Conception et programmation par contrat et traitement des erreurs

G. Tremblay

Été 2005

> *For production software, garbage in, garbage out isn't good enough. A good program never puts out garbage, regardless of what it takes in. A good programs uses "garbage in, nothing out", "garbage in, error message out", or "no garbage allowed in" instead. By today's standards, "garbage in, garbage out" is the mark of a sloppy nonsecure program.*
>
> *S. McConnell [McC04, p. 188]*

> *A useful technique is to add code to handle "can't happen" cases, situations where it is not logically possible for something to happen but (because of some failure elsewhere) it might anyway. [...] Defensive programming [is] making sure that a program protects itself against incorrect use or illegal data. Null pointers, out of range subscripts, division by zero, and other errors can be detected early and warned about or deflected.*
>
> *B. Kernighan et R. Pike [KP01, p. 142]*

Deux caractéristiques importantes d'un programme sont la justesse (*correctness*) et la robustesse [McC04] :

- Correctness *means never returning an inaccurate result; returning no result is better than returning an inaccurate result.*
- Robustness *means always trying to do something that will allow the software to keep operating, even if that leads to results that are inaccurate sometimes.*

Dans ce chapitre, nous allons traiter des différentes façons de s'assurer tant de la justesse que de la robustesse d'un programme.

1 La conception et programmation par contrat — ou — les assertions et leur utilisation

1.1 La notion de contrat

Un élément clé pour le développement d'un programme ou composant logiciel fiable et correct est la présence d'une spécification. En fait, la propriété d'un programme d'être correct ou non est une propriété qui, essentiellement, est *relative*. Plus précisément, on peut dire d'un programme qu'il est correct uniquement en faisant référence (donc *relativement*) à la spécification qu'il doit satisfaire : si le comportement du composant logiciel est celui décrit par sa spécification, alors le logiciel est correct, autrement il est incorrect.

La spécification d'un service fourni par un composant logiciel peut s'exprimer à l'aide de pré-conditions et post-conditions (on dit aussi *antécédents* et *conséquents*). Des spécifications

exprimées à l'aide de telles pré/post-conditions représentent une forme de *contrat*. Plus précisément, une spécification d'un service est un contrat entre le client du service et le fournisseur de ce service. Un contrat entraîne nécessairement certaines obligations, mais permet aussi certains bénéfices, certains avantages. Un contrat, comme un contrat dans le monde légal donc, *protège* les deux parties, puisqu'il rend explicite les obligations et bénéfices de chacun.

	Obligations	**Bénéfices**
Client	Doit satisfaire la pré-condition	Est assuré que la post-condition sera satisfaite
Fournisseur	Doit satisfaire la post-condition	Est assuré que la pré-condition sera satisfaite

Figure 1: Obligations et bénéfices du client et du fournisseur d'un service

Dans un contexte de développement de logiciels, les obligations et bénéfices associés à une spécification sont ceux présentés à la Figure 1. En d'autres mots, donc :

- La pré-condition est un bénéfice pour le fournisseur du service (qui n'est pas obligé de vérifier la condition associée et de faire un traitement spécial si la condition n'est pas satisfaite) et une obligation pour le client (qui doit s'assurer de respecter la condition sur les arguments avant de faire appel au service) ;
- La post-condition est un bénéfice pour le client (une condition appropriée est satisfaite sur le résultat) et une obligation pour le fournisseur (qui doit mettre en oeuvre le service de façon à produire un résultat qui satisfait la post-condition).

Comme l'indique Hunt et Thomas [HT00], l'approche par contrat devrait conduire un fournisseur de service à être "*paresseux*" dans ses offres de service :

- Une routine devrait être *stricte* sur ce qu'elle est prête à accepter (pré-condition forte).
- Une routine devrait en promettre le moins possible sur ce qu'elle va retourner comme résultat (post-condition faible).

Ces notions de contrats spécifiés à l'aide de pré/post-conditions, qui sont à la base des méthodes formelles de spécifications basées sur la modélisation abstraite, peuvent aussi s'étendre et s'appliquer aux étapes de conception et de constructions des logiciels — c'est ce qu'on appelle la *conception par contrat* (DBC = *Design By Contract*), introduite par Bertrand Meyer [Mey92], le concepteur du langage Eiffel [Mey97].

Les notions de contrats et contraintes associées peuvent être appliquées à l'étape de construction (codification et programmation, tests unitaires) en réalisant que si, durant l'exécution d'un programme, il arrive qu'une pré-condition ou qu'une post-condition ne soit pas vérifiée, alors c'est qu'il y a une *erreur* (un *bogue*) *dans le programme*.

