Cours 3
SDA et conception de logiciel

Les SDA servent à pouvoir construire des logiciels performants en cherchant à combiner des "briques" dont on sait qu'on pourra les implémenter en utilisant des techniques standard. Nous en donnons trois exemples utilisant les tableaux, les piles et les files.

1. Exemple d'analyse algorithmique n'utilisant que des tableaux

On veut écrire un programme qui répond à des questions sur le classement des 500 meilleurs joueurs de tennis. On suppose pour simplifier que tous les noms sont différents, et qu'il n'y a pas d'ex aequo.

1. Données : les noms des joueurs, pour chacun son classement et son nombre de points
2. Services à offrir :
   s1. entrée : un nom ; sortie : le classement et le nombre de points de cejoueur
   s2. entrée :un classement ; sortie : le nom du joueur, son nombre de points
3. Traitements :
   [t0] initialisation des données
   [t1] filtrage ses données par comparaison de chaînes (pour s1)
   [t2] filtrage ses données par comparaison de nombres (pour s2)
4. Représentation des données : un type de données complexe Joueur = {Chaîne : nom du joueur, Entier : classement, Entier : nombre de points}
   500 données de ce type →

version 1 :

Un tableau de 500 joueurs : joueurs[1...500]

joueurs

nom	Classement	points
Ekberg	15	440
Mc Entosh	83	375
Borotra	42	402

5. Appel des traitements
   [t0] doit être exécuté une seule fois, avant toute exécution des autres traitements.
   [t1] et [t2] sont indépendants l'un de l'autre. On peut les appeler depuis un autre programme, qui peut être une simple interface de dialogue au clavier, une requête à travers le réseau ou n'importe quel autre programme qu'on pourra concevoir plus tard.
6. Algorithmique (informellement) :
   [t0] Créer le tableau en lisant un fichier (nombre de données, données)
   [t1-v1] Parcourir le tableau depuis le début. Quand on a trouvé, on renvoie le joueur
   [t2-v1] Parcourir le tableau depuis le début. Quand on a trouvé, on renvoie le joueur

Analyse de la solution :

espace : informations permanentes minimales, pas d'information provisoires ;
réutilisabilité : convient pour n'importe quel problème de ce type, ex :classement des 100 meilleures ventes de disques, des 10000 sites web les plus visités, dirigeants des 3000 plus grosses entreprises du monde, etc.

temps : chaque traitement trouve la réponse à un rang entre 0 et 500 au hasard(pour les noms de non classés : 500) → au moins 250 consultations du tableau par requête en moyenne.

Version 2 :

Si on trie le tableau par nom, on peut faire plus vite pour [t1] par recherche dichotomique : on regarde d'abord au milieu (250) ; si ce n'est pas le joueur cherché, on sait s'il est avant ou après → on a éliminé la moitié des possibilités en une seule consultation, on recommence en regardant au milieu de ce qui reste (125 ou 375, suivant le cas), etc.

4-2. un tableau de 500 joueurs : joueursn[1...500] trié par nom de joueurs

joueursn

nom	classement	points
Borotra	42	402
....		
Ekberg	15	440
.....		
Mc Entosh	83	375

5-2. on a un nouveau traitement [t3] (interne), exécuté une seule fois au début après [t0] :
trier le tableau par noms
[t1-v2] Faire une recherche dichotomique sur le nom du joueur
[t2-v2] Parcourir le tableau depuis le début. Quand on a trouvé, on renvoie le joueur
[t3] voir les algorithmes de tri

Analyse de la solution :
espace et réutilisabilité : exactement les mêmes.
temps :
[t3] est un tri préalable, qui prend du temps une fois pour toutes.
[t1] donne la réponse en 9 consultations du tableau.
[t2] a toujours besoin de 250 consultations en moyenne.

Version 3 :

[t2] est encore trop long. Pour utiliser la même amélioration que [t1], il faudrait un 2ème tableau trié par classement.

4-3. Représentation des données :
un tableau joueursn de 500 joueurs : joueursn[1...500] trié par nom de joueurs
un tableau joueursc de 500 joueurs : joueursc[1...500] trié par classement

joueursn

	nom	classement	points
5	Borotra	42	402
66	Ekberg	15	440
348	Mc Entosh	83	375

joueursc

	nom	classement	points
15	Ekberg	15	440
42	Borotra	42	402
83	Mac Entosh	83	375

5-3. Un nouveau traitement (interne) [t4] d'initialisation du 2ème tableau, exécuté une fois après [t3].

