Systèmes concurrents

Philippe Quéinnec

14 septembre 2012

Quatrième partie

Moniteur

Plan

Limites des sémaphores

- imbrication aspects de synchronisation/aspects fonctionnels
  $\rightarrow$ manque de modularité, code des activités interdépendant
- pas de contrainte sur le protocole d'utilisation des sémaphores
  $\rightarrow$ démarche de conception artisanale, à partir de schémas élémentaires (attendre/signaler un événement, contrôler l'accès à une ressource...)
- approche ($\rightarrow$ raisonnement) *opératoire*
  $\rightarrow$ vérification difficile

Définition – Hoare 1974

Définition

Un moniteur = un module exportant des procédures (et éventuellement des constantes et des types).
Contrainte d'exclusion mutuelle pour l'exécution des procédures du moniteur : au plus une procédure en cours d'exécution.
Mécanisme de synchronisation interne.

Un moniteur est passif : ce sont les activités qui invoquent ses procédures.

Variable condition

Définition

Variables de type condition définies dans le moniteur.
Opérations possibles sur une condition C :

- C.wait : bloque l'activité appelante en libérant l'accès exclusif au moniteur.
- C.signal : s'il y a des activités bloquées sur C, en réveille une ; sinon, nop.

- Pas de mémorisation des « signaux »
- Ne pas confondre condition et sémaphore

Exemple élémentaire

Travail délégué : 1 client + 1 ouvrier

Les activités

```
boucle                          ||  boucle
    ⋮                           ||      ⋮
    déposer_travail(t)          ||      x ← prendre_travail()
    ⋮                           ||      // (y ← f(x))
    r ←lire_résultat()          ||      rendre_résultat(y)
    ⋮                           ||      ⋮
finboucle                       ||  finboucle
```

Exemple élémentaire – Le moniteur

variables d'état : Greq, Grés
variables conditions : TravailDéposé, RésultatDispo

```
déposer_travail(in t)
    Greq ← t
    TravailDéposé.signal
```

```
prendre_travail(out t)
    si Greq = null alors
      TravailDéposé.wait
    finsi
    t ← Greq
    Greq ← null
```

```
lire_résultat(out r)
    si Grés = null alors
      RésultatDispo.wait
    finsi
    r ← Grés
    Grés ← null
```

```
rendre_résultat(in y)
    Grés ← y
    RésultatDispo.signal
```

Transfert du contrôle exclusif

Rappel : le code dans le moniteur doit être exécuté comme une section critique.
Lors d'un réveil par `signal`, qui obtient/garde l'accès exclusif ?

