Notes de Cours - Programmation 1
2011 - 2012

Pascal Ferraro

18 septembre 2011

Table des matières

Chapitre 1

Introduction à la programmation en Langage C

Sommaire

Objectifs Cette unité d'enseignement traite un grand nombre de techniques de base de la programmation en vue d'augmenter la lisibilité et la modularité du code, et donc d'améliorer la maintenabilité.

Équipe pédagogique

- Chargé de cours : Pascal Ferraro
- Chargés de TD : Aurélie Bugeau, Marie-Christine Counilh, Stefka Gueorguieva et Jean-Claude Ville.

Organisation

- Cours Magistraux 20
- Travaux Dirigés 40

Mode d'évaluation :

- Contrôle Continu (CC) (coeff 0.5)
  - Devoir Surveillé obligatoire (DS) - 1h20
  - Travaux Pratiques (TP)
  - 3 Tests (T)
  - Note de CC = (0.5*DS)+(0.25*TP)+(0.25*T)
- Première Session
  - Examen (EX1) : 1h30 (coeff 0.5)
  - Résultat 1ère session : 0.5*EX1 + 0.5*max(CC,EX1)
- Deuxième Session
  - Epreuves 2nde session : Examen (EX2)- 1h30
  - Résultat 2nde session : 0.5*EX2 + 0.5*max(CC,EX2)

L'objectif de ce premier chapitre est de :

1. présenter la chaîne allant de l'écriture d'un programme à son exécution,
2. d'introduire la notion de module avec les concepts d'interface et d'implémentation,
3. de montrer la compilation séparée,
4. d'illustrer les qualités d'un module/programme et de montrer comment une fonction peut être testée.

1.1 Quelques Références

1. Braquelaire (J.-P.). – Méthodologie de la programmation en `C`. – Dunod, 2000, troisième édition.
2. Cassagne (B.). – Introduction au langage `C`. – `http://clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html`.
3. Delannoy (C.). – Programmer en langage `C`. – Eyrolles, 1992.
4. Faber (F.). – Introduction à la programmation en `ANSI-C`. – `http://www.ltam.lu/Tutoriel_Ansi_C/`.
5. Kernighan (B.W.) et Richie (D.M.). – The `C` programming language. – Prentice Hall, 1988, seconde édition.
6. Loukides (M.) et Oram (A.). – Programming with GNU software. – O'Reilly, 1997.

1.2 Historique

Le `C` a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du `C` dans le livre *The C Programming language*. Le `C` devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI `C`. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990.

1.3 Le Langage `C`

Le langage `C` est un langage de programmation qui appartient au paradigme de la *programmation impérative*. Il n'est pas consacré qu'à la programmation système. C'est à la fois un langage :

- compilé,
- typé,
- avec des instructions bas-niveau (au plus près de la machine).

1.4 D'un programme à son exécution

1.4.1 Forme générale d'un programme C

Un programme source C se présente sous forme d'une collection d'objets externes (variables et fonctions), dont la définition est éventuellement donnée par des fichiers séparés.

Tout programme C est composé d'un **programme principal**, c'est une fonction particulière qui porte toujours le nom de `main`.

Une variable est un objet manipulé par le programme, qui possède un nom et un type. Le type définit l'ensemble des valeurs possibles pour l'objet.

En C, tout module (sous-programme) porte le nom de *fonction*. Une fonction est un sous-programme. Il s'agit d'un mécanisme permettant de donner un nom à un bloc afin de pouvoir le réutiliser en différents points du programme. Une fonction permet d'enfermer certains traitements dans une "boite noire", dont on peut ensuite se servir sans se soucier de la manière dont elle a été programmée. Toutes les fonctions, dont le programme principal, sont constituées d'un bloc. Un bloc est une suite d'instructions à l'intérieur d'accolades "{ }".

1.4.2 Un exemple de programme

Exemple le fichier `progSomme.c`

```
#include <stdlib.h>
#include <stdio.h>

int somme(int);

int main(int argc, char **arg)
{

  int i = 10;
  printf("La somme des %d premiers entiers est %d \n", i, somme(i));

}

int somme(int i)
{

  int resultat = 0;
  for (int k = 0; k <= i; k++)
    {
      resultat += k;
    }
  return resultat;
```

```
}
```

