MajecSTIC 2012
Lille, France, du 29 au 31 octobre 2012

Une seule manière d'utiliser les exceptions ? Une étude empirique de 21 applications Java

Maxence G. de Montauzan et Martin Monperrus

Université de Lille & INRIA, Cité scientifique, 59655 Villeneuve d'Ascq – France.

Contact : germain-de-montauzan@etudiant.univ-lille1.fr
martin.monperrus@univ-lille1.fr

Résumé
Le langage Java offre un puissant système de gestion des exceptions et de traitement des erreurs. L'étude de différentes applications Java peut permettre de distinguer différentes utilisations de ce système. Ce papier étudie les utilisations des exceptions pour constater les différences ou similitudes entre 21 applications. Les résultats permettent de constater de fortes différences entre les applications et dans l'ensemble une gestion très variée des exceptions.

Mots-clés : étude empirique, exception, génie logiciel

---

1. Introduction

Java, comme C++ ou Python, est un langage de programmation supportant les exceptions. Ce mécanisme a pour but de faciliter la gestion des erreurs et des cas limites. C'est un outil très puissant dont l'utilisation ne peut se résumer à quelques cas simples.
Dans cet article, nous faisons l'hypothèse que la richesse du modèle d'exception de Java se traduit en plusieurs manières de designer une gestion applicative des exceptions. L'objectif de cet article est de vérifier empiriquement que l'utilisation des exceptions en Java varie en fonction des domaines et des projets.
Notre protocole expérimental définit différentes métriques, appliquées sur 21 applications en Java, afin d'analyser différentes caractéristiques de l'utilisation des exceptions. Les résultats montrent qu'il y a des différences significatives quant à l'utilisation des exceptions entre les applications.
Ce papier présente tout d'abord l'état actuel de l'art, puis explique succinctement le mécanisme des exceptions en Java avant de présenter le processus expérimental utilisé dans cette étude. Nous étudions la distinction entre les librairies et les applications clientes de par l'utilisation des exceptions, pour continuer en étudiant le flot des types d'exceptions des applications. Pour finir, nous étudions une caractérisation du design de la gestion des exceptions.

2. État de l'art

De précédentes études ont été faites sur l'utilisation des exceptions dans des programmes Java.
Barbara G. Ryder et al. [7] présentent l'analyse produite par leur outil JESP sur une trentaine d'applications Java. Ils font l'étude du nombre d'exceptions déclenchées et attrapées par rapport à la taille des applications étudiées, mais ne considèrent pas la différence de nature entre application et librairie.
Darrell Reimer et Harini Srinivasan [5] font l'étude des différents modèles d'utilisations des exceptions et des problèmes que cela peut poser. Ils ne font pas l'étude de la fréquence d'apparition de ces comportements.
Rebecca J. Wirfs-brock [8] décrit différentes façons de mieux gérer les exceptions et le design d'un programme à ce sujet particulièrement. Mais elle n'étudie pas non plus la propension qu'ont les développeurs à appliquer ou non ces solutions dans les programmes.

3. Système d'exception de Java

Nous présentons brièvement le système de gestion d'exception proposé par Java. En Java, une exception est déclenchée quand un traitement, pour une raison ou une autre, n'arrive pas à s'exécuter complètement. Autrement dit, le code n'a pas pu s'exécuter et faire ce qu'il devait faire. Afin de mieux identifier la raison du problème, les exceptions ont un certain type. Par exemple, une `NullPointerException` indique que l'un des objets manipulés n'existe pas (est nul), une `IllegalArgumentException` indique que l'un des paramètres passés à la méthode n'est pas correct, etc. Ajoutons qu'il est possible de lancer manuellement une exception, à la suite d'une condition par exemple, avec le mot-clef `throw`. Lorsqu'une méthode peut lancer une exception, alors le type de l'exception lancé (ou un type plus générique, père de l'exception qui est lancé) doit être indiqué dans la signature de la méthode, c'est-à-dire au moment de sa déclaration.Sauf lorsque l'exception est dite « *Runtime* ». En effet, toutes exceptions héritant de `RuntimException` n'ont pas besoin d'être déclarées dans la signature de la méthode. Pour capturer une exception, on met le code qui risque de déclencher une exception dans un bloc `try`, qui sera suivi d'un bloc `catch`, dans lequel on indique le type d'exception qu'on veut attraper et qui va contenir le traitement spécifique que l'on veut voir appliqué. Il peut y avoir une succession de bloc `catch`, permettant d'attraper différents types d'exceptions. Le traitement spécifique qui peut être appliqué lors de l'arrivée d'une erreur peut permettre, entre autres, d'afficher un message d'erreur détaillée de l'exception (où a-t-elle été déclenchée, l'ensemble des méthodes qu'elle a traversées avant d'être attrapée, etc.). À la suite du ou des blocs `catch`, il peut y avoir un autre bloc, le bloc `finally`. Ce bloc à la particularité d'être exécuté quoiqu'il arrive, après la succession de blocs `catch`.

