

GÉNIE LOGICIEL AVANCÉ

Cours 7 : Conception

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

vendredi 12 mars 2010

La conception

De la spécification à la conception

- ▶ Spécifier, c'est définir le **quoi**.
- ▶ Concevoir, c'est définir le **comment**.



"There are two ways of constructing a software design : One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."

C.A.R. Hoare

Les difficultés de la conception

- ▶ La conception ne se contente pas d'identifier le problème.
- ▶ Elle tente d'y apporter une **solution valide**.
- ⇒ Un processus **créatif** et **rigoureux**.

Différences entre conception et implémentation ?

- ▶ L'implémentation est la mise en œuvre des choix issus de la conception.
- ▶ L'implémentation doit pouvoir répondre aux contraintes de réalisation sans mettre en cause les choix de conception.

Exemple typique

- ▶ On introduit, lors de l'implémentation, une optimisation qui brise une abstraction issue de la conception.
- ⇒ Ceci témoigne d'une **mauvaise conception** : une spécification non fonctionnelle concernant l'efficacité n'a pas été prise en compte.
- ⇒ Ceci témoigne d'une **mauvaise implantation** : les choix de conceptions doivent toujours être respectés par l'implémentation.



"Premature optimization is the root of all evil in programming."

D. Knuth

Le principe de génie logiciel récurrent

- ▶ Pour bien concevoir un logiciel, il faut pouvoir raisonner sur ce logiciel.
- ▶ Pour pouvoir raisonner, il faut travailler sur des objets de taille raisonnable.
- ⇒ La **programmation structurée**, introduite par Hoare, Wirth et Dijkstra.

Les procédés introduits par la programmation structurée

Avant la programmation structurée

```
10 INPUT "What is your name: ", U$
20 PRINT "Hello "; U$
30 INPUT "How many stars do you want: ", N
40 S$ = ""
50 FOR I = 1 TO N
60 S$ = S$ + "*"
70 NEXT I
80 PRINT S$
90 INPUT "Do you want more stars? ", A$
100 IF LEN(A$) = 0 THEN GOTO 90
110 A$ = LEFT$(A$, 1)
120 IF A$ = "Y" OR A$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; U$
140 END
```

Réduire la liberté pour mieux contrôler

- ▶ Le langage machine offrait une liberté totale :
 - ▶ Grâce aux JUMPS, toute instruction était un point d'entrée potentiel.
 - ⇒ Les triplets de Hoare sont plus compliqués à calculer dans ce cadre.
 - ▶ On pouvait modifier le code lui-même.
 - ⇒ Le raisonnement sur les programmes auto-modifiants est très complexe.
- ⇒ On s'est vite aperçu que tant de liberté était une épée à double-tranchant :
 - ▶ Certes, elle autorisait des optimisations.
 - ▶ Néanmoins, la correction des programmes étaient trop difficiles à vérifier.

La programmation structurée



- ▶ La programmation structurée a mis en avant les concepts suivants :
 - ▶ Les structures de contrôle restreintes.
 - ▶ Les différents niveaux d'abstraction.
 - ▶ Le raffinement.
 - ▶ Les types abstraits de données.
 - ▶ L'architecture d'un logiciel.

Les structures de contrôle restreintes

- ▶ À l'aide d'un GOTO, on peut tout faire... mais le flot de contrôle est alors arbitrairement complexe.
- ▶ Pour raisonner sur un programme, il y a trois grands outils (Dijkstra) :
 - ▶ Le raisonnement par cas.
 - ▶ L'induction.
 - ▶ L'abstraction.

"Use recursive procedures for recursively-defined data structures."

B. Kernighan (du K&R C)
The Elements of Programming Style

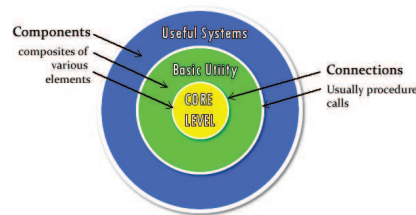
Les structures de contrôle restreintes

- ▶ À l'aide d'un GOTO, on peut tout faire... mais le flot de contrôle est alors arbitrairement complexe.
- ▶ Pour raisonner sur un programme, il y a trois grands outils (Dijkstra) :
 - ▶ Le raisonnement par cas. *Types algébriques, Pattern matching*
 - ▶ L'induction. *Fonctions récursives, itération*
 - ▶ L'abstraction. *Appels de fonctions*

"Use recursive procedures for recursively-defined data structures."

