

# Projet Maths-Info CIR3-CNB3

## I. Objectif

Le document ci-présent a pour but de vous aider à rédiger des rapports/cahiers apportant une valeur ajoutée à votre travail de codage. Les points que ces rapports/cahiers vous aideront à améliorer sont :

- formalisation des caractéristiques du produit ;
- aide au test de validité du produit ;
- aide à la visualisation du fonctionnement du produit.

## II. Déroulement du projet

Il vous est demandé de progresser en confiance vers une réalisation qui satisfasse le client. Pour ce faire, nous vous donnons les conseils suivants, basés sur la création régulière de documents sur lesquels vous pourrez vous appuyer pour créer un produit allant dans le sens de ce qui est demandé, et s'assurer que c'est bien le cas.

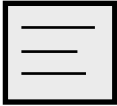

### 1. Conception générale

Le but de cette phase d'aboutir à une compréhension totale du sujet. Le cahier des charges est souvent vague, c'est pourquoi il vous est fortement conseillé d'élaborer dans un rapport les points suivants :

- écrivez un rapide scénario qui décrira comment fonctionne le produit fini ; n'hésitez pas à séparer les différents cas d'utilisation, même s'ils sont très simples (au contraire) ;
- listez de manière formelle et aussi courte que possible (sujet-verbe-complément) chaque *exigence* du programme, qu'elle concerne une caractéristique à très haut niveau d'abstraction ou un fin détail de fonctionnement ;  
par exemple, on ne dira pas :  
« On fabrique une voiture verte », mais :
  - « Le système sera une voiture automobile » puis :
  - « Le système sera de couleur verte »,avec éventuellement des précisions sur la nature de la couleur verte, sur ce que signifie « fabriquer », etc. ; la liste des exigences doit contenir au moins toutes les caractéristiques cruciales du programme (ce document peut même être contractuel !) (voir annexe 1) ;
- pour chacune des exigences ci-dessus, indiquez dans un second document (le « cahier de tests ») comment tester chaque exigence ; chaque exigence devra en effet faire l'objet d'au moins un test (voire plus), car vous devez pouvoir montrer que vous respectez la liste des exigences ;
- dessinez les modules fonctionnels du logiciel de manière à garantir l'absence de « trou de conception » ; cette représentation est très graphique, et a pour but de montrer les différents blocs du programme, ainsi que les données que les blocs s'échangent (voir annexe 2) ;

- définissez les *structures de données* (en langage intermédiaire, et pas forcément dans le langage de programmation choisi), des *actions sur ces données* (quels calculs/transformations vont leur être appliqués), de la *visibilité entre ces données* (quelle donnée englobe telle autre, quelle donnée est fabriquée à partir de telle autre, etc.) (voir annexe 2) ;
- décrivez les fonctions principales et les fonctions « compliquées » en langage familier mais formel (à quoi servent ces fonctions, quels sont les pré-requis, les résultats ainsi que les méthodes/moyens qu'elles doivent employer) ;
- faites un arbre des fonctions (au moins sur les principales fonctions) définissant l'arborescence des appels entre fonctions, ces fonctions pouvant ici appartenir à un même module ou non (voir annexe 3).

*Rapport* : cette phase donne lieu à la rédaction :

- d'un « rapport de conception générale » contenant tous les éléments cités ci-dessus (liste des exigences, modules, fonctions et structures) à l'exception du cahier de tests ;  À rendre
- d'un « cahier de tests » qui donne la liste des tests validant les exigences du projet final dans son ensemble, ainsi que l'état de validation du test ou pas.  À rendre

**NB** : un module est un groupe de fonctions qui rendent un service identifié (par exemple : service d'affichage, service de lecture de fichiers...) et qui doit pouvoir être testé unitairement (c'est-à-dire indépendamment des autres modules du projet).

## 2. Tests

Vous devez pouvoir tester *chaque module, voire chaque fonction de module, de manière indépendante*. Il n'est absolument pas question de tout écrire puis d'essayer de déboguer l'ensemble du moteur de calcul : vous devez pouvoir effectuer régulièrement chaque test pour valider les évolutions correctives.

Cela deviendra impératif lors des évolutions liées à l'intégration qui ne doivent pas se solder par l'introduction de nouvelles erreurs (il y aurait alors ce que l'on appelle une *régression*).

## 3. Recette finale du logiciel

**Objet** : présentation par le groupe du programme fonctionnel pour en vérifier la conformité avec les exigences et le cahier de test défini en phase d'analyse. Tous les membres des groupes participent à la présentation du résultat.

