Un Point sur la Modélisation des Exceptions

Yolande Ahronovitz

Un point sur la modélisation des exceptions

Yolande Ahronovitz

septembre 2005

Rapport de recherche RR-05 12719

LIRMM

1 Introduction

Dans une présentation au workshop sur les exceptions de ECOOP 2000, puis dans un article publié début 2001 [AH01], nous constations que la gestion d'exceptions était largement étudiée dans la littérature, mais que la conception proprement dite des exceptions (comment les trouver, les décrire, les organiser) était très rarement abordée.
Nous proposions dans ce papier un guide pour la conception des exceptions à partir des éléments du modèle objet statique. Nous avons réalisé depuis un certain nombre d'investigations pratiques en Java, travail de fond non encore abouti.
Parallèlement, nous voudrions faire le point de la situation actuelle : est-ce que la conception des exceptions est plus étudiée maintenant qu'il y a cinq ans ?
Il faut se rendre à l'évidence : il est possible que le sujet intéresse, mais ça ne se traduit pas en publications qui lui seraient entièrement dédiées.
Par ailleurs, les progrès en "conceptualisation" des systèmes de gestion d'exceptions se poursuivent : du papier de base de R. Miller et A. Tripathi [MT97] au dernier workshop sur le sujet à ECOOP 2005 [RDKT05], où l'on trouve bon nombre de présentations sur la modélisation des systèmes de gestion d'exceptions (dont une en UML [CGGP05]), le chemin parcouru est significatif.
Or, lorsqu'on veut modéliser les systèmes de gestion d'exceptions, il faut mettre un peu d'ordre dans les exceptions elles-mêmes.
D'où l'idée d'essayer de recenser, dans les articles sur la gestion d'exceptions, les bouts et morceaux qui parlent des catégories d'exceptions, de leur organisation, de l'art de les utiliser.
Ces morceaux très variés se rangent plus ou moins dans trois rubriques : classement des exceptions suivant divers critères, conseils et aides pour bien faire, résultats d'expériences et analyses de programmes.
Le texte qui suit donne les premiers résultats de ce recensement, toujours en cours actuellement.

2 À chacun sa classification : les catégories d'exceptions

Comme le dit J. Kiniry dans [Kin03], il faut considérer ce que les exceptions représentent : une structure pour gérer les situations anormales imprévisibles, une structure pour le flot de contrôle, ou quelque chose entre les deux ?
Suivant le sens vers lequel on penche, les critères de classification sont plutôt décrits par des "états du système" ou représentés par une situation relativement précisément modélisée.

Un état anormal Dans leur papier [MT97], R. Miller et A. Tripathi définissent une exception comme "un état anormal dans un calcul", et identifient quatre causes d'exceptions :
- une *erreur :* état système non valide et non autorisé ;
- une *déviation :* état système non valide autorisé ;
- une *notification :* information à l'appelant que l'état système précédent ou supposé a changé ;
- un *idiome :* autres cas, où l'occurrence d'exception n'est pas une anomalie, mais plutôt une chose rare.

Compartiments À l'opposé, M.Robillard et G.Murphy [RM00], après avoir été débordés par des exceptions un peu désordonnées, cherchent une solution dans une

organisation de la structure du logiciel. Ils proposent de découper un logiciel en compartiments (subdivisions faites en fonction de la logique du programmeur), et de fixer complètement les interfaces entre compartiments. Ils distinguent ensuite trois cas :
- les exceptions *de haut niveau*, qui sont inter-compartiments, et sont donc lancées en sortie d'un compartiment ;
- les *bas-niveau* ou *internes*, qui sont intra-compartiment, sont lancées à l'intérieur d'un compartiment, et sont censées être traitées à l intérieur ;
- si une exception interne ne peut pas être traitée dans son compartiment, on lance à l'extérieur une exception de haut niveau qui la remplace.

D.Malayeri et J.Aldrich [MA05] se réclament de Robillard et proposent une notion de module, avec la même idée de découpage du logiciel.
Ils ne semblent cependant pas estimer la granularité de la même façon. Les deux équipes travaillent en Java, où il y a une notion de package. M.Robillard et G.Murphy voient le compartiment comme un sous-ensemble logique du package, alors que D.Malayeri et J.Aldrich voient le module comme un regroupement de plusieurs packages.