1.4.3 Commentaires sur le code

- Un bloc commence par une accolade ouvrante { et se termine par une accolade fermante }.
- Les commentaires se mettent entre `/*` et `*/`. Les commentaires peuvent alors être sur plusieurs lignes. On peut également utiliser `\\`, mais alors le commentaire ne s'étale que sur une seule ligne.
- `void main()` définit une fonction appelée `main` qui ne reçoit pas d'arguments. C'est le nom du programme principal. `void` signifie que la fonction ne retourne rien. On parle souvent dans ce cas de *procédure*.
- La directive `#include` permet l'inclusion des fichiers `stdlib.h` et `stdio.h`
- Il y a une différence entre **définition** et **déclaration** :
  - La déclaration de la fonction `somme`
  - La définition de la fonction `somme`
- Toutes les instructions sont terminées par un " `;` "
- On définit un programme principal par la fonction `main`
- On utilise la notion de *bloc* d'instructions.
- On utilise le type `int`
- On définit une variable et on l'initialise en même temps
- On utilise la fonction `printf` : le programme principal (fonction `main`) appelle la fonction `printf`, de la bibliothèque `stdio.h`, pour afficher la séquence de caractères "`La somme des %d premiers entiers est %d /n`" dans laquelle chaque % indique l'endroit où l'un des arguments suivants (le deuxième, troisième, *etc.*) doit se substituer, et sous quel format l'afficher. Le caractère `/n` est un caractère spécial permettant le retour à la ligne
- On utilise une boucle d'instruction `for ( ... ; ... ; ... )` Le code qui suit immédiatement le `for` va être éxécuté autant de fois qu'il y a de passages dans la boucle.
  A l'intérieur des parenthèses, il y a trois parties :
  1. initialisation : `iCompteur=1k=0` Elle s'effectue une seule fois, avant l'entrée dans la boucle.
  2. test de la condition : `k<=i` Cette partie contrôle le déroulement de la boucle. Cette condition est évaluée :
     - Si la condition est vraie, on exécute le corps de la boucle (`resultat+=k`), puis on passe à la phase d'incrémentation (`k++`).
     - Si la condition est fausse, la boucle se termine.
  3. incrémentation : l'instruction `k++` est équivalente à `k = k + 1`. Après cette phase, la boucle reprend en 2.
- On passe des paramètres et on récupère une valeur en retour de la fonction `somme`.

1.5 La compilation

Le `C` est un langage compilé (par opposition aux langages interprétés). Cela signifie qu'un programme C est décrit par un fichier texte, appelé fichier source. Ce fichier n'étant évidemment

pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé compilateur. La compilation se décompose en fait en 4 phases successives :

1. Le traitement par le préprocesseur : le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source ...).
2. La compilation : la compilation proprement dite traduit le fichier généré par le préprocesseur en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.
3. L'assemblage : cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé fichier objet.
4. L'édition de liens : un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier dit exécutable.

Les différents types de fichiers utilisés lors de la compilation sont distingués par leur suffixe. Les fichiers source sont suffixés par `.c`, les fichiers prétraités par le préprocesseur par `.i`, les fichiers assembleur par `.s`, et les fichiers objet par `.o`. Les fichiers objets correspondant aux librairies pré-compilées ont pour suffixe `.a`.

Le compilateur `C` sous UNIX s'appelle `cc`. On utilise de préférence le compilateur `gcc` du projet GNU. Ce compilateur est livré gratuitement avec sa documentation et ses sources. Par défaut, `gcc` active toutes les étapes de la compilation. On le lance par la commande :

```
gcc [options] fichier.c [-llibrairies]
```

Par défaut, le fichier exécutable s'appelle `a.out`. Le nom de l'exécutable peut être modifié à l'aide de l'option `-o`.

Les éventuelles librairies sont déclarées par la chaîne `-llibrairie`. Dans ce cas, le système recherche le fichier `liblibrairie.a` dans le répertoire contenant les librairies pré-compilées (généralement `/usr/lib/`). Par exemple, pour lier le programme avec la librairie mathématique, on spécifie `-lm`. Le fichier objet correspondant est `libm.a`. Lorsque les librairies pré-compilées ne se trouvent pas dans le répertoire usuel, on spécifie leur chemin d'accès par l'option `-L`.

Les options les plus importantes du compilateur `gcc` sont les suivantes :
- `-c` : supprime l'édition de liens ; produit un fichier objet.
- `-E` : n'active que le préprocesseur (le résultat est envoyé sur la sortie standard).
- `-g` : produit des informations symboliques nécessaires au débogueur.
- `-Inom-de-répertoire` : spécifie le répertoire dans lequel doivent être recherchés les fichiers en-têtes à inclure (en plus du répertoire courant).
- `-Lnom-de-répertoire` : spécifie le répertoire dans lequel doivent être recherchées les librairies précompilées (en plus du répertoire usuel).

