UNIVERSITÉ DE MONTRÉAL

ANALYSE ET AMÉLIORATION DU PROCESSUS DE TEST D'UNE ÉQUIPE DE DÉVELOPPEMENT DANS LE DOMAINE DU DIVERTISSEMENT

TIEGO FRANÇOIS-BROSSEAU
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2005

Your file Votre référence
ISBN: 0-494-01326-5
Our file Notre référence
ISBN: 0-494-01326-5

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE ET AMÉLIORATION DU PROCESSUS DE TEST D'UNE ÉQUIPE DE DÉVELOPPEMENT DANS LE DOMAINE DU DIVERTISSEMENT

présenté par: FRANÇOIS-BROSSEAU Tiego
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment acceptée par le jury d'examen constitué de:

M. GUIBAULT François, Ph.D., président
M. ROBILLARD Pierre-N., Ph.D., membre et directeur de recherche
M. LAVIER Sébastien, M.Sc., membre

À Marième,
Cette rose de métal

REMERCIEMENTS

J'aimerais tout d'abord remercier chaleureusement mon directeur, M. Pierre-N. Robillard, qui a su bien me diriger dans mon travail. Ses réflexions ont été indispensables à l'évolution de ce travail.

J'aimerais aussi remercier Nicolas Rioux de chez Ubisoft pour son ouverture d'esprit et pour m'avoir donné la chance de faire ce projet, ainsi que Sébastien Lavier qui a également supporté cette recherche. Un grand merci à Richard, Christian, Jean-François et Franc pour leur participation à ce projet, ainsi que pour leur patience et leur esprit critique. Je remercie particulièrement Stéphane, sans qui je n'aurais probablement pas pu avoir toutes ces données, ou même cette recherche. Merci pour sa patience, son temps et sa mémoire phénoménale ! Je veux également remercier la compagnie Ubisoft en général pour son support.

Merci à ma famille qui m'a encouragé et supporté tout le temps.

Merci à mes amis qui en ont fait autant.

Et finalement, je ne pourrais jamais assez remercier Marième. Elle a été mon support durant tous ces mois.

RÉSUMÉ

Ce mémoire traite de l'analyse et de l'amélioration d'un processus de test dans le contexte du développement des logiciels de divertissement. Plus particulièrement, il s'agit d'une analyse des défauts produits par une équipe de développement et de l'application de nouvelles méthodes pour améliorer les pratiques de test. L'équipe étudiée dans cette recherche est une équipe de développement de moteur et d'outils reliés à l'audio pour les jeux vidéo.

Cette recherche vise à remplir plusieurs objectifs. Le premier est de faire un survol du processus de développement d'un logiciel de divertissement et la place qu'occupe l'équipe étudiée dans ce processus. Le second est d'identifier, via une analyse de la littérature, une métrique centrée sur la classification des défauts permettant d'extraire des informations intéressantes sur le processus de test. L'objectif principal de cette recherche est d'adapter la métrique dans l'environnement informel étudié dans le but de décrire l'état des pratiques de test de l'équipe. Enfin, le dernier objectif visé est d'appliquer de nouvelles pratiques, identifiées à l'aide des résultats de la métrique, pour améliorer le test de l'équipe et de constater les améliorations potentielles à l'aide d'une deuxième analyse avec la métrique de classification.

Ainsi, ce mémoire propose d'abord un survol du processus de développement d'un logiciel de divertissement dans l'entreprise collaborante, Ubisoft Divertissements inc. Une revue globale des différents travaux concernant l'analyse des défauts logiciels sera ensuite présentée, avec une attention particulière pour les métriques de classification. L'implantation d'une de ces méthodes, la classification orthogonale des défauts ou ODC, sera le sujet principal de ce travail.

Les résultats de la première analyse indiquant que les efforts de découverte des défauts sont surtout le fruit du hasard, nous tentons ensuite d'implanter la pratique des

tests unitaires formels parmi les pratiques de l'équipe. Une deuxième analyse démontre que, tandis que les effets primaires attendus par la mise en place de ces nouvelles pratiques ne sont pas visibles, il existe des améliorations qui sont des effets secondaires de cette modification du processus de test.

Cette recherche peut donc être utile à quiconque veut mieux comprendre le processus de développement du jeu vidéo et ses spécificités. Également, celle-ci montre la possibilité d'adapter une métrique formelle spécifique à un environnement informel, ainsi que les bienfaits qui en découlent. Finalement, ce travail confirme que la prescription de pratiques de génie logiciel n'est pas une tâche facile et qu'il n'est pas évident de prouver la présence d'améliorations dans un milieu réel.

ABSTRACT

This thesis presents the analysis and improvement of a software test process in the context of video game development. More specifically, this paper treats of the analysis of software defects produced by a development team and of the application of new testing methods to improve the test practices. The team studied in this research is responsible for the development of the engine and tools related to the audio aspects of video games.

This research has many objectives. The first objective is to expose summarily the entertainment software's development process. The second objective is to identify a classification metric suitable for the extraction of information regarding the test process. This is done by the analysis of current literature. The main objective is to adapt the formal metric identified to the studied informal development environment to describe the current state of the team's test practices. The final objective aims at the improvement of the test process through the application of new test methods and to measure this potential improvement with a second data analysis.

The first section presents the Ubisoft Divertissements' game development process. A global review of the current research on software defect analysis, with an emphasis on defect classification schemes follows. The implementation of one of these schemes, Orthogonal Defect Classification, is this thesis' main subject.

The first analysis of the initial process shows that defect discovery is mostly ad hoc. We implement formal unit testing as a new team practice. A second analysis shows that while the expected primary effects related to this new practice are not visible, there are nonetheless second order improvements associated to the modification of the test process.

This research can be of use to understand the game development process and its original aspects. Also, this research shows the possibility of adapting a formal metric to an informal environment and its positive effects. Finally, this thesis confirms that prescriptive software engineering practices are not easy to implement and that process improvements can be hard to measure in a real development situation.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CP	Chargé de projet (général)
CPGD	Chargé de projet Game Design
CPI	Chargé de Projet Informatique
CMM	Capability Maturity Model
DA	Directeur Artistique
DAA	Directeur Artistique Animation
DARE	Digital Audio Rendering Engine
DT	Directeur Technique
E3	Electronic Entertainment Expo
FTS	Final Title Specification
F/C/O	Function/Class/Object
HP	Hewlett-Packard
IEEE	Institute of Electrical and Electronics Engineers
ODC	Orthogonal Defect Classification
O-O	Object-Oriented

LISTE DES ANNEXES

GLOSSAIRE FRANÇAIS-ANGLAIS

Dans ce document, le lecteur trouvera plusieurs termes de langue anglaise. Ces termes ont été conservés dans leur langue originale, car ils ont été utilisés tels quels dans l'équipe étudiée et proviennent généralement directement de la littérature. Dans ce glossaire, nous traduisons et expliquons ces termes pour éviter toute confusion. Plusieurs mots ou expressions sont également expliqués dans les annexes.

Data Manager : Gestionnaire de données.

External : Externe. Pour cette étude, à l'extérieur de l'équipe étudiée.

Final Title Specification : Spécification finale du titre.

Game Design : Design de jeu. Se dit d'un ensemble d'activités créatives qui ont pour but la définition des règles et de l'environnement d'un jeu.

Internal : Interne. Pour cette étude, à l'intérieur de l'équipe étudiée.

Lead Character Modeler : Modeleur de personnage en chef.

Lead Game Designer : Designer de jeu en chef.

Lead Programmer : Programmeur en chef.

Level : Niveau. Dans le contexte des jeux, un niveau est une subdivision du jeu correspondant à des sous-objectifs qui doivent être remplis par un joueur.

Master : Terme utilisé (sans traduction) pour designer le produit fini et prêt à être soumis aux manufacturiers de consoles ou pour publication.

Planned : Planifié(e)(s). Se dit des activités qui sont effectués de façon prévue par les développeurs

Opportunistic : Opportuniste(s). Dans le contexte de cette étude. Se dit des activités qui sont effectués à l'insu du développeur, par exemple, trouver une erreur par hasard.

Il ne suffit pas de dire, je me suis trompé;
il faut dire comment on s'est trompé

Claude Bernard

Qui pense peu
Se trompe beaucoup

Léonard de Vinci

CHAPITRE I : INTRODUCTION

Le concept de génie logiciel est apparu tout d'abord en 1968 alors qu'on réalisait que les logiciels devenaient plus complexes et que leur conception était plus coûteuse. Il était clair qu'il devenait nécessaire de faire l'ingénierie des logiciels plutôt que de simplement les programmer. Depuis cette époque, on tente de développer des méthodes et des processus qui permettraient d'augmenter la qualité des logiciels, ainsi que de diminuer leur coût et leur temps de développement. Toutefois, étant donné la jeunesse relative de ce domaine et la vitesse à laquelle la technologie évolue, l'ingénieur logiciel n'a toujours pas maîtrisé le développement du logiciel.

Plusieurs méthodes sont proposées et plusieurs recherches sont faites pour comprendre la nature du logiciel et son développement, mais les chercheurs sont encore loin de saisir parfaitement ce qui rend la chose si difficile. L'industrie reflète actuellement cette situation : trop peu d'entreprises se donnent la peine d'investir dans l'implantation de méthodes pour assurer la qualité de leurs produits et de leur processus. De plus, il est reconnu que le type de logiciel développé, aussi appelé domaine, peut influencer l'approche prise pour le développement (Gomaa 2000). Dans ce mémoire, nous présentons les résultats d'une recherche qui analyse le processus d'une équipe développant des logiciels dans un domaine particulier.

Plus particulièrement, nous mesurons et tentons d'améliorer le processus de test d'une équipe de développement dans une entreprise qui produit des jeux vidéo en analysant les défauts de leurs produits. Nous avons plusieurs objectifs pour cette recherche. Tout d'abord, nous tentons de mieux comprendre ce qui distingue le domaine du développement du jeu. Ensuite, nous implantons un système de mesure pour analyser le processus de test de l'équipe en question. Enfin, nous apportons des améliorations à ce processus et vérifions les impacts de ces changements en utilisant le même système de mesure.

Ce travail est divisé en fonction de tels objectifs. Le premier chapitre présente une introduction à l'industrie du divertissement électronique, ainsi qu'une explication du processus de production du jeu vidéo chez Ubisoft Divertissements inc. et du processus de développement de l'équipe étudiée. Le deuxième chapitre est consacré à une revue de la littérature traitant des défauts logiciels et à la présentation de la technique retenue pour l'étude. Le troisième chapitre présente les résultats obtenus lors de la première partie de l'expérimentation. Comme nous tentons d'améliorer le processus de l'équipe, le quatrième chapitre présente les nouvelles méthodes et pratiques qui ont été mises en place. Finalement, le cinquième chapitre montre les résultats de l'expérience à la suite des améliorations du chapitre précédent. Une courte critique des résultats et de l'expérience suivra. Le lecteur familier avec les concepts de base en génie logiciel devrait comprendre facilement la plupart des concepts de cet ouvrage.

CHAPITRE II : LE CONTEXTE INDUSTRIEL ET L'ENVIRONNEMENT EXPÉRIMENTAL

L'industrie du jeu vidéo se distingue des autres industries de développement de logiciels de plusieurs manières, il est par conséquent important de définir l'environnement expérimental de la recherche. Ce premier chapitre est donc dédié à l'explication de plusieurs aspects de la production de jeux. En l'occurrence, il est question tout d'abord de la situation de l'industrie concernée dans la société, ainsi que de quelques éléments culturels spécifiques à celle-ci. Sont ensuite exposés les concepts généraux liés au développement des jeux vidéo avec une explication sur le cycle de vie du développement d'un jeu, ainsi que la structure organisationnelle d'un projet. Puis, l'équipe étudiée est présentée, ainsi que son rôle dans la compagnie et les produits développés par celle-ci. Finalement, le processus général utilisé par l'équipe est analysé avec une emphase particulière sur le processus de test, dont la mesure est le sujet principal de cette recherche.

2.1 L'industrie du jeu vidéo

Avec un chiffre d'affaire maintenant supérieur à celui du cinéma au États-Unis et une importance économique grandissante au niveau international, l'industrie du jeu vidéo devient un joueur intéressant dans le domaine du divertissement. Ceci augmente, par le fait même, la place occupée par les logiciels dans notre vie courante. Comme les logiciels des autres domaines, le jeu vidéo ne cesse de se complexifier et il en va ainsi pour son processus de production.

À ses débuts, quoique « vieille » de seulement 25 ans, l'industrie abritait des équipes de développement moins élaborées qu'aujourd'hui. Effectivement, dans bien des cas, le produit était développé par un seul individu qui s'occupait à lui seul de la programmation, du *Game Design* et du contenu artistique. Les coûts de production

étaient donc relativement peu élevés et la qualité du produit dépendait exclusivement de l'individu ou du très petit groupe concerné. Avec les années, la vitesse vertigineuse de l'évolution du matériel informatique, couplée à la demande d'un public grandissant, a fait évoluer le jeu informatique à un taux effarant. Si bien qu'aujourd'hui, on retrouve des équipes de développement de plus d'une cinquantaine de personnes, des projets pouvant s'étaler sur plusieurs années et surtout des budgets de plusieurs millions de dollars. Le programmeur solitaire muni de pauvres talents artistiques a cédé sa place à de talentueuses équipes multidisciplinaires. Actuellement, une équipe de développement de jeu n'est donc pas uniquement constituée d'ingénieurs en informatique, mais aussi d'artistes de différents domaines.

Cette expansion rapide, sur quelques années seulement, combinée au fait que la génération la plus intéressée par le développement de ce type de produit est aussi une des plus jeunes, a fait en sorte que les processus de développement sont restés relativement chaotiques, alors que le reste de l'industrie logicielle reconnaît depuis longtemps qu'elle doit réviser sa façon de faire. Évidemment, cette différence de procédés n'est pas seulement due à la jeunesse de l'industrie et de ses participants, mais aussi au produit lui-même qui fait appel à plusieurs métiers très différents et dépendants les uns des autres. La prochaine section présente ces différents métiers tels qu'on les retrouve au sein de la compagnie dans laquelle l'étude a été effectuée.

2.2 Présentation des concepts généraux liés au développement des jeux

Afin de mieux comprendre la place du développement logiciel dans l'élaboration d'un jeu, il importe d'expliquer tout d'abord le cycle de vie d'un produit. Dans cette section, les phases de ce cycle sont d'abord présentées avec une explication sur leurs objectifs. Puis, suivent la structure organisationnelle générale d'un projet et les interactions entre les différentes équipes internes et externes. Il est à noter que ce qui est présenté dans cette section est potentiellement spécifique à l'entreprise étudiée. Il serait

intéressant dans un autre temps d'étudier d'autres compagnies afin de comparer les aspects présentés ci-après. Afin de respecter une certaine confidentialité du processus d'Ubisoft, il a été décidé que certaines informations ne seraient pas divulguées. La section suivante ne peut donc pas exposer parfaitement la situation.

2.2.1 Cycle de vie d'un projet

Selon la théorie, un cycle de vie est séparé en plusieurs phases qui se terminent généralement avec un ou des livrables quelconques (Robillard 2003). Cette section est dévouée à la présentation des phases et des grands livrables d'un projet typique de l'entreprise étudiée. Pour chaque phase, les objectifs des métiers artistiques et informatiques sont présentés distinctement. Le schéma suivant illustre les phases du cycle de production d'un jeu vidéo.

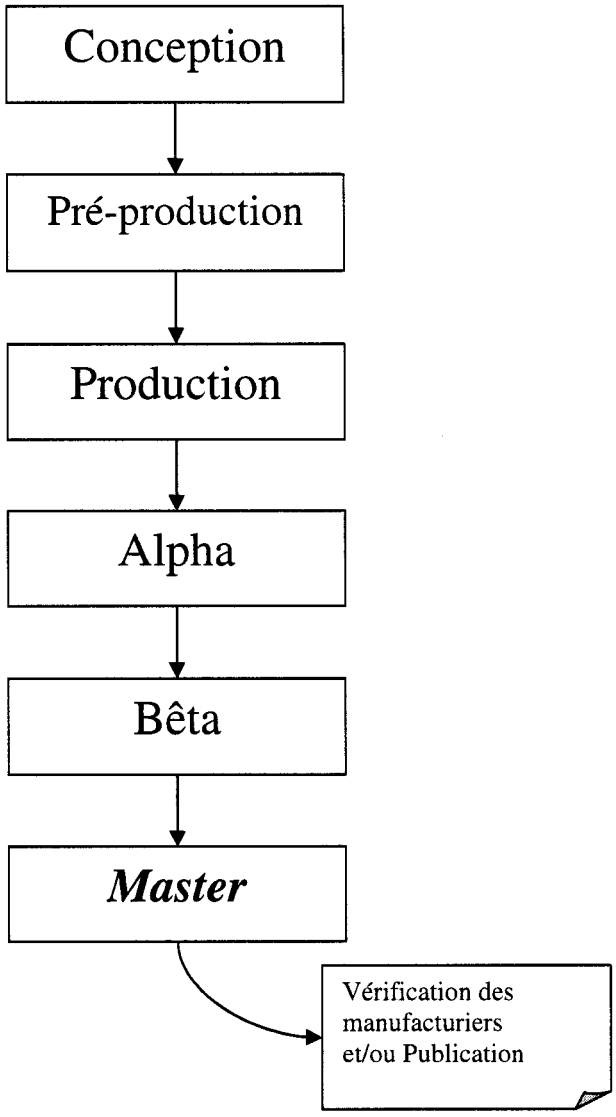

Figure 2.1: Schéma représentant les phases de production d'un jeu

2.2.2 La conception

La conception est la période durant laquelle l'essence du jeu est élaborée. L'équipe du projet est alors formée d'un nombre restreint de personnes, généralement

plus expérimentées, qui déterminent les concepts principaux du jeu au niveau de l'histoire, de la mise en scène, du genre, etc. À ce stade-ci, l'équipe inclut généralement un producteur, un chef de projet *Game Design*, un chef graphique, un scénariste et un chef informatique. Durant cette phase, on analyse donc le public cible, les jeux existants similaires, la compétition à venir, ainsi que toute autre information susceptible d'être utile. Le but premier est de déterminer les objectifs du projet sans nécessairement tenir compte de « comment » ils seront atteints. On développe aussi la « chaîne de montage » du jeu, c'est-à-dire les étapes de la production, ainsi que la planification et une évaluation du coût.

La conception est une période de créativité pour les métiers non informatiques. On élabore les idées directrices du jeu et l'on prépare le document de concept. La conception est aussi marquée par une réunion qui présente le concept initial aux instances hiérarchiques de la compagnie, à l'éditorial et au marketing. Durant cette rencontre, on essaie de régler les questionnements principaux face aux concepts du projet, mais on peut aussi laisser en suspens des interrogations jusqu'à la présentation du prototype. Le but de cette rencontre est aussi de recevoir l'approbation pour le projet, ainsi que des propositions/modifications de fonctionnalités.

Du côté informatique, on doit partir des concepts initiaux obtenus des autres métiers et de l'idée générale brute afin d'analyser les différents requis technologiques. Cette information est souvent floue, car plusieurs concepts généraux du jeu ne sont pas encore coulés dans le béton et ne le seront peut-être pas bientôt. La préoccupation principale de l'informatique à ce stade du projet concerne la réutilisation d'un moteur de jeu. Effectivement, on pourrait séparer les projets de jeu en deux catégories du point de vue informatique : ceux qui réutilisent un moteur existant et ceux qui nécessitent le développement d'un nouveau moteur. La place de l'informatique au début du projet est évidemment plus importante dans le deuxième cas, mais il y a toujours un certain travail à accomplir pour modifier les moteurs existants aux nouveaux projets. Une explication

plus complète de ce concept sera présentée plus bas. Par contre, quelque soit le type de projet, l'informatique doit affronter deux tâches : la mise au point du moteur et le développement des outils qui seront utilisés par les autres métiers dans le contexte du projet. Pour ces deux tâches principales, l'équipe informatique doit déterminer les parties réutilisables, les composantes à construire, ainsi qu'un plan d'intégration. L'équipe informatique analyse et priorise donc les fonctionnalités du moteur et des outils qui doivent être implantées. Cette hiérarchisation des fonctionnalités doit tenir compte aussi du temps qui doit être passé à l'apprentissage des outils en développement. La fin de la conception pour l'informatique, quoiqu'un peu floue en pratique, correspond à la prise des décisions sur certains aspects technologiques importants.

La fin de la phase de conception comporte toutefois une particularité. Effectivement, l'équipe informatique passe à la phase suivante avant les autres équipes du projet. Ceci implique que le travail technique de l'informatique doit commencer alors que plusieurs décisions créatives n'ont pas été prises. On a même décrit la fin de la conception comme étant une barrière psychologique, plutôt qu'un jalon, puisque l'on passe rapidement de la période de réflexion à la période d'essais technologiques.

2.2.3 La pré-production

La nature de cette phase est celle de l'exploration des principes directeurs du concept du jeu. Le but final de la pré-production est de construire un prototype du jeu et de créer, sur papier, la conception élaborée du jeu dans un document appelé le FTS (*Final Title Specification*). Cette phase est aussi caractérisée par un agrandissement graduel de l'équipe du projet pour élaborer la création du contenu et développer le prototype.

Au niveau des métiers artistiques, l'objectif est de préparer l'ensemble du jeu pour la production, qui est la phase suivante. On déterminera donc le design du jeu sur papier, on explorera des styles artistiques, on écrira l'histoire en détail, etc. Le but est de tenter de répondre au plus de questions possibles, de spécifier le jeu le plus possible, afin qu'on ne se pose pas (trop) de questions durant la production. On remarque aussi que plusieurs des décisions prises et idées obtenues sur la nature du jeu lors de la conception sont changées. L'expérience dicte également que l'on oublie tout de même souvent de spécifier plusieurs éléments lors de la pré-production. On compare souvent la pré-production d'un jeu aux préparatifs d'un film, car tout doit être théoriquement préparé pour la production en tant que tel.

La fin de la pré-production est une étape importante, car, en principe, le FTS et le prototype sont complétés et peuvent être présentés au comité d'approbation de projets. Il n'est pas impossible qu'un projet soit annulé à cette étape. On doit aussi être en mesure de produire un plan de production qui concerne les tâches à accomplir, une révision des coûts et des besoins au niveau du personnel. Théoriquement, les fonctionnalités du jeu doivent être aussi coulées dans le béton, mais en pratique, on voit souvent des changements durant la production, à cause de la nature organique du développement de jeux.

Au niveau de l'informatique, cette phase est caractérisée par plusieurs aspects particuliers. Premièrement, comme il a été mentionné pour la conception, cette équipe se trouve à être en avance sur les autres métiers durant la pré-production. Effectivement, l'équipe informatique ayant la tâche d'adapter ou de construire le moteur et les outils pour le prototype, celle-ci se trouve à être à mi-chemin dans sa production alors que les autres métiers terminent la pré-production. Ensuite, on remarquera aussi que le prototype est souvent complètement réutilisé pour la production, plutôt que d'être jeté. Le prototype est donc incrémental, surtout que certaines fonctionnalités seront souvent ajoutées durant la deuxième moitié de la production informatique. Finalement, la pré-

production est souvent qualifiée, et ce par les différents métiers, comme étant une phase particulièrement chaotique.

Dans un dernier temps, il est important de mentionner que l'informatique prépare un rapport qui sera inclus dans le FTS, mais ce rapport n'est pas de nature technique. À la fin de la pré-production officielle, le prototype est présenté à des agents externes au projet, tels que le comité éditorial international, et doit être approuvé par ceux-ci.

2.2.4 La production

Le but de la production est simplement de produire tous les éléments de contenu du jeu, ainsi que les fonctionnalités informatiques. Ceci est effectué à partir de la planification élaborée à la fin de la pré-production, et en principe, les équipes ne devraient pas déroger de celle-ci. En pratique toutefois, il arrive souvent qu'il y ait des modifications sur les outils, le moteur et les concepts du jeu. Comme il a été souligné dans la section sur la pré-production, la création des jeux est une activité créative et donc difficilement contrôlable. La créativité des métiers artistiques se trouve donc à affecter le cours normal du développement logiciel. Bien souvent, on retrouve donc plusieurs coupures et ajouts par rapport à la planification originale. Également, l'équipe du projet grandit encore à la production pour incorporer toutes les personnes qui vont exécuter les tâches de production.

La deuxième étape de la production est le test du produit en construction. À cause de la nature différentiée de cette étape, chacune des phases qui constituent cette étape est présentée en particulier. Il est à noter que ces différentes phases impliquent également des jalons officiels.

2.2.5 L'Alpha

Le but de la phase Alpha est d'avoir un produit qui est presque complètement codé et dont le contenu est relativement bien avancé en concordance avec le FTS. Autrement dit, il manque des éléments mineurs, mais l'essentiel du concept et du logiciel est là. Les critères ne sont pas exacts, mais on doit pouvoir jouer au jeu sans qu'il y ait (trop) de problèmes sérieux. Avant la fin de cette phase, des tests avec des groupes de focus sont organisés pour parfaire le jeu au niveau de son design. Les tests plus officiels avec les testeurs internes commencent déjà durant cette phase. Pour ce qui est du test au niveau de l'équipe informatique, il semblerait qu'il n'y ait pas de plan de test formel. Les objectifs de cette étape sont de montrer le produit dans son état avancé pour l'approbation de continuité, de présenter du contenu au marketing et de commencer les tests finaux.

2.2.6 La Bêta

Durant la phase Bêta, l'équipe est censée travailler avec un produit relativement fini. Les équipes les plus actives sont celles de l'informatique qui corrigent les problèmes techniques et celle des testeurs. Les testeurs forment une équipe indépendante d'individus qui travaillent à trouver des erreurs sur l'entièreté du produit. Les métiers artistiques corrigent les erreurs trouvées dans leur travail. À cette étape, l'utilisation d'un système de base de données pour les anomalies[1] est une pratique nécessaire, leur nombre dépassant souvent 10 000. Cette base indique plusieurs informations par rapport aux anomalies trouvées, telles que le numéro de version, l'état de l'erreur, etc. La durée et l'efficacité de la phase Bêta dépend beaucoup de l'expérience des testeurs. On continue aussi avec les groupes de focus. Les objectifs de cette étape sont généralement

[1] À noter que les anomalies enregistrées ne sont pas toutes de nature technique, parfois, on notera des erreurs de données artistiques, telles que le manque de polygones, des couleurs erronées, ou encore des erreurs liées au design de jeu.

de « geler » les données de contenu et d'entrer dans la dernière phase de déverminage et de contrôle de qualité.

2.2.7 Le Master

Le master est théoriquement le dernier livrable pour le projet. Durant cette dernière étape, on finalise donc la correction des erreurs, mais on fera aussi du « polissage », spécialement au niveau informatique. L'équipe informatique travaillera sur l'optimisation surtout à ce moment et on ira même jusqu'à la modification et l'ajout de certains éléments (peaufiner l'intelligence artificielle, etc.).

Après cette étape, le produit « final » doit être approuvé par les instances décisionnelles de la compagnie. Ensuite, selon la plateforme d'exécution visée, les produits doivent passer par les manufacturiers de consoles (Sony, Nintendo, etc). Si le produit n'est pas accepté à une de ces étapes, l'équipe du projet sera appelée à faire des changements jusqu'à ce que le produit soit approuvé. Le produit qui sera publié, aussi appelé Gold Master, est ensuite envoyé au manufacturier.

2.3 Structure générale d'une équipe

Plusieurs expertises techniques et artistiques sont nécessaires dans la création d'un jeu vidéo. Répartis dans différentes équipes internes ou externes à un projet, ces différents métiers sont nécessairement appelés à interagir entre eux. Dans cette section, les différentes équipes qui composent l'effectif d'un projet sont présentées, puis l'interaction entre ces équipes, tant au niveau artistique qu'au niveau informatique, est expliquée.

Une bonne façon de représenter l'équipe d'un projet est par son organigramme. La figure suivante présente l'organigramme d'un projet générique, à la suite de laquelle les explications sur les équipes et les postes importants de cette structure sont présentés.

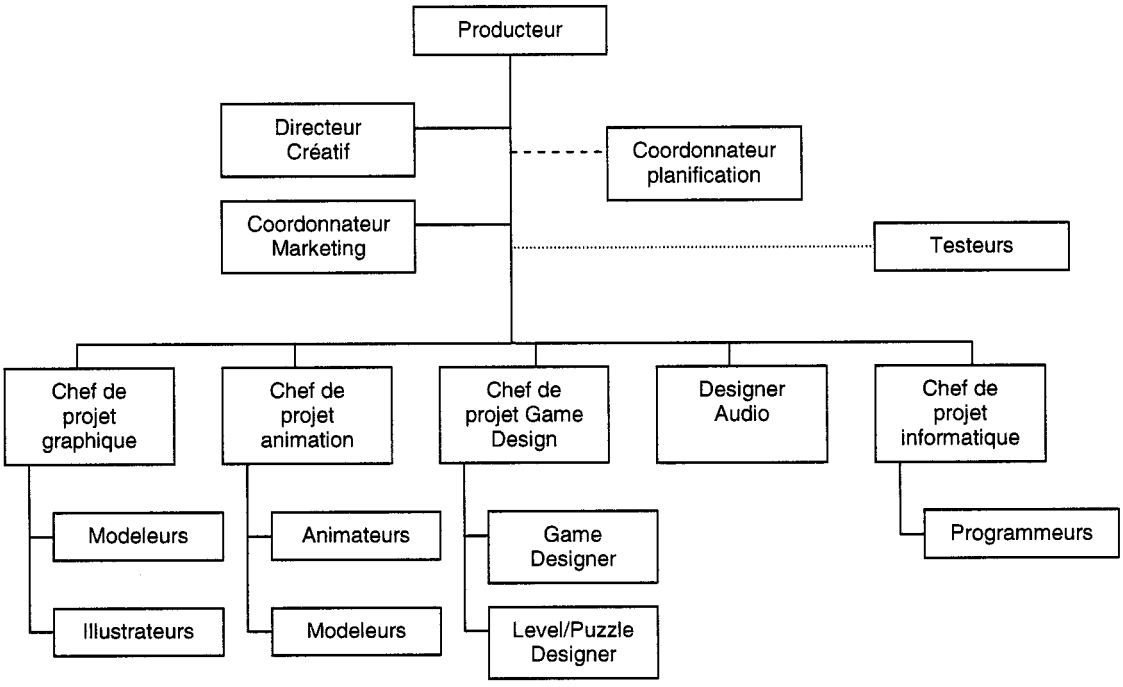

Figure 2.2: Organigramme générique d'un projet pour la compagnie étudiée

2.3.1 Le producteur

Le producteur est à la tête du projet. Ce rôle est actif dès la conception de la vision originale du jeu et doit être présent tout au long du projet. Au début, durant la phase de conception, le producteur travaille sur le concept du jeu, mais il se distancie de ce rôle par après pour prendre le rôle de gestionnaire, et ce pour le reste du projet. Il doit guider l'équipe à travers le projet et l'encourager. Il est au courant de tout se qui se passe dans toutes les équipes et doit être capable de répondre aux questions des différentes personnes du projet ou de les diriger vers la personne du projet qui détient

l'information. C'est aussi le représentant à l'extérieur du projet et il doit « défendre » le projet auprès du marketing et l'administration. D'une façon plus générale, le producteur s'occupe de la communication à l'intérieur et à l'extérieur de l'équipe. À cause de cette responsabilité, le producteur doit aussi être capable de parler le langage de tous les métiers. Finalement, les objectifs du producteur sont de faire un bon jeu, à temps, à l'intérieur des limites du budget, avec une équipe qui a le goût de continuer après (et durant) le projet.

2.3.2 Le directeur créatif

Ce rôle est relativement récent chez Ubisoft, et on le voit apparaître dans l'industrie du jeu en général. Il est responsable du contenu du jeu et est présent durant tout le projet. Hiérarchiquement, on pourrait le positionner au dessus de toutes les équipes, mais sous le producteur, avec un bémol sur l'équipe informatique, car il ne valide pas leur travail en tant que tel. Le travail du directeur créatif est plus important dans les phases de conception et de pré-production, pendant lesquelles il travaille à l'élaboration et au maintient de la vision globale du jeu. Durant les phases subséquentes, il testera le jeu au niveau logique et du design de jeu. Son travail est essentiellement de nature créative car il valide et conseille les équipes s'occupant de la création du contenu en fonction de la vision artistique du jeu.

2.3.3 Le marketing et la planification

On retrouve également dans un projet des coordonnateurs marketing et de planification. Les rôles de ces postes sont clairs, l'un s'occupe du suivi de la planification et l'autre du marketing du produit en développement. Les fonctions exactes de ces rôles ne sont pas détaillées, mais il est important de mentionner que leur présence

peut influencer le cours du projet, surtout celle du coordonnateur marketing. Effectivement, ce dernier demande souvent des images ou des programmes de démonstration à des fins de publicité, ce qui peut chambarder la planification.

2.3.4 L'équipe informatique

Au niveau des équipes d'un projet, il y a premièrement l'équipe informatique. Le premier membre de cette équipe est le chef de projet informatique (CPI). Il est le responsable de l'aspect informatique d'un jeu et est aussi connu sous le nom de *Lead Programmer*. En tant que chef d'équipe, ce rôle doit coordonner les efforts de son équipe, faire le lien avec les autres chefs de projet et le producteur, ainsi que l'accomplissement de certaines tâches reliées à l'informatique (programmation, design, test). On pourrait donc séparer ce rôle en deux parties : gestionnaire d'équipe et ingénieur informaticien. Nous avons remarqué que la proportion de travail dans ces deux rôles variait selon l'individu et le projet. Le CPI accomplit donc des tâches d'analyse et de design, de programmation et de test ainsi que des tâches de planification. C'est aussi lui qui s'occupe de l'intégration du travail des différents programmeurs et qui est au courant de l'avancement global du projet au niveau informatique. Étant plus expérimenté, le CPI agira aussi comme référence technique aux autres programmeurs, c'est pour cela aussi qu'il doit bien souvent vérifier les travaux de ceux-ci. En tant que spécialistes en informatique, les programmeurs sont les analystes, les designers, les codeurs et les testeurs du domaine informatique d'un projet. Il s'agit là d'un poste bien connu de notre domaine.

2.3.5 L'équipe de design de jeu

Cette équipe est composée de *game designers*, de *levels*/puzzles designers et d'un chef de projet *Game Design*. Le travail de cette équipe est de créer et de maintenir

le contenu du jeu. Cette deuxième activité est relativement importante dans le processus. Effectivement, à cause du caractère créatif des activités reliées au design du jeu, des modifications sont souvent apportées à ce design, influençant le cours du développement du projet. L'équipe de *Game Design* est donc une force motrice dans le développement d'un jeu, car ce sont ses besoins qui font appel aux autres métiers. Les gens rencontrés ont comparé cette équipe au conducteur d'une automobile, les passagers étant les autres métiers. Cette analogie illustre que l'équipe de design de jeu conduit le projet, mais est aussi influencée par les passagers qui proposent des idées et imposent des contraintes. Au niveau des postes, il y a le chef de projet *game design*, aussi connu sous le nom de *Lead Game Designer* dans l'industrie. Ce rôle est responsable de l'équipe de la création du contenu du jeu au niveau de l'histoire, des règles du jeu, des expériences vécues par le joueur, etc. Le chef doit, comme le directeur créatif, connaître le jeu parfaitement pour pouvoir diffuser l'information aux autres équipes. Il s'occupe de la validation et de l'intégration des divers éléments produits par son équipe, assiste et contrôle les groupes de focus et fait du « polissage » général sur le produit en développement. La planification de ces activités est très difficile car l'inspiration, la force motrice de ce groupe, est par définition aléatoire. La qualité du jeu est aussi vérifiée par le chef *Game Design* et ce, de façon informelle, car celle-ci est de nature « organique » (émotions, etc.). Les *levels*/puzzles designers, quant à eux, créent les différents environnements virtuels dans lequel se passe l'action ou conçoivent les problèmes que doivent résoudre les joueurs. Ces métiers sont plutôt de nature artistique.

2.3.6 L'équipe animation

Cette équipe produit les concepts de personnages, leurs modèles, ainsi que leurs animations. Comme les autres chefs de projet, le chef de projet animation est responsable de la gestion de son équipe d'animation et fait le lien avec les autres chefs. Il s'occupe aussi de la planification du travail et valide le travail fait par les animateurs.

Il travaille entre autres avec les concepts des *game designers* afin de connaître le contenu d'animation à créer. Certains projets auront aussi un *lead character modeler* qui s'occupera des aspects modélisation et artistique des personnages principaux, ceux-ci demandant un travail plus approfondi. Finalement, les animateurs sont des artistes qui produisent les animations des personnages et créatures du jeu.

2.3.7 L'équipe modélisation

L'équipe de la modélisation s'occupe de tout ce qui concerne l'illustration, la modélisation et tout autre élément visuel qui n'est pas un personnage. Elle est aussi connue sous le nom d'équipe graphique. Le rôle du CP dans cette équipe est de faire la planification, la supervision et la gestion de l'équipe et du travail à accomplir. Les modeleurs sont responsables de la création des modèles en trois dimensions qui seront intégrés dans le jeu. Finalement, les illustrateurs sont des artistes qui travaillent à l'élaboration des concepts visuels du jeu en créant des esquisses qui représentent les éléments importants de la vision artistique du jeu.

2.3.8 Les designers de son

Les designers de sons sont ceux qui construisent la « bande sonore » du jeu. Ils n'enregistrent pas les sons ou les musiques, ils décident plutôt de ce qu'ils veulent intégrer dans le jeu, demandent les sons ou la musique au studio son et construisent les bases de données sonores nécessaires aux jeux. La tâche de l'enregistrement des sons ou de la composition des musique peut être fait par le studio son de la compagnie, qui est une entité externe au projet, ou peut être sous-traité à une autre compagnie. Les designers de son sont aussi ceux qui demandent des fonctionnalités au niveau du moteur audio et de ses outils.