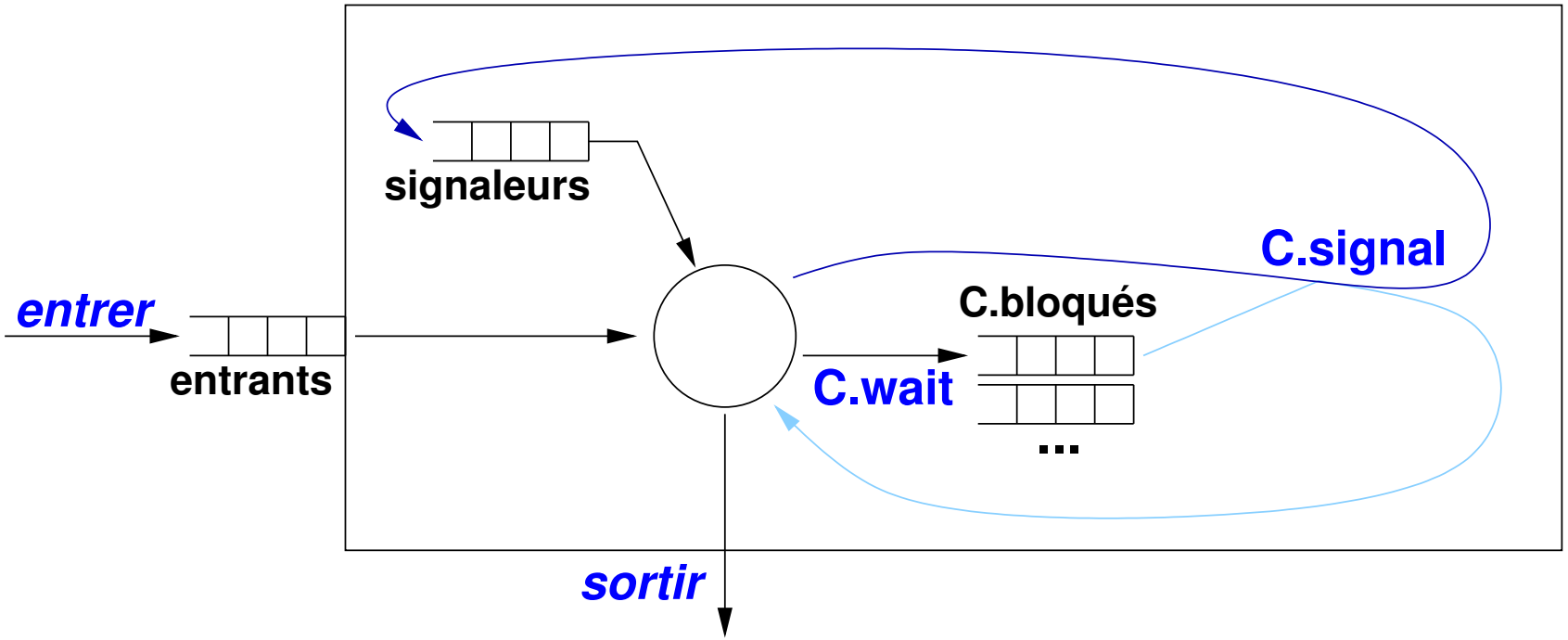
Priorité au signalé

Lors du réveil par `signal`, l'accès exclusif est transféré à l'activité réveillée (signalée) ; l'activité signaleuse est mise en attente de pouvoir ré-acquérir l'accès exclusif.

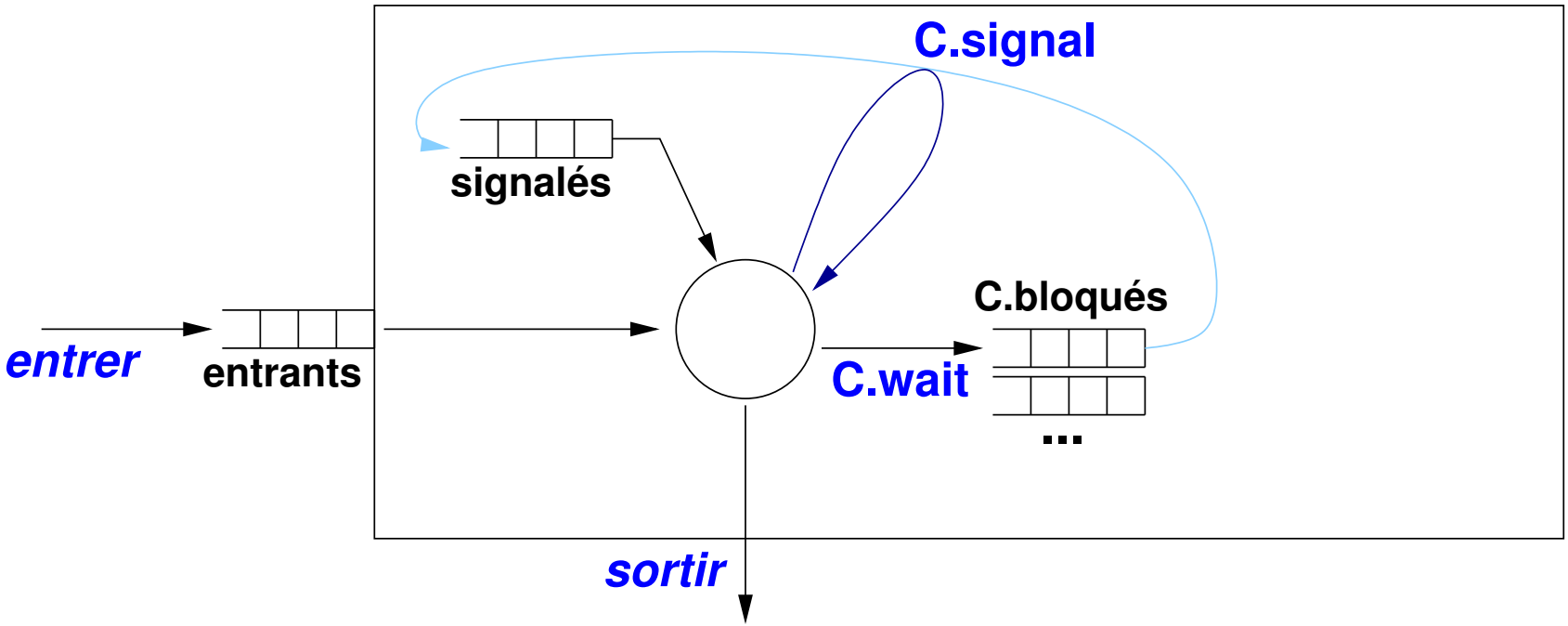
Priorité au signaleur

Lors du réveil par `signal`, l'accès exclusif est conservé par l'activité réveilleuse ; l'activité réveillée (signalée) est mise en attente de pouvoir acquérir l'accès exclusif.