B. Kernighan (du K&R C)
The Elements of Programming Style

Les différents niveaux d'abstraction



- ▶ L'organisation de la complexité d'un système sous la forme de couches, chacune correspondant à un niveau d'abstraction donné, telle que :
 - ▶ Une couche de haut-niveau s'appuie sur les primitives fournies par la couche située sous elle.
 - ▶ Une couche de bas-niveau ne peut pas accéder aux primitives des couches de niveau plus élevé qu'elle.
 - ▶ L'implémentation du système débute par les couches de plus bas niveau.
- ⇒ Nous reviendrons plus en détails sur cette architecture...

Le raffinement

- ▶ On conçoit un système par *raffinements* successifs (c.f. la méthode B).
- ▶ On raffine données et algorithmes *en parallèle*.
- ▶ Mais, comment décider *dans quel ordre* raffiner la spécification ?
- ▶ Il faut maintenir le plus longtemps possible une "notation naturelle".
- ▶ Chaque raffinement correspond à un choix de conception.
- ▶ Si on voit l'espace de conception comme un domaine de recherche, alors les choix de conceptions peuvent être structurés comme un arbre de décision dont les nœuds les plus discriminants sont placés près de la racine.
- ▶ En d'autres termes, il faut comment par raffiner les aspects qui ont les plus importantes conséquences : on définit ainsi des *familles* de programmes répondant à un problème particulier.
- ▶ Cette **structuration explicite de la conception** facilite le raisonnement sur les programmes ainsi que leur réutilisation et leur adaptation.

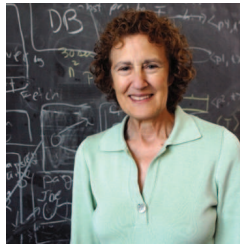
Le raffinement : un exemple

- Pour concevoir un algorithme de tri :
 1. Fait-on des hypothèses sur la distribution des entrées ?
 2. Veut-on un algorithme parallèle ou séquentiel ?
 3. Utilise-t-on une structure de données qui trie par insertion d'éléments ?
 4. ...

Le raffinement : une lecture

- *Program Development by Stepwise Refinement*
Niklaus Wirth
<http://sunnyday.mit.edu/16.355/wirth-refinement.html>

Types abstraits



- Un type abstrait définit une classe de composants totalement caractérisés par un ensemble des opérations.
- L'utilisateur du type abstrait n'a pas connaissance de la représentation concrète des données.
- Le maintien des invariants est **circonscri**t aux définitions des opérations.
- *Programming with abstract data types* – Barbara Liskov

“Architecturer” n’est pas “Programmer”

- Programmation :
 - Fixer les détails d'implémentation.
 - Implémenter/Prouver un algorithme.
- Architecturer (Modulariser) :
 - Minimiser les interdépendances ;
 - Ordonner/Classifier/Organiser les composants ;
 - Rendre explicite les interactions.
- *Programming-in-the large versus programming-in-the-small*
F. DeRemer, H. Kron

La conception des composants

- ▶ Bien concevoir les composants d'un système est donc essentiel puisque :
 - ▶ Leur interface fournit une barrière d'abstraction.
 - ▶ La granularité d'un raffinement se situe généralement au niveau des modules.
 - ▶ Souvent, on a une correspondance "un module = un type abstrait".
 - ▶ De plus, la modularisation rend possible le développement parallèle.

Principes de conception des composants

- ▶ Cacher les "secrets" comme :
 - ▶ la représentation des données (c.f. ADT).
 - ▶ les fonctions internes d'un composant qui ne sont pas nécessaires au client.
 - ▶ les détails d'implémentation.
- ▶ Trouver le lieu des potentielles évolutions :
 - ▶ Si deux composants peuvent évoluer de façon indépendante, ils devraient être cachés l'un à l'autre.
 - ▶ Seuls les services dont l'évolution est peu probable doivent figurer dans les interfaces.
- ▶ Les interfaces doivent être explicitées.
- ▶ Les interfaces doivent être en nombre restreint et petites.
- ▶ Un composant doit correspondre à une unité sémantique cohérente.

