TELECOM SudParis

Projet Informatique 1ère Année
CSC 3502

APPLICATION EXEMPLE

Enseignants CSC3502 - Département INF

Enseignant responsable : *coordonnateurs du module*

2010

Table des matières

Chapitre 1

Introduction

Ce document, produit avec LaTeX [1], décrit de façon succincte le développement d'une application minimale de gestion de bibliothèque réalisé à titre d'exemple pour le module CSC3502. Il donne un aperçu d'un contenu possible de rapport final et n'a pas vocation à servir de modèle, le plan et le contenu devant être adaptés pour chaque projet.

Ce document comporte trois parties à la suite de la présente introduction : une présentation du cahier des charges, la description du développement proprement dit détaillant les différentes phases, et un manuel utilisateur ; il se termine par une conclusion proposant quelques pistes d'amélioration. En fin de document, une annexe présente quelques fichiers extraits du code source de l'application.

Différentes modalités d'utilisation de l'application sont disponibles selon l'interface proposée à l'utilisateur : mode "terminal", mode "graphique" ou encore mode "web" (offrant uniquement la consultation). Ce rapport décrit essentiellement le développement de la version "terminal" de l'application ; les deux autres versions sont utilisées dans les parties pratiques du module CSC3502 pour illustrer les principes du développement d'applications graphiques et d'applications Internet.

Le développement de cette application a été réalisé au département Informatique de TELECOM SudParis sur de nombreuses années ; y ont notamment contribué Chantal Taconet, Paul Gibson, Djamel Belaïd, Dominique Bouillet, Daniel Millot, Gilles Protoy, Christian Schüller, Daniel Ranc et Michel Simatic.

Chapitre 2

Cahier des charges

Il s'agit d'écrire une application minimale de gestion des ouvrages d'une bibliothèque. Les fonctionnalités qui devront être offertes par l'application sont :

- ajouter un ouvrage
- supprimer un ouvrage
- rechercher un ouvrage
- afficher l'ensemble des ouvrages

Ces fonctionnalités sont à implanter dans un premier temps avec une interface mode "terminal", puis avec une interface graphique.

Les contraintes techniques sont les suivantes :

- le langage de développement choisi est le C.
- les fichiers sources doivent être livrés avec un makefile.
- l'interface de l'application doit être conforme aux maquettes fournies ci-dessous :

Une extension est envisagée dans un deuxième temps : une consultation en mode web à travers l'utilisation d'un navigateur (une application complète en ligne avec ajout et suppression demanderait la gestion de l'accès concurrent aux données de l'application côté serveur). Cette extension devra être utilisable avec un client navigateur indépendant de l'OS. On utilisera un serveur Apache sous Linux.

La maquette de l'IHM (interface homme-machine) de l'extension envisagée est fournie ci-dessous.

D'autres extensions seraient envisageables, par exemple pour améliorer le confort d'utilisation de l'application :

- demander une confirmation à l'utilisateur lors des suppressions,
- ajouter une liste de mots-clés pour aider à la recherche,
- effectuer un tri sur l'auteur ou le titre pour l'affichage
- donner la possibilité d'annuler la dernière opération d'ajout ou de suppression (*undo*)...

Ces extensions n'ont pas été implantées pour l'instant.

Chapitre 3

Développement

L'organisation de cette partie est la suivante : une première section est dédiée à l'analyse du problème et à la spécification fonctionnelle ; les deux sections suivantes traitent de la conception, d'abord préliminaire puis détaillée ; la section suivante aborde le codage, et les trois dernières sections détaillent successivement les tests unitaires, d'intégration et de validation effectués.

3.1 Analyse du problème et spécification fonctionnelle

Une relecture détaillée du cahier des charges initial avec le "client" de l'application a permis de :

- lever toutes les ambiguïtés,
- donner des réponses à de nombreuses questions.

Nous donnons ci-dessous les principales questions et les réponses qui y ont été apportées.

- Qu'est-ce qu'un ouvrage ?

  Un ouvrage est en général composé des informations suivantes :

  - le titre,
  - l'auteur (principal),
  - l'année d'édition,
  - l'éditeur...

  On ne considère pas ici le cas d'ouvrages avec plusieurs auteurs, et on se limitera aux trois premières informations (titre, auteur, année d'édition).
- Est ce que l'utilisateur a accès à toutes les fonctionnalités ? Sinon, quels sont les types d'utilisateurs et leurs droits respectifs, et comment l'utilisateur s'authentifie-t-il ?
  On considère ici que tous les utilisateurs ont le même privilège (c'est-à-dire qu'il n'y a pas de notion d'administrateur) et qu'ils n'ont pas à s'authentifier.
- Doit-on gérer les accès simultanés ?
  On considère qu'à un moment donné, il n'y a qu'un seul utilisateur de l'application.
- Connaît-on le nombre maximum d'ouvrages ?
  La collection d'ouvrages est de taille quelconque.
- Faut-il créer un seul fichier pour stocker les ouvrages ou plusieurs ?
  On considère qu'il faut créer un seul fichier, c'est-à-dire s'assurer a minima de la persistance des données.
- Le fichier doit-il être localisé sur le compte de l'utilisateur ou sur un compte spécifique ?
  Sur le compte de l'utilisateur.
- La recherche a-t-elle lieu dans le fichier ou en mémoire ?
  Afin de limiter les accès fichier, la recherche sera effectuée sur la représentation interne de la collection d'ouvrages en mémoire et non pas dans le fichier.