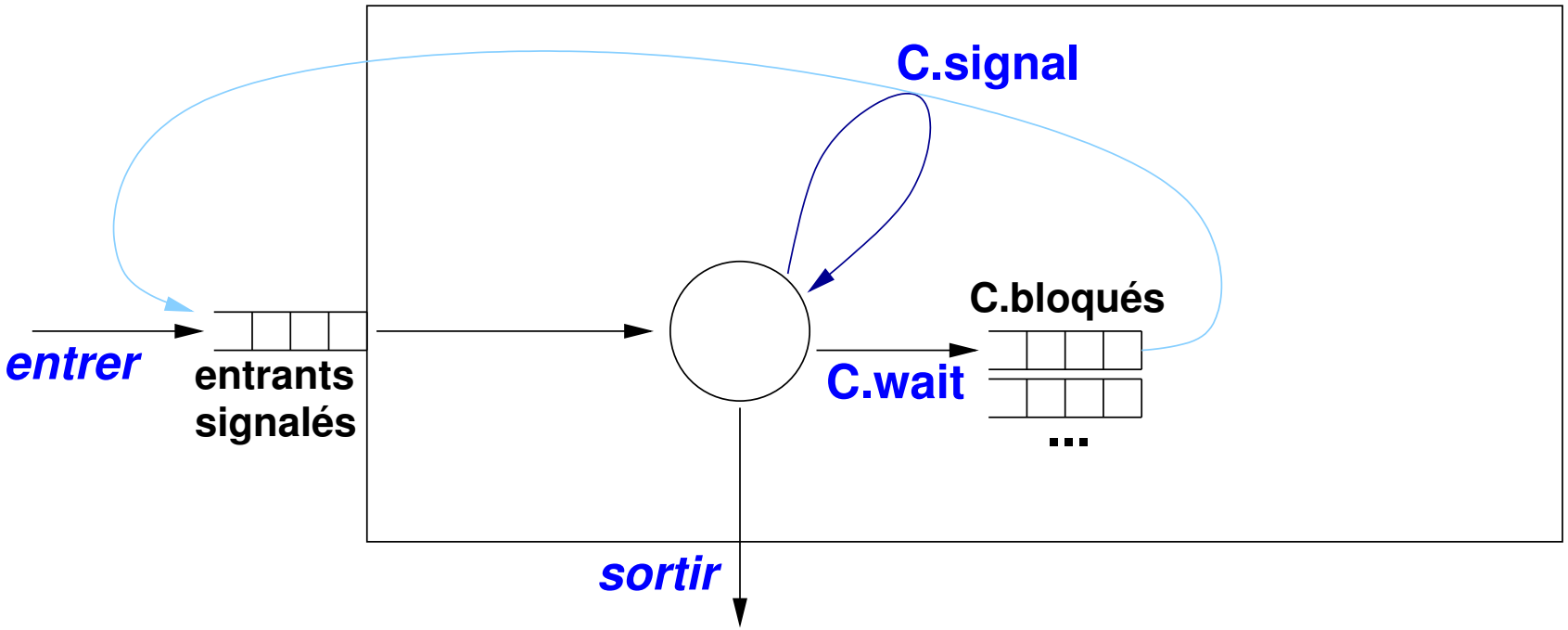
Priorité au signalé



Priorité au signaleur



Priorité au signaleur



Et donc ?

