UNIVERSITÉ DU QUÉBEC

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

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR
SÉBASTIEN PERRON

DÉTECTION D'ERREURS ET CONFINEMENT LOGICIEL : UNE ÉVALUATION EMPIRIQUE

MARS 2021

REMERCIEMENTS

J'aimerais exprimer ma gratitude envers mes proches, envers ceux qui m'ont soutenu durant la réalisation de ce projet. Merci à mes amis et à ma famille pour leur amour, leur présence et leur dévouement. Merci aux correcteurs : Linda Badri, Francine Chicoine, Louis Houde et François Meunier. Merci à Maryse Côté pour sa bienveillance. Merci à ceux croisés sur la route pour le soutien dont ils ont su m'apporter. Finalement, toute ma reconnaissance à Mourad Badri pour sa guidance et son support continu.

RÉSUMÉ

La complexité d'un logiciel et la pluralité des dépendances intercomposantes, présentes à plusieurs niveaux du système, peuvent affecter sa fiabilité, en particulier sa capacité à tolérer les fautes. Dans un logiciel orienté objet, les interactions qui découlent de telles dépendances impliquent un couplage pouvant entraîner la propagation des erreurs. Ce phénomène peut être résolu par l'implantation de mécanismes de détection et de confinement d'erreurs, techniques couramment utilisées pour tolérer les fautes. Après avoir situé notre approche parmi les techniques de tolérance aux fautes existantes (soit les techniques de détection et de confinement), nous présentons trois solutions orientées objet où un composant est renforcé dans le but de détecter les erreurs qui peuvent y survenir et d'en limiter les répercussions sur d'autres composants. Ces solutions sont implantées dans un logiciel Java où leur mise en œuvre s'appuie sur une conception et sur une méthode de validation basées sur les spécifications.

ABSTRACT

The complexity of software and the plurality of intercomponent dependencies, found in many layers of a system, can affect its reliability, in particular its ability to tolerate faults. In object-oriented softwares, the interactions derived from such dependencies imply a coupling which can lead to propagation of errors. This problem can be resolved by implementing error detection and error containment mechanisms. Such techniques are used to tolerate faults. After having localized our approach among existing fault tolerance techniques (i.e. detection and containment techniques), we present three object-oriented solutions where a component is reinforced in order to detect errors that occur on it and in order to limit the impacts on other components. These solutions are deployed in Java where their implementation and validation are based on software specifications.

TABLE DES MATIÈRES

LISTE DES FIGURES ET ILLUSTRATIONS

Un logiciel est constitué d'unités dont le comportement est défini selon les entrées et les sorties qu'il produit (logique déterministe). Un système peut être décomposé en sous-systèmes nommés « composants ». Les sorties d'un composant sont conséquentes aux entrées reçues et à sa logique interne. Une défaillance se produit lorsque l'environnement du système observe une sortie du système qui n'est pas conforme aux spécifications du logiciel (Laprie, 1995). Une erreur trouve généralement son foyer d'origine dans une faute latente située dans un composant. L'apparition de telles erreurs favorise l'apparition de nouvelles erreurs sur ce composant ou dans tout autre composant du logiciel, pouvant conduire à une défaillance observable par l'utilisateur. À mesure qu'un système fortement couplé augmente en complexité, les possibilités de défaillances augmentent également (Perrow, 1984).

1.1 Problématique

Une défaillance empêche un logiciel de fournir des services conformément à ses spécifications (Avizienis, Laprie, Randell et Landwehr, 2004). Elle entrave le bon fonctionnement d'un système et nuit ainsi à l'expérience utilisateur. La tolérance aux fautes est associée à la fiabilité logicielle, elle-même largement dépendante de la distribution des erreurs et de leurs conséquences sur l'utilisation de ces systèmes (Horner et Symons, 2019). La capacité d'un système à continuer à exécuter ses fonctions normalement en présence de fautes se nomme « tolérance aux fautes » (Johnson, 1984, cité dans Dubrova, 2013). Un système tolérant aux fautes fournit un service ininterrompu et sécuritaire en présence de fautes (Spitzer, Ferrell et Ferrell, 2015). De là la nécessité de détecter les fautes avant qu'elles ne soient la cause d'une défaillance, autrement dit, avant l'apparition de symptômes.

La tolérance aux fautes a longtemps été exclusivement déployée dans les systèmes de haut niveau : avions, centrales nucléaires, véhicules spatiaux. On justifie cela aisément puisque la conception de tels systèmes doit prendre en compte toutes les fautes pouvant y advenir (NASA, 2012). Les fautes ne sont toutefois pas exclusives à ces systèmes complexes, elles adviennent dans tout type de logiciel. Dans certaines situations, un logiciel est en dépendance directe vis-à-vis de certains services externes. Ce type de dépendances n'est pas nouveau, les produits commerciaux *off-the-shelf* (COTS) en sont un exemple. Les possibilités de défaillance dans les logiciels implémentant des COTS s'en retrouvent plus élevées puisque les résultats découlent de résultats produits par une source externe, un composant dont le niveau de fiabilité peut nous être inconnu. De récentes inquiétudes quant à l'utilisation de librairies d'apprentissage profond connues témoignent de cette problématique (Xiao, Li, Zhang et Xu, 2018). La tolérance aux fautes nous paraît non seulement nécessaire dans les systèmes et applications de haut niveau, mais également dans tout logiciel destiné à un usage

grand public. La fiabilité des logiciels a d'autant plus d'incidence sur notre compréhension du statut éthique et politique des technologies.

Une telle situation peut être corrigée par l'implantation d'une solution locale, soit une solution appliquée sur certains composants ciblés d'un logiciel. L'implantation de techniques de tolérance aux fautes vise ici à renforcer un composant en limitant la propagation d'erreurs d'un composant à un autre, soit de l'intérieur vers l'extérieur et vice versa.

1.2 Approche

Nous tentons de résoudre la problématique de propagation d'erreurs en proposant trois solutions de tolérance aux fautes, validées à l'aide d'une évaluation empirique. L'évaluation est composée de plusieurs expérimentations qui ont pour but de valider notre hypothèse. L'objectif de l'étude est triple : fournir une méthodologie, un cadre d'implémentation et des solutions pour améliorer la robustesse et la fiabilité des logiciels. La méthodologie utilisée vise à traduire les spécifications en code de détection d'erreurs ainsi qu'à supporter la vérification des propriétés à l'exécution, et ce pour un composant logiciel particulier. Les résultats de l'évaluation tenteront de démontrer la pertinence du problème des fautes dans les logiciels destinés. Les parties empiriques et conceptuelles de ce document se limitent aux erreurs dans les logiciels orientés objet.

1.3 Organisation du document

Le document est organisé en six chapitres, le premier ayant brièvement introduit le sujet. Un état de l'art est donné au second chapitre, dressant ainsi un portrait des solutions développées pour répondre à la problématique, présentant leurs limitations inhérentes ainsi qu'un moyen de parvenir à leur résolution. Une présentation des notions essentielles de tolérance aux fautes est fournie au chapitre 3, définissant le phénomène de propagation et de défaillance logicielle. Ce dernier circonscrit la portée du problème à résoudre et conduit, au chapitre suivant, à l'introduction des formalités relatives aux mécanismes étudiés. Les déploiements possibles y sont étudiés dans leur intégralité. Parmi ces déploiements, trois mécanismes de confinement sont sélectionnés pour constituer les solutions du chapitre 5. Le chapitre 6 présente une évaluation des solutions à partir d'une étude de cas concrète, suivie d'une discussion.

CHAPITRE 2 PROPAGATION LOGICIELLE : NÉCESSITÉ D'UNE PROTECTION. UN ÉTAT DE L'ART.

Les mécanismes de protection visent à améliorer la robustesse d'un logiciel en augmentant sa résilience vis-à-vis des fautes. La résilience logicielle ou simplement *résilience* signifie « la capacité d'un système [logiciel] à fournir ses services de manière consistante et fiable, en particulier face aux changements, aux défaillances et aux intrusions (Autili, Di Salle, Gallo, Perucci et Tivoli, 2015 ; Laprie, 2008) ». Un mécanisme de tolérance aux fautes est un ensemble de fonctionnalités mises en place pour gérer les fautes. Lorsqu'une erreur se produit sur l'un des composants d'un système, le dispositif de tolérance aux fautes permet de conserver une certaine stabilité, soit d'éviter d'interrompre les activités du système avec pour conséquence de dégrader le service affecté. L'opération est accomplie en deux étapes : d'abord en détectant l'erreur puis en la confinant. Alors que la première permet de déceler la présence d'erreurs, la seconde, définie comme « l'exclusion logique des composants défectueux à participer à la prestation d'un service » (Avizienis, Laprie, Randell et Landwehr, 2004), empêche que l'erreur ne se propage. L'amalgame de ces deux processus limite la propagation d'une erreur vers d'autres composants et permet ainsi d'augmenter le niveau de robustesse, de disponibilité et de fiabilité d'un logiciel.

2.1 Solutions existantes

La tolérance aux fautes logicielles est couramment développée suivant les règles de la diversité de conception (Avizienis et Kelly, 1984). La diversité de conception prend fondement dans la notion de redondance. Les approches basées sur la redondance sont variées (Lee et Anderson, 1990 ; Dubrova, 2013, chapitre 7). Quoiqu'elles aient largement été utilisées, les solutions basées sur cette approche demeurent limitées. De fait, la redondance implique une augmentation du nombre de composants dans un logiciel et un plus grand nombre d'interactions inattendues ; la redondance rend un système difficile à comprendre et à vérifier (Downer, 2009). Elle est d'autant plus coûteuse qu'elle augmente la complexité d'un système (Anderson, Barrett, Halliwell et Moulding, 1985 ; Carzaniga, Gorla et Pezzè, 2009). À titre d'exemple, la programmation en *N*-version à elle seule implique un plus grand nombre d'interactions, une augmentation de l'utilisation mémoire et une augmentation du temps d'exécution et de la synchronisation (Pullum, 2001).

2.1.1 Solution pour détecter les erreurs

Les tests de validité des données sont tous déployés sous forme d'assertions exécutables (Mahmood, Andrews et McCluskey, 1984). Cette méthode a été utilisée à maintes reprises sous forme de vérification à l'exécution (Salles, Rodriguez, Fabre et Arlat, 1999; Hiller, 2000; Popov, Riddle, Romanovsky et Strigini, 2001; Rodríguez, Fabre et Arlat, 2002). Les assertions exécutables peuvent également être spécifiées par la programmation par contrat (paradigme où les traitements sont régis par des assertions; voir Meyer, 1992), encore une fois proposées de nombreuses fois (Leveson, Cha, Knight, et Shimeall, 1990; Avizienis, 1997; Popov, Riddle, Romanovsky et Strigini, 2001; Anderson, Feng, Riddle et Romanovsky, 2003; Edwards, Sitaraman, Weide et Hollingsworth, 2004; Afonso, Silva, Brito, Montenegro et Tavares, 2008).

2.1.2 Solution pour confiner les erreurs

Le partitionnement de système est l'une des méthodes déployées globalement comme solution pour tolérer les fautes (Rosenblum et collab., 1996; Avizienis, 1997; Kopetz, 1997). Les composants d'un système peuvent être groupés en zones d'isolation des erreurs où de nouveaux services peuvent être implémentés (Rosenblum et collab., 1996). Un design pattern a été développé pour offrir un modèle de protection où un système isole l'erreur en la redirigeant vers une unité d'atténuation (Hanmer, 2007). Le flux d'erreurs est ainsi empêché de circuler entre les composants, et ce à l'aide d'une structure de confinement.

L'encapsulation est une méthode fréquemment utilisée pour confiner les erreurs (Venema, 1992; Voas, 1998; White, 2000; Anderson, Feng, Riddle et Romanovsky, 2003). Contenu à l'intérieur d'un autre composant, le composant est modifié en «l'enveloppant». De fait, il ne devient accessible que par l'interface du wrapper, limitant ainsi son comportement. Pareillement, un design pattern a été conçu pour transformer un composant en un wrapper de protection (Saridakis, 2003). Le modèle proposé est complémentaire au pattern Adaptateur (Gamma, Helm, Johnson et Vlissides, 1994). Le pattern Décorateur (Gamma, Helm, Johnson et Vlissides, 1994) est également proposé comme solution où de nouvelles fonctionnalités de tolérance aux fautes sont greffées sur le composant (Edwards, Sitaraman, Weide et Hollingsworth, 2004).

L'interception est un autre processus de tolérance aux fautes où les interactions sur un composant sont observées et contrôlées à l'aide d'un autre composant. Ils permettent d'implanter de nouvelles fonctionnalités au composant de manière transparente et sans

altération du code source. La solution a d'abord été concrétisée sous forme de réflecteur, où un système peut ainsi raisonner et manipuler une représentation de son propre comportement (Maes, 1987; Fabre, Nicomette, Perennou, Stroud et Zhixue, 1995; Fraser, Badger et Feldman, 1999; Salles, Rodriguez, Fabre et Arlat, 1999; Popov, Riddle, Romanovsky et Strigini, 2001). La propriété réflexive permet d'intercepter des opérations telles que la création d'objets, l'accès aux attributs et les invocations de méthodes, permettant à la fois de détecter les erreurs et de les confiner. La réflexion a ensuite laissé place à l'« aspectisation » de la tolérance aux fautes, c'est-à-dire l'utilisation de la programmation orientée aspect (Shah et Hill, 2003).

2.2 Limitations

Ces approches, pour la plupart, ont été développées dans l'objectif de fournir une sûreté de fonctionnement aux composants COTS, mais également aux micronoyaux COTS, aux systèmes en temps réel (Fabre, Salles, Moreno et Arlat, 1999; Salles, Rodriguez, Fabre et Arlat, 1999; Arlat, Fabre, Rodríguez et Salles, 2002; Rodríguez, Fabre et Arlat, 2002) et aux noyaux de systèmes d'exploitation (Fraser, Badger et Feldman, 1999). Pour la plupart, il s'agit de solutions développées pour une situation particulière. Il nous semble difficile de généraliser, voire d'abstraire ces solutions en des solutions généralisables sans d'abord devoir les réévaluer sous un autre contexte : dans un logiciel orienté objet, par exemple. Par conséquent, nous avons tenté de concevoir des solutions qui peuvent être implantées dans les logiciels du paradigme prévalant.

Habituellement, l'implantation de tolérance aux fautes nécessite la modification du code source (ajout de fonctionnalités, de classes, de variables, modification d'instances d'objet). L'implantation d'une solution peut être envisagée de façon à minimiser les impacts sur le code, c'est-à-dire en n'effectuant qu'un minimum de modifications afin d'éviter l'apparition de nouvelles dépendances. L'opération ne devrait ni être invasive ni modifier le comportement du logiciel. L'externalisation des solutions contribue à une diminution de l'apparition de nouvelles dépendances intercomposantes. Dans ce document, nous avons tenté de proposer des solutions dont l'implantation ne nécessite qu'un minimum de modifications du code source existant. Les solutions sont basées sur l'encapsulation et sur l'interception de composant. Trois solutions sont soumises à l'évaluation. Évaluées conjointement, leurs différences et ressemblances sont mises au jour. Une solution peut alors être envisagée pour doter un logiciel d'une robustesse supplémentaire.

Le dispositif proposé comme solution potentielle est schématisé à la figure 2.1. Une mince couche intercepte et filtre les interactions entre un composant et son environnement: les entrées et les sorties. Cette couche se compose d'un mécanisme de détection d'erreurs et d'un mécanisme permettant leur confinement. Trois configurations possibles sont détaillées au chapitre 5 et implantées par la suite dans un logiciel orienté objet pour être évaluées. Bien que Java soit utilisé, tout autre langage orienté objet aurait pu convenir. Java est utilisé comme représentant du paradigme orienté objet.

Figure 2.1 Structure du dispositif de protection

Les entrées correspondent à des données passées en paramètres de méthode; les sorties sont produites selon la logique interne de la méthode.

Un dispositif de tolérance aux fautes se présente d'abord sous forme d'unité de confinement d'erreurs. Le terme même d'« unité » ne peut être tenu pour acquis vu la pluralité d'appellations utilisées pour le désigner : *wrapper* (Venema, 1992 ; Voas, 1998), *opaque wrapper* (Black et Singh, 2019), *sandbox* (Gama, 2011), *protector* (Popov, Riddle, Romanovsky et Strigini, 2001), *error containment barrier* (Hanmer, 2007) ou *error-containment region* (Kopetz, 1997). Ces termes sont employés pour signifier la même notion, soit celle d'une couche interceptrice. Si *wrapper* a maintes fois été utilisé depuis son introduction il y a une trentaine d'années (Venema, 1992), c'est qu'il désigne aussi le nom d'un design pattern proposé dans l'ouvrage *Design Patterns* (Gamma, Helm, Johnson et Vlissides, 1994).

En élaborant une structure composée du composant à protéger et deux mécanismes de tolérance aux fautes, un tel dispositif agit comme une barrière entre le composant et son environnement. Le dispositif n'est toutefois pas toujours conçu comme une couche englobante ou comme une enveloppe. Une distinction entre la technique (l'encapsulation à titre d'exemple) et sa concrétisation (l'unité) est nécessaire. Puisque le mécanisme de confinement correspond à la forme même du dispositif et que ce mécanisme peut être implémenté de différentes manières, le terme « solution » est utilisé comme terme représentant du processus de tolérance aux fautes à l'œuvre, mais différentes de leur implantation (envelopper, englober, etc.).

Une distinction fondamentale peut être posée entre la cause d'une erreur et le phénomène lié à sa manifestation : la faute et l'absence de service (défaillance). On propose dans cette section de définir la singularité de la faute et de ses phénomènes. Les fautes sont ensuite catégorisées dans le but de cerner celles sur lesquelles les solutions agissent. Comme les fautes sont la synthèse d'un événement (cause, origine) et d'un environnement spécifique, les dépendances et interactions intercomposantes sont à prendre en considération. Nous expliquons en quoi l'architecture des logiciels, leur structure plus généralement, facilite la propagation des erreurs. Finalement, les deux mécanismes de tolérance aux fautes implantés dans les solutions sont expliqués : la détection et le confinement d'erreurs.

3.1 Pathologie des fautes

Von Neumann et Turing étaient bien au fait que des erreurs survenaient inévitablement dans les systèmes informatiques. Turing avait déjà posé une distinction entre les types d'erreurs : « Les erreurs de fonctionnement sont dues à une défaillance mécanique ou électrique

qui fait que la machine se comporte différemment de ce pour quoi elle a été conçue. [...] Par définition [les machines] sont incapables d'erreurs de fonctionnement. En ce sens, on peut vraiment dire que "les machines ne peuvent jamais commettre d'erreur" » (Turing, 1950, p. 449). L'auteur continue en évoquant la possibilité de l'apparition d'autres types d'erreurs : « Les erreurs de conclusion ne peuvent survenir que si un sens est associé aux signaux de sortie de la machine » (Turing, 1950, p. 449).

À la manière des machines, les logiciels ne s'exécutent pas toujours comme prévu. Les fautes ont tendance à y demeurer malgré les efforts visant à éviter leur introduction lors de leur conception, de leur implémentation ou lors de leur développement. Ni les tests ni même le débogage ne permettent de détecter toutes les fautes dans leur conception, ainsi la formule de Dijkstra : « Tester un programme démontre la présence de bogues, et non leur absence. » On rencontrerait pas moins de 3,3 erreurs en moyenne par 1000 lignes de code non commentées, et ce, même dans les logiciels de qualité (Myers, 1986).

