

Prof. Burkhard Wolff
wolff@lri.fr

Test fonctionnel

Date : 7 février 2023

Exercice 1 (Analyse partitionnelle et test aux limites)

Une société vend deux produits A et B au prix unitaire de 5 € pour A et de 10 € pour B. Une commande comprend une certaine quantité du produit A et une certaine quantité du produit B. Le coût d'une commande est la somme totale des prix unitaires des produits commandés, à laquelle on applique une réduction selon les règles suivantes :

- Si la somme totale est supérieure ou égale à 200 €, on applique une réduction de 5%, si elle est supérieure ou égale à 1000 €, la réduction est de 20%. Ces deux réductions ne sont pas cumulables et portent sur la somme totale.
- La société souhaitant encourager la vente de A, on applique, sur le prix obtenu grâce à la règle précédente, une réduction supplémentaire de 10% si la commande comprend au moins 45 produits A.

1. Donnez un ensemble de tests pour le calcul du coût total d'une commande. Pour chaque test :
 - expliquez le cas particulier visé par le test ;
 - donnez la formule du résultat attendu ;
 - donnez un exemple de valeurs concrètes et le résultat correspondant attendu.
2. Complétez votre jeu de tests par une analyse aux limites.

Solution :

Objectif de test	Formule du résultat attendu	Données d'entrée		Résultat attendu
		nb A	nb B	
Prix sans réduction : $5A + 10B < 200$ et $A < 45$	$5A + 10B$	3	5	65
Réduction de 5% : $200 \leq 5A + 10B < 1000$ et $A < 45$	$(5A + 10B) \times 0,95$	20	20	285
Réduction de 20% : $5A + 10B \geq 1000$ et $A < 45$	$(5A + 10B) \times 0,8$	20	100	880
Réduction de 5% puis 10% : $200 \leq 5A + 10B < 1000$ et $A \geq 45$	$(5A + 10B) \times 0,95 \times 0,9$	60	10	342
Réduction de 20% puis 10% : $5A + 10B \geq 1000$ et $A \geq 45$	$(5A + 10B) \times 0,8 \times 0,9$	60	100	936

On peut compléter ce jeu de tests par des tests aux limites : $A = 0$, $B = 0$, $5A + 10B = 199$, $5A + 10B = 200$, $5A + 10B = 999$, $5A + 10B = 1000$, $A = 44$, $A = 45$, chacune de ces conditions testées indépendamment des autres.

Exercice 2 (Test fonctionnel formel d'une fonction)

L'opération `middle` prend en entrée trois entiers différents deux à deux et renvoie l'entier parmi les trois qui n'est ni le plus grand ni le plus petit.

1. Donnez une spécification formelle de cette opération.
2. Construisez la forme normale disjonctive de cette spécification et déduisez-en un ensemble de cas de test pour l'opération `middle`.
3. Donnez des tests concrets pour chacun des cas trouvés à la question précédente.

Solution :

Question 1

$middle(int\ x, int\ y, int\ z) : int$

pre : $x \neq y \wedge y \neq z \wedge x \neq z$

post : $(result = x \vee result = y \vee result = z)$

$\wedge (result > x \vee result > y \vee result > z)$

$\wedge (result < x \vee result < y \vee result < z)$

Question 2 On ignore les états σ_{pre} et σ qui ne sont pas référencés dans ce problème. On note $r = result$.

$$\begin{aligned}
 pre_{middle}(x, y, z) \wedge post_{middle}(x, y, z, r) &= x \neq y \wedge y \neq z \wedge x \neq z \wedge (r = x \vee r = y \vee r = z) \\
 &\quad \wedge (r > x \vee r > y \vee r > z) \wedge (r < x \vee r < y \vee r < z) \\
 &= x \neq y \wedge y \neq z \wedge x \neq z \wedge \\
 &\quad ((r = x \wedge r > y \wedge r < z) \vee \\
 &\quad (r = x \wedge r > z \wedge r < y) \vee \\
 &\quad (r = y \wedge r > x \wedge r < z) \vee \\
 &\quad (r = y \wedge r > z \wedge r < x) \vee \\
 &\quad (r = z \wedge r > x \wedge r < z) \vee \\
 &\quad (r = z \wedge r > y \wedge r < x))
 \end{aligned}$$

On obtient 6 cas de test.