## III. Quelques conseils et remarques

- Réalisez avec le plus grand sérieux possible les documents de conception générale afin de faciliter votre travail d'implémentation : « Ce que l'on conçoit bien s'énonce clairement, et les mots pour le dire arrivent aisément. » *Nicolas Boileau* ! ;
- si quelque chose est possible à faire avec ce qui a été vu en cours/TD/TP (graphiques, fichiers, etc.) vous êtes *obligés de l'implémenter avec ce qui vous a été proposé => il est interdit de remplacer* ;
- si quelque chose est impossible à faire avec ce qui a été vu (ajout de boîtes de dialogue, par exemple), *il est alors autorisé d'ajouter*.

## IV. Annexes

### 1. Exemples « d'exigences »

Une exigence décrit une et une seule caractéristique d'un produit, qui devra être implémentée ET testée. Elle est écrite en langage naturel, si possible avec un sujet, un verbe et un complément uniquement (pas de subordonnée, etc. : restez simple).

Par exemple, pour un programme censé lire un fichier texte contenant des nombres, trier ces nombres et écrire le résultat sur disque, tout en l'affichant à l'écran, nous aurons les exigences suivantes (liste non-exhaustive) :

1. Le programme demandera le nom du fichier texte à ouvrir.
2. Le système lira le fichier texte.
3. Le fichier texte sera constitué de nombres au format décimal entier.
4. Les nombres du fichier seront séparés par des séparateurs classiques (espace, tabulation, retour à la ligne)
5. En cas d'erreur de lecture un message sera affiché.
6. Les nombres seront triés par ordre croissant.
7. Le résultat du tri sera affiché à l'écran.
8. Le résultat du tri sera sauvegardé dans un fichier.
9. Le fichier de sauvegarde portera le même nom que le fichier d'entrée préfixé de "tri\_".
10. En cas d'erreur d'écriture un message sera affiché.
11. ...

À vous de voir à quel niveau de détail vous arrêter, mais au plus ce niveau sera fin, au plus le produit final sera maîtrisé.

Les exigences doivent être numérotées, et préférablement regroupées en différentes classes (exigences d'implémentation, exigences de design, exigences d'interface, exigences de sécurité, etc.)

On peut aussi différencier les exigences impératives des exigences facultatives.

Chaque exigence doit donner naissance à une ou plusieurs procédures de test, pour vérifier et valider que l'exigence est bien remplie.

### 2. Exemples de « modules »

Découper votre problème en sous-problèmes, qui eux-mêmes pourront être découpés en sous-sous-problème, etc. est une manière efficace :

- de gérer la complexité d'une réalisation ;
- d'améliorer la qualité de son code, car les petits bouts de code sont plus faciles à tester et à corriger que les longues fonctions de plusieurs dizaines de lignes ;
- d'augmenter ses capacités et sa productivité, en se focalisation sur un seul petit problème à la fois ;
- de se répartir les tâches afin de maximiser la performance d'une groupe et de gagner en temps de développement.

C'est pourquoi il vous est conseillé et demandé :

- d'identifier les données que vous aller manipuler ;
- d'identifier les différentes parties de votre programme ;
- d'identifier quelles parties de votre programme manipulent quelles données.

Ainsi, pour notre programme qui lit des données dans un fichier et les trie, nous aurions besoin pratiquement d'une seule donnée, définie comme suit :

```
Structure TableauDeValeurs
    taille du tableau (entier)
    données du tableau (entiers)
```

En langage C, nous aurions écrit cette structure :