En raison de leur nature immatérielle et mathématique, les logiciels ne peuvent ni se dégrader avec le temps ni dysfonctionner, mais peuvent « mal fonctionner (*malfunction*) » (Floridi, Fresco et Primiero, 2015). « Une faute dans la conception d'un ordinateur, par exemple, ne devrait pas être classée comme un dysfonctionnement opérationnel, mais elle compte quand même pour une erreur de calcul » (Fresco et Primiero, 2013, p. 254). Une faute correspond à une erreur dans un composant ou à une erreur dans la conception d'un logiciel (Lee et Anderson, 1990). Les défaillances sont causées par des fautes de conception (Randell, 1975; Lyu, 1996). Comme elles sont liées à des facteurs humains, elles sont imprévisibles donc difficiles à prévenir puisque leur manifestation est inattendue.

Une faute de composant (nommée « erreur de fonctionnement » par Turing) est une erreur dans l'état interne d'un composant. Au contraire, une faute de conception (« erreurs de conclusion ») est une erreur dans la conception d'un programme, celui-ci déviant des spécifications fonctionnelles. Ces dernières nécessitent l'application de mécanismes de tolérance aux fautes (Spitzer, Ferrell et Ferrell, 2015) et en particulier des méthodes plus efficaces que celles utilisées pour faire face aux fautes de composant (Lee et Anderson, 1990).

Les erreurs qui se manifestent dans un logiciel « ne peuvent survenir que si une signification est attachée aux [...] sortie[s] (Turing, 1950, p. 449) ». Ainsi, une erreur de calcul advient lorsque m calcule une fonction f qui doit, selon une entrée i, donner o_1 et que m donne $o_2 \neq o_1$ pour la même entrée (Piccinini, 2007). Selon leur rapport de finalité, les logiciels sont censés s'exécuter de la manière dont ils ont été conçus. Leur utilisation repose sur un principe d'utilité : ils sont utilisés pour exécuter certaines tâches avec précision. On dira d'un

logiciel censé effectuer *telle* tâche qu'il « fonctionne normalement » si le déroulement de ses opérations est conforme aux spécifications fonctionnelles. « Une spécification définit les problèmes à résoudre, l'ensemble des valeurs d'entrée/sorties permises (pré- et post-conditions du système) » (Fresco et Primiero, 2013, p. 260). Les spécifications désignent les sorties censées être dérivables des entrées lors du bon fonctionnement du système. Le programme spécifie une séquence d'instructions au moyen desquelles ces sorties sont dérivées des entrées (Fetzer, 1999). Les spécifications décrivent « les services qui doivent être fournis en matière de fonctionnalité et de performance (Avizienis, 1997) », elles donnent sens aux résultats obtenus.

Une donnée passée en entrée sur une méthode d'un composant peut causer l'activation d'une faute. Les approches proposées dans ce document visent à détecter ce type de fautes. C'est donc la conformité des paramètres de méthode, des messages et de la consistance de l'information contenue dans les structures de données qui sont vérifiées en rapport aux spécifications du logiciel.

L'objectif est « de minimiser la probabilité de défaillances » (Dubrova, 2013), qu'importe la gravité des conséquences possibles. Une faute est « la cause d'une erreur et une erreur est la cause d'une défaillance » (Lee et Anderson, 1990). Une faute engendre une erreur, engendrant à son tour une défaillance qui active une faute et ainsi de suite. Une défaillance est l'événement où le comportement du système n'est plus conforme à ses spécifications. La défaillance de service survient lorsque le système n'offre plus ses services de la manière spécifiée ; son état externe ne concorde plus aux spécifications. Plus un système est complexe, plus il est difficile de déterminer son exactitude (Fresco et Primiero, 2013, p. 258). La nature des fautes doit être connue pour constituer un mécanisme de détection efficace. Toutefois, il est plutôt coûteux de concevoir un système capable de tolérer tout type de fautes (Spitzer, Ferrell et Ferrell, 2015).

3.2 Classification des fautes

La classification des fautes a été entreprise par différents auteurs (Mariani, 2003 ; Avizienis, Laprie, Randell et Landwehr, 2004 ; IEEE, 2010). L'IEEE propose une classification particulièrement intéressante. De manière générale, cinq catégories sont retenues (Jiang, Zhang, Raymer et Strassner, 2007) : les fautes d'interface et de paramètres ; les fautes sémantiques, où le comportement du système est non consistant (une valeur incorrecte est observée) ; les fautes de service, principalement des fautes ayant un impact sur la qualité du service fourni ; les fautes de communication ou d'interaction (temps d'arrêt et d'indisponibilité du service fourni) ; les exceptions, étant des erreurs liées à la communication des entrées/sorties et à la sécurité. Notre

attention se porte sur les fautes d'interface et de paramètres ainsi que sur les fautes sémantiques. Ces dernières consistent « en une violation des suppositions explicites ou implicites sur le comportement du composant » (Mariani, 2003, p. 4). Les entrées et les résultats en sortie d'un composant seront ainsi soumis à une vérification. Sont regroupées sous le titre de « faute sémantique » : les fautes de comportement, de paramètres, de génération d'événements et de protocole d'interaction. La méthode employée pour tolérer les fautes diffère selon la nature des fautes à cibler. La faute est caractérisée sous différents aspects à la figure 3.1.

Une faute possède trois foyers d'origine : la cause du phénomène (physique ou humaine) ; le site d'observation ; le moment du cycle de vie d'un logiciel où elle est observée. Les fautes sont dites « internes aux limites du système » lorsqu'elles font partie de l'état du système, a contrario « externes » lorsque causées par une interférence ou un bris matériel dans l'environnement physique du système. Leur introduction lors de la phase de conception du système résulte d'imperfections lors du développement (de l'écriture des spécifications jusqu'au déploiement). Les fautes peuvent aussi apparaître lors de la phase de maintenance. Une faute est opérationnelle lorsqu'elle apparaît durant l'opération du système (Lala et Harper, 1994).

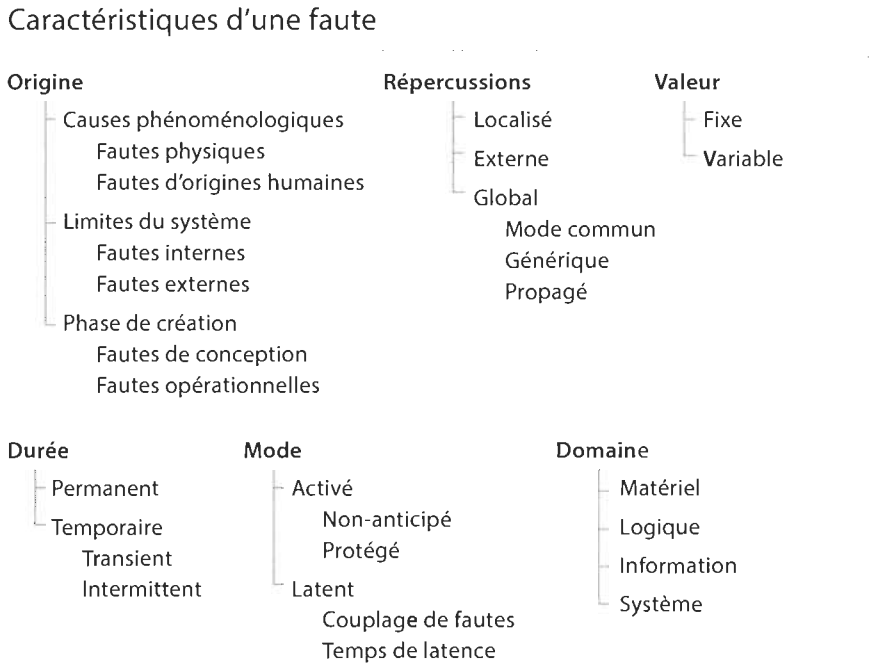

Figure 3.1 Caractéristiques d'une faute (d'après Spitzer, Ferrell et Ferrell, 2015)

Les répercussions d'une faute sont caractérisées en fonction des erreurs qu'elles génèrent, à savoir si l'erreur est localisée ou distribuée.

La valeur indique si la faute génère une valeur erronée fixe ou variable dans le temps ; la durée concerne l'espace de temps où la faute est observée, transitoire ou permanente. Les fautes transitoires occupent une majeure partie des erreurs détectées (Sosnowski, 1994, cité dans Spitzer, Ferrell et Ferrell, 2015). Une faute est « activée » lorsqu'elle cause la production d'un résultat déviant des spécifications. Au contraire, une faute qui ne cause pas un tel état est décrite comme « latente ».

3.3 Couplage, architecture et propagation

Nous concevons des systèmes dans lesquels les interactions entre les composants ne peuvent être ni planifiées, comprises, anticipées ou même protégées (Leveson, 2004). Le but de cette étude est d'agir sur les fautes et de limiter ainsi la propagation causée par les erreurs qu'elles engendrent. Non seulement une faute peut n'être située que sur un seul composant, mais elle peut également être propagée. La propagation se produit via l'interaction de composants entre eux (Avizienis, Laprie, Randell et Landwehr, 2004).

Les métriques de couplage ont un impact significatif sur la prédiction des fautes (Aggarwal, Singh, Kaur et Malhotra, 2009 ; Catal et Diri, 2007 ; Gyimothy, Ferenc et Siket, 2005 ; Olague, Etzkorn, Gholston et Quattlebaum, 2007 ; Pai et Bechta Dugan, 2007 ; Singh, Kaur et Malhotra, 2010 ; Zhou et Leung, 2006 ; Anwer, Adbellatif, Alshayeb et Anjum, 2017). Celles-ci mesurent la prédisposition d'une classe à contenir des fautes. La manière et le degré d'interdépendance entre les modules d'un logiciel révèlent la prédisposition d'une classe aux fautes. Plus les connexions sont nombreuses sur une classe, plus elle sera susceptible de contenir des fautes. Le couplage est également un facilitateur de la propagation des erreurs.

L'architecture d'un logiciel est composée d'éléments, d'une forme et de motivations (Perry et Wolf, 1992). Parmi les premiers se retrouvent : les éléments de traitement, les données, les éléments connecteurs. Les éléments de traitement sont ces modules (les composants) sur lesquels ont lieu des processus de traitement et de communication. L'environnement d'un composant est constitué de composants semblables ou distincts, directement liés par l'entremise d'éléments connecteurs. D'où l'expression de Beck et Cunningham : « aucun objet n'est un îlot. (1989, p. 2) »

Dans un système se déploie un ensemble de décisions qui peuvent assurer la réalisation des fonctionnalités système (Bass, Clements et Kazman, 2002). Les décisions architecturales ont des répercussions sur la manière dont le système peut être rendu fiable. Inversement, « les décisions relatives à la mise en œuvre des fonctionnalités de tolérance aux fautes dans un système peuvent avoir un impact sur son architecture, voire la façonner » (Harrison et Avgeriou, 2008).

Les décisions relatives aux attributs de qualité, dites « tactiques », comprennent les tactiques de disponibilité où se retrouve la tolérance aux fautes. Des tactiques sont attribuées aux logiciels à toute phase du cycle de vie d'un logiciel. L'intégration de tolérance aux fautes dépend du moment où celle-ci est prise en considération, mais également des moyens techniques. Dans certaines situations, la tolérance aux fautes est déployée globalement (dans les systèmes aéronautiques, voir NASA, 2012). Quoique cette globalité assure un niveau de fiabilité et de robustesse élevées, l'implantation reste difficile et coûteuse (Anderson, Barrett, Halliwell et Moulding, 1985).

Les composants sont l'un des éléments d'un système logiciel. De façon générale, un composant est une unité de logiciel exécutant certaines fonctions lors de l'exécution du logiciel (Shaw et Clements, 1997). Celles-ci sont conçues et développées suivant les spécifications fonctionnelles. Le recours aux spécifications permet de vérifier si le comportement du composant et de ses méthodes est acceptable ou non. Puisqu'elles sont consistantes et exhaustives, les spécifications peuvent être utilisées comme valeur de test valides. Un composant est « défini par son interface et par les services qu'il fournit aux autres composants, plutôt que par son implémentation interne. (Fielding, 2000) » Son état interne est partagé parmi ses méthodes, lesquelles sont rendues disponibles ou non à l'interface. L'interface spécifie les services fournis par le composant : messages, opérations et variables. La relation entretenue entre deux interfaces de composants est un élément connecteur, élément pouvant faciliter la propagation des erreurs. Von Neumann avait bien vu cela dans la défaillance de composants : « Dans un réseau complexe, avec de longues chaînes stimulus-réponse, la probabilité d'erreur dans les organes de base rend la réponse des sorties finales peu fiable, c'est-à-dire non pertinente, à moins qu'un mécanisme de contrôle empêche l'accumulation de ces erreurs fondamentales » (1956, p. 346).

3.4 Propagation des fautes

Une faute s'active lorsqu'elle provoque une erreur et qu'elle est disséminée. L'erreur se multiple de causes à effets. La figure 3.2 schématise ce phénomène. Les traitements effectués dans un composant sont souvent dépendants de composants voisins ; une erreur dans les résultats de l'un affecte les traitements subséquents. « Négliger la manière dont les actions d'un nœud affectent d'autres nœuds paralyse facilement des segments entiers des réseaux » (Barabási, 2002, p. 212). Une faute propagée qui n'est pas gérée peut provoquer un état d'erreur, la levée d'une exception ou encore une défaillance du logiciel, lorsque l'exception ne peut être gérée. Cette chaîne d'événements de causes à effets a été observée dans la majorité des accidents fatals impliquant

un ordinateur, où un enchaînement de défaillances inattendues est survenu (NAS, 1980, pp. 40–43). D'ailleurs, ces accidents résultent d'interactions dysfonctionnelles entre les composants et non d'une défaillance de composants individuels (Leveson, 2004, p. 566). Toutes les fautes ou les erreurs n'engendrent pas systématiquement de défaillances.

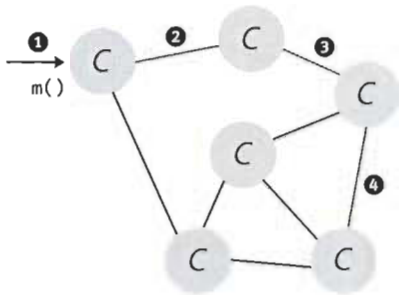

Figure 3.2 Séquence de propagation d'une erreur dans les composants d'un système

Une méthode *m* est appelée. Puisqu'elle contient une faute, l'appel engendre l'activation de la faute et, de là, génère une erreur se propageant d'un composant à un autre.

Lors de l'évolution d'un logiciel, on observe un accroissement du nombre de dépendances intercomposantes, contribuant ainsi à sa complexification. Une dégradation dans la qualité du code peut alors être observée, mais également une dégradation de la fiabilité du logiciel, justifiée par l'apparition de nouvelles fautes dues à l'ajout/modification de fonctionnalités. La simple présence de fautes dans un système en temps réel critique suffit à elle seule pour s'en inquiéter. Les événements sont connus : un bogue dans le code de la machine de radiothérapie Therac-25 a été responsable du décès d'êtres humains entre 1985 et 1987, administrant une dose excessive de radiation (Leveson, 2008) ; le débordement de mémoire d'une fonction de contrôle de vol critique provoqua la perte d'une sonde spatiale (Rushby, 1999) ; l'explosion d'Ariane 5 (Dowson, 1997).

La sûreté de fonctionnement est une propriété permettant aux utilisateurs d'un système d'avoir une confiance justifiée en les services qu'il leur délivre (Arlat et collab., 2006). Cette sûreté peut être vue selon différentes propriétés complémentaires les unes aux autres : disponibilité, intégrité, maintenabilité, fiabilité, sécurité-innocuité ou confidentialité (voir la figure 3.3). Le développement d'un système sûr de fonctionnement passe par une combinaison de méthodes : la prévention des fautes pour empêcher l'introduction de fautes ; la tolérance aux fautes vise à fournir un service conforme aux spécifications en dépit des fautes ; l'élimination des fautes a pour but de réduire la présence de fautes et de diminuer leur sévérité ; la prévision des fautes

se questionne à savoir comment estimer la présence de fautes, leur activation et les conséquences qu'elles engendrent.

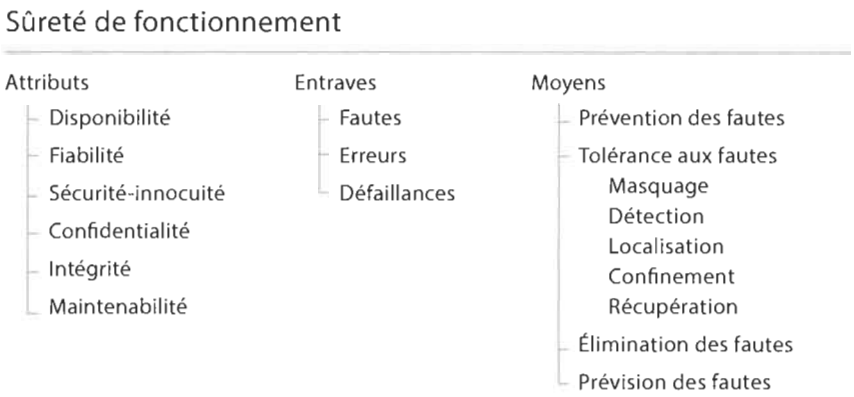

Figure 3.3 Caractéristiques de la sûreté de fonctionnement

La tolérance aux fautes logicielles n'a pas encore atteint la maturité de la tolérance aux fautes matérielles. D'ailleurs, la plupart des efforts en matière de tolérance aux fautes en informatique se sont concentrés sur les fautes physiques (Arlat et collab., 2006, § 8). La tolérance aux fautes logicielles est rendue possible par deux techniques distinctes : les techniques *single-version* et les techniques *multi-version* (McAllister et Vouk, 1996). Alors que les premières visent à renforcer un composant logiciel à l'aide de mécanismes de détection de faute, de confinement et de récupération, les secondes sont basées sur la redondance de composants développés selon la diversité de conception (Avizienis et Kelly, 1984). Notre étude se limite à étudier deux des cinq moyens utilisés pour atteindre une tolérance aux fautes : la détection d'erreurs et le confinement d'erreurs. Pour ce faire, le renforcement de composant est utilisé : les solutions sont implantées sur un composant plutôt que sur le système. Les solutions peuvent être intégrées à tout moment dans la durée de vie d'un logiciel, de la conception à la maintenance. Mais puisque « [l]a majorité du cycle de vie d'un logiciel consiste en son opération (Carter, 1982, p. 41) », nous les appliquons sur un projet existant.

3.5 Détection des erreurs

Afin de s'assurer que le logiciel demeure opérationnel en tout temps et qu'aucun dysfonctionnement n'ait lieu, la pratique courante consiste à s'assurer qu'aucune erreur n'advienne dans les procédures de ce système (von Neumann, 1951). On établit des preuves pour déterminer l'exactitude d'un programme (Goldstine et von Neumann, 1947; Turing, 1949, cité dans Morris et Jones, 1984). Les erreurs ne sont pas inappropriées dans un système, elles adviennent et peuvent être

contrôlées. L'erreur est vue par von Neumann non pas comme un événement accidentel, étranger ou impertinent, mais comme la partie d'un processus essentiel (von Neumann, 1956).

