Conception descendante

IAP1 - cf extrait livre Dubois Ménissier

Catherine Dubois

Dans le contexte :

- spécification,
- analyse et conception,
- programmation,
- VV (vérification, validation - test)

+ documentation (tout au long)

La conception consiste à élaborer à partir de la spécification du problème une solution informatique.

Stratégie générale : décomposer le pb en sous-problèmes et identifier les problèmes pertinents

2 façons de décomposer :

- selon les fonctionnalités, traitements (conception fonctionnelle)
- selon les dnnées à manipuler (conception objet)

Peu importe la méthode de décomposition choisie, la conception peut être :

descendante (*top-down*)

L'approche descendante commence par décomposer le pb initial en sous-problèmes puis chaque sous-problème en de nouveaux sous-problèmes et ainsi de suite jusqu'aux problèmes que l'on peut résoudre par des opérations primitives (ou des fonctions simples).

ou ascendante (*bottom-up*)

L'approche ascendante construit des opérations primitives que l'on assemble pour obtenir des opérations plus complexes et ainsi de suite jusqu'à une opération globale qui résoud le problème initial.

⇒ Décomposition fonctionnelle descendante (IAP1)

Décomposition fonctionnelle descendante

Basée sur la stratégie de résolution *diviser pour régner*

Chaque étape de décomposition est suivie d'une étape de *spécification des sous-problèmes*

Les techniques utilisées dans cette méthode :

- Raffinement successif : chaque étappe de décomposition fait intervenir une seule décision
- Masquage d'information : programmation modulaire
  Les décisions propres à un module sont cachées aux autres modules. Les données et opérations accessibles aux autres modules le sont au travers d'une interface bien définie. Les données et opérations non utiles aux autres modules sont inaccessibles.

La date du lendemain

Etant donné une date, déterminer la date du lendemain

On commence par écrire l'interface de la fonction qui apportera la réponse au problème posé :

```
(* interface lendemain
type : date -> date
args : d
precondition : la date d est valide
postcondition : retourne la date du lendemain
                du jour de date d
raises : lève l'exception Date_invalide
         si d n'est pas valide
tests : à écrire dès maintenant
*)
```

L'analyse rapide du pb amène à la solution informelle suivante (premier niveau d'algorithme)

```
let lendemain jour =
if date est valide then
if jour est le dernier jour du mois
then passer au 1er du mois suivant
else passer au jour suivant
else échec
```

On fait apparaître deux sous-problèmes :

- le calcul du dernier jour d'un mois
- la vérification de la validité d'une date

On prend aussi à ce stade ou plus tard une décision de représentation des données.
$\Rightarrow$ Une date sera représentée par un triplet (no jour, no mois, no année).
Par la suite `date` est un synonyme pour le type `int*int*int`.

On définira ainsi deux fonctions `date_valide` de type `date -> bool` et `nb_jours` de type `int -> int -> int`.

On écrit les interfaces des deux fonctions.

```
(* interface date_valide
type : date -> bool
args : d
precondition : aucune
postcondition : retourne true si d correspond
     à une date valide, false sinon
tests : à écrire dès maintenant *)

(*interface nb_jours
type : int -> int -> int
args : j, a
precondition : m est compris entre 1 et 12,
               a est une année
postcondition : retourne le nombre de
    jours du mois m de l'année a
tests : à écrire dès maintenant *)
```

La fonction `lendemain` devient alors :

```
let lendemain (jour, mois, an) =
  if date_valide (jour, mois, an) then
    if jour = nb_jours mois an
    then if mois = 12 then (1, 1, an+1)
         else (1, mois+1, an)
    else (jour+1, mois, an)
  else raise Date_invalide
```

Et on recommence avec les fonctions manquantes `nb_jours` et `date_valide`.