- Priorité au signalé : garantit que l'activité réveillée obtient l'accès au moniteur dans l'état où il était lors du signal.
  - Raisonnement simplifié
  - Absence de famine facile
- Priorité au signaleur : le réveillé obtient le moniteur ultérieurement, éventuellement après que d'autres activités ont eu accès au moniteur.
  - Implantation du mécanisme plus simple et plus performante
  - Nécessité de tester à nouveau la condition de déblocage au réveil du signalé
  - Absence de famine/vivacité plus difficile

Plan

Méthodologie

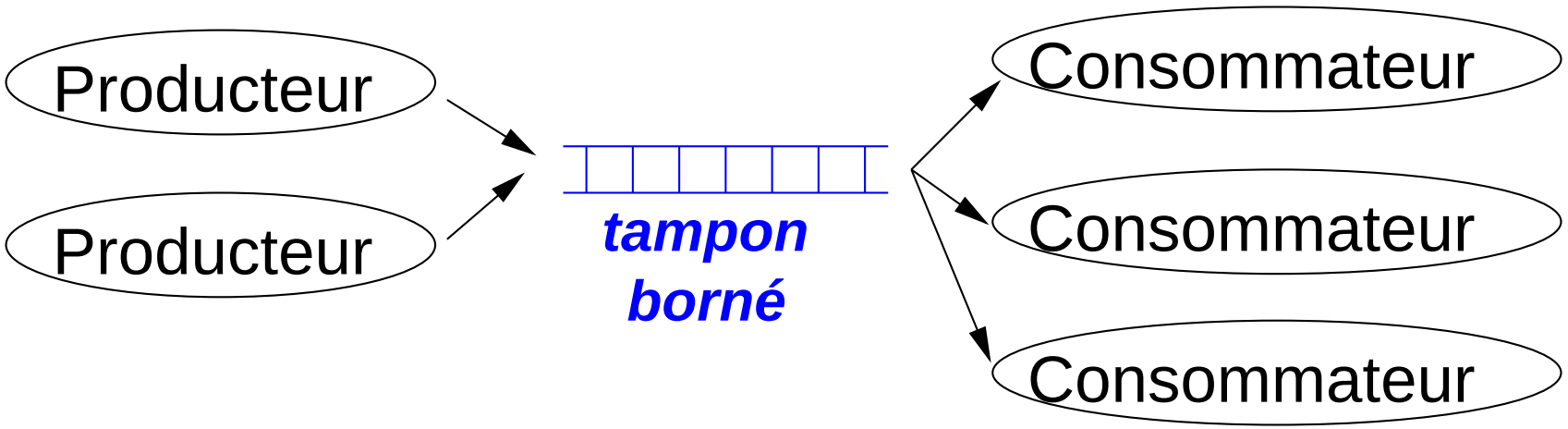
1. Déterminer l'interface du moniteur
2. Énoncer informellement les prédicats d'acceptation de chaque opérations
3. Déduire les variables d'état qui permettent d'écrire ces prédicats d'acceptation
4. Formuler l'invariant du moniteur et les prédicats d'acceptation
5. Pour chaque prédicat d'acceptation, définir une variable condition
6. Programmer les opérations
7. Vérifier l'invariant et les réveils

Schéma standard d'une opération

Si prédicat d'acceptation est faux alors
    Attendre (*wait*) sur la variable condition associée
finsi
{ (1) État nécessaire au bon déroulement }
Maj de l'état du moniteur (action)
{ (2) État garanti }
Signaler (*signal*) les variables conditions dont le prédicat associé est vrai

Vérifier, pour chaque variable condition, que chaque précondition à `signal` (2) implique chaque postcondition de `wait` (1)

Producteurs/consommateurs



- tampon de taille borné et fixé
- nombre indéterminé et dynamique de producteurs
- " " " " de consommateurs
- objectifs : ne pas écraser une case occupée, une unique lecture consommatrice par case, attendre pour déposer si plein, attendre pour retirer si vide

Méthodologie appliquée aux producteurs/consommateurs

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : $\texttt{nbOccupées} < N$
   - retirer : $\texttt{nbOccupées} > 0$
4. Invariant : $0 \leq \texttt{nbOccupées} \leq N$
5. Variables conditions : `PasPlein`, `PasVide`

Méthodologie appliquée aux producteurs/consommateurs

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : $\texttt{nbOccupées} < N$
   - retirer : $\texttt{nbOccupées} > 0$
4. Invariant : $0 \leq \texttt{nbOccupées} \leq N$
5. Variables conditions : `PasPlein`, `PasVide`

Méthodologie appliquée aux producteurs/consommateurs

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : $\texttt{nbOccupées} < N$
   - retirer : $\texttt{nbOccupées} > 0$
4. Invariant : $0 \leq \texttt{nbOccupées} \leq N$
5. Variables conditions : `PasPlein`, `PasVide`

