. K., & Lustman, F. (2002). A change impact model for changeability assessment in object-oriented software systems. *Science of Computer Programming*, 45(2), 155-174. doi: [https://doi.org/10.1016/S0167-6423\(02\)00058-8](https://doi.org/10.1016/S0167-6423(02)00058-8)
- Chaumon, M. A., Kabaili, H., Keller, R. K., & Lustman, F. (2002). A change impact model for changeability assessment in object-oriented software systems. *Science of Computer Programming*, 45(2-3), 155-174.
- Chidamber, S., & Kemerer, C. (1994). A Metrics Suite for Object- Oriented Design. *IEEE Transaction on Software Engineering*, 20(6), 476-493.
- Conti, G., Dean, E., Sinda, M., & Sangster, B. (2008). *Visual reverse engineering of binary and data files*. Paper presented at the International Workshop on Visualization for Computer Security.
- Cornell, J. A. (2011). *Experiments with mixtures: designs, models, and the analysis of mixture data* (Vol. 403): John Wiley & Sons.
- Dagnelie, P. (2012). PRINCIPES D'EXPERIMENTATION Planification des expériences et analyse de leurs résultats. 2012, *LES PRESSES AGRONOMIQUES DE GEMBLoux, A.S.B.L. Passage des D'éportés 2 – B-5030 Gembloux (Belgique) D/2012/1665/117 ISBN 978-2-87016-117-3*.

- Dahane, M. (2011). *Validation empirique d'un modele d'impact de changement pour les système à objets*. Magister en informatique, Univesite d'oran.
- Dahane, M., Abdi, M. K., Bouneffa, M., Ahmad, A., & Basson, H. (2019). Using Design of Experiments to Analyze Open Source Software Metrics for Change Impact Estimation. *International Journal of Open Source Software and Processes (IJOSSP)*, 10(1), 16-33.
- Davim, J. P., & Aveiro, P. (2016). *Design of experiments in production engineering*: Springer.
- de Aguiar, P. F., Bourguignon, B., Khots, M., Massart, D., & Phan-Than-Luu, R. (1995). D-optimal designs. *Chemometrics and intelligent laboratory systems*, 30(2), 199-210.
- De Lucia, A., Fasano, F., & Oliveto, R. (2008). *Traceability management for impact analysis*. Paper presented at the 2008 Frontiers of Software Maintenance.
- Dinedane, M. Z. (2008). Prédiction de l'impact de changement dans un système à Objets. *mémoire d'ingénieur d'état, Université d'oran Es-Sénia*
- Dit, B., Wagner, M., Wen, S., Wang, W., Linares-Vásquez, M., Poshyvanyk, D., & Kagdi, H. (2014). *Impactminer: A tool for change impact analysis*. Paper presented at the Companion Proceedings of the 36th International Conference on Software Engineering.
- Electrical, I. o., & Engineers, E. (1988). *IEEE guide to software configuration management*: IEEE.
- Elmidaoui, S., Cheikhi, L., & Idri, A. (2016, 19-20 Oct. 2016). *A survey of empirical studies in software product maintainability prediction models*. Paper presented at the 11th International Conference on Intelligent Systems: Theories and Applications (SITA).
- Fasolino, A. R., & Visaggio, G. (1999). *Improving software comprehension through an automated dependency tracer*. Paper presented at the Proceedings Seventh International Workshop on Program Comprehension.
- Ferens, D. V. (1999). The Conundrum of Software Estimation Models. *IEEE AES Systems Magazine*, 23-29.
- Ferreira, S. C., Bruns, R., Ferreira, H., Matos, G., David, J., Brandao, G., . . . Souza, A. (2007). Box-Behnken design: an alternative for the optimization of analytical methods. *Analytica chimica acta*, 597(2), 179-186.
- Franco, J. (2008). *Planification d'expériences numériques en phase exploratoire pour la simulation des phénomènes complexes*. Ecole Nationale Supérieure des Mines de Saint-Etienne.
- Galbo, S. P. D. (2017). *A Survey of Impact Analysis Tools for Effective Code Evolution*.
- Gilb, T., & Finzi, S. (1988). *Principles of software engineering management* (Vol. 11): Addison-wesley Reading, MA.
- Gotel, O. C., & Finkelstein, C. (1994). *An analysis of the requirements traceability problem*. Paper presented at the Proceedings of IEEE International Conference on Requirements Engineering.