Le génie logiciel reconnaît l'importance des méthodes formelles pour assurer le bon comportement d'un système vis-à-vis des spécifications fonctionnelles implicites ou explicites (Sommerville, 2016). La logique de Hoare (1969) – inspirée par la notation syntaxique de Chomsky (1957) et la définition sémantique formelle de Floyd (1967) –, les langages de spécification algébriques, la logique modale et la sémantique mathématique sont utilisés pour exprimer les spécifications. Ces techniques sont toutefois lourdes à utiliser puisqu'elles requièrent un effort important de la part des programmeurs, en particulier lorsque les connaissances mathématiques sous-jacentes sont manquantes (Pierce, 2002).

Les assertions exécutables peuvent être dérivées en fonction des spécifications formelles d'un système, on parle alors de vérifications à l'exécution (Diaz, Juanole et Courtiat, 1994 ; Jahanian, Rajkumar et Raju, 1994 ; Mok et Liu, 1997 ; Savor et Seviora, 1997 ; Schneider, 1998). Ce type de détection d'erreurs permet de détecter potentiellement toutes les erreurs dans les données internes causées par des fautes logicielles, les fautes matérielles même (Leveson, Cha, Knight et Shimeall, 1990). L'assertion exécutable vérifie les contraintes du système à l'exécution vis-à-vis d'une description formelle du système. L'activité de conception des assertions est toutefois reconnue comme étant difficile, exercice reposant sur la formation ou l'expérience du programmeur (Leveson, Cha, Knight et Shimeall, 1990). Un mécanisme de détection d'erreurs est l'une des étapes nécessaires pour introduire une sûreté de fonctionnement dans un système.

La première étape consiste à détecter les symptômes des fautes, les erreurs (Hiller, 2000). « Une erreur est détectée par un algorithme ou un mécanisme de détection qui permet de la reconnaître comme telle » (Arlat et collab., 2006). L'étape de détection consiste à reconnaître qu'un défaut est survenu. Qu'il s'agisse de *détection de fautes* ou de *détection d'erreurs,* dans les deux cas, ce sont les symptômes d'une faute activée qui sont détectés plutôt que la faute en soi. La faute n'est donc pas localisée, seuls les phénomènes découlant de son activation sont détectés.

On aperçoit à la figure 3.4 les mesures couramment implantées dans les systèmes tolérants aux fautes pour détecter les erreurs (Lee et Anderson, 1990) : le contrôle de réplication, le contrôle temporel, les codes correcteurs, le contrôle d'inversion, la vérification sémantique, la vérification structurelle et le contrôle de diagnostics. Ces mesures correspondent à des tests d'acceptation. Plusieurs techniques de tests d'acceptation ont été proposées (Mahmood, Andrews et McCluskey, 1984 ; Rabéjac, Blanquart et Queille, 1996 ; Stroph et Clarke, 1998),

certains reposent sur des méthodes de modélisation et des modèles mathématiques (voir l'étude exhaustive de Isermann, 2006). Les tests d'acceptation ont prouvé être une méthode efficace pour détecter les erreurs logicielles (Leveson, Cha, Knight et Shimeall, 1990). La majorité de ces tests sont basés sur les assertions exécutables (Hecht, 1976 ; Saib, 1977).

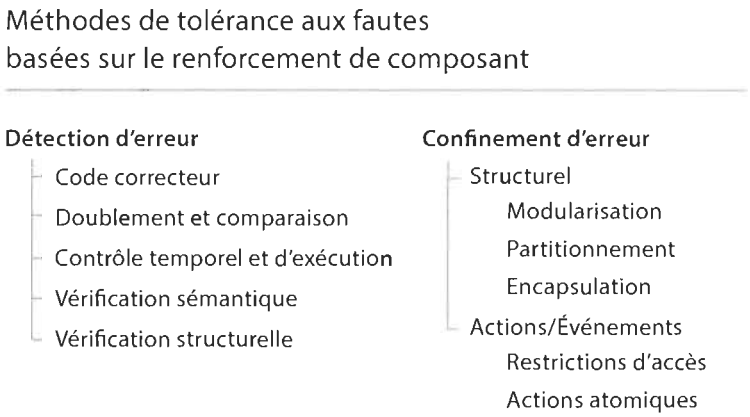

Figure 3.4 Méthodes de tolérance aux fautes

La vérification sémantique est l'un des tests qui, suivant l'exécution d'une opération dans un système, « doit établir si les résultats se situent dans [une] plage d'acceptabilité (Randell, 1975) ». Les variables de ce processus sont mesurées, surveillées et vérifiées, à savoir si leur valeur absolue dépasse une certaine valeur limite (Isermann, 2006). D'ordinaire, deux seuils sont définis : une valeur minimale Y_{min} et une valeur maximale Y_{max}. Les tests sont exprimés sous forme d'expressions booléennes. Selon la variable mesurée $Y(t)$, la condition est rencontrée lorsque $Y_{min} < Y(t) < Y_{max}$. Autrement dit, la condition n'est satisfaite que si la variable mesurée demeure dans l'intervalle de valeurs spécifiées. Non seulement ce type de test permet de détecter les erreurs à l'exécution dans le cadre de mécanismes de tolérance aux fautes (Rabéjac, Blanquart et Queille, 1996), mais il est également utilisé à des fins de test (Andrews, 1979 ; Mahmood, Andrews et McCluskey, 1984). La diversité de conception s'appuie sur les assertions pour détecter les erreurs (Avizienis, 1985).

Nos solutions n'emploient que les assertions exécutables comme mécanisme de détection d'erreurs. L'interface d'un composant sera vérifiée suivant la méthode axiomatique. Lorsqu'une donnée s'introduit dans l'une des méthodes du composant par son interface, celle-ci est sujette à des vérifications qui détermineront sa validité. Les sorties sont vérifiées suivant cette même logique. Ce type de vérification agit comme précondition ou comme contrat (Meyer, 1992), il « spécifie l'état qui doit être satisfait par l'appelant de la méthode » (Karaorman et Abercrombie, 2005). Un logiciel ne pourra utiliser le composant

que si les données en entrée sont valides, les sorties également. L'échec d'une validation signifierait la présence d'une erreur. Aucune procédure corrective ne sera entreprise, seule une exception sera soulevée. La concrétisation des assertions exécutables est discutée au chapitre suivant.

3.6 Confinement des erreurs

Une seconde propriété est nécessaire pour parvenir à limiter la propagation d'erreurs : le confinement d'erreurs. On peut distinguer « confinement » et « isolation ». Alors que le premier consiste à empêcher que les autres composants ne soient informés qu'une erreur est survenue sur un composant, l'isolation assure qu'aucune nouvelle demande entrante ne soit soumise à un composant défaillant en phase de récupération (Li, Chen, Huang, Mei et Chauvel, 2009).

Traditionnellement, cette propriété est réalisée « en modifiant la structure du système et en imposant un ensemble de restrictions définissant quelles actions sont autorisées dans le système » (Dubrova, 2013). La propagation étant ainsi évitée, les erreurs internes ne peuvent plus provoquer de nouvelles erreurs. Aucune erreur externe ne peut également survenir. Les mécanismes utilisés dans les solutions proposées s'appuient sur le renforcement de composant. Elles ne s'appuient donc pas sur la modularisation, le partitionnement, les restrictions d'accès ou les actions atomiques, qui sont des techniques couramment employées, mais étant basées sur une altération globale du système (Dubrova, 2013).

D'ordinaire, un composant est intégré à l'intérieur d'un *wrapper* de protection (Voas, 1998 ; Arlat, Fabre, Rodríguez et Salles, 2002). Initialement définie par la Defense Advanced Research Projects Agency (DARPA), la notion de *wrapper* consiste en une entité logicielle composée de deux fonctionnalités précises : un adaptateur fournissant de nouvelles fonctionnalités et un mécanisme d'encapsulation responsable de lier les composants. Ce *wrapper* est conçu à partir d'assertions exécutables (Mahmood, Andrews et McCluskey, 1984 ; Rabéjac, Blanquart et Queille, 1996 ; Hiller, 2000). La notion de *wrapper* pour composants logiciels a été définie il y a une vingtaine d'années (Voas, 1998).

Le renforcement de composant peut être divisé en deux catégories : le confinement structurel et le confinement événementiel, comme l'indique la figure 3.4. Les solutions basées sur la première sont déployées à même la structure du logiciel existant, dans le code source. Les techniques appartenant au confinement événementiel ajoutent les fonctionnalités de tolérance aux fautes à un métacomposant logiciel. Ainsi, les interactions sur un composant peuvent être capturées sans trop modifier les dépendances intercomposantes. À l'inverse des

techniques basées sur le confinement structurel, aucune modification sur la structure n'a lieu. Cette gamme de techniques englobe les processus réflexifs, les proxies et la programmation orientée aspect. La concrétisation d'un mécanisme de confinement d'erreurs est présentée au chapitre qui suit.

Dans ce chapitre, nous décrivons les déploiements possibles pour implémenter les mécanismes de détection et de confinement d'erreurs. Dans un premier temps, nous présentons la notion d'assertion exécutable comme détecteur. Les structures de confinement d'erreurs présentées recouvrent une variété de techniques et de design patterns. Dans les exemples, l'implantation a lieu sur une méthode de classe.

4.1 Détection des erreurs

L'assertion exécutable est utilisée pour déterminer la validité des données, permettant ainsi de détecter l'apparition d'erreurs sur le composant, en particulier dans les entrées et sorties de méthode. Schématisées à la figure 4.1, les assertions exécutables vérifient la structure des messages et des paramètres de l'interface du composant ainsi que la consistance des données échangées. Ces vérifications à l'exécution induisent généralement un coût en performance (Edwards, Sitaraman, Wiede et Hollingsworth, 2004).

Une assertion repose sur la définition de prédicats. En logique, un prédicat est considéré comme une fonction de valeur booléenne. $P: X \rightarrow \{\text{TRUE}, \text{FALSE}\}$ est appelé prédicat sur X. Un prédicat est une déclaration spécifiant que certaines variables satisfont une propriété (Cunningham, 2012, p. 29), écrite avec la logique de Boole. Ces expressions sont définies comme des instructions permettant de vérifier si une condition donnée est satisfaite parmi les variables et permet d'assurer que le comportement d'une méthode, d'un composant ou d'un système soit conforme aux spécifications du logiciel. Lorsque cette condition n'est pas remplie, des actions peuvent être entreprises. La définition de prédicats constitue un moyen formel pour définir une assertion exécutable. La validité est vérifiée « en testant les valeurs des variables hors de portées, les relations entre les variables et les entrées, et les états connus comme étant corrompus » (Pullum, 2001). Lorsqu'elles sont conçues pour transformer un état invalide en un état valide, les assertions exécutables agissent comme mécanisme de récupération (voir la figure 3.3). Certains langages de programmation comme Java ou C# fournissent des constructions particulières pour les assertions exécutables. Les prochains paragraphes énumèrent les concrétisations possibles.

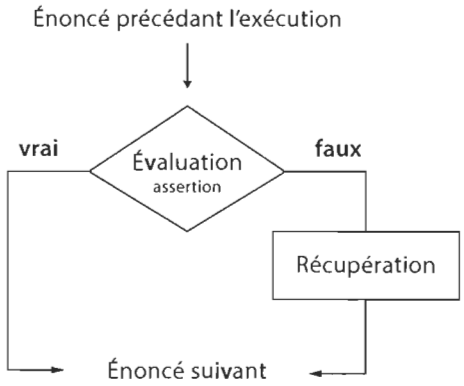

Figure 4.1 Fonctionnement d'une assertion exécutable (d'après Saib, 1977)

Instruction conditionnelle. Généralement, l'opérateur conditionnel *if* est utilisé pour formuler une assertion. Cette vérification permet le contrôle du flot de données à l'entrée et à la sortie d'un composant ou d'une méthode. Comme le présente la figure 4.2, une exception est levée lorsque la condition n'est pas satisfaite, c'est-à-dire lorsque la valeur à vérifier n'est pas dans les bornes spécifiées. Autrement, la méthode est appelée.

```
if (condition)
   method();
else
   throw Exception("Condition non satisfaite");
```

Figure 4.2 Exemple d'instruction conditionnelle en Java

Assertion Java. Offert depuis JDK 1.4, le mécanisme d'assertion de Java permet de vérifier une condition dynamiquement. Lorsque la condition à tester est évaluée comme vraie, le programme continue son exécution. Dans le cas contraire, une exception de type `AssertionError` est levée (Oracle, s.d.-b, § 14.10). Ces assertions Java sont utilisées pour vérifier les pré- et post-conditions durant le développement, mais non lors du déploiement logiciel. Pour ce faire, leur compilation doit être activée manuellement dans l'environnement de développement. Les assertions Java sont équivalentes aux instructions conditionnelles, mais possèdent la syntaxe donnée par la figure 4.3. En comparaison aux techniques qui suivent, l'assertion Java ne dépend d'aucune librairie externe (Sprunck, s. d.).

```
assert condition;
```

Figure 4.3 Exemple d'une assertion Java

Java Modeling Language. Le langage de spécification JML (Leavens et Cheon, 2006) est un langage de modélisation formelle associé à Java. Le langage tire son inspiration des contrats d'Eiffel (Meyer, 1992), paradigme où le déroulement des traitements est régi selon des règles. Formées par des préconditions, des post-conditions et des invariants, ces règles constituent des contrats précisant les responsabilités entre le client (l'utilisateur du composant) et le composant lui-même. JML fournit une sémantique pour décrire formellement le comportement d'un composant. Les spécifications sont écrites sous forme d'annotations commentées dans le code source, précédant l'entête d'une méthode. La compilation est effectuée par le compilateur de Java. Afin que la méthode puisse s'exécuter, le client doit satisfaire la précondition spécifiée par `requires`, comme le montre la figure 4.4. On dira que l'exécution s'est déroulée normalement si la post-condition spécifiée par `ensures` est satisfaite. La violation d'une précondition, d'une post-condition ou d'un invariant engendre la levée d'une exception. Différents outils de vérification à l'exécution fonctionnent de manière similaire : ESC/Java (Flanagan et collab., 2002) ; Dafny (Leino, 2010).

```
/*@
  @ requires condition;
  @ ensures condition;
  @*/
public int method(args) {...}
```

Figure 4.4 Exemple d'une assertion en JML

Autres librairies externes. Guava (O'Madadhain, 2016) ou Apache Commons (s. d.) sont des librairies offrant un ensemble de classes utiles pour concevoir les assertions. Dans les deux cas, une méthode `checkArgument` ou `isTrue` permet de vérifier une condition donnée. Suivant la figure 4.5, la syntaxe utilisée dans ces librairies est semblable à celle des assertions Java.

```
// Guava
Preconditions.checkArgument(condition, "Condition non
 satisfaite");

// Apache Commons
Validate.isTrue(condition, "Condition non satisfaite");
```

Figure 4.5 Exemple d'instruction conditionnelle en Java

Les déploiements retrouvés ici correspondent en partie à des design patterns existants. La plupart sont connus des développeurs expérimentés et sont fréquemment utilisés. Les trois premiers sont basés sur la composition et la délégation ; ils encapsulent un objet et délèguent le traitement au composant. *Adaptateur* utilise la composition pour transférer les appels d'une interface ciblée vers une autre interface adaptée ; *Décorateur* est basé sur le même principe et y définit un nouveau comportement ; *Façade* groupe plusieurs composants sous un seul ; *Proxy* transfère les requêtes à l'aide de la composition et de la délégation. La dernière concrétisation recourt au paradigme orienté aspect.

Adaptateur. Ce design pattern est décrit par Gamma, Helm, Johnson et Vlissides (1994, pp. 139–150) comme un pattern permettant de « convertir l'interface d'une classe en une autre attendue par les clients ». À l'origine, *Adaptateur* permet de résoudre un problème d'incompatibilité entre les interfaces de classe. Le pattern encapsule une, voire plusieurs interfaces et y délègue les appels, comme le présente la figure 4.6. Aussi connue sous le nom de *wrapper,* une variante orientée objet de ce pattern est basée sur la composition et la délégation plutôt que sur l'héritage multiple. L'approche permet ainsi à un adaptateur instanciant un objet de la classe adaptée d'y déléguer les requêtes.

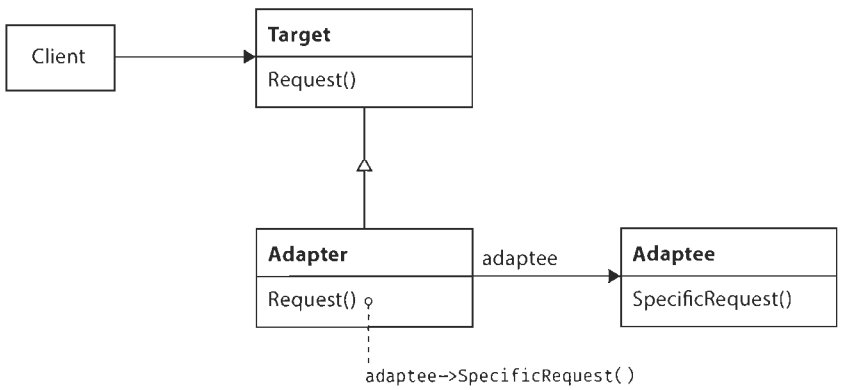

Figure 4.6 Design pattern Adaptateur

Décorateur. Ce pattern peut être vu comme un type particulier d'adaptateur où l'interface du décorateur est un sous-type du composant décoré (Gamma, Helm, Johnson et Vlissides, 1994, pp. 175–184). Comme le montre la figure 4.7, le composant « décoré » est utilisé plutôt que celui qui ne l'est pas. L'interface de ce composant n'est pas convertie. L'interface d'origine de l'objet est implémentée de sorte qu'elle puisse être passée à une méthode qui accepte l'objet d'origine. Des fonctionnalités peuvent également être ajoutées aux méthodes déjà constituées dans le composant d'origine. La signature de l'objet décoré est identique à celle d'origine. Contrairement au pattern adaptateur, *Décorateur* modifie l'implantation du composant plutôt que son interface.

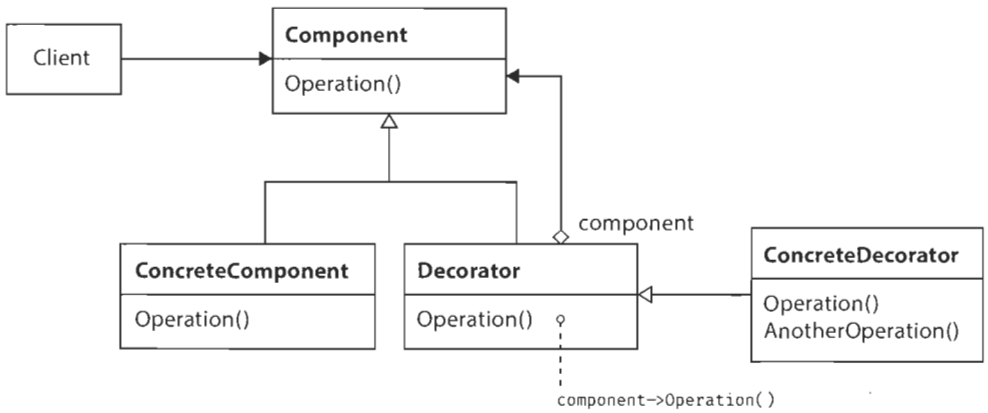

Figure 4.7 Design pattern Décorateur

Façade. Une façade fournit « une interface unifiée à un ensemble d'interfaces dans un sous-système » (Gamma, Helm, Johnson et Vlissides, 1994, p. 185). Comme le présente la figure 4.8, le client utilise la façade, une interface de haut niveau facilitant l'utilisation du sous-système, plutôt que d'accéder directement aux composants individuellement. La façade délègue les appels aux unités respectives lorsqu'elle reçoit une requête du client.