Plus précisément :

- La violation d'une pré-condition est le signe d'une erreur dans le code *du client* : le client ne s'est pas assuré avant l'appel du service, comme il aurait dû le faire, que ses arguments étaient valides étant donné le service qu'il désirait utiliser. L'exécution du service demandé *ne devrait donc pas* se faire.
- La violation d'une post-condition est le signe d'une erreur dans le code *du fournisseur* : le fournisseur n'a pas été capable de fournir le service demandé. Si la situation est telle qu'il soit impossible de corriger cette erreur, alors l'exécution du programme devrait être avortée.

Dans la prochaine section, nous verrons comment le langage C permet de supporter l'approche de conception et programmation par contrat, donc permet de spécifier le comportement attendu d'un composant logiciel et, surtout, de *vérifier en cours d'exécution* que le comportement obtenu est bien celui attendu.

1.2 Les assertions en C = la macro `assert`

Le langage C supporte de façon simple l'approche de conception par contrat et l'utilisation d'assertions. En fait, une seule et unique construction est disponible, à savoir la macro `assert`, qu'on rend disponible à l'aide de l'inclusion de fichier suivante :[1]

```
#include <assert.h>
```

Ce fichier d'interface n'exporte qu'une seule opération, laquelle reçoit en argument une expression booléenne :[2]

```
void assert( int expression );
```

```
#include <assert.h>

main()
{
  int x = 0;

  assert( x != 0 );

  printf( "x = %d\n", x );
}
```

Extrait de code 1: Exemple simple d'utilisation de la macro `assert` en C

L'exécution de l'instruction `assert(exp)` durant l'exécution du programme évaluera donc l'expression `exp` : si le résultat obtenu est vrai (non nul), alors l'exécution se poursuit ; autrement, l'exécution du programme se termine (avorte) en produisant un message d'erreur qui indique la source du problème : nom du fichier, nom de la routine, numéro de ligne et forme de l'assertion. Par exemple, supposons que l'on ait l'extrait de code 1. L'exécution de ce programme (compilé dans le fichier `a.out`) générera alors un message d'erreur de la forme suivante :

```
a.out: test.c:7: main: Assertion 'x != 0' failed.
```

En d'autres mots, comme l'indique Maguire [Mag95][3] : "`assert` est uniquement une macro de débogage qui fera capoter l'exécution lorsque l'argument est faux".

En C, l'utilisation d'une macro plutôt que d'une fonction ordinaire confère deux avantages clés :

1. Les messages d'erreurs qui sont générés peuvent inclure le nom du fichier où le problème est survenu de même que le numéro de ligne (par le biais de références aux identificateurs `__FILE__` et `__LINE__`).
2. Il est facile, si désiré, de supprimer la vérification des assertions dans le code final en redéfinissant la macro.

```
#ifdef  NDEBUG
  # define assert(expr) 0
#else
  # define assert(expr) \
    (expr) \
    ? 0    \
    : __assert_fail(__STRING(expr), __FILE__, __LINE__, __ASSERT_FUNCTION)
#endif
```

Extrait de code 2: Définition de la macro `assert` en C

Plus précisément, si la variable `NDEBUG` est définie au moment où le programme est compilé[4], alors aucun code de vérification des assertions ne sera généré. En gros, la macro `assert` est donc définie (sur Linux) telle qu'illustrée dans l'extrait de code 2. La macro `__assert_fail` est celle qui génère le message d'erreur approprié (indiquant l'expression spécifiée dans le `assert` avec le nom du fichier et le numéro de ligne) puis qui avorte l'exécution (avec `abort()`).

Soulignons qu'il existe aussi des outils qui permettent, en C, d'utiliser une approche comme celle disponible en Java (commentaires spéciaux et pré-processeur). Ainsi, l'outil APP [Ros95] permet d'écrire la spécification d'une fonction calculant la racine carrée d'un nombre de la façon suivante :

```
float racineCarree( float x, float precision )
/*@
   assume (x >= 0.0) && (precision > 0.0);
   return res where (res >= 0.0) && (abs((res * res) - x) <= precision);
  @*/
{ ... }
```

C'est alors le pré-processeur qui génère le code (les appels aux macros) qui permettra de vérifier les assertions — ici, la pré-condition et la post-condition.

1.3 Quand et comment utiliser les assertions

Les sections précédentes ont présenté brièvement tant les principes généraux de la conception et programmation par contrat, avec utilisation d'assertions évaluées dynamiquement, que la façon dont les assertions peuvent être exprimées en C à l'aide de la macro `assert`. Dans la présente section, nous allons maintenant aborder la *pragmatique* de l'utilisation des assertions, c'est-à-dire, les règles générales d'une bonne utilisation de telles assertions — en d'autres mots, nous allons identifier dans quelles situations les assertions devraient être utilisées.