4. Processus expérimental

Le processus qui a été suivi pour analyser le code source Java est de transformer le code source Java en XML, pour pouvoir ensuite l'analyser statiquement.

4.1. Transformation du code Java en XML

Nous extrayons l'AST du code Java en XML avec le logiciel srcML[1] développé par Michael L. Collard et Jonathan I. Maletic [3]. La raison première de cette transformation est de pouvoir raisonner sur le code (analyse, requête) au moyen de requêtes *XQuery*. Les XML obtenus font entre 105000 nœuds et 4826000 nœuds pour des tailles allant de 1172 ko à 52893 ko. La page [1] donne des détails sur les XML produits. L'exemple 2 montre la transformation en XML d'un extrait de code source Java comportant un `try` et un `catch`.

EXEMPLE 1 – Un bloc Try/Catch simplifié du projet DNSJava renvoyant une exception

```
try {
    s.update(); }
catch (GeneralSecurityException e) {
    throw new DNSSECException(e.toString()); }
```

EXEMPLE 2 – Un bloc Try/Catch du projet DNSJava transformé en XML par srcML

```
<try>try <block>{
<expr_stmt><expr><call><name><name>s</name>.<name>update</name></name>
<argument_list>()</argument_list></call></expr>;</expr_stmt>
}</block>
<catch>catch (<param><decl><type><name>GeneralSecurityException</name>
</type> <name>e</name></decl></param>) <block>{
<throw>throw <expr>new <call><name>DNSSECException</name><argument_list
    >(<argument><expr><call><name><name>e</name>.<name>toString</name><
    /name><argument_list>()</argument_list></call></expr></argument>)
</argument_list></call></expr>;</throw>
}</block></catch></try>
```

1. http://www.sdml.info/projects/srcml/

4.2. Analyse du XML

L'analyse du XML a été effectuée à l'aide du langage de requête *XQuery*. Ce langage permet d'extraire des informations des différents XML et de construire de nouveaux documents XML à partir de ces informations, mais aussi d'effectuer des calculs sur les éléments des XML.

4.3. Corpus

Cette étude est faite sur 21 applications Open-Source Java : les 14 applications du CVS Vintage [4] ont été prises, ainsi que 7 applications du banc d'essai Dacapo [2]. Ces projets ont tous entre 5 et 10 ans et sont assez importants : *jUnit*, le plus petit projet étudié, fait 6826 lignes de code et *lucene*, le plus gros projet de cette étude fait quant à lui 326990 lignes de code. Cet ensemble d'applications comporte aussi bien des applications clientes (*ArgoUML*, *Columba*, *org.eclipse.ui.workbench* [2], *pmd*, *Scarab*, *sunflow* et *jEdit*) que des librairies (*batik*, *DNSJava*, *fop*, *jHotDraw*, *jUnit*, *log4j*, *lucene*, *org.eclipse.jdt.core* [3], *Struts*, *xalan*) en passant par des middlewares (*Carol*, *Tomcat*) et des applications serveur (*jBoss*, *h2*). Des détails sur ces applications sont disponibles sur [1].

5. Étude de la distinction entre librairies et applications clientes

Il existe une différence de nature entre les applications : les applications librairies et les applications clientes. Les premières offrent un panel d'outils, de traitements et de méthodes qui servent de fondation à un programme. Les secondes se basent sur ces librairies pour offrir un environnement répondant aux demandes des utilisateurs. Cette différence doit influencer les programmeurs sur leur façon d'utiliser les exceptions dans leurs applications.