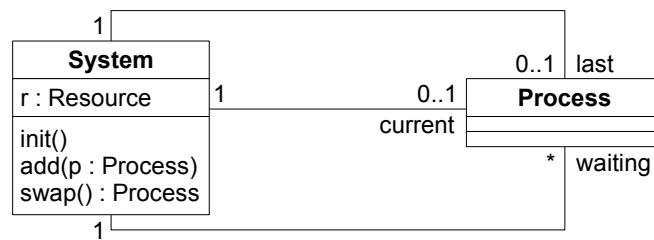
Question 3

Cas de test	Données d'entrée			Résultat attendu
	x	y	z	
$C1$	12	5	20	12
$C2$	7	9	2	7
$C3$	-2	0	2	0
$C4$	61	-1	-10	-1
$C5$	5	8	6	6
$C6$	95	-39	-4	-4

Exercice 3 (Test fonctionnel formel d'une méthode)

On considère la spécification suivante.

L'instance de **System** gère une (unique) ressource, par exemple un processeur, qu'il doit partager entre des processus. Un processus ne termine jamais et ne libère jamais spontanément la ressource mais uniquement sur la requête du système via l'opération **swap**, qui permet d'attribuer la ressource à l'un des processus en attente. Le système n'est pas forcément équitable et se contente simplement d'essayer de ne pas redonner la ressource au processus qui le possédait déjà lors de l'échange précédent. Il dispose pour cela de **current**, l'éventuel processus qui possède actuellement la ressource, de **last**, l'éventuel processus qui possédait la ressource avant **current**, et de **waiting**, l'ensemble des processus demandeurs (qui inclut **last**).



L'opération **init()** met le système dans un état initial où il n'existe aucun processus. L'opération **add(p :Process)** ajoute un processus au système. S'il n'existe aucun processus, **p** devient le processus courant qui détient la ressource, sinon il est ajouté à **waiting**. L'opération **swap()** change le processus actif qui détient la ressource.

$$\begin{aligned}
inv_{\text{System}}(s) &\equiv s.\text{waiting} \neq \emptyset \longrightarrow s.\text{current} \neq \text{NULL} \\
&\wedge s.\text{last} \neq \text{NULL} \longrightarrow (s.\text{last} \neq s.\text{current} \wedge s.\text{last} \in s.\text{waiting}) \\
&\wedge s.\text{current} \neq \text{NULL} \longrightarrow s.\text{current} \notin s.\text{waiting}
\end{aligned}$$

$$pre_{\text{swap}}(s) \equiv s.\text{waiting} \neq \emptyset$$

$$\begin{aligned}
post_{\text{swap}}(result)(s) &\equiv result \in old(s.\text{waiting}) \wedge s.\text{current} = result \\
&\wedge s.\text{waiting} = old(s.\text{waiting}) \setminus \{result\} \cup old(s.\text{current}) \\
&\wedge s.\text{last} = old(s.\text{current}) \\
&\wedge (old(s.\text{last}) \neq \text{NULL} \wedge |old(s.\text{waiting})| > 1 \longrightarrow result \neq old(s.\text{last}))
\end{aligned}$$

1. Donnez informellement les différents cas de test pour l'opération **swap** en fonction des valeurs de **waiting**, **last** et **current** avant et après l'opération.
2. Calculez la DNF pour l'opération **swap** et construisez les cas de test. Commencez le calcul de la DNF avec $inv_{\text{System}}(s)(\sigma_{pre}) \wedge pre_{\text{swap}}(s)(\sigma_{pre}) \wedge post_{\text{swap}}(s)(\sigma_{pre}, \sigma) \wedge inv_{\text{System}}(s)(\sigma)$.
3. Sélectionnez des tests concrets pour chacun des cas de test déduits à la question précédente, en choisissant des valeurs pour **waiting**, **last** et **current**.
4. Pour chacun des tests obtenus à la question précédente, construisez la suite d'appel des fonctions **init**, **add** et **swap** permettant d'exécuter ce test.

Solution :

Question 1 On voit dans la spécification de l'opération **swap** que le choix du processus *result* dans $old(s.\text{waiting})$ dépend de deux conditions : si $old(s.\text{last})$ est null ou si $old(s.\text{waiting})$ ne contient qu'un seul processus, alors *result* sera n'importe quel processus dans $old(s.\text{waiting})$, sinon, on le choisit de manière à ne pas reprendre $old(s.\text{last})$. On va donc avoir quatre cas de test différents, selon qu'aucune de ces conditions n'est vraie, qu'une de ces deux conditions seulement est vraie ou que les deux conditions sont vraies. On a donc les cas de test suivants.

