Notions de base de conception de logiciels

G. Tremblay

Été 2005

There are two ways of constructing a software design. One is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

The Emperor's Old Clothes, CACM, February 1981

C.A.R. Hoare

Je vous aurais écrit une lettre plus courte si j'en avais eu le temps.

Voltaire

1 Introduction : Qu'est-ce que la conception de logiciels?

1.1 De façon générale, qu'est-ce que la conception (de logiciels ou non)?

- Le grand dictionnaire terminologique[1] définit la conception, *au sens général*, comme suit[2] :

  > Activité créatrice qui consiste à élaborer un projet, ou une partie des éléments le constituant, en partant des besoins exprimés, des moyens existants et des possibilités technologiques dans le but de créer un produit ou un service.

- L'activité de conception est une forme de résolution de problèmes. Toutefois, la résolution d'un problème n'est pas nécessairement une activité de conception. Par exemple, une personne qui résout un mot croisé ne fait pas de la conception, par opposition à la personne qui a créé, c'est-à-dire *conçu*, ce même mot croisé.

- Une caractéristique importante de la plupart des problèmes de conception est la suivante : il n'y a pas de solutions complètement "vraies" ou complètement "fausses" ; il y a plutôt des solutions (plus ou moins) "bonnes" ou "mauvaises", qui ont des avantages ou désavantages, qui satisfont ou non divers critères de qualité.

- Dans son sens général, l'activité de conception peut être comprises en termes de cinq concepts clés : objectifs, contraintes, alternatives, représentations et solutions [SB93].

  Le problème de concevoir une automobile permet d'illustrer intuitivement ces notions. L'*objectif* général de concevoir une nouvelle automobile (si on ignore les questions de *marketing* ;) est de développer un moyen de transport. De nombreuses *contraintes* limitent les solutions possibles : type de moteur et de sources d'énergie, routes existantes, politiques

[1] `http://www.granddictionnaire.com`

[2] Ce même grand dictionnaire indique aussi que le terme *design* n'est pas approprié au sens de conception de logiciels : "En français, le mot design est réservé aux activités qui visent une harmonisation esthétique de l'environnement humain à partir des formes données aux productions industrielles. Il ne faut pas l'étendre à toutes les activités créatrices."

et règles environnementales ou de sécurité, etc. À partir de l'objectif général et de ces contraintes, les ingénieurs doivent utiliser leur créativité et leur expérience pour identifier un certain nombre d'*alternatives* possibles pour résoudre le problème. Des *représentations* (maquettes, modèles CAD/CAM, etc.) de ces solutions potentielles peuvent alors être développées pour mieux comprendre les avantages et désavantages de ces diverses alternatives, et ce dans le but d'identifier une *solution* qui servira de point de départ à la production de la nouvelle automobile.

1.2 Le contexte de la conception de logiciels

Pour comprendre le rôle de la conception de logiciels, il est important de comprendre son *contexte*, donc les principales étapes du cycle de développement des logiciels :

- L'analyse des besoins et la spécification des exigences, où l'on tente de comprendre les besoins des usagers et clients et de préciser les exigences requises du logiciel. Ces exigences sont ensuite utilisées comme point de départ de la conception.
- La conception, où les principaux composants du logiciel sont identifiés et leurs interfaces sont décrites (conception architecturale), et où le comportement de ces composants sont aussi décrits (conception détaillée) pour permettre leur réalisation ultérieure — voir plus bas.
- La construction du logiciel (codification et tests unitaires), où les divers composants identifiés à l'étape de conception sont développés et testés.
- L'intégration et les tests (de système ou d'acception), où les divers composants sont intégrés et testés pour assurer qu'ils fonctionnent correctement ensemble et assurer que les exigences initiales sont bien satisfaites.

1.3 Le processus de conception de logiciels

Les deux grandes "étapes" du processus de conception de logiciels sont les suivantes (tel que décrites dans un standard tel que *ISO/IEC 12207 Software Life Cycle Processes* [ISO95]) :

- *La conception architecturale* (parfois aussi appelée conception de haut niveau) qui vise à décrire de quelle façon le système est décomposé et organisé en *composants*, quels sont les liens entre ces composants, quelles sont leurs *interfaces* (qu'est-ce qui est public et utilisable par d'autres composants?) — ce qu'on appelle l'*architecture du logiciel* [IEE00].
- *La conception détaillée* qui décrit le comportement spécifique des divers composants identifiés dans l'architecture.

La conception s'effectue à différents niveaux, donc s'applique à divers types de composants logiciels :

- La décomposition (du système) en sous-systèmes. Lorsqu'un système logiciel est très gros, il peut être nécessaire de le décomposer en sous-systèmes relativement indépendants les uns des autres.

  Par exemple, de nombreux systèmes sont décomposés en *couches*, les relations et communications entre composants se faisant essentiellement entre les couches adjacentes :

  - Une première couche qui gère l'interface personne machine.
  - Une couche intermédiaire qui gère le fonctionnement de l'application, les règles d'affaires associées.
  - Une couche inférieure qui gère l'accès aux (bases de) données.

- La décomposition (d'un sous-système) en modules (ou, plus spécifiquement, en classes dans le cas d'une approche orientée objets) — on verra un peu plus loin différents types de modules.
- La décomposition (d'un module) en routines. Un module regroupe habituellement plusieurs routines, chaque routine étant soit une fonction, soit une procédure :
  - Une fonction retourne un résultat pouvant être utilisé directement dans une expression.
  - Une procédure ne retourne pas de résultat utilisable dans une expression. Elle est donc utilisée pour ses effets de bord, par exemple, lecture ou écriture, modification d'une variable passée en argument ou d'une variable globale.
- La conception interne des routines : une fois qu'on a identifié les grandes lignes de ce que doit faire une routine, on doit alors décrire de façon plus détaillée comment fonctionnera cette routine. Pour ce faire, avant d'écrire du code, il est souvent préférable d'écrire une version simplifiée, en pseudo-code :

> *If you can't write it down in English, you can't code it.*
>
> *Peter Halpern*

1.4 L'objectif de la conception de logiciels

> *When I am working on a problem I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.*
>
> *R. Buckminster Fuller (tiré de [McC04]).*

L'objectif de l'étape de conception est de créer un *modèle de la solution*, c'est-à-dire une représentation simplifiée du programme, plutôt que le programme lui-même. Ce modèle va identifier les différents composants (modules) qui formeront la solution, leur organisation et leur structure (i.e., les liens entre ces composants). À l'aide du modèle ainsi produit, on pourra alors, avant même de commencer à construire le logiciel, tenter d'évaluer la qualité du logiciel qu'on désire construire, s'assurer (tenter de se convaincre) qu'il pourra résoudre le problème.