6-3.[t1-v3] faire une recherche dichotomique sur le nom

[t2-v3] faire un accès direct sur le classement
[t4] recopier chaque enregistrement au rang égal à son classement

Analyse de la solution :
espace : 2 fois plus de place
réutilisabilité : inchangée (en supposant qu'il n'y a pas d'ex-aequo)
temps : on ajoute le temps d'une recopie au début. [t1] est en 9 consultations du tableau, [t2] en une consultation.

Version 4 :

C'est toujours risqué d'avoir deux fois les mêmes informations (risque d'incohérence). On peut diminuer la redondance si l'on remarque que dans le tableau trié par classement, le classement est égal au rang dans le tableau. On peut se contenter de mettre dans le tableau le rang de la même donnée dans le premier tableau. Un tableau qui donne l'emplacement des données dans une autre structure de données s'appelle un index.

4-4. un tableau joueursn de 500 joueurs : joueursn[1...500] trié par nom de joueurs

un index i_joueursc de 500 entiers : i_joueursc[1...500] trié par classement

joueursn

	nom	**classement**	**points**
5	Borotra	42	402
66	Ekberg	15	440
348	Mc Entosh	83	375

i_joueursc

	rang alpha
15	66
42	5
83	348

5-4. On remplace [t4] par [t4-v4] : création de l'index, exécuté une foisaprès [t3].
6-4. [t1-v4] faire une recherche dichotomique sur le nom
[t2-v4] faire un accès direct sur le classement
[t4-v4] parcourir joueursn – écrire l'indice de ligne dans i_joueursc au rang égal au classement de cette ligne.

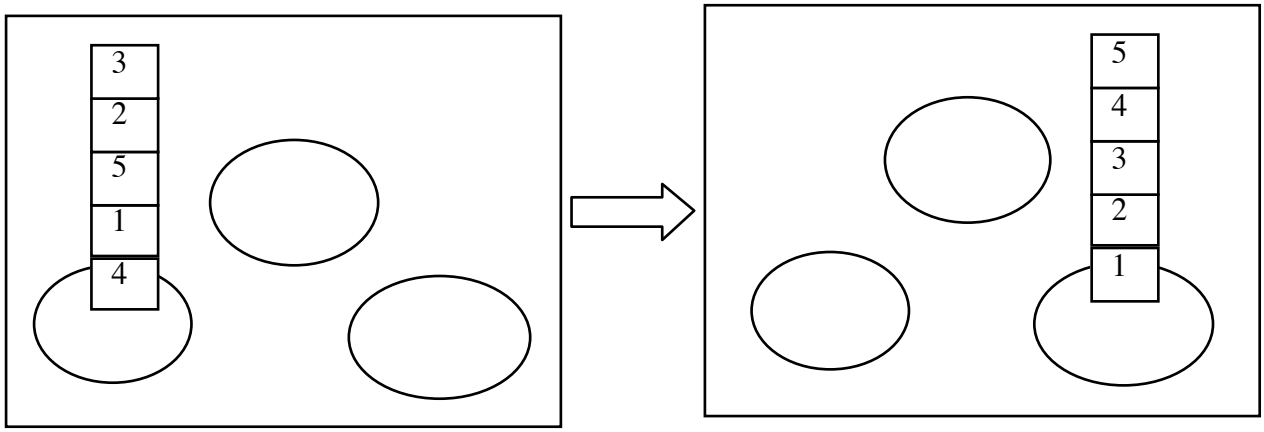
Analyse de la solution :
espace : Le même qu'à l'origine *plus* un entier par ligne
réutilisabilité : inchangée (en supposant qu'il n'y a pas d'ex-aequo)
temps : on ajoute le temps d'un parcours au début (initialisation de i_joueursc)
[t1] est en 9 consultations du tableau,
[t2] en une consultation.

7-. implémentation : si on implémente en C : déclarer deux tableaux globaux static, écrire les traitements, compiler en objet (ou bibliothèque). Si on implémente en java : créer une classe avec 2 tableaux en variables membres privées (ou protected) et les traitements en méthodes publiques. Si on implémente en ada : créer un module. Si on implémente en basic : ....