- Goupy, J. (1995). Plans d'expériences non conventionnels. Théorie et applications (ou comment sauver un plan raté). *Analisis*, 23, 152-158.
- Goupy, J. (1996). Unconventional experimental designs theory and application. *Chemometrics and intelligent laboratory systems*, 33(1), 3-16.
- Goupy, J. (2000). Boolean experimental designs. *Analisis*, 28(7), 563-570.
- Goupy, J., & Lee, C. (2006). Introduction au plan d'experiences. 3^{eme} edition DUNOD.
- Gunasegaram, D., Farnsworth, D., & Nguyen, T. (2009). Identification of critical factors affecting shrinkage porosity in permanent mold casting using numerical simulations based on design of experiments. *Journal of materials processing technology*, 209(3), 1209-1219.
- Halpern, M. I. (1965). Machine independence: its technology and economics. *Commun. ACM*, 8(12), 782-785. doi: 10.1145/365691.365943
- Harrison, W., & Cook, C. (1990). *Insights on improving the maintenance process through software measurement*. Paper presented at the Proceedings. Conference on Software Maintenance 1990.
- Harrold, M. J., & Malloy, B. (1993). A unified interprocedural program representation for a maintenance environment. *IEEE Transactions on Software Engineering*, 19(6), 584-593.
- Heck, R. H., & Thomas, S. L. (2015). *An introduction to multilevel modeling techniques: MLM and SEM approaches using Mplus*: Routledge.
- Hegedűs, P., Kádár, I., Ferenc, R., & Gyimóthy, T. (2018). Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, 95, 313-327. doi: <https://doi.org/10.1016/j.infsof.2017.11.012>
- Henrard, J., Roland, D., Englebert, V., Hick, J.-M., & Hainaut, J.-L. (1998). Outils d'analyse de programmes pour la rétro-conception de bases de données.
- Ibrahim, N., Kadir, W. M. N. W., & Deris, S. (2008). *Comparative evaluation of change propagation approaches towards resilient software evolution*. Paper presented at the 2008 The Third International Conference on Software Engineering Advances.
- Ibrahim, S., Idris, N. B., Munro, M., & Deraman, A. (2005). *Implementing a Document-based Requirements Traceability: A Case Study*. Paper presented at the IASTED Conf. on Software Engineering.
- Jazayeri, M. (2005). *Species evolve, individuals age*. Paper presented at the Eighth International Workshop on Principles of Software Evolution (IWPSE'05).
- José, P. M., Mauricio, D., & Rodríguez, G. (2014). A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications (IJSEA)*, Vol.5(No.6), 31-53. doi: 10.5121/ijsea.2014.5603

- Kabaili, H., Keller, R. K., & Lustman, F. (2005). *Assessing object-oriented software changeability with design metrics*. Paper presented at the Software Engineering: IASTED International Conference Proceedings.
- Kan, S. H. (2002). *Metrics and models in software quality engineering*: Addison-Wesley Longman Publishing Co., Inc.
- Keller, A., & Demeyer, S. (2012). Change impact analysis for UML model maintenance *Emerging Technologies for the Evolution and Maintenance of Software Models* (pp. 32-56): IGI Global.
- Khoder, K. (2011). *Optimisation de composants hyperfréquences par la technique des plans à surfaces de réponses*. Doctorat Thesis, Limoges.
- Kitchenham, B. A., Travassos, G. H., Von Mayrhauser, A., Niessink, F., Schneidewind, N. F., Singer, J., . . . Yang, H. (1999). Towards an ontology of software maintenance. *Journal of Software Maintenance: Research and Practice*, 11(6), 365-389.
- Koshal, R. (1933). Application of the method of maximum likelihood to the improvement of curves fitted by the method of moments. *Journal of the Royal Statistical Society*, 96(2), 303-313.
- Kung, H.-J., & Hsu, C. (1998). *Software maintenance life cycle model*. Paper presented at the Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272).
- Leblang, D. B. (1995). *The CM challenge: Configuration management that works*. Paper presented at the Configuration management.
- Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1060-1076.
- Lehman, M. M., Perry, D. E., & Ramil, J. F. (1998). *On evidence supporting the feast hypothesis and the laws of software evolution*. Paper presented at the Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262).
- Lehman, M. M., & Ramil, J. F. (2002). Software Evolution and Software Evolution Processes. *Annals of Software Engineering*, 14(1), 275-309. doi: 10.1023/a:1020557525901
- Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). *Metrics and laws of software evolution-the nineties view*. Paper presented at the Proceedings Fourth International Software Metrics Symposium.
- Lehnert, S. (2011). A review of software change impact analysis. *Ilmenau University of Technology, Tech. Rep.*
- Li, B., Sun, X., Leung, H., & Zhang, S. (2013). A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8), 613-646.
- Lin, D. K. (1995). Generating systematic supersaturated designs. *Technometrics*, 37(2), 213-225.