Structure des Composants Dans le même esprit d'organisation, mais avec une optique différente, A. Romanosvsky et ses collègues [GRRX01] disent qu'un composant système retourne une exception quand il ne peut pas satisfaire une requête, et classent les exceptions en :
- *exceptions internes :* lancées par le composant pour invoquer sa propre gestion interne de tolérances aux fautes ;
- *exceptions externes* subdivisées en :
  - *exceptions d'interface :* lancées quand une requête est non conforme (ce n'est pas le composant qui les rencontre qui peut les gérer) ;
  - *exceptions de défaillance (failure) :* lancées lorqu'un composant constate que, pour une raison quelconque, il ne peut pas rendre le service demandé.

Dynamique des Composants Dans un papier qui traite de gestion d'exceptions dans les systèmes à base de composants [SUVD03], F. Souchon et ses collègues déterminent trois catégories :
- *exceptions critiques :* elles doivent être traitées au plus vite ;
- *exceptions non critiques :* elles peuvent être mises en attente ;
- *exceptions concertées :* quand la conjonction de plusieurs exceptions mises en attente permet de diagnostiquer un problème unique, on lance une seule exception, dite exception concertée, qui représente ce problème (type d'exception présenté dans [Iss01]).

Systèmes collaboratifs Avec une notion d'organisation basée sur la dynamique du système, A.Tripathi et ses collègues [TKA05] définissent des catégories d'exceptions pour les systèmes collaboratifs. Ces systèmes comportent des membres qui ont des rôles à exécuter, d'où une exception *RoleMember* pour un membre qui quitte son rôle de façon inattendue, et une exception *Obligation* pour une opération de rôle non exécutée alors que la précondition d'exécution est vraie depuis un certain temps, à côté d'exceptions plus classiques (*Object Binding Exception*, *Application Exception*).

Scénarios Un critère de classification très différent est retenu par N. Maiden et ses collègues dans [MMS+99]. Cet article traite de scénarios (voir plus de détails dans la section 3), et s'intéresse aux comportements anormaux dans le modèle dynamique d'une application. Il propose trois grands types d'exceptions :

– *exception générique :* liée aux composants de base d'un scénario (événements, états, objets, actions, agents) ;
– *exception de permutation :* liée à la séquence d'événements du scénario : inversion d'événements, répétition incorrecte d'un événement, survenue prématurée d'un événement, etc ;
– *exception "problème" :* liée à un événement ou un état inattendu à l'interface entre le système qu'on modélise et son environnement externe.

Synthèse Dans tous ces exemples (et quelques autres non cités), on trouve une belle palette de possibilités de catégorisations, définies chacune de façon indépendante. Elles ont en fait deux caractéristiques remarquables :
– elles ne se contredisent pas : si elles sont pertinentes pour un même problème, elles peuvent se croiser ou se compléter ;
– elles reflètent presque toutes la forte poussée du besoin de modularité et d'encapsulation : d'une façon ou d'une autre, elles séparent les exceptions "locales", à gérer sur place, des exceptions qu'il faut envoyer se faire gérer ailleurs.

3 À tout le monde la même éthique : les conseils et aides pour bien faire

Ces conseils et aides s'adressent, suivant les articles, aux programmeurs, aux concepteurs de langages de programmation, et aux concepteurs de modèles (il n'y a rien pour les concepteurs de langages de modélisation, on n'en est pas encore là).

3.1 Conseils aux programmeurs

Ils s'adressent surtout, apparemment, aux programmeurs Java, non parce qu'ils se débrouillent moins bien que les autres, mais parce qu'ils sont à peu près les seuls à utiliser un langage qui les oblige à prendre en compte les exceptions[1] , que ça leur plaise ou non (voir section 4).
Une fois habitués à gérer des exceptions, ils ont tendance à vouloir en définir, donc les conseils portent à la fois sur la définition et la gestion des exceptions.Ceux cités dans cette section ne concernent bien sur que la définition des exceptions.

De désormais très classiques conseils sont donnés par B. Venners [Ven98] :
– prévoir des exceptions pour un mauvais usage d'une classe, pas pour des choses normales ;
– distinguer les exceptions de précondition (mauvais appel d'une méthode, par exemple) des exceptions de postcondition (ex : il y a un problème dans une méthode appelée correctement) ;
– définir des exceptions spécifiques (pas d'exception "fourre-tout" unique à lancer en toutes circonstances) ;
– lorsqu'on définit une exception, prévoir correctement si elle doit être contrôlée ou non ("checked" ou "unchecked") ; si elle est contrôlée, le client est obligé de s'en occuper, donc il doit en avoir les moyens.