1.3.1 Assertions pour spécifier des pré-conditions

Dans son approche par contrat, Meyer souligne que la responsabilité de traiter une contrainte spéciale sur un argument peut se faire de deux façons :

1. Approche exigeante (*demanding*) : La responsabilité est assignée au client, auquel cas la condition apparaît de façon explicite dans la pré-condition du service.

[1] La présentation qui est faite ici se réfère au langage C. Toutefois, elle pourrait tout aussi bien s'appliquer au langage C++, où la macro `assert` est aussi disponible. Notons aussi que des pré-processeurs, dans le style de `iContract` pour Java [Ens01], sont aussi disponibles pour C/C++ : voir plus bas, à la fin de la section.

[2] Rappelons que langage C standard ne possède pas de type `bool`, et que ce rôle est joué par le type `int`, où `0` dénote la valeur fausse, et où une valeur non nulle dénote la valeur vraie.

[3] Plus précisément, le traducteur de Maguire.

[4] Par exemple, en appelant le compilateur avec la commande "`gcc -DNDEBUG ...`".

2. Approche tolérante : La responsabilité est assignée au fournisseur, ce qui fait que la condition n'apparaît pas dans la pré-condition, mais que le service contiendra du code qui testera la condition et effectuera alors le traitement approprié (avec un `if`, possiblement ensuite en tentant de corriger le problème, sinon en signalant explicitement une exception).

Laquelle des deux approches est préférable dépend du problème, du contexte. Dans de nombreux cas, l'approche tolérante n'est pas pertinente dans la mesure où un aucun traitement approprié ne pourra être effectué (par exemple, obtenir l'élément au sommet d'une pile vide), si ce n'est devoir signaler une exception... ce qui ne fait que reporter le problème au niveau supérieur.

De façon générale, Meyer suggère que l'approche tolérante est préférable lorsque le service traite avec des modules qui gèrent des données provenant *de l'extérieur du système* (par exemple, interfaces personne-machine, senseurs), alors que l'approche exigeante est préférable lorsque que les clients du service sont d'autres modules ou composants logiciels.

Mais que signifie exactement l'approche exigeante? Elle implique simplement que les pré-conditions doivent être strictes, donc *fortes*. En d'autres mots, on s'assure, à l'aide d'assertions appropriées, que les arguments qui seront traités par le service seront bien valides et que tout comportement indéterminé du code qui pourrait survenir à cause d'arguments invalides est rendu impossible.

```
void *memcpy( void *dest, void *src, size_t size )
{
  char *destB = (char *) dest;
  char *srcB  = (char *) src;

  /* Pointeur nuls: impossible a traiter. */
  assert( destB != NULL && srcB != NULL );
  /* Chevauchement des blocs: utiliser une autre procedure. */
  assert( destB >= srcB+size || srcB >= destB+size );

  /* On effectue la copie. */
  while(size-- > 0)
    *destB++ = *srcB++;

  return(dest);
}
```

Extrait de code 3: Une routine `memcpy` avec des pré-conditions exigeantes (adaptée de [Mag95])

Un exemple, présenté dans l'extrait de code 3 et adapté de [Mag95], illustre ces principes de base. La routine `memcpy` provient de la librairie standard du langage C. Sa description, obtenue à l'aide de "`man memcpy`" sur un système Linux, est la suivante : "La fonction `memcpy()` copie `n` octets depuis la zone mémoire `src` vers la zone mémoire `dest`. Les deux zones ne doivent pas se chevaucher. Si c'est le cas, utilisez plutôt `memmove(3)`."

Une première condition, indiquée explicitement dans la description de la fonction produite par `man`, est donc que les deux zones mémoire ne doivent pas se chevaucher. Une autre condition, implicite celle-ci, est évidemment que les pointeurs reçus ne doivent pas être nuls. Le rôle des assertions dans une approche de programmation avec contrat et assertions est de rendre *explicite* les conditions requises pour le bon fonctionnement d'une routine. C'est exactement ce rôle que jouent les deux assertions ajoutées dans le code de `memcpy` :

- `assert( destB != NULL && srcB != NULL );`
- `assert( destB >= srcB+size || srcB >= destB+size );`

Dans certains cas, il peut aussi être intéressant d'ajouter une assertion juste avant un appel d'une routine, donc dans le code du client (dans l'appelant), pour expliciter le fait que la pré-condition requise pour l'appel de la routine est bien satisfaite. Ceci peut être particulièrement utile lorsqu'il n'est pas clair, en regardant le code client entourant l'appel, que la pré-condition soit satisfaite.