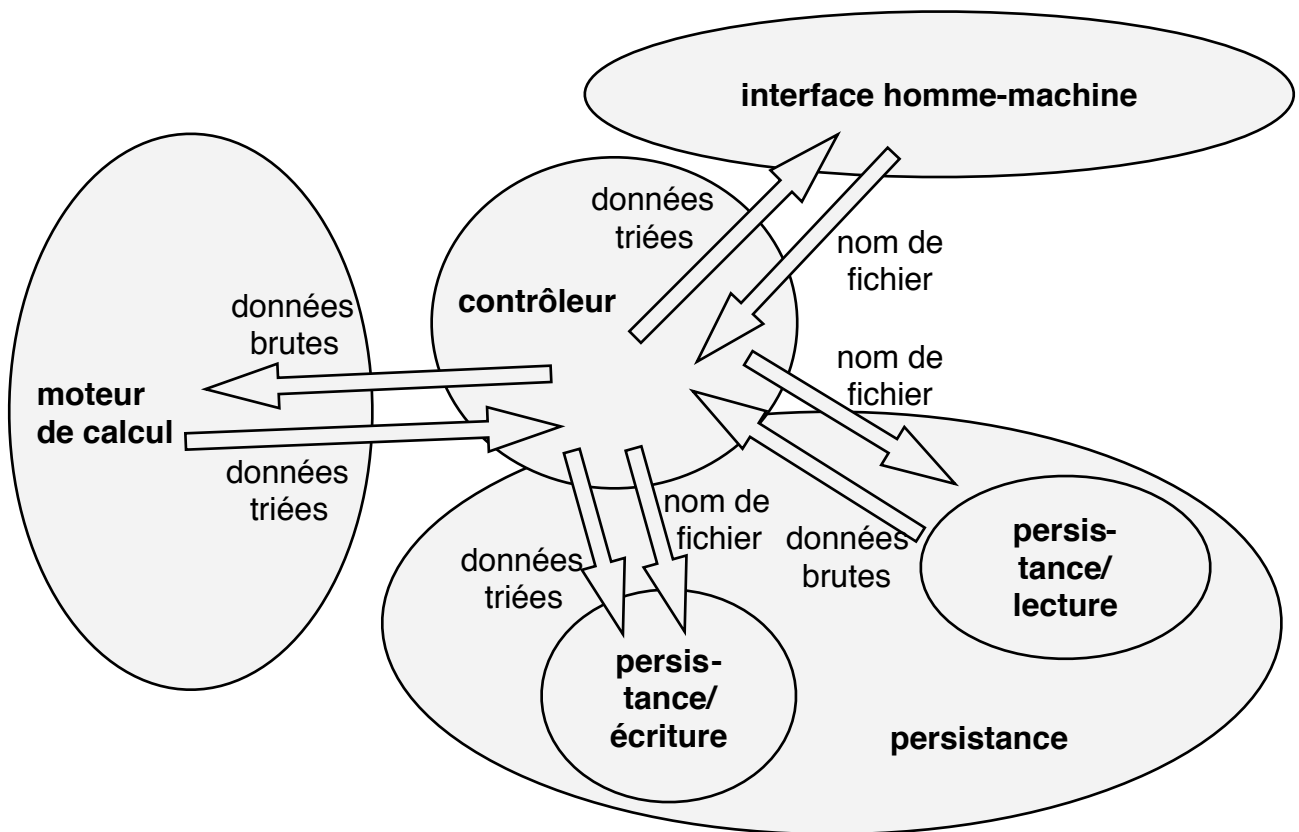
```
typedef struct
{
    int taille;
    int *donnees;
} TableauDeValeurs;
```

Au vu des exigences listées au chapitre précédent, nous pouvons identifier quatre modules différents :

- le module « moteur de calcul », qui va faire le travail de tri ;
- le module « persistance », qui va lire et écrire les données sur disque ;
- le module « interface homme-machine » qui va permettre de dialoguer avec l'utilisateur ;
- le module « contrôleur », qui joue le rôle de chef d'orchestre.

Ces modules peuvent être découpés en sous-modules, eux mêmes découpables, etc. Nous pourrions par exemple avoir un module lecture de fichier et un module écriture de fichier, un module lecture des informations données par l'utilisateur et un module affichage des informations, etc.

Il est important de faire figurer les informations que s'échangent les modules. Toujours pour donner un exemple (qui pourrait être amélioré), voici ce à quoi pourrait ressembler un diagramme simple de modules pour le problème qui est le nôtre :



### 3. Exemple « d'arbre de fonctions »

Chaque module sera constitué d'une ou de plusieurs fonctions. Il vous est demandé d'écrire le prototype de chacune de ces fonctions, c'est-à-dire de décrire leurs « interfaces ».

Ceci est très important, car cela va permettre de donner des informations suffisantes à vos collègues codeurs et/ou à votre sous-traitant.

Un arbre de fonction est une représentation arborescente dans laquelle chaque nœud est une fonction racine, et chaque sous-arbre est une fonction appelée par cette fonction racine.

Ainsi, la fonction *trieDonnees* du moteur de calcul pourrait avoir comme arbre de fonction (voir vos cours/TD pour le fonctionnement de *triSelectionIteratif*) :