- `-o nom-de-fichier` : spécifie le nom du fichier produit. Par défaut, le exécutable fichier s'appelle `a.out`.
- `-O`, `-O1`, `-O2`, `-O3` : options d'optimisations. Sans ces options, le but du compilateur est de minimiser le coût de la compilation. En rajoutant l'une de ces options, le compilateur tente de réduire la taille du code exécutable et le temps d'exécution. Les options correspondent à différents niveaux d'optimisation : `-O1` (similaire à `-O`) correspond à une faible optimisation, `-O3` à l'optimisation maximale.
- `-S` : n'active que le préprocesseur et le compilateur ; produit un fichier assembleur.
- `-v` : imprime la liste des commandes exécutées par les différentes étapes de la compilation.
- `-W` : imprime des messages d'avertissement (warning) supplémentaires.
- `-Wall` : imprime tous les messages d'avertissement.

1.5.1 Illustration de la chaîne programme - exécutable

Pour compiler le programme on utilise le compilateur `gcc` :

```
gcc -std=c99 -c progSomme.c
```

La compilation génère un fichier `progSomme.o`
- Où est la fonction `printf` ?
- Comment le compilateur vérifie que cette fonction est correctement utilisée ?
- On ne peut pas l'exécuter.

Pour pouvoir le rendre exécutable on utilise l'éditeur de liens :

```
gcc progSomme.o -o progSomme
```

L'édition de liens génére d'un fichier `progSomme` qui est exécutable.
- L'éditeur de lien a trouvé la fonction `printf` dans la bibliothèque standard (`libstdc.a` ou `libc.a`).
- définition d'une bibliothèque,
- Quel est le résultat de l'exécution ?
- Le programme principal : Quelle est la fonction qui est appelée au moment du lancement de l'exécutable ?
  `int main(int argc, char **argv)`

1.6 Le premier module

1.6.1 Décomposition du programme

On découpe le même programme en plusieurs fichiers :
- `somme.h` : ce fichier constitue l'interface du module `somme`. Il contient les déclarations nécessaires à l'utilisation du module `somme` par un client.
- `somme.c` : Ce fichier constitue l'implémentation du module `somme`. Il contient les définitions (codes) nécessaires au fonctionnement du module `somme`.

– `client1Somme.c` : Ce fichier est l'utilisation par un client des fonctionnalités présentées par le module `somme`.

Le code du fichier `somme.h`

```
#ifndef _Somme_h_
#define _Somme_h_
extern int somme(int);
#endif
```

Le code du fichier `somme.c` :

```
#include "Somme.h"
int somme(int i)
{

  int resultat = 0;

  for (int k = 0; k <= i; k++)
    {
      resultat += k;
    }
  return resultat;
}
```

Le code du fichier `client1Somme.c` :

```
#include <stdlib.h>
#include <stdio.h>

#include "Somme.h"

int main(int argc, char **arg)
{

  int i = 10;
  printf("La somme des %d entiers est %d \n", i, somme(i));
}
```

On dit que le **module** `client1Somme` est un client du module **fournisseur** `somme`. Un module fournit un ensemble d'informations qui peuvent être utilisées par un module client. Cette notion sera précisée plus tard. Mais on peut remarquer que le fichier `client1Somme` inclut le fichier `somme.h` pour vérifier la bonne utilisation du module `somme`.

Un module en langage `C` est composé de deux fichiers :

1. Le fichier `.h` représente **l'interface** d'un module. Il contient l'ensemble des **déclarations** (fonctions, variables) qui peuvent être utilisés par les clients du module. Il peut également contenir des **définitions de types** ainsi que des **pseudo-constantes** ou des **macros**.

De manière conceptuelle l'interface d'un module présente l'ensemble des services/variables du module qui peuvent être utilisés par un des clients du module. Elle représente la perception par l'extérieur des fonctionnalités d'un module. L'interface d'un module peut évoluer, mais elle doit le faire de manière compatible. C'est-à-dire que la manière dont un client percevait un module à un instant donné ne peut diminuer, elle ne peut que croître. Pourquoi est-il facile de trouver TOUS les clients d'un module ?

2. Le fichier `.c` représente **l'implémentation** d'un module. Il doit fournir une implémentation (du code) à ce qui est présenté par l'interface (services, types, variables). Il s'agit donc d'une solution informatique choisie pour réaliser l'interface. Cette solution informatique peut donc évoluer pour être plus efficace, plus lisible, plus sécuritaire ... L'implémentation doit donc donner du code à tous les services décrits par l'interface et il peut y avoir aussi du code pour des services internes à l'implémentation. On peut remarquer que le fichier `somme.c` inclut l'interface du module `somme` à savoir le fichier `somme.h`.

1.6.2 La compilation séparée

On doit dans un premier temps, compiler séparément le module `somme`, pour cela on exécute la commande

```
gcc -std=c99 -c somme.c
```

Puis on compile ensuite le fichier `client1Somme` :

```
gcc -std=c99 -c client1Somme.c
```

On a donc obtenu deux fichiers `.o` qui sont `somme.o` et `client1Somme.o`. Ces deux fichiers doivent maintenant être assemblés pour créer un exécutable.

```
gcc client1Somme.o somme.o -o client1Somme
```

On peut maintenant exécuter le programme.