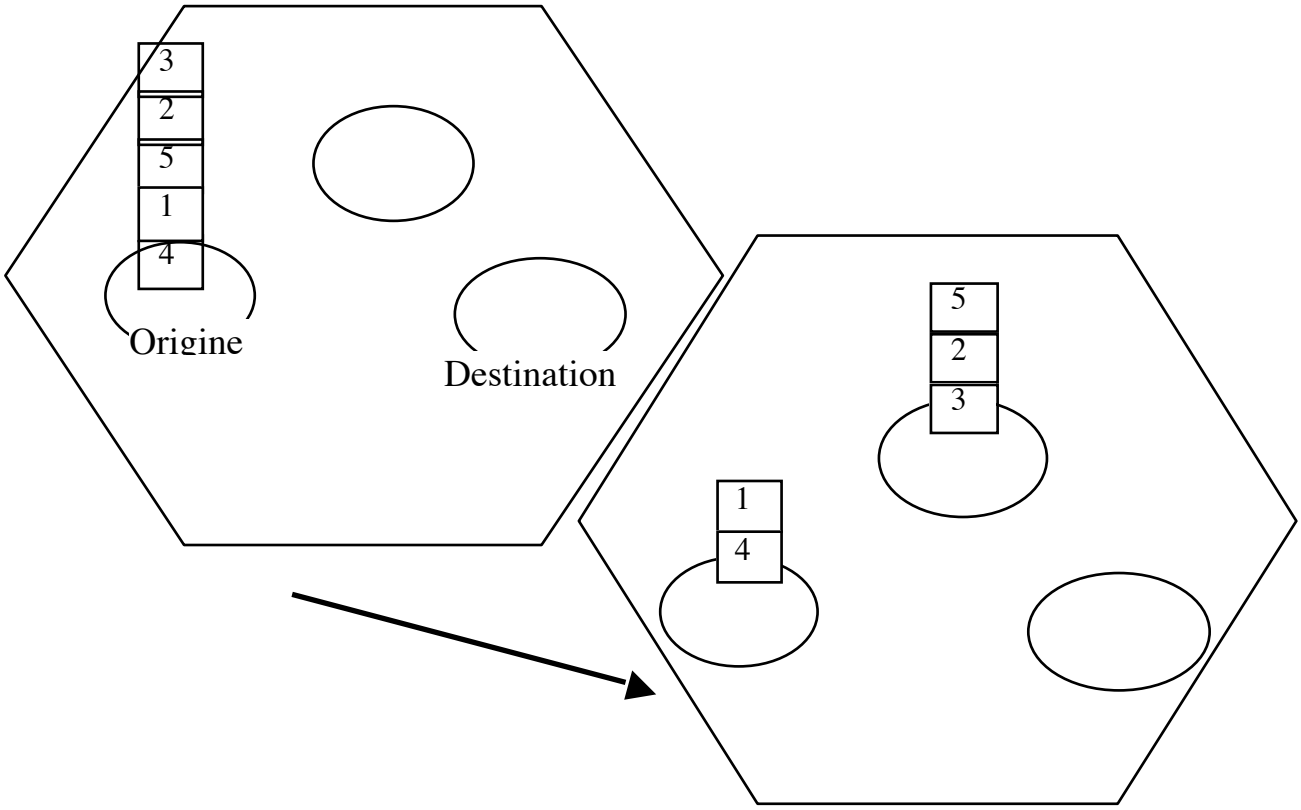
2. Un exemple utilisant des piles

2.1. Analyse et algorithme : le robot manipulateur

2.1.1. Le problème



2.1.2. L'analyse : traitements et représentation des données



1. Données : les emplacements, les palettes et leur position.
2. Services : (s0) prendre connaissance des emplacements ; (s1) ranger les palettes
3. Traitements : (t0) initialisation ; (t1) algorithme de tri (utilise t2 et t3); (t2) trouver sur quel emplacement est la palette cherchée ; (t3) enlever toutes les palettes au dessus de la palette cherchée (utilise t4) ; (t4) déplacer une palette d'un emplacement à un autre.

4. Représentation des données :
   palette = un entier ;
   emplacement = un entier ;
   un tas de palettes = une pile d'entiers (haut du tas = sommet de la pile) ;
   l'état du système à un moment = un tableau de piles (une pile par emplacement)
5. Réalisation des traitements.

Un problème à résoudre : la structure de pile ne permet pas de voir autre chose que le sommet → comment réaliser (t2) ???
Solution :
- on sait où sont toutes les palettes au début
- on sait chaque fois qu'on change une palette d'emplacement
→ noter pour chaque palette son emplacement

4-bis. Ajouter un tableau des positions : position[n] est l'emplacement de la palette n
5-bis. incorporer à t4 : on note la nouvelle position de la palette