J. Bloch, dans [Blo01], donne à peu près les mêmes.
M.Robillard, dans [RM00], s'en tient à des conseils de bonne conduite ou de cohérence élémentaire : documenter complètement les interfaces, ne pas utiliser les exceptions prédéfinies du système pour n'importe quoi ....

[1] Une *checked exception* doit obligatoirement être traitée, soit en la rattrapant, soit en disant explicitement qu'on la relance ; et un bon nombre d'exceptions du système sont de ce type. La plupart des langages sont plus accommodants (voir C++)

Tous ces conseils relèvent du bon sens et doivent bien sûr être suivis. Mais ils ont un côté très inquiétant : s'ils n'apportent pas d'aide à l'identification des exceptions dans une application précise, ce n'est pas parce qu'ils supposent que ce travail a été fait à la modélisation, avant de programmer. Bien au contraire, ils partent du principe que rien n'a été prévu, et qu'il faut se débrouiller "au vol".

3.2 Conseils pour les concepteurs de langages de programmation

Les articles qui les prodiguent sont beaucoup plus rares que ceux qui s'adressent aux programmeurs.

Le plus ancien est le papier de R. Miller et A. Tripathi [MT97], qui essaie d'identifier avec précision les conflits entre les exigences de la gestion des exceptions et les buts de la modélisation orientée objet.
Il ne donne pas de conseils à proprement parler, mais il met en évidence des propriétés qu'aucun langage n'implémente (mais sont-elles implémentables ?). Parmi ces propriétés, une est la consistance d'une exception : chaque exception doit avoir un seul sens associé, indépendant de l'endroit où elle est signalée.

L'article de J.Kiniry [Kin03] est un des rares qui s'intéressent à la représentation des exceptions, et non à leur gestion.
En s'appuyant sur deux langages à points de vue opposés (Java : les exceptions comme structure de contrôle, Eiffel : les exceptions pour gérer les situations imprévisibles), il présente sept points-clés, dont quatre sont des conseils.
Deux de ces conseils portent sur la sémantique des exceptions :
- le premier rejoint celui de Miller et Tripathi : une exception doit avoir une sémantique consistante et uniforme ;
- le deuxième concerne les exceptions utilisées pour représenter les erreurs sur les assertions : leur conception et leur sémantique doivent appartenir au noyau de la spécification du langage.

Les deux autres conseils concernent la représentation d'une exception :
- elle doit être uniforme (pas deux types de représentation possibles, comme c'est le cas par exemple en Eiffel avec les nombres et les chaînes) ;
- s'il existe une représentation compréhensible à la fois pour l'homme et la machine, elle ne doit pas comporter de parties optionnelles (pas de méthodes par défaut qu'on peut ne pas redéfinir, comme le getMessage de Java).

Ces conseils sont assez simples à exprimer, mais dans la mesure où ils concernent principalement la sémantique, leur mise en œuvre n'est pas forcément facile.

3.3 Conseils et aides pour les concepteurs de modèles

Il s'agit ici de modélisations appuyées sur UML.

Voici d'abord un bref rappel de nos propres travaux. Dans [AH01], nous avons proposé un guide pour aider à l'identification d'exceptions en inspectant les classes, les propriétés, les associations et les contraintes définies dans le modèle objet statique d'une application.
Ce guide, assorti de quelques ajouts au métamodèle UML, permet de rechercher toutes les possibilités d'exceptions décelables "en statique".
Ce parcours exhaustif donne un résultat en général volumineux, que le concepteur doit ensuite trier pour prévoir uniquement les exceptions utiles.

Nous travaillons actuellement à une "aide au tri" plus élaborée que de simples conseils.

Un très intéressant travail, complémentaire du nôtre, a été réalisé par N. Maiden et ses collègues [MMS+99], pour aider à l'identification d'exceptions dans le modèle dynamique.
Ils ont mis au point un logiciel de génération et d'utilisation de scénarios, appelé CREWS-SAVRE.
On utilise habituellement les scénarios pour simuler la façon dont le futur système doit se comporter. Les auteurs de ce papier suggèrent de les utiliser pour simuler aussi ce que le futur système ne doit pas faire.
Pour pouvoir mettre en place ces scénarios d'anomalies, il faut identifier les cas d'exceptions.
N. Maiden et ses collègues proposent donc une classification des exceptions en trois groupes,déjà présentée ici en section 2.
Pour les deux premiers groupes, liés aux composants de base du scénario pour l'un (exceptions génériques), aux défauts de séquence dans le déroulement pour l'autre (exceptions de permutation), ils donnent des listes assez précises d'exceptions courantes. Pour le troisième groupe, lié à des problèmes à l'interface du système et de son environnement externe, ils fournissent toute une série de taxonomies, en fonction des domaines d'application.
Ces différentes listes et taxonomies sont stockées dans une base de données associée au système de simulation, donc l'utilisateur dispose d'un outil assez complet.