Pour chacune des fonctionnalités offertes par l'application, il convient de préciser les conditions d'utilisation si nécessaire :

- Afficher l'ensemble des ouvrages
  → doit afficher tous les ouvrages.
- Ajouter un ouvrage
  → doit permettre à l'utilisateur de saisir les informations d'un ouvrage.
  On vérifiera au préalable que l'ouvrage n'existe pas déjà pour ne pas créer de doublon, et on refusera la création d'un ouvrage lorsque les informations sont incomplètes (cette spécification n'est pas prise en compte dans l'implantation proposée).
- Supprimer un ouvrage
  → doit permettre à l'utilisateur de saisir les informations utiles pour la suppression (selon un ou plusieurs critères d'un ou plusieurs ouvrages)
  On affichera tous les ouvrages trouvés. Pour simplifier, on considère qu'on supprime tous les ouvrages trouvés selon tous les critères proposés, et pour simplifier l'implantation, un seul critère sera utilisé dans la suite, le nom de l'auteur.
- Rechercher un ouvrage
  → doit permettre à l'utilisateur de saisir les informations utiles pour la recherche (selon un ou plusieurs critères d'un ou plusieurs ouvrages)
  On affichera tous les ouvrages trouvés. Pour simplifier l'implantation, la recherche ne portera dans la suite que sur le critère nom de l'auteur (et, comme prévu lors de l'analyse, elle sera effectuée sur la représentation interne de la collection d'ouvrages en mémoire).

La spécification permet de prévoir les tests de validation. Par exemple, pour la fonctionnalité "Ajouter un ouvrage", les tests suivants seraient pertinents :
- Donnée : un ouvrage déjà existant à ajouter
  Résultat : message d'erreur : "L'ouvrage existe déjà", la collection ne change pas.
- Donnée : un ouvrage incomplet à ajouter
  Résultat : message d'erreur : " Veuillez renseigner toutes les informations demandées", la collection ne change pas.
- Donnée : un ouvrage complet (pas déjà existant) à ajouter
  Résultat : la collection est mise à jour.

Pour des raisons de simplicité, la contrainte consistant à éviter les doublons n'est pas retenue par l'implantation réalisée, et le premier de ces tests n'est donc pas été utilisé.

D'autres tests de validation sont également nécessaires pour tester la robustesse de l'application (données très longues pour le titre ou l'auteur, année non valide, fichier de données corrompu, etc), ses performances et sa portabilité.

3.2 Conception préliminaire

On a choisi de décomposer l'application en cinq modules :
- *main* : correspond à la fonction principale, avec le point d'entrée de l'application (initialisation, affichage d'un menu),
- *menu* : dédié à l'interaction utilisateur (affichage/saisie d'informations, boucle d'interaction avec l'utilisateur),
- *ouvrage* : dédié à la gestion des ouvrages,
- *bibliotheque* : dédié à la gestion des listes d'ouvrages,
- *util* : fonctionnalités de plus bas niveau liées au stockage de l'information (sur disque et en mémoire).

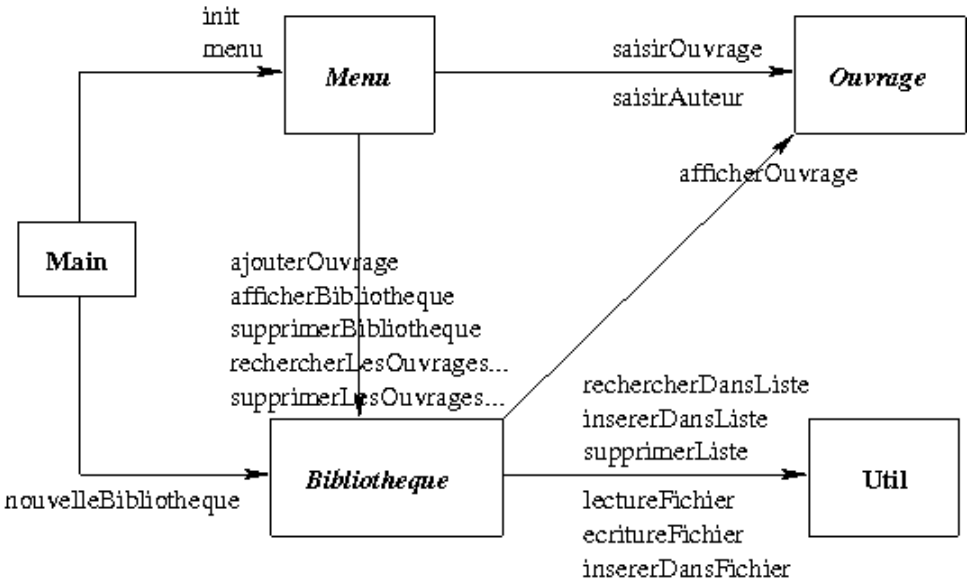
Le module *menu* sera clairement impacté par la déclinaison selon les différentes modalités d'utilisation (terminal, graphique) ; c'est aussi le cas de *ouvrage* et *bibliotheque* pour ce qui concerne la saisie et l'affichage.