Qu'est-ce qu'une interface ?

- ▶ Une **interface** correspond à tout ce qui est **publié** par un composant :
 - ▶ Syntaxique : nom de types, de fonctions, d'événements, de sous-composant. . .
 - ▶ Sémantique : pré/post-condition, . . .
- ▶ Les clients dépendent potentiellement de tout ce qui est publié.
- ▶ Le rapport :
$$\frac{\text{utilisation des fonctionnalités internes}}{\text{utilisation de ce qui est publié}}$$
est un bon indicateur de la qualité de la modularisation.
- ▶ On mesure la **cohésion** d'un composant aux nombres de ses sous-composants qui effectuent des tâches similaires et ont des interactions continues.
- ▶ On mesure l'**interdépendance** entre deux composants *A* et *B* à la quantité de modifications à faire sur *B* lorsque *A* est modifié (et réciproquement).

Différents degrés de cohésion

- ▶ Voici une liste de caractérisation qualitative de la cohésion d'un système :
 1. cohésion aléatoire ;
 2. cohésion logique ;
 3. cohésion temporelle ;
 4. cohésion procédurale ;
 5. cohésion communicationnelle ;
 6. cohésion fonctionnelle ;
 7. cohésion informationnelle.

Ce classement va de la cohésion la plus faible/la moins intéressante à la plus forte/la plus pertinente.

Cohésion aléatoire

- ▶ Rien n'est plus déroutant qu'une modularisation aléatoire.
- ▶ Ceci arrive lorsque l'on décompose le système « parce qu'il faut bien le faire »
- ▶ Exemples :
 - ▶ des critères inflexibles comme :
 - ▶ « Le nombre de lignes d'une fonction doit être inférieur à 30. »
 - ▶ « Le nombre de méthodes d'un objet doit être inférieur à 10. »
 - ▶ etc ...
 - ▶ pour « compiler plus vite », on découpe un module en deux sous-modules interdépendants.
 - ▶ pas de réflexion du tout.

Cohésion logique

- ▶ Les composants sont groupés parce qu'ils partagent des opérations.
- ▶ Les paramètres expriment les variations entre les composants.

```
let dictionary_and_ranker flag data key =  
  let comparison =  
    if flag = true then (* Use the function as a dictionary : compare keys. *)  
      fun (word1, _) (word2, _) → compare word1 word2  
    else (* Use a function as a ranker : compare scores. *)  
      fun (_, score1) (_, score2) → compare score1 score2  
  in  
    (* This part is common. *)  
    Array.sort comparison data;  
    (* End of the common part. *)  
    if flag = true then  
      (* If the function is a dictionary then look for the key in the sorted array. *)  
      match dico data key with  
      | None → 0  
      | Some _ → 1  
    else  
      (* Otherwise, we are looking for the index of the key in this sorted array. *)  
      index_of key data
```

La cohésion logique

- ▶ Les composants obtenus ainsi ont une logique interne complexe.
- ▶ On introduit des interdépendances malvenues.

La cohésion temporelle

- ▶ Les composants intervenant à une même étape d'un processus sont groupés.
- ▶ Elle peut conduire à des duplications de code. . .
- ▶ Cette décomposition suppose l'existence de phases bien délimitées.
- ▶ Exemple : Dans un compilateur, on crée un module pour l'analyse syntaxique, un module pour le typage, un module pour les analyses de flots de contrôle, . . .
Où définir alors le type et les opérations de l'arbre de syntaxe abstraite ?

La cohésion procédurale

- ▶ Les composants sont groupés de façon à réaliser une séquence d'opérations.
- ▶ Exemple :

```
module DisplayMPEGMovie = struct
  let open_file filename = ...
  let decompress_data chunks = ...
  let display_each_frame data = ...
  let display filename =
    display_each_frame (decompress_data (open_file (filename)))
end

module DisplayAVIMovie = struct
  let open_file filename = ...
  let decompress_data chunks = ...
  let display_each_frame data = ...
  let display filename =
    display_each_frame (decompress_data (open_file (filename)))
end
```

- ▶ Cette décomposition peut aussi mener à des duplications de code.