- Lindvall, M., & Sandahl, K. (1998). Traceability aspects of impact analysis in object-oriented systems. *Journal of Software Maintenance: Research and Practice*, 10(1), 37-57.
- Madhavji, N. H., Lehman, M., Perry, D., & Ramil, J. (2006). *Software evolution and feedback*: Wiley Online Library.
- Mamone, S. (1994). The IEEE standard for software maintenance. *ACM SIGSOFT Software Engineering Notes*, 19(1), 75-76.
- Martin, R. C. (2005). Agile software development: principles, patterns, and practices. *Computing Reviews*, 46(2), 91.
- Martin, R. J., & Osborne, W. M. (1985). Guidance on Software Maintenance *Computer Science and technology U.S*, 500-106.
- McGraw-Hill. (1992). *Software Engineering: A Practitioner's Approach*. (Third Edition), . Chapitre 10.
- Mens, T. (2008). Introduction and roadmap: History and challenges of software evolution *Software evolution* (pp. 1-11): Springer.
- Montgomery, D. C. (1995). Design of experiments. *New York*, 225-364.
- Mozzo, G. (1990). Plan quadratique gigogne. *Revue de statistique appliquée*, 38(3), 23-34.
- Müller, K., & Rumpe, B. (2014). A Model-Based Approach to Impact Analysis Using Model Differencing. *CoRR*, abs/1406.6834.
- Musco, V., Monperrus, M., & Preux, P. (2014). A Generative Model of Software Dependency Graphs to Better Understand Software Evolution. *CoRR*, abs/1410.7921.
- Narayanaswamy, K., & Scacchi, W. (1987). Maintaining configurations of evolving software systems. *IEEE Transactions on Software Engineering*(3), 324-334.
- Nuez-Varela, A. S., Prez-Gonzalez, H. G., Martinez-Perez, F. E., & Soubervielle-Montalvo, C. (2017). Source code metrics. *Journal of Systems and Software*, 128(C), 164-197. doi: 10.1016/j.jss.2017.03.044
- Parnas, D. L. (1994). *Software aging*. Paper presented at the Proceedings of 16th International Conference on Software Engineering.
- Perry, D. E., & Evangelist, W. M. (1987). *An empirical study of software interface faults—an update*. Paper presented at the Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences.
- Pillet, M. (2000). *Les plans d'expériences par la méthode Taguchi*: Maurice Pillet.
- Pirzada, S. S. (1988). A statistical examination of the evolution of the UNIX system.
- Podgurski, A., & Clarke, L. A. (1990). A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9), 965-979.



- Qu, X. (2007). Statistical properties of Rechtschaffner designs. *Journal of Statistical Planning and Inference*, 137(7), 2156-2164.
- Queille, J.-P., Voidrot, J.-F., Wilde, N., & Munro, M. (1994). *The impact analysis task in software maintenance: A model and a case study*. Paper presented at the Proceedings of the International Conference on Software Maintenance.
- Rajlich, V. (1997). *A model for change propagation based on graph rewriting*. Paper presented at the 1997 Proceedings International Conference on Software Maintenance.
- Rajlich, V. T., & Bennett, K. H. (2000). A staged model for the software life cycle. *Computer*, 33(7), 66-71.
- Ren, X., Shah, F., Tip, F., Ryder, B. G., & Chesley, O. (2004). *Chianti: a tool for change impact analysis of java programs*. Paper presented at the ACM Sigplan Notices.
- Robles, G., Amor, J. J., Gonzalez-Barahona, J. M., & Herraiz, I. (2005). *Evolution and growth in large libre software projects*. Paper presented at the Eighth International Workshop on Principles of Software Evolution (IWPSE'05).
- Roquemore, K. (1976). Hybrid designs for quadratic response surfaces. *Technometrics*, 18(4), 419-423.
- Ryder, B. G., & Tip, F. (2001). *Change impact analysis for object-oriented programs*. Paper presented at the Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.
- Saraiva, J. d. A. G., de França, M. S., Soares, S. C. B., Filho, F. J. C. L., & de Souza, Renata M. C. R. (2015). Classifying metrics for assessing Object-Oriented Software Maintainability: A family of metrics' catalogs. *Journal of Systems and Software*, 103, 85-101. doi: <https://doi.org/10.1016/j.jss.2015.01.014>
- Schneidewind, N. F. (1987). The state of software maintenance. *IEEE Transactions on Software Engineering*(3), 303-310.
- Shi, N., & Olsson, R. A. (2006). *Reverse engineering of design patterns from java source code*. Paper presented at the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06).
- Silva, F. F., Borel, E., Lopes, E., & Murta, L. G. P. (2014, Sept. 28 2014-Oct. 3 2014). *Towards a Difference Detection Algorithm Aware of Refactoring-Related Changes*. Paper presented at the Brazilian Symposium on Software Engineering.
- Sommerville, I. (2011). Software engineering 9th Edition. ISBN-10, 137035152.
- Stachour, P., & Collier-Brown, D. (2009). You don't know jack about software maintenance. *Communications of the ACM*, 52(11), 54-58.