2.1.3. L'algorithme

6-bis. Algorithme de tri :

```
paramètres : int n ; /* le nombre d'objets */
        pileInt source, dest, aux ; // source est le tas initial
                                    // dest et aux sont vides
variables :
        int prochain ; //la prochaine palette à ranger sur dest
        int saisi ; // la palette que le robot attrappe
        pileInt position[0..n-1] ; // à chaque instant, position[i] est la pile où est l'objet n° i
        pileInt pileCherche, pilePose; // variables de renommage
initialisation :
        position = {source, source, source,... source } ; // au début, tous sur
        l'emplacement source
        prochain = 0 ; // on range d'abord la palette 0…
        pileCherche = source ; // On la cherche sur source
        pilePose = aux ; // On dépose les palettes au dessus sur aux
fonction :
        fonction pileInt autre(pileInt pile)
        début
            si (pile == source) renvoie(aux) ;
            sinon renvoie(source) ; finsi //car pile == aux
        fin
début :
    tant que (prochain != n) faire
            saisi = prendre(@pileCherche) ;
            si (saisi != prochain)
                poser(@pilePose, saisi);
                position[saisi] = pilePose;
            sinon
                poser(@dest, saisi);
                position[saisi] = dest ;
                prochain = prochain + 1 ;
                si(prochain != n)
```

```
                    pileCherche = position[prochain] ;
                    pilePose = autre(pileCherche) ;
                finsi
            finsi
    fin tant que
fin
```

2.2. L'implémentation

2.2.1. Implémentation de l'algorithme

Les trois emplacements sont représentés par trois piles source, dest et aux qui sont des variables d'entrée de l'implémentation de l'algorithme. En Java, on pourra utiliser un constructeur Robot(source, dest, aux) pour passer ces arguments, ou une méthode manipuler(source, dest, aux) ; en C tout dépend de la technique utilisée pour implémenter les piles ; la plus simple consiste à attribuer à chaque pile un numéro qui sera argument de empiler(), dépiler(). La traduction en C ou en Java du transfert d'une palette d'un emplacement à l'autre est immédiate. En Java, la gestion du tableau position[] et la fonction autre() peuvent être traduits directement : le tableau position[] contiendra en fait des adresses d'objets (fonctionnement interne de Java), comme PileCherche et PilePose. On peut aussi mettre à profit la technique C en numérotant les piles : 0 pour source, 1 pour aux et 2 pour dest. Le tableau position[] est plus compact, et la fonction autre(x) est un échange entre 0 et 1, qui se traduit par pilePose = 1 – pileCherche. Pour faire cela, on utilise un tableau auxiliaire pPal[] qui associe une pile à chaque numéro : pPal[0] = source ; pPal[1] = aux ; pPal[2] = dest. On empilera alors sur pPal[2] pour poser sur dest.