5.1. Métriques

Nous avons extrait de notre ensemble d'applications (cf. section 4.3) celles qui sont des librairies ou des applications clientes, puis nous avons mesuré le nombre de fois où une exception quelconque est lancée et le nombre de fois où une exception quelconque est attrapée. Nous considérons ici qu'une exception est lancée dès l'utilisation du mot-clef `throw`. De même, on considère qu'une exception est attrapée dès que l'on trouve le mot-clef `catch`.

5.2. Hypothèse

Une étude précédente [7] constate que les applications sont séparées en deux catégories : celles dominées par les `throw`s et celles dominées par les `catch`s. Ils font alors l'hypothèse que les applications avec une majorité de `throw` étaient probablement des librairies, et qu'à l'inverse les applications avec une majorité de `catch`s étaient des applications client. De par leur nature, il semble en effet probable que les librairies soient amenées à lancer plus d'exceptions qu'à en attraper. À l'inverse, les applications clientes, qui utilisent des librairies, sont amenées à attraper ces exceptions lancées par les librairies, afin de procéder au(x) bon(s) traitement(s), plutôt que d'en lancer de nouvelles.

5.3. Résultats

La figure 1 présente, pour chaque application étudiée, le nombre de `catch` (sur l'axe x) sur le nombre de `throw` (sur l'axe y). Les points en rouge représentent les applications de type librairies, et en bleu les applications de type client. Le nombre d'exceptions attrapées varie d'environ 80 à 1500 et le nombre d'exceptions lancées atteint des valeurs maximums qui sont quasiment doubles (de 50 à 2800 environs). Une application se situant en haut à gauche envoie beaucoup d'exceptions, mais n'en attrape pas beaucoup, elle serait donc selon nos hypothèses une librairie. À l'inverse, une application en bas à droite de la figure attrape beaucoup d'exceptions, mais en envoie peu, elle serait une application cliente.

Voyons tout d'abord quelques cas précis avant de discuter de la validité de notre hypothèse. *Colomba* répond parfaitement à notre hypothèse et dans une moindre mesure, *DNSJava* y répond assez bien. Cependant, *pmd* qui est une application cliente et *log4j* qui est une librairie, sont à l'opposé de notre hypothèse. Pour finir, on peut constater que *xalan* ou encore *JDT* ne répondent absolument pas à l'hypothèse puisqu'ils comportent un nombre quasi égal de `throw` et de `catch`.

2. renommé dans ce papier « Workbench »
3. renommé dans ce papier « JDT »

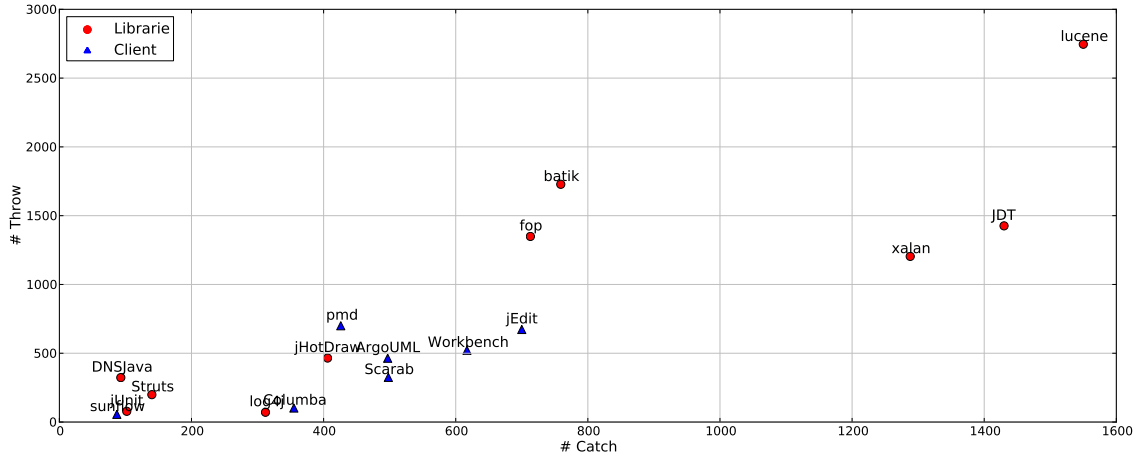

FIGURE 1 – Nombre d'exceptions lancées sur le nombre d'exceptions attrapées par projet.