Décomposition communicationnelle

- ▶ Il s'agit de regrouper des composants qui réalisent une séquence d'opérations sur des données de même nature.
- ▶ Exemple :

```
module ImageFilters = struct
  type t = image
  let noise_reducer : t → t = ...
  let mean_convolution : t → t = ...
  let copy : t → t = ...
end

module SoundFilters = struct
  type t = sound
  let noise_reducer : t → t = ...
  let mean_convolution : t → t = ...
  let copy : t → t = ...
end
```

Décomposition fonctionnelle

- ▶ Les composants sont regroupés car ils réalisent ensemble une “fonction”.
- ▶ Ici, on parle de “fonction” au sens général du terme, c'est-à-dire une association **bien définie** entre une entrée et une sortie.
- ▶ Exemple :

```
(** This module translates a C program into a i386 program. *)
module Compiler = struct
  let parse = ...
  let type_check = ...
  let compile : c_program → i386_program = ...
end
```

Décomposition informationnelle

- ▶ Les composants sont regroupés car ils réalisent chacun une (et une seule) action distincte sur une même donnée.
- ▶ Ceci est typique des définitions de types abstraits et de classes d'objet.
- ▶ Exemples

```
module Stack : sig
  type t
  val empty : t
  val push : int → t → t
  val pop : t → int × t
end = struct
  ...
end
```

Différentes formes d'interdépendances

- ▶ Il y a plusieurs formes d'interdépendances que l'on peut classer de la plus néfaste à la moins dangereuse :
 1. Les interdépendances entre contenu.
 2. Les interdépendances de partage de données.
 3. Les interdépendances de flot de contrôle.
 4. Les interdépendances de nommage.
 5. Les interdépendances de flot de données.

Interdépendances de contenu

- ▶ Lorsque un composant dépendant de la définition exacte du code d'un autre composant et réciproquement, on parle de dépendance de contenu.
- ▶ Depuis l'ère de la programmation structurée/de haut niveau, ce type de dépendance n'existe plus et heureusement !
- ▶ Exemple :

```
// Module A
struct A { int x; char y };

// Module B
void foo (A a) { *((unsigned char*)&a) + 4) = 42; }
```

- ▶ L'inversion des champs "x" et "y" dans *A* serait fatal pour le module *B*.

Interdépendances par partage de données

- ▶ La dépendance est créée par une variable globale partagée.
- ▶ Exemple :

```
let cache = Hashtbl.create 13

module A = struct
  let f x = ...
  let cached_f x = ... cache ... (** Look in the cache for [f x]. *)
end
module B = struct
  let g x = ...
  let cached_g x = ... cache ... (** Look in the cache for [g x]. *)
end
(** Neither cached_f nor cache_g are reliable! *)
```

- ▶ Plus de raisonnement local : il faut parcourir tout le code pour connaître les éventuels composants pouvant modifier une variable globale.
- ▶ Les dépendances induites par une variable globale sont implicites !

Interdépendances de flot de contrôle

- ▶ Une dépendance de flot de contrôle est induite par un module *A* sur un module *B* si l'ordre des opérations effectuées par *B* peut être influencé par *A*.
- ▶ Exemple :

```
(* Module A *)
let process data = ... raise ErrorWhenProcessing ...

(* Module B *)
let rec iter data_list = ... process data ...
```

- ▶ Le flot de contrôle de « iter » laisse à penser que tous les éléments de la liste « data_list » sont traités mais cela dépend en réalité du lancement de l'exception par la fonction « process ».

Interdépendances de nommage

- ▶ Lorsque l'on fait référence à un composant par son nom dans un autre composant, on crée une dépendance de nommage.
- ▶ Si on doit renommer un composant, il faut alors renommer tous les composants qui en dépendent.
- ▶ Exemple :

```
let sort array = ... compare x y ... compare u t ...
```

et

```
let sort compare array = ... compare x y ... compare u t ...
```

- ⇒ Dans les deux cas, quel est le travail à effectuer si on veut changer d'opérateur de comparaison sur les éléments du tableau ?

Interdépendances de flot de données

- ▶ On crée une dépendance de flot de données quand un composant attend le résultat d'un calcul d'un autre composant, parce qu'elle lui est nécessaire pour ses propres opérations.
- ⇒ C'est une dépendance **minimale**.
- ▶ Cependant, pas de dépendance du tout, c'est encore mieux !