Il est à remarquer que ces deux systèmes sont des aides de base, et que le concepteur est censé utiliser son discernement, sa logique, et sa connaissance du domaine.
Peut-on espérer une évolution de la recherche vers de véritables systèmes d'aide élaborée pour l'identification des exceptions ?

4 La dure réalité du programmeur : résultats d'expériences et analyses de programmes

4.1 Résultats d'expérience

Un papier de R. Maxion et R. Olszewski [MO00] rend compte d'une expérience menée avec des étudiants. Elle montre surtout que des programmeurs qui collaborent et qui appliquent une méthodologie commune obtiennent de meilleurs résultats que s'ils travaillent séparément et regroupent leurs résultats à la fin, ce qui doit être vrai dans d'autres domaines que les exceptions.
Une autre constatation, plus spécifique de ce qui nous intéresse, est que beaucoup des programmeurs de cette expérience ne savent pas traiter les exceptions qu'ils ont eux-même prévues, faute d'avoir réfléchi à ce qu'elles représentent.
Ceci rejoint ce qui avait déjà été constaté dans la section 3.1 : le principe de base semble être : se débrouiller "au vol".

4.2 Résultats d'analyses de programmes

4.2.1 Analyse de structure

Un des problèmes fondamentaux posés par la gestion des exceptions est qu'elle complique singulièrement le flot de contrôle d'un programme. Le fait de pouvoir, en caricaturant, "sauter n'importe où n'importe quand", rappelle fâcheusement les `goto` qui ont été si difficiles à museler et à encadrer.

Plusieurs auteurs font donc des tentatives pour suivre le flot de contrôle d'un programme et le visualiser, sans faire d'exécution effective.

Parmi eux, S. Sinha et M.J. Harrold [SH00] décrivent tout ce qu'il faut ajouter aux principaux types d'analyse de flot classiques pour traiter les exceptions (explicites et synchrones).

M.Robillard et G.Murphy [RM99] proposent un outil plus facile à manipuler : le logiciel JEX extrait le flot d'exception du texte d'un programme Java, et visualise les structures `try...catch` et les appels pouvant donner lieu à exception.

Dans le même domaine, D. Malayeri et J. Aldrich [MA05] proposent ExnJava, qui permet en plus d'agir en réorganisant un peu le code (il s'agit principalement de bien utiliser les outils fournis par Java pour gérer les exceptions).

Tous ces outils sont faits pour mettre en évidence la façon dont les exceptions sont gérées par un programme. Mais les résultats qu'ils fournissent conduisent tout naturellement à s'interroger sur la façon dont elles sont définies.

4.2.2 Analyse de contenu

Une autre façon d'analyser les programmes consiste à regarder, non pas la syntaxe utilisée par les programmeurs pour traiter les exceptions, mais la sémantique qu'ils y ont mise.
C'est à ce travail que se sont attelés D. Reimer et H. Srinivasan [RS03] : ils ont étudié l'usage fait des exceptions dans les applications Java de grande taille.

Ce qu'ils ont trouvé est catastrophique :
- de véritables sottises : exceptions rattrapées avec un `catch` vide, utilisation d'un `catch` unique pour rattraper toutes les exceptions possibles dans un bloc `try`, alors qu'elles n'ont aucun rapport entre elles (sauf le fait de pouvoir être lancées dans le même bloc `try`) ;
- des défauts d'utilisation patents : handlers limités à un message sans intérêt, propagation d'exceptions à travers une bonne partie du programme pour les traiter le plus loin possible de là où elles ont été lancées.

Ces sottises et défauts prouvent d'abord que, lorsqu'on veut forcer les programmeurs à utiliser des exceptions, ils le font mal. Mais ils montrent aussi une incompréhension fondamentale de ce que représente une exception.

4.3 Discussion

Que tirer de tout cela ? De toute évidence, la pratique des exceptions n'est pas une réussite.

