9

Les bases de la programmation par contrat

L'idée de la *programmation par contrat* est de signer un contrat entre l'utilisateur d'une fonction et la fonction elle-même par lequel la fonction s'engage, si l'utilisateur lui donne quelque chose de valide, à fournir à l'utilisateur un résultat valide, cohérent, et clairement déterminé. Autrement dit, nous pourrions dire que la programmation par contrat se résume, en simplifiant cependant beaucoup, à :

> Donne-moi ce que je veux, tu obtiendras ce que tu attends.

Les éventuels problèmes surviennent lorsque, pour une raison ou pour une autre, la fonction appelée ne reçoit pas ce qu'elle voulait ou – car cela arrive aussi – lorsque la fonction appelante ne récupère pas le résultat qu'elle attendait. Il y a alors rupture de contrat, et il n'y a dès lors plus aucune garantie quant au résultat obtenu.

La programmation par contrat prévoit trois aspects qui peuvent être vérifiés séparément :

- les préconditions ;
- les postconditions ;
- les invariants.

Chaque aspect poursuit un but bien déterminé afin de s'assurer que le contrat est exécuté correctement et mène à une bonne fin.

9. Préconditions et postconditions

Lorsque deux parties signent un contrat, elles se mettent d'accord sur un ensemble d'obligations que chaque partie se doit de respecter.

Si l'une des parties ne respecte pas ses obligations, il y a rupture de contrat et c'est alors que les problèmes commencent. Dans la vie réelle, cela se traduit souvent par des procédures légales dans lesquelles chacun essaye de rejeter la faute sur l'autre afin d'être considéré "dans son droit" et d'obtenir la réparation à laquelle la partie lésée estime avoir droit.

En dehors de la procédure légale intentée par la partie lésée, il en va finalement strictement de même lorsque l'on parle de programmation par contrat : il y a bel et bien deux parties en présence – l'utilisateur d'un type particulier ou d'une fonction d'une part et le type particulier ou la fonction de l'autre – qui sont liés par un ensemble d'obligations réciproques.

Les obligations de l'utilisateur sont de veiller à fournir des données correctes au type ou à la fonction qu'il utilise, alors que les obligations du type ou de la fonction utilisé(e) sont de fournir un service ou un comportement correct.

Il est forcément possible de vérifier les données fournies par l'utilisateur pour s'assurer qu'elles correspondent à ce qui est attendu, tout comme il est forcément possible de vérifier le résultat d'un service ou d'un comportement afin de s'assurer que "ce qui devait être fait l'a été dans les règles de l'art".

Les obligations imposées à l'utilisateur lorsqu'il décide de faire appel à un comportement sont appelées *préconditions*, alors que les obligations imposées au comportement observé après son exécution s'appellent les *postconditions*. Les préconditions doivent impérativement être respectées au plus tard au début du processus de traitement. Les postconditions doivent impérativement être respectées au plus tard lorsque le processus de traitement s'achève.

De la cohérence dans les données

En un mot comme en cent, on peut dire que le but intrinsèque des préconditions et des postconditions est de s'assurer (pour les préconditions) ou de garantir (pour les postconditions) que les données seront dans un état cohérent par rapport à l'utilisation qui en sera faite.

Le but est toujours de s'assurer que le système, de manière générale, est dans un état cohérent avant d'entreprendre n'importe quelle manipulation et qu'il sera encore dans un état cohérent une fois que la manipulation a été effectuée.

10. Invariant ? kesako ?

S'il est très facile de comprendre ce que représente une précondition ("donne-moi ce que je veux") ou une postcondition ("tu auras ce à quoi tu t'attends"), il est peut-être plus compliqué de comprendre la notion d'invariant.

Un *invariant* est une propriété (au sens le plus large possible du terme) qui, dans un contexte donné, doit pouvoir être vérifiée en permanence.

Par exemple, un carré présente trois invariants, ou trois propriétés qui doivent être vérifiables (et vérifiées) à n'importe quel moment :

- Il doit avoir quatre côtés : s'il n'a pas quatre côtés, ce sera, selon le nombre de côtés qu'il a, un triangle, un pentagone, un hexagone, un octogone ou n'importe quoi sauf un carré.
- Les quatre côtés doivent être égaux : s'ils ne sont pas égaux, ce sera selon le cas un rectangle, un trapèze ou même simplement un quadrilatère, mais sûrement pas un carré.
- Les quatre angles qui le composent doivent être des angles droits : si les quatre angles qui le composent ne font pas 90° (mais que les quatre côtés sont égaux), ce sera au mieux un losange ou sinon un trapèze, un parallélogramme voire simplement un quadrilatère.

Ces trois propriétés ne doivent pas être vérifiées uniquement avant (après) un déplacement ou un changement de taille : elles doivent être vérifiées en toutes circonstances.

Cet exemple montre parfaitement qu'un invariant peut parfaitement être constitué de valeurs (représentant en l'occurrence le nombre de côtés, l'angle formé par deux côtés contigus et la taille de chacun des côtés), qui peuvent parfaitement ne pas être des valeurs constantes (comprenez: immuables) : la taille des côtés en elle-même n'a aucune importance, c'est bel et bien le fait que les quatre côtés ont systématiquement la même taille qui représente l'invariant.

En termes de conception orientée objet, vous avez grandement intérêt à considérer que toute information qui est disponible au travers de l'interface publique (comprenez : tout ce qui se trouve dans l'accessibilité publique) de vos objets correspond à un invariant.