1. $old(s.\text{last})$ n'est pas vide et $old(s.\text{waiting})$ contient plus d'un processus. Dans ce cas, *result* est n'importe quel processus dans $old(s.\text{waiting})$ à l'exception de $old(s.\text{last})$.
2. $old(s.\text{last})$ est vide et $old(s.\text{waiting})$ contient plus d'un processus. Dans ce cas, *result* est n'importe quel processus dans $old(s.\text{waiting})$.
3. $old(s.\text{last})$ n'est pas vide et $old(s.\text{waiting})$ contient un seul processus. Dans ce cas, *result* est l'unique processus contenu dans $old(s.\text{waiting})$ (même s'il est égal à $old(s.\text{last})$).
4. $old(s.\text{last})$ est vide et $old(s.\text{waiting})$ contient un seul processus. Dans ce cas, *result* est l'unique processus contenu dans $old(s.\text{waiting})$.

On va retrouver de façon formelle ces quatre cas en calculant la DNF pour l'opération **swap**.

Question 2 On abrège $s.current$, $s.waiting$, $s.last$ et $result$ respectivement en $s.c$, $s.w$, $s.l$ et r .

On a l'expression suivante de l'invariant (où les implications $A \longrightarrow B$ ont été transformées en disjonctions $\neg A \vee B$) :

$$\begin{aligned} inv_{System}(s) &\equiv (s.w = \emptyset \vee s.c \neq NULL) \\ &\quad \wedge (s.l = NULL \vee (s.l \neq s.c \wedge s.l \in s.w)) \\ &\quad \wedge (s.c = NULL \vee s.c \notin s.w) \end{aligned}$$

Juste pour expliquer d'où viennent les *old*'s. La notation dessus est une abbreviation pour :

$$\begin{aligned} inv_{System}(s)(\sigma) &\equiv (s.w(\sigma) = \emptyset \vee s.c(\sigma) \neq NULL) \\ &\quad \wedge (s.l(\sigma) = NULL \vee (s.l(\sigma) \neq s.c(\sigma) \wedge s.l(\sigma) \in s.w(\sigma))) \\ &\quad \wedge (s.c(\sigma) = NULL \vee s.c(\sigma) \notin s.w(\sigma)) \end{aligned}$$

... ce qui explique que $nv_{System}(s)(\sigma_{pre})$ est equivalent de :

$$\begin{aligned} inv_{System}(s)(\sigma_{pre}) &= (old(s.w) = \emptyset \vee old(s.c) \neq NULL) \\ &\quad \wedge (old(s.l) = NULL \vee (old(s.l) \neq old(s.c) \wedge old(s.l) \in old(s.w))) \\ &\quad \wedge (old(s.c) = NULL \vee old(s.c) \notin old(s.w)) \end{aligned}$$

On obtient l'expression suivante pour l'opération *swap* :

$$\begin{aligned} pre_{swap}(s)(\sigma_{pre}) \wedge post_{swap}(s)(\sigma_{pre}, \sigma) &= old(s.w) \neq \emptyset \\ &\quad \wedge r \in old(s.w) \\ &\quad \wedge s.c = r \\ &\quad \wedge s.w = old(s.w) \cup \{old(s.c)\} \setminus \{r\} \\ &\quad \wedge s.l = old(s.c) \\ &\quad \wedge (old(s.l) = NULL \vee |old(s.w)| \leq 1 \vee r \neq old(s.l)) \end{aligned}$$

(1)
(2)
(3)

On n'a pas besoin de déplier plus les invariants, étant donné qu'on va à chaque fois pouvoir simplifier les disjonctions. Par exemple, dans la post-condition, on a $old(s.w) \neq \emptyset$, la première disjonction de l'invariant $old(inv_{System}(s))$ se réduit donc au second terme $old(s.c) \neq NULL$.

On prend la conjonction globale des invariants dans les deux états et des pré-post de *swap*, en simplifiant les parties des invariants qui peuvent l'être. L'invariant dans l'état s est clairement vérifié.

$$\begin{aligned} inv_{System}(s)(\sigma_{pre}) \wedge inv_{System}(s)(\sigma_{pre}) \\ \wedge pre_{swap}(s)(\sigma_{pre}) \wedge post_{swap}(s)(\sigma_{pre}, \sigma) &= old(s.c) \neq NULL \\ &\quad \wedge (old(s.l) = NULL \vee (old(s.l) \neq old(s.c) \wedge old(s.l) \in old(s.w))) \\ &\quad \wedge old(s.c) \notin old(s.w) \\ &\quad \wedge old(pre_{swap}(s)) \wedge post_{swap}(s) \end{aligned}$$