Ultimement, l'objectif de la conception est de développer un logiciel *de qualité*. La notion de qualité n'est pas une notion facilement spécifiable ou quantifiable. Par contre, on sait que quelques-unes des caractéristiques importantes d'un logiciel bien conçu sont les suivantes :

- Le logiciel est *peu complexe*, donc élégant et facile à comprendre.
- Le logiciel est facile à maintenir.
- Les liens entre les composants sont faibles, ce qui permet de bien séparer les composants, d'en modifier un sans que cela ait trop d'impact sur les autres.
- Les composants sont facilement réutilisables.
- Le logiciel est facilement portable (sur une autre machine, un autre système d'exploitation).
- Le logiciel est *maigre*, c'est-à-dire qu'il n'y a pas de composants ou fonctionnalités inutiles.[3]

[3] Certains considèrent que de nombreux logiciels modernes, particulièrement ceux de *MicroMou*, sont des *obèsiciels* (*bloatware*).

2 Quelques concepts clés pour la conception de logiciels

2.1 Abstraction

La notion d'abstraction est fondamentale en informatique. Selon le grand dictionnaire terminologique, l'abstraction est définie comme suit :

> Démarche de l'esprit qui consiste, au cours d'un raisonnement, à éliminer les aspects les moins pertinents de la réflexion pour ne considérer que ceux qui sont essentiels.
>
> [...]
>
> Par extension, le terme «abstraction» désigne également le résultat (le modèle) issu de la démarche d'abstraction.

Une autre définition de l'abstraction est la suivante : "*[abstraction is] the process of forgetting information so that things that are different can be treated as if they were the same*" [LG01].

Les deux mécanismes clés d'abstraction sont les suivants :

- *Abstraction par paramétrisation* : on fait abstraction de données spécifiques en introduisant des paramètres.
- *Abstraction par spécification* : on fait abstraction de la façon dont un module est mis en oeuvre en référant à une spécification de son interface plutôt qu'à son fonctionnement interne (principe de la boîte noire.)

Ces deux mécanismes conduisent à trois formes d'abstractions couramment rencontrées dans le contexte du développement de logiciels :

- Abstraction procédurale.
- Abstraction de données.
- Abstraction de contrôle

Ces trois formes sont expliquées plus en détails dans les sections qui suivent.

2.1.1 Abstraction procédurale

Cette forme d'abstraction permet de considérer les procédures ou fonctions logicielles comme des "boîtes noires", pour lesquelles on connaît le comportement, mais sans savoir comment ce comportement est obtenu. On donne donc un nom à cette fonction ou opération, avec possiblement des paramètres pour la rendre plus générique.

```
float racineCarre( float x, float precision )
// PRECONDITION
//   x >= 0 && 0.0001 < precision < 0.1
// POSTCONDITION
//   | x - res^2 | <= precision
```

Extrait de code 1: Abstraction procédurale : Une fonction pour calculer la racine carré d'un nombre

Exemple : une routine (fonction) pour calculer la racine carré d'un nombre avec une certaine précision, tel qu'illustrée dans l'extrait de code 1.

```
// Remarque: Dans ce qui suit, "res" dénote le résultat retourné par la fonction.

typedef int Ensemble;

Ensemble creerVide( int n );
// POSTCONDITION
//   res = {}

Ensemble creerPlein( int n );
// POSTCONDITION
//   res = {1, 2, ..., n}

Ensemble ajouter( Ensemble s, int e );
// POSTCONDITION
//   res = s U {e}

Ensemble supprimer( Ensemble s, int e );
// POSTCONDITION
//   res = s - {e}

int estElement( Ensemble s, int e );
// POSTCONDITION
//   res <=> e IN s

int cardinalite( Ensemble s );
// POSTCONDITION
//   res = |s|

int estVide( Ensemble s );
// POSTCONDITION
//   res <=> s == {}
```

Extrait de code 2: Abstraction de données : Un type abstrait `Ensemble`

2.1.2 Abstraction de données

Cette forme d'abstraction permet d'introduire de nouveaux types de données, types qui ne sont pas directement disponibles dans le langage de programmation et dont la représentation interne n'a pas besoin d'être connue (type *abstrait* de données — *ADT = Abstract Data Type*).

Exemple : un type abstrait pour des ensembles *bitset* (ensembles formés de petits entiers compris entre 1 et un certain `n`), dont l'interface est illustrée dans l'extrait de code 2.

2.1.3 Abstraction de contrôle

Cette forme d'abstraction permet de définir un mécanisme de contrôle (par exemple, itération) sans spécifier les détails internes de sa réalisation. Elle est généralement liée à l'abstraction de données.

```
typedef struct IteratorEnsStruct *IteratorEns;

IteratorEns creerIterateur   ( Ensemble s );
int         aUnProchain      ( IteratorEns it );
int         prochain         ( IteratorEns it );
void        detruireIterateur( IteratorEns *it );

// UTILISATION
//   IteratorEns it = creerIterateur( s );
//   while ( aUnProchain(it) ) {
//     int i = prochain(it);
//     ... traiter i ...
//   }
//   detruireIterateur( &it );
```

Extrait de code 3: Abstraction de contrôle : Un itérateur (à la Java) pour le type abstrait `Ensemble`

Exemple : un itérateur, dans le style des collections Java, qui permet d'obtenir l'un après l'autre chacun des éléments d'un `Ensemble s`, tel qu'illustré dans l'extrait de code 3.

2.2 Diviser-pour-régner

```
PROCEDURE resoudreDiviserPourRegner( p: Probleme ): Solution
  SI p est un problème simple ALORS
    sol ← on résout le problème simple p
  SINON
    (p1, ..., pk) ← on décompose le problème p en k sous-problèmes
    POUR i ← 1 A k FAIRE
      soli ← resoudreDiviserPourRegner( pi )
    FIN
    sol ← on combine les solutions sol1, ..., solk des sous-problèmes
  FIN
  RETOURNER( sol )
FIN
```

Algorithme 1: Diviser-pour-régner générique (décomposition en k sous-problèmes)

Au sens purement algorithmique du terme, l'approche diviser-pour-régner est une technique qui vise à résoudre un problème complexe en le divisant en deux ou plusieurs sous-problèmes plus

simples, en résolvant récursivement ces sous-problèmes, puis en combinant les solutions aux sous-problèmes pour obtenir la solution au problème initial. L'algorithme 1 présente une version *générique* d'une telle approche récursive de résolution de problèmes.

Dans le contexte plus général de la programmation impérative, l'approche diviser-pour-régner consiste plus simplement à décomposer une tâche complexe en une série de sous-tâches qui peuvent être traitées de façon indépendante, une stratégie qui est à la base de l'approche *par raffinements successifs* (voir section 5.1).

2.3 Dissimulation de l'information et encapsulation

La dissimulation de l'information est une stratégie de conception introduite par D. Parnas où "*every module [...] is characterized by its knowledge of a design decision which it hides from all others*" [Par72] — on verra cette notion plus en détail à la section 5.2.