2.3.9 Les testeurs

À la tête de cette équipe, on retrouve le chef de projet de test qui détermine un plan de test, ainsi qu'une planification des tâches et des ressources humaines pour les tests. Appartenant au groupe de contrôle de qualité, les testeurs sont des gens qui sont affectés au test d'un jeu alors que celui-ci est en phase bêta et parfois même alpha. En fin de projet, il arrive parfois qu'il y ait des testeurs 24 heures par jour. Les testeurs n'ont généralement pas de formation informatique ou sont des gens qui désirent accéder à des postes artistiques en passant par cet emploi.

2.3.10 Les autres équipes

Finalement, il y a toutes les équipes qui sont externes au projet, mais qui ont pour client les projets de jeux. Ces équipes sont des équipes de développement logiciel qui travaillent sur le développement de composantes qui sont réutilisées par la plupart des projets. Dans cet ouvrage, nous étudions l'une de ces équipes externes. Le fonctionnement et l'interaction de cette équipe avec les équipes de projet sont décrits plus loin. Il était toutefois important de décrire la structure d'un projet et son fonctionnement pour comprendre le contexte dans lequel se trouvent les informaticiens. Effectivement, comme les équipes externes ont toujours au moins un projet comme client, elles sont assujetties aux mêmes types de demandes et de contraintes qu'une équipe informatique de projet.

2.4 Explication de l'interaction entre les équipes

Afin d'en arriver à un produit intéressant et divertissant, les différentes équipes d'un projet doivent se concerter. Dans cette section, certaines particularités et certains problèmes du contexte de développement d'un jeu sont expliqués.

L'équipe informatique a pour client le reste de l'équipe du projet. Leur mandat est de développer le moteur de jeu de façon à ce qu'il supporte les idées et les concepts des artistes et les outils de façon à ce que les artistes produisent les données qui fonctionnent avec le moteur. Le produit qui est développé comporte donc deux sortes de processus qui sont concurrents et interdépendants, soit le développement des logiciels et la création du contenu. Comme il a été mentionné précédemment, les phases de développement logiciel doivent toujours être en avance sur le projet, car les artistes ont besoin de la technologie pour créer et/ou faire fonctionner leurs idées. Le développement des logiciels est donc définitivement sur le chemin critique du projet.

Certaines équipes de développement logiciel sont externes aux projets. Ces équipes peuvent développer des composantes ou des outils qui sont utiles à plusieurs projets. Leur mandat est donc assez similaire à celui des équipes internes. Toutefois, pour ces équipes, chaque projet de jeu peut être un client qui demande des fonctionnalités différentes. Ces équipes externes doivent donc gérer l'ordonnancement de leurs développements afin de rencontrer les objectifs de tous.

Dans les deux cas, le même problème peut survenir. À cause de la nature créative des métiers clients des équipes informatiques, il est possible que le développement de nouvelles fonctionnalités ou l'évolution du concept d'un jeu fasse changer les requis, et ce, de façon potentiellement drastique. Tandis que le problème du changement des requis n'est pas spécifique au développement de jeu, il peut être particulièrement présent dans ce domaine à cause de la nature créative du produit, mais également à cause de la proximité des clients aux développeurs.

Un autre problème possible relève du fait que ces deux types de métiers n'aient pas le même langage. On pourrait comparer ceci au problème classique de la définition des requis avec un client ayant peu de connaissances informatiques. Heureusement, la

culture locale de l'entreprise, voire même celle liée aux jeux, permet de réduire ces écarts de compréhension.

Suite à l'analyse du processus local, nous avons ensuite considéré les différents sujets de recherche possibles qu'offrait ce domaine particulier. Cette analyse a permis de constater que les produits développés sont difficiles à tester et qu'il serait intéressant d'en comprendre davantage sur le processus de test de l'entreprise. C'est donc à partir de cette prémisse qu'il a été décidé d'analyser les erreurs informatiques d'une équipe en particulier. L'environnement de développement qui est étudié est présenté dans la section suivante.

2.5 Présentation de l'environnement expérimental

Certaines contraintes organisationnelles obligeant, nous avons été restreints à l'analyse d'une équipe de développement externe. Mais comme mentionné précédemment, une équipe externe se trouve à répondre au même type de demandes qu'une équipe de projet et développe des produits ayant les mêmes caractéristiques qu'un jeu. Ainsi, l'équipe étudiée est celle qui développe le moteur audio et les outils reliés au design du son. Afin de bien comprendre le contexte de l'expérimentation, une description sommaire de cette équipe et des produits qu'elle développe est présentée.

L'équipe de développement son, aussi appelé D.A.R.E. (*Digital Audio Rendering Engine*) est composée de cinq membres employés à temps plein, dont un CPI et quatre développeurs. L'expérience de travail dans l'industrie de chaque membre en date du début de l'expérimentation était de trois années à sept années, soit au sein de l'entreprise même ou dans un domaine connexe. La formation des membres est en informatique de niveau universitaire. Finalement, il est à noter que l'environnement physique de l'équipe est une salle à aire ouverte. L'équipe est regroupée dans une partie de cette salle ; elle côtoie donc d'autres équipes de développement. Tous les membres

sont très accessibles l'un à l'autre, n'ayant au plus que des tables qui les séparent. Finalement, les clients de l'équipe peuvent également venir à leur aise pour poser des questions et pour rapporter des défauts.

L'équipe son développe plusieurs produits qui sont utilisés au sein de la compagnie. Les deux produits les plus importants sont le moteur son, composante qui est intégrée aux jeux, ainsi que *SoundEditor*, un logiciel permettant aux designers de son de créer les banques de sons qui seront utilisées par un jeu. L'équipe développe également d'autres outils qui permettent, entre autres, de tester le moteur ou de convertir des fichiers, mais l'essentiel de l'effort est concentré sur le développement des deux produits principaux.

Ces deux produits sont d'une grande importance pour les projets de jeux. Premièrement, le moteur son est utilisé à l'intérieur même du produit final. L'équipe est donc assujettie à des contraintes de temps similaires à celle de leur client. Ensuite, l'éditeur de son est l'outil de base qu'utilisent les designers de son dans leur travail. Ceux-ci produisant du contenu fort important pour les jeux, il est donc impératif que l'outil soit fonctionnel et offre toutes les possibilités créatives aux artistes du son. Le développement de cet éditeur est donc également affecté par la planification des clients. En ajoutant à l'équation le fait que l'équipe a plusieurs clients avec des cédules de développement différentes, il devient évident que les efforts déployés par l'équipe sont critiques et affectent la planification des projets clients.

De plus, d'un point de vue plus technique, il est utile de noter que le moteur est développé pour plusieurs plateformes différentes, dont les consoles de jeu actuelles, ainsi que les ordinateurs personnels et toutes leurs configurations possibles. L'éditeur son, quant à lui, est seulement utilisé sur ordinateur. Finalement, il faut mentionner que les produits de l'équipe son existent depuis près de sept ans, ce qui constitue un patrimoine informatique important.

2.6 Processus actuel de l'équipe étudieé

Dans cette section, une brève description du processus de développement de l'équipe est présentée. Cette description est sommaire et qualitative et permet de mieux comprendre le contexte de l'expérimentation. Le processus en général et le processus de test en particulier font l'objet de cette section.

Vu que le moteur son et ses outils existent depuis plus de sept ans, le développement de ces produits relève en fait de la maintenance améliorative et corrective. Effectivement, les efforts de l'équipe sont concentrés d'abord sur le développement de nouvelles fonctionnalités, la réingénierie de certaines parties et la correction d'erreurs. En pratique, l'équipe a décidé que chaque développement de fonctionnalité serait pris en charge par un seul individu. Ainsi, chaque développeur est responsable du design, de l'implantation et du test d'une fonctionnalité. La conséquence de ceci est que le produit ne comporte pas de cycle de développement en tant que tel, il s'agit plutôt d'une série de cycles concurrents. Le processus de développement peut donc être décrit en suivant l'évolution d'une seule fonctionnalité.

Les fonctionnalités à ajouter sont généralement des demandes faites par un projet. Ces demandes peuvent être faites au niveau du moteur ou de l'éditeur (ou tout autre produit, mais ce sont généralement les deux produits impliqués). Une demande est assignée à un développeur en particulier. Lorsque le développement d'une fonctionnalité est entamé, une partie de l'équipe tente tout d'abord de bien cerner les requis de la fonctionnalité. Ceci est fait de façon généralement informelle et non documentée. Ensuite, un temps de développement est prévu et inscrit dans la planification des tâches.

Le développeur élabore ensuite le design de la fonctionnalité. Puis, selon la complexité du développement, il est possible qu'une réunion de révision du design soit faite. Lorsque le design est terminé, le développeur commence à implanter la

fonctionnalité. Il est possible que le développeur fasse des tests informels durant l'implantation ou à la fin de celle-ci. La section suivante traite des tests plus particulièrement. Suite à l'implantation, une inspection de code est effectuée par un autre développeur. Les inspections sont faites à un moment opportun, ce qui peut être avant les tests ou après les tests. Lorsque la fonctionnalité est jugée assez stable, elle est intégrée dans la branche principale de l'équipe. De temps à autre, une version officielle du moteur et des outils est publiée avec une ou plusieurs nouvelles fonctionnalités et/ou corrections. Ces versions sont des instances de la branche principale. Tout au long du développement, le développeur a accès à un logiciel de gestion de configuration dans lequel il possède une branche de travail. Cette branche est sauvegardée, peut être mise à jour régulièrement et éventuellement réintégrée à la branche principale.

Il est important de donner quelques précisions par rapport à ce dernier paragraphe, le processus général étant réellement assez informel. Premièrement, la notion de design est assez mal définie : aucune notation officielle ou format n'est utilisée pour le design. La notion du design est donc très personnalisée. Il arrive aussi que l'implantation et le design soient effectués en même temps. Du côté des révisions de design, celles-ci sont très peu fréquentes et sont souvent informelles. Finalement, les inspections de code existent sous la forme d'une simple relecture du code par un individu autre que le développeur, aucune autre technique n'est utilisée.

Au niveau du processus de test, si celui-ci est analysé plus en profondeur, il apparaît qu'il n'y ait pas d'activités de test en tant que tel. Effectivement, le test de chaque fonctionnalité est laissé à la discrétion du développeur. Ainsi, la plupart du temps, celui-ci se contentera de compiler son code et d'exécuter de simples tests du point de vue fonctionnel. Aucune technique de test n'est donc officiellement en place. Lorsque le développeur juge que son travail est fonctionnel, il en avise l'équipe et intègre son travail dans la branche principale. Comme mentionné précédemment, une inspection de code sera éventuellement faite sur ce code. Il y a tout de même certaines

mesures de test au niveau système qui sont en place. Effectivement, l'équipe possède un logiciel qui permet, avec l'usage de scripts, de simuler des appels au moteur, permettant d'exécuter des scénarios d'utilisation typiques. Tandis que ceci est un bon moyen pour construire et exécuter des tests de système, ce logiciel n'est pas utilisé à son plein potentiel. Des scénarios simples et pas nécessairement représentatifs de l'utilisation du moteur sont exécutés par l'outil. La conclusion à dégager de cette brève analyse est qu'aucun processus de test officiel n'était en place dans l'équipe et que la qualité des tests était dépendante de l'individu et non contrôlée.

Cette analyse du processus de test qualitative est intéressante, mais incomplète. Afin de mieux comprendre cet aspect du processus, nous mettons en place dans cette étude un système de mesure des défauts logiciels. Dans le chapitre suivant, une revue des recherches concernant les défauts logiciels est présentée pour mieux cerner ce qui peut être retiré de ce type d'information.

CHAPITRE III : ÉTUDE BIBLIOGRAPHIQUE ET MÉTHODE UTILISÉE

Dans ce chapitre, il sera question tout d'abord de quelques définitions associées aux erreurs logicielles et de leur place dans le cycle de développement. Ensuite, ce que la littérature propose au niveau de l'analyse des erreurs est présenté. Dans un troisième temps, la méthode que nous avons sélectionnée pour analyser le processus de l'équipe est proposée. Finalement, l'application de la méthode à l'environnement de l'expérience et le déroulement de celle-ci sont présentés.

3.1 L'unité de mesure

L'erreur est humaine, certes, mais c'est l'évolution de ces erreurs qui mènent à l'introduction de défauts dans un produit logiciel. Effectivement, le « cycle de vie » d'une erreur commence chez l'être humain lorsque celui-ci fait une omission, une faute de raisonnement, etc. Ces erreurs, lorsque traduites dans un artéfact appartenant au processus de développement logiciel, deviennent des défauts de requis, de conception et de programmation. Ainsi, lorsque ces défauts sont implantés dans le produit d'une manière quelconque, on se retrouve avec un produit comportant des anomalies. Une anomalie est une déviance d'un produit ou d'une documentation par rapport à ce qui est attendu (IEEE 1994). Ce sont ces anomalies, mieux connues sous le nom anglais de « bug », qui sont la source de nombreux maux de tête des développeurs et de leurs clients. Il est à noter qu'une anomalie n'est pas nécessairement issue d'une erreur, mais toutes les erreurs conduisent à des anomalies (IEEE 1994).

Figure 2.1 : Schéma de l'évolution de l'erreur

Comme l'humain est impliqué dans chaque activité, un défaut peut être introduit à plusieurs occasions dans le cycle de développement. Que ce soit un défaut dans les requis ou une mauvaise implantation d'une condition dans le code, toute activité est susceptible de générer des défauts. Par le fait même, chaque défaut pourrait théoriquement être découvert après son introduction, à différentes occasions tout au long du processus de développement, quoiqu'en pratique il soit impossible de découvrir tous les défauts.

Un bon processus définit donc des activités de vérification et de la validation. Ceci permet de réduire les coûts de découverte et de correction et d'éviter le plus possible que les clients soient les découvreurs des anomalies. Parmi ces activités, on retrouve les révisions de requis, les inspections de code, les tests unitaires, etc. Il a été observé que les organisations qui donnent de l'importance à ce type d'activités doivent investir beaucoup afin de découvrir et corriger les défauts (Hetzel 1993). D'une façon générale, on peut reconnaître que les défauts peuvent influencer bien souvent le cours et la complétion d'un projet de développement.

En fait, si l'on analyse la définition de ce qu'est un artéfact dans un processus logiciel, on réalise que le défaut répond bien à cette définition, car il est le produit d'une activité. Si ces défauts sont enregistrés et analysés, il pourrait être possible d'extraire de l'information intéressante sur le processus. En général, il est reconnu que l'utilisation de métriques adaptées aux besoins d'une organisation amène celle-ci à améliorer son processus de développement (Grady 1994, Iversen 2000). Alors, pourquoi ne pas

analyser les défauts créés pour comprendre comment le développeur de logiciel travaille?

Une analyse de la littérature montre en fait que plusieurs chercheurs et praticiens se sont penchés sur l'analyse des défauts logiciels. Tandis que la pertinence des résultats peut être assez variable, ce qui est normal lorsque l'on fait de la recherche, bon nombre d'analyses sont assez intéressantes. Dans la prochaine section, différentes méthodes qui ont pour base l'analyse des défauts sont présentées.

3.2 Études sur les défauts

Plusieurs recherches ont été effectuées sur les erreurs, les défauts et les anomalies. Dans cette section, quelques études et méthodes qui sont centrées autour des défauts sont présentées. Ensuite, des méthodes de classification des défauts sont exposées, vu que c'est une méthode de ce type que nous avons retenu. Finalement, il est important de noter que nous n'avons pas étudié les recherches se rapportant aux erreurs humaines, car ce type d'étude se rapproche plus de la psychologie.

3.2.1 Quelques métriques classiques

Certaines métriques reliées aux défauts reviennent régulièrement dans les études. Dans cette section, quelques-unes de ces métriques qui servent généralement à évaluer simplement le processus sont présentées.

Une première métrique utile est la densité de défauts. Cette métrique indique la quantité de défauts trouvés par une unité de quantité donnée. Généralement, cette unité est la ligne de code, mais il est également possible de la calculer avec d'autres mesures, telles que les points de fonction. Cette métrique est souvent utilisée pour représenter la

fiabilité du code développé, voire même l'efficacité du processus (Eickelmann 2003, Fredericks 1999). Certaines recherches utilisent d'ailleurs cette métrique pour valider une amélioration du processus (Malaiya 2000, Neufelder 2000).

Une autre métrique qui est également utile est le taux de découverte des défauts. Exprimée en nombre de défauts en fonction d'une unité, généralement de temps, cette métrique est souvent utilisée pour évaluer l'effort des tests et peut également être utilisée pour constater des améliorations de processus. L'unité peut être le temps de test, le nombre de cas de test, etc.

3.2.2 Ce qu'ils ont appris en combattant des défauts

Une récente étude menée de front par plusieurs centres académiques auprès de plusieurs entreprises a permis aux chercheurs de regrouper quelques statistiques réelles concernant les défauts logiciels (Basili 2002). Dans cet article, les nombreux co-signataires soulignaient plusieurs concepts relatifs aux défauts qui sont généralement reconnus par la communauté académique du génie logiciel. Le but de cet article était de voir si ces règles s'appliquaient dans des cas pratiques. Comme plusieurs de ces règles ont un lien intime avec cette recherche, les plus pertinentes à notre sujet sont présentées.

La première règle de cette recherche dit que l'effort, et par le fait même le coût, pour trouver et corriger un défaut augmente très rapidement, voire exponentiellement, avec l'avancée du projet. La recherche confirme que ceci est généralement vrai, quoique cette règle dépend aussi de l'évolution de l'effort déployé en fonction du temps et de la sévérité du défaut, mais il en demeure pas moins que ce concept est important, car il lie le coût d'un défaut à son temps avant la découverte. Une organisation devrait donc construire son processus en ayant pour but de réduire ce temps de découverte.

La deuxième règle intéressante dit qu'une bonne partie de l'effort (variable selon l'auteur) consacré à un projet est dépensée sur des tâches de ré-ingénierie évitables. Autrement dit, une bonne partie du temps d'un projet est utilisé pour corriger des erreurs. Ce constat s'imbrique bien avec la première règle qui a été énoncée.

Une troisième règle dit que 80% des défauts proviennent de 20% des modules. Les chercheurs s'entendent généralement bien sur ces proportions et les praticiens également. Ce concept, même s'il ne sera pas mesuré dans cette étude, sera fort utile pour développer des méthodes de tests dans l'organisation concernée.

Finalement, on dit qu'une grande proportion des défauts peut être trouvée par des inspections de toutes sortes. Plusieurs études démontrent effectivement que plus de 50% des défauts peuvent être trouvés par des activités de révision tout au long du cycle (et non seulement par des inspections de code) (Basili 2002). Ceci peut donner un étalon intéressant pour quantifier l'efficacité d'un processus de test que l'on mesure. D'autres règles ont également été identifiées dans cet article, mais nous nous limitons à ces concepts pour l'instant.

3.3.3 Les modèles de croissance de fiabilité

Un autre aspect intéressant contre lequel les chercheurs et les praticiens se heurtent souvent est le moment à partir duquel on doit arrêter de tester. Effectivement, plus la phase de test avance, moins il y a de défauts (Wood 1997), causant un effort de test superflu.

Par rapport à ceci, plusieurs chercheurs ont tenté de déterminer un moyen pour connaître statistiquement le nombre de défauts qui restent à l'intérieur d'un produit afin de mieux contrôler les efforts de test. Ces méthodes s'appellent les modèles de

croissance de fiabilité. Comme ces méthodes sont basées sur des calculs relativement complexes, les détails de ses calculs ne seront pas présentés dans cet ouvrage.

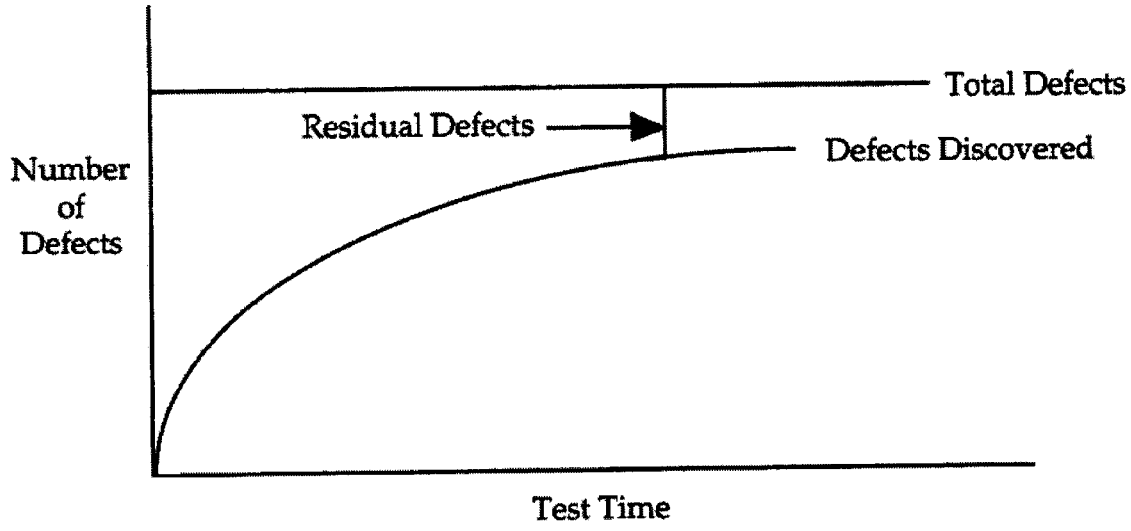

Figure 3.2: Courbe théorique représentant le nombre de défauts trouvés et les défauts résiduels[2]

Les modèles de croissance de fiabilité sont basés sur le fait qu'un produit contient, à un moment donné, un certain nombre de défauts. Si cela était possible, on pourrait soustraire du nombre total de défaut d'un produit, le nombre de défaut qui ont été trouvés par l'effort de test, pour savoir le nombre de défauts restants (Figure 3.2). Évidemment, ceci est impossible. Toutefois, il est possible de construire un modèle qui permet de corréler certains paramètres pour approximer ce nombre total de défauts et par le fait même mieux contrôler les efforts de test.

Il existe évidemment plusieurs de ces modèles dans la communauté. Chaque modèle possède ses propres paramètres, mais les plus courants sont les modèles

[2] WOOD, Alan. 1997. « Software Reliability Growth Models : Assumptions vs Reality ». *Proceedings of the Eighth International Sympoisum on Software Reliability Engineering, 1997.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 136-141.

exponentiels et les modèles de courbe en S (Figure 3.3). À ces modèles, on peut généralement trouver les paramètres suivants : la quantité de tests effectués et les données sur les défauts. Pour le premier paramètre, plusieurs éléments ont été identifiés comme unité de mesure : le temps d'exécution, le nombre de jours calendrier de test, le nombre de cas de tests exécutés, etc. Le deuxième paramètre est généralement le nombre de défauts trouvés, mais certaines études tentent d'inférer des informations supplémentaires, telles que la sévérité des défauts, etc.

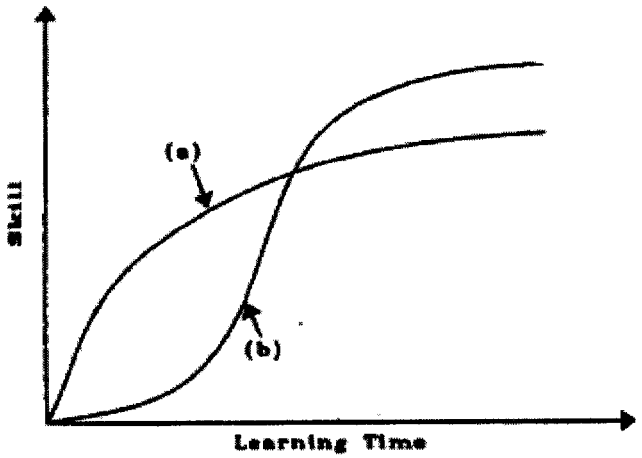

Fig.1 (a) Exponential learning curve; (b)S-shaped learning curve.

Figure 3.3: Représentations de courbes exponentielle et en S[3]

Combinés à des techniques statistiques, il est possible de dériver des courbes qui permettent de prévoir le nombre de défauts restants. Plusieurs recherches ont été effectuées sur ces modèles et les résultats sont assez variables

[3] HOU, Rong-Huei. 1994. « Applying Various Learning Curves to Hyper-Geometric Distribution Software Reliability Growth Model ». *Proceedings of the 5th International Symposium on Software Reliability Engineering*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 8-17.

3.3.4 L'analyse causale

Une autre technique utilisant l'information trouvée dans les défauts est l'analyse causale des défauts. Cette technique, qui est utilisée dans de nombreuses organisations avec succès, base son concept sur l'idée que nous ne devons pas oublier les erreurs que nous faisons, mais bien apprendre d'elles. Le concept de cette technique est que les équipes de développement doivent, de temps à autre, étudier les défauts importants qui ont été découverts dans les cycles/tâches de développement précédentes et analyser leur raison d'être afin d'apporter des améliorations dans le processus.

Selon les chercheurs qui ont analysé cette méthode, les équipes d'analyse causale doivent être composées de développeurs qui connaissent « intimement » le produit développé (Card 1993, Card 1998). Également, ces réunions doivent typiquement être effectuées à des moments particuliers, tels qu'après une phase de test, après un cycle de développement et quelques mois après une version livrée à un client. Comme certains types de réunion d'inspection de code, il est recommandé d'avoir un modérateur et de ne pas avoir des membres du personnel de gestion dans l'équipe d'analyse. Pour la réunion, il est recommandé d'avoir un bassin de défauts d'environ 20 échantillons qui représentent bien les erreurs typiques faites par l'équipe. Ceci implique tout de même un travail préparatoire, voire même une classification des défauts selon une méthode établie et reconnue. Il peut également être intéressant de construire des diagrammes de cause à effet du processus pour tenter d'identifier les problèmes.

Les causes des défauts trouvés gravitent généralement autour des catégories suivantes, soit le manque de méthode de développement ou leur ambiguïté, l'inefficacité des outils ou des environnements de développement, le manque de formation des individus et la mauvaise définition des requis. Comme pour bien des métriques, une bonne formation des participants est nécessaire et il faut essayer de ne pas cibler des gens en particulier, mais bien le processus de développement en tant que tel. Les

résultats qui ont été obtenus lors de l'utilisation de cette méthode ont été assez intéressants, certaines études démontrant que les défauts ont diminué de 50% dans quelques cas.

3.4 Méthodes de classification

Une autre méthode populaire consiste en la classification des défauts selon certains critères afin d'extraire de l'information pertinente. Dans cette section, des méthodes de classification trouvées dans la littérature sont présentées, soit une classification proposée par un standard de l'IEEE et une par Hewlett-Packard.

3.4.1 Classification de l'IEEE

Publié en 1994 et révisé en 1996 (IEEE 1994, IEEE 1996), le standard de classification des défauts de l'IEEE propose une méthode assez élaborée pour classifier les données.

La méthode de l'IEEE se décompose en quatre étapes de classification. Ces étapes sont la reconnaissance, l'investigation, l'action et la disposition. Pour chaque défaut trouvé, il faut rentrer de l'information de classification à chaque étape. Certains critères de classification sont obligatoires, tandis que d'autres sont optionnels. Chaque étape est également caractérisée par trois activités : la classification du défaut, l'enregistrement des informations et l'identification de l'impact du défaut. Ces deux dernières activités sont importantes selon le standard car elles permettent de conserver un historique des défauts avec l'enregistrement et de montrer l'effet des défauts sur le projet par l'impact. Les informations les plus importantes pour chaque étape sont présentées ici.

L'étape de la reconnaissance est entamée lors de la découverte du défaut. Cette étape peut arriver à n'importe quelle phase du produit et peut être initiée par n'importe quel participant au projet, du client au développeur. Dans cette étape, les données obligatoires sont l'activité du projet qui a trouvé le défaut, la phase du projet dans laquelle le défaut a été trouvé et le symptôme associé au défaut. Les données optionnelles sont la cause suspectée, la répétitivité du défaut et l'état du projet.

L'étape suivante, l'investigation, consiste en l'analyse du défaut afin de déterminer si on doit trouver une solution pour corriger le défaut ou non. Les données pour cette étape sont la cause réelle du défaut, la source (artéfact) du défaut et le type de défaut. Les types possibles sont définis par une liste de l'IEEE.

L'étape de l'action implique l'élaboration d'un plan d'activités pour corriger le défaut, ainsi que la planification d'activités pour réviser le processus ou autres éléments susceptibles d'empêcher l'apparition de défauts similaires. À ce stade-ci, il y a une information obligatoire à remplir, la méthode de résolution, et une information optionnelle, l'action corrective. La première se rapporte plus à la correction du défaut, tandis que la deuxième se rapporte plus à l'amélioration du processus.

Finalement, la disposition concerne la fermeture du cas du défaut soit par la résolution positive des actions de l'étape précédente ou l'identification d'actions correctives à long terme. La seule information liée à cette étape concerne l'état final du défaut. Par exemple, le défaut peut être fermé ou encore référé à un autre projet.

À chaque étape, l'impact du défaut et des activités reliées doit être identifié. Les informations liées à l'impact sont toujours les mêmes, quelle que soit l'étape, et doivent toujours être entrées. Les informations obligatoires associées à l'impact sont la sévérité du défaut, l'impact sur la cédule du projet et l'impact sur le coût du projet. Les informations facultatives sont la priorité de résolution, l'impact sur le client, l'effet sur

la complétion du projet, le risque du projet, la qualité nécessaire à appliquer au projet et l'impact social du projet. Il est à noter que ces quatre derniers critères sont généralement utilisés pour des projets critiques ou liés à la sûreté (centrales nucléaires, etc.)

En analysant cette méthode, nous avons jugée qu'elle était certes intéressante à utiliser, mais sans doute trop élaborée. Effectivement, il semblait que cette méthode serait plus appropriée pour des projets très critiques. Également, une méthode aussi administrativement lourde serait mal adaptée aux besoins d'une petite équipe. Notons toutefois que plusieurs des critères de classification sont forts intéressants et certains d'entre eux se retrouvent d'ailleurs dans la méthode que nous avons retenue. Finalement, il est également important de mentionner que peu d'études documentées ont été menées en utilisant cette méthode.

3.4.2 Classification de HP

La compagnie Hewlett-Packard a développé ses propres métriques concernant les défauts et leur classification. Leur approche pour analyser les défauts est analysée dans cette section.

La classification proposée par HP est beaucoup plus simpliste que celle de l'IEEE. Effectivement, cette méthode implique que chaque défaut doit être classifié pour trois paramètres seulement (Huber 2000, Freimut 2002). Ces paramètres sont l'origine, le type et le mode. L'origine est l'activité du cycle de vie qui aurait dû découvrir le défaut analysé. Le type correspond à la catégorie d'erreur à laquelle le défaut appartient. Le mode, quant à lui, est un qualificateur qui définit pourquoi le défaut est apparu. Dans ce paramètre, on peut trouver des choix tels que : manquant, imprécis, incorrect, etc. Il existe également un lien entre l'origine et le type en ce sens que certains types sont associés directement à des origines particulières. Il existe aussi une extension à ce

modèle adaptée au processus de test, mais les attributs de classification ne changent pas, seulement les choix associés à ces attributs changent.

Tout comme le standard de l'IEEE, cette classification est très peu documentée au niveau de son utilisation dans des environnements réels. C'est pour cette raison que la technique de classification présentée dans la prochaine section a été retenue.

3.5 La classification ODC et son extension

Après avoir analysé les méthodes qui existaient pour analyser et comprendre les défauts, ainsi que pour améliorer le processus de test, nous avons décidé de retenir la classification orthogonale des défauts (*Orthogonal Defect Classification* ou ODC) pour notre expérimentation. Dans cette section, cette méthode est présentée en profondeur, incluant les concepts de base, les paramètres de classification de base, ainsi que les extensions au modèle et les métriques que nous avons ajoutées.

Le concept d'ODC a été développé au début des années 90' par Ram Chillarege et d'autres chercheurs chez IBM. Avec cette méthode, les auteurs affirment faire le « pont » entre deux types d'analyse des défauts (Chillarege 2002). D'une part, les méthodes d'analyses statistiques, telles que les modèles de croissance, sont quantitatives, mais elles ne peuvent qu'être faites tard dans le cycle de développement. D'autre part, l'analyse causale, qui permet d'améliorer un processus, voire en plein milieu d'un projet, n'est pas aussi rigoureuse qu'une analyse quantitative. La classification orthogonale des défauts fait le lien entre ces deux extrêmes en étant une méthode quantitative d'analyse qui donne des résultats en temps réel pour un projet. Depuis sa conception, cette méthode a été utilisée dans plusieurs compagnies et plusieurs recherches ont été faites pour montrer l'utilité du modèle et pour tenter de le faire évoluer (Bassin 1997, Butcher 2002). Certaines compagnies au niveau 5 du CMM

utilisent d'ailleurs cette méthode pour remplir les critères clés de ce niveau. IBM continue également de supporter cette méthode sur son site Web (IBM Research 2002) et la méthode a été modifiée quelques fois et même étendue à certains domaines reliés au développement logiciel, telle que l'ergonomie des interfaces usager.

3.5.1 Les attributs originaux

Avant de tenter d'expliquer quelles informations peuvent être extraites du processus de test avec ODC, il convient d'expliquer les différents paramètres de classification qui sont utilisés dans cette méthode. Les définitions originales se trouvent également dans certaines publications trouvées dans les références (Chillarege 1992, IBM Research 2002). La classification des défauts dans ODC se fait en deux étapes, la première étant lorsque que le défaut est découvert et la deuxième étant lorsque le défaut est corrigé. Ces étapes sont appelées la section d'ouverture et la section de fermeture respectivement. Il est également important de noter que tous les défauts doivent être classifiés, incluant ceux qui sont découverts par les clients. D'ailleurs, ces derniers s'avèrent forts importants dans le contexte de notre étude.

Le premier paramètre de classification de la section d'ouverture est l'activité (*Activity*). Ce paramètre représente le type d'activité que la personne effectuait lorsqu'elle a trouvé le défaut. Les choix pour ce paramètre sont les suivants (à noter que les noms anglais ont été conservés pour l'expérimentation).

Tableau 3.1: Description des choix pour le paramètre activité (*Activity*)

Activité	**Description**
Design Review	Le défaut a été découvert lors d'une réunion de revue de design. Peut contenir les erreurs de design et des erreurs liées à la comparaison aux requis.
Code Inspection	L'erreur a été découverte lors d'une inspection de code.
Unit Test	L'erreur a été trouvée en effectuant des tests en boîte blanche.
Function Test	L'erreur a été trouvée en effectuant des tests en boîte noire, en utilisant les requis.
System Test	L'erreur a été trouvée en effectuant des tests systèmes, ce qui implique le produit entier dans un environnement d'utilisation représentatif de la réalité ou l'utilisation réelle.

La classification selon l'activité doit également être indépendante de la phase du produit. Par exemple, si un développeur décide de faire une inspection de code durant la phase de test système, il doit tout de même choisir l'inspection de code comme activité de découverte. Pour ce qui est des défauts qui sont découverts par les clients, ils doivent presque obligatoirement être classifiés dans la catégorie test de fonction ou test système. Effectivement, il est rare qu'un client et utilisateur d'un produit informatique ait accès au code ou qu'il fasse des tests unitaires, ce qui était la règle générale dans notre cas. Des informations supplémentaires sont données sur les choix de classification adaptés à notre environnement dans une section subséquente.

Le prochain paramètre d'ouverture est le déclencheur (*Trigger*). Ce paramètre désigne la condition ou le contexte qui a permis de trouver le défaut pour une activité donnée. Il existe donc une liste de déclencheurs pour chaque type d'activité choisie. À cause du grand nombre de définitions reliées à ce paramètre, les définitions de chaque déclencheur peuvent être retrouvés dans l'annexe I. Ce paramètre est sans doute le plus complexe parmi tous ceux qui composent ODC. Effectivement, pour les activités de

revue et d'inspection, la personne qui classifie doit savoir qu'est-ce qui a attiré son attention pour découvrir ce défaut, ce qui n'est pas nécessairement évident. Pour les autres sortes de test, cela implique que le testeur soit conscient de ses actions pour reconnaître le déclencheur qu'il employait. Pour les besoins de l'environnement, nous avons également simplifié cette liste afin de la rendre plus adaptée ; nous y reviendrons plus tard.

Le dernier paramètre original d'ODC pour la section d'ouverture est l'impact que le défaut aurait eu sur le client s'il avait échappé aux tests. Si le défaut provient du client, l'impact est celui qui a été ressenti par celui-ci. La liste des impacts est également longue, elle sera donc présentée dans l'annexe II. Selon les objectifs de l'équipe, ce paramètre de classification est moins important ; l'analyse sera donc plus brève à ce niveau.

La section d'ouverture à elle seule donne de l'information intéressante sur le processus de test. Effectivement, le paramètre de l'activité donne une distribution des activités qui ont découvert les erreurs. Les déclencheurs, quant à eux, permettent de mieux comprendre comment les défauts ont été trouvés et quelles méthodes ont été moins exploitées, toujours pour chaque type d'activité. Finalement, les impacts permettent de montrer quels problèmes ressurgissent souvent auprès du client. Ces trois paramètres permettent déjà d'avoir des informations intéressantes sur le processus. L'interprétation de ces informations sera présentée plus tard.

Ensuite, vient la section de fermeture. La première information de classification pour cette section est la cible de la correction. Dans la version non étendue d'ODC que nous utilisons, il est possible de classifier la cible selon deux catégories : le design et le code. Ceci reflète l'entité qui a été corrigée lors de la résolution du défaut.

Ensuite, il y a le type de défaut (*Defect Type*). Ce paramètre représente la correction qui a été apportée pour corriger le défaut.

Tableau 3.2: Description des types de défauts (*Defect Type*)

Type de défaut	**Description**
Assignment/Initialization	Valeurs mal assignées ou pas assignées du tout. (À noter que des assignations multiples peuvent être plutôt du type *Algorithm/Method*)
Checking	Erreurs causées par des conditions incorrectes ou manquantes au niveau des variables ou des valeurs de comparaison.
Algorithm/Method	Erreurs causées par l'efficacité ou la validité d'un algorithme ou d'une structure de données locale. L'erreur est corrigée en réimplantant l'algorithme ou la structure de données, ce qui n'affecte pas le design du projet.
Function/Class/Object	L'erreur requiert un changement du design au niveau des structures de classes ou d'une structure de données globale.
Timing/Serialization	Erreurs liées au partage de certaines ressources ou de problèmes de concurrence.
Interface/ O-O Message	Problème de communication entre des modules, des fonctions, etc. affectant l'interface des routines/fonctions
Relationship	Problèmes d'associations entre des structures, des fonctions ou des objets.