- Standard, I. (2006). Software engineering—software life cycle processes—maintenance. *ISO Standard, 14764*, 2006.
- Standardization, I. O. f. (2008). *Systems and Software Engineering: Software Life Cycle Processes*: ISO.
- Sudhish, P. S., Jain, A. K., & Cao, K. (2016). Adaptive fusion of biometric and biographic information for identity de-duplication. *Pattern Recognition Letters*, 84, 199-207. doi: <https://doi.org/10.1016/j.patrec.2016.10.011>
- Swanson, E. B. (1976). *The dimensions of maintenance*. Paper presented at the Proceedings of the 2nd international conference on Software engineering.
- Telford, J. K. (2007). A brief introduction to design of experiments. *Johns Hopkins apl technical digest*, 27(3), 224-232.
- Tempero, E., & Ralph, P. (2018). A framework for defining coupling metrics. *Science of Computer Programming*. doi: <https://doi.org/10.1016/j.scico.2018.02.004>
- Tinsson, W. (2010). Plans d'expérience: constructions et analyses statistiques. *Springer, ISBN 978-642-11471-7*.
- Tiwari, S., & Rathore, S. S. (2018). *Coupling and Cohesion Metrics for Object-Oriented Software: A Systematic Mapping Study*. <http://doi.acm.org/10.1145/3172871.3172878>
- Tonella, P. (2005). *Reverse engineering of object oriented code*. Paper presented at the Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.
- Truver, R. J., & Munro, M. (1991). An Early Impact Analysis Technique for Software Maintenance. *in proceedings of the journal of Software Maintenance Research and Practice*, Vol. 6(No. 1).
- Tu, Q. (2000). *Evolution in open source software: A case study*. Paper presented at the Proceedings 2000 International Conference on Software Maintenance.
- Turver, R. J., & Munro, M. (1994). An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, 6(1), 35-52.
- Umezawa, T. (1967). A Method of Two-Level Simplification of Boolean Functions. *Nagoya Mathematical Journal*, 29, 201-210.
- Van Den Berg, K., Conejero, J. M., & Hernández, J. (2007). Analysis of crosscutting in early software development phases based on traceability *Transactions on aspect-oriented software development III* (pp. 73-104): Springer.
- Vanaja, K., & Shobha Rani, R. (2007). Design of experiments: concept and applications of Plackett Burman design. *Clinical research and regulatory affairs*, 24(1), 1-23.
- Vivier, S. (2002). *Stratégies d'optimisation par la méthode des Plans d'Expériences, et Application aux dispositifs électrotechniques modélisés par Eléments Finis*. Doctorat thesis, Université des Sciences et Technologie de Lille-Lille I.