D'autres décompositions en modules sont possibles. Celle que nous avons choisie tente de concilier d'une part la contrainte des différentes modalités de fonctionnement (terminal, graphique) prévues par le cahier des charges et d'autre part une factorisation maximale des fonctionnalités communes aux différentes versions.

Ce premier découpage va s'avérer insuffisant lors de l'étape de conception détaillée, mais il permet néanmoins de recenser les fonctionnalités indispensables et de préciser les interactions entre modules. Dans un deuxième temps, nous distinguerons des opérations spécifiques à chaque modalité d'utilisation dans l'implantation des modules *menu*, *ouvrage* et *bibliotheque*.

Concernant par exemple le module *menu*, il est pertinent d'envisager une fonction *menu()* affichant les options possibles et saisissant le choix de l'utilisateur ; une approche descendante de la conception de cette fonction amène ensuite à définir des fonctions *choixAjouter()*, *choixRechercher()*, *choixSupprimer()* et *choixAfficher()* correspondant aux différentes options offertes. Ces fonctions, utilisées en interne dans le module, n'apparaissent pas au niveau de son interface décrite dans le fichier d'entête menu.h. Cette interface expose en revanche les prototypes des fonctions fournies par le module (services disponibles) et que les fonctions des autres modules peuvent appeler.

Les dépendances entre ces modules liées à ces appels de fonction sont représentées sur le graphe ci-dessous où sont listées les fonctions publiées par chaque module.



Le découpage en modules est l'occasion de spécifier les tests d'intégration. Ces tests permettent de vérifier la bonne coordination entre les fonctions lors des différentes interactions. A titre d'exemple, l'un des tests vérifiera que la coordination entre les fonctions *afficherBibliotheque* et *afficherListeOuvrages* est correcte : appel de la fonction *afficherBibliotheque* avec une bibliothèque vide pour vérifier que l'affichage correspondant est bien réalisé, puis ajout d'un ouvrage et appel avec une bibliothèque non vide pour vérifier que *afficherListeOuvrages* est bien appelée exactement une fois.

Il faut également choisir les structures de données pour représenter les objets manipuler par l'application. La représentation d'un ouvrage est la suivante :

```
structure ouvrage
titre : Chaîne de caractères
nomAuteur : Chaîne de caractères
anneeEdition : Entier
fstruct
```

Il faut aussi pouvoir gérer un ensemble d'ouvrages : pour cela, on pourra considérer la structure de tableau ou de liste chaînée. Indépendamment de la représentation interne choisie, nous parlerons dans la suite de liste d'ouvrages, sous la forme du type abstrait ListeOuvrages. On peut ainsi introduire également une structure bibliothèque :

```
structure bibliotheque
nomFichier : Chaîne de caractères
listeOuvrages : ListeOuvrages
fstruct
```

La structure bibliothèque regroupe des informations sur le stockage fichier de la collection d'ouvrages (le nom du fichier, information de type Chaîne de caractères), et sur sa représentation en mémoire (information de type ListeOuvrages).

Concernant le stockage des informations de l'ensemble des ouvrages, il est réalisé dans un fichier texte *data.txt*, modifiable dans un éditeur de texte, ce qui facilitera les tests. La structure du fichier est volontairement simplifiée :

- une information par ligne,
- une ligne vide entre deux ouvrages successifs.

par exemple :

```
Introduction au génie logiciel
J-L. Raffy
2005

Mon Système Linux
C. Schüller
2002
```

On aurait pu faire des choix différents, avec une structure de fichier plus sophistiquée ; par exemple :

- une première ligne avec un commentaire précisant les significations des informations.
- les lignes suivantes : 1 ouvrage par ligne, avec les champs séparés par un ' ;' (ou encore format CSV, i.e. Comma Separated Values).

3.3 Conception détaillée

Cette étape de la conception fournit le contenu détaillé des différentes structures de données ainsi que le corps de toutes les fonctions et procédures de l'application.

On procèdera d'une part module par module (par ordre hiérarchique d'importance des modules) et d'autre part selon une méthode descendante (top-down), en commençant par la fonction principale :

```
fonction principale() : CodeRetour
    bibliotheque : Bibliotheque
    bibliotheque = nouvelleBibliotheque(nomFichier)
    init() // Initialisations diverses (affichage...)
    menu(bibliotheque)
    retourner OK
ffct
```

Les algorithmes éventuels seront décrits à l'aide de pseudo-langage. Dans le cas d'algorithmes classiques et connus on pourra éventuellement s'affranchir de ce détail (exemple : tri à bulles, arbre fils-frère etc.).