À cause de la généralité de certaines définitions du tableau précédent, nous avons dû modifier quelques définitions afin de les rendre plus utilisables en pratique. Ces changements seront présentés dans une section subséquente. Ce paramètre est intéressant car il montre la distribution des défauts selon leur type, ce qui donne une information pertinente sur les erreurs typiques de l'équipe.

Le prochain paramètre est le qualificateur (*Qualifier*). Ce nom provient du fait que ce paramètre qualifie le type de défaut, et par le fait même la correction. Voici les choix qui s'offrent pour ce paramètre.

Tableau 3.3: Description des choix pour le qualificateur (*Qualifier*)

Qualificateur	**Description**
Missing	L'erreur est liée à une omission
Incorrect	L'erreur est liée à une mauvaise application
Extraneous	L'erreur est liée à quelque chose qui est de trop ou non pertinent

Ce paramètre est utile pour montrer comment les développeurs se trompent. Oublient-ils souvent quelque chose, se trompent-ils ou en font-ils trop ?

Ensuite, il y a la source du défaut (*Source*). Ce paramètre exprime la provenance du défaut par rapport au produit développé.

Tableau 3.4: Description des choix pour le paramètre de la source (*Source*)

Source	**Description**
Developed In-House	Le défaut a été trouvé dans un artéfact développé sur place par l'équipe concernée.
Reused from Library	Le défaut a été trouvé dans un artéfact qui fait partie d'une librairie d'objets réutilisés.
Outsourced	Le défaut fait partie d'un artéfact qui est produit hors de la compagnie ou qui a été acheté.
Ported	Le défaut provient du port d'un artéfact provenant d'un autre environnement.

Ce type d'information permet d'identifier la source (d'où le nom) des erreurs et peut permettre de mettre l'emphase de qualité sur certaines provenances d'artéfacts.

En dernier lieu, il y a l'âge du défaut (*Age*). Ce paramètre concerne la vieillesse de l'artéfact ou de la partie d'artéfact dans laquelle le défaut a été trouvé.

Tableau 3.5: Description des choix pour le paramètre de l'âge (*Age*)

Age	**Description**
Base	Le défaut est dans une partie d'un artéfact qui n'est pas présentement en développement. C'était un défaut latent.
New	Le défaut a été découvert dans une section présentement en développement.
Rewritten	Le défaut a été introduit suite à une réécriture de l'artéfact.
Re-fixed	Le défaut a été introduit suite à la correction d'un autre défaut.

Ce paramètre donne une idée sur la vitesse à laquelle un défaut est trouvé, mais également sur la raison de l'existence de ce défaut.

Une des parties les plus importantes d'ODC est le lien qui existe entre les activités de découverte des défauts et les types de défauts. Effectivement, selon les auteurs (Chillarege 1992), certains types de défauts devraient être trouvés principalement par certains types d'activités (Figure 3.4). On affirme également que, pour un projet de développement donné, il serait attendu que le nombre de certains types de défauts devrait converger avant d'autres types (Chillarege 1992)(Figure 3.5). À titre d'exemple, pour un projet en contrôle, il serait attendu que le nombre de défauts de type *Function/Class/Object* converge avant le nombre de défaut de type *Assignment/Initialization*, car en principe, ces types doivent être découverts principalement par des activités de révision de design. L'analyse en fonction du temps

peut être intéressante, mais les méthodes de travail de l'équipe étudiée empêchent ce type d'analyse. Effectivement, puisque chaque développeur travaille sur un module de façon indépendante, il y aurait trop peu de défauts par phase pour vraiment constater des tendances. La notion de phase s'applique donc très difficilement à ces produits. La seule analyse faite est celle concernant l'association entre les activités et les types de défauts.

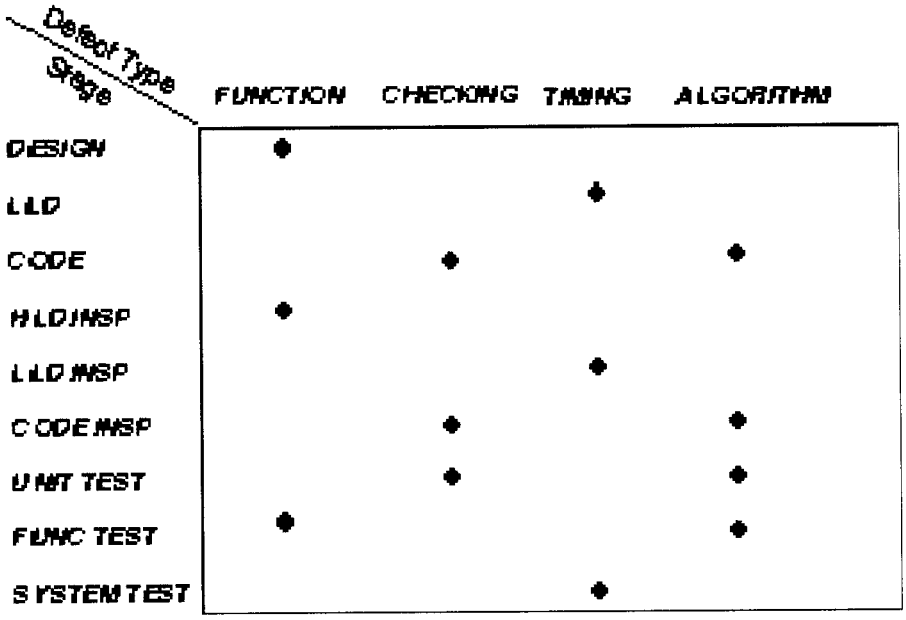

Figure 3.4 : Présentation de quelques associations entre activités et types de défauts[4]

[4] CHILLAREGE, Ram. Novembre 1992. « Orthogonal Defect Classification – A Concept for In-Process Measurements ». *IEEE Transactions on Software Engineering*. 18:11. 943-956.

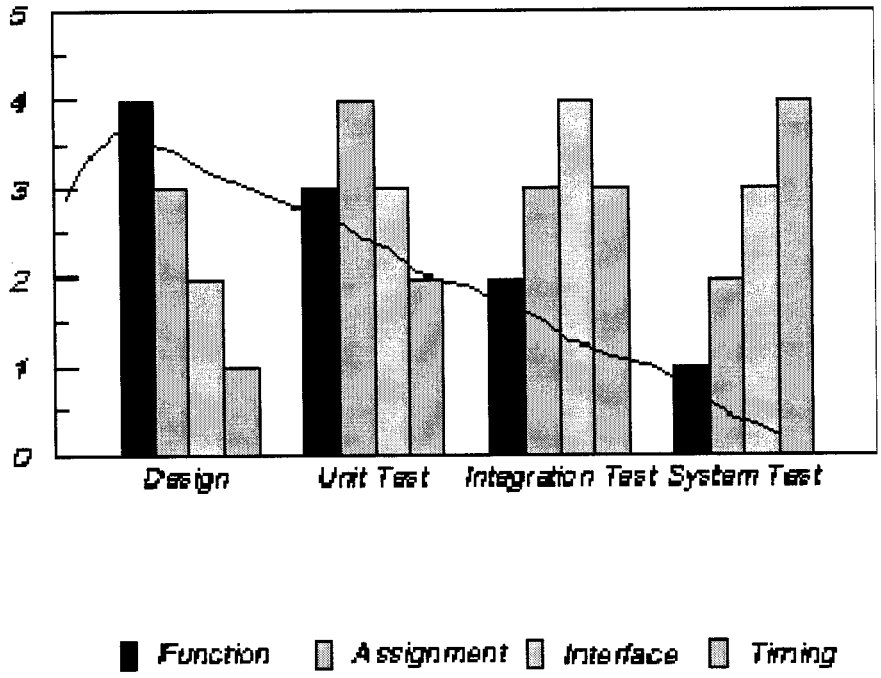

Figure 3.5: Évolution des types de défaut en fonction de l'avancée du projet[5]

Les paramètres qui ont été présentés ici sont ceux qui se retrouvent sur le site officiel d'ODC sur le site d'IBM, dont la dernière mise à jour fut le premier février 2002. En fait, depuis son invention en 1992, cette méthode de classification a été modifiée quelques fois. La prolifération des méthodes orientées objet ont, par exemple, modifié le nom de certains choix dans la catégorie du type de défaut, et aussi ajouté un nouveau type, *Relationship*. Également, l'utilisation de cette méthode dans des contextes différents a permis de développer des extensions au modèle de classification. IBM s'en sert entre autres pour classifier des erreurs d'interface usager, des erreurs de localisation de logiciel, etc. Dans notre contexte, nous avons choisi de ne pas utiliser ces extensions, car les produits développés et le domaine s'appliquent mal dans ce cas-ci. Également, certaines de ces extensions relèvent plus de d'autres domaines, connexes au génie logiciel certes, mais pas pertinents à notre étude.

---

[5] *Id*

Tandis que le modèle original d'ODC est fort intéressant, il est possible que certaines informations échappent à celui-ci. Ainsi, certaines extensions devaient être ajoutées pour rendre l'expérience plus intéressante.

3.5.2 Attributs et métriques supplémentaires

Cette section présente les informations que nous avons ajoutées à notre modèle pour complémenter ODC, ainsi que les métriques qui sont utiles à l'analyse de l'étude.

Vu qu'il est question d'un environnement où les clients sont en contact intime avec l'équipe de développement, un attribut qui spécifie le découvreur du défaut a été ajouté. Ce paramètre, que nous avons nommé *Discoverer Category*, indique simplement qui a découvert le défaut. Ainsi, le développeur qui classifie un défaut indique s'il s'agit d'un défaut trouvé par un membre de l'équipe dans lequel cas il est classifié interne (*Internal*), ou s'il a été découvert par un client, dans lequel cas il est classifié externe (*External*). Ce paramètre permet ainsi de mesurer quantitativement la proportion des défauts qui sont trouvées par les clients versus ceux qui sont trouvés par les développeurs.

Ensuite, vu qu'il s'agit d'un contexte de processus et de planification informels, certaines activités sont planifiées, tandis que d'autres sont plutôt opportunistes, car elles sont initiées par les développeurs personnellement, en dehors de leur planification. Un paramètre supplémentaire a donc été ajouté dans la classification des défauts. Intitulé qualificateur d'activité (*Activity Qualifier*), ce facteur peut prendre deux valeurs : planifié (*planned*) ou opportuniste (*opportunistic*). Une activité est planifiée si celle-ci était prévu dans la planification du développeur (ou même du client), tandis qu'une activité est opportuniste si elle est accidentelle ou imprévue. À l'aide de cette

information, il sera possible d'évaluer la distribution des erreurs qui sont trouvées de façon aléatoire (opportuniste) et celles qui sont trouvés par des activités planifiées.

Au niveau des métriques, la première information qui est extraite est le temps avant découverte d'un défaut. Pour obtenir cette information, la date à laquelle le défaut a été découvert et la date approximative à laquelle le défaut a été introduit devront être trouvées. La première date sera obtenue par la base de données assez facilement. La date d'introduction, quant à elle, sera obtenue en fouillant les archives de gestion de configuration du code et en interrogeant les développeurs impliqués, vu qu'il est possible de le faire. En extrayant cette information, il sera possible d'avoir une information qui complémentera très bien l'âge du défaut d'ODC.

Finalement, quoique moins nécessaire, le temps de correction de défaut est également mesuré. Cette information sera obtenue en soustrayant la date de correction à la date de découverte. Ces données seront surtout intéressantes pour l'équipe de développement que pour notre étude.

Les informations données par ODC étendue et les métriques sont très intéressantes théoriquement, toutefois, l'environnement de l'expérimentation pourrait amener des spécificités ou des problèmes d'implantation.

3.6 Application de la méthode à l'environnement

À cause de la spécificité de l'environnement de développement, la méthode retenue pour l'observation doit être adaptée. Dans cette section, l'application de la méthode d'analyse des défauts à l'équipe est présentée. Les besoins de l'équipe et de l'administration sont d'abord identifiés, puis l'outil de capture des données est présenté. Il y aura ensuite discussion sur les interprétations possibles des données et également de

la validation des données et des problèmes potentiels de classification. Finalement, le plan de l'expérimentation est proposé.

Dans le milieu industriel, il peut être difficile parfois de marier les intérêts du chercheur et du praticien sur le terrain. Dans ce contexte-ci, certains objectifs doivent être respectés afin de produire des résultats valides en ne perturbant pas trop les habitudes des développeurs, du moins pour la première partie de l'expérience. Premièrement, il est important que l'équipe ne perde pas trop de temps à classifier les défauts. Nous avons donc intégré les paramètres de classification à la base de données déjà existante de l'équipe. Également, nous avons modifié et clarifié certaines définitions qui étaient ambiguës ou s'appliquaient mal à l'environnement étudié. Finalement, nous avons expliqué les implications et la signification des résultats que nous obtiendrions en bout de ligne, afin de tempérer les attentes face à cette expérience.

3.6.1 Outil de classification

L'outil utilisé était une base de données conçue expressément pour la saisie d'informations relatives aux défauts. Cet outil, qui existe d'ailleurs sous plusieurs versions différentes selon l'équipe qui l'utilise, possède une interface Web qui est relativement simple à utiliser et à modifier selon les besoins. L'équipe étudiée est évidemment déjà familière avec le fonctionnement de cet outil et nous avons tout simplement intégré les champs dans les fiches de classification. Également, l'équipe responsable du développement des outils administratifs de l'entreprise fournit un support technique en cas de problèmes ou d'ajustements. L'outil comporte plusieurs paramètres de classification qui sont déjà en place pour l'équipe. Voici un tableau présentant brièvement ces champs et leur utilité.

Tableau 3.6: Description des attributs de la base de défauts de l'équipe avant ODC

Paramètre	**Description**
Date de soumission (*Submit Date*)	Date à laquelle le défaut a été trouvé.
Soumissionnaire (*Submitter*)	Nom de la personne qui a entré le défaut
Version Testée (*Tested product*)	Numéro de version du produit dans lequel le défaut a été trouvé
État 1 (*State*)	Ouvert ou fermé, selon l'état 2
État 2 (*Status*)	Ce paramètre regroupe un ensemble d'états dans lequel le défaut peut être. Les plus sélectionnés sont *Fixed* (corrigé), *New* (nouveau) et *Not a Defect* (Pas un défaut). Cette liste contient également d'autres états, qui sont moins importants.
Sévérité** (*Impact gravity*)	Auto explicatif. Peut prendre les valeurs : A (arrêt complet du logiciel), B (perte d'une fonctionnalité), C (perte temporaire de fonctionnement), Q (question d'optimisation ou d'ergonomie)
Module (*Module*)	Nom du produit dans lequel le défaut se trouve
Plateforme (*Platform*)	Type de machine pour lequel le défaut existait. Ceci peut être un type spécifique (ex. XBox, PC) ou être multi plateforme.
Remarque du soumissionnaire (*Submitter's remark*)	Remarques données par la personne qui entre le défaut. À noter qu'aucune information particulière n'est demandée
Priorité de correction** (*Fixing priority*)	Importance du bug au niveau de l'impact sur le client. Peut prendre les valeurs : A, B, C, Q, où A est le plus urgent et Q, le moins urgent.
Correcteur (*Solver*)	Nom de la personne ayant corrigé le défaut

Date de correction * (*Correction date*)	Date à laquelle le défaut a été corrigé.
Version de correction (*Fixed Product*)	Numéro de version du produit dans lequel le défaut a été trouvé.
Remarque du correcteur (*Solver's Remark*)	Remarques données par la personne qui a corrigé le défaut. À noter qu'aucune information particulière n'est demandée

*À noter qu'initialement, les dates étaient rentrées automatiquement, donc ces dates représentaient en fait la date à laquelle le défaut était entré dans la base et la date à laquelle le défaut était mis dans un état fermé. Ceci pouvait causer des erreurs minimes sur certaines métriques de temps. Nous avons changé l'outil de manière à ce que la date puisse être éditée par le soumissionnaire et le correcteur.

** Initialement, un seul paramètre existait qui réunissait ces deux informations. Après quelques discussions avec l'équipe, cet ancien paramètre a été scindé en deux parties soit la gravité et la priorité de correction, afin de mieux organiser les objectifs de correction.

Une autre information importante qui est retenu par l'outil est l'historique de modification des fiches. Cette fonctionnalité permet de montrer quel individu avait changé quels paramètres, avec quelles valeurs et à quelle date. Ceci nous donne certaines informations qui sont utiles à la validation des données.

Suite à la présentation des paramètres ODC, l'équipe trouvait que certaines définitions d'ODC étaient floues ou moins applicables. Tandis que certaines définitions ont été conservées suite à la production d'exemples pratiques et représentatifs, d'autres ont dû être retirés de la liste originale ou ont vu leur définition partiellement modifiée. L'annexe III présente les définitions modifiées et les retraits. Ces retraits ou modifications n'affectent pas la validité des résultats. Effectivement, l'adaptation de la

méthode à l'environnement contribue plutôt à augmenter la validité des résultats car ils deviennent plus significatifs dans le contexte.

3.7 Interprétations possibles

Comme indiqué dans une section précédente, plusieurs interprétations sont possibles grâce à l'utilisation d'ODC. Combiné aux paramètres de classification existant de l'équipe, il est possible d'extraire de l'information intéressante sur le processus de test de l'équipe. Cette section les présente brièvement.

Premièrement, il est possible d'identifier quels produits de l'équipe D.A.R.E ont le plus défauts. Cette information permet de comprendre quels types de logiciels sont plus susceptibles aux erreurs et sur quels produits l'équipe doit concentrer ses efforts de test. Il est également possible de coupler cette information en fonction de la gravité des erreurs trouvées.

Ensuite, une analyse de la distribution des activités de découverte peut être effectuée. Provenant exclusivement d'ODC, cette information permet de comprendre quels types d'activités trouvent le plus fréquemment les erreurs dans les produits. Cette information est recoupée premièrement avec le découvreur et le qualificateur d'activité. La première inférence montre la distribution des activités internes et externes, ce qui est intéressant pour comprendre les efforts de l'équipe d'une part et le type d'utilisation du logiciel par les clients d'autre part. Le couplage avec le deuxième paramètre identifie quelles activités sont plus susceptibles d'être opportunistes et quelles activités planifiées sont plus efficaces.

La distribution des déclencheurs en fonction des activités est ensuite analysée. La définition de ce paramètre indique comment les erreurs sont trouvées par l'équipe et

les clients en fonction des activités. Les distributions des déclencheurs donnent donc une idée du processus de vérification, car elles montrent quelles « techniques » sont utilisées pour trouver les défauts. Finalement, pour terminer la section d'ouverture, la distribution de l'impact des défauts donne un autre indice sur la nature des défauts

Du côté des paramètres de fermeture, les distributions de chacun des paramètres de cette section sont analysées de façon individuelle: type de défaut, age, source, qualificateur de correction. Des inférences entre l'âge des défauts et les découvreurs, ainsi que les qualificateurs d'activités sont effectuées également. Toutes ces analyses sont utiles pour déterminer la nature des défauts, et leur provenance (dans le temps et l'espace).

Un des aspects les plus importants d'ODC est l'inférence entre les activités de découverte et les types de défauts. Cette analyse est effectuée sur l'échantillon, mais comme mentionné précédemment, les convergences dans le temps ne sont pas analysées. Cette information indique si le processus de test détecte les défauts au moment attendu ou non.

Finalement, les distributions statistiques du temps de correction et du temps de découverte sont analysées. Cette dernière information est combinée aux activités de découverte et au qualificateur d'activité afin de montrer si celles-ci ont un impact sur le temps de découverte.

3.8 Déroulement de l'expérience

Cette expérimentation se réalise en trois grandes phases. Une première phase pour évaluer la situation actuelle du processus. Une deuxième phase pour interpréter les

résultats de la première partie et élaborer un plan d'amélioration et une troisième phase pour évaluer l'évolution du processus en fonction des changements proposés.

Les objectifs de la première phase sont clairs. Il s'agit premièrement de mettre en place le système de saisie de donnée et de former les développeurs afin qu'ils puissent utiliser la méthode de classification. L'accumulation des données reliées aux défauts commence lorsque tout le système est en place. Tout au long de cette première phase, il est important de rester à l'écoute de l'équipe afin de s'assurer que ceux-ci comprennent bien les définitions de classification et qu'ils les appliquent bien. Les données sont validées au fur et à mesure qu'elles sont saisies et toutes les erreurs sont corrigées en révisant s'il y a lieu les modifications ou les ajouts aux définitions. Comme le facteur temps n'est pas analysé directement dans cette expérience, nous prévoyons environ 6 mois d'accumulation ou un nombre suffisant de défauts afin d'avoir un échantillon intéressant.

Une fois cette phase terminée, les résultats obtenus seront analysés pour élaborer une stratégie d'amélioration du processus de test dans le cadre de la deuxième phase. Pour se faire, une recherche est effectuée pour trouver des méthodes et des outils pour améliorer le processus. Suite à cette recherche, le processus est révisé avec l'équipe de développement. Lorsqu'une entente est atteinte sur les nouvelles pratiques à implanter, la troisième phase est entamée. Il est à noter que les données reliées aux défauts seront toujours cueillies durant cette période.

La troisième phase est essentiellement semblable à la première phase. Les données de défauts sont accumulées afin de mesurer si les améliorations que nous avons proposées ont eu un effet sur l'efficacité du processus de test. Nous continuons à valider les données et à réviser les définitions s'il y a lieu. L'objectif est également d'obtenir un échantillon au moins aussi grand que dans la première partie. Si le processus de test devient plus efficace et qu'un plus grand nombre de défauts sont trouvés, il est possible

que le temps de saisie soit plus court pour obtenir le même nombre de défauts qu'à la première partie. Dans ce cas-là, nous pourrons continuer l'expérience pour avoir un temps de saisie aussi long qu'à la première partie, mais nous aurons alors sans doute un échantillon plus grand. L'expérience se termine par une analyse des données de la deuxième partie avec une comparaison par rapport à la première.

En conclusion, ce chapitre a présenté quelques recherches qui concernent les défauts. Également, la technique retenue pour l'étude et son application au contexte de l'étude a été proposée. Dans les prochains chapitres, les résultats et les analyses des différentes étapes de cette expérimentation sont dévoilés.

CHAPITRE IV : LES RÉSULTATS DE LA PREMIÈRE PARTIE

Ce chapitre est consacré à l'analyse de l'échantillon des défauts obtenus à la première partie de l'expérimentation. Ce chapitre présente d'abord une description générale de l'échantillon, et par la suite une analyse en profondeur des données au niveau des paramètres de classification. L'interprétation de ces analyses est faite au fur et à mesure.

4.1 Un survol de l'échantillon

Avant de rentrer dans les détails de l'analyse de l'échantillon, il est important de regarder cet échantillon afin de mieux comprendre ce qui peut en être retiré. Dans cette section, l'échantillon est analysé de plusieurs points de vue dont le temps d'échantillonnage, la représentativité de l'échantillon, la quantité de défaut, le retrait des défauts inutilisables et les erreurs liées à la classification.

La période d'échantillonnage des défauts qui est étudiée pour cette première partie s'est étendue de la fin du mois de février jusqu'à la fin du mois d'août pour l'année 2003. La période de saisie a commencé au moment où les définitions ont été raffinées une première fois et que l'outil était en place. Durant cette période de temps, plusieurs développements de fonctionnalités ont été complétés, et certains projets clients de l'équipe ont eu des étapes importantes à compléter dont la préparation de démonstrations pour la très importante exposition de l'industrie (E3), ainsi que la complétude de projets. Selon les membres de l'équipe, la période d'échantillonnage a été représentative de l'effort qu'elle déploie normalement.

Au cours de cette période, près de 300 défauts ont été recensés dans la base. Tandis qu'aucune donnée quantitative relative au nombre de défauts durant une période

de temps existe au sein de l'équipe, les membres affirmaient que celui-ci n'était pas particulièrement bas ou particulièrement haut. À la lumière de ces informations, nous en déduisons que le nombre de défauts est représentatif du nombre habituel de défauts pour le contexte donné.

Toutefois, l'échantillon analysé ne comporte pas ces 300 défauts et ce pour plusieurs raisons. Premièrement, il est impossible de retrouver l'historique temporel pour certains défauts à cause de l'absence de certaines informations clés. Également certains défauts qui ont été insérés dans la base de données ne sont pas des défauts que l'on peut qualifier « d'ingénierie ». À titre d'exemple, des erreurs dans les commentaires de code n'ont pas été gardées, et de toute façon notre système de classification n'est pas prévu pour ce type d'erreur. La plupart de ces erreurs étaient d'ailleurs qualifiées de « Q » au niveau de la sévérité, et n'étaient donc pas importantes pour l'équipe. On se retrouve donc avec un échantillon de 225 défauts valides, qui sont analysés dans la section suivante.

4.2 Analyse

Cette section présente sous forme de graphiques les analyses statistiques de l'échantillon. Sont d'abord présentées les distributions qui ne sont pas liées directement à ODC, suivies des distributions relatives à ODC, puis finalement les temps de découverte. Le temps de correction n'est pas traité dans cet ouvrage.

Une première analyse de la distribution des défauts dans les différents produits de l'équipe montre que la plupart des erreurs se trouvent dans le moteur et l'éditeur de sons (*SoundEditor*). Ceci est normal, car l'essentiel des efforts de l'équipe est consacré au développement de ces deux produits.

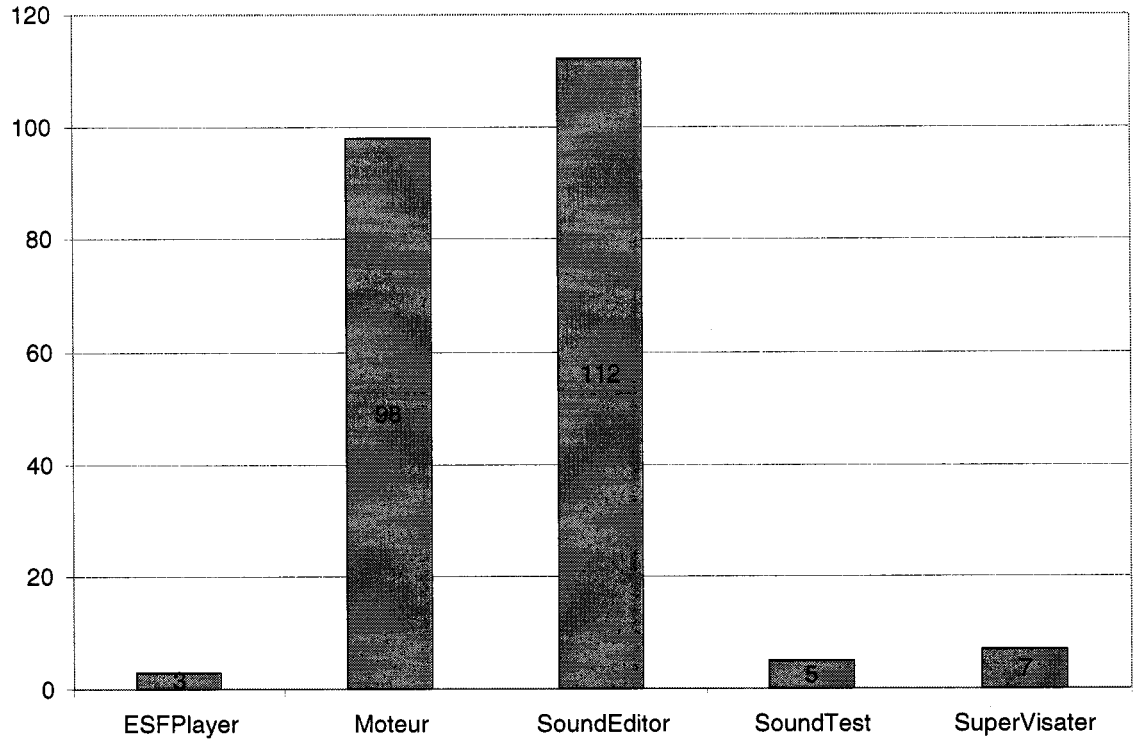

Figure 4.1: Distribution des défauts valides dans les produits de l'équipe pour le premier échantillon

Comme les critères de qualité sont moins importants pour les trois autres produits, les 15 défauts qui leur sont associés sont retirés de l'étude. L'échantillon se retrouve donc diminué à 210 défauts.

Sur ce total de 210 défauts, il y en a 184 qui sont corrigés (88%) et 26 défauts qui ne le sont pas encore. Nous conservons les défauts de l'échantillon dans cet état même si certains des défauts furent corrigés par après par souci de montrer la situation réelle au moment de l'analyse.

Ensuite, la distribution de la sévérité dans les deux produits restants montre qu'une bonne proportion des défauts (82% et 67%) est d'une assez grande importance

(A et B). L'équipe a donc affaire à des défauts sérieux. À noter que les défauts de sévérité « Q » résiduels sont des défauts qui ne sont pas très sévères, mais représentent des défauts assez sérieux pour être gardés dans l'étude. La discrimination entre ces deux types de défauts « Q » est faite par les membres de l'équipe qui évaluent la pertinence de la sévérité de chaque défaut.

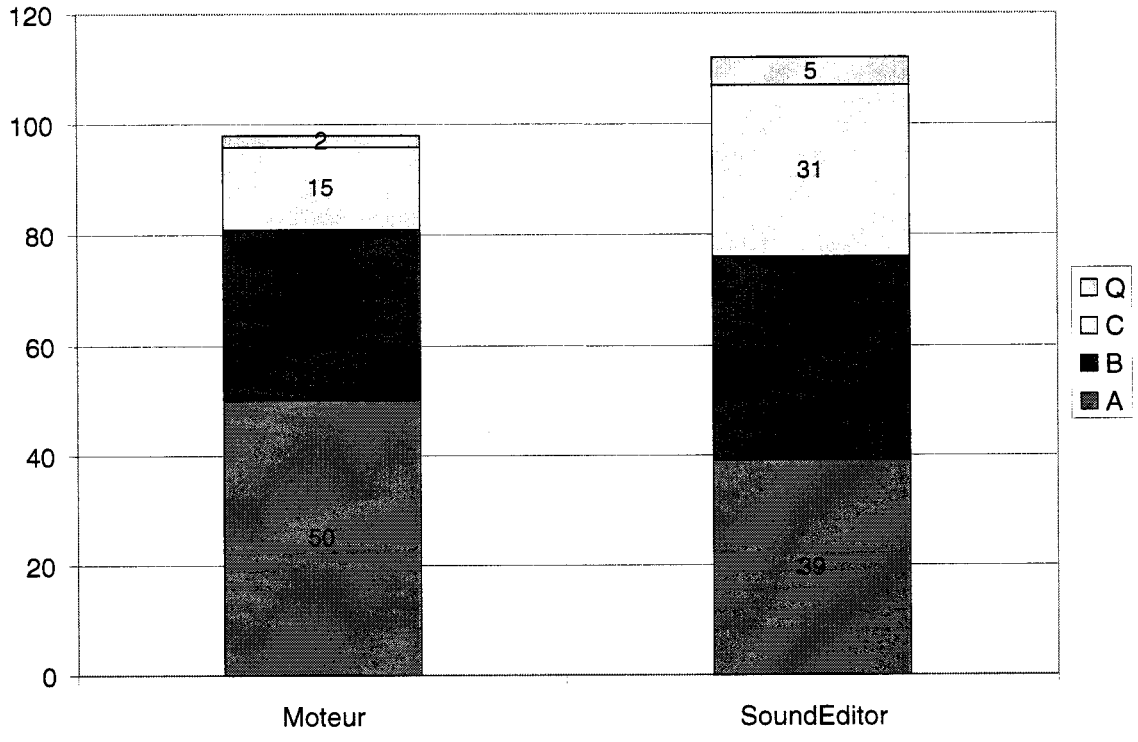

Figure 4.2: Distribution des défauts dans les produits avec les informations de sévérité pour le premier échantillon

Un premier fait intéressant à considérer est que les deux produits restants ne sont pas du même type. Effectivement, le moteur est un logiciel qui s'apparente beaucoup à un de jeu en ce sens qu'il n'est pas directement interactif avec l'utilisateur, tandis que « SoundEditor » est un outil principalement interactif, selon la définition habituelle, et

est utilisé par les métiers des projets clients. Il s'avère donc que les deux types de logiciel causent autant d'ennui aux développeurs de l'équipe.

Les analyses qui suivent concernent le processus de test en tant que tel. Le premier graphique présente la distribution des activités de découverte des défauts.

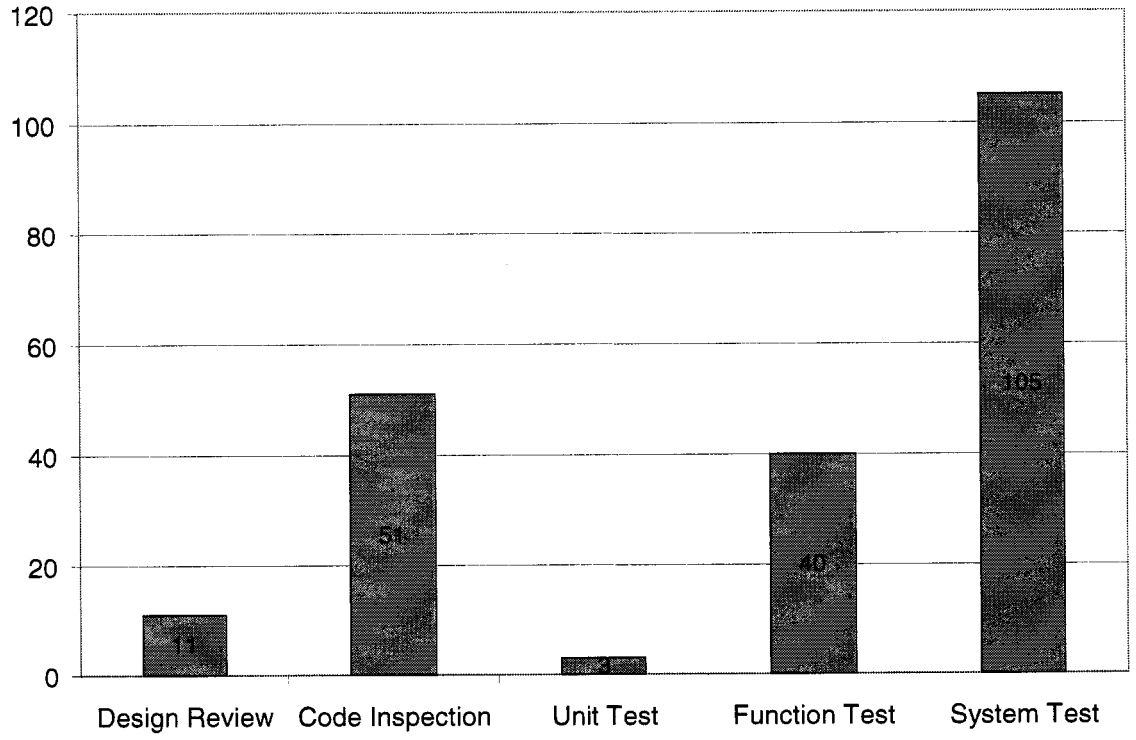

Figure 4.3: Distribution des activités de découverte pour le premier échantillon

Ce graphique montre qu'environ la moitié des erreurs sont trouvées par des tests de type système. Les deux autres types dominants sont l'inspection de code et les tests de fonction. Par contre, une très petite proportion des défauts est trouvée par tests unitaires et par revues du design.

À première vue, le fait qu'un grand nombre de défauts (50%) soient découverts par des tests système annonce que ceux-ci sont probablement trouvés tard dans le cycle

de développement. Par contre, la forte dominance secondaire des inspections de code semble de prime abord plus positive, car cela indique que cette activité trouve une part appréciable des défauts. Finalement, la proportion également importante des défauts trouvés par test de fonction peut être un signe négatif ou positif. Effectivement, tout dépendra de l'opportunisme de ces activités et des types de défauts identifiés par celles-ci. Il demeure possible que les types de défauts trouvés ne soit pas associés à leur activité normale. Ces interprétations initiales sont certainement insuffisantes, il faudra une analyse supplémentaire en fonction d'autres attributs afin d'avoir une meilleure compréhension de la situation.

La première analyse complémentaire implique l'ajout des attributs de découvreur et de qualificateur d'activité. Les deux graphiques suivants montrent respectivement l'effet de ces deux attributs. La distribution des activités de découverte pourra prendre plus de sens avec ces informations.