- Warshall S. (1962). A theorem on Boolean matrices *Journal of the ACM*, 9(1), 11-12.
- Watson, P. F., & Petrie, A. (2010). Method agreement analysis: A review of correct methodology. *Theriogenology*, 73(9), 1167-1179. doi: <https://doi.org/10.1016/j.theriogenology.2010.01.003>
- Wu, C.-F. J., & Hamada, M. (2000). Experiments: planning, analysis, and parameter design optimization.: John Wiley.
- Yang, H., & Ward, M. (2003). *Successful evolution of software systems*: Artech House.
- Yau, S., Collofello, J., & MacGregor, T. (1993). *Ripple effect analysis of software maintenance*. Paper presented at the Software engineering metrics I.
- Youden, W. J. (1951). Statistical methods for chemists: John Wiley and Sons. New-York. 126 pages.
- Yu, L., & Mishra, A. (2013). An Empirical Study of Lehman's Law on Software Quality Evolution. *Int. J. Software and Informatics*, 7(3), 469-481.
- Zhang, L., Kim, M., & Khurshid, S. (2012). *FaultTracer: a change impact and regression fault analysis tool for evolving Java programs*. Paper presented at the Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.
- Zhao, J., Yang, H., Xiang, L., & Xu, B. (2002). Change impact analysis to support architectural evolution. *Journal of software maintenance and evolution: Research and Practice*, 14(5), 317-333.

Annexes



Figure 30 - Environnement de l'outil "SACI"

SACI dispose de deux principaux modules :

- **Fichier** : cette option regroupe deux fonctionnalités, à savoir :
 - ✓ *Charger* : permet le chargement du logiciel sur lequel l'expérimentation va être réalisée.
 - ✓ *Enregistrer* : fournit la possibilité de sauvegarder les modifications effectuées sur l'expérience.

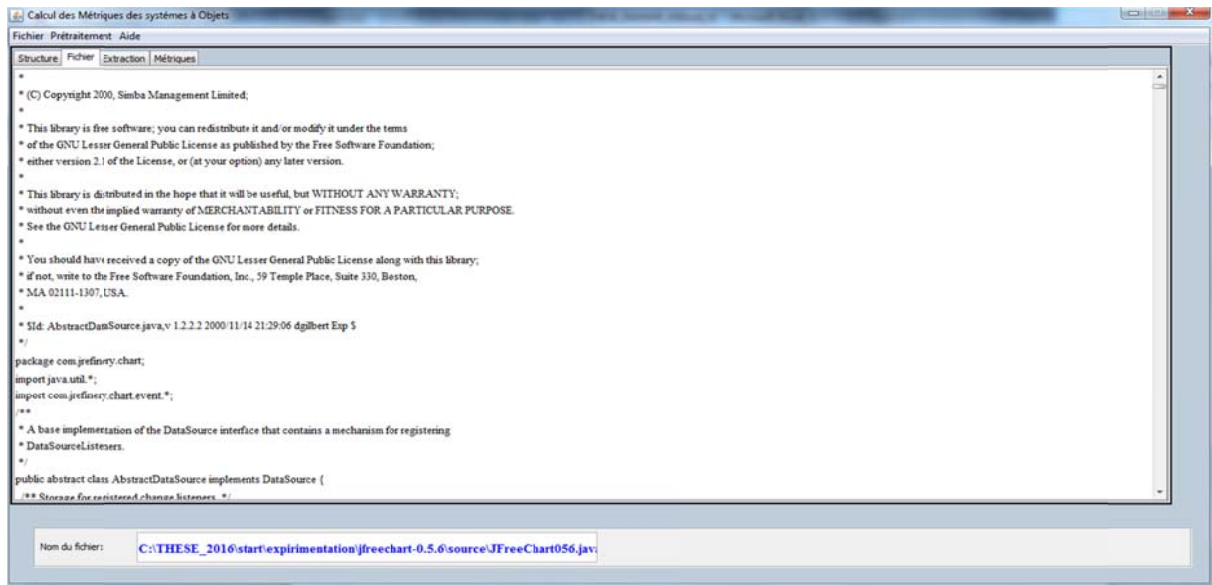


Figure 31 - Volet "Fichier" de l'outil "SACI"