Un principe associé est celui de la *séparation de l'interface et de la mise en oeuvre*, où "*[the] interface or definition [of a module is] chosen to reveal as little as possible about its inner workings*" [Par72]. En d'autres mots, une interface publique, connue des clients du module, est spécifiée, et ce de façon indépendante des détails de la réalisation de ce module — il s'agit donc du mécanisme d'abstraction par spécification.

2.4 Couplage et cohésion

Dans les années 70, Yourdon et Constantine [YC79] ont introduit deux concepts qui permettent de caractériser certains aspects de la qualité d'un logiciel, d'évaluer dans quelle mesure le logiciel est *modulaire* : la cohésion et le couplage.

Une qualité importante d'un logiciel bien conçu est d'être *modulaire*, c'est-à-dire qu'un composant peut facilement être modifié ou remplacé sans que cela n'ait un impact trop important sur le reste des autres composants. Règle générale, pour obtenir un système modulaire, il faut que le couplage entre les modules soit *faible* et que la cohésion de chacun des modules soit *forte*, donc que chaque module soit *fonctionnellement indépendant*.

Intuitivement, le couplage entre deux modules représente *la force des liens* qui existent entre ces deux modules, alors que la cohésion représente la force des liens entre les éléments à l'intérieur d'un même module.

Note : dans la présentation qui suit des notions de couplage et de cohésion, nous utiliserons le terme de "module", puisque c'est celui utilisé par Yourdon et Constantine, module étant entendu essentiellement au sens de "routine". Toutefois, ces notions sont aussi applicables dans le cas de modules, classes, etc.

2.4.1 Couplage

Le couplage entre deux modules décrit la force des dépendances qui existent entre ces deux modules. Plus le couplage est élevé, plus des changements dans un module risquent d'avoir des répercussions sur l'autre module.

Le couplage peut être caractérisé par le nombre de connexions entre les modules. Par exemple, une routine qui possède un seul paramètre, donc qui reçoit un seul argument, conduit à un niveau de couplage moins élevé avec ses clients qu'une routine qui compte six ou sept paramètres.

Le couplage peut aussi être caractérisé par la visibilité des connexions entre les modules. Par exemple, un couplage *explicite* par l'intermédiaire d'arguments est préférable à un couplage *implicite* par l'intermédiaire de variables globales. Ainsi, dans l'extrait de code 4, le couplage entre `additionner` et `additionner1` est clair et explicite (passage par valeur d'un argument). Par contre, le couplage entre `add` et `addUn` est implicite — invisible au niveau des en-têtes des routines — parce qu'il est lié à l'utilisation de la variable globale `increment`.

```
int additionner( int x, int y )
{ return( x + y ); }

int additionner1( int x )
{  return( additionner(x, 1) ); }


static int increment;

int add( int x )
{ return( x + increment ); }

int addUn( int x )
{ increment = 1;  return( add(x) ); }
```

Extrait de code 4: Couplage explicite par opposition à couplage implicite

Il existe un certain nombre de formes typiques de couplage entre modules. Les formes suivantes, qui représentent des niveaux de couplage nul ou faibles, sont considérées acceptables :

- Aucun couplage : les deux modules ne sont aucunement liés entre eux, c'est-à-dire qu'aucun module n'utilise ou n'interagit avec l'autre. Dans ce cas, on peut donc changer un module sans que cela n'ait d'impact sur l'autre module.
- Couplage par données simples : les deux modules s'échangent des données simples (types de base ou tableaux homogènes) par l'intermédiaire de paramètres.
- Couplage par objets : les deux modules s'échangent des objets ou des types abstraits (structures de données complexes mais fonctionnellement cohésives) par l'intermédiaire de paramètres.

Par contre, les formes de couplage suivantes sont généralement à éviter :

- Les deux modules communiquent par l'intermédiaire de variables globales.

  Notons que cette forme de couplage est acceptable lorsque les modules ainsi couplés sont en fait des routines et que les variables globales partagées par les routines sont strictement privées au module. L'extrait de code 5 illustre ainsi la mise en oeuvre d'une machine abstraite modélisant une pile d'entiers, où le couplage entre les routines se fait par l'intermédiaire des variables globales `nbElements` et `elems`, variables qui sont strictement privées (préfixe `static`) et qui représentent l'état interne de l'objet modélisé par le module.

- Un module passe un indicateur de contrôle à l'autre module pour lui indiquer ce qu'il doit faire. Cette forme de couplage n'est acceptable que si l'indicateur de contrôle est un type spécifiquement défini pour cette tâche (par exemple, type défini avec un `enum`).

Certains langages permettent aussi des formes de couplage plus pernicieuses. Par exemple, en C, une routine pourrait manipuler certaines adresses pour accéder à l'espace local associé à une autre routine, ce qui est dangereux et inacceptable.

En conclusion, l'objectif est donc de minimiser le plus possible le couplage entre deux modules, de rendre les liens les plus clairs et explicites possibles. De façon générale, on peut se poser la question suivante pour tenter d'évaluer le couplage possible relatif à un module : si on désirait rendre ce module disponible dans une bibliothèque publique, de quelle façon le module devrait-il recevoir ses données et retourner ses résultats pour qu'il soit le plus simple possible à utiliser et à comprendre?

```
#define MAX_ELEMS 100

static int nbElements = 0;
static int elems[MAX_ELEMS];

int estVide()
{ return( nbElements == 0 ); }

int estPleine()
{ return( nbElements == MAX_ELEMS ); }

int sommet()
// PRECONDITION: !estVide()
{ return( elems[nbElements-1] ); }

void empiler( int x )
{ elems[nbElements++] = x; }

void depiler()
{ nbElements--; }
```

Extrait de code 5: Couplage (acceptable) par variables globales privées pour une machine abstraite

2.4.2 Cohésion

Alors que le couplage mesure la force des associations entre deux modules distincts, la cohésion mesure la force d'association entre les éléments composant un module donné. La cohésion permet donc de mesurer le degré auquel les activités qui forment le module sont liées à des tâches similaires, à des tâches *fonctionnellement reliées entre elles*. L'objectif est donc d'obtenir un module ayant la cohésion la plus forte possible, donc un module qui réalise *une seule et unique tâche, fonctionnellement et logiquement complète*.

Différents niveaux de cohésion ont été initialement identifiés. Le premier niveau est celui considéré comme étant le niveau idéal de cohésion :

- Cohésion fonctionnelle : le module contient tous les éléments qui contribuent à la réalisation d'une seule et unique tâche, logiquement complète — le fait d'être *logiquement complète* ou non dépend du niveau d'abstraction. La tâche peut donc être complexe, en autant que le module réalise une seule fonction liée au problème à résoudre.