L'approche descendante amène de proche en proche à décrire toutes les fonctions et procédures nécessaires. par exemple :

```
procédure menu (donnée-résultat bibliotheque : Bibliotheque)
    afficherMenu()
    tq (choix ← menu()) <> quitter faire
         cas où choix vaut
              ajout:
                   choixAjouter(bibliotheque)
              supprimer:
                   choixSupprimer(bibliotheque)
              rechercher:
                   choixRechercher(bibliotheque)
              afficher:
                   choixAfficher(bibliotheque)
              autre:
```

```
                    afficher("Erreur dans le choix du menu")
            fcas
            afficherMenu()
    ftq
fproc
```

À titre d'exemple, nous considérons ci-dessous la fonctionnalité d'ajout d'un ouvrage, qui doit permettre à l'utilisateur de saisir les informations d'un ouvrage. Ces informations saisies doivent être persistantes (sauvegardées dans le fichier) :

```
procédure choixAjouter (donnée-résultat bibliotheque : Bibliotheque)
    Ouvrage ouvrage
    ouvrage ← saisirOuvrage()
    ajouterOuvrage( bibliotheque, ouvrage )
fproc
```

Cette procédure *choixAjouter()* utilise une fonction et une procédure que nous explicitons sommairement :

```
fonction saisirOuvrage (): Ouvrage
    // Il faut saisir successivement les informations
    // caractéristiques de l'ouvrage
    // en utilisant des modalités spécifiques pour chaque version
ffct
```

Ainsi, au cours de la descente dans l'écriture, nous finissons par rencontrer la nécessité de tenir compte des modalités d'interaction avec l'utilisateur. Dans la suite, nous nous intéressons exclusivement à la version "terminal" de l'application, et l'implantation de chacun des trois modules *menu*, *ouvrage* et *bibliotheque* comportera une partie générique et une partie propre à la version "terminal".

```
procédure ajouterOuvrage (donnée ouvrage : Ouvrage,
                          donnée-résultat bibliotheque : Bibliotheque)
    // Garder l'information dans la liste
    insererDansListe(ouvrage, bibliotheque -> listeOuvrages)
    // Garder l'information dans le fichier
    insererDansFichier(ouvrage, bibliotheque->nomFichier)
fproc
```

Nous devons bien entendu expliciter les deux procédures utilisées par *ajouterOuvrage()* :

```
procédure insererDansListe (donnée ouvrage : Ouvrage,
                            donnée-résultat liste : ListeOuvrages)
    // L'insertion d'un élément dans une
    // liste est un problème connu
fproc

procédure insererDansFichier (donnée ouvrage : Ouvrage,
                              donnée-résultat fichier : Fichier)
    // Ecrire un item dans un fichier est connu
    // Il faudra cependant étudier le format
fproc
```

Cette étape est aussi l'occasion de spécifier les tests unitaires qui permettront de vérifier que chaque fonction produit bien les résultats attendus.

3.4 Codage

Les programmes sources seront commentés de manière à permettre la génération automatique de la documentation à l'aide de l'outil doxygen [2]. Pour cela, les commentaires doivent respecter un format,

comme le montre l'exemple donné en annexe (cf A.1). En positionnant la macro GENERATE_LATEX à la valeur YES dans le fichier de configuration doxy.conf, on peut obtenir dans un sous-répertoire latex/ un ensemble de fichiers source latex et un Makefile permettant d'obtenir une documentation dont le contenu est équivalent à celui de la documentation mise en ligne. Le fichier pdf obtenu est lui aussi navigable, comme la documentation html que génère doxygen.

Il y a ainsi continuité entre les spécifications fonctionnelles et techniques et la documentation du code, permettant un contrôle facile de la cohérence entre ces différentes vues du système.

3.5 Tests unitaires

Chaque module du code se voit affecté un module de test unitaire reposant sur l'outil de test CUnit. A l'intérieur d'un module, *chaque fonction fait l'objet d'une suite de tests CUnit*.

Il est préférable que la personne effectuant les tests adopte une démarche de type "boîte noire", évitant d'examiner de trop près le code lui-même. Le testeur se focalisera de préférence sur la documentation afin de réaliser les tests pertinents en fonction des besoins fonctionnels. La documentation essentielle est donnée par les commentaires présents au niveau des fichiers "headers" et du code.

Un test est ajouté à la suite de tests pour chaque besoin fonctionnel. Par exemple, au niveau d'une fonction agissant sur une collection d'entités on aurait probablement un test pour respectivement une collection vide, une collection pleine et une collection mi-pleine.

Ces tests unitaires devraient également tester un ensemble raisonnable et représentatif de valeurs des paramètres d'entrée, avec une attention particulière portée sur les bornes (de tailles de tableaux etc.) et sur les cas exceptionnels. Une bonne pratique consiste à tester les valeurs de retour s'il y a lieu et les effets de bord sur le système.

Enfin, si le code est disponible, il pourraît s'avérer utile de réaliser des tests en "boîte blanche" afin de fournir une bonne couverture des différents chemins d'exécution dans chaque fonction.

Noter que si l'on n'arrive pas à écrire les tests unitaires tout simplement parce qu'on ne sait pas quoi tester, alors cela signifie que les spécifications et commentaires doivent être améliorés. De la même façon, si l'on ne sait pas comment intégrer les test dans le système, cela veut dire que les spécifications de conception doivent être améliorées.

Lorsqu'un test unitaire se réalise avec succès, il est ajouté à la librairie de tests pour réutilisation sur de futures versions du code (ce sont les fameux "tests de non-régression"). Lorsqu'au contraire le test échoue, un certain nombre de causes potentielles non nécessairement exclusives doivent être étudiées :

- un bogue dans le code ;
- un bogue dans le code de test ;
- une mauvaise compréhension des besoins fonctionnels.

Dans tous les cas, la correction nécessite d'aligner le code et les tests avec les besoins fonctionnels. Dans les cas difficiles il faudra observer l'exécution au débogueur.