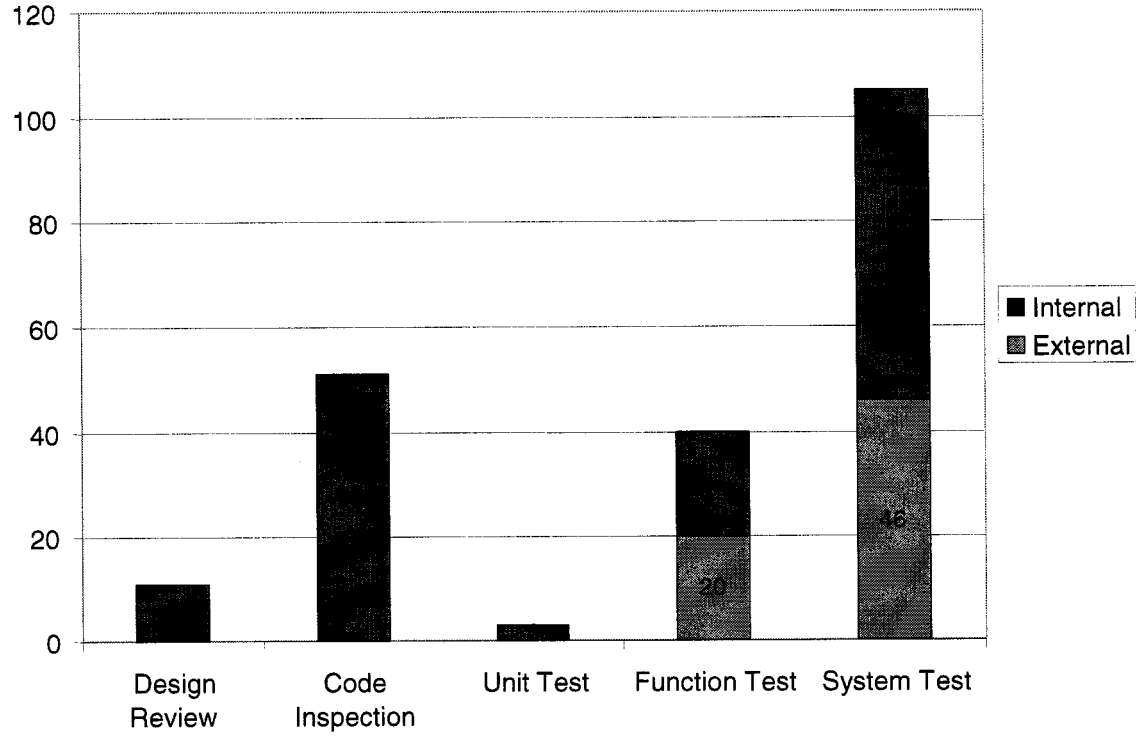

Figure 4.4: Distribution des activités de découverte avec l'information du découvreur pour le premier échantillon

Dans ce premier graphique, l'effet de la distribution des découvreurs de défauts est visible. La plupart des défauts sont trouvés par les développeurs de l'équipe. Aussi, les activités qui sont traditionnellement associées aux développeurs, telles que les revues de design, les revues de code et les tests unitaires, ont été effectuées par les développeurs eux-mêmes, ce qui est normal. À première vue, la distribution des découvreurs semble également normale, car la plupart des défauts sont trouvés par les développeurs.

Toutefois, en regardant de près la distribution des découvreurs pour les activités de test de fonction et de système, il apparaît qu'il y a une importante proportion des défauts qui sont trouvés à l'externe. Même si la plupart des défauts pour l'ensemble des

activités sont trouvés par l'équipe, il en demeure tout de même que plus de la moitié des erreurs sur les produits complétés sont trouvées par des clients, ce qui n'est généralement pas souhaitable. Effectivement, les tests de fonction et de système sont des types de test généralement effectués sur des produits intégrés et utilisables. Une conclusion à ce point serait sûrement hâtive, les analyses suivantes devraient mieux dévoiler le portrait de la situation.

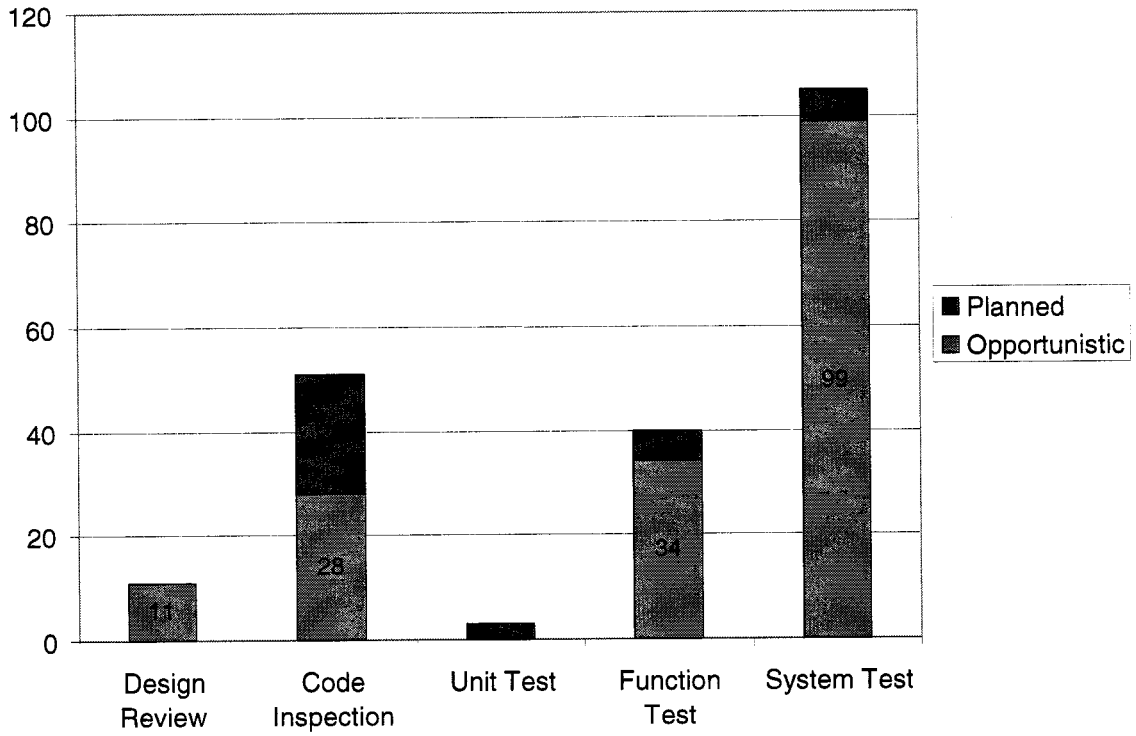

Figure 4.5: Distribution des activités de découverte avec l'information de qualificateur d'activité pour le premier échantillon

Le graphique en Figure 4.5 présente la distribution des qualificateurs d'activité en fonction des activités. Une analyse de la situation montre que la plupart des défauts sont trouvés de façon opportuniste. La plupart des défauts trouvés par test de fonction ou de systèmes sont trouvés de façon opportuniste. Ceci ne serait pas surprenant, étant

donné la grande proportion des défauts trouvés à l'externe, mais les « opportunistes » semblent englober les internes également, à première vue. Au niveau des inspections, on remarque que celles-ci aussi s'avèrent très opportunistes. Effectivement, plus de la moitié des inspections de code sont opportunistes et tous les défauts trouvés par révision de design sont également opportunistes. Les tests unitaires sont les seuls à être exclusivement planifiés, mais ce type d'activité n'est pas pratique courante dans l'équipe, et donc, ne sont pas très significatifs. Il est toutefois intéressant de noter la faible présence de ce type d'activité.

À ce stade-ci, on pourrait se questionner sur la grande quantité de découvertes de défauts opportunistes. Effectivement, comme il a été mentionné plus tôt dans cet ouvrage, les tests ne sont pas une pratique établie dans les habitudes de travail de l'équipe étudié. Il n'est donc pas surprenant de trouver, très peu de tests unitaires dans un premier temps, et beaucoup de tests de fonction et de tests système opportunistes. Toutefois, qu'en est-il des inspections et revues opportunistes? Étant dans un contexte de maintenance, il arrive que les développeurs regardent de l'ancien code à tout hasard, trouvant parfois des défauts dans le code. Les revues de design opportunistes sont peut-être plus bizarres. Lorsqu'interrogés, les développeurs concernés ont dit qu'il s'agissait bel et bien de revue de design, mais que celles-ci avaient été effectuées sur un produit fini, ce qui n'est généralement pas souhaitable. La conclusion à extraire de cette analyse est qu'il y a trop d'opportunisme dans les activités de test.

La figure 4.6 présente la distribution de l'opportunisme en fonction des découvreurs. Le fait que l'opportunisme soit fort même dans les découvertes internes est assez évident dans ce graphique. On pourrait regarder le même type de distribution en fonction des activités, mais cela donnerait très peu d'informations supplémentaires. L'important fait à noter, encore une fois, est que la plupart des découvertes de défauts sont tout à fait fortuites.

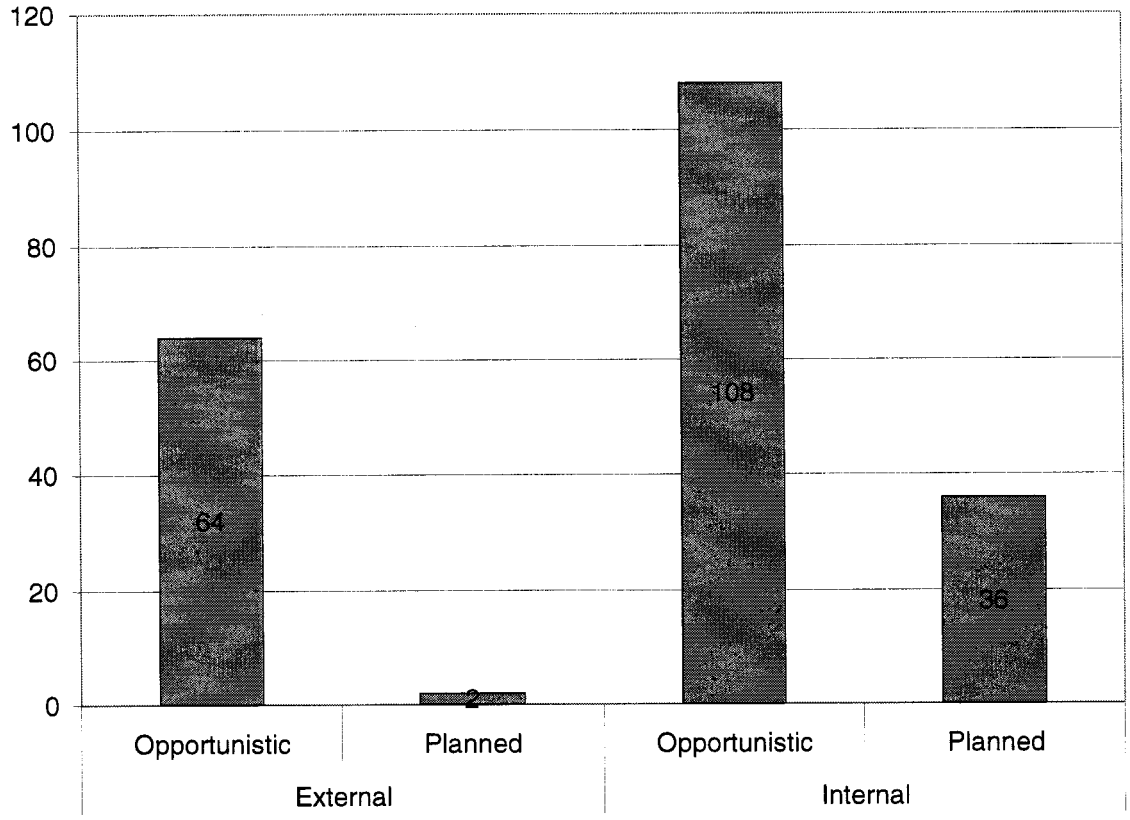

Figure 4.6: Distribution des qualificateurs d'activité avec les découvreurs pour le premier échantillon

Les prochaines figures (4.7 à 4.10) montrent la distribution des déclencheurs pour chaque type d'activité, sauf pour les tests unitaires, qui ne sont pas assez significatifs. Ces distributions risquent de donner un autre signe d'opportunisme.

La figure 4.7 présente la distribution des déclencheurs pour les révisions de design avec l'information de qualification d'activité. Le fait que le déclencheur *Design Conformance* soit majoritaire indique que les erreurs trouvées ont rapport au non-respect des requis. Ceci n'est pas vraiment surprenant pour des révisions de design, ne serait-ce que l'aspect opportuniste de l'activité, qui indique que ces défauts ont été trouvés trop

tard. De toute façon, les erreurs trouvées par révision de design sont moins importantes dans ce contexte-ci.

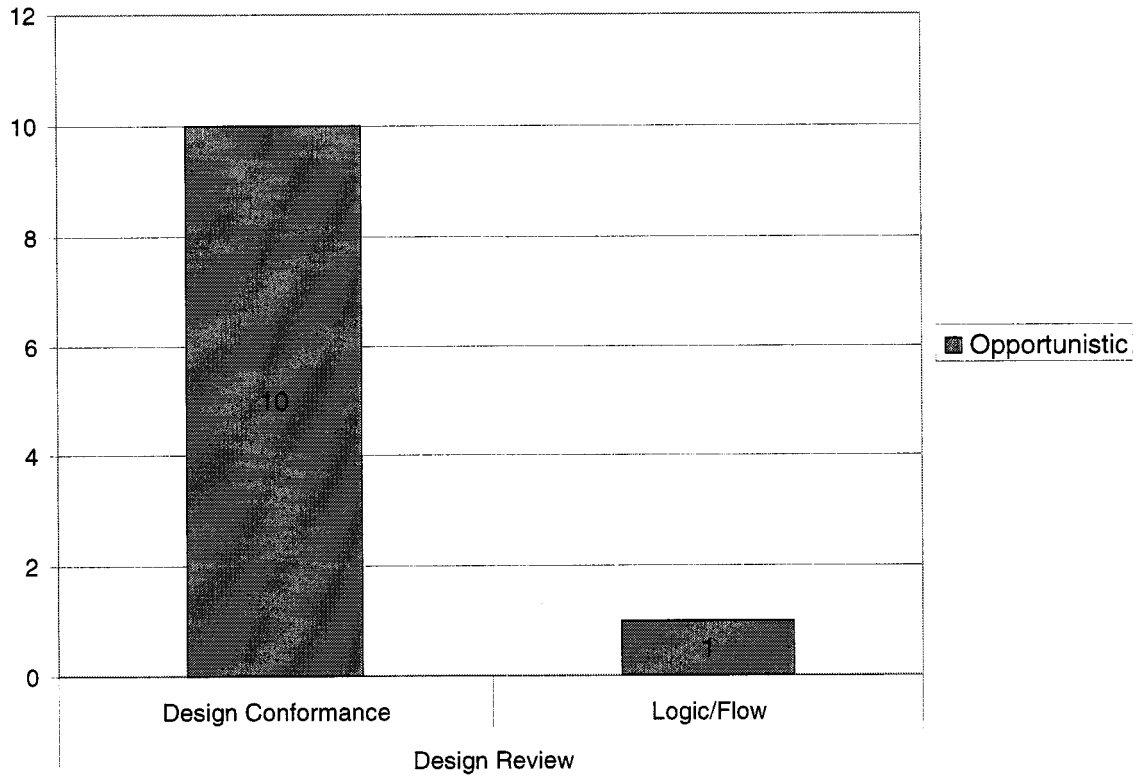

Figure 4.7: Distribution des déclencheurs pour l'activité de révision de design avec l'information de qualificateur d'activité

Le prochain graphique (Figure 4.8) montre la distribution des déclencheurs pour les inspections de code, toujours avec l'information sur l'opportunisme. Il apparaît premièrement que les déclencheurs sont autant opportunistes que planifiés, comme l'ensemble des défauts trouvés par inspections de code, ce qui est relativement normal. Le non-conformisme au design et les erreurs de logiques sont les déclencheurs les plus importants dans ce cas-ci et sont assez standards pour ce type d'activité. Ce qui est peut-être plus surprenant est le nombre important de déclencheurs de type *Side Effects* et

Rare Situations. Contrairement à ce que pourrait indiquer le nom de ces déclencheurs, il n'est

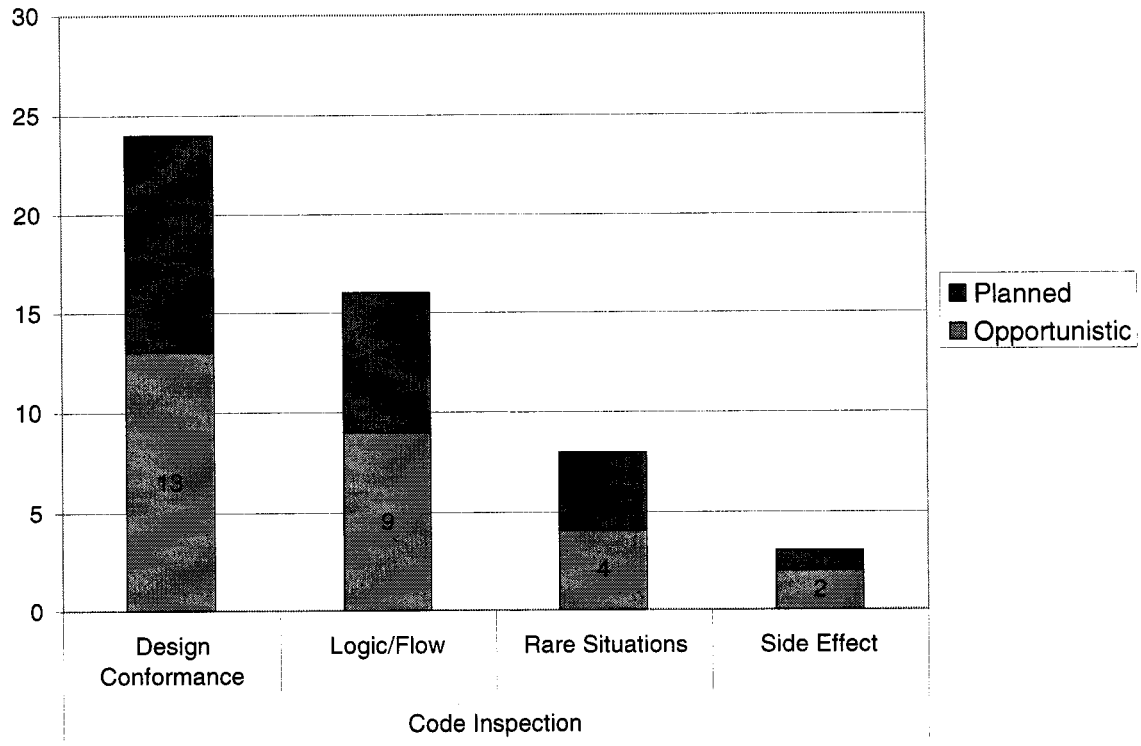

Figure 4.8: Distribution des déclencheurs pour l'activité d'inspection de code avec l'information de qualificateur d'activité pour le premier échantillon

pas si surprenant qu'il y ait autant de ces types. Effectivement, comme il s'agit de logiciels assez vieux, et que le design est assez complexe par le fait même, plusieurs modules ont des éléments en commun qui pourraient causer des problèmes (variables partagées, etc.). Il est normal qu'un développeur qui connaît bien les produits utilise ses connaissances pour identifier des problèmes de cette façon. Pour toutes les erreurs trouvées par ces déclencheurs, le découvreur était le développeur le plus expérimenté du groupe, ce qui supporte la sélection de ces déclencheurs.

Au niveau du prochain graphique (Figure 4.9), concernant les déclencheurs pour l'activité de test de fonction, on remarque quelques points intéressants. Premièrement, comme observé dans une figure précédente, la plupart des défauts ont été trouvés par des activités opportunistes. En fait, l'aspect le plus grave est que le déclencheur le plus

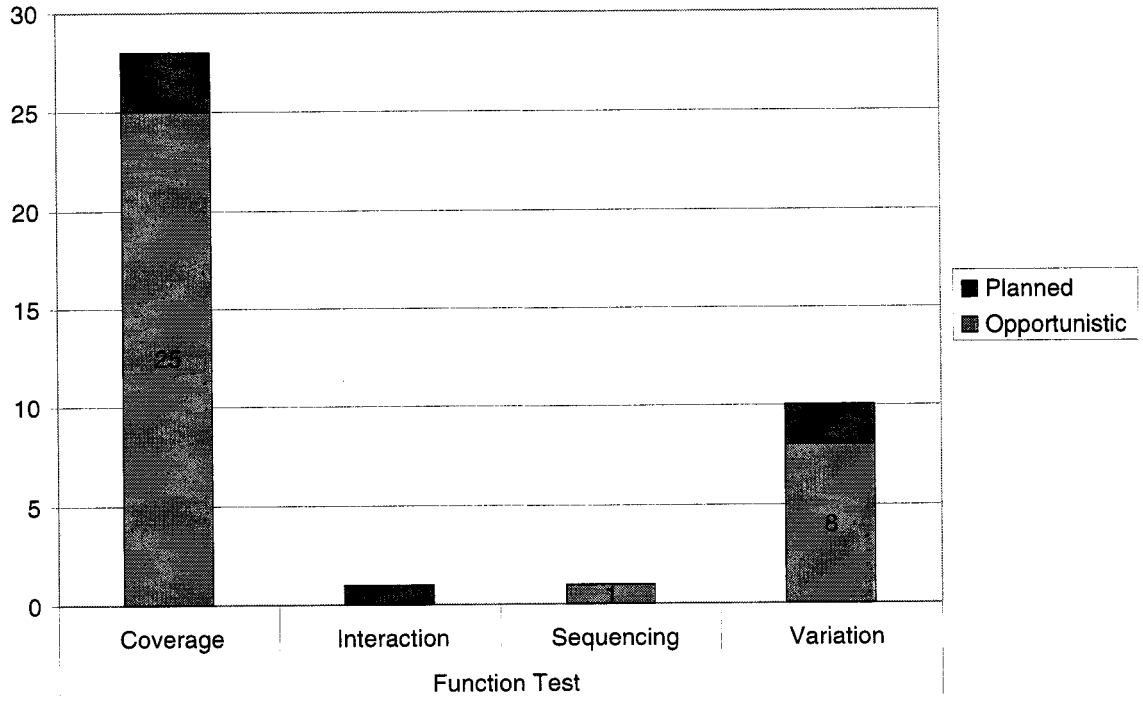

Figure 4.9: Distribution des déclencheurs pour l'activité de test de fonction avec l'information de qualificateur d'activité

important est *Coverage*. Effectivement, comme sa définition l'indique, ce déclencheur est sélectionné pour des tests (ou des exécutions) simples ou la fonctionnalité est invoquée sans paramètre ou avec un seul ensemble de paramètre. La forte proportion d'erreurs trouvées de cette façon indiquerait que plusieurs fonctionnalités ne sont pas testées avant d'être publiées, ce qui n'est pas souhaitable. Le deuxième déclencheur en importance est *Variation*. Ce déclencheur est sélectionné pour des tests avec plusieurs ensembles de paramètres ou des valeurs limites. Il est moins surprenant de trouver ce type de déclencheur pour une activité opportuniste, car l'équipe ayant peu de pratiques

de tests, elle ne teste sans doute pas les cas limites des fonctionnalités. Les deux autres types sont très peu significatifs dans ce cas-ci, donc il n'est pas intéressant de s'y attarder.

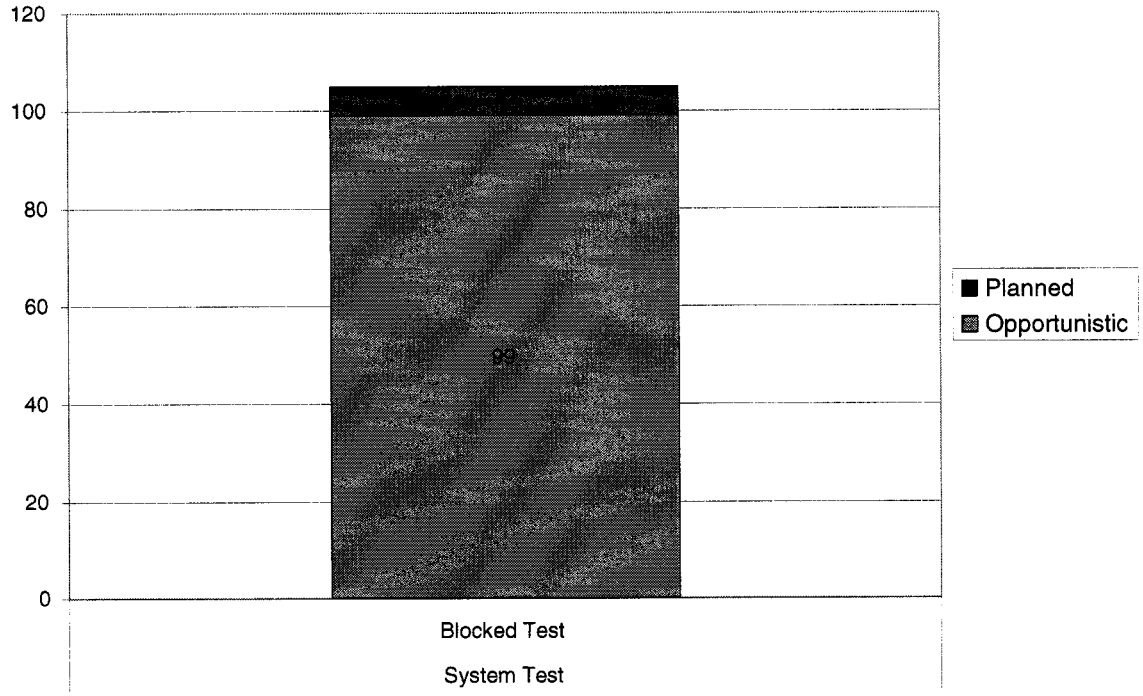

Figure 4.10: Distribution des déclencheurs pour l'activité de test système avec l'information de qualificateur d'activité

La dernière figure (4.10) concernant les déclencheurs présente la distribution des déclencheurs pour les tests système. L'information qui se démarque toutefois le plus dans ce graphique est le fait que toutes les erreurs trouvées par test système ont comme déclencheur *Blocked Test*. Comme il a été mentionné dans l'annexe, ce déclencheur peut être choisi dans plusieurs contextes, dont certains sont spécifiques à notre environnement. Le déclencheur *Blocked Test* est choisi lorsqu'on ne peut entamer un test car le testeur est bloqué ou interrompu par une anomalie n'ayant pas de lien avec le test ou lorsqu'une anomalie est détectée dans un contexte où il est impossible de

déterminer la cause du blocage, ce qui arrive souvent avec les jeux. Un grand nombre de ce type de déclencheur renforce l'hypothèse que l'effort de test est insuffisant dans l'équipe. Effectivement, vu que les développeurs ne testent pas suffisamment leurs produits, il arrive donc fréquemment que leurs tests ne puissent être exécutés et que les clients trouvent des erreurs dans le produit publié, sans comprendre ce qui a conduit à l'anomalie.

Le prochain graphique (Figure 4.11) montre la distribution des impacts des erreurs. Ce graphique illustre qu'une très forte proportion des défauts cause des problèmes de fiabilité du logiciel, ce qui implique que la plupart des défauts proviennent d'erreurs à l'intérieur des fonctionnalités présentes. Les autres impacts sont en très petit

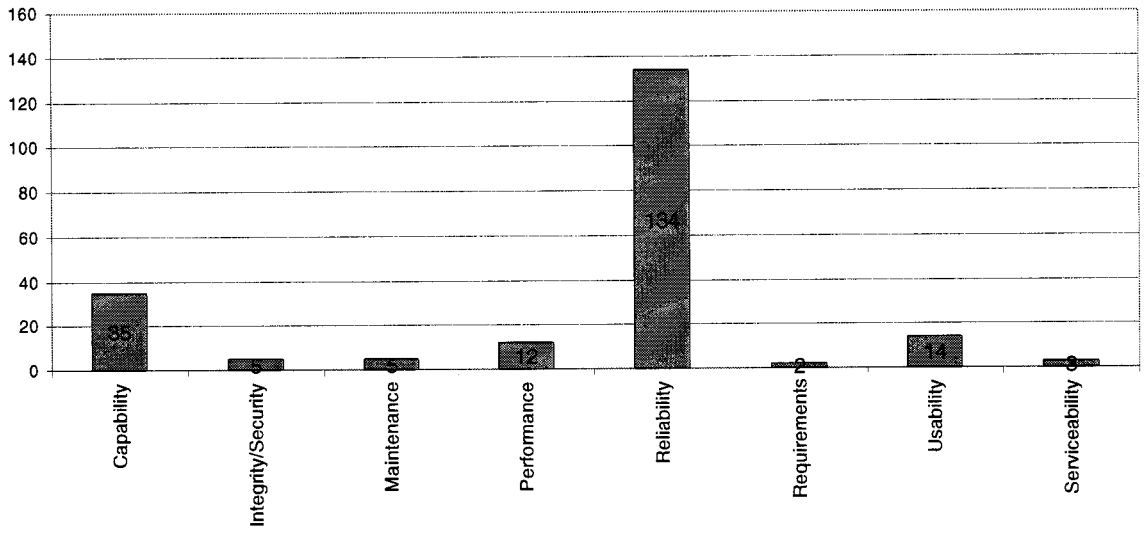

Figure 4.11: Distribution des impacts pour le premier échantillon

nombre par rapport à cet impact. Le fait que la fiabilité soit un impact très récurrent pourrait signifier que les fonctionnalités ne sont pas stables. On pourrait se demander si le grand nombre de choix pour ce paramètre pourrait causer des problèmes de classification. Toutefois, un rapide coup d'oeil à la base montre que bien des défauts

causent des problèmes d'inconstance dans des fonctionnalités, ce qui correspond à ce type d'impact. D'ailleurs plusieurs de ces défauts ont comme description des problèmes de *glitchs* audio, qui sont également du même type d'impact.

Les prochaines analyses concernent les paramètres de fermeture d'ODC. La figure 4.12 présente la distribution des types de défaut.

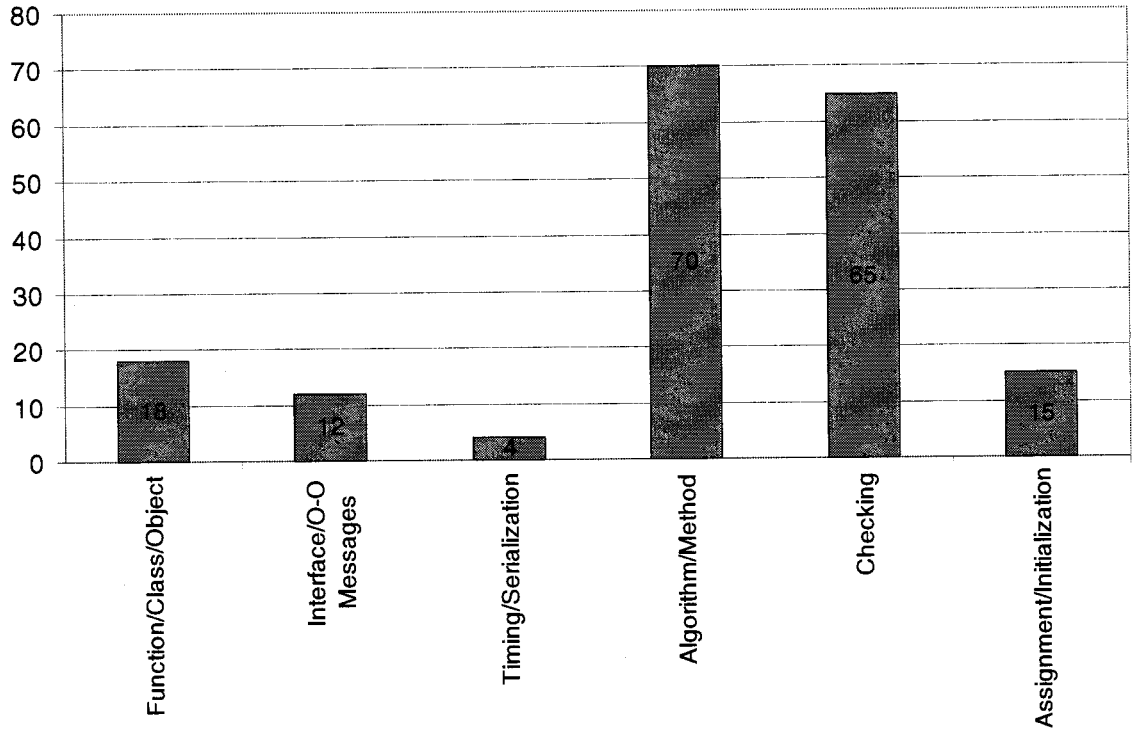

Figure 4.12: Distribution des types de défauts pour le premier échantillon

Il apparaît que les types de défauts les plus dominants sont les types *Algorithm/Method* et *Checking* (voir Tableau 3.2 pour les définitions). Selon la théorie d'ODC, ces types sont plutôt associés à des défauts introduits en aval du processus, soit des erreurs qui sont faites lors de l'implantation du code. La présence majoritaire de ces types de défauts implique deux choses. D'une part, cette distribution est positive en ce

sens que les défauts trouvés ne sont pas des types plus graves, tels que des erreurs de type *Function/Class/Object* ou *Timing/Serialization*, et sont donc des défauts faciles à corriger. Par contre, ces défauts sont certainement trouvés par des activités opportunistes et l'analyse en fonction de celles-ci risque de montrer que ces défauts sont trouvés par le mauvais type d'activité. Cette analyse sera effectuée subséquemment dans ce chapitre.

Ensuite, la Figure 4.13 illustre la distribution des erreurs selon le qualificateur de défaut (Tableau 3.3 pour les définitions).

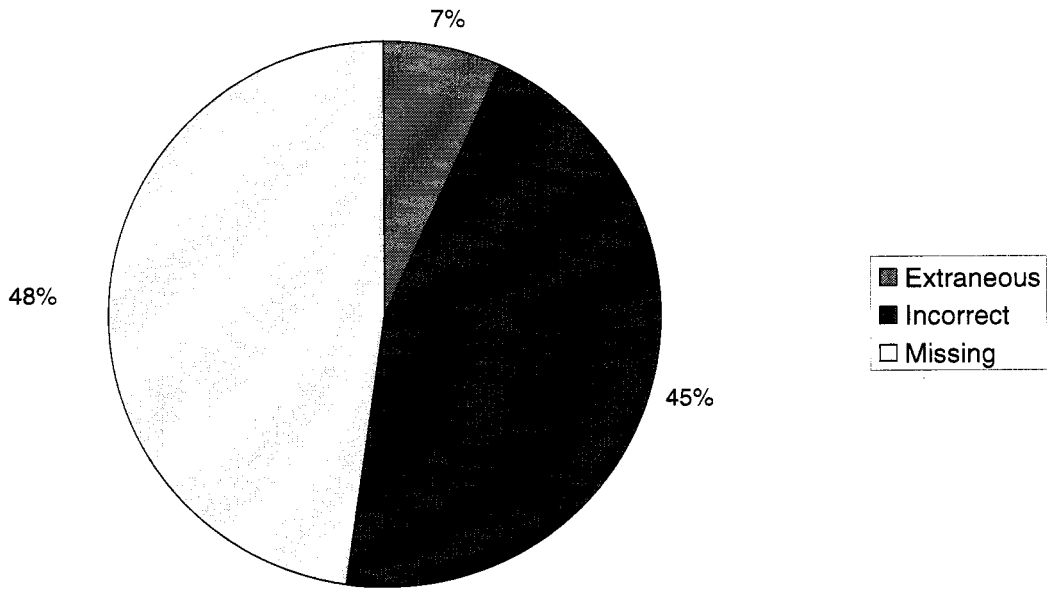

Figure 4.13: Distribution des défauts par qualificateurs pour le premier échantillon

Cette distribution indique que près de la moitié des défauts sont causées par des omissions (*Missing*), tandis que l'autre « moitié » correspond à des défauts incorrects (*Incorrect*). Le petit reste correspond à des erreurs d'éléments superflus (*Extraneous*).

Ceci est tout à fait normal, car cela signifie que les défauts ne sont pas associés à un problème dominant. Ce paramètre est plutôt révélateur lorsqu'il y a un déséquilibre entre les types *Incorrect* et *Missing*. La distribution des *Extraneous* n'est également pas surprenante, car il est normal qu'une faible part des défauts soit due à une surabondance de code.

Le graphique 4.14 présente la distribution du paramètre de l'âge des défauts.

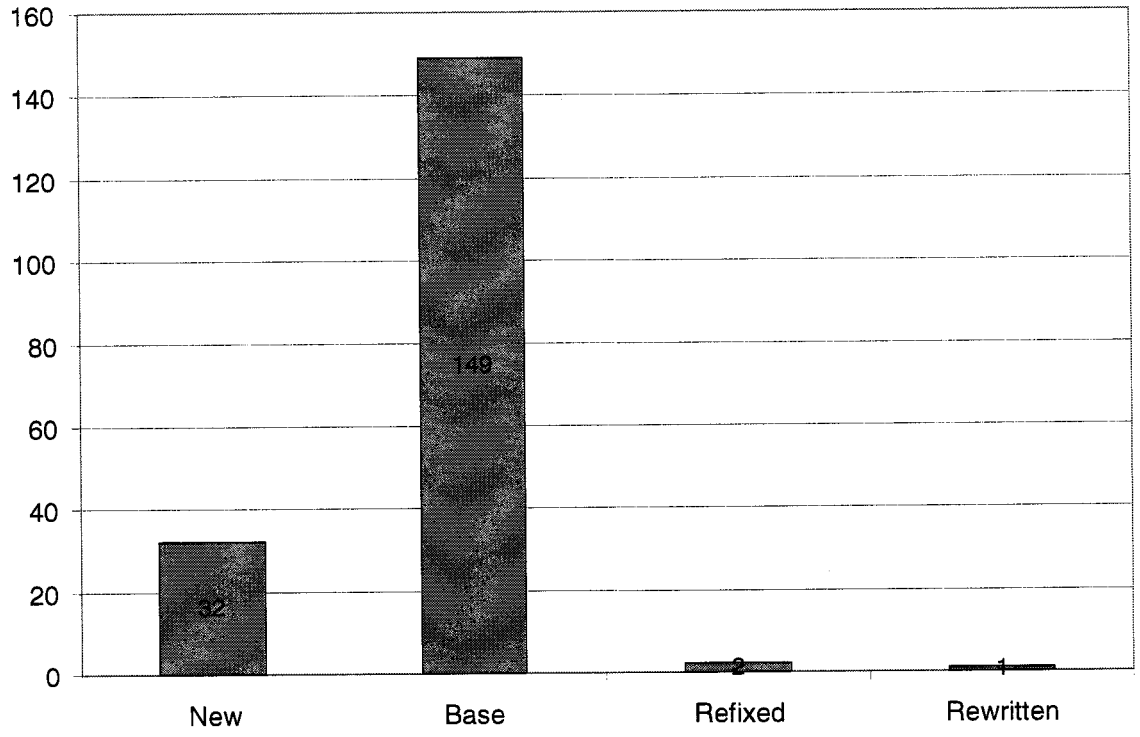

Figure 4.14: Distribution de l'âge des défauts pour le premier échantillon

L'analyse montre que la plupart des défauts sont catégorisés avec le choix *Base*. Ceci implique que les erreurs trouvées font partie d'une fonctionnalité déjà publiée, ce qui confirme les conclusions que nous avons faites pour les autres graphiques. Effectivement, des activités opportunistes risquent de trouver de vieux défauts, ceci sera confirmé dans les études du temps avant découverte. Le nombre de défauts *Refixed* et

Rewritten est peut-être plus bas que prévu, étant donnée l'âge du produit. Il est fort possible que certaines erreurs de classification aient été faites à ce niveau, car les développeurs ne se rappellent pas nécessairement que certains défauts soient liés à des corrections d'anciennes erreurs ou des réécritures de design ou de code. Toutefois, ces défauts se retrouvent sûrement dans la catégorie *Base* à cause de leur âge, ce qui ne cause aucun problème avec notre étude.