Finalement, on constate clairement que les deux groupes d'applications attendus n'apparaissent pas. Nos résultats nous permettent de constater qu'à l'inverse de l'hypothèse faite, il est très difficile de faire une réelle distinction entre les logiciels librairies et les logiciels clients en utilisant le nombre de `catch` et le nombre de `throw`.

5.4. Discussion

Notre étude produit des résultats très différents de ceux de Barbara G. Ryder et al. [7]. Nous pouvons l'expliquer par le fait que l'étude [7] est faite sur une majorité de petits projets (moins de 1000 lignes de code pour plus d'un tiers des projets) qui étaient en cours d'élaboration. Il est en effet fort probable que des projets de quelques centaines de lignes n'utilisent que très peu les `throw`s, mais utilisent des `catch`s surtout par obligation, certaines méthodes proposées par le SDK de Java renvoyant des exceptions qu'il est nécessaire d'attraper.
Nos résultats nous montrent aussi des différences de taille entre les projets. À l'inverse des résultats de [7], il y a clairement une corrélation entre la taille du projet (voir figure 1) et le nombre d'exceptions lancées et attrapées. *lucene*, qui est le plus gros projet lance plus de 2500 exceptions et en attrape plus de 1500. *jUnit*, qui est notre plus petit projet, lance à peine une centaine d'exceptions et en attrape encore moins.

6. Étude du flot des types d'exceptions

Nous pouvons nous intéresser de plus près aux types des exceptions qui sont attrapées et lancées. Les différences de nature entre les librairies, les applications clientes et les middlewares (qui font communiquer des logiciels entre eux, aussi bien des librairies que des applications clientes) peuvent influencer les types d'exceptions utilisées. Les librairies ont sans doute besoin de lancer des types particuliers d'exceptions pour informer les développeurs qui les utilisent sur ce qu'il s'est passé, alors que les applications clientes ne lancent probablement qu'un minimum de type d'exceptions, puisqu'elles ne sont pas censées être utilisées par un développeur comme une librairie.

6.1. Métriques

Nous avons mesuré le nombre de types différents d'exceptions qui sont lancées (`throw`) mais jamais attrapées (`catch`). De même, nous avons ensuite mesuré le nombre de types d'exceptions qui sont attrapées mais jamais lancées. Ce, pour chaque projet clairement identifiable comme librairie, client ou middleware [4]. Si dans un des projets on ne trouve jamais de « `throw Throwable` »

4. Nous n'avons pas pris les middlewares dans la section précédente afin d'avoir des résultats clairs, les middlewares pouvaient faire du bruit en n'étant pas clairement des librairies ou des applications clientes.

par exemple, mais un ou plusieurs « `catch (Throwable e)` »[5], il s'agit d'un type d'exception attrapée mais jamais lancée. À l'inverse, si on trouve un ou plusieurs « `throw Throwable` » mais aucun « `catch (Throwable e)` », alors il s'agit d'un type d'exception lancée mais jamais attrapée.

6.2. Hypothèses

Nous pouvons supposer que des types d'exceptions attrapées mais jamais lancées indiqueraient que le logiciel est une application cliente, qui attrape un grand nombre de types d'exceptions, afin que les exceptions ne ressortent pas à l'utilisateur (hypothèse *H1*). L'inverse c'est-à-dire des types d'exceptions qui sont lancées mais jamais attrapées indiquerait que le logiciel est une librairie (hypothèse *H2*), qui renvoie ses propres types d'exceptions. Finalement, on peut supposer qu'une application qui attrape un grand nombre de types d'exceptions différents sans lancer ces mêmes types, et qui lance un grand nombre de types d'exceptions différents sans les attraper est une application de type middleware (hypothèse *H3*), c'est-à-dire un logiciel qui va aussi bien s'appuyer sur une librairie que servir de librairie. Ce type d'application a tout autant besoin de communiquer avec la librairie du bas (et d'attraper des exceptions) qu'avec les développeurs du dessus (et de lancer des exceptions).