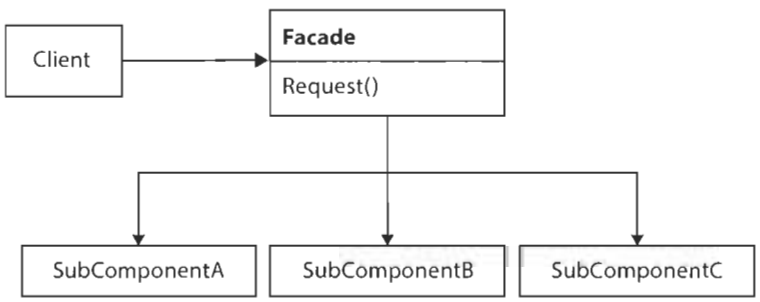

Figure 4.8 Design pattern Façade

Proxy. Gamma et collab. (1994, pp. 207–217) définissent un proxy comme le mandataire d'un composant ayant un contrôle sur certains de ses aspects. Quoique sa structure soit similaire à celle de l'adaptateur ou du décorateur (voir la figure 4.9), le but ultime est de contrôler l'accès à un composant. Son comportement peut être différent de celui du composant. Il peut entre autres déléguer des requêtes au composant. Le client ne voit aucune différence quant à l'utilisation du proxy et celle de l'objet réel, car les deux implémentent la même interface. Le pattern proxy est légèrement distinct du proxy dynamique. Alors que le premier correspond dans sa forme générale à l'interface de quelque chose d'autre, le second « implémente une liste d'interfaces spécifiées à l'exécution de manière à ce qu'une invocation de méthode à l'une des interfaces sur une instance de classe soit encodée et déléguée à un autre objet par une interface uniforme » (Oracle, s. d.-a). Un proxy prend fondation sur le principe de réflexion, principe qui « se produit lorsqu'un système accède aux processus qui en font [un] système » (Fontana, 2019, 28 min 10 s). La réflexion introduit une modularité par laquelle le logiciel peut obtenir des informations sur lui-même et modifier sa structure ou son comportement à l'exécution (Maes, 1987).

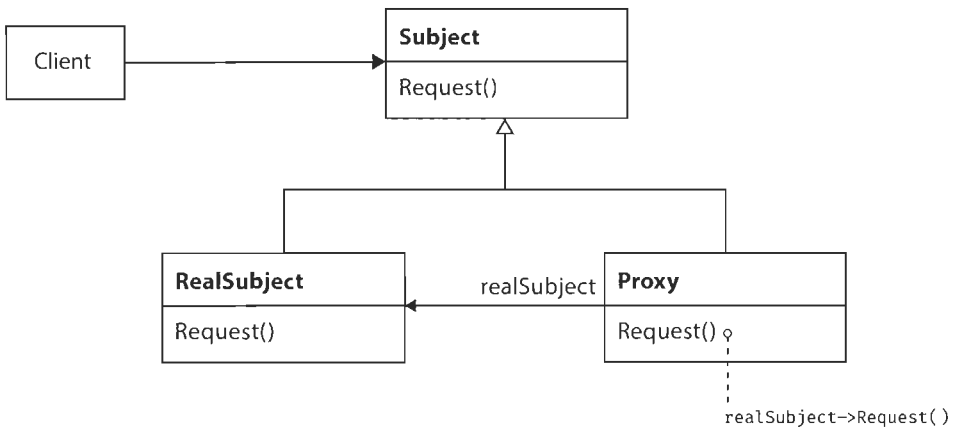

Figure 4.9 Design pattern Proxy

Aspect. La programmation orientée aspect (Kiczales et collab., 1997) est un paradigme de programmation visant à augmenter la modularité en permettant la séparation des préoccupations transversales. Les unités nommées « aspects » spécifient quelles parties d'un système devraient voir leur comportement modifié pour accueillir une préoccupation particulière : des fonctionnalités de tolérance aux fautes, par exemple. En capturant les appels de méthode, le point de coupure de l'aspect greffe un comportement nouveau dans le code de la méthode. Le paradigme fournit un langage avec différents mécanismes d'abstraction et de composition. La figure 4.10 démontre qu'un processeur de

langage dédié appelé « tisseur d'aspect » est utilisé pour coordonner la co-composition des aspects et des composants orientés objet. Les aspects sont fréquemment utilisés pour les préoccupations telles que la gestion des utilisateurs, la persistance ou la journalisation.

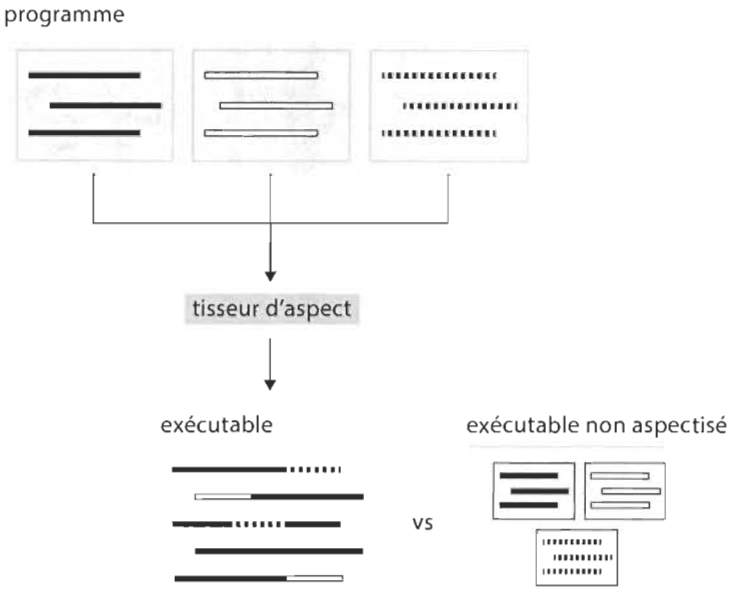

Figure 4.10 Comparaison entre une compilation avec une extension orientée aspect et une compilation standard

Dans ce qui suit, nous présentons trois solutions concrétisées à partir des méthodes de détection et de confinement existantes. Chaque solution est présentée de manière exhaustive et un exemple de code est fourni.

5.1 Adaptateur

Cette solution est basée sur les assertions exécutables comme moyen de détection d'erreurs et sur l'encapsulation de composants comme méthode de confinement. Tel que le présente la figure 5.1, un composant est encapsulé par un adaptateur (Gamma, Helm, Johnson et Vlissides, 1994). On peut employer cette solution lorsque le code du composant ciblé est accessible et modifiable. Plusieurs classes pourraient être réunies sous un même adaptateur, à la manière d'une façade.

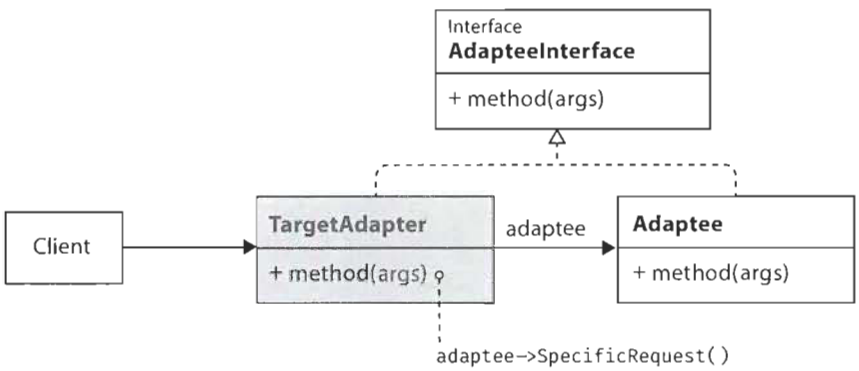

Figure 5.1 Solution basée sur un adaptateur

Target est responsable des opérations demandées et utilise l'adaptateur. *Client* collabore avec les objets selon l'interface de *Target*. *AdapteeInterface* définit l'interface commune utilisée par *Adaptee* et *Adapter*. *Adapter* adapte l'interface d'*Adaptee* pour *Target*. Finalement, *Adaptee* est une classe préexistante nécessitant une protection.

5.1.1 Solution

On commence par créer une interface où sont ajoutés les en-têtes des méthodes de la classe ciblée. Ces méthodes sont à la fois redéfinies par une nouvelle classe, l'adaptateur, et la classe ciblée. Autrement dit, les deux classes implémentent l'interface nouvellement créée. La première correspond à un adaptateur objet du design pattern *Adaptateur*. La classe ciblée demeure indépendante de l'adaptateur; son niveau de visibilité demeure inchangé. L'adaptateur instancie un objet du type de la classe ciblée et implémente les méthodes de la classe ciblée de par

l'interface nouvellement créée. Puisque l'adaptateur traite dorénavant les requêtes de la classe ciblée, le client doit modifier les instanciations d'objet de l'ancienne classe pour l'adaptateur. Lorsqu'une méthode est appelée sur l'adaptateur, celui-ci vérifie d'abord si les paramètres d'entrée sont valides à l'aide d'une assertion exécutable. La vérification est concrétisée par une instruction *if*. Lorsque les paramètres sont conformes, la méthode originale est appelée dans la classe ciblée. Une seconde vérification est effectuée au retour d'un résultat de la méthode pour assurer la validité du résultat. Lorsque la condition n'est pas satisfaite, une exception est levée. La concrétisation de cette solution en Java est présentée à la figure 5.2.

```
interface AdapteeInterface {
   public int method();
}

class Adaptee implements AdapteeInterface {
   public int method() {...}
}

class Adapter implements AdapteeInterface {
   private Adaptee _adaptee;

   public Adapter() {
      _adaptee = new Adaptee();
   }

   public int method() {
      if (precondition) {
         int result = _adaptee.method();

         if (!postcondition)
            throw new IllegalArgumentException();
      }
   }
}
```

Figure 5.2 Code source de l'adaptateur

5.1.2 Collaborations

Le client appelle les méthodes sur `Adapter`, lui-même instanciant un objet `Adaptee`. L'adaptateur appelle les méthodes sur `Adaptee`, permettant ainsi de répondre aux requêtes du client.

5.1.3 Conséquences

Un adaptateur objet permet à une seule classe d'interagir avec plusieurs adaptés, soit l'adapté lui-même et ses sous-classes. L'adaptateur fournit des fonctionnalités de tolérance aux fautes aux classes adaptées sans en modifier le comportement. La solution est néanmoins considérée comme « invasive » puisqu'une modification de la classe ciblée est nécessaire. La signature des méthodes demeure toutefois inchangée. Une fois l'adaptateur développé, les instances d'`Adaptee` dans le client sont modifiées pour `Adapter`. Ainsi, les requêtes sont redirigées vers l'adaptateur et les méthodes peuvent être supplémentées d'une tolérance aux fautes. Quoiqu'elle puisse être coûteuse dans certains logiciels complexes, la modification des instances d'objet dans le client pourrait être automatisée à l'aide d'un outil de refactoring.

5.1.4 Utilisations connues

L'intégration de la classe ciblée dans l'adaptateur est une variante du design pattern *Adaptateur* définit dans *Design Patterns* (Gamma, Helm, Johnson et Vlissides, 1994). *Façade* agit comme l'interface unifiée d'un ensemble de composants, une sorte de passerelle présentant une interface unifiée pour l'ensemble d'un sous-système. Selon le nombre de composants impliqués, une façade pourrait plutôt être utilisée lorsque plusieurs composants liés entre eux doivent être protégés. Voas (1998) a proposé une approche où les composants COTS dont le code est inaccessible sont prémunis d'une robustesse supplémentaire. Le pattern proposé par Saridakis (2003) étend l'*Adaptateur* en garantissant que l'adaptation du composant ne provoque aucune propagation d'erreurs. L'adaptateur est vu comme un *wrapper* transformant un composant en une unité de confinement de fautes. L'unité forme ainsi une barrière « empêchant la propagation des erreurs dans les deux sens, de l'intérieur de l'unité vers l'extérieur et vice versa » (Saridakis, 2003, paragr. 1).

Cette solution est basée sur les assertions exécutables comme moyen de détection d'erreurs et sur la réflexion Java comme méthode de confinement. L'observation réflexive est concrétisée par une implémentation de `java.lang.reflect.InvocationHandler` (Oracle, s.d.-a). Une version courante ou supérieure à JDK 1.1 est requise pour utiliser l'API réflexive.

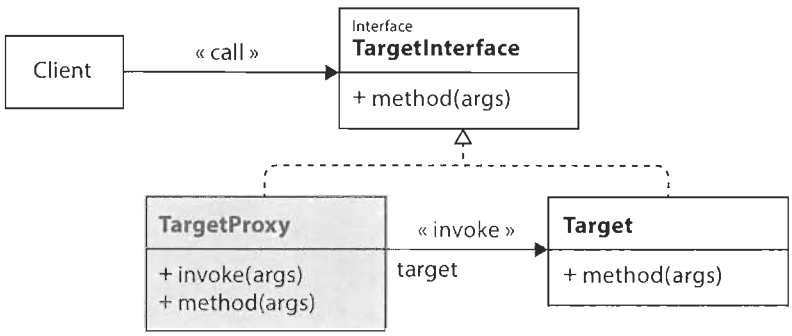

Figure 5.3 Solution basée sur un proxy

Proxy est métacollaborateur et contrôle les interactions entre *Target* et le client. Client collabore avec les objets selon l'interface commune de *Proxy* et *Target*. *TargetInterface* définit l'interface commune utilisée par le proxy et *Target*. *Target* est une classe préexistante nécessitant une protection.

5.2.1 Solution

Pour implémenter la solution suggérée, on commence par créer une interface où l'on ajoute les en-têtes des méthodes de la classe ciblée (voir la figure 5.3) Dans un second temps, on conçoit une classe implémentant `InvocationHandler` (package `java.lang.reflect`); cette classe correspond au proxy, et est donc un gestionnaire des invocations de `Target`. La méthode `invoke` est implémentée par le proxy. Dans le client, chacune des instances d'objet de la classe ciblée doit être réécrite. Les instances d'objet sont converties en instance de proxy via un appel à la fabrique `java.lang.reflect.Proxy`. Le proxy peut dorénavant observer et contrôler les requêtes de la classe ciblée. Lorsqu'un appel de méthode est effectué sur la cible, `invoke` reçoit l'invocation. Comme toutes les invocations de méthodes y transitent, on vérifie d'abord le nom de la méthode appelée. Lorsque le nom de la méthode correspond à celui de la méthode ciblée, la validité des paramètres en entrée est vérifiée par une instruction *if*. Lorsque les paramètres sont conformes, la méthode originale est invoquée sur le composant ciblé en invoquant `invoke`. Le résultat de l'invocation est ensuite vérifié, assurant ainsi la validité du résultat en sortie. Lorsque la condition n'est pas satisfaite, une exception est levée. On retrouve à la figure 5.4 la concrétisation de cette solution en Java.

```
interface TargetInterface {
   public int method();
}

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

class DynamicProxy implements InvocationHandler,
 TargetInterface {
   private Target _target;

   public Proxy() {
      _target = new Target();
   }

   public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
      if (!method.getName().equals("method"))
         return method.invoke(_target, args);

      if (precondition) {
         int result = _target.method();

         if (!postcondition)
            throw new IllegalArgumentException();
      }
   }
}

TargetInterface proxy =
 (TargetInterface) Proxy.newProxyInstance(
 TargetProxy.class.getClassLoader(),
 new Class[] { TargetInterface.class },
 new TargetProxy()
);
```

Figure 5.4 Code source du proxy

5.2.2 Conséquences

Le proxy fournit des fonctionnalités de tolérance aux fautes à la classe ciblée sans en modifier son comportement. La solution est néanmoins considérée comme « invasive » puisque le code de la classe ciblée doit être modifié. L'interface de la solution sera implémentée par le proxy. Une fois le proxy développé, les instances d'`Adaptee` dans le client sont modifiées pour un appel à la fabrique `Proxy`. Ainsi, les requêtes transitent vers le proxy par réflexion où elles peuvent être interceptées puis contrôlées. L'utilisation de la réflexivité en Java soulève toutefois des préoccupations quant aux performances (Tudose, Odubăşteanu et Radu, 2013). Une dégradation significative des performances pourrait être observée. Par conséquent, l'utilisation de la réflexion pourrait être restreinte aux endroits où la performance n'est pas une préoccupation majeure.

5.2.3 Utilisations connues

L'API de réflexion Java est une concrétisation du design pattern *Proxy* (Gamma, Helm, Johnson et Vlissides, 1994, pp. 207–217). Son utilisation comme moyen de tolérance aux fautes logicielles a été proposée par Salles, Rodriguez, Fabre et Arlat (1999) et dans un second temps par Rodríguez, Fabre et Arlat (2002).

5.3 Aspect

Cette solution est basée sur les assertions exécutables comme moyen de détection d'erreurs et sur le tissage d'aspects. L'utilisation de la librairie AspectJ (Eclipse Foundation, 2019-a) est requise pour compiler la solution.

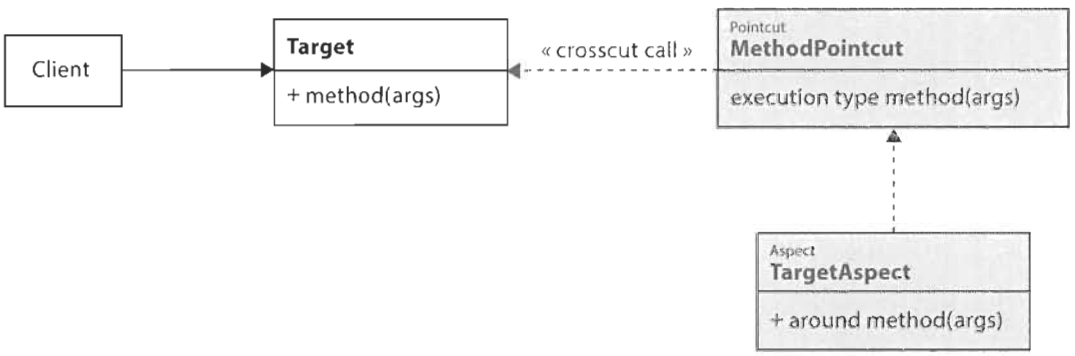

Figure 5.5 Solution basée sur un aspect

TargetAspect est métacollaborateur et contrôle les interactions entre *Target* et le client; il contient le point de coupure. *MethodPointcut* est représentant d'un point de coupure. *Client* collabore avec *Target*. *Target* est une classe préexistante nécessitant une protection.