Méthodologie appliquée aux producteurs/consommateurs

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : $\texttt{nbOccupées} < N$
   - retirer : $\texttt{nbOccupées} > 0$
4. Invariant : $0 \leq \texttt{nbOccupées} \leq N$
5. Variables conditions : `PasPlein`, `PasVide`

Méthodologie appliquée aux producteurs/consommateurs

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : `nbOccupées` $< N$
   - retirer : `nbOccupées` > 0
4. Invariant : $0 \leq$ `nbOccupées` $\leq N$
5. Variables conditions : `PasPlein`, `PasVide`

Méthodologie appliquée aux producteurs/consommateurs

1. Interface :
   - `déposer(in v)`
   - `retirer(out v)`
2. Prédicats d'acceptation :
   - déposer : il y a de la place, le tampon n'est pas plein
   - retirer : il y a quelque chose, le tampon n'est pas vide
3. Variables d'état :
   - `nbOccupées` : natural
   - déposer : $\texttt{nbOccupées} < N$
   - retirer : $\texttt{nbOccupées} > 0$
4. Invariant : $0 \leq \texttt{nbOccupées} \leq N$
5. Variables conditions : `PasPlein`, `PasVide`

déposer(in v)

```
si ¬(nbOccupées < N) alors
     PasPlein.wait
finsi
{ nbOccupées < N }
// action applicative (ranger v dans le tampon)
nbOccupées++
{ N ≥ nbOccupées > 0 }
PasVide.signal
```

retirer(out v)

```
si ¬(nbOccupées > 0) alors
     PasVide.wait
finsi
{ nbOccupées > 0 }
// action applicative (prendre v dans le tampon)
nbOccupées--
{ 0 ≤ nbOccupées < N }
PasPlein.signal
```

Plan

Régions critiques

- Éliminer les variables conditions et les appels explicites à `signal` ⇒ déblocages calculés par le système.
- Exclusion mutuelle plus « fine », en listant les variables partagées effectivement utilisées.

```
region liste des variables utilisées
when prédicat logique
do code
```

1. Attente que le prédicat logique soit vrai
2. Le code est exécuté en exclusion mutuelle vis-à-vis des autres régions ayant (au moins) une variable commune
3. À la fin du code, évaluation automatique des prédicats logiques des régions pour débloquer éventuellement.

Régions critiques

Exemple

```
tampon : shared array 0..N-1 of msg;
nbOcc : shared int := 0;
retrait, dépôt : shared int := 0, 0;

déposer(m)                          | retirer()
  region                            |   region
    nbOcc, tampon, dépôt            |     nbOcc, tampon, retrait
  when                              |   when
    nbOcc < N                       |     nbOcc > 0
  do                                |   do
    tampon[dépôt] ← m               |     Result ← tampon[retrait]
    dépôt ← dépôt + 1 % N           |     retrait ← retrait + 1 % N
    nbOcc ← nbOcc + 1               |     nbOcc ← nbOcc - 1
  end                               |   end
```

Plan

Programmation par aspects

Séparer proprement différents points de vue :

- aspect fonctionnel
- aspect synchronisation
- aspect sécurité
- aspect persistance
- aspect distribution
- aspect configuration
- ⋮

Tissage de ces aspects $\Rightarrow$ application

Aspect fonctionnel

Exemple

```
class ProdCons<T> {
    T[] tampon = new T[N];
    int retrait = 0;
    int dépôt = 0;

    void déposer (T msg) {
        tampon[dépôt] = msg;
        dépôt = (dépôt + 1) % N;
    }
     T retirer() {
        T r = tampon[retrait];
        retrait = (retrait + 1) % N;
        return r;
    }
}
```

Aspect synchronisation

Exemple

```
coordinator of ProdCons {
    int #déposer = 0;
    int #retirer = 0;
    invariant 0 ≤ #déposer - #retirer ≤ N
    selfex déposer, retirer;
    mutex { déposer, retirer };
    guard déposer :
        require #déposer - #retirer < N;
        onexit { #déposer++; }
    guard retirer :
        require #déposer - #retirer > 0;
        onexit { #retirer++; }
}
```

Objectifs/défauts

Objectifs

- séparation des soucis
- réutilisabilité des aspects
- facilité de programmation
- adaptation dynamique

En pratique

- composition des aspects ?
- interférences entre aspects (p.e. sécurité + cache)
- étroite dépendance entre les aspects