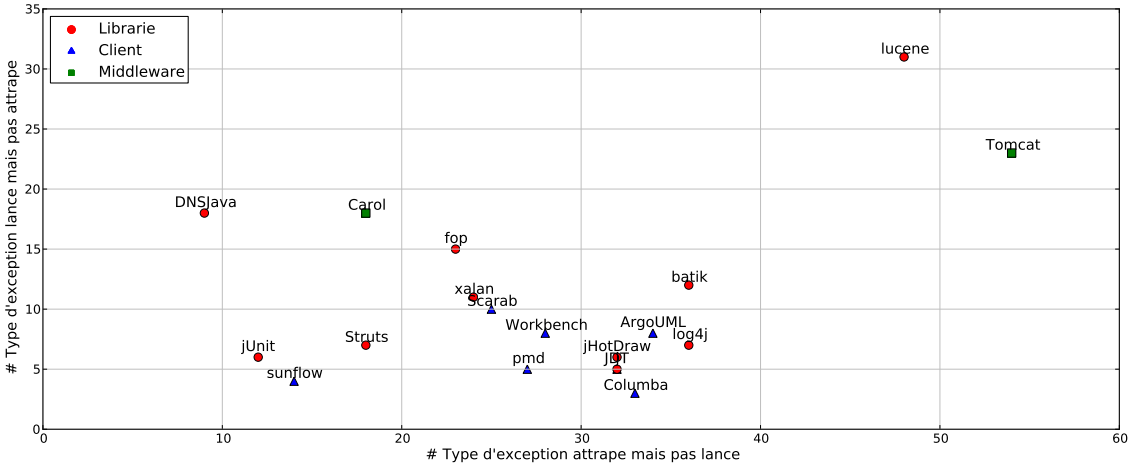

FIGURE 2 – Nombre de types d'exceptions lancées mais jamais attrapées sur le nombre de types d'exceptions attrapées mais jamais lancées.

6.3. Résultats

La figure 2 nous montre le nombre de types d'exceptions attrapées mais jamais lancées (sur l'axe x) et le nombre de types d'exceptions lancées mais jamais attrapées (sur l'axe y) et ce, par projet. De même que pour la figure 1, les points en rouge représentent les applications de type librairie et les points en bleu les applications de type client. Les points en vert représentent les applications de type « middleware », c'est-à-dire les applications entre la librairie et l'application cliente. Le nombre de types d'exceptions lancées mais jamais attrapées varie entre 3 et 31 alors que le nombre de types d'exceptions attrapées mais jamais lancées varie entre 9 et 54.

Pour l'ensemble d'applications étudié ici, toutes les applications sont amenées à attraper des types d'exceptions qui ne sont jamais lancées. Fait logique, puisque toutes les applications de cette étude utilisent au moins une librairie externe, la SDK de Java, qui envoie des types d'exceptions que le logiciel n'a pas lancées lui-même. Mais on note aussi que toutes les applications étudiées lancent des types d'exceptions qui ne sont jamais attrapées. Nous reviendrons sur ce point dans la section

5. Le *e* représente le nom de la variable qui va désigner l'exception dans le bloc de code qui suivra. Notre étude ne prend pas en compte le nom de cette variable, seul le type de l'exception nous importe.

suivante. *Columba* répond parfaitement à notre hypothèse *H1* en attrapant beaucoup de types d'exceptions qu'il ne lance jamais. *DNSJava* qui, à l'inverse, lance beaucoup de types d'exceptions et en attrape assez peu, répond à l'hypothèse *H2*. *Carol* répond à l'hypothèse *H3* en ayant un même nombre de types d'exceptions lancées et attrapées : c'est un middleware.
On peut tout de même constater qu'une majorité de librairies ont tendance à envoyer plus de types d'exceptions qu'elles n'attrapent pas, alors qu'une majorité des applications clientes ont tendance à attraper plus de types d'exceptions qu'elles ne lancent pas, ce qui valide nos hypothèses *H1* et *H2*.