5.3.1 Solution

Avant toute chose, le projet doit être converti en projet AspectJ. On commence par créer un aspect où l'on insère un point de coupure avec pour cible l'une des méthodes de la classe ciblée. Un advice est également ajouté. L'advice, qui détermine le moment où sera exécuté l'aspect, est exécuté autour du déroulement du point de coupure. Le type de retour dans la signature de l'advice correspond au type de retour de la méthode dans la classe ciblée. Ainsi, les traitements ont lieu avant et suivant l'exécution du point de coupure. Autrement dit, l'aspect sera exécuté lors de l'exécution de la méthode. Le corps de l'advice correspond au code détecteur d'erreurs. À l'aide d'une assertion exécutable, on vérifie d'abord si les paramètres d'entrée de la méthode sont valides. La vérification est concrétisée par une instruction *if*. Lorsque les paramètres sont conformes, la méthode originale est appelée dans la classe cible avec `proceed`. Une seconde vérification est effectuée à la fin de l'exécution de la méthode, assurant ainsi la validité du résultat. Lorsque la condition n'est pas satisfaite, une exception est levée. La solution est schématisée à la figure 5.5 et concrétisée en Java à la figure 5.6.

```
aspect TargetAspect {
   pointcut MethodPointcut(args):
     target(arg)
     && args(arg)
     && execution(public void Target.method(type));

   int around(args): MethodPointcut(args) {
      if (precondition) {
         int result = proceed(args);

         if (!postcondition)
            throw new IllegalArgumentException();
      }
   }
}
```

Figure 5.6 Code source de l'aspect

5.3.2 Conséquences

En permettant une séparation des préoccupations transversales, la programmation orientée aspect permet un découplage du code métier et du code de tolérance aux fautes. Aucune modification du code source existant n'a lieu. Le client n'y voit aucun changement puisque l'implantation de l'aspect laisse le code orienté objet intact. Il est toutefois reconnu que l'implantation d'AspectJ impose un coût d'exécution supplémentaire (Eclipse Foundation, 2009). Le moment où l'aspect est exécuté, « autour » du déroulement du point de coupure dans notre cas, génère un *overhead* plus significatif qu'un advice *before* ou *after* (Šuta, Martoš et Vranić, 2015).

5.3.3 Utilisations connues

Bien que cette solution repose sur un framework, Shah et Hill (2003) ont vu dans l'utilisation de la programmation aspect une façon d'instaurer une tolérance aux fautes de manière globale dans un logiciel. La même approche est utilisée dans l'article de Afonso, Silva, Brito, Montenegro et Tavares (2008), où chacune des classes d'un système est transformée en aspect avec AspectC++. À la différence de notre solution, leur solution s'exécute autant avant l'exécution que suivant l'exécution de la méthode (respectivement un advice *before* et *after*).

6.1 Objectif

Trois solutions sont soumises à l'évaluation. Comme elles n'ont pas été évaluées dans un contexte orienté objet, nous procédons à leur évaluation dans Java. Il nous semble nécessaire d'évaluer ces trois solutions sous un même contexte. Une évaluation permet ainsi de quantifier leur efficacité de manière adéquate, de constater leurs ressemblances et différences. Une comparaison de leur performance individuelle devient possible puisqu'elles sont appliquées sur une même étude de cas où l'environnement est stable et qu'une même configuration est utilisée. De ce fait, nous serons en mesure de mieux définir la robustesse, la fiabilité et l'efficacité des solutions dans l'optique de limiter la propagation des erreurs. Une évaluation préliminaire a été effectuée afin de stabiliser la méthodologie d'évaluation. Les résultats de cette évaluation préliminaire viennent appuyer ceux d'une seconde évaluation, cette fois-ci effectuée sur une étude de cas connue.

6.2 Méthodologie

Nous décrivons dans cette section la méthodologie adoptée pour parvenir à l'évaluation des solutions retenues.

6.2.1 Choix de l'étude de cas

L'évaluation préliminaire emploie un simulateur de guichet automatique Java comme étude de cas (Bjork, 2001). Le choix de la classe ciblée est déterminé par les métriques *fan-in* et *fan-out* (Henry et Kafura, 1981), respectivement notées F-IN et F-OUT. *Fan-in* est défini comme le nombre de classes qui réfèrent une classe donnée ; *fan-out* correspond au nombre de classes appelées par celle-ci. Dans ce logiciel, la classe `Money` est ciblée par l'implantation des solutions vu le nombre élevé de dépendances (F-IN = 15). Chacune des méthodes de cette classe effectue une opération arithmétique : additionner, soustraire, multiplier et diviser. La méthode de soustraction `subtract` est sélectionnée au hasard pour implanter les solutions.

L'évaluation se concentre toutefois davantage sur une étude de cas connue, le logiciel open source orienté objet JHotDraw 7 (Randelshofer, 2010-a). Un logiciel open source convenait parfaitement aux exigences de l'évaluation puisque les deux premières solutions proposées nécessitent une modification du code source. JHotDraw est un framework de dessin graphique en 2D développé en Java. Sa conception s'appuie sur certains design patterns de *Design Patterns*

(Gamma, Helm, Johnson et Vlissides, 1994). C'est d'ailleurs pour démontrer l'applicabilité des patterns que cette application a été lancée en 2000. La version 7.7.0 est datée de 2010 et est celle utilisée pour l'évaluation. Le tableau 6.1 fournit des détails sur JHotDraw 7.7.0. Le logiciel contient 84 077 lignes de code non commentées réparties en 1094 classes (dont 65 sont des interfaces). Le facteur de couplage, soit le degré de couplage du projet dans son ensemble, est d'environ 2,01 % et la complexité cyclomatique moyenne (de toutes les méthodes non abstraites du projet) est de 2,39. Cette étude de cas permettra de démontrer la faisabilité de notre approche et de la positionner vis-à-vis de celles de Voas (1998) et de Salles, Rodriguez, Fabre et Arlat (1999).

Tableau 6.1
Statistiques sur JHotDraw 7.7.0

Version	7.7.0
Langage de programmation	Java
Taille approximative SLOC	84 077
Nombre de classes NOC	1 094
Nombre d'interfaces NIC	65
Facteur de couplage CF	2,01 %
Complexité moyenne v(G)avg	2,39

Le choix du composant à utiliser pour l'évaluation est crucial puisque la qualité de l'évaluation en dépend. Le composant correspond ici à une classe. Un modèle de classe partiel est présenté à la figure 6.1. Un composant fortement couplé est préféré parce qu'il est plus susceptible de pouvoir engendrer une propagation d'erreurs (Page-Jones, 1988). Un parcours de la littérature (§ 2.1) relève une similarité dans les études de cas utilisées pour l'évaluation. Dans tous les cas, les solutions ont été conçues pour augmenter la robustesse d'un composant, soit d'un composant COTS (Voas et Miller, 1997 ; Voas, 1998 ; Ghosh, Schmid et Hill, 1999 ; Fetzer et Xiao, 2003), d'un composant COTS utilisé comme régulateur PID (Anderson, Feng, Riddle et Romanovsky, 2003 ; van der Meulen, Riddle, Strigini et Jefferson, 2005) ou soit d'un noyau de système d'exploitation COTS (Fraser, Badger et Feldman, 1999 ; Fabre, Salles, Rodríguez Moreno et Arlat, 1999 ; Salles, Rodriguez, Fabre et Arlat, 1999 ; Arlat, Fabre, Rodríguez et Salles, 2002 ; Rodríguez, Fabre et Arlat, 2002). Puisque le but est de déterminer l'efficacité des solutions sur un seul composant plutôt qu'un système dans sa globalité, un seul composant est sélectionné. Parmi les classes du logiciel évalué, plusieurs dépendent de `BezierFigure` (F-IN = 42). Cette classe qui redéfinit les méthodes de `AbstractAttributedFigure` est utilisée pour représenter une figure avec des courbes de Bézier. Une figure Bézier peut posséder des segments droits et des segments courbés (Randelshofer, 2010-b). Les opérations effectuées par la

classe sont relatives à la manipulation de cette figure : remplissage; connexion; ajout et suppression de points; configuration des limites et des points; dessiner sur l'écran; etc. Cette classe est reliée à plusieurs autres (F-OUT = 15) et en étend d'autres dont `AbstractAttributedFigure` étant en dépendance directe avec l'interface `Figure`.

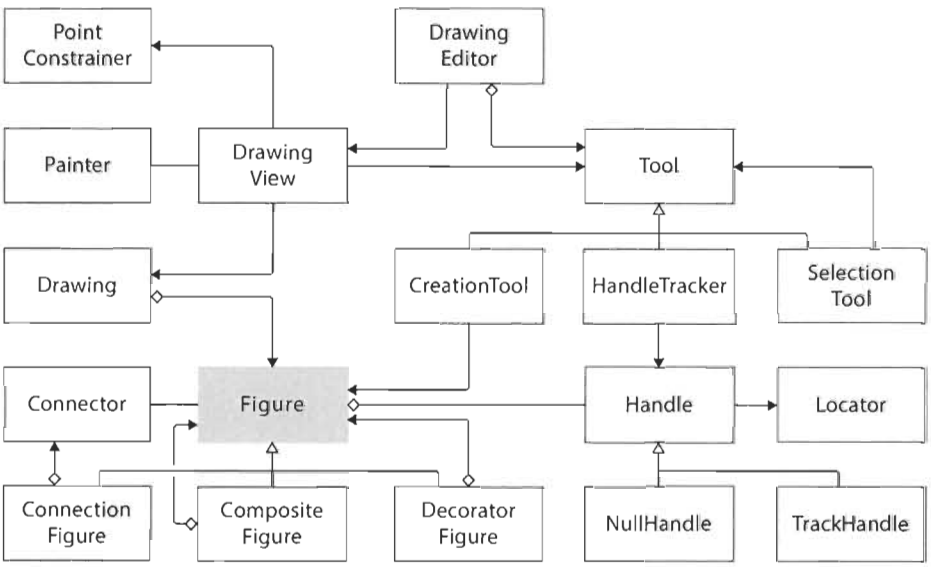

Figure 6.1 Modules essentiels de JHotDraw (tiré de Riehle, 2007 et adapté par l'auteur)

Vu la densité de `BezierFigure`, une seule méthode est désignée sur cette classe. Une méthode fortement couplée est préférée: `getNode`. L'entier passé en paramètre rend la méthode facilement manipulable par une injection de fautes. La méthode permet d'obtenir un nœud de la courbe appartenant à la courbe de Bézier. Le nœud obtenu est cloné puis retourné au client. Une seule ligne de code compose la méthode : un appel à une méthode de `BezierPath`, d'où la possibilité de propagation d'erreurs. Comme le présente la figure 6.2, la fonction, avant de retourner les coordonnées XY du nœud recherché, en appelle une autre de Java où une assertion vérifie si la valeur de l'index paramétré dans `getNode` est située entre zéro et la taille de la longueur du tableau.

```
public BezierPath.Node getNode(int index) {
   return (BezierPath.Node) path.get(index).clone();
}
```

Figure 6.2 Code source de la méthode ciblée par l'évaluation

6.2.2 Implantation des solutions

Pour l'évaluation préliminaire, les assertions exécutables sont conçues selon les spécifications fonctionnelles fournies par le concepteur du logiciel (Bjork, 2001, § *Detailed Design*). Il est spécifié que le montant passé en paramètre de méthode doit être un nombre positif et inférieur ou égal à la valeur monétaire courante (la valeur de l'objet autrement dit). Comme aucune valeur n'est retournée par la méthode, les sorties ne sont pas vérifiées. L'évaluation a été effectuée avec un processeur 3.4 GHz Intel Core i7 avec 8 Go de mémoire dans l'environnement de développement Eclipse 2019-03 (Eclipse Foundation, 2019-b) sous Windows 10 64 bit (17763.557). Les versions de logiciels suivants sont utilisées : JRE 1.8.0_201 ; JUnit 4.12 ; AspectJ Runtime 1.9.2.

Pour JHotDraw, les assertions exécutables sont écrites en fonction des commentaires d'en-tête des méthodes dépendantes (méthodes `get` dans la classe `BezierPath`, `get` dans `java.util.ArrayList` et `checkIndex` dans `java.util.Objects`). Comme on peut le voir à la figure 6.3, l'index doit correspondre à la position d'un nœud situé sur la courbe, la courbe correspondant à une liste à dimension variable. La précondition d'exécution de `getNode` spécifie que la valeur de l'entier paramétré x' doit être supérieure à zéro et inférieure à la taille de la liste x : $x' \in \mathbb{Z} \wedge 0 \leq x' < x$. Implicitement, cet entier signé 32 bits est non nul et situé dans les bornes de son type (entre -2^{31} et $2^{31} - 1$).

```
if ( !(index ≥ 0 && index < super.getNodeCount()) )
   throw new AssertionError();

BezierPath.Node result = getNode(index);

if (result == null)
   throw new AssertionError();
```

Figure 6.3 Code source des assertions exécutables développées

L'objet `Node` en sortie est le doublon d'un nœud existant; nous supposons qu'il ne peut être nul. On se référera à la section précédente pour la méthodologie d'implantation des mécanismes de confinement d'erreurs. Ces expérimentations ont été exécutées sur un processeur 3,1 GHz Intel Core i7 double cœur avec 16 Go de mémoire dans l'environnement de développement IDEA 2020.1.1 (Intellij, 2020) exécuté sous macOS 10.15.4. Les versions de librairies et logiciels suivants ont été utilisées : JRE 1.8.0_241 ; JUnit 4.13 ; AspectJ Runtime 1.9.5 ; Javassist 3.26 ; JMH 1.23.

6.2.3 Procédé d'évaluation

L'injection de fautes permet de tester les mécanismes de tolérance aux fautes selon des entrées spécifiques, soit les fautes qu'elles sont censées tolérer. On emploie généralement ce processus pour évaluer la robustesse d'une application (Voas, 1997; Feinbube, Pirl et Polze, 2018). Les tests en boîte noire (Voas, 1998) et les tests de performance (Fraser, Badger et Feldman, 1999) peuvent aussi être utilisés. L'injection de fautes désigne l'insertion volontaire de fautes et d'états erronés dans un système en exécution et peut être considérée comme complémentaire au test logiciel. Tandis que les fautes sont injectées dans un composant, le comportement du système est observé. La technique permet d'assurer le bon fonctionnement d'un logiciel sous une charge de travail importante, contrairement au test logiciel où la charge de travail est planifiée et plus régulière. L'injection de fautes est efficace pour évaluer la fiabilité de composants COTS (Koopman et De Vale, 1999), mais également d'autres composants (Natella, Cotroneo et Madeira, 2016). Dans notre cas, les fautes sont injectées au niveau des paramètres de la méthode ciblée. Généralement, l'interface d'une classe est ciblée en y injectant des valeurs générées aléatoirement (Feinbube, Pirl et Polze, 2018). Les statistiques obtenues sont utilisées afin de déterminer le taux de couverture du mécanisme de détection. Le taux de couverture de détection d'erreurs dépend du mécanisme de détection et de confinement d'erreurs utilisé (Constantinescu, 1994). L'inférence statistique est fréquemment utilisée pour dériver des mesures de fiabilité significatives suite à l'injection d'un nombre limité de fautes aléatoires (Arlat et collab., 1990; Powell, Martins, Arlat et Crouzet, 1993).

L'utilisation d'une technique d'évaluation telle que l'injection de fautes contribue à définir la robustesse des solutions. L'IEEE définit la robustesse comme « [l]a mesure dans laquelle un système ou un composant peut fonctionner correctement en présence d'entrées non valides ou de conditions environnementales stressantes » (IEEE, 1990, p. 64). Par le passé, les tests de robustesse ont été utilisés pour évaluer le type de solutions évaluées ici (Leveson, Cha, Knight et Shimeall, 1990; Voas, 1998; Fabre, Salles, Rodríguez Moreno et Arlat, 1999; Ghosh, Schmid et Hill, 1999; Fetzer et Xiao, 2003), en particulier pour valider la fiabilité des logiciels (Clark et Pradhan, 1995). Deux types d'injections de fautes sont effectués : l'un dans les paramètres de méthode; l'autre au niveau structurel. Pour chaque test effectué, le rapport établi entre les résultats obtenus avant et suivant l'implantation permet d'évaluer la robustesse du mécanisme de détection d'erreurs. Ainsi, on arrive à déterminer la couverture d'erreurs détectées. D'autres critères sont complémentaires à l'évaluation : la taille de la solution en lignes de code (Leveson, Cha, Knight et Shimeall, 1990);

la performance selon le temps d'exécution (Leveson, Cha, Knight et Shimeall, 1990; Fraser, Badger et Feldman, 1999; Arlat, Fabre, Rodríguez et Salles, 2002; Rodríguez, Fabre et Arlat, 2002; Afonso, Silva, Brito, Montenegro et Tavares, 2008); l'utilisation processeur (Afonso, Silva, Brito, Montenegro et Tavares, 2008); l'observation d'une propagation ou non (Fabre, Salles, Rodríguez Moreno et Arlat, 1999; Salles, Rodriguez, Fabre et Arlat, 1999; Arlat, Fabre, Rodríguez et Salles, 2002). Ces critères viendront appuyer la discussion des résultats.

Les *fuzz tests* (Miller, Fredriksen et So, 1989) opèrent dans les paramètres de la méthode désignée en y injectant des valeurs aléatoires. Un cas de test JUnit accomplit cette tâche, JUnit étant un framework permettant de développer et exécuter des tests unitaires (JUnit, 2020). Dans les deux études de cas, un entier pseudoaléatoire uniformément réparti entre -2^{31} et $2^{31} - 1$ est généré à l'aide du singleton `ThreadLocalRandom` (ainsi la génération est isolée du thread courant) et est injecté à cent reprises dans le paramètre de la méthode évaluée. Puisque le nombre injecté est généré aléatoirement, le nombre passé en paramètre est différent d'une exécution à une autre; ainsi le nombre de fautes injectées diffère d'une évaluation à une autre. La répétition de la même expérience avec différentes entrées sur une méthode où les altérations ont lieu sur les paramètres permet de déterminer un score, une fréquence à laquelle un module agit de manière robuste (Voas et Miller, 1997). Le nombre d'itérations permet donc d'obtenir une plus grande consistance dans les résultats obtenus. Le cas de test est paramétré avec `@RunWith(Parameterized.class)` pour pouvoir exécuter les tests de manière individuelle, améliorant ainsi la lisibilité des résultats. Selon les résultats obtenus, le taux de couverture de détection d'erreurs est déterminé. La latence de détection pourrait également être dérivée des résultats générés.

Pour JHotDraw, une seconde injection de fautes est effectuée au niveau structurel, dans le bytecode. Le bytecode est une version bas niveau du logiciel qui s'exécute sur la machine virtuelle Java (Sedgewick et Wayne, 2014). La technique permet de remplacer les données contenues dans la méthode par des données invalides. Le processus d'instrumentation est schématisé à la figure 6.4. On utilise un agent Java (Oracle, s.d.-c) pour intercepter la classe ciblée et Javassist (Chiba, 2019) pour manipuler le bytecode. Le bytecode est plus facilement manipulable avec Javassist qu'avec l'API fournie par le JDK. L'agent permet d'intégrer le code fautif à la JVM à l'exécution, soit une valeur négative. Le code source de l'agent est donné en annexe.