Évaluer la qualité d'une architecture

Comment savoir si on fait "bien" ?

- ▶ En dehors des grands principes de minimisation des interdépendances et d'augmentation de la cohésion, il n'y a pas de solution miracle pour évaluer la qualité d'une architecture.
- ▶ Et pourtant, une bonne architecture facilite l'obtention des aspects suivants :
 - ▶ Robustesse
 - ▶ Bonne documentation
 - ▶ Évolutivité
 - ▶ Interface bien définie
 - ▶ Maintenabilité
 - ▶ Ergonomie
 - ▶ Compréhensibilité
 - ▶ Réutilisation des composants
 - ▶ Adaptabilité
 - ▶ Développement rapide
 - ▶ Minimisation des erreurs
 - ▶ Lisibilité.
 - ▶ Portabilité.
 - ▶ Efficacité.
 - ▶ Facilité d'apprentissage, de mémorisation, d'utilisation.
 - ▶ Tolérance aux pannes.

Étudier les architectures qui ont fait leur preuve

- ▶ Une solution empirique consiste à réutiliser les solutions qui ont fonctionné dans le passé en les adaptant éventuellement au contexte du système à concevoir.
- ▶ C'est l'idée des **patrons de conception** (sujet du prochain cours).
- ▶ Aujourd'hui, la dernière partie du cours va porter sur une comparaison entre différentes architectures classiques.

Quelques exemples d'architectures

Méthode de comparaison

- ▶ Pour comparer les différentes architectures, nous allons les décrire de façon homogène.
- ▶ Nous allons voir une architecture comme la donnée de composants et d'une description des interactions entre ces composants, que nous appellerons connecteurs. Une architecture sera donc représentée comme un graphe.
- ▶ La description d'un style architectural fixera :
 - ▶ la nature des composants et des connecteurs formant de l'architecture ;
 - ▶ les contraintes sur la façon dont le graphe peut être construit.
- ▶ Les questions qui vont diriger cette étude comparée sont les suivantes :
 - ▶ Comment peut-on décrire ce style ?
 - ▶ Quel est le modèle calculatoire sous-jacent ?
 - ▶ Quels sont les invariants/propriétés maintenus par ce style ?
 - ▶ Quels sont les exemples d'utilisation de ce style ?
 - ▶ Quelles sont ses variantes ?
 - ▶ Quels sont les avantages et les inconvénients de ce style ?

Exemple de problème

- ▶ Nous allons essayer d'imaginer l'instanciation de chacun des styles architecturaux sur le problème suivant :

Concevoir un logiciel de dessin permettant à son utilisateur de placer des formes géométriques sur un canevas et d'enregistrer le résultat dans un format graphique de son choix.

Architecture « de filtres et de tuyaux »

► Composants :

Des **filtres** transformant des flux d'entrées en flux de sorties.

► Connecteurs :

Des **tuyaux**, des conduits pour les flux.

► Modèle calculatoire sous-jacent :

Les flux sont calculés incrémentalement au fur et à mesure de la disponibilité des flux desquels ils dépendent.

► Exemples :

Le shell UNIX, les logiciels de traitement du signal, les systèmes réactifs, les compilateurs, ...

► Variantes :

- Les tuyaux peuvent être bornés ou encore typés.
- La topologie contrainte à être linéaire.
- Le rafraîchissement des données rythmé par une horloge globale.

Exemple

- On se donne un format textuel de description de dessins.
- Ensuite, on crée autant de petits programmes que d'opérations possibles sur ces descriptions de dessin.
- L'utilisation du logiciel est alors très simple :

```
% empty-canvas | square s 10 10 50 | circle c 0 0 10 > fig
% cat fig | rotate s 0.33 | save png fig1.png
% cat fig | translate c 10 10 | save png fig2.png
```

Avantages

► Typique de la philosophie UNIX (qui a fait ses preuves!) :

1. *Small is beautiful.*
2. *Make each program do one thing well.*
3. *Build a prototype as soon as possible.*
4. *Choose portability over efficiency.*
5. *Store data in flat text files.*
6. *Use software leverage to your advantage.*
7. *Use shell scripts to increase leverage and portability.*
8. *Avoid captive user interfaces.*
9. *Make every program a filter.*

Mike Gancarz – *Unix Philosophy*

Avantages

- Chaque filtre est une unité atomique et claire de traitement (forte cohésion).
- Chaque filtre est réutilisable (pas d'interdépendance).
- La composition de deux filtres est la composition de leur comportement.
- Le système est extensible : il suffit de rajouter un nouveau filtre !
- Le modèle de calcul est adapté à la concurrence.