1.7 Un changement d'implémentation

On change maintenant l'implémentation du module `somme`.

```
#include "Somme.h"
int somme(int i)
{

  int resultat = 0;
```

```
  while(i >=0)
{
      resultat += i;
   i--;
    }
  return resultat;
}
```

Qu'est-ce que l'on doit refaire pour que le module `client1Somme` puisse fonctionner avec la nouvelle implémentation ?

1. Il faut refaire la compilation du module `somme`.
2. Il faut refaire l'édition de lien.

1.7.1 Un nouveau client du module Somme

```
#include "Somme.h"
int somme(int i)
{

  int resultat = 0;

  while(i >=0)
{
      resultat += i;
   i--;
    }
  return resultat;
}
```

Que doit-on faire pour que le nouveau client puisse utiliser le module `somme`.

1. Il faut compiler le module `client2Somme`.
2. Il faut faire l'édition de lien avec le module `somme`.

1.8 Résumé

- Le fichier `.h` contient les déclarations.
- Le fichier `.c` contient les définitions.
- Le client d'un module contient les appels reflétant l'utilisation du module.
- Il faut que les définitions soient en accord avec les déclarations. On inclut toujours les déclarations dans l'implémentation. C'est à dire que dans un module "Module", la première ligne du fichier "`module.c`" est toujours `#include module.h`.

– Il faut que les utilisations soient en adéquation avec les déclarations. Dans un module “CLIENT” qui utilise un module “FOURNISSEUR” on met toujours l'inclusion de “`FOURNISSEUR.H`” dans `CLIENT.C`. Sur notre exemple :

```
somme.o: somme.h somme.c
    gcc -c -std=c99 somme.c
client1Somme.o: somme.h client1Somme.c
    gcc -c std=c99 client1Somme.c
client1Somme: somme.o clientSomme.o
    gcc std=c99 Client1Somme.o
somme.o -o clientSomme.o
    gcc std=c99 client1Somme.o somme.o -o clientSomme.
```

– Si `somme.h` est modifié toutes les directives de compilation doivent être réxécutées.
– Si `somme.c` change, on refait la compilation de `somme.c` et l'édition de lien, on ne recomppile pas `client1Somme.c`.
– Si `client1Somme.c` change on refait la compilation de `client1Somme.c` et on refait l'édition de lien.

1.9 Une courte introduction à la qualité d'un logiciel

Selon l'IEEE, un logiciel est : Des programmes, procédures, ainsi que possiblement de la documentation et des données liées à l'opération d'un système informatique.

Les Bugs/défauts/fautes sont la conséquence d'erreurs humaines. Ils résulte de la non-conformiteé aux exigences et se manifeste comme une panne lors de l'exécution.

1.9.1 Sources d'erreurs

Neuf sources d'erreurs :

1. Mauvaise définition des exigences
2. Problèmes de communication entre clients et développeurs
3. Déviations délibérées des exigences du logiciel
4. Erreur de conception (logique)
5. Erreurs de programmation
6. Non conformité à la documentation ainsi qu'aux instructions de programmation
7. Insuffisance du processus de tests
8. Erreurs de l'interface usagers ainsi que de de la procédure
9. Erreurs de documentation