  Lorsqu'on peut décrire la tâche effectuée par le module à l'aide d'une courte phrase — donc en répondant à la question "Que fait le module?" — et que cette description utilise des termes spécifiques et précis, avec un verbe actif et un complément d'objet spécifique, alors il s'agit généralement d'un module avec une cohésion fonctionnelle. Par exemple, dans le cas de routines : *Calculer le salaire hebdomadaire, Déterminer l'âge d'une personne, Supprimer un fichier, Obtenir le nom du client, Affecter une tâche à un employé, etc.*

Les niveaux suivants peuvent être acceptables, mais ne sont pas idéaux :

- Cohésion séquentielle : les éléments du module sont liés par le fait que la sortie d'une des sous-tâches est utilisée comme entrée par la sous-tâche suivante, mais que l'ensemble de ces sous-tâches ne forment pas une fonction complète.

  Une description d'un module qui contient plusieurs verbes, avec des indications d'ordre, est souvent le signe d'un niveau de cohésion séquentielle. Par exemple : un module qui, à partir de la date de naissance d'un employé et de sa date d'entrée en fonction, calcule son

âge et sa date de retraite (cette dernière dépend de l'âge et de la date d'entrée en fonction). Dans ce cas, il serait préférable d'avoir deux routines distinctes : une première qui calcule l'âge à partir de la date de naissance, une seconde qui calcule la date de retraite à partir de l'âge et de la date d'entrée en fonction.

- Cohésion communicationnelle : les éléments du module sont liés par le fait que les diverses sous-tâches utilisent ou manipulent les mêmes données, mais que l'ensemble de ces sous-tâches ne forment pas une fonction complète.

  Une description d'un module qui contient plusieurs verbes est souvent le signe d'un niveau de cohésion communicationnelle. Par exemple : un module qui, à partir du code permanent d'un étudiant, détermine son adresse et son programme d'études (les deux sous-tâches utilisent le code permanent). Dans ce cas, il serait préférable d'avoir deux routines indépendantes : une première pour obtenir l'adresse, une autre pour déterminer le programme d'études.

- Cohésion temporelle : les éléments du module sont liés simplement par le fait qu'ils doivent être exécutés plus ou moins en même temps, sans nécessairement de liens entre eux.

  Un exemple typique de cette forme de cohésion est une routine qui initialise l'ensemble des éléments du programme — fichiers, structures de données, etc. Bien qu'il puisse être acceptable d'avoir une telle routine qui coordonne l'ensemble des activités d'initialisation, il est préférable que chaque sous-tâche d'initialisation soit clairement déléguée à une routine spécifique.

Les formes suivantes de cohésion sont, quant à elles, considérées comme généralement inacceptables, puisqu'elles conduisent à du code mal organisé, difficile à comprendre et à modifier :

- Cohésion procédurale : les éléments du module sont impliqués dans des activités différentes et possiblement sans liens entre elles (donc ne sont pas liées par le traitement des mêmes données), mais ces activités doivent quand même être exécutées dans un ordre spécifique.

  Un exemple serait une routine qui lirait à l'écran le nom d'un étudiant, son code permanent, son adresse, alors qu'une autre routine lirait son numéro de téléphone.

- Cohésion logique : les éléments du module sont des activités d'une même catégorie, mais sans véritables liens entre elles. Un tel niveau de cohésion survient souvent lorsqu'une routine reçoit en argument un indicateur de contrôle et doit ensuite effectuer une tâche spécifique associée à cet indicateur.

  Par exemple, une routine qui calculerait le salaire de n'importe quel type d'employé (employé payé avec un salaire horaire et des heures supplémentaires, cadre avec salaire annuel fixe, employé à commission, etc.) aurait très certainement une cohésion logique.

  Notons que, toute règle ayant des exceptions, il existe certains types de routine avec une cohésion logique qui sont acceptables : les routines de type *disptacher*, qui ne font qu'identifier une caractéristique de l'entité à traiter et qui appellent une routine spécifique de traitement. L'extrait de code 6 illustre une telle routine.

```
typedef
   enum{
      IMPRIMANTE_POSTSCRIPT,
      IMPRIMANTE_A_POINTS,
      IMPRIMANTE_PDF
   }
   TypeImprimante;

void imprimer( Document d, TypeImprimante ti )
{
  switch( ti ) {
    case IMPRIMANTE_POSTSCRIPT:
      imprimerPostscript( d );
      break;
    case IMPRIMANTE_A_POINTS:
      imprimerAPoints( d );
      break;
    case IMPRIMANTE_PDF:
      imprimerPDF( d );
      break;
  }
}
```

Extrait de code 6: Un cas acceptable de routine avec une cohésion logique : un *dispatcher*

3 Les caractéristiques d'une routine de qualité (abstraction procédurale)

3.1 Pourquoi créer une routine

Il existe diverses raisons justifiant la création d'une routine :

- Une routine permet de réduire la complexité du code. Le corps de la routine permet de dissimuler divers détails qu'on peut ensuite oublier (principes d'abstraction par spécification et de dissimulation d'information).
- Une routine permet d'introduire une abstraction (procédurale) qui permet, à l'aide d'un nom bien choisi, de bien documenter son rôle.
- Une routine permet d'éviter de dupliquer du code.
- Une routine permet d'améliorer la portabilité : les détails liées à un environnement ou une machine peuvent être dissimulés dans une routine.

3.2 Comment bien nommer une routine

Le nom d'une routine devrait décrire *ce que fait* la routine (quoi?) et non pas comment elle le fait. Ce nom qui décrit ce que fait la routine devrait être le plus complet possible. Il ne faut pas hésiter à utiliser des noms significatifs et comptant plusieurs mots (mais sans toutefois abuser).

Plus spécifiquement :

- Pour nommer une fonction, il faut décrire la valeur retournée par la fonction, en utilisant un prédicat dans le cas d'une fonction retournant un booléen, par exemple : `prochainClient()`, `couleurDuFond()`, `date()`, `sommet()`, `estVide()`, `estEnTete()`, etc.

- Pour nommer une procédure, on utilise un verbe actif (à l'infinitif) préférablement suivi d'un complément d'objet lorsqu'approprié, par exemple : `depiler()`, `traiterClient()`, `calculerTotal()`, `calculerTauxHoraire()`, `imprimerDocument()`, etc.

3.3 Comment bien déclarer les paramètres d'une routine

- Il est préférable de définir les paramètres dans un ordre logique et d'utiliser le même ordre pour les routines semblables.

  Par exemple, l'ordre suivant est proposé par certains auteurs : *i*) les paramètres associés à des entrées qui ne sont pas modifiées ; *ii*) les paramètres associés à des entrées mais qui sont aussi modifiées ; *iii*) les paramètres utilisés uniquement pour retourner un résultat.