6.4. Discussion

Chaque application de cette étude lance des types d'exception qui ne sont jamais attrapés. Pourtant, contrairement à l'hypothèse *H3*, ce ne sont pas tous des middlewares. Une explication probable est que ces applications utilisent des `catch`s de type d'exception hérité.
Concrètement, on suppose que les développeurs définissent un type d'exception `MyException`. Puis ils définissent deux types d'exception, `MySubException1` et `MySubException2` qui héritent de `MyException`. Par la suite, dans le projet, ils lancent des exceptions de type `MySubException1` ou `MySubException2` mais attrapent uniquement les exceptions de type `MyException`, ce qui inclue les deux types d'exceptions qui en héritent. Ce genre d'utilisation expliquerait que des programmes attrapent des types d'exceptions définis dans le projet mais jamais lancés. Le cas extrême consiste à attraper l'exception la plus générique (e.g. avec un `catch (Exception e)` en Java), fait évoqué dans une précédente étude [5]. C'est une explication à l'invalidation de notre hypothèse. Ceci dit, comme le montre notre étude, à la diversité des pratiques de design correspondent sûrement plusieurs autres raisons que nous n'avons pas entrevues.

7. Caractérisation du design de la gestion des exceptions

Les développeurs doivent probablement apporter un certain soin sur le « design » des exceptions dans les projets afin de permettre à l'application une capture efficace des exceptions et à rendre cette partie plus maintenable. Par design nous entendons la répartition et la dispersion des `catch`s dans le projet, ainsi que la propension des `catch`s à relancer des exceptions. Comme l'expliquent Derramm Reimer et Harini Srinivasan [5], une certaine utilisation des exceptions peut rendre inefficace la gestion des erreurs et rendre le code difficile à maintenir.

7.1. Métriques

Pour chaque projet, nous avons mesuré le nombre de méthodes comportant un ou plusieurs `catch`s, donc attrapant une ou plusieurs exceptions, que nous avons divisé par le nombre de méthodes (*R1*). Nous avons aussi mesuré le nombre de blocs `catch` qui comportent un ou plusieurs `throw`s, donc qui relancent une ou plusieurs exceptions, que nous avons divisé par le nombre de `catch`s (*R2*). Nous obtenons ainsi le ratio de méthodes comportant des `catch`s et le ratio des `catch`s comportant des `throw`s. Par construction, les valeurs de *R1* et de *R2* sont comprises entre 0 et 1.
Le premier ratio nous indique le taux de répartition des `catch`s dans le projet. Le deuxième nous indique le taux de `catch` renvoyant une exception. Il est courant de renvoyer une exception depuis un `catch` pour faire en sorte que l'exception soit traitée autre part, lorsqu'on ne sait pas la traiter directement. Ces mesures donnent une idée quant à la richesse du design d'exception des applications. En effet, le taux de répartition des `catch`s dans le projet peut indiquer l'attention qui a été mise à ne pas éparpiller la gestion des exceptions dans le projet. De même, le taux de `catch` renvoyant une exception peut indiquer la propension des développeurs à renvoyer une exception pour que celle-ci soit traitée autre part, rendant ainsi le traitement des exceptions très localisé dans le projet.

7.2. Hypothèses

Un *R2* élevé indique que beaucoup de `catch`s renvoient des exceptions, ce qui signifie que lorsqu'on ne sait pas traiter l'erreur, on renvoie une exception pour que l'erreur soit traitée autre part. Nous pouvons faire l'hypothèse *H4* que les middlewares et les applications serveur devraient avoir de fortes valeurs de *R2*, pour relancer des exceptions de la couche du dessous à celle du

dessus. Nous faisons aussi l'hypothèse *H5* que les librairies non fonctionnelles devraient avoir des valeurs de *R2* faibles, car elles doivent arrêter les erreurs au plus tôt, chaque `catch` devrait traiter l'exception. Par exemple, *log4j*, qui est une librairie de logging [6] ne devrait pas arrêter l'application en cas d'erreur, même si le logging se passe mal.
Lorsque *R1* est faible, il est facilement déductible que les `catch`s sont assez localisés dans l'application. Un *R1* faible permet de supposer que le design d'exception a été pensé pour traiter les erreurs à des endroits définis dans le projet. À l'inverse, un *R1* très élevé indique une forte dispersion des `catch`s dans l'application. Nous faisons donc l'hypothèse que *R1* est plutôt faible, compte tenu de la maturité des applications que nous avons étudiées (cf. section 4.3).