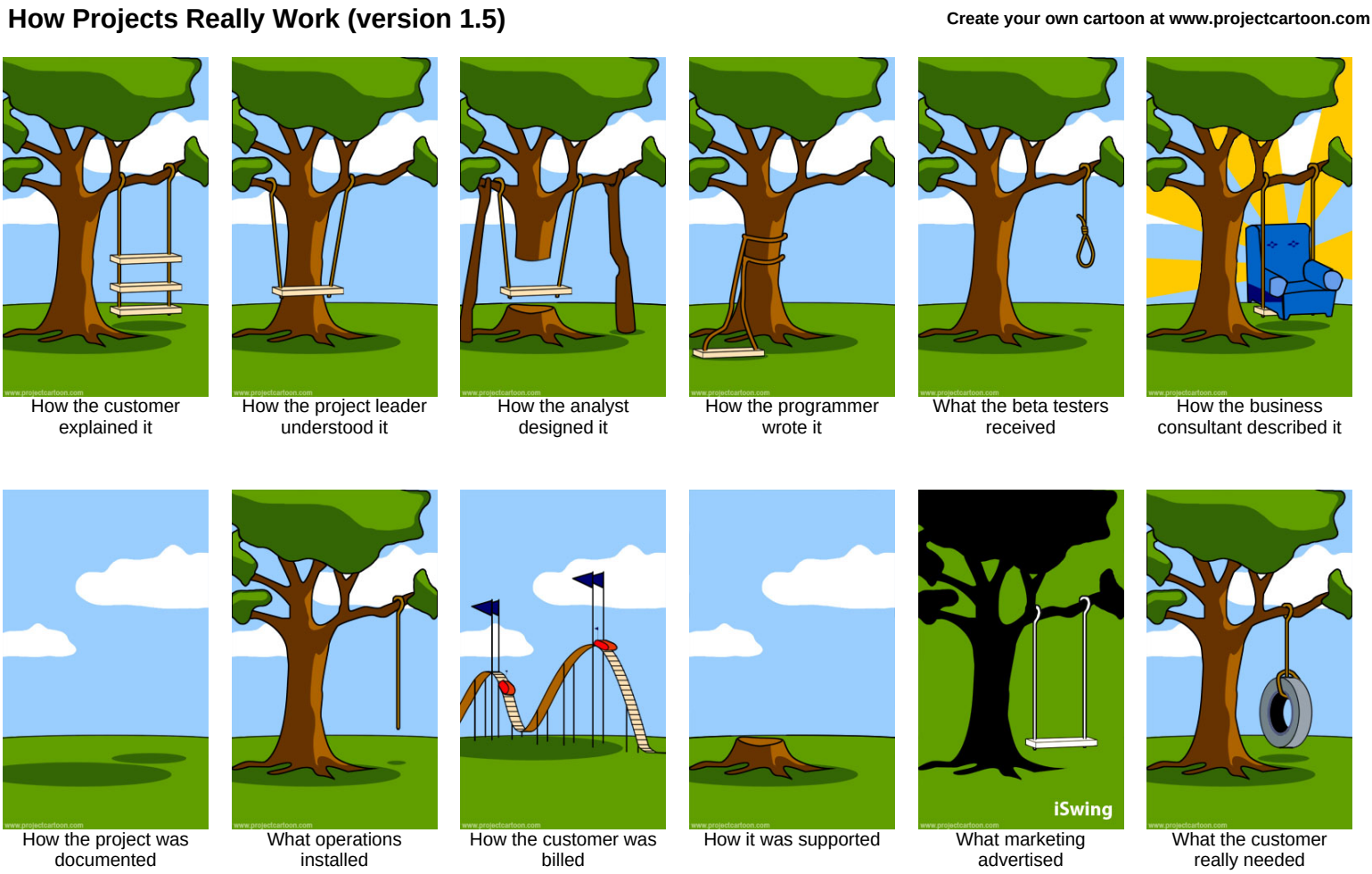

FIGURE 1.1 – Une illustration d'une conception logicielle

1.9.2 La non qualité des systèmes informatiques a des conséquences qui peuvent être très graves

Un bug informatique est une anomalie dans un programme informatique l'empêchant de fonctionner correctement. Sa gravité peut aller de bénigne (défauts d'affichage mineurs) à majeure (explosion du vol 501 de la fusée Ariane 5). Ces erreurs involontaires de conception et de codage représentent un tiers du coût des sinistres informatiques ! La malveillance quant à elle cause 60% de ce coût.

Voici quelques bugs bien identifiées :

- 12/10/2006 La "mauvaise utilisation" d'un logiciel à l'origine d'accidents de radiothérapie à Epinal - Entre mai 2004 et août 2005, des patients traités aux rayons pour des cancers de la prostate ont subi des surdosages dus à des erreurs de paramétrage d'un logiciel. Conséquences actuelles : 5 décès et des complications chez 721 patients...
  Cause : "erreur humaine". Cause réelle : "mauvaise ergonomie d'un logiciel obsolète".
- "C'est la faute de l'informatique". Arrêt de la distribution par écrit de leur évaluation aux élèves lors de la dernière séance de chaque cours dans une grande école.
  Cause évoquée : mise en place d'un nouveau logiciel de gestion.
- Convocation de centenaires à l'école. Convocation à l'école primaire de personnes âgées de 106 ans.
  Cause : codage sur deux caractères.
- Mission Vénus : passage à 5 000 000 de Km de la planète, au lieu de 5 000 Km prévus.
  Cause : remplacement d'une virgule par un point (au format US des nombres).
- Mariner 1 : la première sonde spatiale du programme Mariner, envoyée par la NASA le 27 juillet 1962. La sonde fut détruite peu de temps après son envol. Coût : 80 millions de dollars.
  Cause : un trait d'union oublié dans un programme Fortran (≪ plus coûteux trait d'union de l'histoire ≫, Arthur C. Clarke).
- Passage de la ligne. Au passage de l'équateur un F16 se retrouve sur le dos.
  Cause : changement de signe de la latitude mal pris en compte.
- Y2K : Le bug de l'an 2 000 La lutte contre le bogue de l'an 2000 a coûté à la France 500 milliards de francs. **Cause** : la donnée "année" était codée sur deux caractères, pour gagner un peu de place.
- Socrate. Les plantages fréquents du système de réservation de places Socrate de la SNCF, sa mauvaise ergonomie, le manque de formation préalable du personnel, ont amené un report important et durable de la clientèle vers d'autres moyens de transport.
  Cause : rachat par la SNCF d'un système de réservation de places d'une compagnie aérienne, sans réadaptation totale au cahier des charges du transport ferroviaire.
- Sécurité de la carte bleue. Le secret des cartes bancaires repose essentiellement sur un algorithme, qui a été publié sur un newsgroup !
- Terminaux de paiement. Le 22 décembre 2001 les 750 000 terminaux de payement chez les commerçants ne répondaient plus, ce qui entraîné de longues files d'attente en cette période d'achats de Noël. Cause : saturation des serveurs de la société Atos chargés des autorisation de paiements dépassant 600F. Les autorisation de débit prennent habituellement quelques dizaines de secondes, l'attente a frôlé la demi-heure.
  Conséquence : des clients abandonnent leurs chariots pleins. Le groupe Leclerc a chiffré son préjudice à 2 millions d'euros.