  Exemple :

```
void procedure calculerTauxHoraire( CategorieEmploye c,
                                    int anciennete,
                                    float *tauxHoraire );
```

  Soulignons toutefois que certaines librairies C indiquent plutôt les paramètres qui sont modifiés en début de liste — par exemple, `strcpy`, dont le premier argument est la destination et le deuxième argument est la source.
- Si plusieurs routines utilisent des paramètres similaires, alors il est préférable d'utiliser le même ordre et les mêmes noms.
- La routine doit utiliser tous les paramètres. Si un paramètre n'est pas utilisé, alors il devrait être supprimé de l'en-tête de la routine.
- Les paramètres utilisés pour retourner un code de statut ou un code d'erreur devraient apparaître à la fin de la liste des paramètres.
- Sauf rares exceptions, une routine ne devrait pas avoir plus de sept (7) paramètres.

3.4 Quand utiliser une procédure par opposition à une fonction

Plusieurs auteurs considèrent qu'une fonction, qu'on utilise pour retourner un résultat utilisable dans une expression, ne devrait jamais avoir d'effets de bord, c'est-à-dire ne devrait modifier ni ses arguments, ni de variables globales. Ainsi, dans le langage Ada, les arguments d'une fonction doivent absolument être des arguments passés en mode `IN`, donc non modifiables.

Par contre, dans le langage C, on rencontre fréquemment des fonctions qui retournent un résultat — donc résultat de type *autre* que `void` — et qui ont des effets de bord. La fonction `getchar()` est un premier exemple. Un autre exemple est `printf`, qui retourne le nombre d'octets effectivement imprimés (donc un code de statut ou d'erreur) en plus d'imprimer sur `stdout`.

4 Ce qu'est un module

Bien qu'initialement (dans les années 70), on utilisait le terme module pour une routine (cf. section 2.4), de nos jours on utilise le terme module pour des structures plus complexes qu'une simple routine. Ainsi :

- Une routine est un simple sous-programme, qui réalise soit une fonction, soit une procédure.
- Un module regroupe un ensemble de routines et, possiblement, des données manipulées par ces routines.

Il existe différents types de modules, selon ce qui est exporté par le module — des routines seulement, un ou plusieurs types *et* des routines associées — et selon les liens qui existent entre les routines — partage ou non de variables globales :

- Bibliothèque de routines : un tel module exporte uniquement des routines, sans que des donnée soient partagées par ces routines.
- Machine (abstraite) : un tel module exporte uniquement des routines, mais ces routines partagent des données (privées) qui représentent l'état d'un objet modélisé par le module.
- Type abstrait : un tel module exporte un type principal ainsi que des opérations qui manipulent les instances de ce type. La différence entre un type abstrait qui définit une collection de valeurs et un type abstrait qui définit une classe d'objets dépend de la façon dont les instances du type sont utilisées et manipulées.
  - Collection de valeurs : les entités du type ne possèdent pas d'état interne qui évolue dans le temps, donc elles représentent des *valeurs* (immuables).
  - Classe d'objets : les instances du type possèdent un état interne qui évolue dans le temps, donc représentent des *objets* (mutables).

Les sections qui suivent décrivent plus en détails les différentes formes de module à l'aide de quelques exemples. Soulignons que ces exemples visent essentiellement à illustrer les différences entre les diverses formes de module. Ces exemples sont loin d'être complets : pas de documentation détaillée des interfaces, pas de vérification des erreurs, absence de certaines opérations clés, etc.

Les deux parties d'un module : interface vs. mise en oeuvre

Soulignons que chaque exemple présenté dans les sections qui suivent sera défini à l'aide de deux fichiers :

- Un fichier d'interface (avec une extension `.h`) qui *déclare* les divers éléments *exportés* par le module — donc les éléments visibles et utilisables par les clients du module —, que ces éléments soient des routines ou des types.
- Un fichier de mise en oeuvre (avec une extension `.c`) qui *définit* les divers éléments déclarés dans le fichier d'interface, ainsi que possiblement d'autres éléments qui, eux, sont strictement *privés* au module.

Rappelons la différence entre déclaration et définition : une déclaration ne fait qu'indiquer le type de l'entité, sans allouer l'espace mémoire associé (dans le cas d'une variable) ou sans donner le code associé (dans le cas d'une routine). Par contre, une définition alloue l'espace mémoire associé ou donne le code associé. Un fichier d'interface ne contient que des déclarations alors qu'un fichier de mise en oeuvre contient des déclarations *et* des définitions.

4.1 Bibliothèque de routines

Le fichier d'interface 1 déclare l'interface d'un module qui représente une bibliothèque de routines. Les routines sont logiquement liées entre elles (calcul de fonctions trigonométriques), mais ne partagent aucune donnée. On remarque l'utilisation d'une définition conditionnelle de macro : lorsqu'un client inclut le fichier `biblio-trigo.h`, si le symbole `BIBLIO_TRIGO_H` n'est pas encore défini, alors on le définit et on inclut donc les déclarations des diverses fonctions. Par contre, si le symbole `BIBLIO_TRIGO_H` est déjà défini, alors aucune nouvelle déclaration ne sera incluse.

Fichier d'interface 1 Bibliothèque de routines trigonométriques : `biblio-trigo.h`

```
#ifndef BIBLIO_TRIGO_H
#define BIBLIO_TRIGO_H

float sinus  ( float x );
float cosinus( float x );
float tan    ( float x );
float cotan  ( float x );

#endif
```

```
#include "biblio-trigo.h"

float sinus( float x )
{ ... }

float cosinus( float x )
{ ... }

float tan( float x )
{ ... }

float cotan( float x )
{ ... }
```

Fichier de mise en oeuvre 1: Bibliothèque de routines trigonométriques : `biblio-trigo.c`

L'intérêt d'une telle définition conditionnelle dans l'en-tête d'un fichier d'interface est donc qu'aucune erreur ne sera générée si le fichier d'interface est inclus plusieurs fois de suite dans un même fichier. Ainsi, le code suivant ne générera aucune erreur :

```
#include "biblio-trigo.h"
#include "biblio-trigo.h"
...
```

Le fichier de mise en oeuvre 1 illustre la structure qu'aurait le corps d'une telle bibliothèque de routines. Tel qu'illustré dans cet exemple, lorsqu'on décompose un module en une partie interface et une partie mise en oeuvre, il est toujours préférable d'inclure le fichier d'interface dans le fichier de mise en oeuvre — ceci assure que si des modifications sont effectuées à la partie interface, ces modifications seront correctement réflétées dans la partie mise en oeuvre.

4.2 Machine abstraite

Le fichier d'interface 2 décrit l'interface d'un module qui réalise une pile sous forme d'une machine abstraite. Une machine est une entité mutable, donc un objet avec un état qui évolue dans le temps. Les structures de données qui définissent l'état de la pile sont dissimulées et encapsulées dans le fichier de mise en oeuvre : un client du module n'a pas besoin de connaître ces structures de données pour utiliser la pile.[4]

Ce module, en supposant évidemment que les noms des opérations représentent correctement ce qu'elles font, définit bien une machine abstraite puisque le module exporte uniquement des routines et que ces routines modifient (dans le fichier de mise en oeuvre) certaines structures de

[4]Soulignons que certaines des routines ont été brièvement annotées d'un commentaire indiquant la précondition requise pour que l'opération s'effectue correctement.