Il peut être intéressant de regarder l'âge des défauts avec les découvreurs et les qualificateurs d'activité pour savoir si nous pouvons avoir des informations intéressantes. La Figure 4.15 montre les résultats de l'amalgame de ces informations.

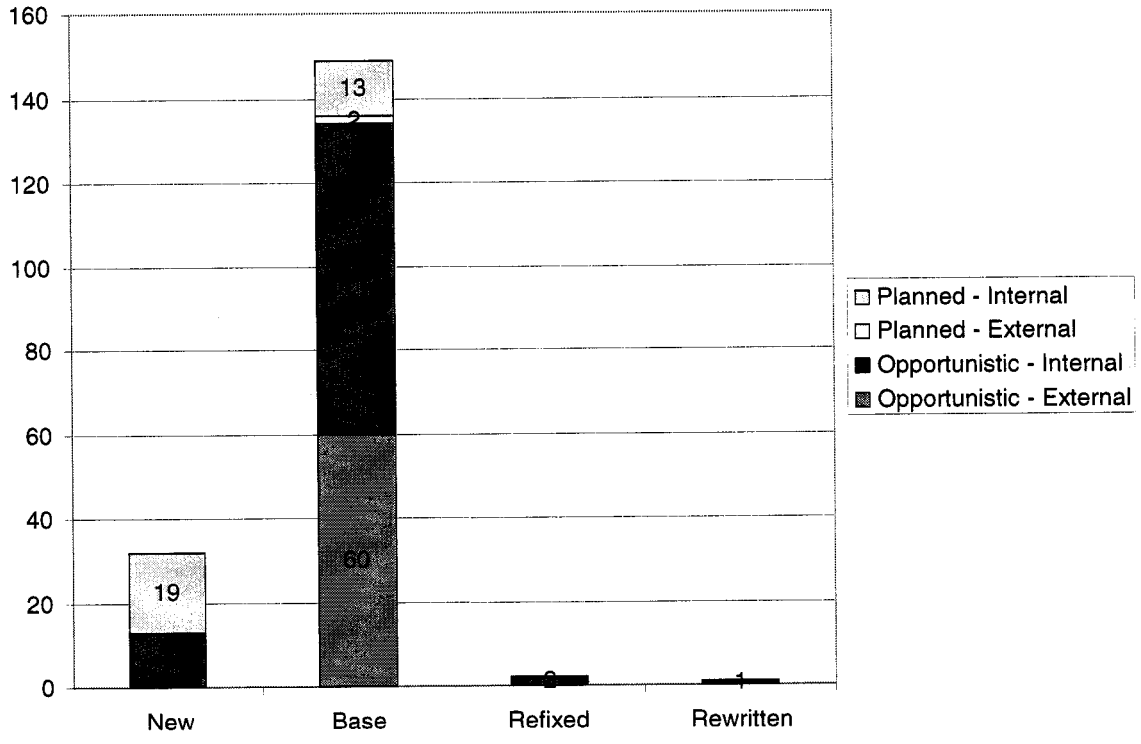

Figure 4.15: Distribution de l'âge avec les qualificateurs d'activité et le découvreur pour le premier échantillon

Il devient apparent que les activités opportunistes trouvent des défauts qui sont plus vieux, ainsi qu'une petite part des défauts qui sont nouveaux. Ceci est aussi symptomatique d'un processus de test insuffisant. La petite part des erreurs planifiées externe est due aux tests que les programmeurs des projets clients effectuent parfois pour chercher des erreurs. Ces activités sont peut-être planifiées, mais les défauts sont tout de même trouvés trop tard par l'équipe.

Le prochain graphique (figure 4.16) montre la source des défauts. Il n'est pas surprenant de constater que la majorité des défauts ont été développés par l'équipe.

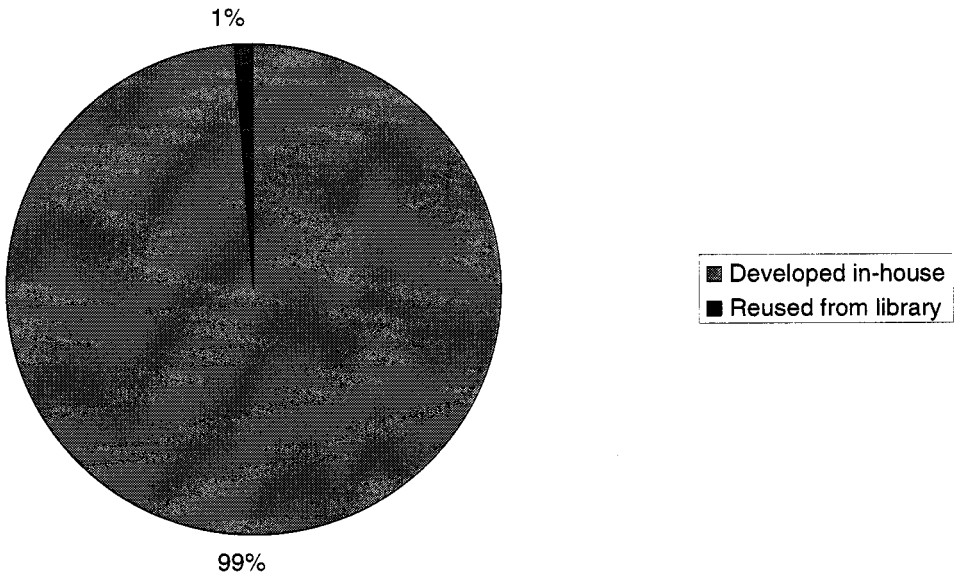

Figure 4.16: Distribution de la source des défauts pour le premier échantillon

Effectivement, très peu de libraires externes problématiques sont utilisées et le code provenant de l'ancienne équipe française n'est presque plus présent. L'autre information

importante que l'on retrouve ici est que l'équipe étudiée est la source principale des défauts.

Pour l'analyse des types de défaut en fonction des activités, une matrice (Tableau 4.1) sera utilisée pour représenter l'information. Ceci permettra de montrer toute l'information sur un seul tableau. Dans cette matrice, plus il y a de défauts dans une catégorie, plus le ton de gris est foncé. Les types de défauts les plus importants, selon la figure 4.12, sont *Checking* et *Algorithm/Method*. Selon la théorie d'ODC, ces deux types

Tableau 4.1: Distribution des types de défaut en fonction des activités de découverte pour le premier échantillon

Act. \ DT	F/C/O	Interface	Timing	Algorithm	Checking	Assign/Init
Design Rev	2	1		3		
Code Insp.	5	3	1	10	21	3
Unit T.		1		1	1	
Function T.	3		1	15	14	3
System T.	8	7	2	41		9

sont typiquement introduits à la phase de l'implémentation. Les activités les plus susceptibles de trouver les défauts de type *Checking* sont l'inspection de code et le test unitaires. D'autre part, les activités de découverte associées au type *Algorithm/Method* sont l'inspection de code, le test unitaire et le test de fonction.

Pour le type *Checking*, la matrice révèle que la plupart (66%) des défauts de ce type sont trouvés par des tests système et des tests de fonction. Même si le restant de ces défauts est trouvé généralement par des inspections de code, une trop grande proportion de ceux-ci sont trouvés trop tard. Du côté des défauts de type *Algorithm/Method*, la

plupart des défauts sont trouvés par tests système, avec une plus faible proportion pour les activités d'inspection de code et de test de fonction. Ceci est également un signe de problèmes au niveau du processus de test, car cela implique que des défauts sont trouvés dans les fonctionnalités alors qu'il était attendu que celles-ci soient fonctionnelles. Les autres types de défaut sont délaissés à cause de leur poids statistique. Il est important toutefois de remarquer la forte présence des tests systèmes dans leur découverte.

En fait, le vrai problème montré par cette dernière analyse découle du fait que les défauts sont découverts de façon opportuniste. Effectivement, une découverte opportuniste signifie probablement qu'un défaut est découvert après un effort de test quelconque et surtout d'une façon inattendue. Les résultats montrés par ce tableau renforcent la tendance qui était observée dans les autres analyses : les défauts sont trouvés trop tard et par hasard. Les prochaines analyses, concernant les temps avant découverte, devraient également supporter ces résultats.

Comme mentionné précédemment, le temps avant la découverte d'un défaut est calculé en soustrayant la date d'introduction d'un défaut à la date à laquelle il a été entré dans la base. La valeur résultante est un nombre de jours calendrier, ce qui signifie que ce nombre inclut les fins de semaine et les journées de congé. Tandis que cette méthode est moins précise dans l'évaluation réelle du temps, elle est beaucoup plus simple à calculer et plus près du temps réel perçu par les développeurs et autres personnes impliquées. Également, nous avons découvert que plusieurs défauts dans l'échantillon avaient été introduits bien avant l'utilisation des outils de gestion de configuration. Pour ces défauts, nous soustrayons la date du 23 novembre 2001 à la date de découverte, car c'est à cette date que la première gestion de configuration a été mise en place. Encore une fois, ceci est une technique imprécise pour calculer le temps, mais au moins, les très vieux défauts sont conservés dans notre étude. Ceux-ci sont d'ailleurs représentés dans une autre couleur et sont nommées données « Incomplète » dans les légendes des

graphiques. La première distribution du temps avant découverte est illustrée à la figure 4.17. Ce graphique présente l'ensemble des données sur une échelle uniforme.

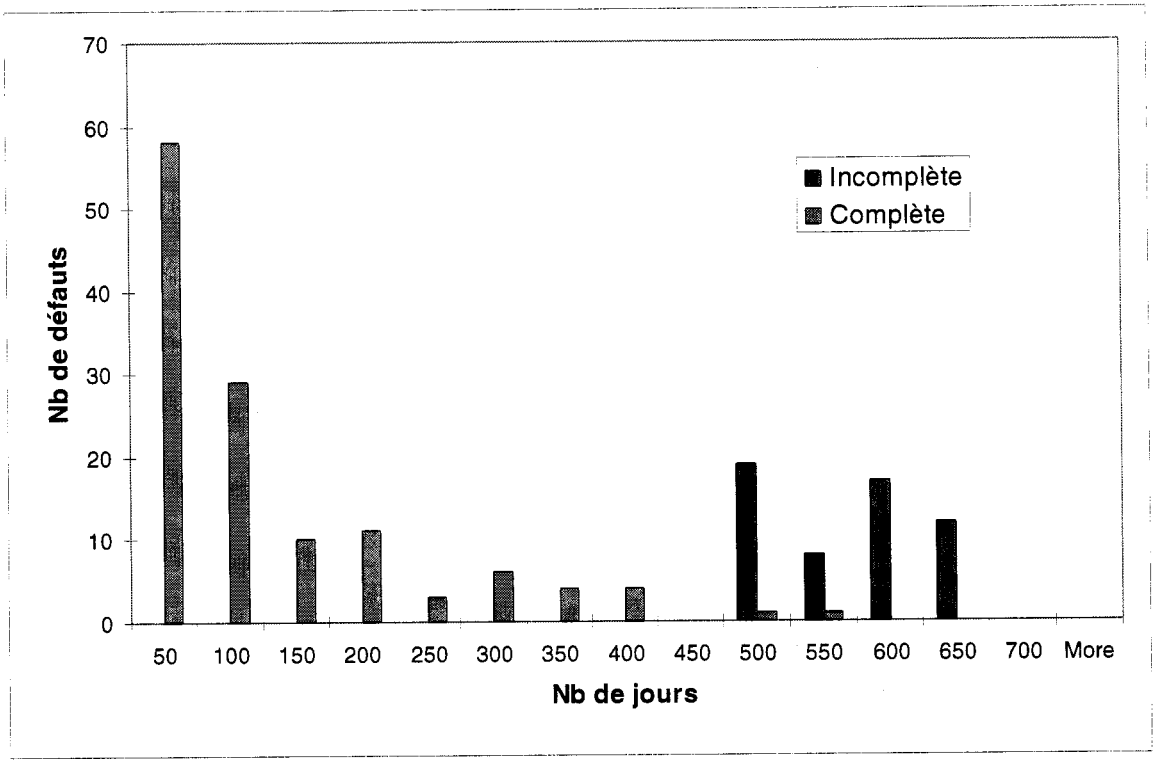

Figure 4.17: Distribution des temps avant découverte pour l'échantillon entier (premier échantillon)

Cette figure illustre que l'âge des défauts découvert varie entre quelques jours et deux ans. On voit également que beaucoup de défauts appartiennent à la catégorie des données incomplètes. Comme cette échelle est très grande, il est difficile de trouver une tendance pour les défauts plus jeunes. Les prochaines figures présentent les données en deux parties : la première (figure 4.18) montre les défauts les plus jeunes, c'est-à-dire les défauts ayant été découverts en moins de 200 jours, ce qui représente 60% des défauts avec une échelle à 5 jours près, tandis que la deuxième montre les défauts plus vieux avec la même échelle de 50 jours.

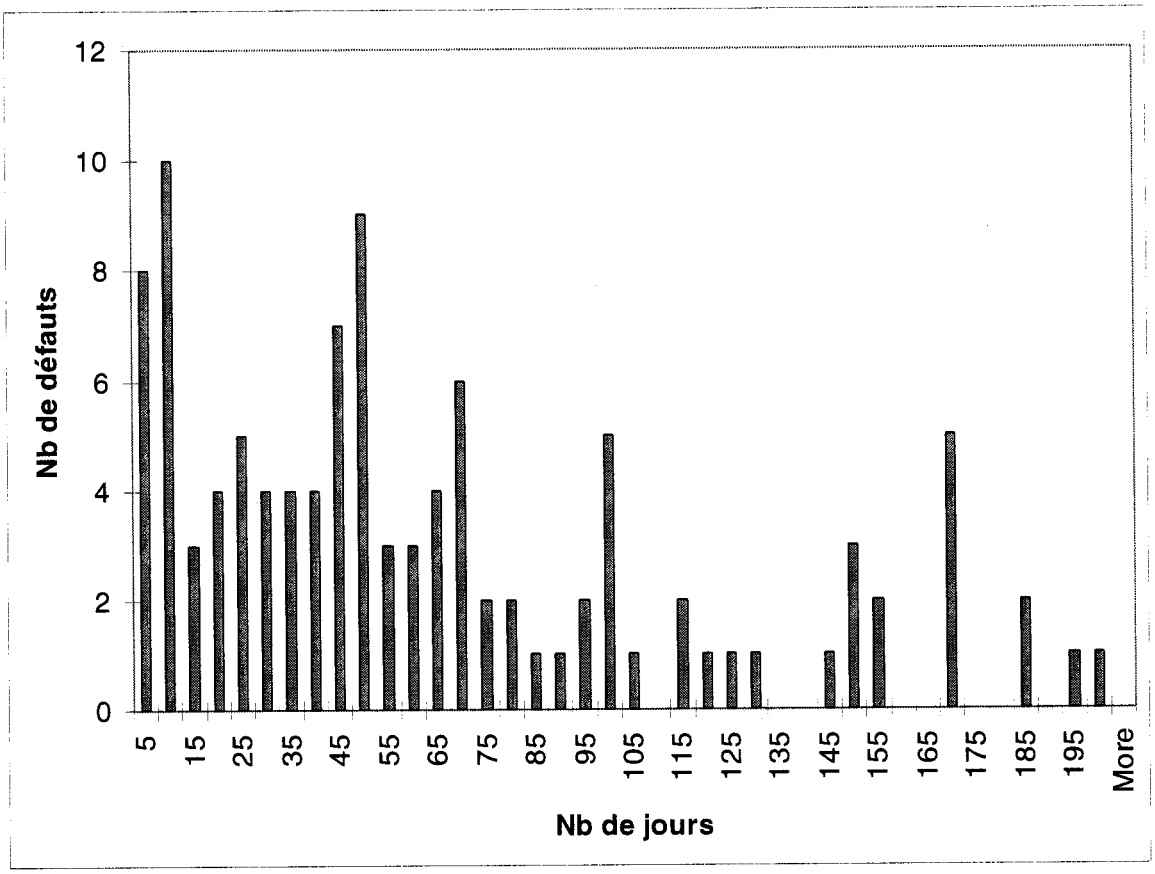

Figure 4.18: Distribution du temps avant découverte pour les défauts les plus jeunes pour le premier échantillon

Sur cette première figure, il est plus apparent que le temps avant découverte varie beaucoup. Il ne semble pas y avoir de distribution reconnaissable : certains défauts sont trouvés rapidement, tandis que d'autres sont trouvés après plus d'un mois, ce qui est assez long. Des analyses supplémentaires seront nécessaires pour voir si l'opportunisme jour un rôle dans le temps avant la découverte, comme semble l'indiquer les autres analyses. Sur la deuxième figure, la distribution des données incomplètes est plus visible. Ceci ne dit pas grand-chose, car certains défauts pourraient avoir jusqu'à sept ans, mais cette distribution donne une idée de a vieillesse de défauts.

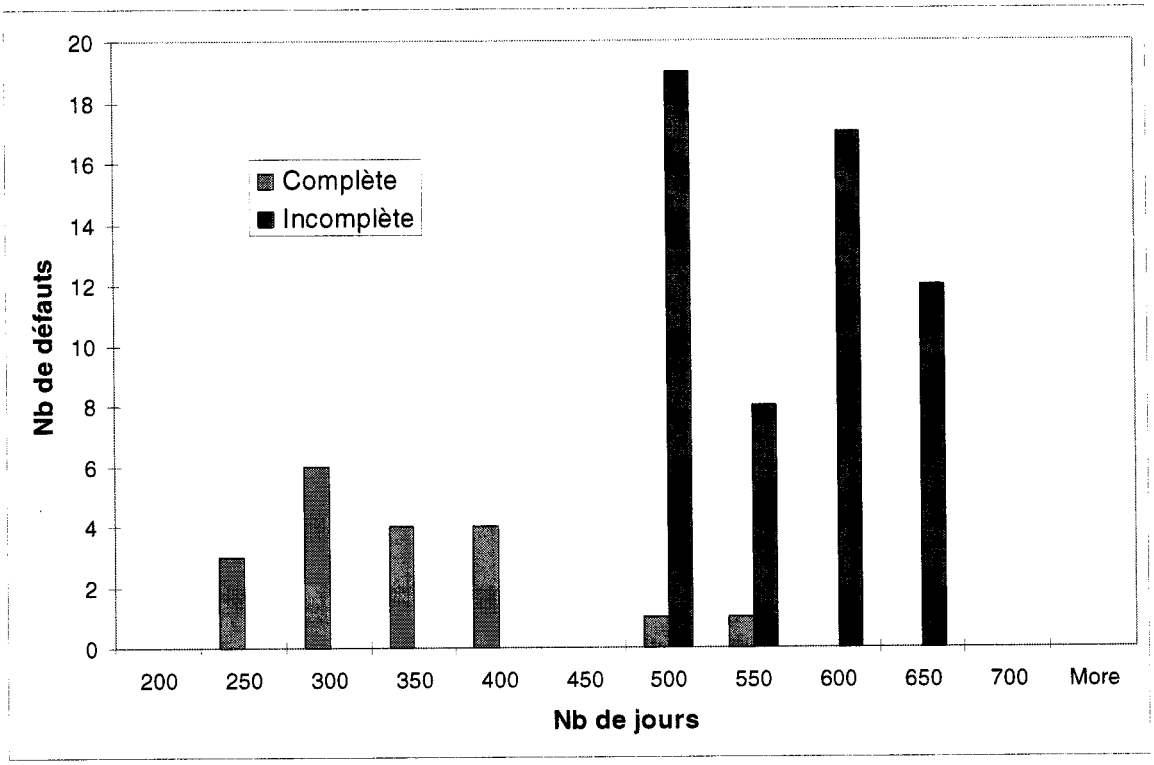

Figure 4.19: Distribution du temps avant découverte pour les défauts les plus anciens pour le premier échantillon

Afin de mieux comprendre ce qu'indique le temps avant la découverte, les prochaines analyses incluent les qualificateurs d'activités afin de voir si une tendance se dessine. Dans ces graphiques, les données complètes et incomplètes ne sont pas différenciées, mais il ne sera pas difficile de savoir où les données incomplètes se trouvent. L'approche est la même que précédemment : une figure présente l'échantillon complet et l'autre présente les défauts les plus jeunes.

La figure 4.20 illustre le temps avant découverte avec l'information de qualification d'activité pour l'échantillon complet. Évidemment, la majorité de l'échantillon présente des défauts trouvés de façon opportuniste, il est donc normal que

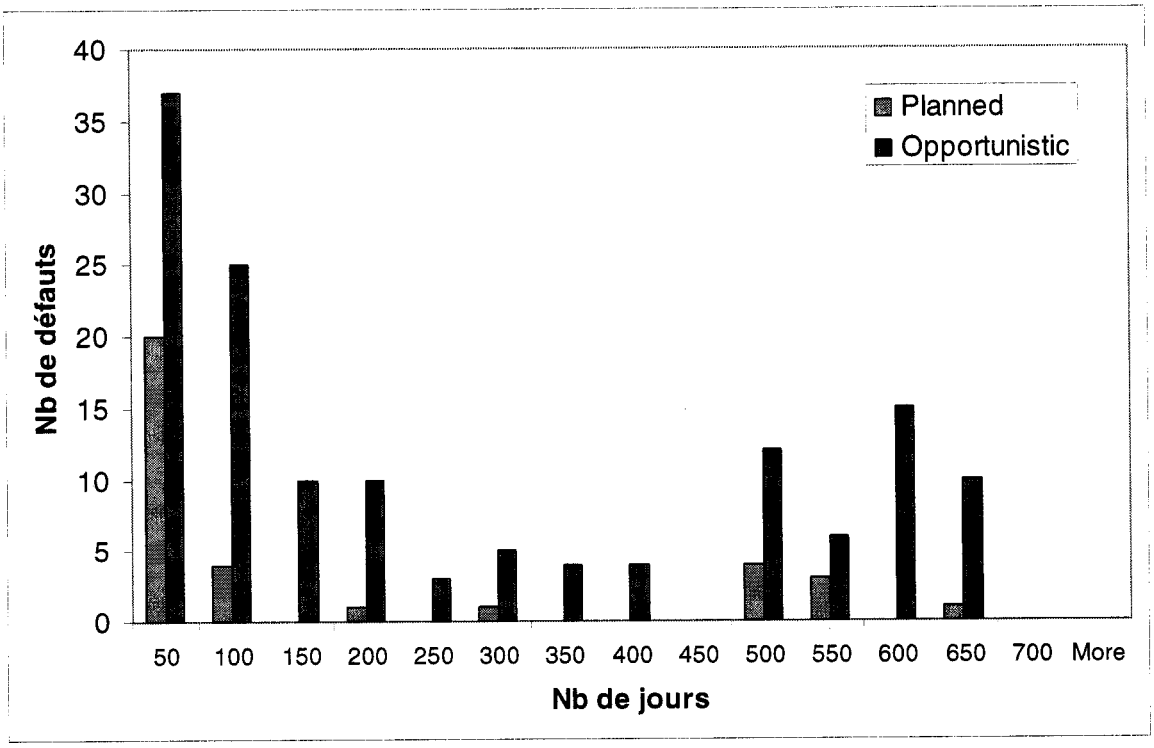

Figure 4.20: Distribution du temps avant découverte avec qualificateur d'activité pour le premier échantillon

ceux-ci soient très présents dans la distribution. D'ailleurs, la majorité des défauts trouvés par les deux types d'activité sont dans la tranche de 50 jours et moins. Les anciens défauts trouvés par activités planifiées peuvent être expliqués par le contexte de maintenance de l'équipe.

Ce dernier graphique à la figure 4.21 présente le temps avant découverte pour les jeunes défauts. Il devient alors apparent que l'opportunisme des activités rallonge le temps de découverte. Il est vrai que certains défauts sont trouvés rapidement de façon opportuniste, mais leur distribution est beaucoup plus large que celle des défauts trouvés par des activités planifiées.

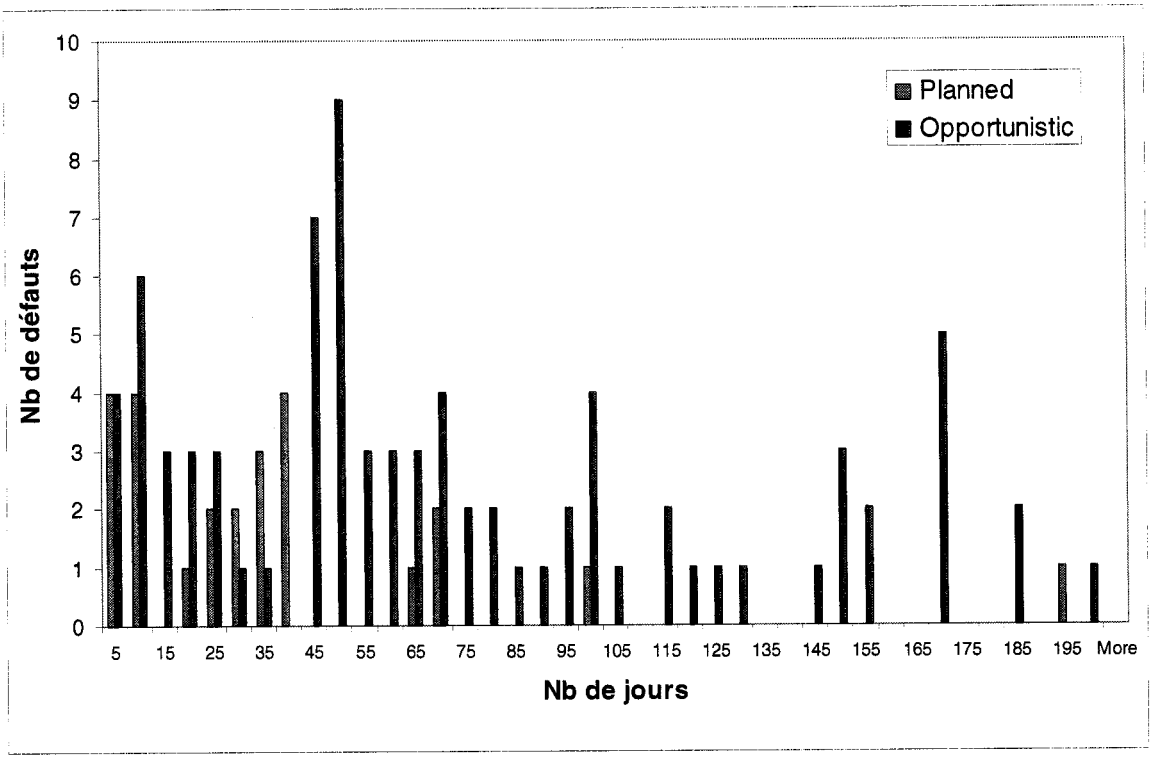

Figure 4.21: Distribution du temps avant découverte inférieur à 200 jours avec la qualification d'activité pour le premier échantillon

Cette dernière analyse confirme la tendance qui a été observée dans ce chapitre, que les défauts sont trouvés trop tard. La raison principale qui pourrait expliquer ceci est que l'effort de test n'est pas suffisant et que, par le fait même, les erreurs sont trouvées par hasard par les développeurs, ainsi que par les clients. Une analyse en fonction des activités de découverte elles-mêmes pourrait être faite, mais les résultats indiqueraient sûrement le même problème d'opportunisme.

Ainsi, en guise de conclusion pour ce chapitre, on remarque qu'il y a plusieurs aspects à améliorer dans le cadre du processus de test de l'équipe étudiée. Effectivement, un trop grand nombre de défauts sont trouvés par des activités opportunistes, donc bien souvent par hasard. De plus, les activités de découverte dominantes sont souvent beaucoup trop en aval par rapport aux types de défauts

principaux qui ont été enregistrés. À partir de ces informations, il est possible d'envisager un plan d'amélioration. Dans le chapitre suivant, ce plan est mis en place pour tenter d'améliorer à la situation.

CHAPITRE V : PROPOSITIONS DE MODIFICATIONS AU PROCESSUS DE TEST.

Suite à l'analyse faite au chapitre IV, il est possible de proposer des changements au processus de l'équipe qui s'attaquent aux problèmes découverts. Ainsi, ce chapitre est dédié à la présentation et à la discussion des changements qui seront apportés. Il sera d'abord question d'identifier les objectifs exacts d'amélioration. Ensuite, les possibilités de changements sont approchées et finalement, un plan de changement est élaboré.

5.1 Objectifs d'amélioration

À partir des résultats quantitatifs obtenus au dernier chapitre, des objectifs d'améliorations mesurables peuvent être identifiés. Avec ces objectifs en main, il est possible d'identifier des analyses clé qui peuvent supporter l'impact des changements. Certaines métriques sont également proposées en plus d'ODC, mais ne font pas l'objet de cette étude. Les améliorations proposées ici ont été présentées et raffinés avec l'aide des membres de l'équipe.

Le premier objectif identifié concerne la réduction de la proportion des défauts trouvés par les clients. Effectivement, même si ce pourcentage n'est pas majoritaire, une grande proportion des défauts étaient tout de même trouvées à l'externe. Ainsi, l'objectif serait de diminuer la proportion des anomalies trouvées à l'externe et par le fait même le nombre d'anomalies trouvées dans une version publiée.

Le deuxième objectif est de réduire la proportion des défauts qui sont découverts de façon opportuniste. Effectivement, une découverte opportuniste signifie généralement que le défaut a été trouvé beaucoup plus tard qu'il n'aurait dû et souvent par un client. Une réduction notable de l'opportunisme des activités est donc souhaitable.

Finalement, il est également intéressant de diminuer la proportion des erreurs trouvées par des tests systèmes, en particulier par le déclencheur *Blocked Test*. Vu que les principaux types de défauts trouvés doivent avoir des activités de découvertes principales plus en amont du processus de développement, il est important de diminuer le nombre de tests système. À noter que trouver des erreurs par test système n'est pas nécessairement une mauvaise chose, selon le type de défaut. Toutefois, le déclencheur *Blocked Test* implique presque toujours une certaine part d'opportunisme et d'activités non contrôlées. Ainsi, une diminution de la proportion de ce déclencheur est un objectif certainement intéressant, qui va de pair avec la diminution des activités de test système.

La figure 5.1 rappelle la distribution actuelle des activités de découverte, en fonction des qualificateurs d'activité. Les objectifs sont donc de réduire les tests système et l'opportunisme des activités en général.

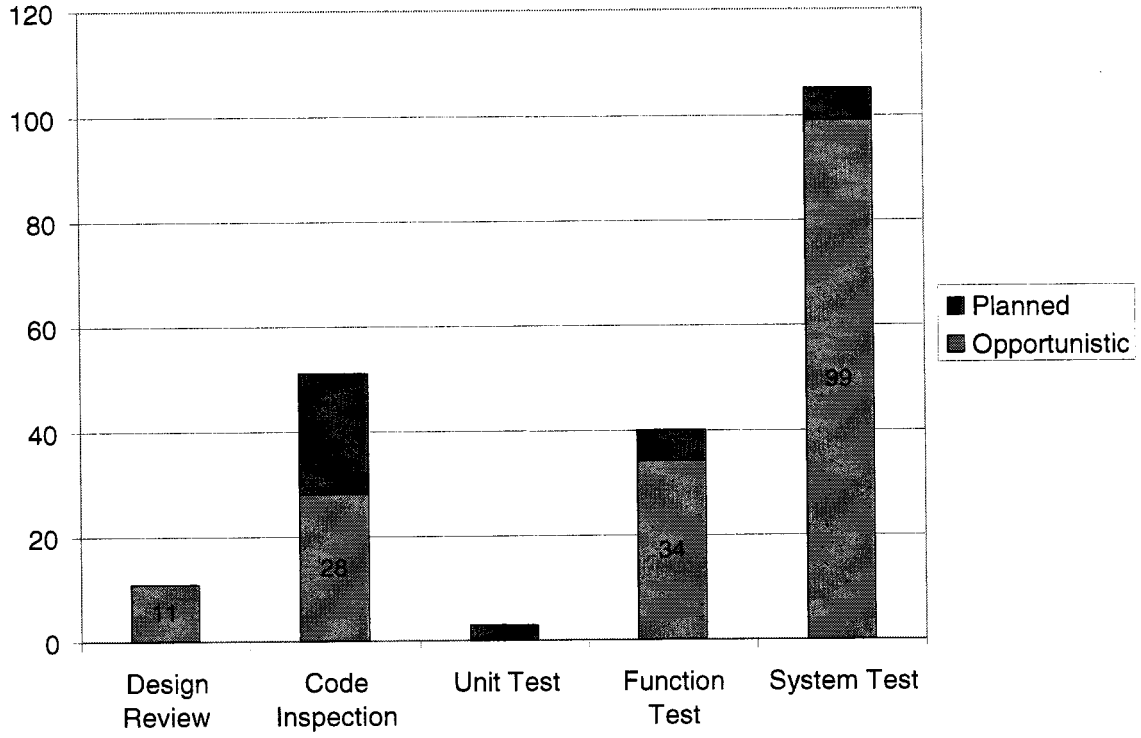

Figure 5.1: Distribution des activités de découverte en fonction du qualificateur d'activité

Il est très clair qu'à la lumière des résultats obtenus et des objectifs identifiés que le problème réside dans le fait qu'il y a un manque de sensibilisation à l'importance du test au sein de l'équipe. Ainsi, une partie du travail d'amélioration est de mettre en place un processus plus formel de test et de vérification, ainsi que de prévoir du temps dans la planification de l'équipe pour ces activités. Dans la prochaine section, sont présentées certaines discussions relatives à l'amélioration du processus de test.

5.2 Une discussion sur les changements

Améliorer un processus n'est pas une chose simple. Que ce soit implanter de nouvelles pratiques de développement, de nouvelles métriques ou des nouveaux pipelines de production, les changements doivent être adaptés à la situation. De plus, il ne suffit pas seulement de décider d'un ensemble de changements et de les implanter tout de suite et soudainement. Il y a souvent un travail à faire auprès des équipes concernées pour s'assurer que la transition se fasse sans trop de problèmes. Dans cette section, les améliorations que peuvent être implantées dans l'équipe étudiée sont présentées.

Comme mentionné plusieurs fois au cours de cet ouvrage, l'équipe étudiée n'a pas de processus formel, surtout au niveau des activités de test. Ceci constitue, en fait, une lame à double tranchant. D'une part, les membres de l'équipe sont ouverts à toute forme de pratique qui est susceptible de rendre leur développement et leur test plus efficaces. D'autre part, l'absence d'activités contrôlées, que celles-ci soient de test ou de d'autres disciplines, favorise, en apparence, un temps de développement plus court et l'ajout de nouvelles pratiques semblera ralentir la cadence de travail habituelle. Dans ces situations, il est donc plus logique de construire un plan qui intégrera de nouvelles pratiques par étapes.

Tout d'abord, un survol des activités qui peuvent être améliorées ou instaurées dans l'équipe est effectué. L'approche utilisée est de se concentrer sur les mêmes activités que celles utilisées dans ODC. À cause de l'absence générale des pratiques de tests, il est sans soute préférable que les méthodes proposées soient des techniques de test simples, appuyés sur des concepts de base.

Au niveau des révisions de design, celles-ci sont déjà effectuées au niveau de l'équipe, mais il y a place à amélioration. En ce moment, seulement certains designs plus complexes sont révisés, alors que de façon générale, le design du logiciel est peut-être la partie la plus cruciale du développement. Une amélioration serait de premièrement rendre cette étape obligatoire pour tous les développements. Ensuite, il serait intéressant que ces révisions soient faites en fonction des requis. Le seul problème est que les requis ne sont pas formalisés non plus, ce qui peut rendre la chose plus difficile. Finalement, il serait intéressant de mesurer l'efficacité de cette activité en calculant le temps passé sur les revues, la grandeur des designs révisés (plus difficile) et le nombre de défauts trouvés par session de révision.

Au niveau des inspections de code, la pratique est également déjà implantée dans l'équipe. Certaines améliorations peuvent toutefois être apportées à cette activité. La mise en place d'une méthode plus complète, telle que des *walkthrough*, ou l'augmentation du nombre de vérificateurs sont des options intéressantes. Également, le nombre de ligne de code vérifiées, les temps passé par chaque vérificateur sur le code révisé et le nombre d'erreurs trouvées peuvent être comptabilisés pour mieux contrôler cette activité.

Les tests unitaires sont une pratique très ad hoc dans l'équipe en ce moment. Les erreurs qui ont été trouvés par cette activité l'ont été plus par hasard que par pratique conventionnelle. L'implantation de ce type d'activité de test peut être intéressante. Premièrement, les développeurs ont une connaissance intime du produit développé, ce qui favorise cette pratique. Également, les tests unitaires sont généralement beaucoup plus près des développeurs que les tests de fonction, qui demandent une intuition différente. Ces tests sont préparés par le développeur du code à tester et sont effectués dans une optique de couverture du code, selon des techniques reconnues (Beizer 1990). Ces tests peuvent être scriptables et automatisables. Finalement, cette activité peut être

vérifiée en mesurant le temps de préparation des tests, le temps d'exécution des tests, le nombre de tests construits et exécutés et le nombre d'erreurs trouvées.

Des activités de test de fonction peuvent être intégrées dans les nouvelles pratiques. Ces tests en boîte noire, basés exclusivement sur les spécifications d'un développement, peuvent être préparés et effectués par un développeur indépendant d'une fonctionnalité développée. Il y a, par contre, le problème des requis non formalisés. D'autre part, les tests de fonctions peuvent être effectués manuellement pour tester les outils, tandis que du côté du moteur, les développeurs peuvent user des outils d'exécution automatique que l'équipe possède déjà. Cette activité peut être contrôlée en mesurant le temps de préparation des tests, le temps d'exécution des tests, le nombre de tests construits et exécutés et le nombre d'erreurs trouvées, comme pour les tests unitaires.