1.3.2 Assertions pour spécifier des post-conditions

Dans de nombreux cas, il peut être utile de vérifier explicitement les post-conditions d'une routine, pour s'assurer que le code produit le bon résultat. Par exemple, supposons qu'on ait développé une routine qui réalise une opération de tri. Si l'algorithme utilisé est très complexe, il peut alors être crucial de s'assurer que le programme résultant, lorsqu'on l'exécute, fonctionne correctement. Ceci peut se faire, par exemple, en ajoutant des assertions appropriées comme suit — on suppose que le résultat est dans le tableau `A_trie` et que ce tableau compte `NB_ELEMS` éléments :

```
for ( int i = 0; i < NB_ELEMS-1; i++ ) {
  assert( A_trie[i] <= A_trie[i+1] );
}
```

Notons que le fait de désactiver l'évaluation dynamique des assertions aura pour effet de produire une boucle vide. Or, la plupart des compilateurs modernes sont suffisamment sophistiqués pour reconnaître une telle boucle et, dans ce cas, ne générer aucun code.

1.3.3 Assertions pour rendre explicites des hypothèses ou suppositions implicites

```
void proc( ... )
{

  #ifdef LONG_AVEC_4_OCTETS
    /* Code special et optimise pour des long a 4 octets. */
    assert( sizeof(long) == 4 );
    ...

  #else
    /* Code normal. *.
    ...
  #endif
}
```

Extrait de code 4: Assertions pour rendre explicites les hypothèses clés sur lesquelles repose le bon fonctionnement d'une routine (adaptée de [Mag95])

Lorsque le bon fonctionnement du code repose sur certaines hypothèses ou suppositions importantes, des assertions devraient toujours être utilisées pour rendre *explicite* ces hypothèses. Par exemple, supposons une routine `proc` qui a été optimisée de façon à ce qu'elle fonctionne plus efficacement sur des machines où les variables de type `long` occupent quatre (4) octets. Supposons de plus que cette routine *ne fonctionnerait plus correctement* si cette supposition n'était pas valide. Il serait alors important de rendre *explicite* ces conditions. Ceci pourrait être fait tel qu'illustré dans l'extrait de code 4 : ici, on suppose qu'un symbole `LONG_AVEC_4_OCTETS` est défini dans l'environnement pour indiquer que la version optimisée de la routine `proc` doit être

sélectionnée (information fournie au compilateur[5]). Pour plus de sûreté, toutefois, dans l'esprit de ce qui vient d'être discuté, le code conçu pour ce cas spécial est précédé d'une assertion qui rend explicite les conditions appropriées :

```
assert( sizeof(long) == 4 );
```

```
if ( 'a' <= jeton[0] && jeton[0] <= 'z' ) {
  /* Traitement d'un identificateur. */
  ...
} else if ( '0' <= jeton[0] && jeton[0] <= '9' ) {
  /* Traitement d'un nombre. */
  ...
} else {
  /* Traitement d'une chaine. */
  assert( jeton[0] == '"' );
  ...
}
```

Extrait de code 5: Assertion pour rendre explicite la condition d'un `else` complexe

Un autre exemple qui illustre une situation où il est intéressant et utile de rendre explicite les conditions d'exécution est celui d'une branche `else` d'une instruction `if` complexe (respectivement la branche `default` d'une instruction `switch`). La condition associée à un `else` (resp. au `default`) devrait être la négation de toutes les conditions (respectivement de tous les cas) qui précèdent. Une assertion peut donc être utilisée pour indiquer explicitement cette condition et vérifier à l'exécution que cette condition est bien satisfaite. Un exemple, adapté de [Ros95], est présenté dans l'extrait de code 5, où l'instruction conditionnelle est supposée traiter trois (3) cas possibles : un identificateur, un nombre, ou une chaîne (devant nécessairement débuter par le caractère `'"'`). Ainsi, si jamais un quatrième type de cas devait être ajouté, on saurait immédiatement que cette partie du code doit être modifiée en conséquence, puisque l'assertion ne serait pas valide.