Fichier d'interface 2 Machine pour une pile : `pile.h`

```
#ifndef PILE_H
#define PILE_H

int  estVide();
int  estPleine();
int  sommet();          // PRECONDITION: !estVide()
void empiler( int x );  // PRECONDITION: !estPleine()
void depiler();         // PRECONDITION: !estVide()

#endif
```

```
#include "pile.h"

#define MAX_ELEMS 100

static int nbElements = 0;
static int elems[MAX_ELEMS];

int estVide()
{ return( nbElements == 0 ); }

int estPleine()
{ return( nbElements == MAX_ELEMS ); }

int sommet()
{ return( elems[nbElements-1] ); }

void empiler( int x )
{ elems[nbElements++] = x; }

void depiler()
{ nbElements--; }
```

Fichier de mise en oeuvre 2: Machine pour une pile : `pile.c`

données partagées par les diverses routines. Ceci est illustré dans le fichier de mise en oeuvre 2. On remarque que les structures de données communes aux routines exportées par le module — le nombre d'éléments dans la pile (`nbElements`) et les éléments présents (`elems`) — sont strictement privées au module puisqu'elles ont été précédées du mot clé `static`.

4.3 Collection de valeurs

Fichier d'interface 3 Collection de valeurs pour des fractions : `fractions.h`

```
#ifndef FRACTIONS_H
#define FRACTIONS_H

typedef struct {
  int numerateur;
  int denominateur;
} Fraction;

Fraction fraction( int num, int den );         // PRECONDITION: den != 0

Fraction plus   ( Fraction f1, Fraction f2 );
Fraction fois   ( Fraction f1, Fraction f2 );
Fraction inverse( Fraction f1 );               // PRECONDITION: !estZero(f1)

int      estZero( Fraction f1 );
int      egales ( Fraction f1, Fraction f2 );

#endif
```

Le fichier d'interface 3 présente l'interface d'un module qui définit un type abstrait représentant une collection de *valeurs*. Il s'agit bien d'un type abstrait puisque le module, dans sa partie publique, déclare un nouveau type (`Fraction`) accompagné des opérations qui permettent de manipuler les instances de ce type. Dans le cas présent, le type n'est pas un type *opaque*, c'est-à-dire qu'on connaît la définition du type.[5]

La caractéristique importante d'un tel module est que chaque élément du type `Fraction` dénote *une valeur*. Par définition, une valeur ne change pas — une valeur est `immuable`. Par exemple, la valeur 2 reste toujours la même valeur. Si on effectue l'opération "2+4", une nouvelle valeur est retournée (6), sans aucun effet sur les valeurs 2 et 4. Dans un module définissant une collection de valeurs, les routines ne modifient donc pas l'état des arguments qu'elles reçoivent. Comme l'illustre la mise en oeuvre des opérations `plus`, `fois` et `inverse`, présentée dans le fichier de mise en oeuvre 3, ces opérations reçoivent donc une ou plusieurs valeurs du type, effectuent certains calculs, puis retournent une nouvelle valeur du type, sans modifier les valeurs reçues en argument.

Lorsqu'on utilise un module définissant une collection de valeurs, les manipulations se font généralement à l'aide d'expressions, comme on le ferait pour des valeurs numériques primitives. L'extrait de code 7 illustre cette caractéristique : il s'agit là d'un segment de code qui permet de tester (de façon minimale) les diverses opérations exportées par le module `fractions`.

4.4 Classe d'objets

Le fichier d'interface 4 présente l'interface pour un type abstrait qui représente une classe d'objets. Comme dans l'exemple précédent, il s'agit bien d'un type abstrait puisque le mo-

[5] Il aurait été possible de définir le type `Fraction` de façon opaque, en introduisant un pointeur, mais ceci aurait compliqué inutilement la mise en oeuvre.

```
#include "fractions.h"

Fraction fraction( int num, int den )
{
  Fraction f;
  f.numerateur = num;
  f.denominateur = den;
  return( f );
}

Fraction plus  ( Fraction f1, Fraction f2 )
{
  Fraction f;
  f.numerateur = f1.numerateur*f2.denominateur + f2.numerateur*f1.denominateur;
  f.denominateur = f1.denominateur * f2.denominateur;
  return( f );
}

Fraction fois  ( Fraction f1, Fraction f2 )
{
  Fraction f;
  f.numerateur = f1.numerateur*f2.numerateur;
  f.denominateur = f1.denominateur * f2.denominateur;
  return( f );
}

Fraction inverse( Fraction f1 )
{
  Fraction f;
  f.numerateur = f1.denominateur;
  f.denominateur = f1.numerateur;
  return( f );
}

int estZero( Fraction f )
{
  return( f.numerateur == 0 );
}

int egales( Fraction f1, Fraction f2 )
{
  return( f1.numerateur * f2.denominateur == f2.numerateur * f1.denominateur );
}
```

Fichier de mise en oeuvre 3: Collection de valeurs pour des fractions : `fractions.c`

```
#include <assert.h>
int main()
{

  Fraction f0 = fraction( 1, 1 );
  Fraction f1 = fraction( 10, 20 );
  Fraction f2 = fraction( 2, 4 );

  assert( egales(f1, f2) );

  Fraction f3 = plus(f1, f2);
  assert( egales(f3, f0) );

  Fraction f4 = fois(f1, f3);
  assert( egales(f4, f2) );

  assert( egales( f4, inverse(inverse(f4)) ) );

  assert( estZero( fraction(0, 10) ) );
  assert( estZero( fraction(0, 1200) ) );

  printf( "OK\n" );
}
```

Extrait de code 7: Programme pour tester le type abstrait `fractions`

Fichier d'interface 4 Classe d'objets pour des comptes bancaires : `comptes.h`

```
#ifndef COMPTES_H
#define COMPTES_H

typedef struct CompteStruct *Compte;

Compte creerCompte( char *nomClient, int soldeInitial );

int   solde  ( Compte c );
char* client ( Compte c );
int   folio  ( Compte c );
void  deposer( Compte c, int montant );
void  retirer( Compte c, int montant );

#endif
```

```
#include "comptes.h"

struct CompteStruct {
  char *client;
  int solde;
  int folio;
};

#define PREMIER_FOLIO 10000

Compte creerCompte( char *nomClient, int soldeInitial )
{
  static int prochainFolio = PREMIER_FOLIO;

  Compte c = (Compte) (malloc (sizeof(struct CompteStruct)));
  c->client = nomClient;
  c->solde = soldeInitial;
  c->folio = prochainFolio;
  prochainFolio += 1;
  return( c );
}

int solde( Compte c )
{ return( c->solde ); }

char* client( Compte c )
{ return( c->client ); }

int folio( Compte c )
{ return( c->folio ); }

void deposer( Compte c, int montant )
{ c->solde += montant; }

void retirer( Compte c, int montant )
{ c->solde -= montant; }
```

Fichier de mise en oeuvre 4: Classe d'objets pour des comptes bancaires : `comptes.c`

dule, dans sa partie publique, déclare un nouveau type (`Compte`) accompagné des opérations qui permettent de manipuler les instances de ce type. Dans le cas présent, le type est *opaque*, c'est-à-dire qu'on connaît pas la définition détaillée du type (on sait seulement qu'il s'agit d'un pointeur vers une structure, laquelle n'est pas définie).