Au fur et à mesure de l'évolution des besoins et du code, il se peut qu'une librairie de test devienne obsolète si elle contient un ou plusieurs tests ne correspondant plus au code ou aux besoins actuels. Il est d'autant plus important que les besoins fonctionnels soient bien compris et bien structurés, et ceci dès le début du développement.

Une modification de la conception du système, de la répartition/organisation des modules etc. est bien entendu encore plus grave en terme de répercussion au niveau des tests.

Remarquons que les tests unitaires sont du code, et à ce titre, nécessitent d'être bien documentés à l'aide de commentaires reliant le code aux besoins fonctionnels.

Un bon exemple de test unitaire a été vu en cours lorsqu'a été abordé le test de la fonction `ouvrageToString`. D'autre part le test unitaire de toutes les fonctions du module `CollectionOuvrages` est traité dans le CI sur CUnit.

Un exemple de programme de test unitaire est donné en annexe (cf A.2).

3.6 Tests d'intégration

Après validation des différentes fonctions par les tests unitaires, les tests d'intégration vont contrôler le bon fonctionnement des interactions entre les différentes fonctions dans le cadre de l'intégration des différents modules.

Les tests d'intégration sont réalisés *après* que les tests unitaires soient totalement validés ; en effet si ce n'était pas le cas, un bogue pourrait être dû soit à un défaut à l'intérieur des fonctions, soit à un défaut d'intégration. Si nous savons que les fonctions ont été validées par des tests unitaires, nous pouvons au contraire nous focaliser sur un problème d'intégration, cause la plus probable dans ce cas.

Les tests d'intégration représentent la phase la plus difficile phase des tests. Il est conseillé d'adopter une approche strictement hiérarchique du bas vers le haut ("bottom-up"). La clé est de trouver les interdépendances importantes entre modules : celles qui partagent l'état du système, et où des fonctions préjugent d'un séquencement particulier des appels.

Comme pour les tests unitaires, les tests d'intégration une fois validés sont conservés dans une librairie de tests pour un usage futur. Les commentaires des tests d'intégration (leur documentation) devra être reliée directement à la documentation de conception.

Pour des systèmes simples, on pourra éventuellement choisir de ne pas procéder à des tests d'intégration et de passer directement à des tests de validation. En effet si ceux-ci se passent avec succès, ce peut être une preuve – pour un système simple – que l'intégration a été correcte.

Un exemple simple de test d'intégration a été abordé dans le cours lorsqu'a été testée la fonction `afficherBibliotheque`, qui intègre les fonctions `afficherListeOuvrages` et `afficher`.

Un exemple de programme de test d'intégration est fourni en annexe (cf A.3).

3.7 Tests de validation

Les tests de validation commencent *après* validation des tests d'intégration, autrement on risque la situation suivante :

- succès de la validation du système, alors que des bogues du code n'ont pas été trouvés ;
- échec de la validation du système, avec localisation difficile du ou des bogues en raison d'une intégration complexe ;
- échec de la validation du système, mais la correction du bogue nécessite des modifications de la conception.

D'un autre côté, les test de validation sont une excellente opportunité de découvrir des malentendus au niveau des besoins de l'utilisateur ou du client. Plus on exécute ces tests tardivement, plus le risque de tels malentendus est grave. Pour éviter cet inconvénient, on pourra concevoir/prévoir/coder les tests de validation très tôt dans le cycle de développement, même s'ils ne seront réellement exécutés qu'à la fin.

Comme pour les autres tests, les tests de validation feront partie de la librairie de test en vue de future réutilisation. Les commentaires des tests de validation (leur documentation) devra être reliée directement aux besoins du client et aux cas d'utilisation (scenarii).

Un exemple simple de test de validation a été abordé en cours lors du test de l'interface utilisateur du système en version terminal, avec focalisation sur les données entrées lors de l'ajout d'éléments au système.

Le test des interfaces utilisateur est un des aspects les plus importants des tests de validation.

Le CI a montré comment les tests de validation peuvent égaleement être implémentés avec CUnit. Cette possibilité est pertinente si l'on est capable d'automatiser ou simuler l'interaction utilisateur à l'aide d'un unique appel de fonction.

Chapitre 4

Manuel utilisateur

Les sources sont livrés sous forme d'une archive comportant, outre les fichiers sources :
- un makefile
- un fichier de données pour utiliser l'application
- des fichiers sources de test
- un shell-script enchainant une série de tests

L'archive est disponible sur le site moodle du module CSC3502, dans la section “Application exemple mode "terminal"”.

4.1 Production de l'exécutable

Le makefile livré dans l'archive comporte un ensemble de cibles dont l'usage est le suivant :
- cible par défaut : pour produire l'exécutable
- cible `tests` : pour produire les exécutables de test
- cible `runtest` : pour exécuter les tests
- cible `valgrind` : pour exécuter l'application en testant les fuites memoire
- cible `doc` : pour produire la documentation
- cible `cleanall` : pour nettoyer le répertoire

Le flag `-Wall` est activé par défaut.

4.2 Réalisation des tests

L'archive comporte un shell-script, `runtest.sh`, permettant d'enchaîner l'exécution des programmes de test livrés dans l'archive. Chaque fichier source de test comporte un commentaire final reproduisant la sortie d'une exécution et mettant en évidence les résultats respectivement attendus et obtenus. Le commentaire se termine par des suggestions de modification.