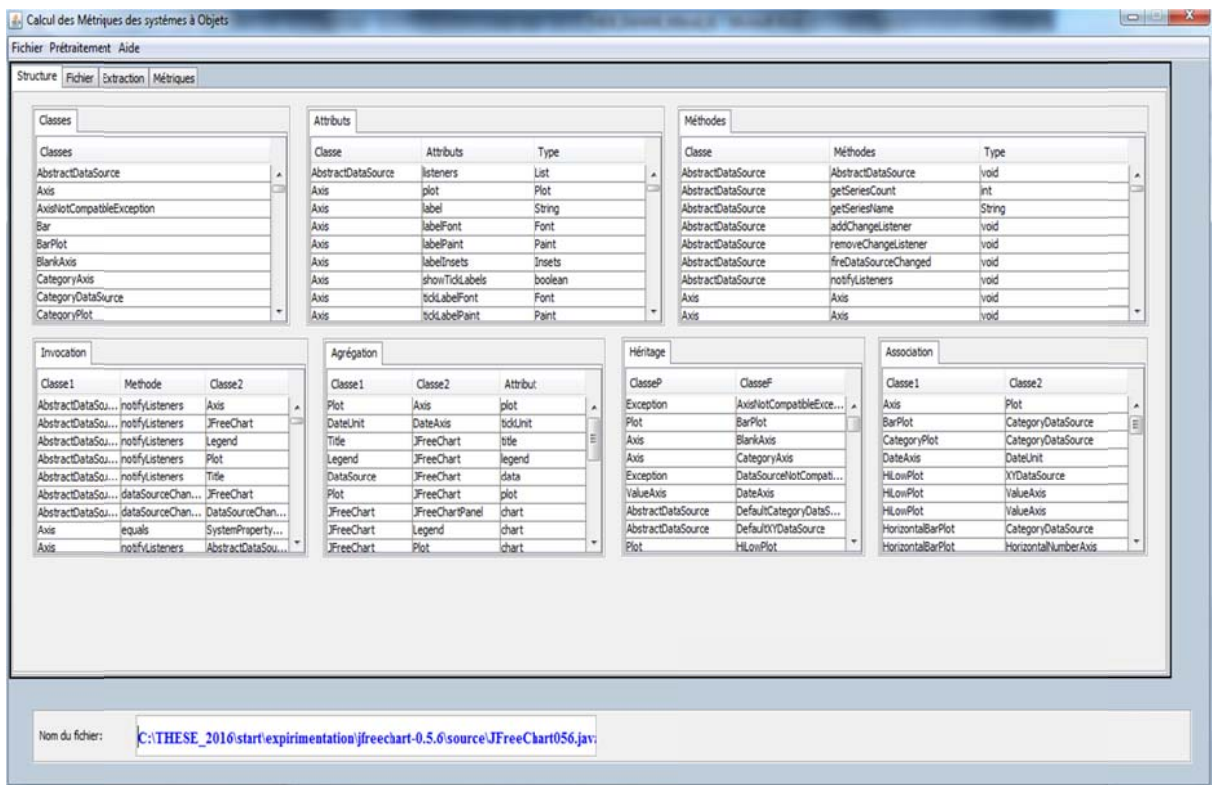
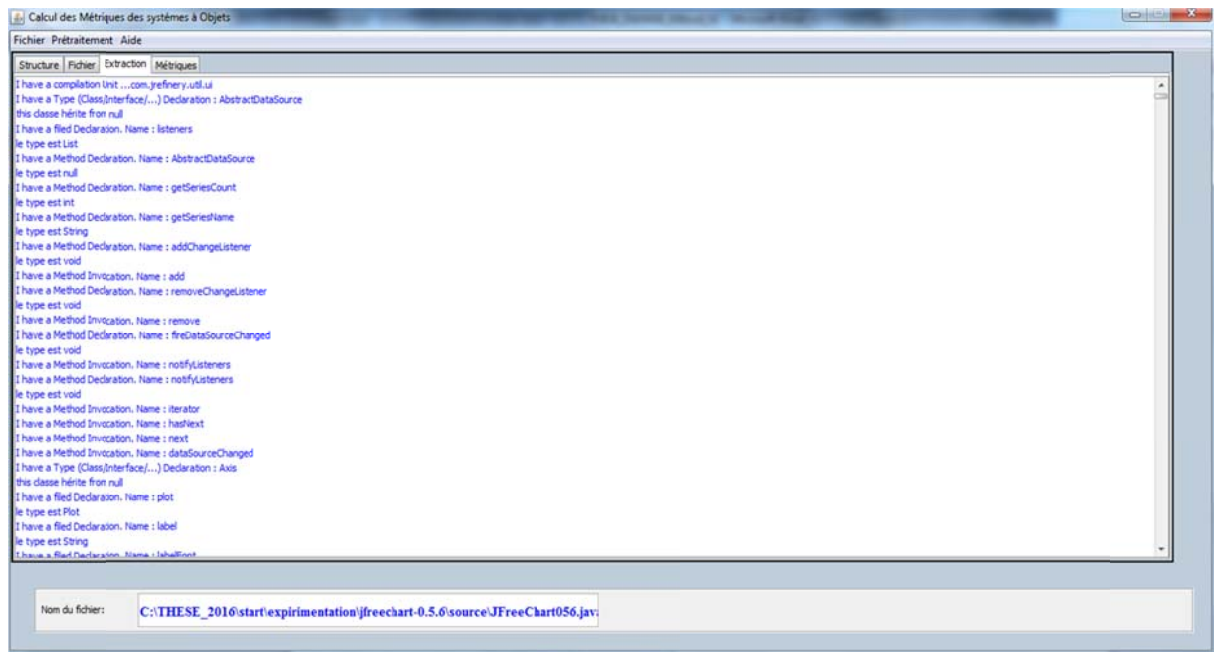