– Echec du premier lancement d'Ariane V. Au premier lancement de la fusée Ariane V, celle ci a explosé en vol.
La cause : logiciel de plate forme inertielle repris tel quel d'Ariane IV sans nouvelle validation. Ariane V ayant des moteurs plus puissants s'incline plus rapidement que Ariane IV, pour récupérer l'accélération dûe à la rotation de la Terre. Les capteurs ont bien détecté cette inclinaison d'Ariane V, mais le logiciel l'a jugée non conforme au plan de tir (d'Ariane IV), et a provoqué l'ordre d'auto destruction. En fait tout se passait bien...
Coût du programme d'étude d'Ariane V : 38 milliards de Francs, pour 39 secondes de vol après 10 années de travail ...
`http://www.inria.fr/actualites/inedit/inedit14\_evea.fr.html .`

On dit qu'il y a des bugs dans tous les logiciels, en petit nombre, et elles ne gênent généralement pas le fonctionnement du système et peuvent demeurer inconnues pendant une longue période. Par contre, certains logiciels sont dits buggés, ils contiennent beaucoup de bugs qui perturbent parfois gravement le fonctionnement du système.

De façon générale, les programmes des élèves (et des profs) ne marchent jamais du premier coup ! Pourtant ce sont des gens réputés intelligents ? Alors où est le problème ? Quelles sont les solutions ?

Etes-vous prêts à garantir la qualité des logiciels que vous écrivez ? Leur validite et leur fiabilité ? Pourriez-vous démontrer la qualité ? Pourquoi hésitez-vous ?

Les bugs surviennnent quand le logiciel ne correspond pas au besoin.

Un bug est un non-respect de la spécification du système, c'est-à-dire de la définition de ses fonctionnalités, de ce que le système est censé faire. Un programme buggé est un programme dont la mise en œuvre ne vérifie pas la spécification.

1.9.3 Évaluation de la qualité logicielle

La norme ISO 9126 définit six groupes d'indicateurs de qualité des logiciels :
– *la capacité fonctionnelle*. c'est-à-dire la capacité qu'ont les fonctionnalités d'un logiciel à répondre aux besoins explicites ou implicites des usagers. En font partie la précision, l'interopérabilité, la conformité aux normes et la sécurité ;
– *la facilité d'utilisation*, qui porte sur l'effort (le peu d') nécessaire pour apprendre à manipuler le logiciel. En font partie la facilité de compréhension, d'apprentissage et d'exploitation et la robustesse - une utilisation incorrecte n'entraîne pas de dysfonctionnement ;
– *la fiabilité*, c'est-à-dire la capacité d'un logiciel de rendre des résultats corrects quels que soient les conditions d'exploitation. En font partie la tolérance de pannes - la capacité d'un logiciel de fonctionner même en étant handicapé par la panne d'un composant (logiciel ou matériel) ;
– *la performance*, c'est-à-dire le rapport entre la quantité de ressources utilisées (moyens matériels, temps, personnel), et la quantité de résultats délivrés. En font partie le temps de réponse, le débit et l'extensibilité - capacité à maintenir la performance même en cas d'utilisation intensive ;
– *la maintenabilité*, qui porte sur l'effort (le peu d') nécessaire en vue de corriger ou de

transformer le logiciel. En font partie l'extensibilité, c'est-à-dire le peu d'effort nécessaire pour y ajouter de nouvelles fonctions ;
- *la portabilité*, c'est-à-dire l'aptitude d'un logiciel de fonctionner dans un environnement matériel ou logiciel différent de son environnement initial. En font partie la facilité d'installation et de configuration pour le nouvel environnement.