L'analyse du processus montre également une absence de tests système organisés. Ce type de test, dont le but est de faire fonctionner les produits dans un environnement réel, peut être implanté dans l'équipe. Ces tests peuvent vérifier principalement la performance des produits, vu que c'est sans doute l'aspect le plus important de ceux-ci. Comme pour les tests de fonction, les outils peuvent être utilisés pour encoder les tests pour le moteur. Des cas de tests à la main peuvent être conçus pour le logiciel *SoundEditor*. Finalement, les mêmes métriques que pour les tests de fonction peuvent être mesurées.

Comme l'équipe développe un produit qui est constamment en évolution, il est également intéressant de mettre en place un système de tests de régression, préférablement automatisable. Ainsi, l'équipe peut régulièrement vérifier le bon fonctionnement d'anciennes fonctionnalités, voire seulement les plus critiques, au fil de l'intégration de nouvelles composantes.

Toutes ces propositions ne peuvent évidemment être implantées en même temps, car cela impliquerait un trop grand chambardement des pratiques. Dans la prochaine section, les changements qui sont implantés pour cette recherche sont présentés

5.3 Plan des améliorations

Suite à une analyse de ce qui est possible et une discussion avec certains membres de l'équipe étudiée, nous en sommes arrivés à un plan de changements de pratiques. Dans cette section, le plan des changements à venir dans l'équipe est présenté. L'ordre dans lequel ces changements sont apportés est justifié.

Il est à noter que, comme le temps est limité, cette étude se concentre exclusivement sur les effets de la première phase de changements. L'intégration des changements proposés à la section précédente est prévue en trois étapes. Idéalement, chaque étape sera suivie par une analyse statistique des défauts trouvés durant cette période.

La première phase concerne l'intégration de pratiques relatives aux tests unitaires, à l'instauration d'une base de tests (répertoire des tests unitaires conçus et effectués), à la mise en place de la régression de ces tests et à la cueillette de métriques supplémentaires relatives à ces nouvelles activités. La planification des tests dans la cédule de l'équipe devient également une pratique nécessaire. Dans la deuxième phase, nous prévoyons améliorer les pratiques relatives aux inspections de code, implanter les tests de fonction et ajouter les métriques relatives à ces activités. La troisième phase inclura la mise en place de tests système, une amélioration des techniques de révision de design et l'ajout des métriques, comme pour les phases suivantes.

La première étape d'amélioration du processus de test est par l'implantation des tests unitaires, au profit d'autres activités plus en amont du processus de détection des erreurs, telles que les inspections de code ou des revues de design, et ce, pour plusieurs raisons. Premièrement, les types d'erreur dominants qui sont identifiés dans la première partie de l'étude sont associés, entre autres, aux tests unitaires dans la théorie d'ODC. Effectivement, les défauts liés aux conditionnelles (*Checking*) et les défauts d'algorithme peuvent être trouvés par test unitaire. L'autre activité favorisant la découverte de ces types de défauts est l'inspection de code. La raison pour laquelle il a été décidé de repousser l'amélioration des inspections à la deuxième étape concerne la difficulté de ce type d'activité. Effectivement, tandis qu'il est généralement moins coûteux d'effectuer des inspections, celles-ci sont plus difficiles à maîtriser. Accomplir des inspections de code efficaces nécessite de l'expérience, une certaine organisation et une certaine intuition de la part des inspecteurs. D'autre part, les tests unitaires sont beaucoup plus près des connaissances et des habitudes de développement des développeurs de l'équipe. L'hypothèse est que, si les développeurs comprennent et appliquent bien les tests unitaires, ils seront sans doute de meilleurs inspecteurs de code par après.

La même logique s'applique pour les tests de fonction. Ce type de test est souvent décrit comme étant une autre discipline, différente du développement, car ils nécessitent une façon de penser qui est différente. L'introduction des tests unitaires pourrait alors développer ce sens du test chez les développeurs. Finalement, nous avons choisi d'améliorer les revues de design et d'incorporer les tests système à la fin seulement, à cause de la moins grande présence de leurs types de défauts associés.

5.4 L'implantation des changements

Dans la dernière section de ce chapitre, les recommandations pratiques pour l'implantation des tests unitaires et autres techniques sont présentées. Les changements décrits ici sont ceux qui sont implantés pour la deuxième partie de l'étude.

Le premier aspect qui a été implanté est l'inclusion des tests unitaires dans la planification de l'équipe. Comme ces tests deviennent effectivement une nouvelle étape, ils doivent être intégrés au plan de développement. Ceci implique, entre autres, que les membres de l'équipe doivent évaluer le temps qu'il leur faudra pour accomplir ces nouvelles tâches. Comme ils n'ont aucune information ni expérience sur comment faire cela, nous nous attendons à ce que ceci soit moins précis au début. Deuxièmement, des métriques sont prélevées à la fin d'une session de tests unitaires, afin de contrôler l'efficacité de cette pratique. Les métriques qui sont prélevées sont celles qui ont été décrites dans une section antérieure. Troisièmement, il faut intégrer les projets de tests unitaires dans la gestion de la configuration, entre autres pour éventuellement considérer l'automatisation et la régression de ces tests.

Au niveau de la conception des tests unitaires, il existe plusieurs techniques relevant du concept de couverture du code, qui peuvent être utilisées. Les techniques recensées dans la littérature sont les suivantes :

Tableau 5.1: Présentation des avantages et désavantages des différentes techniques de test unitaire

Type de couverture	Avantages / Inconvénients
Instruction	A : Simple I : Ne couvre pas nécessairement tout le code
Décisions	A : Simple I : Ne couvre pas nécessairement tout le code
Conditions	A : Relativement simple I : Ne couvre pas nécessairement tout le code
Conditions multiples	A : Simple avec les tables de vérités Devrait tout couvrir le code I : Nombre exponentiel des cas de tests, trop long
Condition/Décision	A : Couvre bien tout le code. Nécessite un nombre moins grand de cas de tests pour le même résultat que conditions multiples I : Demande un travail plus grand au niveau conceptuel
Chemins	A : Couvre tout le code et toutes les possibilités d'exécution I : Plus prohibitif au niveau pratique, à cause du nombre de chemins possibles
Boucles	A : Technique complémentaire pour traiter les boucles I : Peut augmenter le nombre de cas rapidement
Par base de chemin (Watson 1996)	A : Technique qui couvre les chemins importants selon une méthode mathématique I : Compliquée à comprendre Difficile à implanter sans un outil approprié

Après une présentation des techniques à l'équipe, nous avons convenu d'utiliser seulement certaines techniques. En premier lieu, pour les fonctions où le nombre de décision n'est pas trop grand, ou lorsque les décisions sont interreliées de façon à réduire le nombre de chemins, la technique par couverture de chemin est utilisée. Autrement, la technique utilisée est celle de la couverture par décision/condition. Ensuite, dans tous les cas, une couverture au moins partielle des boucles est nécessaire.

Il est certain que les décisions qui ont été prises dans ce chapitre risquent d'être modifiées par la volonté des développeurs impliqués. Si tel est le cas, les chapitres ultérieurs spécifieront les changements qui ont été apportés à ce plan de travail. Cette nouvelle méthode de travail étant en place, les impacts de ces implantations sont présentés dans le chapitre suivant.

CHAPITRE VI : LES RÉSULTATS DE LA DEUXIÈME PARTIE

Ce chapitre présente les résultats obtenus lors de la deuxième période d'échantillonnage. Une discussion sur l'implantation des changements présentés dans le chapitre précédent constitue la première partie de ce chapitre. Ensuite, l'analyse du deuxième échantillon est présentée. Tout comme dans le chapitre III, l'échantillon est d'abord décrit de façon générale, puis les analyses en fonction des paramètres ODC, ainsi que des paramètres supplémentaires, sont effectuées. Les comparaisons aux résultats de la première partie sont faites, ainsi que les interprétations de ces comparaisons. Dans la dernière partie, des analyses supplémentaires sont présentées pour expliquer les changements enregistrés dans cette deuxième partie de l'expérience.

Avant de commencer les analyses, il est important de revenir sur les objectifs identifiés à la fin de la première analyse. Nous avons identifié trois objectifs pour cette deuxième partie. Il est premièrement question d'une réduction de l'opportunisme dans les découvertes des défauts. Le deuxième objectif identifié est de réduire la proportion des défauts trouvés par System Test, Blocked Test. Finalement, le troisième objectif est de réduire le nombre de défauts trouvés à l'externe. Mais avant tout, le but est de tenter d'identifier les changements entre les deux analyses et de l'expliquer. Voici un tableau regroupant les analyses les plus importantes de ce chapitre, avec un aperçu des résultats obtenus.

Tableau 6.1: Résumé des analyses principales du chapitre avec résultats sommaires

Analyse	**Sommaire des résultats**
Distribution des activités de découverte	Faible augmentation des tests unitaires Statu quo pour les autres activités
Distribution des découvreurs	Diminution des découvertes externes
Distribution des qualificateurs d'activité	Diminution de l'opportunisme
Analyse des déclencheurs de découverte	Aucun changement important
Distribution des impacts	Aucun changement important
Distribution des types de défaut	Augmentation du type *Assignment/Init.* Statu quo pour les autres types
Distribution de l'âge des défauts	Statu quo
Distribution des activités par types de défaut	Faibles améliorations
Distribution des temps de découverte	Distributions semblables
Distribution des temps de découverte avec qualificateur d'activité	Distribution montrant une plus grande disparité dans le temps au niveau des découvertes planifiées

6.1 Discussion sur les changements

Il est important de mentionner, dans un premier temps, comment les changements décrits au chapitre précédent ont évolué. Effectivement, il n'est généralement pas évident de changer les habitudes de travail des gens. Dans ce cas-ci, comme il s'agissait d'ajouter de nouvelles pratiques, il est normal qu'il y ait un temps de stabilisation pour bien comprendre les méthodes proposées.

Plus particulièrement, les méthodes implantées ont subi des modifications par les développeurs impliqués. Ceux-ci n'étant pas habitués à cette charge de travail supplémentaire, ils ont naturellement diminué la quantité de documentation à produire. Ainsi, alors que les premières tâches de test unitaire ont été accomplies tel que prévu par la méthode du chapitre précédent, il y a eu une tendance à diminuer l'effort tout au long de cette deuxième période d'échantillonnage.

Une première modification que nous avons observée est que l'analyse de risque n'était plus documentée. Tandis que celle-ci était encore effectuée par les développeurs, elle est toutefois devenue implicite dans le processus. Ensuite, l'analyse du graphe de fonction a été retirée pour éviter la manipulation de ces graphes. Finalement, la documentation des cas de test développés a été déplacée à l'intérieur des fichiers de code source des tests unitaires. Les descriptions des cas de test se retrouvent donc en commentaire dans le code. Il est à noter toutefois que les techniques de couverture ont été conservées pour la création des tests.

Le retrait de plusieurs éléments de documentation n'est pas surprenant en tant que tel. Effectivement, l'équipe a un historique de non documentation des informations en général. Il est normal que la réaction des membres ait été de réduire la charge de documentation proposée. Lorsqu'interrogés sur la raison des retraits, les membres de l'équipe répondaient généralement qu'ils ne voulaient pas perdre de temps sur la documentation. D'ailleurs, ceux-ci ne rejetaient pas l'utilité de l'information dans la documentation proposée, ils favorisaient plutôt une approche intégratrice où la même information pouvait se retrouver ailleurs (généralement dans le code).

Il est difficile de mesurer l'impact qu'a eu cette réduction des procédures de test unitaire du processus. Cette activité n'ayant pas été abandonnée comme tel, l'expérimentation a suivi son cours. La présentation des résultats qui suit prouve en quelque sorte que la présence des tests unitaires a eu un certain effet, indépendamment

de l'implantation de leur pratique. L'information importante à noter dans ce cas-ci est qu'implanter de nouvelles méthodes dans une équipe peut résulter possiblement en une divergence entre les pratiques prescrites et les pratiques effectuées.

6.2 Survol du deuxième échantillon

Il est important de regarder les différences entre les deux échantillons, car ces différences peuvent possiblement influencer la validité des résultats. Dans cette section, l'échantillon sera analysé sous plusieurs angles dont le temps d'échantillonnage, la représentativité de l'échantillon, la quantité de défauts et le retrait des défauts inutilisables, en comparaison au premier échantillon.

La période d'échantillonnage pour cette deuxième partie s'est étendue du 25 février au 19 août 2004. Ces deux dates correspondent à celles de la première partie qui s'est déroulée un an plus tôt. Cette coïncidence n'est pas accidentelle, car un effort assez important ayant été déployé entre les deux périodes d'échantillonnage pour analyser les données, pour faire une recherche sur les différentes méthodes de test unitaire, pour mettre en place le processus de test, ainsi que pour chercher des outils pour seconder les développeurs dans leur effort de test, nous avons décidé de commencer la deuxième période à la même date que la première. Ce délai important entre les deux périodes d'échantillonnage fut finalement un avantage car ces deux périodes couvrent la même période de l'année. Ainsi, les mêmes événements annuels, tels que la préparation des démos pour l'E3 et l'approche des échéances des projets clients qui se situent au début de l'automne, ont été couverts.

Au cours de cette période, 239 défauts ont été recensés. De ce nombre, seulement 149 défauts ont pu être analysés. Plusieurs raisons expliquent cet échantillon plus faible. Premièrement, plusieurs fonctionnalités complexes ont été menées de front

durant cette période et certaines d'entre elles n'ont pas été terminées durant la période d'échantillonnage. Ensuite, durant cette même période, il y a eu deux stagiaires à des moments différents dans l'équipe. Le premier stagiaire, présent du mois de janvier au mois de mai, n'a pas travaillé sur de nouvelles fonctionnalités critiques. Le deuxième stagiaire, par contre, s'est fait assigner au développement d'une nouvelle fonctionnalité de priorité moyenne. Malheureusement, ce développement n'a pu être conservé à cause de divers problèmes. Ceci a pour conséquence le retrait d'un certain nombre de défauts de ce deuxième échantillon, que nous avons choisi de ne pas analyser. La raison pour laquelle nous avons décidé de les retirer est que le travail du stagiaire n'était pas représentatif du processus de développement de l'équipe, créant un nombre anormal de défauts. Ce stagiaire a également entré plusieurs fiches qui ne décrivaient pas des défauts, mais plutôt des critiques de design. Ces dernières ont également été retirées de l'analyse. Par contre, tous les autres défauts n'appartenant pas à ces deux groupes, mais trouvés par les stagiaires ont été conservés. Effectivement, ceux-ci ont participé à la mise en place et à l'exécution des tests unitaires, leur apport en défauts est par conséquent important.

D'autre part, certains défauts ont dû être retirés de l'échantillon, en plus de ceux du stagiaire, à cause de l'impossibilité de retrouver l'historique ou à cause de leur bas niveau de sévérité, tout comme dans la première analyse. Tous ces facteurs font en sorte que le nombre de défauts dans cet échantillon est plus bas que celui de la première partie de l'expérience.

6.3 Analyse

Dans cette section, l'analyse en détail du deuxième échantillon est effectuée sous forme de graphiques. Comme dans le chapitre III, les analyses en fonction des

paramètres ne faisant pas partie d'ODC sont d'abord présentées. Suivent ensuite les analyses d'ODC et les différentes analyses du temps avant la découverte.

La première analyse effectuée concerne la distribution des défauts dans les différents produits de l'équipe D.A.R.E. L'analyse de la figure 6.1 démontre encore une fois que la plupart des défauts sont trouvés dans le moteur et dans le produit *SoundEditor*. Il y a plusieurs choses à noter en plus de cette observation toutefois. Premièrement, le graphe semble indiquer que plus de produits ont été impliqués dans cette analyse. Toutefois, un certain nombre de ces nouveaux produits sont en fait des modules d'extensions aux autres produits et sont donc assujettis aux mêmes critères de qualité. Également, tout comme dans la première analyse, certains produits ont été retirés de l'échantillon pour cause de critères de qualité plus faibles. Ainsi, les seuls nouveaux produits requérant un niveau de qualité supérieur sont *Dare_Plus*, *E_E* et *OBC* (noms incomplets pour confidentialité). Le retrait des autres produits de l'analyse réduit encore une fois le nombre de défauts de l'échantillon, ramenant celui-ci à 149 défauts analysables.

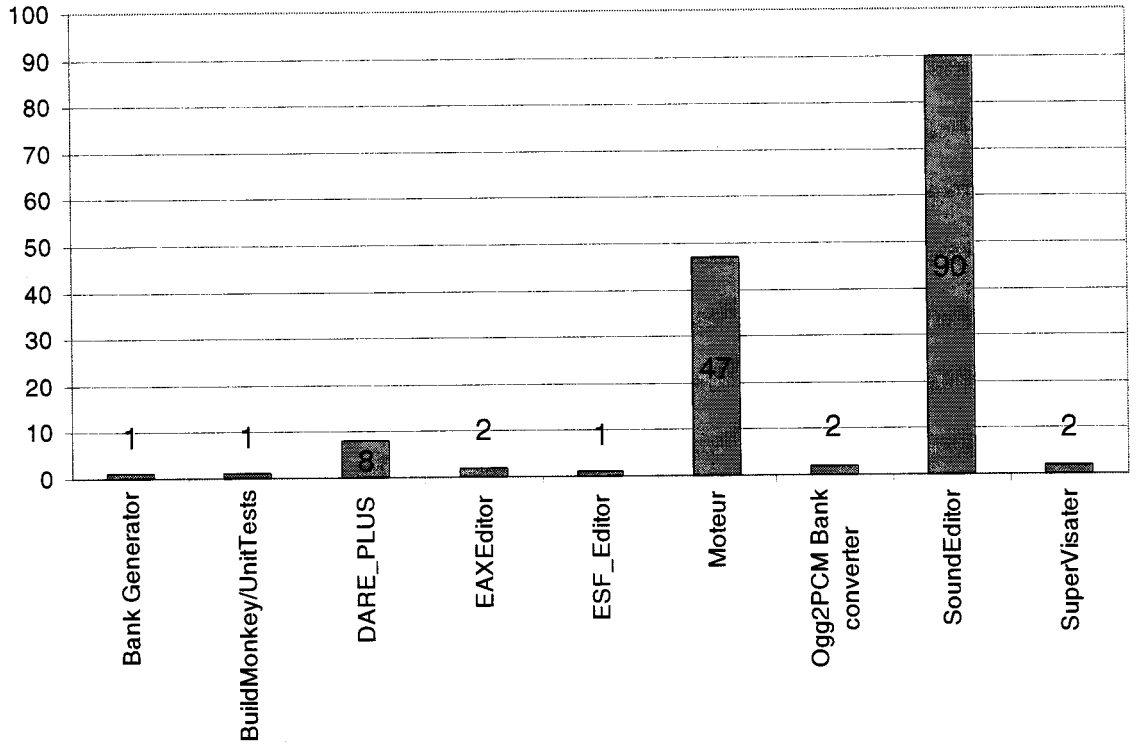

Figure 6.1: Distribution des défauts dans les différents produits pour le deuxième échantillon complet

Une deuxième remarque est qu'il y a nettement plus de défauts dans le logiciel *SoundEditor* que dans le moteur, contrairement à la première analyse. Tandis que ceci n'affecte pas les résultats de notre analyse, il est intéressant de noter ce fait.

La prochaine analyse, celle de la figure 6.2, concerne la distribution de la sévérité des défauts dans les produits restants. Comme dans la première partie, le nombre de défauts plus sévères (A et B) est encore relativement élevé, avec des pourcentages de 80% et 61% pour le moteur et *SoundEditor* respectivement. Il y a donc eu peu de changement à ce niveau entre la première partie et la deuxième partie.

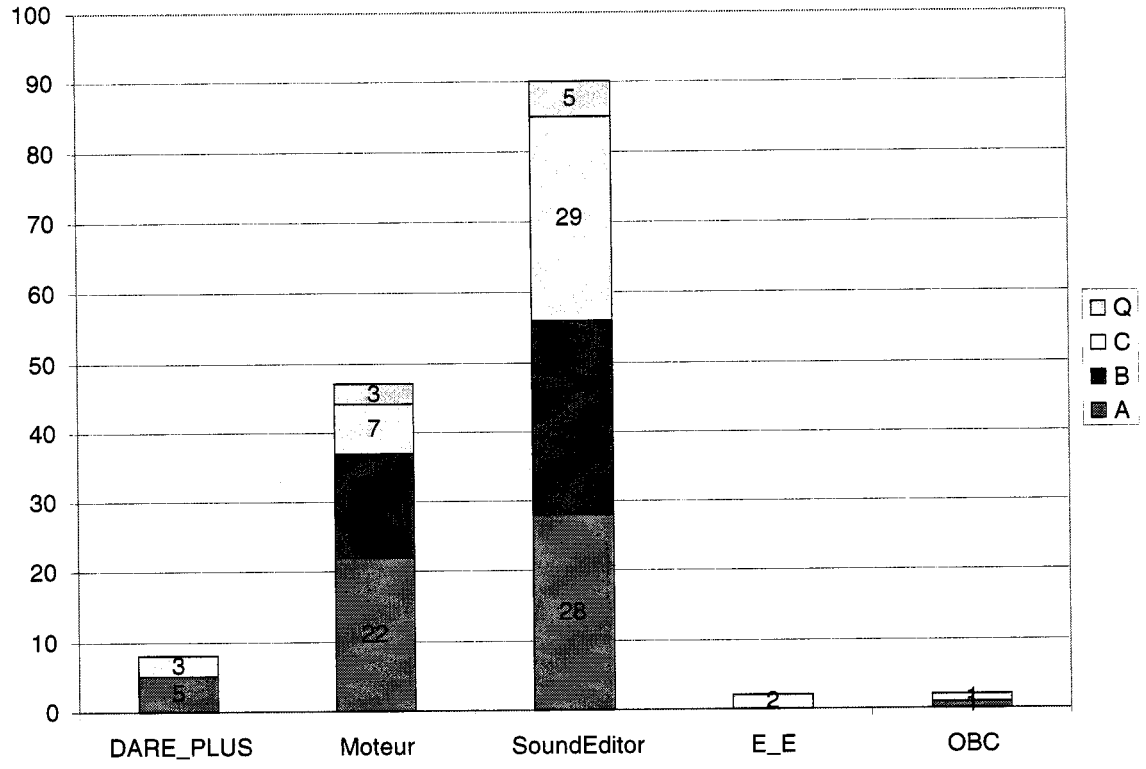

Figure 6.2: Distribution de la sévérité des défauts dans les différents produits pour le deuxième échantillon réduit

L'échantillon qui est conservé pour toute l'analyse a donc 149 défauts. De ces 149 défauts, 111 sont corrigés et peuvent être analysés pour la fermeture. On peut dire que la population du deuxième échantillon est environ la moitié de la première. Comme mentionné précédemment, ceci n'est pas surprenant, étant donné qu'il y a eu une diminution du nombre de fonctionnalités publiées dans la deuxième période d'échantillonnage.

La première analyse d'ODC présentée est la distribution des activités de découverte. Le graphique de la figure 6.3 montre une comparaison entre les deux distributions, l'échantillon marqué A est le premier échantillon et B est le deuxième

(cette nomenclature sera utilisée pour le restant du chapitre). Un lien très intéressant se dégage de ce graphe.

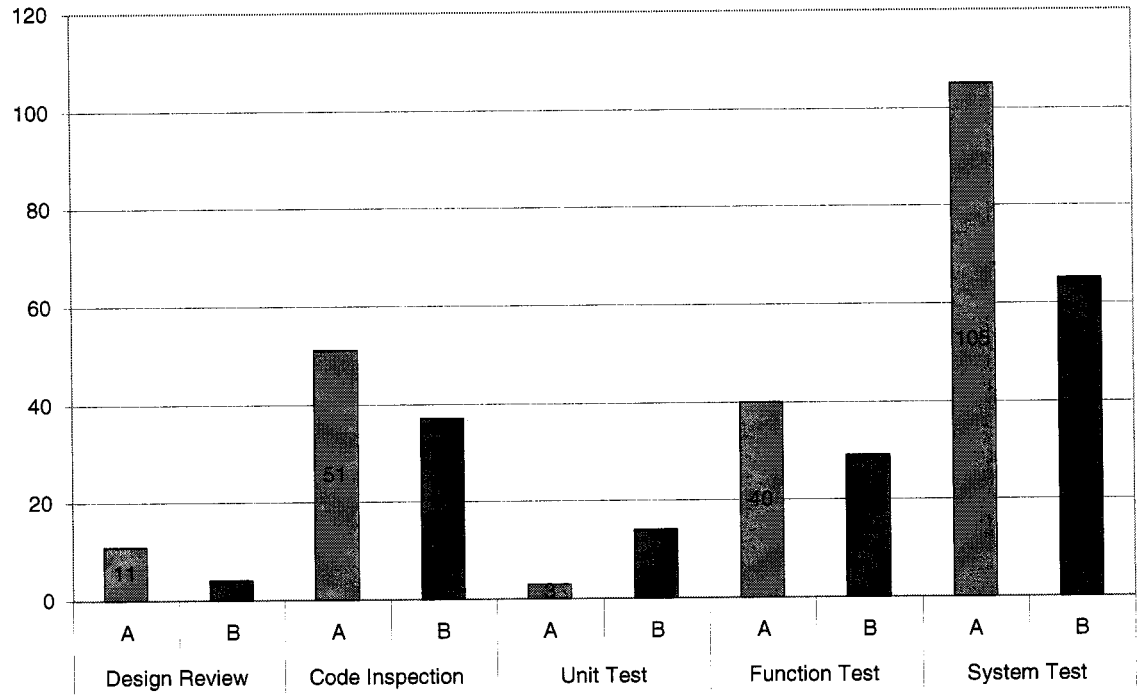

Figure 6.3: Comparaison des deux distributions pour les activités de découverte

Effectivement, la distribution des activités de découverte pour la deuxième partie de l'étude est très semblable à celle de la première partie. Toutefois, une analyse objective par le test du chi carré révèle que ces deux distributions sont différentes. Une analyse plus profonde montre par contre que l'augmentation des tests unitaires est responsable à plus de 80% de ce changement. Ainsi, à part une faible augmentation en absolu du nombre de défauts trouvés par tests unitaires, les deux distributions sont essentiellement les mêmes. Seul le nombre absolu de défauts a diminué dans la plupart des catégories. Le test système est toujours l'activité principale de découverte. La même tendance se répète pour les deux autres activités dominantes. Au niveau des inspections de code, 24% des découvertes étaient attribuables à cette activité dans le premier échantillon pour 24% dans le deuxième. Pour l'activité de test de fonction, ces nombres

respectifs sont 19% et 19%. Les revues de design ont aussi très peu bougé (5% à 3%), mais leur petit nombre rend toute interprétation difficile. Finalement, la seule activité qui a augmenté en proportion est le test unitaire, passant de 1% à 9%. C'est la seule activité qui a augmenté en pourcentage et en nombre absolu.

On remarque toutefois que le test unitaire reste une activité marginale de découverte et il est difficile d'affirmer qu'il y a amélioration par rapport au premier échantillon. De plus, un regard plus en profondeur sur les défauts trouvés par ce type de test montre qu'une bonne partie de ces défauts ont été trouvés par les stagiaires. Également, tous les autres défauts trouvés par cette méthode l'ont été par le chef de l'équipe, avec qui nous avons mis en place les techniques et les méthodes. Ceci nous dit donc que très peu de choses ont changé au niveau des activités de découvertes. Toutefois, les analyses qui suivent nous montrent une évolution intéressante.

La figure 6.4 concerne les distributions des activités de découverte en fonction du découvreur pour les deux échantillons.

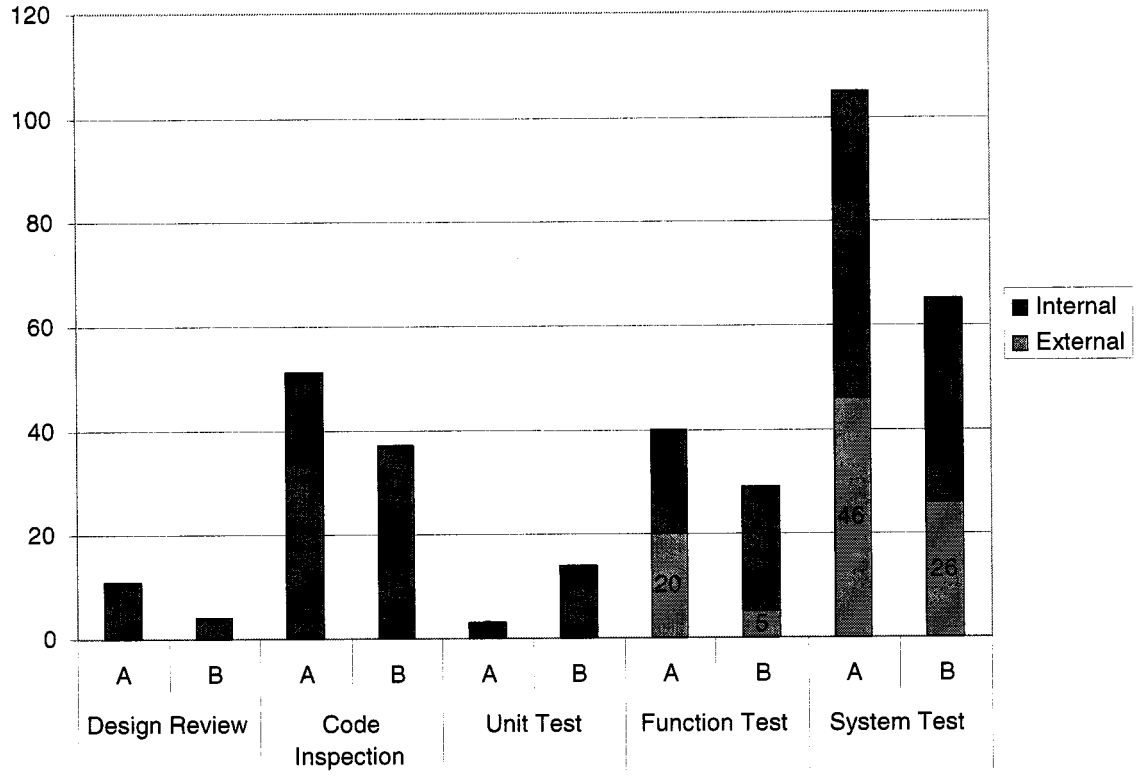

Figure 6.4: Distribution des activités de découverte avec l'information du découvreur pour les deux échantillons

Dans les deux échantillons, les activités de découverte liées aux développeurs (révision de design, inspections de code, test unitaire) sont toutes effectuées à l'interne, ce qui est normal. D'autre part, la proportion des erreurs trouvées par des clients a effectivement diminué en pourcentage, passant de 31% à 21% pour les deux activités concernées. Ceci constitue, en fait, en une amélioration intéressante par rapport au premier échantillon, surtout que l'activité principalement affectée par ce changement est le test de fonction. Il n'est toutefois pas immédiatement clair pourquoi ce changement a

eu lieu, après tout, les activités de découverte semblent avoir très peu changé. Les analyses qui suivent en disent un peu plus long sur l'explication de ce changement.

Le graphique de la figure 6.5 montre les distributions des activités en fonction des qualificateurs d'activité pour les deux échantillons.

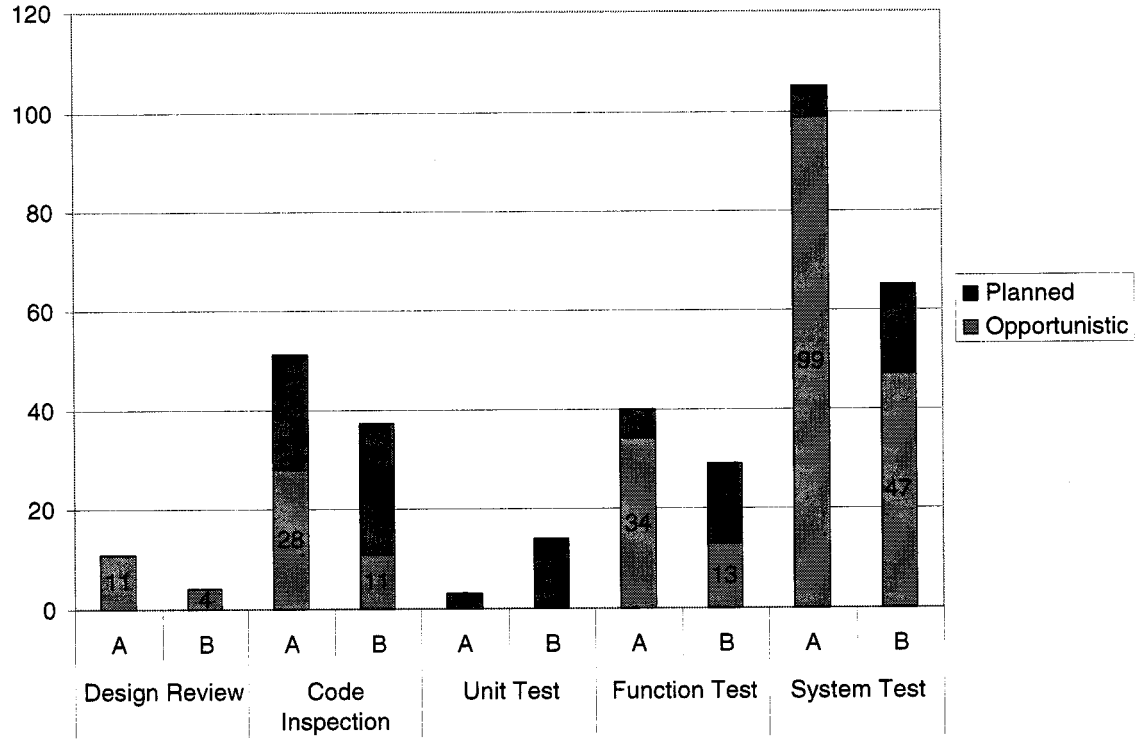

Figure 6.5: Distribution des activités avec l'information de qualificateur d'activité pour les deux échantillons

L'information principale à comprendre de ce graphique est qu'il y a eu amélioration du taux d'opportunisme des activités de découverte. Effectivement, 82% de défauts ont été trouvés de façon opportuniste dans la première partie de l'étude, par rapport à 50% pour la deuxième partie. En fait, la véritable amélioration provient de l'augmentation du taux d'activités planifiées pour les inspections de code et les tests de fonction. Il semble y avoir eu amélioration du même taux pour les tests systèmes, certes,

mais ceci est tempéré par une grande proportion du déclencheur *Blocked Test*, qui implique un certain opportunisme. Finalement, l'amélioration marquée ici n'est pas facilement explicable étant donné les changements que nous avons apportés. Les analyses qui suivent confirment quelque peu qu'il y a eu changement, mais ne l'expliquent pas en tant que tel. Seules les analyses supplémentaires expliqueront les changements observés.

Le graphique 6.6 présente la distribution des qualificateurs d'activité avec l'information de découvreur seulement.

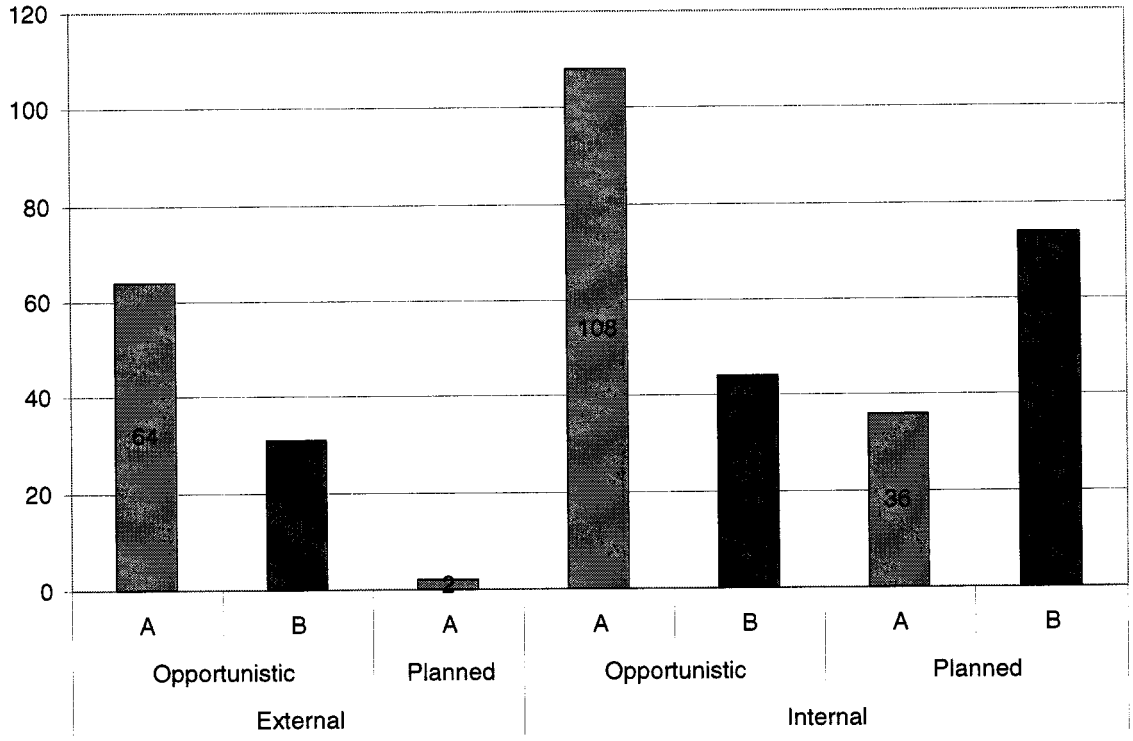

Figure 6.6: Distribution des qualificateurs d'activité en fonction des découvreurs pour les deux échantillons

Encore une fois, il apparaît qu'il y a eu diminution du taux d'opportunisme pour les défauts trouvés à l'interne. Les défauts trouvés à l'interne de façon planifiée passent

de 17% à 50%, tandis que les défauts trouvés à l'interne de façon opportuniste passent de 51% à 30%. Ceci confirme évidemment l'information trouvée à la figure 6.5 en montrant l'amélioration notée concerne les découverte internes. Cette analyse confirme le changement, mais ne n'expliquent pas.