Figure 32 - Volet "Structure" de l'outil "SACI"

- **Prétraitement** : cette option regroupe à son tour deux fonctionnalités, à savoir :
 - ✓ *Extraction* : permet l'extraction des différents composants du logiciel (Méthodes, Attributs et Classes).

- ✓ *Calcul Métrique* : elle retourne les différentes valeurs des métriques logicielles sur la base de leurs définitions.



Calcul des Métriques des systèmes à Objets

Fichier Prétraitement Aide

Structure	Fichier	Extraction	Métriques								
classe name	CBO	CBO'	RFC	MPC	DAC	CBO-ILB	CBO-ILJ	OMMEC	OMMEC	DMMEC	AMMIC
AbstractDataSource	103	100	14	7	0	7	96	0	0	0	10
Axis	244	241	96	64	0	68	176	0	0	0	106
AxisNotCompatibl	1	0	1	0	0	1	0	0	0	0	0
Bar	97	97	7	0	0	0	97	0	0	0	0
BarPlot	107	104	63	49	0	61	66	0	0	0	62
BlankAxis	61	60	6	0	0	1	60	0	0	0	0
CategoryAxis	6	3	2	0	0	1	6	0	0	0	0
CategoryDataSource	38	38	3	0	0	0	38	0	0	0	0
CategoryPlot	62	62	3	0	0	1	61	0	0	0	0
DataSource	44	44	4	0	0	2	42	0	0	0	0
DataSourceNotCo	1	0	1	0	0	1	0	0	0	0	0
DataSources	96	96	86	82	0	82	14	0	0	0	22
DateAxis	33	31	38	24	0	26	7	0	0	0	41
DateUnit	9	9	6	1	0	2	7	0	0	0	1
DefaultCategoryD	62	61	19	3	0	4	58	0	0	0	6
DefaultXYDataSou	57	56	11	1	0	2	56	0	0	0	0
XYDataItem	3	3	3	1	0	1	2	0	0	0	0
DomainInfo	6	6	2	0	0	0	6	0	0	0	0
HiLow	42	42	21	8	0	8	34	0	0	0	0
HiLowPlot	361	360	291	276	0	280	81	0	0	0	153
HorizontalAxis	10	10	2	0	0	0	10	0	0	0	0
HorizontalBarPlot	434	433	348	331	0	335	99	0	0	0	176
HorizontalCatego	145	144	92	83	0	84	61	0	0	0	37
HorizontalDateAxis	308	307	154	137	0	138	170	0	0	0	81
HorizontalNumbe	277	276	172	167	0	168	119	0	0	0	97
HorizontalValueP	4	4	2	0	0	0	4	0	0	0	0
JFreeChart	402	402	207	170	0	186	223	0	0	0	259
JFreeChartPanel	49	48	32	28	0	30	19	0	0	0	28
Legend	112	111	8	2	0	8	108	0	0	0	0
Line	26	26	8	0	0	0	26	0	0	0	0
LinePlot	319	318	231	218	0	222	97	0	0	0	88

Nom du fichier: C:\THESE_2016\start\expirimentation\jfreechart-0.5.6\source\JFreeChart056.java

Figure 34 - Volet "Calcul Métrique" de l'outil "SACI"

Tableau 5 - Outils d'analyse d'impact de changement (Galbo, 2017)