Inconvénients

- ▶ La totale indépendance des filtres rend difficile une synchronisation des composantes du système pour obtenir un comportement global donné.
- ▶ Par exemple, la session suivante de notre logiciel :

```
% empty-canvas | square s 10 10 50 | circle c 0 0 10 > fig  
% cat fig | rotate s 0.33 | save png fig1.png  
% cat fig | rotate s 0.33 | save png fig2.png
```

pourrait être avantageusement interprétée comme

```
% empty-canvas | square s 10 10 50 | circle c 0 0 10 > fig  
% cat fig | rotate s 0.33 | save png fig1.png  
% cp fig1.png fig2.png
```

mais cela suppose une communication globale inter-filtres pour se souvenir qu'une certaine opération a déjà été calculée par un autre filtre.

Architecture « par abstraction des données »

- ▶ **Composants** :
Gestionnaire chargé de l'intégrité d'une ressource
- ▶ **Connecteurs** :
Appels de fonction/Envoi de messages
- ▶ **Modèle calculatoire sous-jacent** :
Séquentiel.
- ▶ **Exemples** :
ADTs, objets, contrôleur matériel, ...
- ▶ **Variantes** :
 - ▶ On peut fournir plusieurs vues différentes sur une ressource.
 - ▶ Le modèle séquentiel peut être remplacé par un modèle concurrent/distribué.

Exemple

- ▶ Notre logiciel de dessin peut être organisé comme :
 - ▶ une bibliothèque de fonctions qui travaillent sur un type **figure** abstrait.
 - ▶ un ensemble de classes d'objets géométriques dont on peut intégrer les instances dans une instance d'une figure.

Avantages

- ▶ L'évolutivité de l'implémentation des représentations et des fonctionnalités est facilitée.
- ▶ La modélisation d'un système comme un ensemble d'« agents qui interagissent » peut être proche de l'organisation de son environnement d'exécution.

Inconvénients

- ▶ Pour interagir, les objets utilisent leurs identités respectives.
- ⇒ Le couplage est plus fort que dans le cas des filtres.
- ▶ Ceci induit une interdépendance de nommage.
- ▶ Le flot de contrôle est **implicite** : lorsque l'on envoie un message à un objet B depuis un objet A, l'objet B peut lui-même interagir avec un objet C sans que l'objet A en soit conscient. C'est une dépendance **implicite**.
- ▶ L'existence d'un état associé aux instances objets et la possibilité qu'un message modifie cet état met à mal la **transparence référentielle**. Envoyer deux fois le même message à un objet ne produit pas nécessairement deux résultats identiques.
- ⇒ Ces deux aspects compliquent le raisonnement sur les objets.

Architecture « à événements implicites »

- ▶ **Composants** :
Des composants abstraits exposants des procédures et des **événements**
- ▶ **Connecteurs** :
Des liaisons entre des procédures et des événements. Un composant annonce un événement et tous les composants qui se sont enregistrés comme observateurs de cet événement sont prévenus.
- ▶ **Modèle calculatoire sous-jacent** :
Une sorte de “soupe” d'interaction (~ mélange chimique)
- ▶ **Exemples** :
La bibliothèque de composants graphiques QT, ECLIPSE, AJAX, ERLANG, JOCAML...
- ▶ **Variantes** :
 - ▶ On peut autoriser tout de même les appels explicites de procédure.

Exemple

- ▶ Pour définir une interface utilisateur, on se dote de composants graphiques.
- ▶ Le système traite les entrées de l'utilisateur comme les clics de souris.
- ▶ Un composant graphique peut être notifié d'un clic sur sa portion de fenêtre en s'inscrivant à l'événement `OnClick`.
- ▶ De même, la figure associée à un composant graphique avec un champ nom "*C*" peut s'inscrire à un événement `OnAction("C", A)` qui est annoncé lorsqu'une action utilisateur *A* est appliquée sur *C*.
- ▶ Une figure avec un champ nom valant "*A*" attachée virtuellement à une figure "*B*" peut s'inscrire à un événement `OnRotate("A", B)` pour tourner elle-aussi.
- ⇒ Si un composant graphique n'est pas affiché, il peut être détruit sans mettre à mal la cohérence des figures du dessin.