Les prochaines analyses concernent les distributions des déclencheurs pour chaque type d'activité. Pour simplifier la présentation, seules les informations de la deuxième partie sont présentées, car les aspects les plus importants de cette deuxième analyse ne concernent par les déclencheurs. Les graphiques trop simples seront également retirés en faveur de la description des résultats.

La distribution des déclencheurs pour les revues de design est très similaire à celle de la première partie. Tous les défauts ont comme déclencheur *Design Conformance*, comme dans la première partie, ce qui indique le statu quo à ce niveau. De toute façon, le nombre de défauts trouvés par revue de design est trop faible encore une fois pour en tirer quoique ce soit.

La figure 6.7 montre la distribution des déclencheurs au niveau des inspections de code.

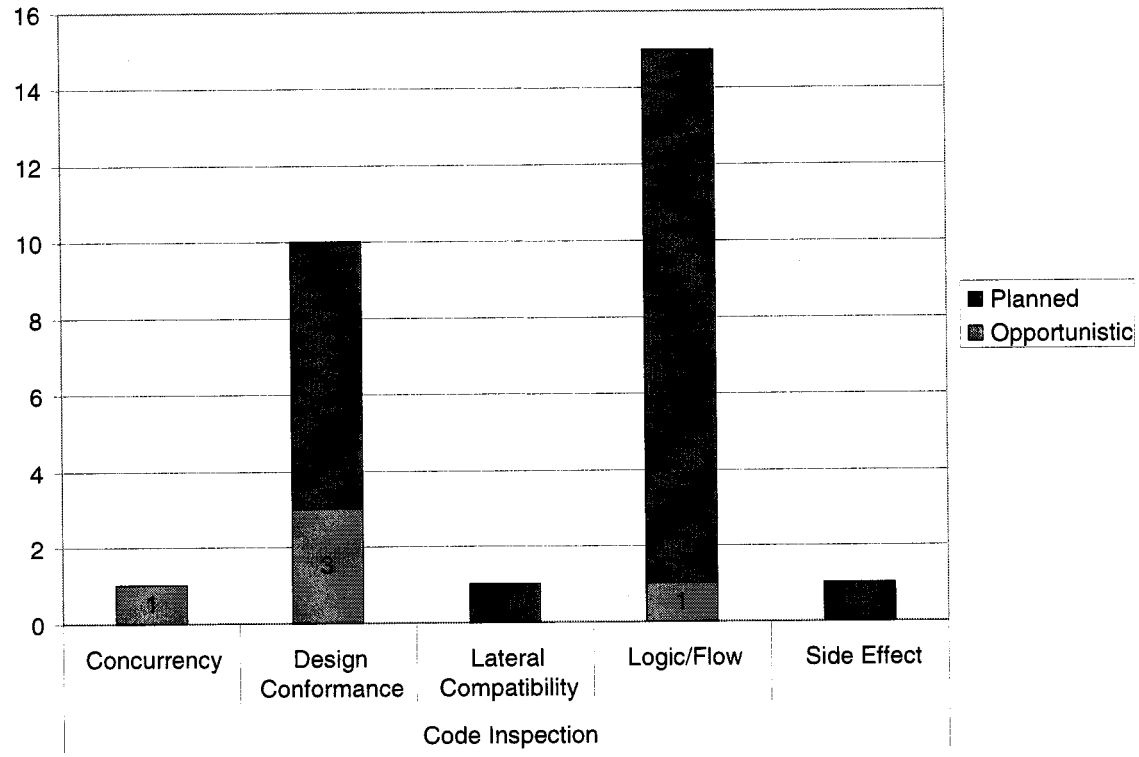

Figure 6.7: Distribution des déclencheurs pour les inspections de code avec l'information de qualification d'activité pour le deuxième échantillon

La distribution des déclencheurs pour ce type d'activité n'a pas vraiment changé non plus. Les déclencheurs les plus importants restent le *Design Conformance* et *Logic/Flow*, les autres étant des exceptions authentiques, identifiées en analysant les défauts individuels. Ce qui est toujours remarquable, par contre, c'est la diminution du taux d'opportunisme.

Au niveau des tests unitaires, il n'y a pas d'information intéressante à retirer des déclencheurs, surtout pour un échantillon aussi faible. Le seul élément à noter, comme

mentionné précédemment est qu'il y ait eu une faible augmentation des découvertes par ce type de test.

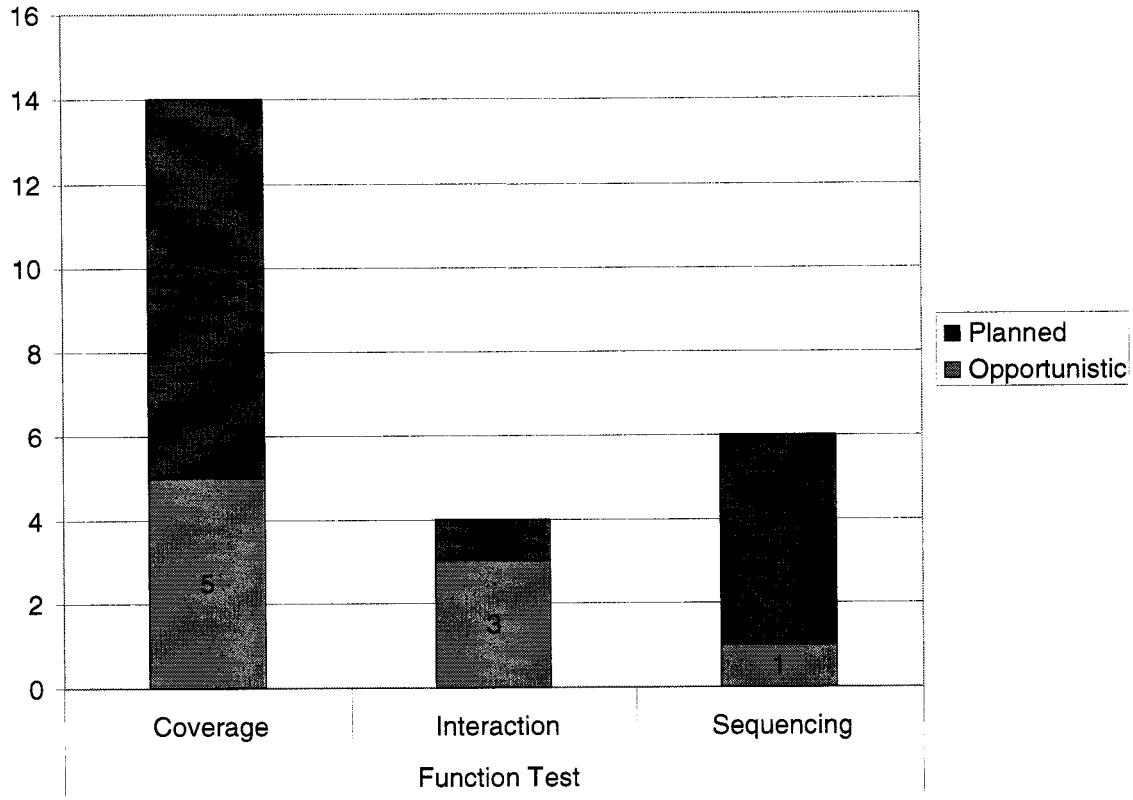

Figure 6.8: Distribution des déclencheurs pour l'activité de test de fonction avec l'information du qualificateur d'activité pour le deuxième échantillon

La figure 6.8 montre la distribution des déclencheurs pour les tests de fonction. Comme le montre le graphique, la situation a très peu évolué par rapport à la première étape. Effectivement, le déclencheur principal reste *Coverage*, et les autres sont toujours plus faibles en nombre. Même si ces derniers ont changé un peu de distribution et que le type *Variation* n'est pas présent, nous croyons qu'il s'agit plus d'une preuve que les techniques de test de ce type sont toujours aussi imprécises.

La dernière distribution des déclencheurs concerne ceux des tests système. La figure 6.9 présente le graphique de cette distribution. Comme dans la première partie, le déclencheur *Blocked Test* est omniprésent, à part pour quelques incidents isolés cette fois. Le seul changement remarquable est la diminution relative du nombre d'occurrences de découvertes opportunistes.

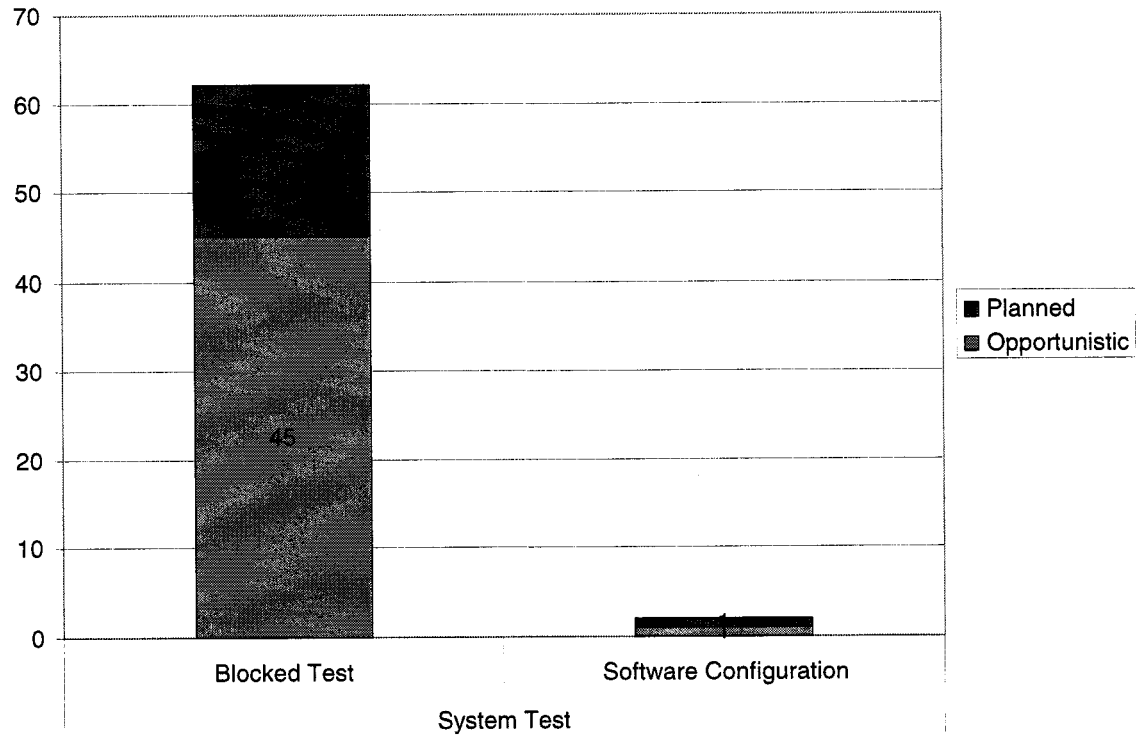

Figure 6.9: Distribution des déclencheurs pour l'activité de test système avec l'information du qualificateur d'activité pour le deuxième échantillon

La toujours forte présence des *Blocked Test* semble indiquer que très peu de choses ont changé au niveau de la découverte des défauts. Effectivement, même si l'opportunisme semble avoir diminué globalement, on trouve toujours les défauts d'une façon aléatoire.

Les premières analyses introduisent donc un questionnement. Effectivement, on retrouve une diminution de l'opportunisme et des découvertes externes, mais les activités de découverte ne semblent pas avoir changé de distribution, hormis une petite augmentation de défauts trouvés par test unitaire. La diminution globale de l'opportunisme des découvertes semble être plus présente au niveau des inspections de code et des tests de fonction. Les analyses faites jusqu'à maintenant supportent donc un certain changement, mais ne l'expliquent pas. De plus, les analyses qui suivent confirment encore un certain statu quo au niveau de l'échantillon des défauts.

La prochaine analyse concerne la distribution des impacts des défauts, exposée dans la figure 6.10.

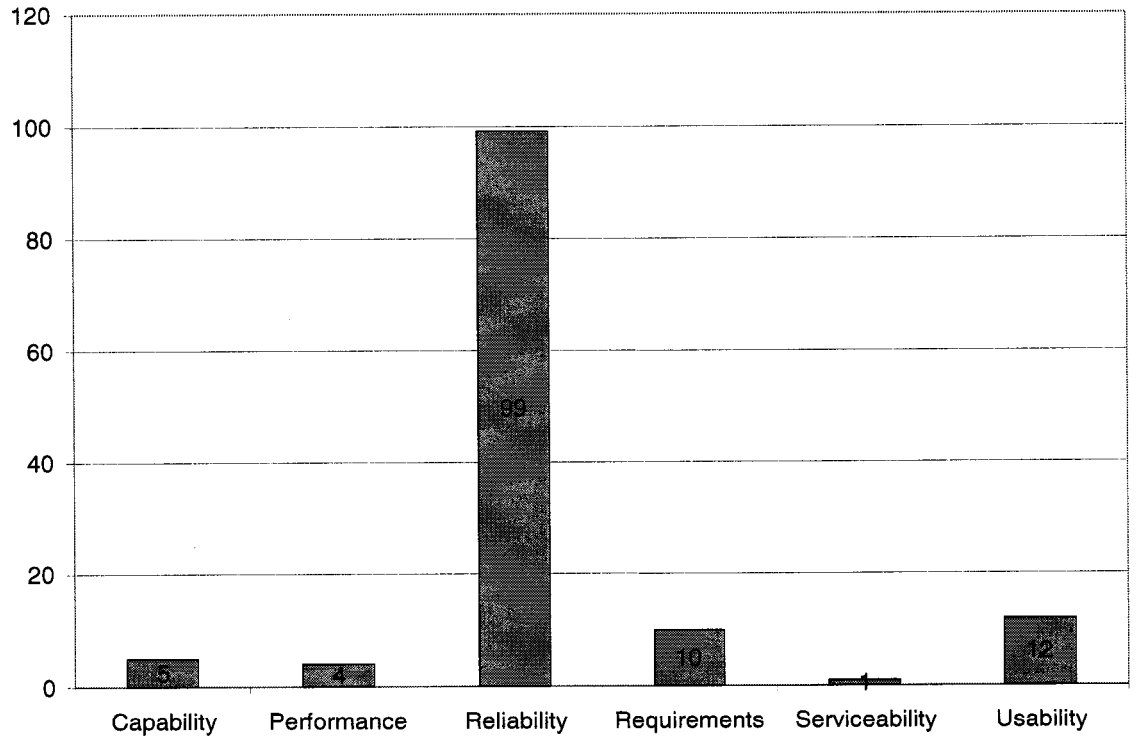

Figure 6.10: Distribution des impacts pour le deuxième échantillon

Une comparaison avec la première partie montre qu'il n'y a pas eu de changement à ce niveau non plus. Effectivement, l'impact dominant reste la fiabilité des logiciels. Même si la première distribution des impacts n'a pas eu d'influence sur les améliorations qui ont été apportées au processus de test, il est tout de même pertinent de regarder si changement il y a eu dans la deuxième partie. Comme très peu de choses ont changé, il n'est pas nécessaire de s'attarder sur ce paramètre.

Les prochaines analyses concernent la section de fermeture pour la classification des défauts. Elles nous montrent que très peu de choses ont changé. Le premier graphique à la figure 6.11 présente la distribution des types de défauts.

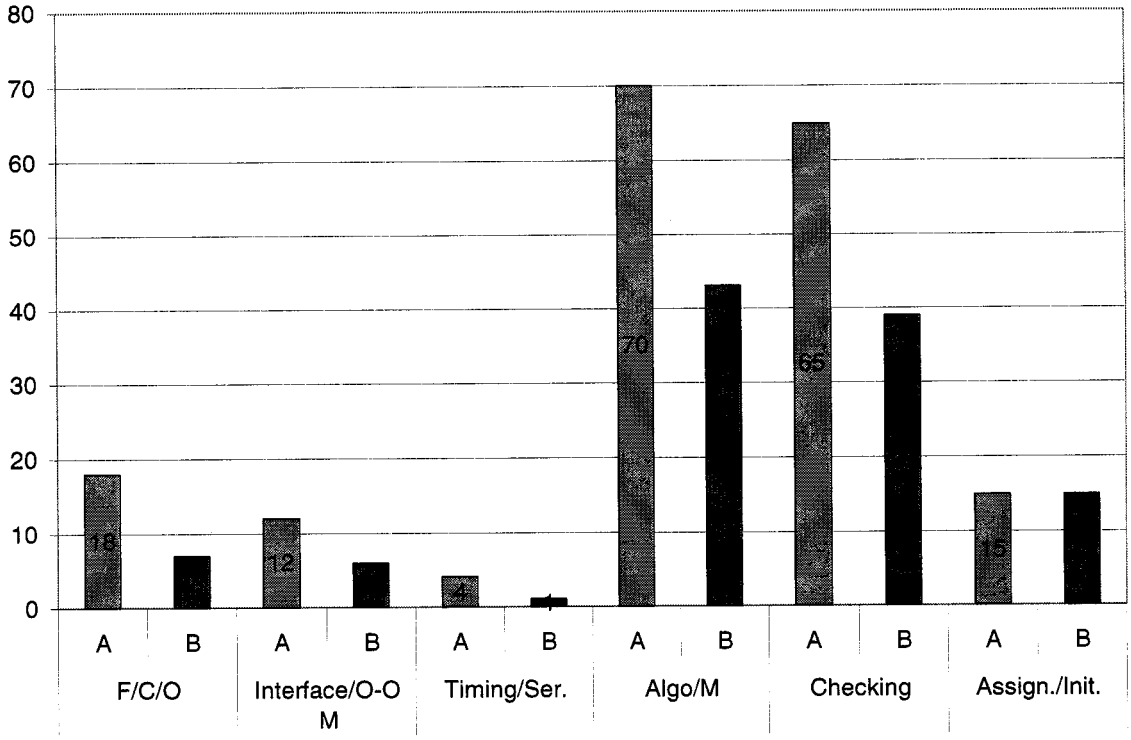

Figure 6.11: Distribution des types de défauts pour les deux échantillons

D'après cette figure, il n'y a pas vraiment eu de changement au niveau de cette distribution. Les deux types les plus dominants sont toujours *Algorithm/Method* et

Checking. Un test du chi carré révèle qu'il y a eu peu de changement dans la distribution ($p = 0.58$), mais que ce changement est lié principalement à l'augmentation relative des défauts de type *Assignment/Initialization* (est responsable à 50% du changement). Ce type est par contre introduit au même moment que les deux types dominants et est typiquement trouvé par le même type d'activités de ces derniers selon ODC. Cette variation n'affecte donc pas nos résultats. L'importance des trois autres types n'a également pas changé.

L'information dégagée de cette analyse est intéressante, car elle indique qu'il y a stabilité relative dans les types d'erreurs commises. La distribution des types de défauts en fonction des activités de découverte devrait confirmer le statu quo également. Cette analyse sera faite un peu plus loin.

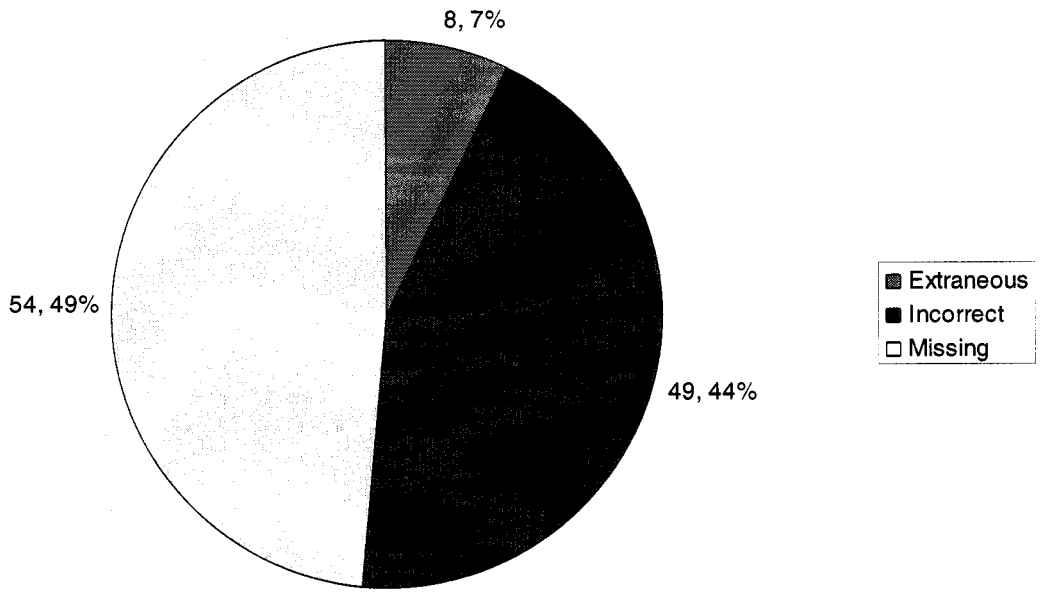

Figure 6.12: Distribution des qualificateurs de défauts pour le deuxième échantillon

Ensuite, la figure 6.12 montre que la distribution des qualificateurs de défauts a très peu changé aussi. Le test statistique du chi carré assure cette absence de changement ($p = 0,98$)

Le prochain graphique, celui de la figure 6.13, concerne les distributions de l'âge des défauts pour les deux échantillons. Il montre que les défauts sont trouvés majoritairement dans des fonctionnalités publiées (*Base*), tout comme il avait été remarqué dans la première partie. Également, la distribution est environ la même que celle de la première étape, avec un facteur $p = 0,70$.

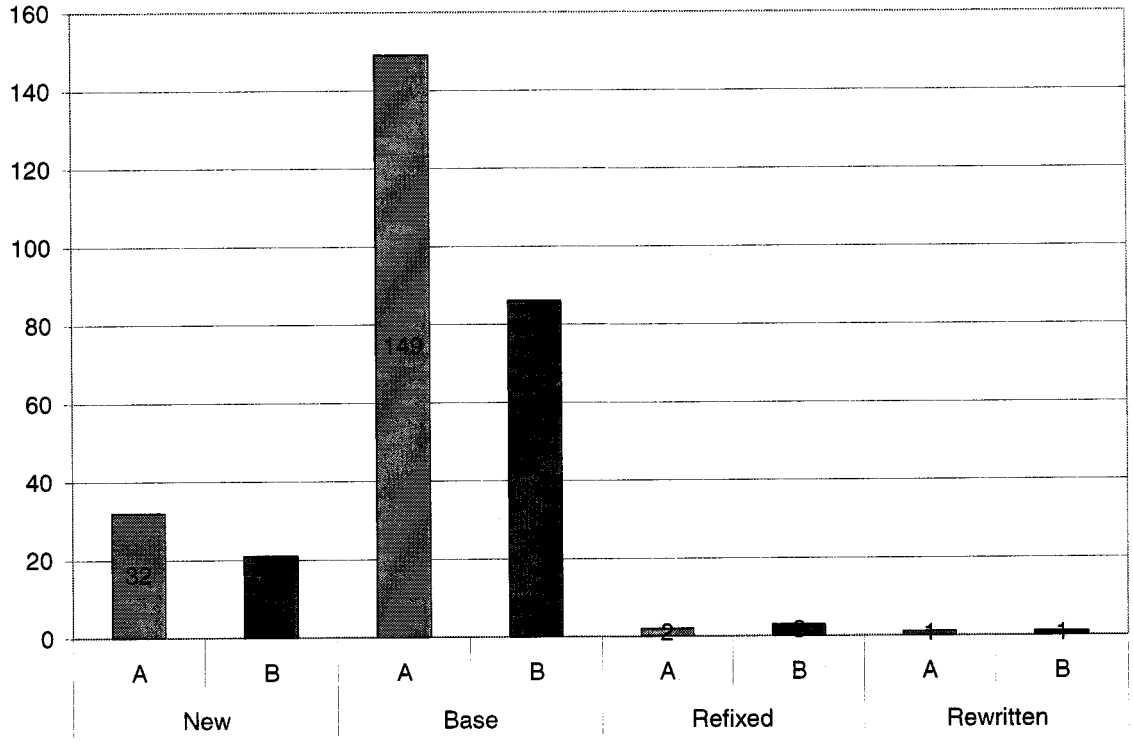

Figure 6.13: Distribution de l'âge des défauts pour les deux échantillons

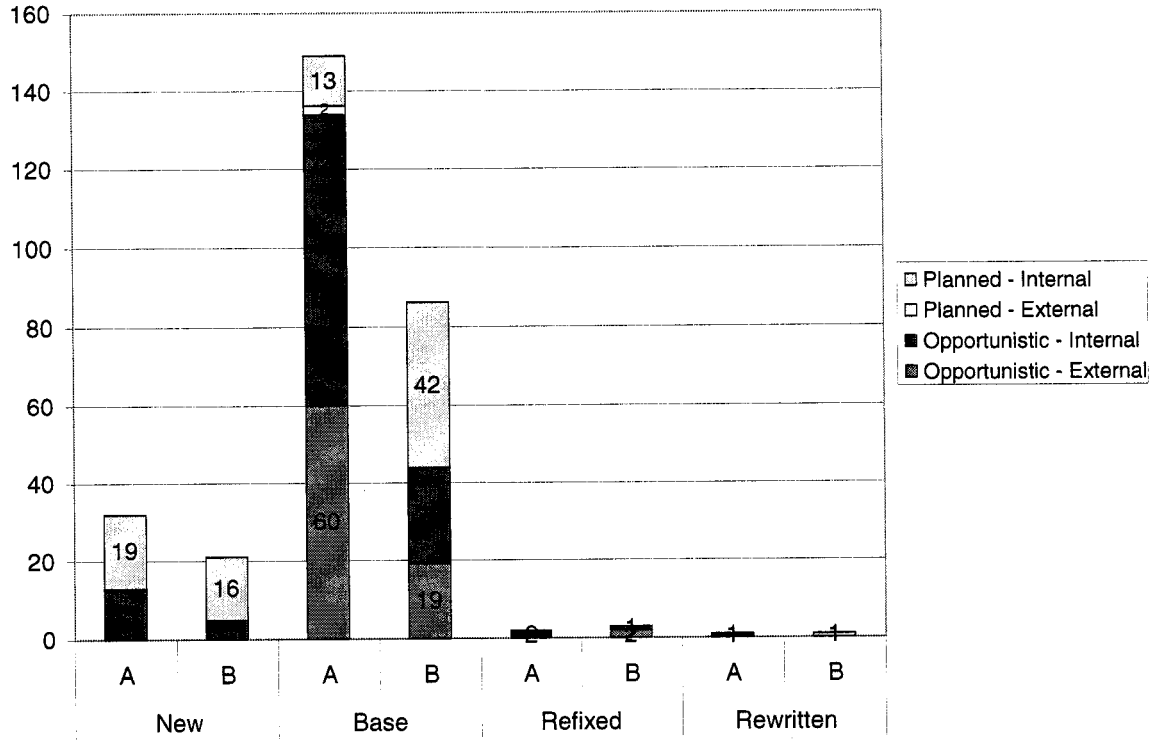

Figure 6.14: Distribution de l'âge avec les qualificateurs d'activité et le découvreur pour les deux échantillons

Comme dans la première partie, il est important de regarder la distribution de l'âge en fonction du qualificateur de défaut et du découvreur. La figure 6.14 montre, encore une fois, qu'il y a eu baisse de l'opportunisme dans l'ensemble des défauts. On remarquera en particulier la grande proportion des activités planifiées pour les défauts de type *New*, mais aussi pour le type *Base*. Ceci est relativement intéressant, car il semblerait que la baisse d'opportunisme d'affecte surtout les défauts trouvés après publication du code. L'analyse des temps de découverte devrait donner un autre indice.

La prochaine analyse concerne la distribution de la source des défauts. Comme le graphique démontrant les résultats est très simple, il ne sera pas inclus dans cette partie. L'analyse démontre que tous les défauts ont été trouvés dans du code développé dans

l'équipe : ce sont tous des *Developed In-House*. Il n'y a donc pas eu de changement à ce niveau.

La prochaine analyse concerne la distribution des types de défauts selon les activités de découverte. Les résultats de la première partie et de la deuxième partie sont présentés respectivement dans les tableaux 6.2 et 6.3.

Tableau 6.2: Distribution des types de défaut en fonction des activités de découverte pour le premier échantillon

Act. \ DT	F/C/O	Interface	Timing	Algorithm	Checking	Assign/Init
Design Rev	2	1		3		
Code Insp.	5	3	1	10	21	3
Unit T.		1		1	1	
Function T.	3		1	15	14	3
System T.	8	7	2	41		9

Tableau 6.3: Distribution des types de défaut en fonction des activités de découverte pour le deuxième échantillon

Act. \ DT	F/C/O	Interface	Timing	Algorithm	Checking	Assign/Init
Design Rev	0	0	0	0	2	0
Code Insp.	2	4	1	12	13	2
Unit T.	0	1	0	6	1	1
Function T.	2	1	0	14	6	3
System T.	3	0	0	24		9

Dans la première partie, la conclusion était que les deux types dominants étaient trouvés trop tard par rapport à leur activité de découverte de prédilection. Ce sont encore les mêmes types qui dominent la distribution et on remarque qu'ils sont toujours trouvés trop tard par rapport à leur activité de découverte. Effectivement, les défauts de type *Checking* sont principalement trouvés par des tests système, alors qu'ils devraient être

trouvés par inspection de code ou par test unitaire. Pour les défauts de type *Algorithm/Method*, on remarque également une forte présence des tests système. Les tests statistiques confirment partiellement ceci. Ils indiquent que la distribution de activités de découverte pour le type *Checking* n'a pas changé (p=0.08), quoique ce test indique une forte probabilité d'influence due au hasard. Le même test pour le type *Algorithm/Method* indique qu'il y a eu changement en faveur des autres activités de découverte. Globalement, on peut dire qu'il y a une tendance à l'amélioration, mais que celle-ci est encore trop faible pour crier victoire.

Les prochaines analyses, concernant le temps avant la découverte vont terminer l'analyse de cet échantillon selon les paramètres de la première partie. Un point important à noter est que les étendues de valeurs de l'axe du temps des prochains graphiques sont plus grandes que pour la première analyse, car la période pendant laquelle il est possible de connaître l'historique est nécessairement plus grande. Il est donc important de tenir compte de ceci dans l'analyse. La première analyse est présentée dans la figure 6.16 et concerne donc l'ensemble des données du deuxième échantillon, tandis que la figure 6.15 représente la distribution pour le premier échantillon pour faciliter la comparaison visuelle.

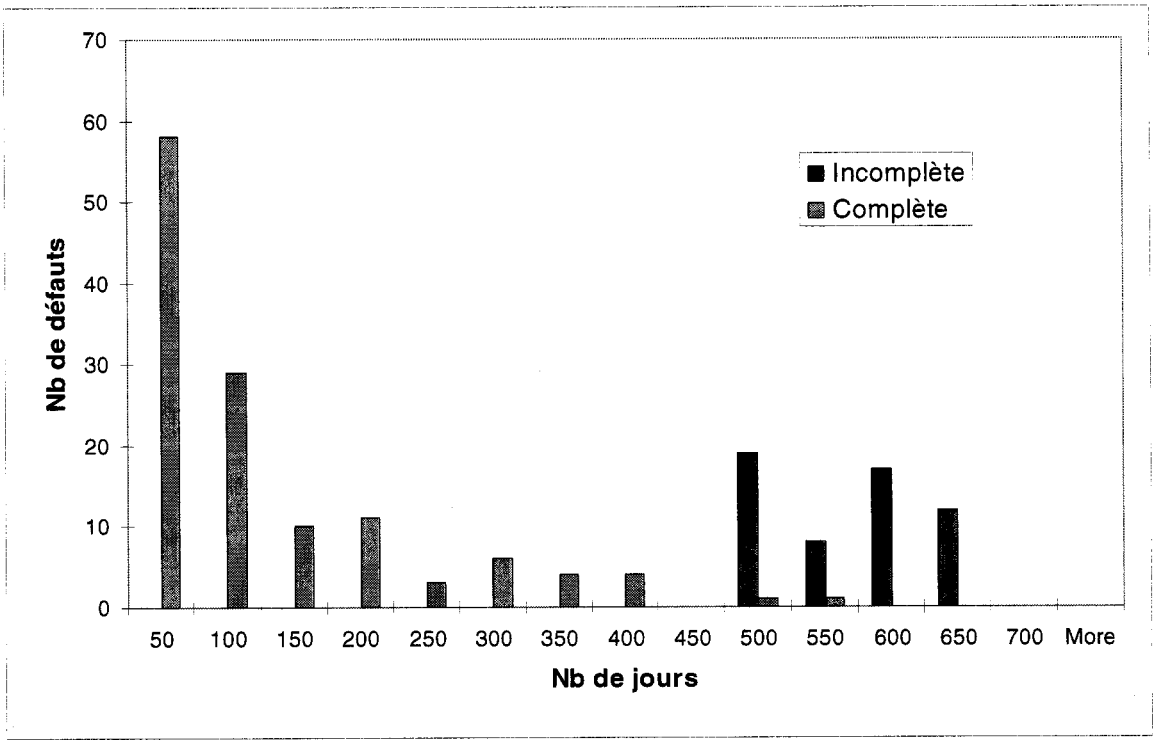

Figure 6.15: Distribution du temps avant découverte pour le premier échantillon

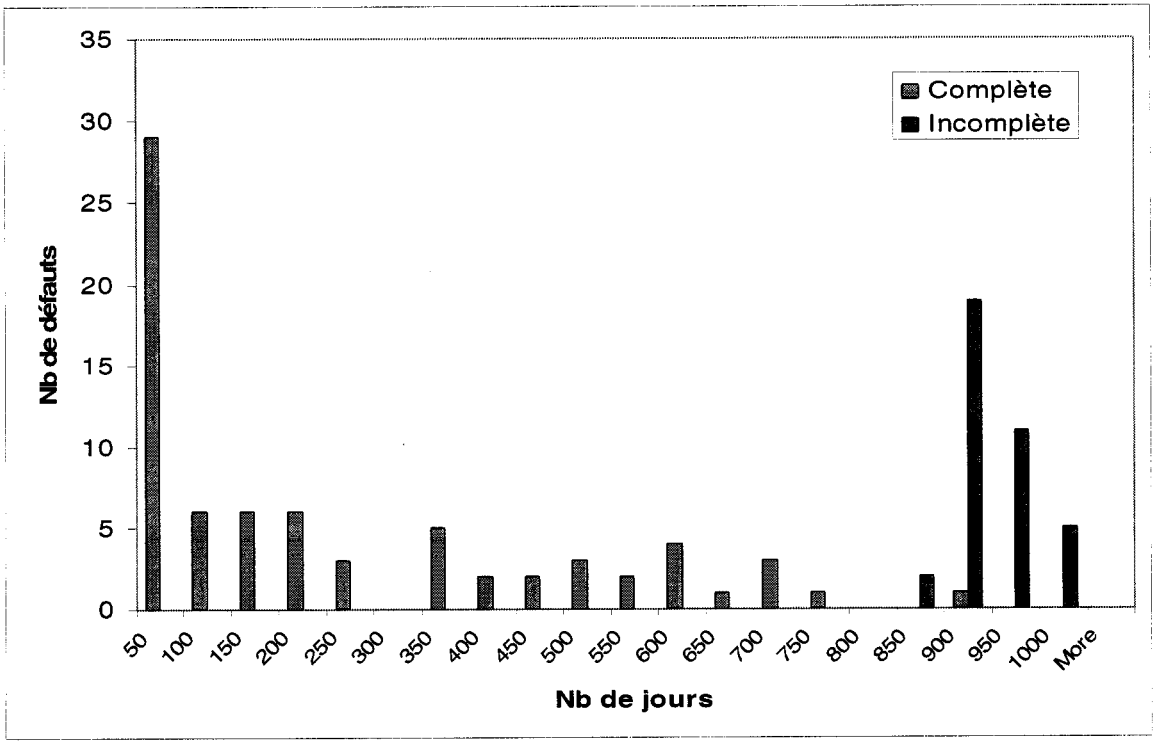

Figure 6.16: Distribution du temps avant découverte pour le deuxième échantillon

Comme dans la première partie, on voit qu'une grande proportion des défauts est trouvée à l'intérieur de 50 jours, mais que la variation pour le temps de découverte est toujours aussi grande. Par le fait même, de très anciens défauts sont découverts régulièrement et ceux-ci sont toujours importants en nombre.