Nom Outil	Année de création.	Dernier Motif.	langage
Chianti	2004	2008	Java
CodeSurfer	1999	2017	C
EAT	2005	n/a	Java
eROSE	2004	2005	Java
FaultTracer	2012	2016	Java
Frama-C	2008	2016	C
Imp	2012	n/a	C/C++
ImpactMiner	2014	2015	Java
ImpactViz	2010	2011	Java
Impala	2008	n/a	Java
Indus	2005	2009	Java
JArchitect	2006	2017	Java
JRipples	2005	2016	Java
JSlice	2008	2008	Java
JUnit/CIA	2008	n/a	Java
Kaveri	2006	2014	Java
Lattix	2004	2017	Java/C/C++
REST	2001	2008	C
SLICE	1995	n/a	C

ملخص

إن قابلية الصيانة للبرنامج هي معيار لجودته التي تقيس قدرته على التغيير بحد أدنى من التأثير، وبالتالي الجهد والوقت وبعبارة أخرى التكلفة. تعتبر قابلية الصيانة عمومًا مزيًا من المعايير مثل قابلية التغيير والقابلية للاختبار. في هذه الأطروحة، نقدم نهجًا وطريقة للتعامل مع قياس الصيانة. والهدف هو أن يكون هناك مقياس يبسط عملية تقييم قابلية الصيانة ويسمح باتخاذ القرارات الاستثمارية من حيث عملية التغيير التي يجب إجراؤها في برنامج معين. يعتمد أسلوبنا على استخدام طريقة إحصائية تسمى "التصاميم التجريبية". تعد معلومات الإدخال عبارة عن مجموعة من إصدارات البرامج ويكون الإخراج عبارة عن تعبير رياضي يقارب قابلية الصيانة وفقًا لبعض المقاييس الهيكلية للبرامج المستخدمة على نطاق واسع.

الكلمات المفتاحية: قابلية الصيانة، الأنظمة الموجهة، تغيير الأثر، مقاييس البرامج، مقاييس الاقتران، التصاميم التجريبية

Abstract

The software evolution generally depends on its structure and some quality criteria like maintainability. It is known as the quality attributes measuring the capability of software to change with minimal impact on side effort. In this thesis, we show an approach and method dealing with quantitative measurement of the maintainability criteria as mathematical function of some structural software metrics. The approach is based on the use of statistical method known as "Design of Experiments". The input of our method is a set of software versions and the output is a mathematical expression approximating the maintainability as a function of some widely used structural software metrics.

Keywords: Maintainability, object oriented systems, change impact, software metrics, coupling metrics, Design of Experiments.

Résumé

La maintenabilité du logiciel est un critère de sa qualité mesurant son aptitude à changer avec un minimum d'impact et donc d'effort, de temps et en d'autres termes de coût. La maintenabilité est généralement considérée comme une combinaison de critères tels que la changeabilité et la testabilité. Dans cette thèse, nous présentons une approche et une méthode traitant de la quantification de la maintenabilité. Le but est de disposer d'une mesure simplifiant le processus d'évaluation de la maintenabilité et permettant une prise de décision concernant l'investissement en termes de processus de changement à engager pour un logiciel donné. Notre approche est basée sur l'utilisation d'une méthode statistique appelée "plan d'expériences". Les informations en entrée sont un ensemble de versions de logiciels et la sortie est une expression mathématique qui donne une approximation de la maintenabilité en fonction de certaines métriques structurelles de logiciels largement utilisées.

Mots clés : Maintenabilité, systèmes à objet, impact de changement, métriques, plan d'expériences.

Résumé

La maintenabilité du logiciel est un critère de sa qualité mesurant son aptitude à changer avec un minimum d'impact et donc d'effort, de temps et en d'autres termes de coût. La maintenabilité est généralement considérée comme une combinaison de critères tels que la changeabilité et la testabilité. Dans cette thèse, nous présentons une approche et une méthode traitant de la quantification de la maintenabilité. Le but est de disposer d'une mesure simplifiant le processus d'évaluation de la maintenabilité et permettant une prise de décision concernant l'investissement en termes de processus de changement à engager pour un logiciel donné. Notre approche est basée sur l'utilisation d'une méthode statistique appelée "plan d'expériences". Les informations en entrée sont un ensemble de versions de logiciels et la sortie est une expression mathématique qui donne une approximation de la maintenabilité en fonction de certaines métriques structurelles de logiciels largement utilisées.

Mots clés :

Maintenabilité; Systèmes à objet; Impact de changement; Métriques; Plan d'expériences; Sciences expérimentales; Génie logiciel; Métriques de couplage; Programmation orientés objets; S.A.C.I.