Chaque caractéristique contient des sous-caractéristiques. Il y a 27 sous-caractéristiques.

En résumé, un programme devrait toujours être :
- Fiable : On peut avoir confiance dans ces résultats, on peut dire aussi conforme à ces spécifications fonctionnelles,
- Robuste : Il peux fonctionner dans des conditions anormales sans s'arèter.
- Extensible : On peut ajouter de nouvelles fonctionnalités, étendre le périmêtre des données facilement.
- Maintenable : Il peut être corrigé facilement (qualité proche de l'extensibilité)
- Sécurisé : Il ne peut compromettre les ressources sur lesquelles il s'éxécute.

Un module doit être :
- Lisible : Facile a comprendre à la première lecture.
- Autonome : Faiblement couplé, c'est à dire le module dépend le moins possible d'autres modules.
- Maintenable : les modifications d'une partie de l'implémentation doivent impliquer un nombre minimal de modifications de code. (NON DUPLICATION DE CODE, séparation entre interface et implémentation)
- Robuste et fiable (même notion que pour un programme, mais au niveau du module).

1.9.4 Amélioration de la qualité

En génie logiciel, la recherche d'abstraction, de dissimulation, de structuration, d'uniformité, de complétude et de confirmabilité sont des mesures destinés à améliorer la qualité du logiciel, en factorisant le code, c'est-à-dire en n'écrivant qu'une fois des instructions similaires. Cela permet que les modifications soient le plus locales possibles. Cette factorisation concerne les structures de données elles-mêmes (notamment par l'usage des classes et de l'héritage), et les traitements (par l'usage des boucles, fonctions et procédures). Un gain complémentaire de réduction du code source est apporté par le polymorphisme et la liaison dynamique (qui éliminent les « procédures aiguillage ») en programmation objet.

La **modularité**, c'est-à-dire la qualité d'un logiciel d'être découpé en de nombreux modules, permet l'abstraction et la dissimulation. Associée avec **un couplage faible**, elle vise à augmenter la maintenabilité du logiciel en diminuant le nombre de modules touchés par des éventuelles modifications, ainsi que la fiabilité en diminuant l'impact que l'échec d'un module peut avoir sur les autres modules.

En jargon de programmation, un *plat de spaghetti* désigne un logiciel de mauvaise qualité au couplage trop fort et au code source difficile à lire, dans lequel toute modification même mineure demande un intense travail de programmation.

L'abstraction vise à diminuer la complexité globale du logiciel en diminuant le nombre de modules et en assurant l'essentiel. Elle peut également apporter une uniformité du logiciel qui augmente son utilisabilité en facilitant son apprentissage et son utilisation.

La **dissimulation** vise à séparer complètement les détails techniques du logiciel de ses fonctionnalités selon le principe de la boîte noire, en vue d'améliorer sa maintenabilité, sa portabilité et son interopérabilité.

La structuration des instructions et des données rend clairement visible dans le code source les grandes lignes de l'organisation des instructions et des informations manipulées, ce qui améliore sa maintenabilité et facilite la détection des bugs.

De nombreux langages de programmation soutiennent, voire imposent l'écriture de code source selon les principes de structuration, de modularité et de dissimulation. C'est le cas des langages de programmation structurée et de programmation orientée objet.

1.10 Tests Unitaires

Le test unitaire consiste à tester la plus petite unité d'une application. Le test unitaire est un composant essentiel du processus de développement. Il augmente la qualité du code produit et réduit les temps de développement. Ces résultats sont obtenus grâce à deux techniques. La première est liée au fait que le test est réalisé au niveau du module. Nous sommes à ce moment là proches des méthodes. De ce fait, les chances de générer les cas de test pouvant provoquer des erreurs, et assurant une couverture de 100% sont plus grandes. La seconde est que le code est testé dès sa création. Ceci simplifie la recherche et la correction d'éventuelles erreurs. Cette détection précoce des erreurs conduit à une réduction du temps de développement - et donc des coûts -, car le temps passé pour trouver un bug et les ressources utilisées sont moindres (des données statistiques indiquent que 2/3 des bogues problématiques en fin d'intégration auraient pu être détectés par un test unitaire). D'autres études montrent qu'un expert en développement passe la moitié de son temps à débugger, cela peut aller jusqu'à 90% du temps de développement pour un développeur non expérimentés.

Le test unitaire est basé sur trois techniques :

- le test dit boîte blanche pour la structure,
- le test dit boîte noire pour la fonctionnalité,
- le test de non-régression pour l'intégrité.

1.10.1 Le test boîte blanche

Le test boîte blanche vérifie si le code est robuste en contrôlant son comportement avec des cas de test inattendus. L'implémentation du module doit être connue. Le but de ce test est d'exécuter chaque branche du code avec différentes conditions d'entrée afin de détecter tous les comportements anormaux.