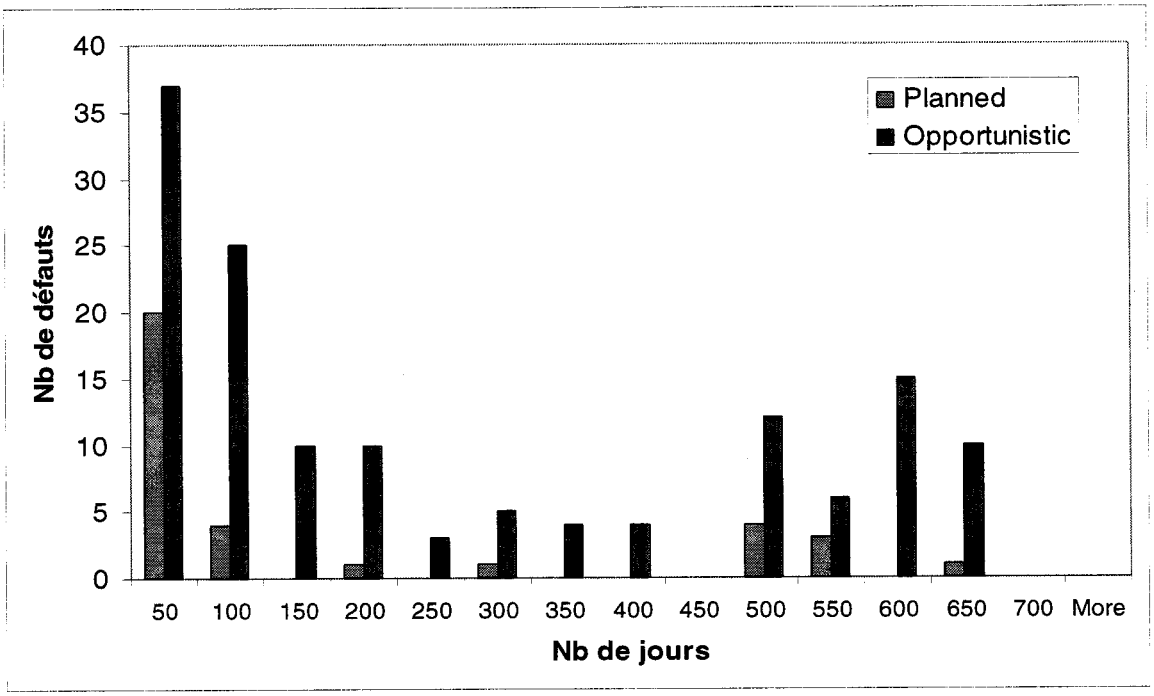

Figure 6.17: Distribution du temps avant découverte avec qualificateur d'activité pour le premier échantillon

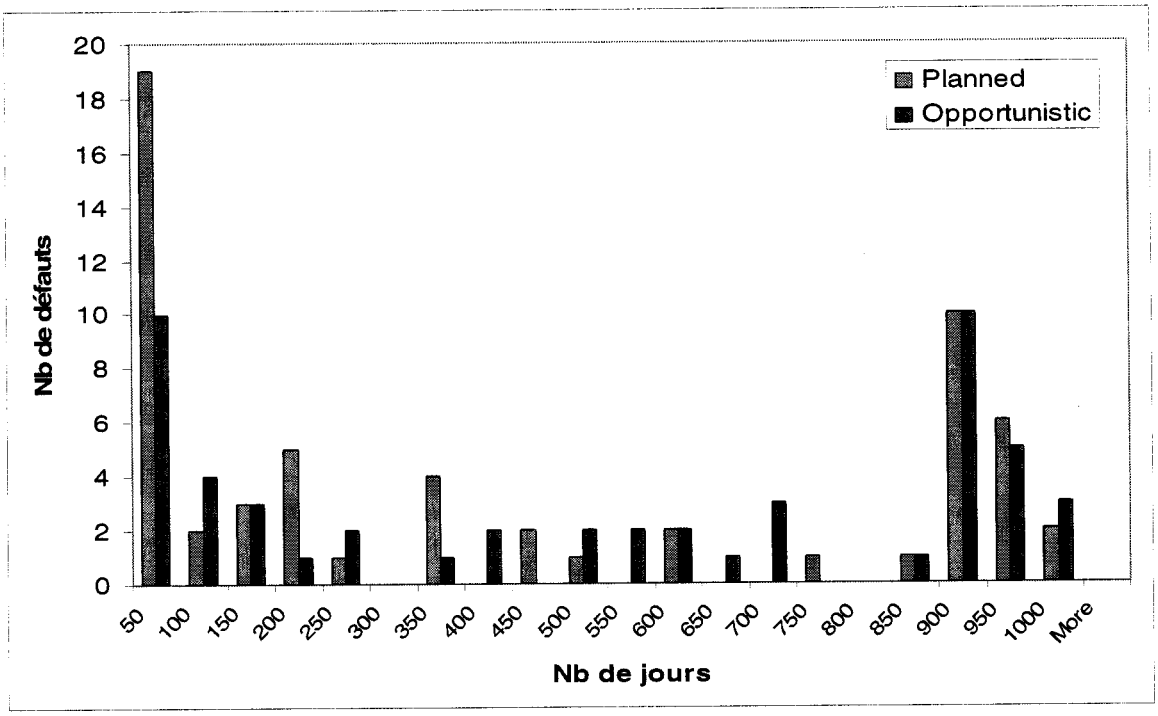

Figure 6.18: Distribution du temps avant découverte avec qualificateur d'activité pour le deuxième échantillon

La figure 6.18 présente la distribution des temps avant découverte avec l'information de qualificateur d'activité pour le deuxième échantillon, tandis que la figure 6.17 présente encore les données du premier échantillon. Ce graphique semble montrer que les défauts trouvés par activités planifiées ont un âge beaucoup plus variable que dans la première partie. Également, il semblerait que plusieurs défauts anciens aient été trouvés par ce même type d'activité.

Que signifient alors ces changements? Tandis que la distribution du temps de découverte globale semble être la même, les activités planifiées semblent trouver d'anciens défauts. Ceci confirme en fait l'information sur l'âge qui est présenté précédemment dans ce chapitre où une augmentation des défauts *Base* trouvés de façon planifiée a été notée.

La réponse à ces interrogations ne se trouvant pas à l'intérieur de notre domaine de mesure, nous avons dû approfondir un peu notre recherche pour trouver une explication. Nous avons donc interrogé les différents membres de l'équipe qui ont effectué des tests unitaires. Ces réponses, bien que qualitatives, nous informent sur l'évolution de la situation.

Les interviews auprès des développeurs ont démontré que le fait d'accomplir une nouvelle tâche plus formelle, les tests unitaires dans ce cas-ci, a donné naissance à plusieurs possibilités supplémentaires pour découvrir des erreurs. Effectivement, les tests unitaires requérant une analyse en profondeur du code, plusieurs opportunités de découvertes par inspections de code ou de tests d'exécution (fonction et système) ont fait surface. Toutefois, lorsque des défauts étaient trouvés de cette manière, ils n'étaient pas classifiés comme étant découverts par test unitaire, mais bien par l'activité inspirée par le test unitaire. Ainsi, les tests unitaires ont causé des effets secondaires plus importants que l'effet primaire escompté.

De plus, ces découvertes, effectués dans un contexte de test planifié (d'où leur classification dans ce type), n'ont pas nécessairement trouvé des erreurs récentes, d'où la distribution plus repartie pour le temps de découverte et l'augmentation des *Base* trouvés par activité planifiée. Alors, pourquoi pas une augmentation des activités d'inspection ou de tests d'exécution n'appartenant pas au déclencheur *Blocked Test*? La réponse réside dans le fait que ces tests ou inspection ne sont pas plus structurés que les tests effectués selon le contexte original. Ils sont tout simplement « inspirés » d'une manière différente.

Une autre information que nous avons jugée intéressante de rechercher est la provenance des défauts ayant pu être trouvés par tests unitaires. Nous avons donc analysé les défauts étant apparus à l'intérieur de la période d'échantillonnage. Il s'avère que ces défauts avaient une distribution fort intéressante. Effectivement, la moitié de ces

jeunes défauts ont été trouvés à l'extérieur de fonctionnalités testées. De plus, la moitié des autres défauts (autrement dit, ceux trouvés dans des fonctionnalités testées) étaient dans des fonctions qui n'ont pas été identifiés pour le test. On pourrait penser qu'il y a eu mauvaise identification des fonctions à risque. Toutefois, l'analyse de ces défauts montre qu'ils se trouvaient bien souvent dans des fonctions simples ou même pas directement impliquées dans les modifications instaurées pour la fonctionnalité. Ceci nous indique que ces défauts auraient eu plus de chance d'être trouvés par des tests d'intégration ou tests de fonction. Finalement, cette information supplémentaire nous permet de poser l'hypothèse que les tests unitaires à eux seuls ne peuvent probablement pas être suffisants pour trouver tous les types de défauts : ils doivent être complémentés par d'autres pratiques de test solides.

Ainsi, l'évolution entre les deux échantillons est donc identifiée. Tandis que les tests unitaires ne sont pas source directe de découvertes de défauts, ils ont engendré des opportunités de découverte ayant un effet positif au niveau de la distribution de certaines informations trouvées par ODC.

CHAPITRE VII : CONCLUSION

Plusieurs éléments intéressant ont été apportés dans cette étude. Ces apports se situent au niveau de la recherche effectuée pour les fins de ce travail, dans l'application d'une métrique formelle à un environnement informel, ainsi que dans les résultats obtenus dans le cadre de l'expérimentation.

Dans un premier temps, ce mémoire contient une des premières expositions au processus de conception des logiciels de divertissement. Avec cette information, le lecteur peut maintenant apprécier davantage les spécificités du processus associé à la production de ce type de produit. Il est important toutefois de garder en tête que l'étude faite ici ne reflète que le processus de l'entreprise étudiée. Il serait intéressant, dans une autre recherche, de vérifier si ce processus de production et ses intervenants se retrouvent sous une forme semblable dans d'autres compagnies développant le même type de produit.

Également, les différentes informations pouvant être extraites par la classification des défauts logiciels ont été présentées dans ce mémoire. L'adaptation d'une de ces méthodes de classification, la classification orthogonale des défauts, à un environnement informel est aussi un apport intéressant à la communauté du génie logiciel. Il apparaît que la méthode ODC permet d'obtenir des informations très intéressantes sur un processus de test. Toutefois, l'adaptation de cette méthode à l'environnement étudié comportait des difficultés. Entre autres, les critères identifiant si un défaut devait être entré dans la base étaient relativement flous, à cause de la non-formalité du processus. Également, la présence du processus de maintenance réduisait l'information que l'on pouvait extraire par ODC. Tout de même, cette méthode nous a permis de faire des constatations intéressantes sur le fonctionnement des activités de test.

Troisièmement, les résultats obtenus dans cette expérimentation indiquent qu'il est possible de retirer de l'information intéressante à partir de la métrique utilisée, mais que ces informations peuvent être incomplètes selon les résultats obtenus. Effectivement, plusieurs autres informations qualitatives ont dû être obtenues afin de trouver une explication. Tandis que ceci n'est pas négatif, on comprend qu'il n'est jamais simple de mesurer un processus. Effectivement, chaque organisation étant différente, il est nécessaire qu'une adaptation doive être faite à une métrique pour que celle-ci soit efficace. De plus, cette calibration peut prendre du temps, selon le type de projet que l'on veut mesurer.

Ensuite, nous avons constaté que l'application des concepts d'ODC pour améliorer la situation n'a pas généré des résultats attendus. Effectivement, les tests unitaires suggérés par la métrique ont amélioré un aspect étranger à la définition original d'ODC, l'opportunisme des découvertes d'erreurs. Plus particulièrement, il s'agit de l'application de méthodes plus formelles qui ont permis d'améliorer certains aspects du processus de test de l'équipe étudiée. Ces méthodes formelles, même si elles ne sont pas la source première de découvertes de défauts, amène tout de même des aspects positifs qui améliorent les pratiques des développeurs. L'étude montre aussi que la prescription de pratiques dans une équipe établie n'est pas nécessairement respectée et que les gens ont tendance à simplifier les pratiques dans un contexte informel.

Comme mentionné précédemment, notre étude a ses limites. Dans un premier temps, il apparaît que l'adaptation d'une métrique de classification à caractère formel dans un environnement informel amène plusieurs problèmes. Effectivement, à cause de la non-formalité des pratiques, de nombreuses situations amenaient les développeurs à utiliser leur jugement plutôt que des critères objectifs pour classifier des défauts. Également, certaines définitions étaient assez floues pour causer des imprécisions lors de la classification. Dans son ensemble, les problèmes liés à cette méthode relèvent du fait que nous étions dans un contexte informel. Aussi, une contrainte importante de la

recherche fut que nous enregistrions l'activité de découverte, et non l'activité cédulée. Il pourrait être intéressant de noter cette information dans une étude subséquente.

Dans un autre temps, nous pouvons également questionner le choix favorisant les tests unitaires suite à la première analyse. Même si ODC indiquait que les tests unitaires et les inspections de code étaient les activités de choix pour les types de défaut dominants, nous aurions pu faire un autre choix. Les tests unitaires se sont avérés plus complexes à mettre en place et demandaient probablement trop de travail supplémentaire auquel les développeurs n'étaient pas habitués. L'implantation de tests de fonction plus formels aurait alors été un choix intéressant à cause de la grande accessibilité de ceux-ci. Effectivement, les tests de fonctions auraient été également pu être un choix intéressant, malgré la moins importante association aux types dominants et ce, pour plusieurs raisons. Premièrement, les tests de fonction auraient pu forcer les développeurs à mieux comprendre leurs requis. Également, les tests de fonction de base sont simples à effectuer et à préparer, justement à cause de leur lien avec les requis. Les tests de fonction auraient aussi forcé les développeurs à tester dans un environnement ou la plupart des fonctions sont intégrés, ce qui aurait donné une autre dimension au processus de test.

Finalement, cette recherche ouvre la porte à d'autres sujets de recherche dans le même domaine. Premièrement, comme il est décrit dans le premier chapitre de cet ouvrage, les ingénieurs informatiques travaillent beaucoup avec des artistes dans ce contexte industriel. Ceci peut causer des problèmes de communication ou pourrait faire appel à d'autres méthodes pour communiquer des requis. Il serait alors intéressant d'étudier le domaine de l'information obtenue dans le design de jeu pour adapter les méthodes d'exprimer les requis. Également, nous avons constaté que les logiciels de divertissement proposent une interaction différente du logiciel interactif habituel. Ceci implique que le test doit être fait d'une façon différente. Une autre avenue intéressante serait donc d'analyser les types de tests les plus efficaces pour ce genre de logiciel.

En conclusion, nous voyons bien que la mesure du processus demeure une avenue intéressante pour l'amélioration des pratiques de développement. Une adaptation des métriques aux environnements est nécessaire, car chaque environnement est différent et présente des spécificités qui peuvent affecter les résultats attendus.

RÉFÉRENCES

BASILI, Vic et al. 2002. « What We Have Learned About Fighting Defects ». *Proceedings of the Eighth IEEE Symposium on Software Metrics 2002*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 249-258.

BASSIN, Kathryn A. et SANTHANAM, P. 1997. « Use of Software Triggers to Evaluate Software Process Effectiveness and Capture Customer Usage Profiles ». *Proceedings of the Eighth International Symposium on Software Reliability Engineering – Case Studies, 1997*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 103-114.

BEIZER, Boris. 1990. *Software Testing Techniques*. 2nd ed. New York: Van Nostrand Reinhold. 503 p.

BUTCHER, M. et al. 2002. «Improving Software Testing via ODC: Three Case Studies». IBM Systems Journal. [En ligne]. http://dx.doi.org/10.1147/sj.411.0031. (Page consultée en 2002-2003).

CARD, David N. July 1993. « Defect-Causal Analysis Drives Down Error Rates ». *IEEE Computer*. 10:4. 98-99

CARD, David N. Janvier-Février 1998. « Learning from Our Mistakes with Defect Causal Analysis ». *IEEE Computer*. 15:1. 56-63.

CHILLAREGE, Ram. Novembre 1992. « Orthogonal Defect Classification – A Concept for In-Process Measurements ». *IEEE Transactions on Software Engineering*. 18:11. 943-956.

EICKELMANN, Nancy et ANANT, Animesh. Mars-Avril 2003. « Statictical Process Control : What You Don't Measure Can Hurt You ». *IEEE Computer*. 20:2. 49-51.

FREDERICKS, Michael. 1999. *Using Defect Tracking and Analysis to Improve Software Quality*. College Park: University of Maryland. 51 p.

FREIMUT, Bernd. 2001. *Developing and Using Defect Classification Schemes*. Fraunhofer IESE. 53 p. IESE-Report 072.01/E.

GRADY, R.B. Sept. 1994. «Successfully Applying Software Metrics». *IEEE Computer*. 27:9. 18-25.

HETZEL, Bill. *The Complete Guide to Software Testing*. 2nd ed. Wellesley: QED Information Sciences. 1993. 296 p.

HOU, Rong-Huei. 1994. « Applying Various Learning Curves to Hyper-Geometric Distribution Software Reliability Growth Model ». *Proceedings of the 5th International Symposium on Software Reliability Engineering*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 8-17.

HUBER, John. 2000. « A Comparison of IBM's Orthogonal Defect Classification to Hewlett Packard's Defect Origins, Types, and Modes ». *Proceedings of the International Conference on Applications of Software Measurement (ASM)*. San Jose.

IBM RESEARCH. 2002. *Orthogonal Defect Classification*. [En ligne]. http://www.research.ibm.com/softeng/ODC/ODC.HTM. (Pages consultées en 2002-2004).

IEEE. 1994. *IEEE Standard Classification for Software Anomalies.* 1st ed. NewYork: The
Institute of Electrical and Electronics Engineers Inc. 32 p. Standard IEEE Std 1044-1993.

IEEE. 1996. *IEEE Guide to Classification for Software Anomalies.* 1st ed. NewYork: The Institute of Electrical and Electronics Engineers Inc. 60 p. Standard IEEE Std 1044.1-1995.

IVERSEN, Jakob et MATHIASSEN, Lars. 2000. «Lessons from Implementing a Software Metrics Program». *Proceedings of the 33rd Hawaii International Conference on System Sciences – 2000.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 908-918.

MALAIYA, Yashwant K. et DENTON, Jason. 2000. « Module Size Distribution and Defect Density ». *Proceedings of the 11th International Sympoisum on Software Reliability Engineering, 2000. ISSRE 2000.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 62-71.

NEUFELDER, Ann Marie. 2000. « How to Measure the Impact of Specific Development Practices on Fielded Defect Density ». *Proceedings of the 11th International Sympoisum on Software Reliability Engineering, 2000. ISSRE 2000.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 148-160.

ROBILLARD, Pierre-N et al. 2003. *Software Engineering Process with the UPEDU.* 1st ed. Montréal: Addison Wesley. 346 p.

WATSON, Arthur H. et McCabe, Thomas J. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* [En ligne]. Computer Systems Laboratory :

National Institute of Standards and Technology. NIS Publication 500-235, 1996. http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/title.htm (Page consultée entre novembre 2003 et janvier 2004).

WOOD, Alan. 1997. « Software Reliability Growth Models : Assumptions vs Reality ». *Proceedings of the Eighth International Sympoisum on Software Reliability Engineering, 1997.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 136-141

BIBLIOGRAPHIE SÉLECTIONNÉE

ANDREWS, Anneliese et STRINGFELLOW, Catherine. 2001. «Quantitative Analysis of Development Defects to Guide Testing: A Case Study ». *Software Quality Journal.* 9:3. 195 – 214.

ANDREWS, Anneliese et STRINGFELLOW, Catherine. 2002. « Deriving a Fault Architecture to Guide Testing ». *Software Quality Journal.* 10:4. 299-330.

BASSIN, Kathryn et al. Novembre-Décembre 1998. « Evaluating Software Development Objectively ». *IEEE Software.* 15:6. 66-74.

BASSIN, Kathryn A. et SANTHANAM, P. 2001. « Managing the Maintenance of Ported, Outsourced, and Legacy Software via Orthogonal Defect Classification ». *Proceedings of the IEEE International Conference on Software Maintenance, 2001.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 103-114.

BHANDARI, Inderpal. Décembre 1993. « A Case Study of Software Process Improvement During Developement ». *IEEE Transactions on Software Engineering.* 19:12. 1157-1170.

BHANDARI, Inderpal et al. 1994. « In-process Improvement Through Defect Data Interpretation ». *IBM Systems Journal.* 33:1. 182-214.

BIYANI, Shriram et SANTHANAM, P. 1998. « Exploring Defect Data from Development and Customer Usage on Software Modules Over Multiple Releases ». *Proceedings of the Ninth International Symposium on Software Reliability Engineering, 1998.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 316-320.

CARD, David N. 2002. « Managing Software Quality with Defects ». *Proceedings of the 26th Annual International Computer Software and Applications Conference, COMPSAC 2002*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 472-474.

CASEY, Brian K. et al. 1991. « Application of Defect Analysis Techniques to Achieve Continuous Quality and Productivity Improvements ». *Conference Records for the IEEE International Conference on Communications, 1991*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 1450-1454.

CHAAR, Jarir K. et al. 1993. « On the Evaluation of Software Inspections and Tests ». *Proceedings of the International Test Conference, 1993*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 180-189.

CHAAR, Jarir K. et al. Novembre 1993. « In-Process Evaluation for Software Inspection and Test ». *IEEE Transactions on Software Engineering*. 19:11. 1055-1070.

CHEN, T.Y. et al. 2000. « White on Black : a White Box-oriented Approach for Selecting Black Box-generated Test Cases ». *Proceedings of the First Asia-Pacific Conference on Software Quality*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 275-284.

CHILLAREGE, Ram. 1991. « Defect Type and its Impact on the Growth Curve ». *Proceedings of the 13th International Conference on Software Engineering, 1991*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 246-255.

CHILLAREGE, Ram et BIYANI, Shriram. 1994. « Identifying Risk Using ODC Based Growth Models ». *Proceedings of the 5th International Symposium on Software*

Reliability Engineering, 1994. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 282-288.

CHILLAREGE, Ram. Septembre 1996. « What is Software Failure? ». *IEEE Transactions on Reliability.* 45:03. 354-355.

CHILLAREGE, Ram et PRASAD, Kothanda Ram. 2002. « Test and Development Process Retrospective – a Case Study using ODC Triggers ». *Proceedings of the International Conference on Dependable Systems and Networks, 2002.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 669-678.

CHRISTENSON, Dennis A. et al. 1990. « Statistical Quality Control Applied to Code Inspections ». *IEEE Journal on Selected Areas in Communications.* 8:2. 196-200.

DOGSA, Tomaz et ROZMAN, Ivan. 1991. « The Influence of Syntactic and Semantic Errors on the Quality of Software ». *Proceedings of the Ninth International Symposium on Software Reliability Engineering, 1998.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 322-333.

DALCHER, Darren et TULLY, Colin. 2002. « Learning from Failures ». *Software Process Improvement and Practice.* 7:2. 71-89.

EL EMAM, Khaled et WIECZOREK, Isabella. 1998. « The Repeatability of Code Defect Classification ». *Proceedings of the International Symposium on Software Reliability Engineering, 1991.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 20-26.

GAME DEVELOPER MAGAZINE. 2004. *Gamasutra – The Art & Science of Making Games.* [En ligne]. http://www.gamasutra.com (Pages consultées en 2002-2003)

Gamedev.net. 2004. *Gamedev.net – all your game development needs.* [En ligne]. http://www.gamedev.net. (Pages consultées en 2002-2003)

GOMAA, Hassan et al. 2000. « Domain Modeling of Software Process Models ». Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems, 2000. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 50-60.

GRADY, Robert B. 1990. « Work-Product Analysis: The Philosopher's Stone of Software ». *IEEE Software.* 7:2. 26-34.

HALL, Tracy et al. 2000. « Towards Implementing Successful Software Inspections ». *Proceedings of the International Conference on Software Methods and Tools, 2000, SMT 2000.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, 127-136.

HOWDEN, William E. 1990. « Comments Analysis and Programming Errors ». *IEEE Transactions on Software Engineering.* 16:1. 72-81.

JOHNSON, Philip M. 1994. « An Instrumented Approach to Improving Software Quality through Formal Technical Review ». *Proceedings of the 16th International Conference on Software Engineering, 1994, ICSE-16.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 113-122.

KOBROSLY, Walid et VASSILIADIS, Stamatis. 1998. « A Survey of Software Functional Testing Techniques ». *Proceedings of the 1988 IEEE Southern Tier Technical Conference.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 127-134.

LAITENBERGER, Oliver. 1998. « Studying the Effects of Code Inspections and Structural Testing on Software Quality ». *Proceedings of the Ninth International Symposium on Software Reliability Engineering, 1998.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, 237-246.

LESZAK, Marek et al. 2000. « A Case Study in Root Cause Defect Analysis ». *Proceedings of the 2000 International Conference on Software Engineering.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 428-437.

LEVENDEL, Ytzhal. Février 1990. « Reliability Analysis of Large Software Systems: Defect Data Modeling ». *IEEE Transactions on Software Engineering.* 16:2. 141-152.

MALAIYA, Yashwant K. et DENTON, Jason. 1999. « Requirements Volatility and Defect Density ». *Proceedings of the 10th International Symposium on Software Reliability Engineering, 1999.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 285-294.

VON MAYRHAUSER, A. et al. 1999. « Deriving a Fault Architecture from Defect History ». *Proceedings of the 10th International Symposium on Software Reliability Engineering, 1999.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 295-303.

MAYS, Robert G. 1990. «Applications of Defect Prevention on Software Development». *IEEE Journal on Selected Areas in Communications.* 8:2. 164-168.

MCCONNELL, Steve. Mai-Juin 1997. « Gauging Software Readiness with Defect Tracking ». *IEEE Software.* 14:3. 135-136.

NAKAJO, Takeshi et KUME, Hitoshi. 1991. « A Case History Analysis of Software Error Cause-Effect Relationships ». *IEEE Transactions on Software Engineering*. 17:8. 830-838.

OHBA, M. 1995. « Software Error Data Collection and Analysis in Industry ». *Proceedings of the Sixth International Symposium on Software Reliability Engineering, 1995*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 270-271.

OWENS, Karen. 1997. « Software Detailed Technical Reviews: Finding and Using Defects ». *WESCON/97 Conference Proceedings*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 270-271. P. 128-133.

RAY, Bonnie K. et al. 1992. « Reliability Growth for Typed Defects ». *Proceedings of the Annual Reliability and Maintainability Symposium, 1992*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 327-336.

SELBY, Richard W. 1988. « Generating Hierarchical System Descriptions for Software Error Localization ». *Proceedings of the Second Workshop on Software Testing, Verification and Analysis, 1988*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 89-96.

SMIDTS, Carol. 1999. « A Stochastic Model of Human Errors in Software Development: Impact of Repair Times ». *Proceedings of the 10th International Symposium on Software Reliability Engineering, 1999*. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 94-104.

SULLIVAN, Mark et CHILLAREGE, Ram. 1991. « Software Defects and their Impact on System Availability – A Study of Field Failures in Operating Systems ». *Digest of*

papers of the Twenty-First International Symposium on Fault-Tolerant Computing, 1991 FTCS-21. New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 2-9.

SULLIVAN, Mark et CHILLAREGE, Ram. 1992. « A Comparison of Software Defects in Database Management Systems and Operating Systems ». *Digest of papers of the Twenty-Second International Symposium on Fault-Tolerant Computing, 1992 FTCS-22.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 475-484.

TACKETT, Buford D. et VAN DOREN, Buddy. Mai/Juin 1999. « Process Control for Error-free Software: A Software Success Story ». *IEEE Software.* 16:3. 24-29.

WELLER, Edward F. 2000. « Practical Applications of Statistical Process Control ». *IEEE Software.* 17:3. 48-55.

WINOKUR, Michael et al. 1998. « Measuring the Effectiveness of Introducing New Methods in the Software Development Process ». *Proceedings of the 24th Euromicro Conference, 1998.* New York: The Institute of Electrical and Electronics Engineers Inc. 2, 800-807.

WITHROW, Carol. 1990. « Error Density and Size in Ada Software ». *IEEE Software.* 7:1. 26-30.

WOHLIN, Claes. 2000. « Understanding the Sources of Software Defects: a Filtering Approach ». *Proceedings of the 8th International Workshop on Program Comprehension, IWPC 2000.* New York: The Institute of Electrical and Electronics Engineers Inc. 1, P. 9-17.

YU, Tze-Jie et al. Septembre 1988. « An Analysis of Several Software Defect Models ». *IEEE Transactions on Software Engineering*. 14:9. 1261-1270.

ANNEXES

ANNEXE I: LISTE DES DÉCLENCHEURS (TRIGGERS) ORIGINAUX

Cette annexe contient la liste des déclencheurs associés aux différentes activités qui peuvent être sélectionnées par le développeur. Le format est le suivant, pour chaque déclencheur, on retrouve l'activité ou les activités pour lesquelles le déclencheur peut être choisi. Ensuite, il y a la description du déclencheur pour déterminer quand celui-ci doit être sélectionné.

Design Conformance (Design Review/Code Inspection)

Le réviseur/inspecteur détecte le défaut en comparant le design ou un segment de code par rapport à la spécification de l'étape précédente. Ceci inclut des documents de design, du code, des standards de programmation, ou pour s'assurer que des requis de design ne sont pas manquants ou ambigus. (Design vs Spécification / Code vs Design)

Logic/Flow (Design Review/Code Inspection)

Le réviseur/inspecteur utilise des notions de base en programmation pour examiner les flots de logique ou de données pour s'assurer qu'ils sont corrects et complets.

Backward Compatibility (Design Review/Code Inspection)

Le réviseur/inspecteur utilise ses connaissances approfondies d'un composant/produit pour identifier une incompatibilité entre le design ou le code, et une version précédente du même composant/produit. Du point de vue du client, la fonctionnalité fonctionnerait avant, mais plus maintenant.

Internal Document (Design Review/Code Inspection)

Il y a une information incorrecte, incompatible ou incomplète avec la documentation interne. Des entêtes de fonction ou de fichier, ainsi que les commentaires de code font partie de cette catégorie.

Lateral Compatibility (Design Review/Code Inspection)

L'inspecteur détecte une incompatibilité entre une fonction décrite par le design ou le code, et tout autre système, produit, service, composant ou module avec lequel elle doit interfacer. Typiquement, ces problèmes mettraient en péril la compatibilité avec un autre système.

Concurrency (Design Review/Code Inspection)

L'inspecteur/réviseur considérait l'ordonnancement nécessaire au contrôle d'une ressource partagée lors de la découverte du défaut. Ces erreurs peuvent être rattachées à la multiplicité de fonctions, de *threads*, de processus ou des contextes de *kernel*, ainsi que le contrôle de sémaphores, etc.

Language Dependency (Design Review/Code Inspection)

L'inspecteur détecte le défaut en vérifiant des détails spécifiques à un langage. Ceci concerne les standards de langage, les spécificités de compilateurs, etc.
Un cas classique est « = » à la place de « = = » dans du code C.

Side Effects (Design Review/Code Inspection)

L'inspecteur utilise sa connaissance du produit/composant pour prédire un comportement du système, d'un produit, d'une fonction ou d'un composant. L'effet

secondaire est caractérisé comme résultant d'un usage commun ou d'une configuration commune, mais dont l'effet ne fait pas partie de la portée du produit/composant.

Rare situations (Design Review/Code Inspection)

L'inspecteur utilise ses connaissances approfondies du produit/composant pour prédire un comportement non considéré par le design ou le code, associé à une configuration inhabituelle ou un usage inattendu. Ceci n'inclus pas une incomplétude ou une absence de traitement d'erreur, qui appartient plutôt au domaine de Design Conformance.

Simple Path (Unit Test)

Le cas de test est basé sur la connaissance d'un seul chemin spécifique dans le code et non sur sa fonctionnalité. Ce déclencheur ne peut être sélectionné suite à un avis d'erreur d'un client, à moins que celui-ci connaisse bien le code et spécifie un chemin bien particulier.

Complex Path (Unit Test)

Dans ce cas-ci, le cas de test exécutait des combinaisons planifiées de chemins de code. Le testeur essaie d'exécuter plusieurs branchements sous plusieurs conditions différentes.

Coverage (Function Test)

Durant un cas de test de type boîte noire, le défaut a été trouvé en essayant une fonctionnalité directement, sans l'utilisation de paramètres ou un seul ensemble de paramètres particuliers, mais non inattendus.

Variation (Function Test)

Durant un cas de test de type boîte noire, le défaut a été trouvé en essayant une fonctionnalité directement mais avec plusieurs paramètres différents. Il est important de noter que les paramètres invalides, les valeurs extrêmes, les conditions frontières font partie de cette catégorie, ainsi que l'utilisation de plusieurs combinaisons de paramètres réguliers.

Sequencing (Function Test)

Durant un case de test de type boîte noire, le défaut a été trouvé en exécutant plusieurs fonctions indépendantes en fonctionnalités dans un ordre bien spécifique. Ce trigger est utilisé lorsque chaque fonction réussit indépendamment, mais il y a erreur dans une séquence précise de ces fonctions.

Interaction (Function Test)

Durant un cas de test de type boîte noire, le défaut a été trouvé en exécutant une interaction entre deux (ou plus) fonctionnalités. Ce déclencheur est utilisé lorsque chaque fonction réussit indépendamment, mais il y a erreur dans une séquence précise de ces fonctions. De plus, ce choix de déclencheur indique que les fonctionnalités impliquées ont un lien fonctionnel entre elles, un lien plus « intime » selon la définition originale du terme.

Workload/Stress (System Test)

Le system fonctionne près d'une limite supérieure ou inférieure de ressources. Ces limites peuvent être atteintes de plusieurs façon, incluant l'arrivée de petites ou grandes

charges, l'exécution de plusieurs ou peu de "produits" à la fois, ou l'exécution du produit pour des longues durées.

Recovery/Exception (System Test)

Le système est testé pour répondre à une exception ou du code de rétablissement. Dans le contexte de la découverte par le client, l'erreur est celle du recouvrement et non l'erreur qui a déclenché le recouvrement.

Startup/Restart (System Test)

Le système ou sous-système s'initialisait ou repartait à la suite d'une terminaison ou une erreur fatale.

Hardware Configuration (System Test)

Le système est testé pour assurer le bon fonctionnement sous une configuration matérielle particulière.

Software Configuration (System Test)

Le système est testé pour assurer le bon fonctionnement sous une configuration logicielle particulière.

Blocked Test (ou Normal Mode) (System Test)

Le système fonctionne dans des conditions dites normales et le défaut est apparu lors de l'exécution d'un scénario de test système. Ce trigger est utilisé quand le scénario n'a pu

être complété car il existe des problèmes qui empêchent son exécution. Ce *trigger* ne doit pas être sélectionné par un défaut trouvé par le client.

En pratique, nous avons modifié quelque peu la définition originale de ce dernier déclencheur. Sa nouvelle définition peut être trouvée dans l'annexe suivante.

ANNEXE II: LA LISTE DES IMPACTS ORIGINAUX

Impact probable (erreurs trouvées en développement/test) ou observé (erreurs trouvées par le client) associé au défaut. Les propriétés du logiciel décrites ci-après sont celles qui seraient affectées par le défaut.

Usability

Le défaut empêche le logiciel et sa documentation à rendre celui-ci facile à comprendre et à utiliser par l'usager.

Installability

Le défaut affecte l'installation normale du logiciel par l'usager (sans inclure *Usability*).

Integrity/Security

Le défaut affecte les protections du système, du programme et des données d'une destruction ou d'une altération, que celles-ci soient malicieuses ou non.

Performance

Le défaut affecte la vitesse telle que vue par l'usager dans l'accomplissement des tâches. Concerne aussi la consommation de mémoire.

Maintenance

Le défaut affecte la facilité d'appliquer des réparations correctives ou préventives sur le logiciel.

Serviceability

Le défaut affecte la fonctionnalité du logiciel qui permet de diagnostiquer les problèmes facilement et rapidement, avec un impact minimal sur le client. Un exemple de ceci peut être un message d'erreur inexact.

Migration (Compatibilité)

Le défaut affecte la possibilité de mise à jour du logiciel au niveau du programme ou des données. Ceci peut également inclure l'inadéquation de la documentation face aux mises à jour.

Documentation

La documentation est incorrecte ou incomplète

Standards

Le défaut fait en sorte que le logiciel ne suit pas un standard particulier et pertinent au contexte. Un exemple de ceci serait un défaut qui affecte l'apparence standard d'une application sous le système d'exploitation Windows

Reliability

Le défaut affecte la fiabilité du logiciel. Il empêche le logiciel de remplir ses fonctions de façon constante sans interruption inattendue. Des *crash* sont des symptômes associés à cet impact.

Requirements

On sélectionne cet impact lorsqu'un requis n'a pas été identifié, a mal été compris ou a été mal priorisé. Cet impact devrait être choisi en développement lorsque les plans concernants les requis sont modifiés. Lorsque le produit a été livré, cet impact est choisi lorsque les clients indiquent une insatisfaction face à leurs attentes (par rapport aux requis).

Accessibility

Le défaut empêche l'utilisation du logiciel par des gens ayant des handicaps.

Capability

Le défaut affecte la capacité du logiciel à remplir entièrement sa fonctionnalité, pour un requis connu.

ANNEXE III: LES MODIFICATIONS APPORTÉES À ODC

Cette annexe inclut les changements qui ont été apportés à ODC pour mieux l'adapter à notre environnement expérimental. Pour chaque paramètre d'ODC, les modifications qui ont été apportées sont indiquées, s'il y a lieu.

Activity

Aucune modification n'a été apportée aux choix de ce paramètre.

Trigger

Pour les déclencheurs, tous les choix ont été conservés, mais une avec une définition étendue. Il s'agit du choix *Blocked Test.*

La définition originale, telle que décrite dans l'annexe I, est la suivante :

Le système fonctionne dans des conditions dites normales et le défaut est apparu lors de l'exécution d'un scénario de test système. Ce trigger est utilisé quand le scénario n'a pu être complété car il existe des problèmes qui empêchent son exécution. Ce *trigger* ne doit pas être sélectionné par un défaut trouvé par le client.

La première chose qui a été changée est que ce déclencheur doit être choisi lorsqu'il est difficile de déterminer quelles actions ont été posées pour trouver l'erreur. C'est effectivement le cas d'un jeu ou d'un logiciel comportant des fonctionnalités invisibles à l'usager. Dans ce cas-là, il serait difficile de choisir un déclencheur approprié, alors il a été décidé que les défauts trouvés de cette manière iraient dans cette catégorie.

Le deuxième changement est donc que ce type de déclencheur peut donc être choisi pour les défauts trouvés par des clients.

Impact

Quelques changements surviennent dans cette catégorie.

Premièrement, l'impact sur la maintenance inclus maintenant le concept de maintenance au niveau du code pour les développeurs. Ainsi, il devient maintenant possible que certains défauts soient choisis car ils affectent la maintenance du code.

Deuxièmement, les impacts Accessibility et Installability ne doivent pas être choisis, car ils ne s'appliquent pas dans la situation de l'expérience.

Target

Rien n'a changé pour ce paramètre.

Defect Type

Les définitions sont conservées, mais le type *Relationship* a été inclus dans le type *Function/Class/Object* afin d'éviter des erreurs. Ces deux types se ressemblent en ce sens qu'ils affectent le design du logiciel.

Qualifier

Rien n'a changé dans cette catégorie.

Source

Le choix *Outsourced* a été étendu aux erreurs trouvées dans de l'ancien code alors que le produit était sous la responsabilité de l'équipe française. Vu que l'équipe a changé et qu'il s'agit de code venant de l'extérieur de cette équipe, on peut considérer qu'il s'agit de code provenant d'une autre source.

Age

Rien n'a changé dans cette catégorie.