1.3.4 Assertions pour tester des conditions supposément impossibles, mais pas des erreurs normales

Lorsque le programmeur sait qu'une condition ne devrait pas survenir, il est important qu'il rende cette condition explicite, de la même façon qu'il doit le faire pour les suppositions implicites.

```
switch( tag ) {
  case CAS1: ...; break;
  case CAS2: ...; break;
  ...
  default:
     assert( FALSE && "Branche default non implantee dans un switch" );
}
```

Extrait de code 6: Assertion dans une branche `default` d'un `switch`

Deux exemples qui illustrent une telle situation sont les suivants :

[5] Rappelons que la directive `#ifdef` est une directive traitée par le pré-processeur standard C, donc évaluée et traitée *à la compilation* : si le symbole indiqué après le `ifdef` est défini, le code qui suit jusqu'au `#else` est inclus dans le code traité par le compilateur. Par contre, si le symbole n'est pas défini, alors c'est le code qui apparaît entre le `#else` et le `#endif` qui sera inclus.

1. Le code d'une routine `proc` n'a pas encore été développé mais il devrait l'être ultérieurement. Une assertion peut alors être utilisée pour signaler cette information :

   ```
   void proc( ... )
   {
     assert( 0 && "Code pour proc pas encore mis en oeuvre" );
   }
   ```

   Cet exemple illustre qu'il est possible, dans l'expression utilisée comme argument à un `assert` en C, de fournir une chaîne qui indique de façon plus détaillée la raison ayant conduit à ce que l'évaluation de l'assertion avorte l'exécution du programme. Rappelons qu'en C, comme on l'a expliqué à la section 1.2, l'expression utilisée dans un `assert` doit être un simple booléen, et que cette expression est affichée (sur `stderr`) si l'évaluation de l'expression conduit à avorter le programme. Or, une valeur (constante) chaîne dénote toujours un pointeur non nul, donc une valeur vraie. De plus, peu importe la valeur `x`, on a toujours que `x ET VRAI = x`. Ceci signifie donc que l'utilisation de l'expression "`&& "Code pour ..."`" comme argument à `assert` ne change pas la valeur de l'expression utilisée dans le `assert` — de toute façon, dans ce cas-ci, l'expression est nécessairement fausse (`0 && x == 0`).

2. Un certain nombre de `case` dans un `switch` sont indiqués et tous ces `case` sont *supposés* représenter tous les cas possibles. Dans ce cas, la branche `default` du `switch` devrait assurer qu'un cas qui n'est pas supposé survenir ne survienne pas, tel que cela est illustré dans l'extrait de code 6.

```
char *strdup(char *str)
{
  char *new;

  /* Assertion utilisee correctement. */
  assert( str != NULL );

  new = (char *) malloc(strlen(str)+1);
  /* Mauvaise utilisation d'une assertion: il faut tester new et, si
     necessaire, signaler une exception ou erreur. */
  assert( new != NULL );   /* !!! Mauvaise utilisation !!! */

  strcpy(new, str);
  return(new);
}
```

Extrait de code 7: Une routine `strdup` avec une mauvaise utilisation d'assertions (adaptée de [Mag95])

Une assertion ne devrait être utilisée que pour tester une condition illégitime, supposément impossible et ne pouvant être traitée, donc qui ne devrait pas survenir en cas de fonctionnement correct du programme. Par contre, une assertion ne devrait pas être utilisée pour tester une condition, même d'erreur, qui pourrait survenir et qui devrait être traitée par le programme. L'exemple de l'extrait de code 7 illustre ce principe :

- La première assertion est utilisée correctement : l'assertion représente une pré-condition et, si `str` est `NULL`, alors c'est que le client n'a pas fait correctement son travail.

- La deuxième assertion, quant à elle, ne représente pas une bonne utilisation : la valeur retournée par `malloc` (dont la tâche est d'allouer un bloc d'espace mémoire) devrait plutôt être testée et, si invalide, une erreur devrait être signalée à l'appelant.

Un autre exemple du même style de mauvaise utilisation d'assertions est le fragment de code suivant :

```
printf( "Entrez 'O' ou 'N': " );
ch = getchar();
assert( (ch == 'O') || (ch == 'N') );  /* !!! Mauvaise utilisation !!! */
```

Ici, il est tout à fait possible, même *prévisible*, que le caractère lu ne soit pas un des deux caractères attendus. Il faudrait donc plutôt tester le caractère reçu et, ensuite, prendre une mesure appropriée pour traiter cette valeur invalide.

Finalement, une assertion ne devrait jamais avoir pour effet de rendre impossible un traitement d'erreur approprié, comme dans l'exemple suivant :

```
assert( solde < 0.0 );    /* !!! Mauvaise utilisation !!! */
if ( solde < 0.0 )  {
   /* Traitement d'erreur si le solde est negatif. */
   ...
}
```

1.3.5 Assertions avec des effets de bord : à éviter!

Un effet de bord survient lorsque l'exécution d'une expression modifie son environnement, par exemple, modifie le contenu d'une variable, avance le curseur de lecture ou d'écriture dans un fichier, alloue de l'espace mémoire, etc.