Comprenez par là qu'il s'agit de propriétés prouvables – au sens du principe de substitution de Liskov – dans la mesure où tout ce qui apparaît dans l'accessibilité publique d'un objet doit être assimilé à une propriété du genre de "expose la valeur XXX" pour les variables publiques ou "expose le service YYY" pour les fonctions membres publiques – et que ces propriétés prouvables sont, d'office, des invariants. Si vous retirez l'une de ces propriétés prouvables de l'interface publique de votre objet, vous obtiendrez, au mieux, un type d'objet tout à fait différent, au pire, un type d'objet totalement (ou du moins partiellement) inutilisable, car incapable de rendre l'ensemble des services que l'on est en droit d'attendre de sa part.

Cependant, cette approche est particulièrement restrictive, car tout membre d'une classe, toute fonction membre quelle que soit son accessibilité, est en réalité un invariant.

11. Un développeur est coupable, mais lequel ?

Le non-respect d'une précondition ou d'une postcondition mène systématiquement à une rupture de contrat. Il y a d'office l'une des parties qui n'obtient pas ce à quoi elle s'attend et le résultat est sans appel : tout peut arriver.

Vient alors l'épineuse question de savoir qui est responsable de la rupture du contrat entre le développeur qui utilise un comportement, que j'appellerai *utilisateur* dans ce chapitre, et celui qui a développé le comportement en question, que j'appellerai – logiquement – *développeur*.

En ce qui concerne les préconditions, il ne faut pas chercher bien loin. C'est forcément l'utilisateur qui est en cause : au moins à un endroit dans le code, il aura oublié quelque chose – quand bien même il s'avérerait particulièrement ardu de localiser l'erreur.

Les oublis peuvent être de différentes natures : il peut avoir oublié de tester une valeur pour s'assurer qu'elle était cohérente ; de transmettre une information importante à "qui de droit" (par exemple, de transmettre à l'utilisateur d'un pointeur que l'objet pointé par ce dernier a été détruit) ; il peut avoir omis de placer un invariant suffisant sur son objet, le cas classique étant de n'avoir pas pris en compte qu'un objet ayant sémantique d'entité ne peut décemment pas être copiable ; il peut avoir oublié d'appliquer une règle particulière comme la règle des trois grands pour les types ayant sémantique de valeur, etc. J'en passe, et très certainement des meilleures. Mais le résultat sera systématique et sans appel : la chose oubliée par l'utilisateur lui reviendra comme un boomerang en pleine figure, et de préférence au pire moment qui soit.

En revanche, les choses ne sont pas forcément aussi claires lorsqu'il s'agit des postconditions. En effet, le non-respect d'une postcondition implique simplement qu'il y a "quelque chose" qui n'a pas été correctement fait. La bonne question est alors de savoir quoi et pourquoi.

Dans certains cas, cela peut provenir d'une erreur de la part de l'utilisateur qui n'a – d'une manière ou d'une autre – pas suffisamment pris en compte qu'une précondition n'était pas respectée. L'explication est simple : si dès le départ une précondition n'est pas respectée, et que les données ne sont pas dans un état cohérent, le contrat est rompu avant même que le comportement ne soit observé. Il n'y a donc aucune raison pour que le comportement respecte sa part du contrat et fournisse le résultat auquel l'utilisateur aurait été en droit de s'attendre si le contrat avait été respecté d'emblée.

En revanche, si les préconditions imposées pour un comportement donné sont respectées, la responsabilité du non-respect des postconditions incombera forcément au développeur de ce comportement.

Peu importe le contexte (manque de mémoire faisant échouer une allocation dynamique, réseau défaillant empêchant de se connecter à un serveur, impossibilité de lire un fichier verrouillé par le système d'exploitation, etc.), si le contrat n'est pas respecté au plus tard lorsque le comportement s'achève, ce sera forcément la faute du développeur de ce comportement parce qu'il n'a simplement pas pris *toutes les précautions nécessaires* pour assurer la bonne fin du contrat. Sans oublier les erreurs de programmation qu'il peut avoir faites : il peut avoir codé, par exemple, une fonction sinus qui peut renvoyer une valeur supérieure à 1 ou avoir commis une erreur de logique dans une fonction de tri.

Cela peut éventuellement dédouaner le développeur qui utilise le comportement en question.

12. Invariants *structurels* et invariants *contextuels*

Il ne me semble pas que les termes de la programmation par contrat indiquent la différence, mais il est possible de distinguer deux grandes familles d'invariants : les invariants que je qualifierais de *structurels* d'une part et ceux que je qualifierais de *contextuels* d'autre part.

Les invariant structurels sont des invariants qui existent du seul fait de la nature même du type observé. Le fait qu'un carré ait quatre côté égaux est un invariant structurel car si vous retirez cet invariant, vous aurez sans doute beaucoup de formes différentes, mais ce ne seront pas forcément des carrés.

Les invariants contextuels sont des invariants qui n'existent que dans un contexte bien particulier. Lorsque vous décidez d'utiliser un paramètre transmis par référence constante, par exemple, vous placez *de facto* un invariant qui n'a strictement rien à voir avec la nature même de l'objet transmis en paramètre, mais qui sera systématique à chaque fois que vous serez dans un contexte correspondant à l'appel de la fonction en question : l'état de l'objet ne pourra être modifié entre l'appel de la fonction et le moment où la fonction s'achève. Et tout ce qui pourra se passer entre ces deux instants clés devra respecter la règle de l'invariant : on ne modifie pas l'état de l'objet sur lequel on travaille. Le fait de déclarer une fonction membre comme étant constante est bien sûr également un invariant contextuel : durant toute la durée de l'exécution de la fonction membre constante, vous ne pouvez pas modifier l'état de l'objet au travers duquel vous avez invoqué cette fonction.