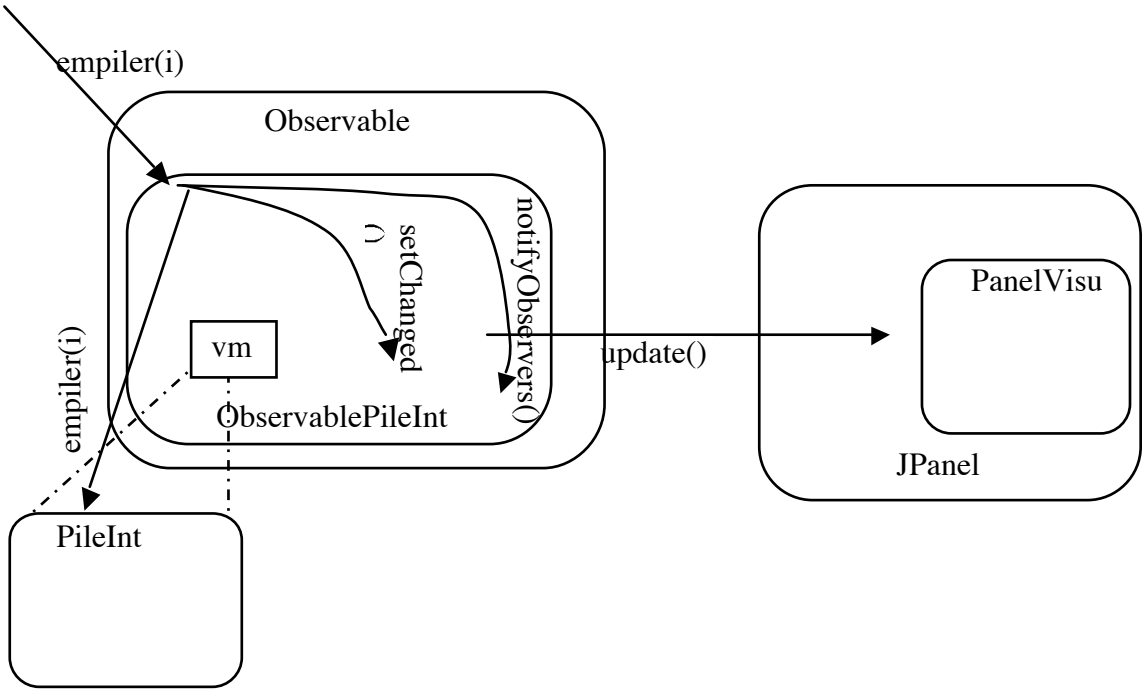
2.2.2. Visualisation

Un avantage secondaire des SDA est de pouvoir créer une architecture de codage claire pour visualiser le déroulement de l'algorithme. Nous utiliserons pour cela le modèle Observateur / Observé qui est fourni par Java. Vous étudierez dans les séances d'algorithmique graphique qui restent ce type d'architecture ; nous en voyons ici le minimum nécessaire.
L'idée est la suivante : on programme de façon quasi indépendante des objets qui réalisent une tâche (les observés : ici le robot et les piles qu'il manipule) et d'autres objets (les observateurs) qui se chargent de visualiser l'état des premiers au fur et à mesure de leur évolution. Nous verrons donc le déroulement de l'algorithme en installant trois visualiseurs de pile et un pour le robot.
Pour qu'un objet puisse être visualisé, il faut qu'il sous-classe la classe Observable. Pour qu'il puisse être un visualiseur, il doit implémenter l'interface Observer. Le dialogue entre les piles et leur d'affichage se passe ainsi :

1. La classe Observable a une méthode addObserver(observer o). Chaque observateur (qui implémente Observer) envoie à ses observés le message addObserver(this) pour s'inscrire comme observateur.
2. Quand un observé exécute une méthode dont il juge qu'elle crée un changement qui pourrait être affiché, il exécute la méthode setChanged(), qui le marque 'changé'
3. Quand un observé décide que son affichage devrait être actualisé, il exécute sa méthode notifyObservers(Object arg). Si il est marqué 'changé', celle-ci envoie à chaque observateur inscrit le message update(this, arg) et annule la marque.
4. Dans l'interface Observer, la méthode update(obs, arg) est un patron. c'est donc à vous de l'implémenter. Quand un observateur reçoit update(Observable o, Object arg), il exécute la méthode que vous avez implémentée pour actualiser l'affichage. L'argument *arg* peut passer une indication de ce qui a changé.

Pour utiliser les classes qui implémentent les SDA (et qui ne sont pas des sous-classes de Observable), on doit passer par l'encapsulation, alors que pour visualiser on peut sous-classer une classe graphique :

- la classe ObservablePileInt est une sous classe de Observable qui a une PileInt en variable membre
- la classe PanelVisu est une sous-classe de JPanel qui implémente update().



3. Un exemple avec des files

3.1. La partie de bataille

La structure de données élémentaire est la **carte** :

```
        public class Carte {
        static String couleur[4] = {"Trèfle";"Carreau";"Cœur";"Pique"} ;
        static String hauteur[13] =
        {"1";"2";"3";"4";"5";"6";"7";"8";"9";"10";"V";"D";"R";"A"}
        int coul, haut ;
        Carte(int c, int h) {
                coul = c ; haut = h ;
        }
        // etc
}
```

Un joueur tient une **main**

- quand il joue …
- quand il gagne …

⇨ la structure de données naturelle est :

```
public class Main extends FileObject {// la carte n'est pas un type primitif
        Main() {
        super() ;
```

```
        }
        // etc.
}
```

Chaque joueur pose des cartes sur le **tas** devant lui
Le joueur qui gagne la bataille ramasse tous les tas.

⇨ la structure de données naturelle est :

```
public class Tas extends PileObject {
        Tas() {
        super() ;
        }
        // etc.
}
```

Pour mélanger les cartes, on prend le **jeu** dans une main et on transfère les cartes par petits paquets dans l'autre main.

Version simplifiée : transférer les cartes une par une, en tirant au sort si on les met au dessus ou en dessous.

⇨ la structure de données naturelle est :

```
public class FileObjectDE extends FileObject {
        Cellule fin ;
        void ajouterEnTete(Object o) {
        // à faire }
        // etc.
}

public class Jeu extends FileObjectDE {
…
        Jeu melanger() {
                nouvJeu = new Jeu ;
                while(! vide()) {
                        Object o = prelever() ;
                        if (Math.random(1) == 1) nouvJeu.stocker(o) ;
                        else nouvJeu.ajouterEnTete(o) ;
                }
                return(nouvJeu) ;
        }
}
```

Une partie, c'est par ex. un jeu, quatre mains et un tas. D'abord le jeu est mélangé, puis il est distribué, puis les cartes sont jouées…