Une expression utilisée dans une assertion *ne devrait jamais avoir d'effet de bord*. Par exemple, le fragment de code suivant illustre une *très mauvaise* utilisation d'une assertion :

```
int i = 0;
while ( i < NB_ELEMS-1 ) {
  assert( (x = A[i++]) != NULL );   /* !!! Mauvaise utilisation !!! */
  ... traiter x ...
}
```

La raison pour laquelle une assertion ne devrait jamais contenir d'expressions avec effets de bord est très simple : si on désactive l'évaluation dynamique des assertions — c'est-à-dire en recompilant le programme avec une définition pour le symbole `NDEBUG` — alors le comportement du programme sans assertion *ne sera plus le même* que celui avec assertions, ce qui rendra le programme très difficile à comprendre et déboguer. Hunt et Thomas [HT00] appellent cette situation, où le débogage d'un programme modifie le comportement du programme, un *Heisenbug*.[6]

2 Techniques pour gérer les "erreurs" (les événements exceptionnels!)

Comme nous l'avons vu à la section précédente, les assertions, en plus d'être utilisées pour spécifier les contrats, peuvent être utilisées pour traiter des situations ou des erreurs qui, en principe, ne devraient jamais survenir — y compris les violations des contrats. Par contre, des techniques différentes peuvent et doivent être utilisées pour traiter des erreurs pour lesquelles on sait qu'elles pourraient survenir.

[6] Ce nom est inspiré du principe de Heisenberg, en physique, où l'observation d'un phénomène peut influer sur le comportement de ce phénomène.

Soulignons toutefois que, dans le contexte présent, le terme erreur n'est peut-être pas tout à fait approprié, puisqu'on pense souvent à erreur en termes d'une erreur dans le programme, dans le code. Ainsi, selon le *Free On-line Dictionary of Computing*[7], on doit distinguer entre *error* (erreur), *fault* (défectuosité) et *failure* (défaillance) :

- *Error : A mental mistake made by a programmer that may result in a program* fault.
- *Fault : A manifestation of an* error *in software. A fault, if encountered, may cause a* failure.
- *Failure : The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered.*

Une distinction similaire est faite dans le chapitre *Software Testing* du *Guide to the Software Engineering Body of Knowledge* :

> *It is essential to clearly distinguish between the* cause *of a malfunction, for which the term* fault *or* defect *[is] used, and an undesired effect observed in the system's delivered service, which [is] called a* failure. *Testing can reveal failures, but it is the faults that can and must be removed.*
>
> *[AMBD04, Chapitre "*Software Testing*", p. 5–2]*

Dans le cas présent, lorsqu'on parle d'erreurs prévisibles mais peu fréquentes, on parle donc d'événements indésirables (*undesired event* [Lam88]) mais prévisibles donc dont on sait qu'ils peuvent survenir, en d'autres mots, on parle d'*exceptions* — même si on continuera d'utiliser parfois plus simplement le terme "erreur".

Par exemple, supposons que l'on désire ajouter un élément dans une pile. Le fait que la pile soit pleine n'est pas nécessairement une défectuosité dans le programme — une défectuosité (*defect*, *fault*) est le résultat, la conséquence d'une erreur dans le code — mais bien un événement prévisible mais *indésirable* dans le cas où la pile est représentée de façon statique (espace alloué au début de l'exécution) ou même dans le cas où la pile est représentée de façon dynamique (manque d'espace mémoire). Dans une telle situation, il faut donc pouvoir transmettre au client du module l'information sur le fait qu'un tel événement indésirable est survenu.

McConnell [McC04, pp. 194–196] et Lamb [Lam88, p. 95] présentent diverses façons de signaler et traiter de tels événements exceptionnels, que nous présentons dans les sections qui suivent.