4.3 Utilisation

L'exécutable est produit sous le nom `gestionBibliotheque-term`. Lorsqu'on démarre son exécution, un menu est affiché à l'écran et les fonctionnalités sont accessibles par la saisie du numéro associé.

Les saisies d'information, que ce soit celles d'un ouvrage pour l'ajout d'ouvrage ou le nom de l'auteur pour les fonctionnalités de recherche et de suppression, sont sécurisées de manière à éviter tout comportement indésirable de l'application en cas de saisie erronée.

Pour son fonctionnement, l'application utilise un fichier de données contenant la collection d'ouvrages. C'est un fichier texte éditable par ailleurs, qui doit impérativement être nommé `data.txt`. Un fichier exemple est livré dans l'archive. L'application peut cependant être lancée sans aucun fichier de données.

Lorsqu'un fichier de données `data.txt` est disponible au lancement, son format est contrôlé par l'application. Si ce format est correct, le lancement de l'application réussit et une copie de sauvegarde `data.bak` du fichier `data.txt` est effectuée pour les lancements ultérieurs, ou pour la fonctionnalité de reprise de "la liste initiale de la session".

Si le format du fichier `data.txt` est incorrect, et si un fichier de sauvegarde `data.bak` utilisable est disponible, celui-ci est utilisé en remplacement du fichier défectueux. S'l n'est pas possible d'utiliser un fichier de sauvegarde `data.bak`, le lancement de l'application échoue.

Les éventuels problèmes de démarrage ainsi que toutes les opérations à effet de bord sur la collection d'ouvrages (ajout/suppression d'ouvrage) sont enregistrées dans un fichier de log appelé `bibliotheque.log`.

Le mécanisme de copie de sauvegarde étant systématique lors du démarrage, tout relancement de l'application occasionne la perte de la dernière sauvegarde. De même, le contenu du fichier de log est réinitialisé lors du relancement de l'application.

Chapitre 5

Conclusion

Ce rapport décrit les principales étapes du développement de la version “terminal” d'une application minimale de gestion de bibliothèque.

Deux versions graphiques de cette application ont également été développées, l'une dont l'interface graphique a été conçue à l'aide de l'outil *Glade* et l'autre dont l'interface graphique est entièrement programmée en *gtk*. L'intérêt de *Glade* est de pouvoir explorer confortablement la bibliothèque de *widgets* pour construire une maquette de l'interface. Cela peut être suffisant lorsque l'interface à réaliser est simple, mais certainement pas pour une interface complexe ou comportant de nombreux éléments similaires (éléments d'un jeu, par exemple). Lorsqu'une maquette satisfaisante a pu être obtenue, il est alors facile de construire la même arborescence de *widgets* avec des appels à la bibliothèque *Gtk*, afin de bénéficier de toute la puissance de la programmation, nécessaire pour produire des interfaces sophistiquées.

En complément, une application Internet de consultation a été dérivée à partir de la même base de code ; elle ne propose pas les opérations à effet de bord sur la collection d'ouvrages (ajout/suppression), qui posent des problèmes délicats de synchronisation en cas d'utilisation simultanée par des internautes.

Des choix simplificateurs ont été faits : pas de contrôle de présence d'un ouvrage dans la collection pour éviter les doublons lors d'une nouvelle entrée, format du fichier de données... On pourrait bien entendu revenir sur ces choix pour proposer des solutions plus sophistiquées.

De nombreuses améliorations seraient possibles et même souhaitables, comme par exemple l'introduction de la fonctionnalité de *undo* pour annuler la dernière opération à effet de bord sur la collection, voire carrément un ensemble d'opérations avec une profondeur paramétrable dans l'historique. Le log systématique des opérations à effet de bord déjà disponible dans la version actuelle prépare bien le terrain pour faciliter l'implantation d'une telle fonctionnalité.

Concernant la recherche, une perspective d'évolution pourrait être l'introduction d'une possibilité de recherche multi-critère.

Bibliographie

[1] Latex. *Une courte introduction à LATEX 2e*. http ://ctan.mines-albi.fr/info/lshort/french/lshort-fr.pdf.

[2] Doxygen. *Générateur de documentation html*. http ://www.doxygen.org.