Les tentatives pour rendre obligatoire la gestion des exceptions (par exemple les *checked exceptions* en Java) induisent beaucoup de mauvais travail.
D'après J. Kiniry [Kin03], qui a étudié un grand nombre de programmes, elles augmentent la fragilité d'un système. Mieux vaut intégrer le plus possible de contrats à la Eiffel dans un langage : le nombre et la complexité des exceptions en seront réduits.
Autre effet négatif, dit-il : cette obligation de gérer provoque ... l'angoisse des programmeurs (il attribue à cette angoisse les sottises citées ci-dessus). Mais il est vrai que gérer des exceptions définies par d'autres personnes est un exercice difficile quand ces exceptions sont peu documentées et que leur signification n'est pas claire.

Faut-il donc imposer d'écrire les spécifications d'exceptions en même temps que les autres spécifications ? Ce peut être un moyen d'apprivoiser les exceptions en se les appropriant.
Toujours d'après J.Kiniry, la réponse n'est pas évidente. Au vu des résultats obtenus, cette "obligation de spécifications" augmente à la fois la qualité du code et la productivité du développeur quand il s'agit de petits programmes, mais pour de grands projets logiciels on ne gagne à peu près rien en qualité et on perd en productivité.

On retrouve là le problème de la sémantique des exceptions, avec deux situations significativement différentes :
- ce sont ceux qui prévoient les exceptions qui les gèrent : petit programme, application écrite par un petit nombre de personnes proches ;
- on est dans une situation client-serveur, où le serveur lance les exceptions et où le client peut/veut les rattraper ou non : la JVM et les programmeurs java, les grandes applications où des équipes séparées collaborent uniquement par interfaces.

Dans le premier cas, le contact humain permet de compléter les spécifications ; dans le deuxième, il faut s'en tenir à ce qui est écrit.
On retombe donc dans le problème général de l'écriture des spécifications : soit elles sont écrites dans un langage très codifié, et elles deviennent syntaxiques, soit elles sont écrites dans un langage un peu informel, et elles sont floues. Et dans le cas des exceptions, on a un inconvénient supplémentaire : au lieu de décrire ce qu'on veut, il faut caractériser ce qui est anormal, ce qu'on ne veut pas.

5 Conclusion

Les tentatives pour cerner la notion formelle d'exception se développent. Il est acquis que l'exception est un objet, une entité reconnue. Le besoin de classification, même s'il se traduit en un foisonnement de critères, est encourageant. Et la nécessité de modularité donne au moins deux critères implicitement présents presque partout : les exceptions locales et les autres.

À côté de ces travaux, on trouve des préoccupations de concepteurs de modèles, qui cherchent à bien identifier les exceptions liées à une application, et à les représenter le plus clairement possible.

Et il y a enfin la pratique quotidienne des programmeurs. Dans beaucoup de cas, c'est un désastre qui peut se résumer de façon simple : on ne s'occupe pas des exceptions, on s'en débarrasse. Il manque manifestement une "culture de l'exception" : c'est un concept mal cerné, donc la mise en œuvre n'est pas bonne.

L'étude de la sémantique des exceptions, qui est un courant de préoccupation récent, commence à répondre à ce problème en caractérisant des besoins de base : consistance, uniformité du sens.

Cet aspect est fondamental pour l'avenir des exceptions : même si on progresse beaucoup dans la classification des types d'exceptions, et dans l'identification des exceptions d'une application donnée, rien ne sera utile si on n'aboutit pas à des spécifications claires pour le programmeur.

Références

[AH01] Y. Ahronovitz and M. Huchard. Exceptions in Object Modeling : Finding Exceptions from the Elementd of the Static Object Model. In *Advances in Exception Handling Techniques*, number 2022 in LNCS, pages 77–93. Springer Verlag, 2001.

[Blo01] J. Bloch. *Effective Java Programming Language Guide*. The Java Series. Addison Wesley, 2001.

[CGGP05] A. Capozucca, B. Gallina, N. Guelfi, and P. Pelliccione. Modeling Exception Handling : a UML2.0 Platform Independent Profile for CAA. In *Workshop Exception Handling in Object Oriented Systems : Developing Systems that Handle Exceptions*, pages 88–99. ECOOP 2005, july 2005. Glasgow.

[GRRX01] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. *Journal of Systems and Software*, 59(2) :197–222, november 2001.

[Iss01] V. Issarny. Concurrent Exception Handling. In *Advances in Exception Handling Techniques*, number 2022 in LNCS, pages 111–127. Springer Verlag, 2001.