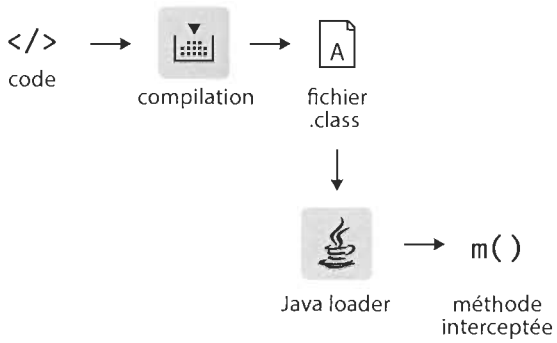

Figure 6.4 Instrumentation d'une classe avec Javassist

Suivant les injections de fautes, on recourt à des métriques de code pour quantifier des attributs de qualité : la maintenabilité, la testabilité et la performance. Un logiciel en constante évolution nécessite une haute maintenabilité pour pouvoir accueillir de nouvelles spécifications. Il nous semble important qu'une solution puisse contribuer à rendre une application maintenable, de manière à ne pas dégrader de façon significative sa maintenabilité. Ainsi, elle contribue à la qualité générale du logiciel. La testabilité, définie comme la simplicité avec laquelle un programme peut être testé (Bach, 1994, cité dans Pressman, 2014, p. 497), contribue à rendre une application maintenable. Cet attribut assure qu'une solution demeure testable lors de son implantation et contribue ainsi à identifier les solutions qui facilitent le processus de test. La performance, elle, vise à assurer que la solution conçue réponde à la problématique, et ce dans un temps raisonnable. Quoique ces métriques soient secondaires à l'interprétation et à la discussion concernant les injections de fautes, l'apport de ces caractéristiques est important : elles permettent de quantifier la qualité logicielle. Les métriques logicielles sont utiles dans l'évaluation des attributs de qualité (Basili, Briand et Melo, 1996; Fenton et Pfleeger, 1997; Pressman, 2014; Sommerville, 2016) et cruciales dans la prise de décisions (Harrison, 1994). La disponibilité de telles métriques pour caractériser les solutions nous apparaît essentielle puisque toutes les solutions n'ont pas le même niveau de maintenabilité, de testabilité et ne performent pas de la même manière. Ainsi pourra-t-on identifier quelle solution est la plus adéquate à être utilisée dans un cas particulier.

Les prédicteurs de maintenabilité les plus couramment utilisés sont ceux basés sur la taille, la complexité et le couplage logiciel (Riaz, Mendes et Tempero, 2009). La maintenabilité est quantifiée à l'aide des métriques de Chidamber et Kemerer (1994) : CBO, F-IN, WMC, RFC, LCOM. La première définit le couplage entre objets ; F-IN définit le nombre de classes dépendantes ; les métriques de complexité WMC et RFC sont décrites, respectivement, comme la somme des complexités des méthodes d'une classe et comme l'ensemble des méthodes qui

peuvent être exécutées en réponse à un appel reçu par un objet; finalement, LCOM mesure le manque de cohésion entre les méthodes d'une classe. Toutes ces métriques sont collectées sur la classe désignée et sur les solutions avec MetricsReloaded (Leijdekkers, 2020). L'indice de maintenabilité MI (Coleman, Ash, Lowther et Oman, 1994) est également utilisé, indice reposant sur les métriques de McCabe (1976) et d'Halstead (1977) puis implanté dans Visual Studio (Naboulsi, 2011). La valeur de MI est obtenue par la formule suivante : $MI = 171 - 5{,}2\ \ln(V) - 0{,}23(VG) - 16{,}2\ \ln(LOC)$; où V correspond au volume d'Halstead, VG à la complexité cyclomatique, LOC au nombre de lignes de code.

La performance des solutions est le dernier critère évalué. L'attribut qu'est la performance est défini comme « [l]a mesure selon laquelle un système ou un composant remplit ses fonctions spécifiées dans des contraintes données, telle que la vitesse, la précision ou l'utilisation de la mémoire» (IEEE, 1990, p. 55). JMH est utilisé pour calculer le temps d'exécution moyen de la méthode ciblée. La collecte du temps d'exécution est précédée d'une période d'« échauffement » de la JVM durant laquelle le code est exécuté sans prendre de mesure. Cinq itérations d'échauffement d'une durée de 10 secondes chacune sont exécutées sur quatre threads pour une période de 50 secondes. Cela permet à la machine d'allouer des ressources nécessaires et de compiler le bytecode pour les itérations calculées. Quoique peu d'itérations sont utilisées, le nombre d'itérations permet d'obtenir une moyenne considérable si l'on considère les phases d'échauffement dans le résultat. Le temps est mesuré en nanosecondes et correspond au temps moyen que prend la méthode pour être exécutée. La comparaison des résultats obtenus sur le logiciel sans solution et implémentant une solution permet d'évaluer la performance de chacune des solutions.

6.3 Résultats

Dans cette section, nous présentons les résultats de l'évaluation des deux études de cas en trois parties : d'abord en présentant les résultats des injections de fautes pour déterminer la robustesse des solutions (tests à données aléatoires et tests structurels) ; ensuite, en calculant les métriques relatives aux attributs de qualité ; finalement, en évaluant la performance des solutions sur le plan du temps moyen d'exécution.

6.3.1 Robustesse

Pour le test à données aléatoires dans l'étude de cas ATM, les milles nombres injectés dans l'unique paramètre de méthode sont vérifiés par une assertion. Cette validation permet de vérifier si le montant passé en paramètre peut être soustrait à l'objet courant. On considère

donc que le montant injecté (type `Money`) doit être inférieur ou égal au montant de l'objet courant. Comme le présente le tableau 6.2, aucun des nombres fautifs injectés dans le logiciel sans solution de détection et de confinement n'a été détecté. Bien qu'aucune erreur n'ait été provoquée, les entrées ont eu pour impact d'assigner une valeur invalide aux objets `Money` instanciés. Toutes ces fautes ont été détectées et confinées par l'adaptateur. Aucune défaillance n'a pu être observée sur l'adaptateur, où 477 fautes ont été injectées. Sur le proxy, les 484 fautes générées ont été détectées puis confinées adéquatement, sans provoquer ni propagation ni défaillance ; même résultat pour la solution aspect où 503 fautes ont été injectées.

Tableau 6.2
Répartition des fautes injectées dans les paramètres de méthode de *subtract*

	injectées	détectées → propagées	
Sans solution	539	0	539
Adaptateur	477	477	aucune
Proxy	484	484	aucune
Aspect	503	503	aucune

Dans JHotDraw, les cent nombres injectés dans le paramètre de méthode sont validés par une assertion vérifiant si le nombre est utilisé par la méthode, autrement dit si la faute devient active. Rappelons que la méthode est utilisée pour obtenir un nœud de la courbe de Bézier. Comme le paramètre de la méthode correspond à l'index d'un nœud, sa valeur doit être située entre zéro et la taille de la longueur du tableau. L'index correspond à la position dans le tableau d'objets `BezierPath` où sont stockés les nœuds. Les résultats observés lors de cette injection de fautes avec et sans les solutions sont présentés dans le tableau 6.3. Le nombre de fautes injectées correspond à la proportion de nombres invalides vis-à-vis des spécifications. Des cent nombres injectés sur le logiciel sans solution, 45 % des nombres étaient considérés comme invalides. Chacune de ces fautes a provoqué la levée d'une exception `IndexOutOfBounds` dans `BezierFigure`, gérée par le bloc `try-catch` du cas de test. Les nombres valides vis-à-vis des spécifications n'ont eu aucun effet sur le logiciel. Pour l'adaptateur, 43 nombres fautifs ont été injectées dans la méthode désignée. Chacune a été détectée comme fautive et l'exception `AssertionError` a été soulevée. Aucune défaillance ni propagation d'erreurs n'ont pu donc être observées. Pour le proxy, 46 nombres fautifs ont été injectées et toutes ont été gérées. 40 nombres fautifs ont été injectées dans la solution aspect et chacune d'elle a pu être détectée par l'advice et une exception a été levée.

Tableau 6.3
Répartition des fautes injectées dans les paramètres de méthode de *getNode*

	injectées	détectées → propagées	
Sans solution	45	0	45
Adaptateur	43	43	aucune
Proxy	46	46	aucune
Aspect	40	40	aucune

Comme les capacités anti-propagatrices des solutions n'ont pas pu être testées sur `getNode`, la même évaluation est répétée, mais sur une classe distincte. Un composant est à nouveau choisi selon des critères différents cette fois-ci. On procède en examinant les classes dépendantes de `BezierFigure`. La classe `BezierControlPointHandle` est choisie vu la facilité avec laquelle l'index paramétré dans le constructeur est traité dans l'une de ses méthodes. La classe représente la poignée manipulatrice d'un point de contrôle de la figure (étend `AbstractHandle`, laquelle implémente `Handle`). Comme toujours, une seule méthode est ciblée : `getBezierNode` (le code source est présenté à la figure 6.5). Lorsque cette classe est instanciée, l'index d'un nœud est passé dans les paramètres. Ce même index est utilisé par la méthode évaluée précédemment. Bien que la méthode n'accepte aucun paramètre, l'index passé dans le constructeur rend possible une injection de fautes. L'assertion conçue doit donc détecter si l'index est supérieur ou non à zéro puisque la méthode ne vérifie pas si l'index est négatif ou hors limite (longueur du tableau - 1).

```
@Nullable
public BezierPath.Node getBezierNode() {
   return getBezierFigure().getNodeCount() > index ?
     getBezierFigure().getNode(index) : null;
}
```

Figure 6.5 Code source de la méthode ciblée par l'évaluation

Les résultats observés lors de cette injection de fautes avec et sans les solutions sont présentés dans le tableau 6.4. Des 42 nombres fautifs générés et injectés dans le logiciel sans solution, aucune n'a pu être détectée. Toutes ont provoqué la levée d'une exception `IndexOutOfBounds` dans `BezierFigure` et ont été gérées par le bloc `try-catch` du cas de test. Chaque erreur a été propagée dans la séquence suivante : lors de l'invocation de `getBezierNode`, l'index passé dans le constructeur puis assigné à une variable de classe est passé à `getNode` dans `BezierFigure`; les fautes s'activèrent dans cette classe et créèrent des erreurs où une exception fût levée et une défaillance s'en suivit. Le même phénomène n'a pas été observé sur le logiciel implémentant l'adaptateur, où

69 fautes injectées ont été détectées et confinées adéquatement. La situation est identique pour le proxy et l'aspect qui ont respectivement détectés puis confinés de 68 et 70 fautes avec succès.

Tableau 6.4
Répartition des fautes injectées dans les paramètres de méthode de *getBezierNode*

	injectées	détectées → propagées	
Sans solution	42	0	42
Adaptateur	69	69	aucune
Proxy	68	68	aucune
Aspect	70	70	aucune

Pour l'injection de fautes effectuée au niveau du bytecode de JHotDraw, la faute injectée assigne une valeur négative à l'index (-1). Comme prévu, la faute n'a pas été détectée ni confinée dans le logiciel sans solution. La faute a provoqué la levée d'une exception `IndexOutOfBoundsException` dans `BezierFigure`, gérée par le bloc `try-catch` du cas de test. La faute injectée dans l'adaptateur a correctement été détectée et a ainsi provoqué la levée de l'exception `AssertionError`, toujours gérée par le cas de test. Le même phénomène a été observé dans le proxy dans une séquence d'événements identique. Pour l'aspect, la faute a été détectée par l'advice et a provoqué la levée de l'exception `AssertionError`.

6.3.2 Maintenabilité et testabilité

Le tableau 6.5 présente les métriques de code des classes des solutions appliquées sur les deux études de cas. Pour l'étude préliminaire, la classe `Money` est évaluée ; `BezierControlPointHandle` pour l'étude de cas JHotDraw. Les résultats ne comprennent pas les métriques d'interfaces puisqu'elles n'ont, pour la plupart, pas de correspondant objet. Pour la métrique F-OUT, seules les dépendances dans le logiciel sont considérées (les dépendances externes sont exclues). Les valeurs ont été arrondies à deux décimales près.

Tableau 6.5
Métriques de code source de JHotDraw

	Maintenabilité						Testabilité	
	CBO	F-IN	WMC	RFC	LCOM	MI V VG LOC	F-OUT	RFC
ATM								
Adaptateur	3	15	14	12	0	73,20 482 10 50	2	12
Proxy	2	15	13	9	0	84,05 199 10 34	1	9
JHotDraw								
Adaptateur	8	4	5	9	1	95,79 260 6 16	4	9
Proxy	6	3	7	12	1	86,72 385 8 24	4	12

La maintenabilité est envisagée sous l'angle de la complexité et du couplage. Pour le couplage : dans l'ATM, l'adaptateur et le proxy possèdent un indice *fan-in* de 15, quinze classes en sont donc dépendantes. Les dépendances sont moindres pour JHotDraw, où l'indice est de 4 pour l'adaptateur et de 3 pour le proxy. Le couplage entre objets (CBO) pour JHotDraw est de 8 pour l'adaptateur et de 6 pour le proxy, mais moindre dans l'ATM où CBO = 3 pour l'adaptateur et 2 pour le proxy. Selon la métrique de complexité WMC, l'adaptateur et le proxy pour ATM sont à presque égalité en termes de complexité (14 et 13, respectivement) ; pour JHotDraw, l'indice est de 5 pour l'adaptateur et 7 pour le proxy. La métrique de complexité RFC pour JHotDraw est de 9 pour l'adaptateur et 12 pour le proxy, et inversement pour l'ATM (12 et 9, respectivement). Les solutions sont parfaitement cohésives dans les deux études de cas, LCOM = 0. Finalement, l'indice de maintenabilité MI est élevé dans tous les cas, à l'exception de l'adaptateur de l'ATM (MI = 73). L'adaptateur et le proxy dans JHotDraw sont hautement maintenables puisque leur indice, respectivement de 96 et de 87, est supérieur au seuil de 85. L'indice pour le proxy dans l'ATM est de 84. La testabilité est quantifiée par le nombre de dépendances efférentes (F-IN) et par RFC. Les dépendances de l'adaptateur et du proxy sont au nombre de 4 dans JHotDraw, mais plus élevées dans ATM (F-IN = 15). Les valeurs pour RFC ont été énumérées un peu plus tôt.

6.3.3 Performance

Seule la performance des solutions dans JHotDraw est calculée. Les temps d'exécution moyens pour la méthode `getBezierNode` dans `BezierControlPointHandle` sont résumés dans le tableau 6.6. Les mesures sont exprimées en nanoseconde. Pour l'adaptateur, le temps

de traitement moyen de la méthode désignée est le plus bas des trois solutions, soit d'environ 42 ns par opération (ns/op). Ces prises de mesure sont similaires à celles où aucune solution n'est implantée. Le proxy admet un temps d'exécution légèrement plus élevé que l'adaptateur, soit de 57 ns/op avec une marge d'erreur d'environ 2,7 (la moitié de celle de l'adaptateur). Avec un score d'environ 100,6 ns/op, l'aspect admet une augmentation de 172 % du temps de traitement, avec une marge d'erreur de 16,7 (triple de celle de l'adaptateur).

Tableau 6.6
Temps d'exécution moyen de *getBezierNode*

	moyenne ns	médiane ns
Sans solution	37,001	2,072
Adaptateur	42,354	5,360
Proxy	57,113	2,736
Aspect	100,584	16,742

6.4 Discussion

Selon les résultats obtenus, les solutions ont toutes été en mesure d'empêcher la propagation des erreurs causées par les deux injections de fautes. L'objectif de l'évaluation a été d'évaluer les solutions de manière à déterminer l'efficacité des assertions exécutables comme moyen de détection des erreurs et l'efficacité de trois méthodes de confinement. L'évaluation s'est limitée à une classe en particulier, où une seule méthode a été retenue. Dans ce qui suit, les résultats obtenus lors de la première étude de cas viennent appuyer ceux de l'évaluation de JHotDraw.

Lors des injections des données aléatoires, toutes les solutions ont pu empêcher l'introduction de données invalides dans la méthode ciblée. Lorsque les assertions sont conçues strictement selon les spécifications du logiciel, celles-ci constituent un moyen de détection d'erreurs efficace, pour le logiciel évalué. Une assertion qui n'aurait pas été écrite selon les spécifications aurait pu occasionner une dégradation du taux de détection d'erreurs. La qualité des assertions repose donc sur une bonne compréhension des caractéristiques du logiciel. Les assertions agissent comme vérificateur d'intégrité sémantique des données, un type de vérification largement utilisée dans les conceptions tolérantes aux fautes (Dubrova, 2013). La couverture d'erreurs détectées dépend des mécanismes de détection et de confinement utilisés (Constantinescu, 1994). Bien que le code source du logiciel ait été accessible dans notre cas, les solutions pourront tout de même être appliquées sur un composant où le code est inaccessible. Il est possible

que, dans ce cas, une vérification de la conformité des variables du composant vis-à-vis des spécifications soit plus complexe.

Les entrées et les sorties de méthode ont été vérifiées. Toutes les fautes ont pu être détectées. La vérification des sorties n'a pas été utile puisque les erreurs ont été provoquées avant même la concrétisation d'un résultat. Ainsi, lors de l'évaluation sur la classe `BezierFigure`, aucune propagation n'a pu être observée. Les critères de sélection de méthode dans la méthodologie devraient être revus en conséquence afin de prendre en compte une méthode pouvant provoquer une propagation d'erreurs. Une méthode fortement couplée aurait facilité l'évaluation. Les solutions empêchent donc la propagation d'erreurs. Les mêmes résultats ont été observés lors de l'injection de fautes au niveau du bytecode : les fautes ont été détectées et confinées adéquatement par les trois méthodes de confinement. Les solutions ont donc été efficaces pour contrer la problématique qu'est la propagation d'erreurs.

Certains attributs de qualité ont été quantifiés à l'aide d'une variété de métriques orientées objet. Au meilleur de notre connaissance, aucune étude n'a évalué de telles solutions sous l'angle de la maintenabilité et de la testabilité. Seules des métriques orientées objet ont été utilisées puisqu'aucune métrique ne peut définir un aspect de manière adéquate. Bien que des métriques de maintenabilité aient été développées pour le paradigme aspect (Ceccato et Tonella, 2004), celles-ci négligent les caractéristiques propres au langage aspect, étant souvent adaptées de l'orienté objet (Burrows, Garcia et Taïani, 2010) : « même les métriques OA [orientés aspect] adaptées des métriques OO [orientés objet] validées empiriquement ne peuvent pas être des prédicteurs théoriquement valables de la maintenabilité (p. 288) ». Pour cette raison, aucune métrique n'a été utilisée pour quantifier la complexité et le couplage de la solution aspect. La maintenabilité de l'adaptateur et du proxy est envisagée selon des métriques de couplage, de complexité et de cohésion. La métrique CBO est à quasi-égalité dans les deux solutions, cependant l'adaptateur est légèrement plus couplé que le proxy (+33 %). Le couplage afférent (F-IN) ne révèle rien de particulier; on note seulement que la valeur pour l'adaptateur est légèrement plus élevée. La valeur pour le couplage efférent (F-OUT) est identique pour les deux solutions. La métrique LCOM définit le manque de cohésion dans les classes. Sa valeur est nulle dans les deux cas, signifiant que les solutions sont parfaitement cohésives. Finalement, l'indice de maintenabilité est situé au-dessus du seuil de haute maintenabilité (défini par Coleman, Ash, Lowther et Oman, 1994), ce qui signifie que l'adaptateur et le proxy sont autant maintenables l'un et l'autre. Une différence importante peut être observée entre l'indice de maintenabilité recueilli sur l'adaptateur dans JHotDraw et celui dans l'ATM : ce dernier se situe dans le seuil de

maintenabilité moyenne. Nous supposons l'héritage d'en être la cause puisque toutes les méthodes du composant adapté ont été redéfinies dans l'adaptateur, à la différence de l'adaptateur objet utilisé dans JHotDraw. Le nombre de lignes de code s'en trouve plus élevé (LOC est utilisé dans le calcul de l'indice MI). L'inconsistance de l'indice de maintenabilité remet toutefois en cause cette conclusion. Comme van Deursen (2014) le soulève : « [i]l n'y a pas d'explication claire sur la formule dérivée ». Ces doutes avaient d'ailleurs été confirmés ailleurs (Sjøberg, Anda et Mockus, 2012). Malgré la promotion faite pour adopter ce « bon prédicteur de la maintenabilité » (Bray et collab., 1997), la métrique a été développée en étudiant des systèmes de Hewlett-Packard écrits en C et en Pascal vers la fin des années 80 (Oman et Hagemeister, 1992). Les programmes écrits dans ce langage peuvent avoir des caractéristiques de maintenabilité différentes des programmes écrits en orienté objet. De ce point de vue, l'importance des résultats obtenus par cette métrique est à considérer, en particulier lorsque les décisions impliquent un système critique (pour définir la maintenabilité de la solution à long terme, par exemple). Cette métrique apporte toutefois une considération supplémentaire dans les prises de décision architecturales.

