IFT159
Analyse et programmation
Chapitre 3 — Conception descendante

Gabriel Girard

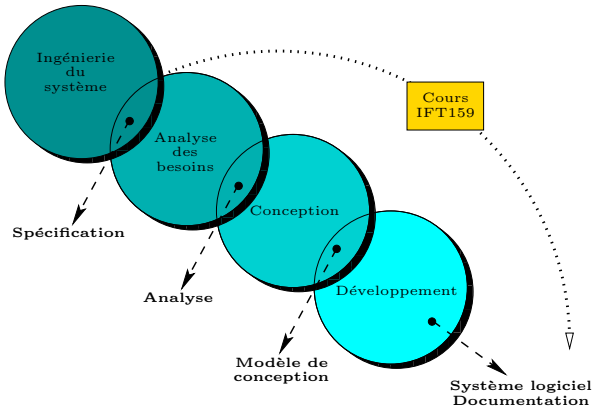
Département d'informatique

UNIVERSITÉ DE SHERBROOKE

7 septembre 2008

Chapitre 3 — Conception descendante

Phases de développement



Spécification

- Texte qui décrit le problème à résoudre
- Doit permettre la compréhension du problème par soi-même
- Décrit les caractéristiques comportementales, fonctionnelles et opérationnelles du système.

Spécification

- Est-on familier avec le domaine d'application ?
- Le problème à résoudre est-il bien défini ?
- Les besoins du client peuvent-ils être modifié avec le temps ?
- Les besoins du client sont-ils bien exprimés ?
- Qui peut répondre à mes questions ?
- Existe-t-il une solution au problème (faisabilité) ?
- Quelles sont les ressources disponibles ?
- Quels sont les résultats à produire ?
- Les données d'entrée sont-elles disponibles ?
- Quels est la durée de vie prévue du logiciel ?
- etc.

Exemples de spécification (1)

Écrivez un programme qui effectue une analyse bidimensionnelle ou profilométrique du relief de la peau humaine basée sur une décomposition spectrale respectant la réalité, c'est-à-dire l'existence de deux familles de plis, l'une liée au derme profond, l'autre au stratum corneum. Une analyse statistique permet de mesurer l'anisotropie du relief et ainsi de suivre le vieillissement cutané. Une étude tridimensionnelle permet à la fois la quantification de la distribution des hauteurs du relief d'un point de vue statistique et morphologique et celle de la répartition et de la densité des plis dans différentes directions de la surface.

Exemples de spécification (2)

- Écrivez un programme qui accepte en entrée un programme écrit en C++ et qui retourne *vrai* si le programme s'arrête ou *faux* autrement.
- Écrivez un programme qui calcule le salaire moyen des travailleurs au noir.
- Écrivez un programme qui calcule la moyenne de trois nombres entiers positifs. On veut que la moyenne soit une valeur entière.

Analyse — méthodologie

- Comprendre le domaine du problème ; (données)
- Définir les fonctions du logiciel ; (fonctionnelle)
- Représenter le comportement du logiciel ; (comportementale)
- Hierarchiser le modèle d'analyse contenant l'information, la fonction et le comportement du logiciel ;
- Aller de l'information essentielle vers les détails du système.

Analyse

On se sert de l'analyse pour :

1. Comprendre et clarifier le problème : ambiguïté, complétude, cohérence ?
2. Déterminer les données d'entrée et de sortie du problème.
3. Déterminer grossièrement le traitement à faire :
   - fixer des hypothèses ;
   - déterminer les formules, équations et constantes ;
   - déterminer les cas d'exception et d'erreurs.
4. Déterminer le comportement du système.

Analyse

Une bonne analyse :

1. Peut servir de documentation à la solution implémentée
2. Détermine :
   - le type des données en entrée et en sortie ;
   - comment les données seront utilisées ;
   - les fonctions les modifiant ;
   - les constantes utilisées ;
   - l'interaction avec l'environnement.

Conception

- Consiste à élaborer, à partir de l'analyse, une solution informatique au problème
- Stratégie générale : décomposer le problème (tâche) en sous-problèmes (sous-tâches) et identifier les problèmes pertinents (modules)
- Deux façons de décomposer
  - selon le traitement (conception fonctionelle)
  - selon les données à manipuler (conception objet)

Conception

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

Descendante (*top-down*)
: L'approche descendante dans laquelle on commence par décomposer le problème 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 à partir d'opérations primitives.

Conception

ou encore

Ascendante *(bottom-up)*
: L'approche ascendante dans laquelle on construit à partir des énoncés du langage de programmation des opérations primitives que l'on assemble pour obtenir des opérations plus complexes et ainsi de suite jusqu'à une opération globale qui résout le problème initial.

Décomposition fonctionnelle descendante

- La décomposition fonctionnelle descendante est basée sur la stratégie de résolution de problèmes *diviser pour régner*.
- Chaque étape de la décomposition est suivie par la spécification de sous-problèmes.