Annexe A

Code source

A.1 Code source commenté “à la doxygen”

```
/**
 * @file bibliotheque.h
 *
 * Interface du module Bibliotheque
 *
 * @version 1.1
 * @author Module projet informatique 1ere annee TELECOM SudParis (CSC3502)
 * @date 28 nov. 2008
 */

#ifndef BIBLIOTHEQUE_H_
#define BIBLIOTHEQUE_H_

#include "ouvrage.h"
#include <stdlib.h>  //malloc
#include <string.h>  //strcpy


/**
 * @def TAILLE_MAX_CHAINE
 * Longueur maximum du nom du fichier associe a la bibliotheque
 */
#define TAILLE_MAX_CHAINE 80

/**
 * @def NB_MAX_OUVRAGES
 * Nombre maximum d'ouvrages dans la liste d'une bibliotheque
 */
#define  NB_MAX_OUVRAGES  100

/**
 * @a ListeOuvrages : tableau de NB_MAX_OUVRAGES pointeurs sur @a Ouvrage
 */
typedef Ouvrage* ListeOuvrages[NB_MAX_OUVRAGES];

/**
 * Bibliotheque_struct : une bibliotheque est une liste d'ouvrages
 * cette liste d'ouvrage est sauvegardee dans un fichier
 */
```

```
typedef struct {
char nomFichier[TAILLE_MAX_CHAINE]; /*!< Le nom du fichier de stockage */
char nomBackUp[TAILLE_MAX_CHAINE]; /*!< Le nom du fichier de backup */
ListeOuvrages listeOuvrages; /*!< L'acces a la representation memoire */
} Bibliotheque_struct;
/**
 * Bibliotheque est un pointeur sur un Bibliotheque_struct (transparent du
 * point de vue d'un utilisateur de Bibliotheque
 */
typedef  Bibliotheque_struct *Bibliotheque;


/*
 * Partie a implantation generique
 */

/**
 * creation d'une variable de type Bibliotheque (allocation dynamique),
 * lecture du fichier dont le nom est fourni en parametre et
 * chargement dans la liste d'ouvrages associee a la bibliotheque
 * @param nomFichier (donnee) : nom du Fichier de stockage
 * @param nomBackUp (donnee) : nom du Fichier de backup
 * @return le pointeur sur la bibliotheque creee ou NULL en cas d'echec
 */
Bibliotheque nouvelleBibliotheque(char *nomFichier, char *nomBackUp);

/**
 * reset de la Bibliotheque a partir du fichier de backup
 * @param bibliotheque (donnee-resultat) : bibliotheque a reseter
 */
void resetBibliotheque(Bibliotheque bibliotheque);

/**
 * suppression de toute la Bibliotheque (liberation memoire)
 * @param bibliotheque (donnee) : bibliotheque a supprimer
 */
void supprimerBibliotheque(Bibliotheque bibliotheque);

/**
 * ajouter un ouvrage dans la bibliotheque
 * @param bibliotheque (donnee-resultat) : bibliotheque a laquelle ajouter l'ouvrage
 * @param ouvrage (donnee) : ouvrage a ajouter
 * @return OK en cas d'ajout reussi, KO si la liste est pleine et PB_ACCES
 * en cas de probleme d'acces fichier
*/
int ajouterOuvrage(Bibliotheque bibliotheque, Ouvrage ouvrage);

/**
 * supprimer des ouvrages d'un auteur dans la bibliotheque
 * @param bibliotheque (donnee-resultat) : la bibliotheque
 * @param auteur (donnee) : auteur dont il faut supprimer tous les ouvrages
 * @return PB_ACCES en cas d'echec, sinon une valeur positive ou nulle
*/
int supprimerLesOuvragesDeAuteur(Bibliotheque bibliotheque, Auteur auteur);

/**
```

```
 * rechercher des ouvrages d'un auteur dans la bibliotheque
 * @param bibliotheque (donnee) : la bibliotheque
 * @param auteur (donnee) : auteur dont on recherche la liste d'ouvrages
 * @param listeTrouve (resultat) : liste d'ouvrages correspondant a cet auteur
*/
void rechercherLesOuvragesDeAuteur(Bibliotheque bibliotheque, Auteur auteur,
                                   ListeOuvrages listeTrouve);

/**
 * afficher tous les ouvrages d'une bibliotheque
 * @param bibliotheque (donnee) : la bibliotheque dont on veut afficher les ouvrages
 */
void afficherBibliotheque(Bibliotheque bibliotheque);


/*
 * Partie a implantation specifique pour chaque version (terminal/graphique)
 */

/**
 * Affiche l'ensemble des ouvrages de la liste @a liste
 * @param liste (donnee) : la liste a afficher
 */
void afficherListeOuvrages( ListeOuvrages liste);

/**
   denombre les ouvrages d'une bibliotheque
   @param bibliotheque (donnee) : la bibliotheque à denombrer
   @return le nombre d'ouvrages de la bibliotheque
*/
int nombreOuvrages(Bibliotheque bibliotheque);

#endif /* BIBLIOTHEQUE_H_ */
```

A.2 Code source d'un test unitaire

```
/**
 *
 * Tests unitaires :
 * <ul>
 * <li> ouvrage - ouvrageToString</li>
 * </ul>
 *
 * @file unitTest.c
 *
 * Tests unitaires
 * (version terminal)
 *
 * @version 1
 * @author J Paul Gibson
 * @date  January 2010
*/

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

```
#include "ouvrage.h"

void unitTestouvrageToString ( );

/**
 *  Unit Test for ouvrageToString
 * <ul>
 * <li>Construct an ouvrage </li>
 * <li> Call function </li>
 * <li> Verify that result of function call is as required. </li>
 * </ul>
 * Repeat for 5 cases:
 * <ul>
 * <li> All fields - title, author and year - "standard" </li>
 * <li> Title maximum length</li>
 * <li> Author maximum length</li>
 * <li> Year maximum length</li>
 * <li> All fields at maximum length</li>
 * </ul>
 */
void unitTestouvrageToString ( ){

printf("=== unitTestouvrageToString\n");

Ouvrage ouvrage1= {"title", "author", 2010};
printf("%s\n", ouvrageToString(ouvrage1));

// This code should provide warning as initializer string for title is too long
// and should be concatenated after v
Ouvrage ouvrage2= {"title too long -  abcdefghijklmnopqrstuvwxyz", "author", 2010};
printf("%s\n", ouvrageToString(ouvrage2));