Il est très difficile de trouver manuellement les bons cas de test assurant une couverture de code globale. Malgré le bénéfice de ce test sur la qualité, le test boîte blanche est un des tests les plus difficiles à réaliser sans outils automatiques appropriés.

1.10.2 Le test boîte noire

Le test boîte noire vérifie la fonctionnalité de l'interface publique d'une unité. Dans ce type de test les détails sur l'implémentation ne sont pas nécessaires.

En général, le test boîte noire nécessite les étapes suivantes :
- créer un plan de test basé sur les spécifications de l'unité,
- créer des jeux de test permettant de tester les spécifications,
- appliquer les cas de test,
- vérifier que les sorties sont conformes.

Principalement, les cas de test doivent être basés sur les spécifications. Dans le cas ou les spécifications sont intégrées dans le code (par exemple, du code utilisant le Design by Contract), il est possible d'automatiser le test boîte noire, l'outil automatique ayant un a priori sur le fonctionnement de l'unité.

1.10.3 Le test de non-régression

Ce test consiste à vérifier si la nouvelle version de l'unité a été corrigée et si la modification n'a pas généré d'effets de bords. Le principe consiste à tester la nouvelle version de l'unité avec le jeu de test précédent.

1.10.4 Outils automatiques

Toutefois, mettre ces techniques en œuvre peut s'avérer difficile voire même impossible dans un projet. Heureusement, il existe des outils ayant la capacité d'automatiser une grande partie du test unitaire. Ces outils créent l'environnement de test et les bouchons nécessaires au test des modules. De plus, ils génèrent et appliquent automatiquement des cas de tests structurels, simplifient le test fonctionnel et automatisent le test de non-régression. Pour `C`,

1.10.5 Conclusion

Peu importe le type de processus de développement que vous utilisez. Le fait de tester le plus tôt possible va permettre de prévenir, de trouver et de corriger les bugs de manière efficace et économique. L'utilisation d'outils automatiques intégrés dans la chaîne de développement vous permettra de réduire les efforts nécessaires à la mise en œuvre des tests, les coûts et les temps de développement avec, en plus, un code de meilleure qualité.

Les tests unitaires conduisent à une méthode de développement TTD : Test Driven Development (développement dirigé par les tests). L'objectif du TDD est de produire du "code propre qui fonctionne". Pour cela, deux principes sont mis en oeuvre :
- un développeur écrit du code nouveau seulement lorsqu'un test automatisé a échoué,
- toute duplication de code (ou plus généralement d'information, ou de connaissances) doit être éliminée. L'acronyme anglais DRY (Do not Repeat Yourself) peut être utilisé comme moyen mnémotechnique pour cette phase très importante.

Ces deux principes doivent être strictement respectés, même s'ils paraissent difficiles ou bizarres dans un premier temps.

1.10.6 Concrètement

On veut tester le module `somme` que l'on vient d'écrire, comme le module ne rend qu'un seul service, il suffit de tester la fonction `somme`.

Les **test unitaires** portent sur le test d'un module, on peut :
- Tester les fonctions 1 à 1 ;
- Tester des suites d'appel de fonction.

Le test du module `somme` : le module `testSomme`

Il faut écrire l'interface `testSomme.h` et l'implémentation `testSomme.c` :

```
#ifndef _Test_Somme_h_
#include <stdbool.h>
#include "Somme.h"
#define _Test_Somme_h_
extern bool testSomme(int);
#endif
```

On peut déjà remarquer la dépendance entre le module `testSomme` et le module `stdbool`. On écrit maintenant l'implémentation du module `testSomme` à savoir le fichier `testSomme.c` :

```
#include "testSomme.h"
#include "somme.h"
#include <stdlib.h>
#include <stdio.h>


static int jeuDeTest [] = {1, 4, 9, 13, 35};

bool testSomme(int valATester)
{
int resultatTheorique = valATester*(valATester+1)/2;
return resultatTheorique == somme(valATester);
}

int main(int argc, char **argv)
{
printf("debut du Test");

int tailleJeuDeTest = sizeof(jeuDeTest)/sizeof(int)- 1;

for(int i=0; i <= tailleJeuDeTest; i++)
{
```

```
  printf("Test de la valeur %d \n", jeuDeTest[i]);
  if(!testSomme(jeuDeTest[i]))
    {
      printf("Code Faux pour la valeur %d \n", jeuDeTest[i]);
      return EXIT_FAILURE;
    }
}

printf("Test Reussi \n");
return EXIT_SUCCESS;
}
```

Attention, il faut regarder plus précisement le jeu de tests. Qu'est ce qui se passe avec la valeur -2 ? ? Qu'est ce qui se passe avec la valuer 1000000000000 ?