Avantages

- ▶ Le couplage entre les composants est nul.
- ▶ La dépendance entre les composants est nulle.
- ▶ Seule compte la convention de nommage des événements.

Inconvénients

- ▶ Réfléchir sur ces systèmes est complexe par leur haut degré de concurrence.
- ▶ Associer des données aux événements pose des problèmes de performances.

Architecture « stratifiée »

- ▶ **Composants :**
Un ensemble d'opérations ayant une visibilité restreinte
- ▶ **Connecteurs :**
Les protocoles qui définissent une visibilité hiérarchisée en couches.
- ▶ **Modèle calculatoire sous-jacent :**
Séquentiels.
- ▶ **Exemples :**
Systèmes d'exploitation, base de données, couche OSI de protocoles réseaux.
- ▶ **Variantes :**
 - ▶ Le modèle calculatoire peut être concurrent ou distribué.
 - ▶ Les couches peuvent autoriser certaines communications bidirectionnelles.

Exemple

1. **Moteur géométrique :**
 - ▶ Est auto-contenu.
 - ▶ Responsable des calculs vectoriels, des transformations, de l'intégrité de la figure...
 - ▶ Fournit un type abstrait permettant d'accéder à une description (en lecture seule) de la figure ainsi qu'un ensemble minimal de primitives qui préservent l'intégrité de la figure.
2. **Bibliothèques d'extensions :**
 - ▶ Utilise exclusivement le moteur géométrique.
 - ▶ Fournit des figures complexes construites uniquement à l'aide des primitives.
3. **Bibliothèques de format de fichier :**
 - ▶ Utilise exclusivement le moteur géométrique.
 - ▶ Fournit des fonctions d'entrées/sorties des figures.
4. **Bibliothèque d'entrées/sorties :**
 - ▶ Utilise exclusivement les bibliothèques de format de fichier.
 - ▶ Donne une vision abstraite de la notion d'enregistrement d'une figure.
5. **Interaction de l'utilisateur :**
 - ▶ Utilise uniquement 2 et 4. (Pourquoi?)
 - ▶ Contrôle les interactions entre l'utilisateur et la figure.
6. **Interface utilisateur :**
 - ▶ Utilise uniquement 5. (Pourquoi?)
 - ▶ Fournit une interface graphique de dessin (avec menus, visualisation, ...)

Avantages

- ▶ Réalise le principe de séparation des problèmes en sous-problèmes :
Chaque couche est une vision plus ou moins abstraite du problème.
- ▶ Les interdépendances sont limitées par construction : on peut changer l'implémentation d'une couche de façon indépendante.

Inconvénients

- ▶ Traverser plusieurs couches peut induire des problèmes de performances.
- ⇒ Problème à mettre en perspective avec la loi de Moore.

Architecture « à repository »

- ▶ **Composants** :
Un composant central détenant les données et des processus indépendants.
- ▶ **Connecteurs** :
Des transactions.
- ▶ **Modèle calculatoire sous-jacent** :
L'exécution des processus est dirigé par le composant central.
- ▶ **Exemples** :
Base de données, MAPREDUCE, système de reconnaissance de formes, ...
- ▶ **Variantes** :
 - ▶ Si c'est l'état du composant central qui provoque l'exécution des processus, on parle d'architecture à **tableau noir** (*Blackboard*).
 - ▶ Si ce sont les modifications apportées au composant central qui provoque l'exécution des processus, il s'agit d'une base de données classique.