// This code should provide warning as initializer string for author is too long
// and should be concatenated after b
Ouvrage ouvrage3= {"title", "author too long - abcdefghijklmnopqrstuvwxyz", 2010};
printf("%s\n", ouvrageToString(ouvrage3));

// This code should provide warning as initializer int for year is too big
// and should be truncated to 1410065407
Ouvrage ouvrage4= {"title", "author", 9999999999};
printf("%s\n", ouvrageToString(ouvrage4));

// Should there be spaces between fields when at maximum length?
Ouvrage ouvrage5= {"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN",
                   "abcdefghiljklmnopqrs",1234567890};
printf("%s\n", ouvrageToString(ouvrage5));
}


int main( int argc, char *argv[]) {


printf("=== unitTests for bibliotheque\n");

unitTestouvrageToString();
```

```
  return EXIT_SUCCESS;
}

/* EXPECTED OUTPUT

=== unitTests for bibliotheque
=== unitTestouvrageToString
                                    title              author      2010

title too long -  abcdefghijklmnopqrstuv              author      2010

                                    titleauthor too long - ab      2010

                                    title              author1410065407

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNabcdefghiljklmnopqrs1234567890

*/

/*  TEST RESULT PASS  - 27 January 2010   12:37, J Paul Gibson

SUGGESTED CHANGE TO REQUIREMENTS

Guarantee spaces between fields to improve presentation

*/
```

A.3 Code source d'un test d'intégration

```
/**
 * Tests d'integration :
 * <ul>
 * <li> bibliotheque co-ordinates afficher and afficherListeOuvrages in
*  function afficherBibliotheque</li>
 * </ul>
 *
 * @file integrationTest.c
 *
 * (version terminal)
 *
 * @version 1
 * @author J Paul Gibson
 * @date  January 2010
*/

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

#include "bibliotheque.h"
#include "menu.h"
#include "ouvrage.h"
#include "util.h"

void integrationTestafficherBibliotheque ( );
```

```
/**
 *  Integration Test for afficherBibliotheque
 * <ul>
 * <li> Create an empty bibliotheque </li>
 * <li> Call afficherBiblitheque with the empty bibliotheque as parameter</li>
 * <li> Verify that afficher has been called (once) </li>
 * <li> Add an element to the bibliotheque </li>
 * <li> Call afficherBiblitheque with the non-empty bibliotheque as parameter</li>
 * <li> Verify that afficherListeOuvrages has been called (once) </li>
 * </ul>
  */
void integrationTestafficherBibliotheque ( ){
printf("=== integrationTest afficherBibliotheque\n");
char n1[] = "n1.txt";
char n2[] = "n2.txt";
Bibliotheque b1 = nouvelleBibliotheque (n1, n2);
afficherBibliotheque(b1);
// Pour tester ajouterOuvrage si pas encore teste :
// Ouvrage ouvrage1= {"title", "author", 2010};
// int result = ajouterOuvrage(b1, ouvrage1);
afficherBibliotheque(b1);
}

int main( int argc, char *argv[]) {

printf("=== integrationTests for bibliotheque\n");

integrationTestafficherBibliotheque ();

  return EXIT_SUCCESS;
}

/* EXPECTED OUTPUT

=== integrationTests for bibliotheque
=== integrationTest afficherBibliotheque
La Liste est vide
Liste des ouvrages
                                  Titre              Auteur     Annee

                                  title              author      2010
*/

/*  TEST RESULT PASS  - 27 January 2010   13:54, J Paul Gibson

SUGGESTED CHANGE TO REQUIREMENTS

Change word "liste" to "Bibliotheque" ??

*/
```

A.4 Code source d'un test de validation

```
/**
```

```
 * @file validationTest6c.c
 *
 * Test6c - l'annee n'est pas valide  (e.g 3000)
 * <ul>
 * <li>   construct ouvrage with annee =3000 </li>
 * <li> add to bibliotheque </li>
 * <li> check for error message </li>
 * <li> display state of bibliotheque on screen </li>
 * </ul>
 *
 * @version 1
 * @author J Paul Gibson
 * @date  Januray 2010
*/

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

#include "bibliotheque.h"
#include "menu.h"
#include "ouvrage.h"
#include "util.h"


int main( int argc, char *argv[]) {
  // les donnees de la bibliotheque sont sauvegardees dans un fichier
  // ce qui permet de ne pas perdre les donnees entre deux executions
  char nomFichier[] = "data.txt";
  char nomBackUp[] = "data.bak";

  Bibliotheque bibliotheque;

  // creation d'une bibliotheque initialisee a partir du fichier
  bibliotheque = nouvelleBibliotheque(nomFichier, nomBackUp);

  if (bibliotheque == NULL) {
    printf(" Impossible de creer la bibliotheque\n");
    return EXIT_FAILURE;
  }

  // initialisation de l'interface utilisateur
  if (init() == 0) {
      printf("Probleme d'initialisation de l'interface utilisateur\n");
      return EXIT_FAILURE;
  }

  // Pour tester ajouterOuvrage si pas encore teste :
  // Ouvrage ouvrageTest = {"title", "author", 3000};
  // int result = ajouterOuvrage(bibliotheque, ouvrageTest);
  afficherBibliotheque(bibliotheque);

  return EXIT_SUCCESS;
}
```