2.1 Signaler l'erreur en retournant une exception ou un code d'erreur approprié

Dans cette approche, on veut être certain que l'erreur sera traitée, préférablement à un niveau supérieur (par l'appelant), et non de façon locale (par l'appelé). Dans ce cas, il faut pouvoir signaler à l'appelant qu'une erreur est survenue. Il existe différentes façons possibles de signaler un tel événement exceptionnel :

Signaler une exception

Lorsque le langage fournit un mécanisme de traitement des exceptions, comme c'est le cas par exemple en Ada ou Java, on peut alors simplement signaler une exception. Toutefois, comme aucun mécanisme d'exceptions n'est disponible en C, on doit alors utiliser une autre approche.[8]

[7] `http://wombat.doc.ic.ac.uk/foldoc`

[8] En fait, il existe un mécanisme en C qui permet une forme *limitée* de traitement des exceptions : les fonctions `setjmp()` et `longjmp()`. Voici ce qu'indique la documentation Linux à ce sujet (`man setjmp`) :

> Les fonctions `setjmp()` et `longjmp()` sont utiles pour gérer les erreurs et les interruptions rencontrées

Assigner une valeur spéciale à une variable de statut

Dans cette approche, il s'agit simplement pour la routine (l'appelée) de retourner à la routine appelante (le client) une indication qu'un problème est survenu. Ceci peut se faire de différentes façons :

1. En affectant la valeur de statut à une variable reçue en argument (donc passée par référence)
2. En affectant une valeur de statut à une variable globale.
3. Dans le cas d'une fonction (donc qui retourne un résultat), en retournant la valeur de statut comme résultat de la fonction.

```
/* Type pour indiquer le statut de l'opération sur une pile. */
typedef enum { PILE_PLEINE, PILE_VIDE, PILE_OK } StatutPile;


/* Procédure pour empiler un élément sur une pile. */
void empiler( Pile p, int e, StatutPile *s )
{
  if ( !estPleine(p) ) {
    ... Traitement du cas normal ...
    *s = PILE_OK;
  } else {
    *s = PILE_PLEINE;
  }
}

...

/* Appel de la procédure et vérification du statut. */
StatutPile res;
empiler( p1, v, &res );
switch( res ) {
  case PILE_OK:
    break;
  case PILE_PLEINE:
    ... Traitement de l'erreur ...
    break;
}
```

Extrait de code 8: Exemple de retour d'un résultat qui indique le statut de l'opération par l'intermédiaire d'un paramètre passé par référence

L'extrait de code 8 illustre la première stratégie. Quant à l'extrait de code 9, il illustre la deuxième stratégie.

Finalement, mentionnons qu'il est parfois possible de combiner certaines de ces stratégies. Par exemple, sous Linux, la deuxième et la troisième stratégies sont combinées, tel que décrit à la Figure 2 (obtenu par "`man errno`"), pour traiter certains appels systèmes ou certains appels de bibliothèque.

dans des routines bas-niveau. `setjmp()` sauvegarde le contexte de pile et d'environnement dans `env` afin de l'utiliser ultérieurement avec `longjmp()`. Le contexte de pile sera invalide si la fonction qui appelle `setjmp()` se termine.

```
/* Type pour indiquer le statut de l'opération sur une pile. */
typedef enum { PILE_PLEINE, PILE_VIDE, PILE_OK } StatutPile;

/* Variable globale pour le statut et opérations associées. */
static StatutPile statutPourPile;

void       definirStatutPile( StatutPile s ) { statutPourPile = s;       }
StatutPile obtenirStatutPile()               { return( statutPourPile ); }


/* Procédure pour empiler un élément sur une pile. */
void empiler( Pile p, int e )
{
  if ( !estPleine(p) ) {
    ... Traitement du cas normal ...
    definirStatutPile( PILE_OK );
  } else {
    definirStatutPile( PILE_PLEINE );
  }
}



...

/* Appel de la procédure et vérification du statut. */
empiler( p1, v );
switch( obtenirStatutPile() ) {
  case PILE_OK:
    break;
  case PILE_PLEINE:
    ... Traitement de l'erreur ...
    break;
}
```

Extrait de code 9: Exemple d'une valeur de statut retournée par l'intermédiaire d'une variable globale et d'une opération pour lire le statut

NOM

`errno` - Code de la dernière erreur.

SYNOPSIS

```
#include <errno.h>
extern int errno;
```

DESCRIPTION

La variable entière `errno` est renseignée par les appels systèmes (et quelques fonctions de bibliothèque) pour expliquer les conditions d'erreurs. Sa valeur n'est significative que lorsque l'appel système a échoué (généralement en renvoyant -1), car même en cas de réussite une fonction de bibliothèque peut modifier `errno`.