Décomposition fonctionnelle descendante

Les techniques utilisées dans cette méthode sont :

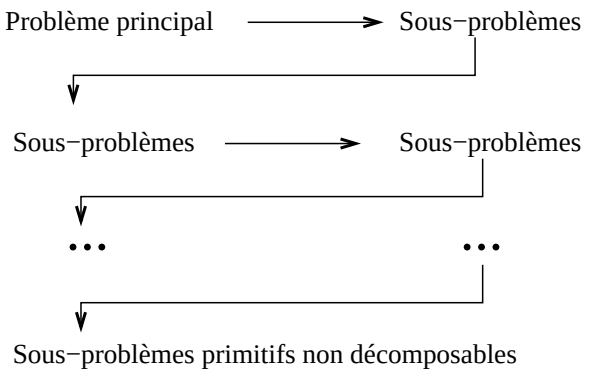
- Le raffinement successif, c'est-à-dire que chaque étape de la décomposition fait intervenir une seule décision de conception.
- Le masquage d'information (critère de décomposition), c'est-à-dire que les décisions propres à un module sont cachées aux autres modules. Les données et opérations accessibles aux autres modules le sont seulement à travers une interface bien définie. Les données et les opérations non utiles aux autres modules sont inaccessibles.

Décomposition de problèmes

- Décomposition selon le traitement
- On obtient des unités de traitement relativement indépendantes (modules)

Décomposition de problèmes

Tâche répétitive :



Conception — Outils

Les deux outils utilisés :

- Diagrammes structurels
  Permettent de visualiser les liens entre un sous-problème et les sous-problèmes générés par la décomposition.
- Pseudo-code ou organigramme
  Permet d'illustrer le traitement fait par une tâche particulière (algorithme).

Pour faire la conception

Questions pertinentes

1. Quelles sont les grandes étapes à réaliser ?
   $\Longrightarrow$ Algorithme
2. Quelles sont les étapes qui devraient être des modules ?
   $\Longrightarrow$ Identification des modules
3. Pour chaque étape qui ne sont pas des modules, on se repose les questions de la conception.
   $\Longrightarrow$ Algorithme

Pour faire la conception

Pour chaque module

1. On se repose les questions pertinentes de l'analyse
   $\Longrightarrow$ Analyse du module
2. On se repose les questions pertinentes de la conception
   $\Longrightarrow$ Identification des modules

Implémentation

- Choix du langage (en théorie)
- Type de programmation
  - Programmation modulaire
  - Programmation structurée
  - Programmation typée
  - Programmation impérative
  - Programmation procédurale (dans la majeure partie du cours)
  - Programmation orientée-objet (à la fin du cours)

Implémentation — Programmation modulaire

Cette méthode divise un système complexe en plusieurs parties, appelées modules, qui peuvent être développées simultanément par plusieurs programmeurs.

Implémentation — Programmation modulaire

Les principes de base de la programmation modulaire sont les suivants :

- Chaque module implémente une seule fonction indépendante des autres.
- Chaque module a un seul point d'entrée et un seul point de sortie.
- La taille d'un module est petite.
- Chaque module est conçu pour être codé et testé séparément.
- Le système est composé de modules (intégration des modules et essais intégrés).

Implémentation — Programmation structurée

idée : forcer le programmeur à utiliser certains types d'énoncés

Objectif : établir que le programme construit satisfait la spécification du problème

Structures :

- la séquence : exécuter une suite d'énoncés ;
- la sélection : exécuter, parmi deux d'énoncés, un énoncé choisi selon une condition donnée ;
- l'itération : exécuter de façon répétitive un énoncé.

Implémentation

Pour démarrer on part de la documentation des étapes précédantes :

- L'analyse décrit nos entrées et sorties
- La conception décrit les algorithmes et les étapes à implémenter
- La décomposition lors de la conception peut servir à la création de modules (dans le cours, elle servira à identifier les fonctions).

Implémentation — Documentation

On doit documenter notre code grâce à des commentaires qui décrivent :

- Ce que fait le programme ;
- Les entrées et les sorties du programme ;
- l'utilité des variables et constantes ;
- Chaque étape importante de l'algorithme ;
- L'utilité de chaque groupe d'instructions.

Qualités d'une documentation

- Concise ;
- Précise ;
- Complète ;
- Simple à maintenir ;
- Uniforme.

La génération automatique de la documentation se fera en utilisant l'outil `doxygen`

Tests et mise au point du programme

- Les résultats doivent être précis.
- Les résultats doivent être conformes à la spécification du problème.
- Le programme doit fonctionner dans tous les cas (preuve).
- Le programme s'exécute dans les délais prescrits.
- Le programme s'exécute dans l'espace mémoire prescrit.

Tests

Quatre types de tests déjà vus :

- Tests unitaires
- Tests d'intégration
- Tests système
- Tests d'acceptation

Nous nous concentrons sur les deux premières catégories.

Tests unitaires

- Vérifier l'implémentation de la conception (fonction, module, etc.) ;
- Assurer partiellement la logique du programme (complétude et correction) ;
- Principe : isoler l'élément à tester pour qu'un autre élément ne fasse pas d'interférence.

Tests d'intégration

- Vérifier que les éléments interagissent correctement ensembles ;
- S'assurer que les objectifs de la conception sont atteints ;
- Combiner et tester les combinaisons d'éléments jusqu'à intégration du système.

Stratégie de tests

- Tests de type boîte noire
  - Tester des données normales
  - Tester des données limites
  - Tester des combinaisons de données
- Tests de type boîte blanche
  - Tester tous les chemins
  - Tester des cas particuliers (probabilité)

Mise au point — Outils

- Traces (commandes d'impression dans le programme)
  - plantage : lorsque le programme produit peu d'affichage, utilisation de `cout` pour voir le moment où le programme plante
  - erreur de logique : ajout de `cout` aux endroits importants pour afficher le contenu de variables clés
- Outils interactifs pour la mise au point : `ddd`
- IDE (Visual Studio, Code : :Blocks, ...)