La performance des solutions a été évaluée selon le temps de traitement moyen par opération. Les mesures varient entre 42 et 100 nanosecondes. L'adaptateur admet une augmentation du temps de traitement d'environ 14,47 %. L'augmentation est plus significative dans les deux autres solutions, soit 54,36 % pour le proxy et 171,84 % pour l'aspect. Dans les deux premiers cas, la dégradation des performances du logiciel nous semble négligeable pour le gain de robustesse observé. La librairie de réflexion Java et le tisseur d'AspectJ induisent toutefois une augmentation plus significative du temps d'exécution. L'adaptateur est la solution la moins coûteuse, faisant d'elle une solution idéale lorsqu'on désire le moins de dégradations possibles dans les performances. À noter toutefois que la taille de l'échantillon ne permet pas une généralisation de ces résultats. Ainsi, il pourrait être intéressant d'augmenter le nombre d'expérimentations pour obtenir un échantillon assez large pour assumer une bonne performance des solutions sur les logiciels orientés objet. Concernant la testabilité, la métrique RFC étant en corrélation directe avec la métrique indiquant la taille d'une suite de test DNOTC (Bruntink, 2003; Bruntink et van Deursen, 2004) indique que plus de cas de tests devront être écrits pour tester le proxy que pour l'adaptateur. Les métriques F-OUT et RFC sont de bons prédicateurs de la taille de la suite de tests (Bruntink, 2003). L'adaptateur possède le plus faible niveau de testabilité des deux solutions. Cette solution requiert donc une plus grande compréhension de la part des développeurs, cela étant justifié par le fait qu'elle hérite de la complexité d'une classe dont il

est nécessaire de comprendre le fonctionnement. Autant l'adaptateur que le proxy ont nécessité un important temps de débogage pour implémenter les mécanismes de tolérance aux fautes.

L'utilisation d'un adaptateur comme méthode de confinement et des assertions exécutables comme moyen de détection d'erreurs a été proposée par Voas (1998) et subséquemment sous forme de design pattern par Saridakis (2003, § 5). La conception de l'adaptateur revient originalement aux auteurs de *Design Patterns* (Gamma, Helm, Johnson et Vlissides, 1994), dont la variante objet a été considérée initialement ici, mais écartée lors de l'implantation en raison de la complexité de la classe ciblée sur JHotDraw. C'est donc l'adaptateur basé sur l'héritage qui a plutôt été déployé. La solution s'en retrouve d'autant plus faiblement couplée lorsque cette dernière est déployée. La nouveauté de notre étude par rapport à ces deux études aura été, entre autres, de faire une évaluation de ce mécanisme de confinement comme une unité cohésive permettant de limiter la propagation d'erreurs.

Le proxy est également faiblement couplé avec les classes du logiciel, pour l'étude de cas utilisée. Un faible couplage est souhaitable puisque nous désirons qu'une modification future ait le minimum de risque d'affecter un autre composant. Ainsi, une diminution du nombre de connexions entre le composant protecteur et les autres composants peut signifier une diminution du risque d'impacts. L'utilisation de la réflexion est un facteur limitant son transfert vers d'autres plateformes, quoique Java ne soit pas le seul langage possédant une librairie d'introspection des classes. Un objet réflexif ne peut être testé de la même manière qu'un adaptateur, ajoutant ainsi un niveau de complexité lors de la phase de test. Pour cette solution, il a été nécessaire de modifier la définition des classes `BezierFigure` et `BezierControlPointHandle` afin d'y implémenter une classe. Le corps de ces classes n'a toutefois pas été altéré. En général, les méthodes de tolérance aux fautes logicielles impliquent l'ajout de code supplémentaire dans le logiciel. Ainsi, « [c]haque fois que du code est ajouté à un programme, il est possible d'introduire des fautes » (Leveson, Cha, Knight et Shimeall, 1990, p. 437). Les assertions exécutables pourraient donc introduire de nouvelles fautes ou des erreurs (utilisation d'une variable non initialisée, erreurs algorithmiques, boucles infinies ajoutées lors de l'instrumentation, référence de tableau hors limite, etc.), vu notre connaissance limitée des caractéristiques du logiciel évalué. Les assertions sont efficaces lorsque la nature des erreurs à détecter est connue. Par conséquent, nos solutions ne permettent pas de détecter les erreurs dans les spécifications d'un logiciel.

L'aspect est considéré comme un métaprocessus, d'une manière comparable au proxy. Ce processus donne la capacité au logiciel de s'examiner et de modifier son comportement à l'exécution. Aucune

difficulté n'a été rencontrée lors de l'implantation de la solution, contrairement aux deux autres. La programmation orientée aspect permet d'insérer du code dans un logiciel sans en altérer le code ou le comportement; le code original en est ainsi préservé (Afonso, Silva, Brito, Montenegro et Tavares, 2008). L'utilisation d'une extension orientée aspect permet d'instrumenter le logiciel en contribuant à la réutilisation du code; un même code peut être appliqué à d'autres projets avec des exigences de fiabilité différentes. À la différence des deux autres solutions, celle-ci ne pourra pas être testée facilement, car aucun outil n'est en mesure de tester les points de coupure dans un aspect (Delamare, Baudry, Ghosh, Gupta et Le Traon, 2011). De plus, le type d'advice utilisé dans la solution implique une augmentation significative du temps de traitement (Šuta, Martoš et Vranić, 2015).

CONCLUSION

La complexité des logiciels et la pluralité de leurs dépendances intercomposantes peuvent affecter leur fiabilité, en particulier leur capacité à tolérer les fautes. Dans un logiciel orienté objet, de telles dépendances impliquent un couplage pouvant entraîner la propagation d'erreurs d'un composant à un autre. Ce phénomène peut être résolu par l'implantation de solutions de détection et de confinement d'erreurs. Nous avons, dans un premier temps, présenté trois solutions dont le but était de limiter la propagation d'erreurs. Ces solutions ont successivement été implantées et évaluées dans un logiciel orienté objet Java. Les solutions proposées sont basées sur l'utilisation d'assertions exécutables comme moyen de détection d'erreurs et sur trois moyens de confinement d'erreurs distincts. La première solution est basée sur la conception du design pattern *Adaptateur;* la seconde emploie la réflexivité dans Java comme métaprocessus; la dernière utilise le tissage d'aspects. Deux types d'injections de faute ont été utilisés : le premier test a permis de déterminer la robustesse des solutions, en injectant des valeurs numériques aléatoires dans les paramètres d'une méthode; le second test a été effectué au niveau du bytecode, où une variable a été instrumentée. Selon les résultats, toutes les solutions ont été en mesure de détecter les fautes injectées adéquatement et ont empêché la propagation des erreurs. Lorsque les assertions sont développées selon les spécifications, ces solutions sont efficaces pour détecter et empêcher la propagation de fautes dans les données d'un logiciel orienté objet.

RÉFÉRENCES

Afonso, F., Silva, C., Brito, N., Montenegro, S. et Tavares, A. (2008). Aspect-oriented fault tolerance for real-time embedded systems. *AOSD workshop on Aspects, components, and patterns for infrastructure software,* Article no 2. doi : 10.1145/1404891.1404893

Aggarwal, K. K., Singh, Y., Kaur, A. et Malhotra, R. (2009). Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study. *Software Process : Improvement and Practice, 14*(1), 39–62. doi: 10.1002/spip.389

Anderson, T., Barrett, P. A., Halliwell, D. N. et Moulding, M. R. (1985). Software fault tolerance: An evaluation. *IEEE Transactions on Software Engineering, SE-11*(12), 1502–1510. doi: 10.1109/TSE.1985.231894

Anderson, T., Feng, M., Riddle, S. et Romanovsky, A. (2003). *Investigative case study: protective wrapping of OTS items in simulated environments* (Rapport no CS-TR-821). Newcastle upon Tyne: University of Newcastle upon Tyne. http://www.cs.ncl.ac.uk/publications/trs/papers/821.pdf

Andrews, D. M. (1979). Using executable assertions for testing and fault tolerance. *FTCS-9,* Madison, WI, 102–105. https://apps.dtic.mil/dtic/tr/fulltext/u2/a081488.pdf

Anwer, S., Adbellatif, A., Alshayeb, M. et Anjum, M. S. (2017). Effect of coupling on software faults: An empirical study. *2017 International Conference on Communication, Computing and Digital Systems (C-CODE),* Islamabad, 211–215. doi: 10.1109/C-CODE.2017.7918930

Apache Commons. (s. d.). Validate (Commons Lang 3.1 API). https://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/Validate.html

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C.,…, et Powell, D. (1990). Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering, 16*(2), 166–182. doi: 10.1109/32.44380

Arlat, J., Crouzet, Y., Deswarte, Y., Fabre, J.-C., Laprie, J.-C. et Powell, D. (2006). Tolérance aux fautes. Dans J. Akoka et I. Comyn-Wattiau (dir.), *Encyclopédie de l'informatique et des systèmes d'information* (partie 1, pp 240–270). Vuibert.

Arlat, J., Fabre, J.-C., Rodríguez, M. et Salles, F. (2002). Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers, 51*(2), 138–163. doi: 10.1109/12.980005

Autili, M., Di Salle, A., Gallo, F., Perucci, A. et Tivoli, M. (2015). Biological immunity and software resilience: Two faces of the same coin? *SERENE 2015: Software Engineering for Resilient Systems,* 1–15. doi: 10.1007/978-3-319-23129-7_1

Avizienis, A. (1985). The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering, SE-11*(12), 1491–1501. doi: 10.1109/TSE.1985.231893

———. (1997). Toward systematic design of fault-tolerant systems. *Computer, 30*(4), 51–58. doi: 10.1109/2.585154

Avizienis, A. et Kelly, J. P. J. (1984). Fault tolerance by design diversity: concepts and experiments. *Computer, 17*(8), 67–80. doi: 10.1109/MC.1984.1659219

Avizienis, A., Laprie, J.-C., Randell, B. et Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing, 1*(1), 11–33. doi: 10.1109/TDSC.2004.2

Barabási, A.-L. (2002). *Linked: The New Science of Networks.* Perseus Books Group.

Basili, V. R., Briand, L. C. et Melo, W. L (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(10), 751–761. doi: 10.1109/32.544352

Bass, L., Clements, P. et Kazman, R. (2002). *Software Architecture in Practice* (2e édition). http://www.ece.ubc.ca/~matei/EECE417/BASS/index.html

Beck, K. et Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. *OOPSLA' 89,* Nouvelle-Orléans, Louisiane. doi: 10.1145/74878.74879

Bjork, R. C. (2001). An example of object-oriented design: an ATM simulation [Logiciel]. http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

Black, P. E. et Singh, M. (2019). Opaque wrappers and patching : negative results. *Computer, 52*(12), 89–93, doi: 10.1109/MC.2019.2936071

Bruntink, M. (2003). *Testability of object-oriented systems: a metrics-based approach* (Mémoire de maîtrise, Universiteit van Amsterdam). https://homepages.cwi.nl/~paulk/theses/Bruntink.pdf

——— et van Deursen, A. (2004). Predicting class testability using object-oriented metrics. *Fourth IEEE International Workshop on Source Code Analysis and Manipulation,* Chicago, IL, 136–145. doi: 10.1109/SCAM.2004.16
Burrows, R., Garcia, A. et Taïani, F. (2010). Coupling metrics for aspect-oriented programming: a systematic review of maintainability studies. Dans L. A. Maciaszek, C. González-Pérez et S. Jablonski (dir.), *Evaluation of novel approaches to software engineering* (pp 277–290). Springer.

Carter, W. C. (1982). A time for reflection. *FTCS 12th annual International Symposium,* 41.
Carzaniga, A., Gorla, A. et Pezzè, M. (2009). Handling software faults with redundancy. Dans R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, M. ter Beek (dir.), *Architecting Dependable Systems VI* (pp 148–171). Springer.
Catal, C. et Diri, B. (2007). Software fault prediction with object-oriented metrics based artificial immune recognition system. Dans J. Münch et P. Abrahamsson (dir.), *Product-Focused Software Process Improvement. PROFES 2007. Lecture Notes in Computer Science* (vol 4589, pp 300–314). Springer.
Ceccato, M. et Tonella, P. (2004). Measuring the effects of software aspectization. *1st Workshop on Aspect Reverse Engineering (WARE 2004),* Delft, Pays-Bas. https://selab.fbk.eu/ceccato/papers/2004/ware2004.pdf
Chiba, S. (2019). Javassist (3.25.0.GA) [Logiciel]. https://www.javassist.org
Chidamber, S. R. et Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20*(6), 476–493. doi: 10.1109/32.295895
Chomsky, N. (1957). *Syntactic structures.* Mouton de Gruyter.
Clark, J. A. et Pradhan, Y. K. (1995). Fault injection : a method for validating computer-system reliability. *IEEE Computer, 28*(6), 47–56. doi: 10.1109/2.386985
Coleman, D., Ash, D., Lowther, B. et Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer, 27*(8), 44–49. doi: 10.1109/2.303623
Constantinescu, C. (1994). Estimation of coverage probabilities for dependability validation of fault-tolerant computing systems. *Proceedings of COMPASS'94 – 1994 IEEE 9th Annual Conference on Computer Assurance,* Gaithersburg, MD, 101–106. doi: 10.1109/CMPASS.1994.318463
Cunningham, D. W. (2012). *A logical introduction to proof.* Springer.

Delamare, R., Baudry, B., Ghosh, S., Gupta, S. et Le Traon, Y. (2011). An approach for testing pointcut descriptors in AspectJ. *Software Testing, Verification & Reliability, 21*(3), 215–239. doi: 10.1002/stvr.458
Deursen, A. van. (2014). Think twice before using the "maintainability index". https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/
Diaz, M., Juanole, G. et Courtiat, J.-P. (1994). Observer – A Concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering, 20*(12), 900–913. doi: 10.1109/32.368136
Downer, J. (2009). *When failure is an option: Redundancy, reliability and regulation in complex technical systems.* http://eprints.lse.ac.uk/36537/1/Disspaper53.pdf
Dowson, M. (1997). The Ariane 5 software failure. *Software Engineering Notes, 22*(2), 84. doi : 10.1145/251880.251992
Dubrova, E. (2013). *Fault-tolerant design.* Springer.

Eclipse Foundation. (2009). *AspectJ Frequently Asked Questions.* https://www.eclipse.org/aspectj/doc/released/faq.php#q:effectonperformance
———. (2019a). AspectJ [Logiciel]. https://www.eclipse.org/aspectj/
———. (2019b). Eclipse IDE for Eclipse Committers (2019-03) [Logiciel]. https://www.eclipse.org/downloads/packages/release/2019-09/r/eclipse-ide-eclipse-committers
Edwards, S. H., Sitaraman, M., Weide, B. W. et Hollingsworth, E. (2004). Contract-checking wrappers for C++ Classes. *IEEE Transactions on Software Engineering, 30*(11), 794–810. doi: 10.1109/TSE.2004.80

Fabre, J.-C., Nicomette, V., Perennou, T., Stroud, R. J. et Zhixue, W. (1995). Implementing fault tolerant applications using reflective object-oriented programming. *International Symposium on Fault-Tolerant Computing,* 489–498. doi: 10.1109/FTCS.1995.466949

Fabre, J.-C., Salles, F., Moreno, M. R. et Arlat, J. (1999). Assessment of COTS microkernels by fault injection. Dependable Computing for Critical Applications 7, 25–44. doi: 10.1109/DCFTS.1999.814288
Feinbube, L., Pirl, L. et Polze, A. (2018). Software fault injection : A practical perspective. Dans F. P. García Márquez et M. Papaelias (dir.), *Dependability Engineering* (pp 47–60). IntechOpen.
Fenton, N. et Pfleeger, S. (1997). *Software metrics: a rigorous and practical approach* (2e édition). PWS Publishing Co.
Fetzer, C. et Xiao, Z. (2003). HEALERS: a toolkit for enhancing the robustness and security of existing applications. *International Conference on Dependable Systems and Networks,* 317–322. doi: 10.1109/DSN.2003.1209942
Fetzer, J. H. (1999). The role of models in computer science. *The Monist, 82*(1), 20–36. doi: 10.5840/monist19998211
Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B. et Stata, R. (2002). Extended static checking for Java. *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation,* 234–245. doi: 10.1145/512529.512558
Floridi, L., Fresco, N. et Primiero, G. (2015). On malfunctioning software. *Synthese, 192*(4), 1199–1220. doi: 10.1007/s11229-014-0610-3
Floyd, R. W. (1967). Assigning meanings to programs. *Proceedings of Symposia in Applied Mathematics, 19,* 19–31. http://www.cs.tau.ac.il/~nachumd/term/FloydMeaning.pdf
Fontana, W. (2019, 24 octobre). *Le vivant et l'ordinateur : le défi d'une science de l'organisation* [vidéo]. Collège de France. https://www.college-de-france.fr/site/walter-fontana/inaugural-lecture-2019-10-24-18h00.htm
Fraser, T., Badger, L. et Feldman, M. (1999). Hardening COTS software with generic software wrappers. *IEEE Symposium on Security and Privacy,* 2–16. doi: 10.1109/SECPRI.1999.766713
Fresco, N. et Primiero, G. (2013). Miscomputation. *Philosophy & Technology, 26*(3), 253–272. doi: 10.1007/s13347-013-0112-0

Gamma, E., Helm, R., Johnson, R. et Vlissides, J. (1994). *Design patterns : elements of reusable object-oriented software.* Addison-Wesley.
Ghosh, A., Schmid, M. et Hill, F. (1999). Wrapping Windows NT Software for Robustness. *International Symposium on Fault-Tolerant Computing,* 344–347. doi: 10.1109/FTCS.1999.781070
Goldstine, H. H. et von Neumann, J. (1947). *Planning and coding of problems for an electronic computing instru- ment.* Institute for Advanced Study. https://library.ias.edu/files/pdfs/ecp/planningcodingof0103inst.pdf
Gyimothy, T., Ferenc, R. et Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering, 31*(10), 897–910. doi: 10.1109/TSE.2005.112

Halstead, M. H. (1977). *Elements of software science.* Elsevier.
Hanmer, R. S. (2007). *Patterns for fault tolerant software.* Wiley.
Harrison, W. (1994). Software measurement: a decision-process approach. *Advances in Computers, 39,* 51–105. doi: 10.1016/S0065-2458(08)60378-2
Hecht, H. (1976). Fault-tolerant software for real-time applications. *ACM Computing Surveys, 8*(4), 391–407. doi: 10.1145/356678.356681
Henry, S. et Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering, SE-7*(5), 510–518. doi: 10.1109/tse.1981.231113
Hiller, M. (2000). Executable assertions for detecting data errors in embedded control systems. *International Conference on Dependable Systems and Networks (DSN 2000),* New York, 24–33. doi: 10.1109/ICDSN.2000.857510
Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM, 12*(10). doi: 10.1145/363235.363259
Horner, J. K. et Symons, J. (2019). Understanding error rates in software engineering: Conceptual, empirical, and experimental approaches. *Philosophy & Technology, 32*(2), 363–378. doi: 10.1007/s13347-019-00342-1

Institute of Electrical and Electronics Engineers (IEEE). (1990). *IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990(R2002)).* doi: 10.1109/IEEESTD.1990.101064
———. (2010). *IEEE standard classification for software anomalies* (Std 1044TM-2009). https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5399061

Intellij. (2020). Intellij IDEA (2020.1.1) [Logiciel]. https://www.jetbrains.com/idea/
Isermann, R. (2006). *Fault-diagnosis systems.* Springer.