Figure 2: Description de la variable `errno` selon "`man errno`"

2.2 Appeler une routine de traitement d'erreurs

Lorsqu'on désire effectuer de façon centrale le traitement des erreurs, par exemple pour faciliter le débogage ou s'assurer de préserver une trace (ou journal) global des événements exceptionnels, il peut être intéressant d'appeler une routine spéciale de traitement d'erreurs. Cette routine peut soit avoir été transmise par l'appelant — auquel cas on a une forme de *call-back* — soit être une routine définie globalement.

```
/* Procédure pour empiler un élément sur une pile. */
void empiler( Pile p, int e, void (*siPleine)() )
{
  if ( !estPleine(p) ) {
    ... Traitement du cas normal ...
  } else {
    (*siPleine)();
  }
}

...

/* Définition du call-back à utiliser. */
void traiterErreurPilePleine()
{ ... Traitement de l'erreur pile pleine. ... }

/* Appel de la procédure (avec le call-back). */
empiler( p1, v, traiterErreurPilePleine );
```

Extrait de code 10: Exemple d'une routine de traitement d'erreurs transmise à la procédure (*call-back*)

L'extrait de code 10 illustre l'utilisation d'une routine de traitement d'erreur reçue en argument, ce qu'on appelle un *call-back*. Voici la définition que donne de ce terme le *Free On-line Dictionary of Computing*[9] :

> *callback*
>
> *A scheme used in event-driven programs where the program registers a subroutine (a "callback handler") to handle a certain event. The program does not call the handler directly but when the event occurs, the run-time system calls the handler, usually passing it arguments to describe the event.*

On peut aussi appeler une routine de traitement des erreurs définie de façon globale. Dans ce cas, le code sera semblable à celui de l'extrait de code 10, à la différence que la procédure `traiterErreurPilePleine` n'aurait pas besoin d'être passées en argument.

2.3 Afficher un message d'erreur

Une telle approche, bien que simple et directe, peut être dangereuse, entre autres parce que l'interface personne–machine peut alors devenir saturée de messages d'erreurs si plusieurs parties du système utilisent cette approche.

De plus, il arrive souvent que lorsqu'une situation exceptionnelle survienne, bien que la routine en cours d'exécution ne puisse pas traiter cette situation ou régler le problème, la routine appelante (ou d'autres de niveau supérieur) pourrait le faire. Il est donc préférable, si une situation exceptionnelle ne peut pas être traitée localement, de propager l'information à la routine appelante.

[9] `http://wombat.doc.ic.ac.uk/foldoc`

2.4 Terminer l'exécution du programme (*shut down*)

Dans certains cas (par exemples, systèmes critiques) ou face à certains types d'erreurs, la seule chose à faire peut être de devoir avorter l'exécution du programme. Notons que c'est ce que permettent de faire, entre autres, les assertions.

Comme l'indique Hunt et Thomas dans la section *Dead Programs Tell No Lies* de leur livre *The Pragmatic Programmer* :

> *Pragmatic Programmers tell themselves that if there is an error, something very, very bad has happened. [...] One of the benefits of detecting problems as soon as you can is that you can crash earlier. And many times, crashing your program is the best thing you can do. The alternative may be to continue, writing corrupted data to some vital database or commanding the washing machine into its twentieth consecutive spin cycle. [...] A dead program normally does a lot less damage than a crippled one.*
>
> *A. Hunt et D. Thomas [HT00, pp. 120–121]*

2.5 Écrire un message d'erreur dans un journal

Lorsqu'une situation exceptionnelle survient, il peut être important d'en garder une trace, ce qu'on peut faire en écrivant les informations appropriées décrivant l'événement dans un journal (*log file*). Une telle écriture peut se faire en conjonction avec l'une ou l'autre des méthodes décrites précédemment.

Références

[AMBD04] A. Abran, J.W. Moore, P. Bourque, and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge (2004 Version)*. IEEE Computer Society Press, 2004.

[Ens01] O. Enseling. iContract: Design by contract in Java. *Java World*, February 2001. http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html.

[HT00] A. Hunt and D. Thomas. *The Pragmatic Programmer—From journeyman to master.* Addison-Wesley, 2000. [QA76.6H858].

[KP01] B.W. Kernighan and R. Pike. *La programmation — En pratique.* Vuibert, Paris, 2001. [QA76.6K48814].

[Lam88] D.A. Lamb. *Software Engineering: Planning for Change.* Prentice-Hall, 1988.

[LG01] B. Liskov and J. Guttag. *Program Development in Java.* Addison-Wesley, 2001.

[Mag95] S. Maguire. *L'art du code.* Microsoft Press, 1995.

[McC04] S. McConnell. *Code Complete — A Practical Handbook of Software Construction (Second Edition).* Microsoft Press, Redmond, WA, 2004.

[Mey92] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction (Second edition).* Prentice-Hall, 1997.

[Ros95] D.S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. on Soft. Eng.*, 21(1):19–31, 1995.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language—Precise Modeling with UML.* Addison-Wesley, 1999.