La caractéristique importante d'un tel module est que chaque instance du type `Compte` dénote *un objet*. Par définition, un objet possède une identité fixe (son "nom") mais un état interne qui évolue dans le temps — un objet est donc une entité `mutable`.

Dans un module définissant une classe d'objets, les routines reçoivent habituellement comme premier argument l'instance de la classe sur laquelle l'opération doit s'appliquer (c'est-à-dire l'objet qu'on désire manipuler). Cette instance est indiquée par un pointeur qui ne sera pas modifié.[6] Par contre, pour certaines opérations (les *mutateurs*) — par exemple, `deposer` et `retire` —, la structure de données référée par ce pointeur sera modifiée. Ces caractéristiques sont illustrées dans le fichier de mise en oeuvre 4. Soulignons qu'une opération qui ne modifie pas l'état de l'objet — par exemple, `solde`, `client`, `folio` — est appelée un *observateur*, alors qu'une opération qui crée un nouvel objet — par exemple, `creerCompte` — est un *créateur*

```
#include <assert.h>

int main()
{
  Compte c1 = creerCompte( "Nellie", 100 );
  deposer( c1, 100 );
  retirer( c1, 50 );
  assert( folio(c1) == PREMIER_FOLIO );
  assert( solde(c1) == 150 );
  assert( strcmp( client(c1), "Nellie" ) == 0 );

  Compte c2 = creerCompte( "Sara", 100 );
  deposer( c2, 100 );
  retirer( c2, 150 );
  assert( folio(c2) == PREMIER_FOLIO+1 );
  assert( solde(c2) == 50 );
  assert( strcmp( client(c2), "Sara" ) == 0 );

  Compte c3 = c2;
  deposer( c3, 1000 );
  assert( solde(c2) == 1050 );

  printf( "OK\n" );
}
```

Extrait de code 8: Programme pour tester le type abstrait `comptes`

Lorsqu'on utilise un module définissant une classe d'objets, les opérations se font alors essentiellement à l'aide d'instructions. L'extrait de code 8 illustre cette caractéristique : il s'agit là d'un segment de code qui permet de tester (de façon minimale) les diverses opérations exportées par le module `comptes`.

Autre particularité d'une classe d'objets : une variable de type `Compte`, puisqu'elle est simplement un pointeur, permet de créer des *synonymes* (des *alias*). Ainsi, dans la dernière série d'instructions de l'extrait de code 8, un dépôt est effectué dans le compte `c3`, l'effet étant aussi visible dans le compte `c2` puisque les deux variables réfèrent au même compte.

[6] Exception possible : une opération pour détruire un objet.

5 Stratégies de modularisation

5.1 Décomposition fonctionnelle

L'idée générale de l'approche par décomposition fonctionnelle, brièvement décrite à la section 2.2, consiste à utiliser l'approche diviser-pour-régner pour décomposer une tâche complexe en sous-tâches plus simples. On dit aussi que cette approche est une approche *descendante* (*top-down*) par *raffinements successifs*, l'une des premières méthodes de conception proposée dès les années 70 [Wir71].

Le point de départ du processus de raffinement est un énoncé, à un haut niveau d'abstraction, de la fonction logicielle à réaliser. Le processus de raffinement consiste alors à élaborer (à raffiner) cet énoncé initial en introduisant, petit à petit, divers détails. Le processus de raffinement se termine lorsque suffisamment de détails ont été spécifiés pour que tous les éléments soient facilement réalisables dans le langage de programmation choisi pour réaliser le programme.

Un élément clé de cette stratégie est d'introduire de façon judicieuse, au cours du processus de raffinement, des abstractions appropriées (procédurales ou de données) — en d'autres mots, de déléguer les détails à des routines.

5.1.1 Exemple : lecture et calcul de la moyenne d'une série de nombres entiers *à grande précisions*

Soit le problème suivant: on veut lire, sur `stdin`, une série de nombres entiers à grande précision (avec un nombre possiblement très grand de chiffres). On suppose que la liste des nombres se termine dès que le nombre 0 est rencontré. Une fois tous les nombres lus, on veut imprimer la moyenne de ces nombres (plus précisément, la partie entière de cette moyenne, puisqu'on travaille uniquement avec des entiers).

```
DEBUT
  nbLus ← 0
  total ← 0
  REPETER
    val ← obtenir la prochaine valeur
    SI val n'est pas égale à 0 ALORS
      nbLus ← nbLus + 1
      total ← total + val
    FIN
  JUSQUE val est égale à 0
  SI nbLus ≠ 0 ALORS
    imprimer val divisé par nbLus
  FIN
FIN
```

Algorithme 2: Première version d'un algorithme pour le problème du calcul de la moyenne d'une série de valeurs

L'algorithme 2 présente une première ébauche, en pseudo-code, d'un programme pour résoudre ce problème. Certaines des opérations indiquées, si on manipulait des entiers simples (`int`), seraient faciles à coder en C. Toutefois, comme on désire manipuler des entiers à grande précision, il nous faut donc introduire des abstractions appropriées. Dans un premier temps, nous allons simplement introduire certaines abstractions procédurales. L'algorithme 3 présente le programme résultant — on remarque qu'il nous a fallu introduire une abstraction même pour une valeur simple telle que 0, puisque l'entier 0 du langage C ne sera pas le même entier que le 0 à grande précision.

```
DEBUT
  nbLus ← 0
  total ← ZERO
  REPETER
    val ← lireEntier()
    SI !egale(val, ZERO) ALORS
      nbLus ← nbLus + 1
      total ← plus(total, val)
    FIN
  JUSQUE egale(val, ZERO)
  SI nbLus ≠ 0 ALORS
    ecrireEntier( diviser(val, Entier(nbLus)) )
  FIN
FIN

Variable et opérations pour la manipulation de grands entiers :
  ZERO: Entier;
  FONCTION lireEntier(): Entier
  FONCTION ecrireEntier( e: Entier );
  FONCTION egale( e1, e2: Entier ): Booléen
  FONCTION plus( e1, e2: Entier  ): Entier
  FONCTION diviser( e1, e2: Entier  ): Entier
  FONCTION Entier( n: int ): Entier
```

Algorithme 3: Deuxième version d'un algorithme pour le problème du calcul de la moyenne d'une série de valeurs

Les prochaines étapes, que nous n'illustrerons pas, consisteraient alors à choisir une structure de données appropriée pour représenter les entiers à grande précision, puis à raffiner les opérations de manipulation de ces entiers. Puisque plusieurs de ces opérations seraient liées à la manipulation d'entiers à grande précision, on introduirait aussi un nouveau module basé sur une abstraction de données, plus précisément, un type abstrait définissant une collection de valeurs.