Jahanian, F., Rajkumar, R. et Raju, S. C. V. (1994). Runtime monitoring of timing constraints in distributed real-time systems. Real-Time Systems, 7, 247–273. doi: 10.1007/BF01088521
Jiang, M., Zhang, J., Raymer, D. et Strassner, J. (2007). A modeling framework for self-healing software systems. https://st.inf.tu-dresden.de/MRT07/papers/MRT07_Jiangl_etall.pdf
JUnit. JUnit (4.13) [Logiciel]. https://junit.org/junit4/

Karaorman, M. et Abercrombie, P. (2005). jContractor: Introducing design-by-contract to Java using reflective bytecode instrumentation. *Formal Methods in System Design, 27*(3), 275–312. doi: 10.1007/s10703-005-3400-1
Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., Akşit, M. et Matsuoka, S. (1997). Aspect-oriented programming. Dans M. Akşit et S. Matsuoka (dir.), *ECOOP' 97 – Object-Oriented Programming. Lecture Notes in Computer Science, 1241*, 220–242. doi: 10.1007/BFb0053381
Koopman, P. et De Vale, J. (1999). Comparing the robustness of POSIX operating systems. *International Symposium on Fault-Tolerant Computing.* https://users.ece.cmu.edu/~koopman/ballista/ftcs99/ftcs99.pdf
Kopetz, H. (1997). *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers.

Lala, J. H. et Harper, R. E. (1994). Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE, 82*(1), 25–40. https://www.cs.unc.edu/~anderson/teach/comp790/papers/safety_critical_arch.pdf
Laprie, J.-C. (1995). Dependable computing and fault tolerance: concepts and terminology. *International Symposium on Fault-Tolerant Computing,* 2–11. doi: 10.1109/FTCSH.1995.532603
Laprie, J.-C. (2008). From dependability to resilience. *IEEE/IFIP International Conference on Dependable Systems and Networks,* G8–G9. http://2008.dsn.org/fastabs/dsn08fastabs_laprie.pdf
Leavens, G. T. et Cheon, Y. (2006). *Design by contract with JML.* http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf
Lee, P. A. et Anderson, T. (1990). *Fault tolerance: principles and practice* (2e éd.). Springer.
Leijdekkers, B. (2020). MetricsReloaded (1.9) [Plugin]. https://plugins.jetbrains.com/plugin/93-metricsreloaded
Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. *LPAR 2010: Logic for Programming, Artificial Intelligence, and Reasoning,* 348–370. doi: 10.1007/978-3-642-17511-4_20
Leveson, N. G. (2004). Role of software in spacecraft accidents. *Journal of Spacecraft and Rockets, 41*(4), 564–575. doi: 10.2514/1.11950
———. (2008). *Medical devices: the Therac-25.* http://sunnyday.mit.edu/papers/therac.pdf
———, Cha, S. S., Knight, J. C. et Shimeall, T. J. (1990). The use of self checks and voting in software error detection: an empirical study. *IEEE Transactions on Software Engineering, 16*(4), 432–443. doi: 10.1109/32.54295
Leveson, N. G., Cha, S. S., Knight, J. C. et Shimeall, T. J. (1990). The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering, 16*(4), 432–443. doi: 10.1109/32.54295
Li, J., Chen, X., Huang, G., Mei, H. et Chauvel, F. (2009). Selecting fault tolerant styles for third-party components with model checking support. Dans G. A. Lewis, I. Poernomo et C. Hofmeister (dir.) *Component-Based Software Engineering. CBSE 2009. Lecture Notes in Computer Science* (vol 5582, pp 69–86). Springer. doi: 10.1007/978-3-642-02414-6_5
Lyu, M. R. (1996). *Handbook of software reliability.* McGraw-Hill.

Maes, P. (1987). Concepts and experiments in computational reflection. *OOPSLA '87,* Orlando, Floride, 147–155. doi: 10.1145/38807.38821
Mahmood, A., Andrews, D. M. et McCluskey, E. J. (1984). *Executable assertions and flight software* (Rapport no 84-258). Stanford, Californie : Center for Reliable Computing, Stanford University.
Mariani, L. (2003). A fault taxonomy for component-based software. *Electronic Notes in Theoretical Computer Science, 82*(6), 55–65. doi: 10.1016/S1571-0661(04)81025-9

McAllister, D. F. et Vouk, M. A. (1996). Fault-tolerant software reliability engineering. Dans M.R. Lyu (dir.), *Handbook of Software Reliability Engineering.* McGraw-Hill.
McCabe , T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320. doi: 10.1109/tse.1976.233837
Meyer, B. (1992). Applying "design by contract". *Computer, 25*(10), 40–51. doi: 10.1109/2.161279
Miller, B., Fredriksen, L. et So, B. (1989). An empirical study of the reliability of UNIX utilities. *Communications of the ACM, 33*(12), 33–44. doi: 10.1145/96267.96279
Mok, A. K. et Liu, G. (1997). Efficient runtime monitoring of timing constraints. *Proceeding of 3rd RealTime Technology and Applications Symposium,* Montreal. doi: 10.1109/RTTAS.1997.601363
Morris, F. L. et Jones, C. B. (1984). An early proof by Alan Turing. *Annals of the History of Computing, 6*(2), 139–143. doi: 10.1109/MAHC.1984.10017
Myers, W. (1986). Can software for the strategic defense initiative ever be error-free? *Computer, 19*(11), 61–67. doi: 10.1109/MC.1986.1663101

Naboulsi, Z. (2011). Code metrics – maintainability index [Billet de blogue]. https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/
Natella, R., Cotroneo, D. et Madeira, H. S. (2016). Assessing dependability with software fault injection: A survey. *ACM Computing Surveys, 48*(3), article no 44. doi: 10.1145/2841425
National Academy of Sciences (NAS). (1980). *Improving aircraft safety: FAA certification of commercial passenger aircraft.* https://www.nap.edu/catalog/557/improving-aircraft-safety
National Aeronautics and Space Administration (NASA). (2012). *Fault Management Handbook* (Rapport no NASA-HDBK-1002). https://www.nasa.gov/pdf/636372main_NASA-HDBK-1002_Draft.pdf

O'Madadhain, J. (2016). google/guava Wiki. https://github.com/google/guava/wiki
Olague, H. M., Etzkorn, L. H., Gholston, S. et Quattlebaum, S. (2007). Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering, 33*(6), 402–419. doi: 10.1109/TSE.2007.1015
Oracle. (s. d.-a). Dynamic proxy classes. https://docs.oracle.com/javase/7/docs/technotes/guides/reflection/proxy.html
———. (s. d.-b). Chapter 14. Blocks and statements. https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.10
———. (s. d.-c). Java™ Platform Standard Ed. 7. Package java.lang.instrument. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

Page-Jones, M. (1988). *Practical guide to structured systems design* (2e éd.). Prentice-Hall.
Pai, G. J. et Bechta Dugan, J. (2007). Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on Software Engineering, 33*(10), 675–686. doi: 10.1109/TSE.2007.70722
Perrow, C. (1984). *Normal accidents : living with high-risk technologies.* Basic Books.
Perry, D. E. et Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes, 17*(4), 40–52. doi: 10.1145/141874.141884
Piccinini, G. (2007). Computing mechanisms, *Philosophy of Science, 74*(4), 501–526. doi: 10.1086/522851
Pierce, B. C. (2002). *Types and programming languages.* MIT Press.
Popov, P., Riddle, S., Romanovsky, A. et Strigini, L. (2001). On systematic design of protectors for employing OTS items. *EUROMICRO Conference 2001,* 22–29. doi: 10.1109/EURMIC.2001.952434
Powell, D., Martins, E., Arlat, J. et Crouzet, Y. (1993). Estimators for fault tolerance coverage evaluation. *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing,* Toulouse, France, 228–237. doi: 10.1109/FTCS.1993.627326
Pressman, R. (2014). *Software engineering : a practitioner's approach* (8e édition). McGraw-Hill.
Pullum, L. L. (2001). *Software fault tolerance techniques and implementation.* Artech House.

Rabéjac, C., Blanquart, J.-P. et Queille, J.-P. (1996). Executable assertions and timed traces for on-line software error detection. *Annual Symposium on Fault Tolerant Computing,* Sendai, Japon, 138–147. doi: 10.1109/FTCS.1996.534602
Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering, SE-1*(2), 220–232. doi: 10.1109/TSE.1975.6312842

Randelshofer, W. (2010a). JHotDraw (7.6) [Logiciel]. https://sourceforge.net/projects/jhotdraw/
———. (2010b). JHotDraw (7.6) [Code source]. https://sourceforge.net/p/jhotdraw/git/ci/master/tree/jhotdraw7/
Riaz, M., Mendes, E. et Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. *International Symposium on Empirical Software Engineering and Measurement,* Lake Buena Vista, FL, 367–377. doi: 10.1109/ESEM.2009.5314233
Riehle, D. (2007). *Case study: the JHotDraw framework*. https://riehle.org/computer-science/research/dissertation/chapter-8.html
Rodríguez, M., Fabre, J.-C. et Arlat, J. (2002). Wrapping real-time systems from temporal logic specifications. *Dependable Computing EDCC-4,* 253–270. doi: 10.1007/3-540-36080-8_22
Rosenblum, M., Chapin, J., Teodosiu, D., Devine, S., Lahiri, T. et Gupta, A. (1996). Implementing efficient fault containment for multiprocessors. *Communications of the ACM, 39*(9), 52–61. doi : 10.1145/234215.234471
Rushby, J. (1999). *Partitioning in avionics architectures: Requirements, mechanisms, and assurance* (Publication no NASA/CR-1999-209347). Repéré sur le site de la NASA : https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19990052867.pdf

Saib, S. H. (1977). Executable assertions – an aid to reliable software. *Asilomar Conference on Circuits, Systems and Computers.* doi : 10.1109/ACSSC.1977.748932
Salles, F., Rodriguez, M., Fabre, J.-C. et Arlat, J. (1999). Metakernels and fault containment wrappers. *International Symposium on Fault-Tolerant Computing.* doi: 10.1109/FTCS.1999.781030
Saridakis, T. (2003). Design patterns for fault containment. *EuroPLoP Conference.* http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2003/2003_Saridakis_DesignPatternsforFaultContainment.pdf
Savor, T. et Seviora, R. E. (1997). An approach to automatic detection of software failures in real-time systems. *Proceedings 3rd IEEE Real-Time Technology and Applications Symposium.* 136–146. doi: 10.1109/RTTAS.1997.601351
Schneider, F. B. (1998). *Enforceable security policies* (Rapport no TR98-1664). Ithaca, New York : Department of Computer Science, Cornell University.
Sedgewick, R. et Wayne, K. (2014). *Algorithms* (4e édition, partie 1). Addison-Wesley.
Shah, V. et Hill, F. (2003). An aspect-oriented security framework. *DARPA Information Survivability Conference and Exposition,* 143–145. doi: 10.1109/DISCEX.2003.1194952
Shaheen, M. R. et Du Bousquet, L. (2010). *Survey of source code metrics for evaluating testability of object oriented systems* (Rapport no RR-LIG-005). https://hal.inria.fr/hal-00953403/document
Shaw, M. et Clements, P. (1997). A field guide to boxology: preliminary classification of architectural styles for software systems. *International Computer Software and Applications Conference (COMPSAC'97).* doi: 10.1109/CMPSAC.1997.624691
Singh, Y., Kaur, A. et Malhotra, R. (2010). Empirical validation of object-oriented metrics for predicting fault proneness models. *Software Quality Journal, 18*(3), 3–35. doi: 10.1007/s11219-009-9079-6
Sjøberg, D. I. K., Anda, B. et Mockus, A. (2012). Questioning software maintenance metrics : a comparative case study. *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement,* 107–110. doi: 10.1145/2372251.2372269
Spitzer, C. R., Ferrell, U. et Ferrell, T. (dir.) (2015). *Digital avionics handbook* (3e édition). CRC Press.
Sommerville, I. (2016). *Software engineering* (10e éd.). Pearson.
Sprunck, M. (s. d.). Comparison of Ways to Check Preconditions in Java – Guava vs Apache Commons vs Spring Framework vs Plain Java Asserts. http://www.sw-engineering-candies.com/blog-1/comparison-of-ways-to-check-preconditions-in-java
Stroph, R. et Clarke, T. (1998). Dynamic acceptance tests for complex controllers. *24th EUROMICRO Conference,* Vasteras, Suède, 411–417. doi: 10.1109/EURMIC.1998.711834
Šuta, E., Martoš, I et Vranić, V. (2015). Usability of AspectJ from the performance perspective. *IEEE 1st International Workshop on Consumer Electronics,* Novi Sad. doi : 10.1109/CEWS.2015.7867162

Tudose, C., Odubăşteanu C. et Radu S. (2013). Java reflection performance analysis using different Java development. Dans L. Dumitrache (dir.), *Advances in Intelligent Control Systems and Computer Science* (vol 187, pp 439–452). doi: 10.1007/978-3-642-32548-9_31
Turing, A. (1950). Computing machinery and intelligence. *Mind, 59*(236), 433–460. doi: 10.1093/mind/LIX.236.433

Venema, W. (1992). *TCP wrapper : Network monitoring, access control, and booby traps.* http://student.ing-steen.se/security/document/tcp-wrapper.pdf

Voas, J. M. (1998). Certifying off-the-shelf software components. *Computer, 31*(6), 53–59. doi: 10.1109/2.683008

Von Neumann, J. (1951). The general and logical theory of automata. Dans A. H. Taub (dir.), *John von Neumann Collected Works* (vol V, pp 288–328). Pergamon Press.

———. (1956). Probabilistic logics and the synthesis of reliable organisms from unreliable components. Dans A. H. Taub (dir.), *John von Neumann Collected Works* (vol V, pp 329–378). Pergamon Press.

White, I. (2000, avril). Wrapping the COTS dilemma. *Commercial Off-the-shelf Products in Defence Applications "The Ruthless Pursuit of COTS" IST Symposium,* Bruxelles, Belgique. http://www.dtic.mil/dtic/tr/fulltext/u2/p010659.pdf

Xiao, Q., Li, K., Zhang, D. et Xu, W. (2018). Security risks in deep learning implementations. *2018 IEEE Security and Privacy Workshops*, San Francisco, pp 123–128. doi: 10.1109/SPW.2018.00027

Zhou, Y. et Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering, 32*(10), 771–789. doi: 10.1109/TSE.2006.102

ANNEXE
CODE SOURCE DE L'AGENT JAVA

L'agent est composé de deux classes : l'agent et le transformateur. Dans le premier, `premain()` et `agentmain()` initient le processus d'instrumentation. Comme une seule des deux ne sera exécutée (l'exécution dépend du moment où l'agent est attaché à la JVM : à l'exécution ou lorsque l'exécution a déjà eu lieu) et puisque l'agent est rattaché au logiciel à l'exécution, `premain()` est invoquée.

Un objet `Instrumentation` du package de Java permet à l'agent d'instrumenter le logiciel, notamment en donnant un accès au bytecode des classes. L'objet et le nom de la classe invoquée sont passés à `transformClass()`, qui tente d'obtenir le *class loader* de la classe invoquée. Le traitement est ensuite délégué à `transform()` qui instancie un transformateur avec l'objet et la classe invoquée.

Le transformateur hérite de l'interface `ClassFileTransformer`. La modification du code source est effectuée dans `transform()`. Comme la méthode ciblée par l'évaluation n'accepte aucun paramètre, c'est plutôt le second constructeur de cette classe qui est instrumenté. Le constructeur est défini comme une instance de `CtConstructor`. Une valeur négative est affectée à `handle.index` lorsqu'une instanciation de cette classe a lieu.

```
public class Agent {

   private static String className = "org.jhotdraw.draw.
    handle.BezierControlPointHandle";

   public static void premain(String agentArgs,
    Instrumentation inst) {
      transformClass(className, inst);
   }

   public static void agentmain(String agentArgs,
    Instrumentation inst) {
      transformClass(className, inst);
   }

   private static void transformClass(String className,
   Instrumentation instrumentation) {
      Class<?> targetClass = Class.forName(className);
      ClassLoader targetClassLoader = targetCls.
       getClassLoader();

      transform(targetCls, targetClassLoader, instrumentation);
      return;
   }

   private static void transform(Class<?> class, ClassLoader
   classLoader, Instrumentation instrumentation) {
      Transformer transformer = new Transformer(class.
       getName(), classLoader);

      instrumentation.addTransformer(transformer, true);

      try {
         instrumentation.retransformClasses(class);
      } catch (Exception ex) {
         throw new RuntimeException();
      }
   }

}
```

```
public class Transformer implements ClassFileTransformer {

   private String TARGET_METHOD = "getBezierNode";
   private String targetClassName;
   private ClassLoader targetClassLoader;

   @Override
   public byte[] transform(ClassLoader loader, String
    className, Class<?> classBeingRedefined, ProtectionDomain
    protectionDomain, byte[] classfileBuffer) {
     byte[] byteCode = classfileBuffer;
     String finalClassName = targetClassName.replaceAll(
      "\\.", "/");

     if (!className.equals(finalClassName))
        return byteCode;

     if (className.equals(finalClassName) && loader.
      equals(targetClassLoader)) {

        try {
           ClassPool cp = ClassPool.getDefault();
           CtClass cc = cp.get(targetClassName);
           CtConstructor ctConstructor = ctClass.
            getConstructors()[1];

           ctConstructor.insertAfter("handle.index = -1;");

           byteCode = cc.toBytecode();
           cc.detach();
        } catch (NotFoundException | CannotCompileException
         | IOException e) {
           throw new RuntimeException();
        }
     }

     return byteCode;
   }

}
```