On va ensuite découper les cas selon la disjonction de la post-condition dont on a numéroté chaque terme par (1), (2) et (3). On obtient alors pour (1) :

$$\begin{aligned}
C_1 = & \quad old(s.w) \neq \emptyset \\
& \wedge r \in old(s.w) \\
& \wedge s.c = r \\
& \wedge s.w = old(s.w) \cup \{old(s.c)\} \setminus \{r\} \\
& \wedge s.l = old(s.c) \\
& \wedge old(s.l) = NULL & (1) \\
& \wedge old(s.c) \neq NULL & old(inv) \\
& \wedge old(s.c) \notin old(s.w)
\end{aligned}$$

Ce cas de test correspond donc au cas où $old(s.last)$ est null : p est alors n'importe quel processus de $old(s.waiting)$ et l'ancien processus courant devient $s.last$ et remplace $result$ dans $s.waiting$ (cas 2 dans la question 1).

Pour (2), on obtient deux cas :

$$\begin{aligned}
C_2 = & \quad old(s.w) \neq \emptyset \\
& \wedge r \in old(s.w) \\
& \wedge s.c = r \\
& \wedge s.w = old(s.w) \cup \{old(s.c)\} \setminus \{r\} \\
& \wedge s.l = old(s.c) \\
& \wedge |old(s.w)| \leq 1 & (2) \\
& \wedge old(s.c) \neq NULL & old(inv) \\
& \wedge old(s.l) = NULL \\
& \wedge old(s.c) \notin old(s.w)
\end{aligned}$$

$$\begin{aligned}
C'_2 = & \quad old(s.w) \neq \emptyset \\
& \wedge r \in old(s.w) \\
& \wedge s.c = r \\
& \wedge s.w = old(s.w) \cup \{old(s.c)\} \setminus \{r\} \\
& \wedge s.l = old(s.c) \\
& \wedge |old(s.w)| \leq 1 & (2) \\
& \wedge old(s.c) \neq NULL & old(inv) \\
& \wedge old(s.l) \neq old(s.c) \wedge old(s.l) \in old(s.w) \\
& \wedge old(s.c) \notin old(s.w)
\end{aligned}$$

Le premier cas correspond au cas où $old(s.last)$ est vide et où $old(s.waiting)$ ne contient qu'un seul processus. À ce moment-là, $result$ est forcément ce processus et le processus courant devient $s.last$ et remplace le processus présent dans $s.waiting$ (cas 4 dans la question 1).

Le deuxième cas correspond aussi au cas où $old(s.waiting)$ ne contient qu'un seul processus mais où $s.last$ n'est pas vide. Le processus *result* est alors forcément le processus contenu dans $old(s.waiting)$, qui est lui-même forcément égal à celui contenu dans $old(s.last)$. C'est le seul cas où le système a le droit de choisir le processus de $old(s.last)$ comme nouveau processus courant (cas 3 dans la question 1).

Pour (3) on obtient :

$$\begin{aligned}
C_3 = & \quad old(s.w) \neq \emptyset \\
& \wedge r \in old(s.w) \\
& \wedge s.c = r \\
& \wedge s.w = old(s.w) \cup \{old(s.c)\} \setminus \{r\} \\
& \wedge s.l = old(s.c) \\
& \wedge r \neq old(s.l) & (3) \\
& \wedge old(s.c) \neq NULL & old(inv) \\
& \wedge old(s.l) \neq old(s.c) \wedge old(s.l) \in old(s.w) \\
& \wedge old(s.c) \notin old(s.w)
\end{aligned}$$

Ce cas correspond au cas général où $old(s.waiting)$ est quelconque et où $old(s.last)$ n'est pas vide. Le processus *result* est alors choisi dans $old(s.waiting)$ de façon à ne pas être égal à $old(s.last)$ (cas 1 dans la question 2).

Question 4 On représente une instance de test par les valeurs de $s.w$, $s.l$ et $s.c$ avant et après l'opération, plus le résultat renvoyé. On prend un ensemble de processus $p_1, p_2, p_3, p_4 \dots$ tous différents.