5.1.2 Décomposition fonctionnelle sous Unix avec des filtres et pipelines

Pour certains problèmes, il est possible dans le contexte Unix d'utiliser des filtres[7] et des pipelines (*pipes*) pour résoudre un problème complexe à l'aide d'une approche par décomposition fonctionnelle. Par exemple, supposons qu'on ait le problème suivant :

- Entrée : un fichier `commentaires.txt` contient des lignes de la forme suivante, où les différents noms peuvent ou non être distincts, et où les différentes lignes sont ordonnées selon la date (en ordre croissant) :

  Date Nom Commentaire
  Date Nom Commentaire
  Date Nom Commentaire
  ...

- Sortie : On désire obtenir le nombre de noms *distincts* qui sont présents dans le fichier `commentaires.txt`.

[7] Sous Unix, un *filtre* est simplement un programme qui lit une série de données sur `stdin` et qui produit sur `stdout` une série de résultats.

L'algorithme suivant utilise une stratégie de décomposition fonctionnelle pour résoudre ce problème — le problème initial est décomposé en sous-problèmes *plus simples*, lesquels sous-problèmes sont résolus de façon indépendante :

```
DEBUT
  noms ← obtenir la liste des noms contenus dans le fichier commentaires.txt
  noms_tries_et_uniques ← trier (de façon unique) les noms
  nb ← nombre d'éléments dans noms_tries_et_uniques
  RETOURNER( nb )
FIN
```

Or, cet algorithme peut être mis en oeuvre de façon *très* simple dans un environnement Unix, et ce au niveau même du *shell* puisque des commandes existent déjà pour réaliser directement les diverses sous-tâches identifiées :

```
awk '{print $2;}' < commentaires.txt | sort -u | wc -l
```

5.2 Dissimulation de l'information

David Parnas, dans les années 70, a introduit une approche de conception basée sur la "dissimulation de l'information" (*information hiding*) [Par72, Par79], approche qui est à la base de l'approche moderne basée sur les objets. Contrairement à l'approche de décomposition fonctionnelle où on introduit un module pour encapsuler un traitement (module au sens de routine, donc), dans l'approche par dissimulation de l'information on introduit un module pour encapsuler *un secret*. Un secret représente une décision de conception, un détail que l'on désire dissimuler pour rendre le système modulaire et facilement modifiable.

Les divers types de secrets ou de décisions de conception qui peuvent être dissimulés peuvent être, entre autres, les suivants :

- Les décisions clés de conception, par exemple, les algorithmes utilisés pour résoudre certaines tâches, les structures de données complexes utilisées pour résoudre le problème, etc.
- Les aspects du systèmes qui sont sujets à changement, par exemple, les dépendances face au matériel ou au système d'exploitation, les formats des entrées/sorties.

La première catégorie vise donc à dissimuler la complexité de certains aspects, alors que l'autre catégorie vise à dissimuler des éléments qui sont sujets à changement. Ces divers secrets peuvent être encapsulés dans des routines, donc à l'aide d'abstractions procédurales, ou dans des modules qui représentent des abstractions de données.

6 Conclusion

En guise de conclusion, nous présentons certaines règles inspirées de la philosophie Unix présentées par E. Raymond dans son livre "*The Art of Unix Programming*" [Ray04] :

- *Rule of Modularity: Write simple parts connected by clean interface.*
- *Rule of Clarity: Clarity is better then cleverness.*
- *Rule of Composition: Design programs to be connected to other programs.*
- *Rule of Simplicity: Design for simplicity; add complexity only where you must.*
- *Rule of Transparency: Design for visibility to make inspections and debugging easier.*

- *Rule of Repair: When you must fail, fail noisily and as soon as possible.*
- *Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.*
- *Rule of Optimization: Prototype before polishing. Get it working before you optimize it.*

Références

[HT00] A. Hunt and D. Thomas. *The Pragmatic Programmer—From journeyman to master.* Addison-Wesley, 2000. [QA76.6H858].

[IEE00] IEEE. Recommended practice for architectural description of software-intensive systems. Technical Report IEEE Std 1471-2000, IEEE, New York, NY, 2000.

[ISO95] ISO/IEC. Information technology—software life cycle processes. Technical Report ISO/IEC Std 12207: 1995, ISO/IEC, 1995.

[KP01] B.W. Kernighan and R. Pike. *La programmation — En pratique.* Vuibert, Paris, 2001. [QA76.6K48814].

[LG01] B. Liskov and J. Guttag. *Program Development in Java.* Addison-Wesley, 2001.

[McC93] S. McConnell. *Code Complete — A Practical Handbook of Software Construction.* Microsoft Press, Redmond, WA, 1993. [QA76.76D47M32].

[McC04] S. McConnell. *Code Complete — A Practical Handbook of Software Construction (Second Edition).* Microsoft Press, Redmond, WA, 2004.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972.

[Par79] D.L. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. on Soft. Eng.*, 5:128–138, 1979.

[Ray04] E.S. Raymond. *The Art of UNIX Programming.* Addison-Wesley, Boston, MA, 2004.

[RK03] P.N. Robillard and P. Kruchten. *Software Engineering Process with the UPEDU.* Addison-Wesley, 2003.

[SB93] G. Smith and G. Browne. Conceptual foundations of design problem-solving. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(5):1209–1219, 1993.

[TA04] G. Tremblay (Associate Editor). Software design. In A. Abran, J.W. Moore, P. Bourque, and R. Dupuis, editors, *Guide to the Software Engineering Body of Knowledge (2004 Version)*, chapter 3, pages 3.1–3.12. IEEE Computer Society Press, Los Alamitos, CA, 2004. `http://www.swebok.org`.

[TP05] G. Tremblay and A. Pons. Software design: An overview. In R.H. Thayer, editor, *Software Engineering, Vol. 1: The Development Process (3rd edition).* IEEE Computer Society Press, Los Alamitos, CA, 2005. To be published (sept. 2005).

[Tre04] G. Tremblay. *Modélisation et spécification formelle des logiciels (édition revue et augmentée).* Loze-Dion Éditeurs, Inc., Montréal, Qué., quatrième trimestre 2004. 635 p.

[Wir71] N. Wirth. Program development by stepwise refinement. *CACM*, 14(4):221–227, 1971.

[YC79] E. Yourdon and L.L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Programming and Systems Development.* Prentice-Hall, 1979. [QA76.6Y67].

Remarque : Ces notes de cours ont été écrites en se basant principalement sur les références suivantes :

- [TP05]
- [McC04]