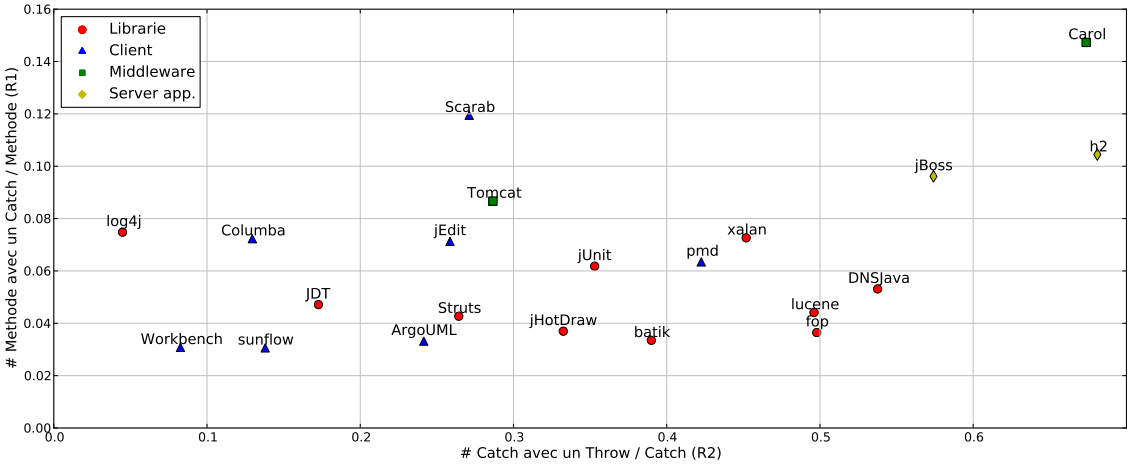

FIGURE 3 – Nombre de méthodes avec des catchs divisé par le nombre de méthodes *R1* sur le nombre de catchs avec des throws divisé par le nombre de catchs *R2*.

7.3. Résultats

Sur la figure 3, nous avons sur l'axe y le nombre de méthodes comportant un ou plusieurs `catch`s, divisé par le nombre total de méthodes, et sur l'axe x nous avons le nombre de `catch`s comportant un ou plusieurs `throw`s sur le nombre total de `catch`s. Alors que R2 varie grandement entre 0.04 et 0.7, R1 reste dans un intervalle beaucoup plus petit (entre 0.03 et 0.14).
Les logiciels *h2* et *Carol* répondent à l'hypothèse *H4* en ayant une très forte valeur de *R2*. Le logiciel *log4j* est une librairie de logging, le logging étant non fonctionnel, *log4j*, répond à l'hypothèse *H5*, en ayant une valeur de *R2* extrêmement faible. Cependant, il ne s'agit que d'un cas isolé de notre corpus d'applications, ce qui ne nous permet pas de valider l'hypothèse.

7.4. Discussion

Tout d'abord, il est important de noter la différence entre les deux ratios : les valeurs pour *R1* sont deux à trois fois plus faibles que les valeurs de *R2*. Autrement dit, cela laisse à supposer que les méthodes avec des `catch`s sont peu éparpillées dans les projets, mais qu'il y a une forte tendance à relancer les exceptions dans les `catch`s et donc à naviguer entre les différentes méthodes.
On peut aussi constater la large gamme de valeurs que prend *R2* : pour certaines applications, plus de la moitié des `catch`s renvoient des exceptions. Une explication souvent rencontrée est la volonté de respecter le contrat d'une méthode. Le contrat de la méthode, défini par les développeurs et explicité dans la documentation, permet d'être sûr de ce qu'elle fait et de ce qu'elle renvoie, facilitant son utilisation. Les contrats concernent aussi les exceptions.
Un développeur peut définir, pour diverses raisons qui ne sont pas le sujet ici, que, si une méthode n'arrivait pas à faire ce qu'elle doit faire, elle renverrait une exception d'un type précis, et pas