Exemple

- ▶ **Scénario 2** :
 - ▶ Supposons que l'on veuille autoriser l'édition d'une figure *via* la collaboration (concurrente) de plusieurs utilisateurs.
 - ▶ La figure peut alors être stockée dans une base de donnée pour autoriser la résolution des modifications concurrentes conflictuelles.
 - ▶ L'utilisation d'une architecture de tableau noir peut servir pour alerter les utilisateurs lorsqu'une contrainte voulue sur le dessin n'est plus vérifiée.
- ▶ **Scénario 1** :
 - ▶ Supposons que la figure soit gigantesque (disons une figure représentant une carte de l'ensemble de la planète).
 - ▶ Grâce à une base de donnée, on se donne la possibilité de faire des recherches concurrentes d'une information en lançant plusieurs processus indépendants qui stockent leur résultat dans un tableau noir (par exemple : existe-t-il une rue "Alphonse Daudet" sur la Terre?).
 - ▶ Lorsque la réponse désirée est obtenue l'état du tableau passe à "résolu" ce qui implique l'exécution des processus d'arrêt de la recherche.

Avantages

- ▶ Tolérance aux pannes.
- ▶ Faible interdépendance, fondamentalement une dépendance aux données.
- ▶ Adapté à la concurrence.
- ▶ Évolutif (même sans arrêter le système !)

Inconvénients

- ▶ Le raisonnement sur ces systèmes est très complexe !
 - ▶ Rien n'empêche les problèmes d'interblocages, de famine, ...
 - ▶ Il y a une forte pression sur la base de données.
- ⇒ Les bases de données distribuées en *peer-to-peer* sont peut-être une solution.

Architecture « par interprétation »

- ▶ **Composants :**
 - ▶ un moteur d'interprétation d'un langage ;
 - ▶ une mémoire ;
 - ▶ une représentation de l'état ;
 - ▶ une représentation du contrôle.
- ▶ **Connecteurs :**
La sémantique du langage.
- ▶ **Modèle calculatoire sous-jacent :**
Le modèle calculatoire est celui du langage interprété.
- ▶ **Exemples :**
 - ▶ La machine virtuelle Java, Javascript, O'CAML, DOT.NET. ...
 - ▶ Les systèmes de transitions (automate, machine à pile, ...).
- ▶ **Variantes :**
 - ▶ On peut faire varier le modèle calculatoire ainsi que les composants. ...

Exemple

- ▶ **Scénario 1 :**
 - ▶ Supposons que l'on veuille rajouter un mécanisme de définition de macros dans le logiciel de dessin. La machine virtuelle correspondante a pour état la figure et devra interpréter un langage formé (au moins) des primitives de dessin.
 - ▶ On facilite en passant l'intégration d'un mécanisme de *plugins* téléchargeables.
- ▶ **Scénario 2 :**
 - ▶ Les interactions entre l'utilisateur et la figure peuvent être complexe et nécessiter plusieurs actions successives de la part de l'utilisateur pour caractériser une certaine fonction du système.
 - ▶ On peut modéliser ce langage des traces d'interaction à l'aide d'un automate, vérifier sa complétude automatique et l'implémenter directement à l'aide de tables interprétées.
 - ▶ On peut faire davantage confiance à cette implémentation de l'automate plutôt qu'à une autre qui serait une traduction manuelle de l'automate sous la forme de code.

Avantages

- ▶ Les interprètes favorisent la portabilité. "*Write once. Run everywhere.*"
 - ▶ La restriction du modèle calculatoire facilite le raisonnement. (C'est particulièrement vrai dans le cas des automates.)
 - ▶ On peut choisir un langage dans lequel le problème se formule bien.
- ⇒ Exemple : SAT, PROLOG, Flash, ...

Inconvénients

- ▶ La conception d'un langage n'est pas toujours aisée.
- ▶ En particulier, trouver le bon rapport entre l'expressivité du langage et les propriétés obtenues grâce aux restrictions par rapport à un langage généraliste.

Synthèse

Synthèse

- ▶ L'architecture des logiciels est une activité créative, sans réponse toute faite.
 - ▶ Il y a cependant deux critères d'évaluation essentiels : la cohésion et les interdépendances.
 - ▶ Pour le reste, il faut étudier les solutions existantes et se les approprier.
- ⇒ C'est de ses solutions issues d'une recherche empirique, les patrons de conception, dont nous parlerons lors du prochain cours.

Références

- ▶ *Software Architecture : Perspective of an Emerging Discipline*
Shaw Garlan (CMU)
- ▶ *Lecture Notes in Software Architecture - ECE 756-3*
Kostas Kontogiannis (Waterloo University)
- ▶ *Software Design*
Nancy Leveson (MIT)