Objectif de test	Données d'entrée			Résultat attendu			
	$old(s.w)$	$old(s.l)$	$old(s.c)$	$s.w$	$s.l$	$s.c$	<i>result</i>
C_1	$\{p_1, p_2, p_3\}$	\emptyset	$\{p_4\}$	$\{p_1, p_2, p_3, p_4\} \setminus \{p\}$ où $p \in \{p_1, p_2, p_3\}$	$\{p_4\}$	$\{p\}$	p
C_2	$\{p_2\}$	\emptyset	$\{p_1\}$	$\{p_1\}$	$\{p_1\}$	$\{p_2\}$	p_2
C'_2	$\{p_3\}$	$\{p_3\}$	$\{p_1\}$	$\{p_1\}$	$\{p_1\}$	$\{p_3\}$	p_3
C_3	$\{p_1, p_3, p_4\}$	$\{p_3\}$	$\{p_2\}$	$\{p_1, p_2, p_3, p_4\} \setminus \{p\}$ où $p \in \{p_1, p_4\}$	$\{p_2\}$	$\{p\}$	p

Pour que les cas restent disjoints, il faut bien choisir l'ensemble $s.waiting$: lorsqu'il n'a pas de contrainte, il faut qu'il contienne plusieurs processus (au moins 3 pour être suffisamment général).

On remarque qu'on a un cas particulier du cas 3 lorsque $old(s.waiting)$ ne contient que deux processus : le processus choisi est alors forcément celui qui n'est pas dans $old(s.last)$.

Question 5 On fera attention à réinitialiser l'état entre chaque test, de manière à assurer l'indépendance des tests. On a qu'un seul observateur, la fonction *swap*, qui renvoie le processus devenu courant. On n'a aucun moyen de connaître la valeur de *last* ni l'ensemble *waiting*. NB : on n'est pas obligé d'utiliser *JUnit* pour vérifier la valeur de *current* à la fin du test, on peut tout aussi bien écrire `return (c == p1)`.

Cas C_2 du tableau précédent.

```
// préambule du test : on crée l'état dans lequel appeler swap pour le cas 2
init()
add(p1)    // current = p1
add(p2)    // current = p1, waiting = {p2}

// corps du test : on appelle swap
c = swap() // current = p2, last = p1, waiting = {p1}

// identification de l'état atteint : c doit être p2. En JUnit :
assertEquals(c,p2)
```

Cas C'_2 du tableau précédent.

```
// préambule du test : on crée l'état dans lequel appeler swap pour le cas 2'
init()
add(p1)    // current = p1
add(p2)    // current = p1, waiting = {p2}
swap()     // current = p2, last = p1, waiting = {p1}

// corps du test : on appelle swap
c = swap() // current = p1, last = p2, waiting = {p2}

// identification de l'état atteint : c doit être p1. En JUnit :
assertEquals(c,p1)
```

Cas particulier du cas C_3 du tableau précédent.

```
// préambule du test : on crée l'état dans lequel appeler swap pour le cas 3
// (le cas où il y a seulement deux processus en attente)
// on ajoute deux processus, on les swap pour savoir lequel est dans last,
// puis on ajoute le 3e
init()
add(p1)    // current = p1
add(p2)    // current = p1, waiting = {p2}
swap()     // current = p2, last = p1, waiting = {p1}
add(p3)    // current = p2, last = p1, waiting = {p1,p3}

// corps du test : on appelle swap
c = swap() // current = p3, last = p2, waiting = {p1,p2}

// identification de l'état atteint : c doit être p3. En JUnit :
assertEquals(c,p3)
```

Cas général du cas C_3 du tableau précédent.


```

// préambule du test : on crée l'état dans lequel appeler swap pour le cas 3
// on ajoute deux processus, on les swap pour savoir lequel est dans last,
// puis on en ajoute deux autres
init()
add(p1)    // current = p1
add(p2)    // current = p1, waiting = {p2}
swap()     // current = p2, last = p1, waiting = {p1}
add(p3)    // current = p2, last = p1, waiting = {p1,p3}
add(p4)    // current = p2, last = p1, waiting = {p1,p3,p4}

// corps du test : on appelle swap
c = swap() // current = p3 ou p4, last = p2, waiting = {p1,p3 ou p4}

// identification de l'état atteint : on ne sait pas si c est égal à p3 ou p4
assertTrue(c == p3 or c == p4)

```

Cas C_1 du tableau précédent.

```

// préambule du test : on crée l'état dans lequel appeler swap pour le cas 1
// on ajoute tous les processus avant d'appeler swap, pour que last reste vide
init()
add(p1)    // current = p1
add(p2)    // current = p1, waiting = {p2}
add(p3)    // current = p1, waiting = {p2,p3}
add(p4)    // current = p1, waiting = {p2,p3,p4}

// corps du test : on appelle swap
c = swap() // current = p2 ou p3 ou p4, last = p1,
           // waiting = {p1,p2,p3,p4} - current

// identification de l'état atteint : on ne sait pas si c est égal à
// p2 ou p3 ou p4
assertTrue(c == p2 || c == p3 || c == p4)

```