6. Le logging consiste à ajouter, dans une application, des traitements permettant de sauvegarder des messages suite à des événements.

autre chose. Par exemple, lorsqu'une méthode attend un objet pour faire un traitement dessus, si celui-ci est vide, et qu'aucun contrôle n'est fait, la méthode va déclencher une exception de type `NullPointerException` grâce à la SDK de Java. Mais le développeur ne veut pas que ce cas arrive, et veut absolument que la méthode renvoie un type défini d'exception. Pour ce faire, il va être obligé d'attraper les types les plus génériques d'exceptions possibles et de renvoyer à chaque fois le type d'exception voulu pour remplir le contrat. Il va s'en suivre une série de `catch` effectuant un traitement mais renvoyant à chaque fois une exception.
Quant au design, dans cette étude, peu d'applications ont des valeurs de *R1* faibles, soit le nombre de méthodes avec des `catch`s divisé par le nombre de méthodes total. Autrement dit, la dispersion des `catch`s dans les projets est assez forte. On peut supposer qu'il en résulte un traitement répétitif des exceptions, et donc une forte similitude du code, soit un design peu soigné. L'explication la plus probante est que, sur des projets open-source tels que ceux de cet ensemble d'application (cf. section 4.3), chaque développeur vient avec ces habitudes et traite les exceptions à sa manière. Le logiciel grandissant, le traitement des exceptions se retrouve éparpillé dans tout le projet. Nous n'étudierons pas dans ce papier le comportement des développeurs vis-à-vis des exceptions, mais nous pensons que cela pourrait être un point intéressant pour de futurs travaux.

8. Conclusion

En analysant statiquement le code source Java de 21 applications, nous avons pu constater qu'il n'y a pas une réelle catégorisation entre les librairies et les applications clientes de par leurs nombres de `catch` et de `throw`. Nous avons vu aussi que chaque application de cette étude lance des types d'exceptions qu'elles n'attrapent pas, de façon plus prononcée pour les librairies, et que toutes les applications attrapent des types d'exception qu'elles ne lancent pas, ceci entre autres à cause de la SDK de Java. Finalement, nous avons vu que les applications de cette étude ont une dispersion des `catch`s plus ou moins marquée, et qu'elles relancent toutes des exceptions après en avoir attrapé, quels que soient leurs types. Pour de futurs travaux, nous comptons nous intéresser plus en profondeur au flot d'exceptions dans des programmes Java à l'aide de l'outil Jex de Martin P. Robillard et de Gail C. Murphy [6] pour essayer de savoir où sont attrapées les exceptions et pourquoi.

Bibliographie

1. Companion web page. `http://goo.gl/YaUhN`.
2. The Dacapo benchmark suite. `http://dacapobench.org/`.
3. Michael L. Collard, Huzefa H. Kagdi, et Jonathan I. Maletic. An XML-Based Lightweight C++ Fact Extractor. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 134–143, 2003.
4. Martin Monperrus. Conservation and replication with cvs-vintage : A dataset of cvs repositories of java software. `http://www.monperrus.net/martin/cvs-vintage-dataset`.
5. Derramm Reimer et Harini Srinivasan. Analyzing Exception Usage in Large Java Applications. In *Exception Handling in Object Oriented Systems : Towards Emerging Application Areas and New Programming Paradigms*, pages 10–19, juillet 2003.
6. Martin P. Robillard et Gail C. Murphy. Analyzing Exception Flow in Java Programs. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 322–337, septembre 1999.
7. Barbara G. Ryder, Donald Smith, Ulrich Kremer, Michael Gordon, et Nirav Shah. A Static Study of Java Exception Using JESP. In *Proceedings of the Ninth Annual International Conference on Compiler Construction*, pages 67–81, mars 2000.
8. Rebecca H. Wirfs-Brock. Toward Exception-Handling Best Practices and Patterns. *IEEE software*, 23(5) :11–13, septembre 2006.

Nous remercions Michael L. Collard et Jonathan I. Maletic pour leur formidable travail sur leur logiciel srcML.