[Kin03] J.R. Kiniry. Exceptions in Java and Eiffel : Two Extremes in Exception Design and Application. In *Workshop Exception Handling in Object Oriented Systems : Towards Emerging Application Areas and New Programming Paradigms*, pages 27–38. ECOOP 2003, july 2003. Darmstadt.

[MA05] D. Malayeri and J. Aldrich. Practical Exception Specifications. In *Workshop Exception Handling in Object Oriented Systems : Developing Systems that Handle Exceptions*, pages 100–111. ECOOP 2005, july 2005. Glasgow.

[MMS+99] N. Maiden, S. Minocha, A. Sutcliffe, D. Manuel, and M. Ryan. A Cooperative Scenario Based Approach to Acquisition and Validation of System Requirements : How Exceptions Can Help ! *Interacting with Computers*, 11 :645–664, november 1999.

[MO00] R. Maxion and R. Olszewski. Eliminating Exception Handling Errors with Dependability Cases : a Comparative,Empirical Study. *IEEE Transactions on Software Engineering*, 26(9) :888–906, september 2000. Volume "Special Section on Current Trends in Exception Handling".

[MT97] R. Miller and A. Tripathi. Issues with Exception Handling in Object-Oriented Systems. In M. Aksit and S. Matsuoka, editors, *ECOOP '97 Object Oriented Programming, 11th European Conference*, number 1241 in LNCS, pages 85–103. Springer Verlag, june 1997.

[PAT00a] D. Perry, A.Romanovsky, and A. Tripathi, editors. *Special Section on Current Trends in Exception Handling*, volume 26 number 9 of *IEEE Transactions on Software Engineering*. IEEE Computer Society, september 2000.

[PAT00b] D. Perry, A.Romanovsky, and A. Tripathi, editors. *Special Section on Current Trends in Exception Handling-Part II*, volume 26 number 10 of *IEEE Transactions on Software Engineering*. IEEE Computer Society, october 2000.

[RDKT01] A. Romanovsky, C. Dony, J.L. Knudsen, and A. Tripathi, editors. *Advances in Exception Handling Techniques*. Number 2022 in LNCS. Springer Verlag, 2001.

[RDKT03] A. Romanovsky, C. Dony, J.L. Knudsen, and A. Tripathi, editors. *Workshop Exception Handling in Object Oriented Systems : Towards Emerging Application Areas and New Programming Paradigms*. ECOOP 2003, july 2003. Darmstadt.

[RDKT05] A. Romanovsky, C. Dony, J.L. Knudsen, and A. Tripathi, editors. *Workshop Exception Handling in Object Oriented Systems : Developing Systems that Handle Exceptions*. ECOOP 2005, july 2005. Glasgow.

[RM00] M. Robillard and G. Murphy. Designing Robust Java Programs with Exceptions. In D. Rosenblum and J. Knight, editors, *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering, FSE '00*, pages 2–10. ACM Press, november 2000.

[RM99] M. Robillard and G. Murphy. Analyzing Exception Flow in Java Programs. In O. Nierstrasz and M. Lemoine, editors, *ESEG/FSE'99, 7th European conference*, number 1687 in LNCS, pages 322–337. Springer Verlag, september 99. also in Software Engineering Notes, SIGSOFT, vol.24 number 6.

[RS03] D. Reimer and H. Srinivasan. Analyzing Exception Usage in Large Java Applications. In *Workshop Exception Handling in Object Oriented Systems : Towards Emerging Application Areas and New Programming Paradigms*, pages 10–19. ECOOP 2003, july 2003. Darmstadt.

[SH00] S. Sinha and M.J Harrold. Analysis and Testing of Programs with Exceptions Handling Constructs. *IEEE Transactions on Software Engineering*, 26(9) :849–871, september 2000. Volume "Special Section on Current Trends in Exception Handling".

[SUVD03] F. Souchon, C. Urtado, S. Vauttier, and C. Dony. Exception Handling in Component-Based Systems : a First Study. In *Workshop Exception Handling in Object Oriented Systems : Towards Emerging Application Areas and New Programming Paradigms*, pages 84–91. ECOOP 2003, july 2003. Darmstadt.

[TKA05] A. Tripathi, D. Kulkarni, and T. Ahmed. Exception Handling Issues in Context Aware Collaboration Systems for Pervasive Computing. In *Workshop Exception Handling in Object Oriented Systems : Developing Systems that Handle Exceptions*, pages 113–124. ECOOP 2005, july 2005. Glasgow.

[Ven98] B. Venners. Designing with Exceptions. *Java World*, july 98. 7 pages in electronic publication `http://javaworld.com`.