BIBLIO

un logiciel pour la gestion de prêts de livres

G. Tremblay, P. Zentilli et B. Malenfant

Version de novembre 2007

Table des matières

Chapitre 1

Introduction : les origines du système de gestion de prêts de livres

Une première version d'un logiciel de gestion de prêts de livres a été développée, à l'été 2003, par un groupe d'étudiants du certificat en informatique, dans le cadre du cours INM5000 (Atelier), et ce sous la supervision des professeurs Tremblay et Makarenkov. Plus précisément, le logiciel qui fut développé visait à gérer les prêts de livres provenant de la bibliothèque personnelle du professeur Tremblay. Ses principales fonctionnalités étaient les suivantes — on trouvera à l'annexe A les spécifications d'origine, telles qu'elles furent transmises aux étudiants :

1. Indiquer l'emprunt d'un livre.
2. Indiquer le retour d'un livre.
3. Identifier tous les livres empruntés par une personne.
4. Identifier la personne qui a emprunté un livre... puis lui envoyer un courriel lui demandant de rapporter ce livre.
5. Demander le rappel, par courriel, de tous les livres prêtés.

Bien que la première version du logiciel était «minimalement» fonctionnelle, elle comportait plusieurs problèmes et faiblesses — décrits plus en détail à la section 2.2.

Un projet, subventionné par le Fonds de Développement Pédagogique de l'UQAM, a donc été mis sur pied dans le but de restructurer, réécrire et, de façon générale, améliorer le logiciel existant, et ce en utilisant les techniques professionnelles de construction de logiciels — utilisation d'assertions, gestion de configuration, outils de tests, etc. L'objectif de ce projet était principalement de construire un «corpus de maintenance», c'est-à-dire un programme réel sur lequel les étudiants pourraient effectuer des travaux de maintenance.

Dans les sections qui suivent, nous décrivons plus en détail la première version du logiciel, la version actuelle produite après les travaux de restructuration, ainsi que des améliorations futures qui pourraient être apportées.

Chapitre 2

État de la première version du logiciel

2.1 Exemples d'utilisation

Dans cette première section, nous allons illustrer les principales fonctionnalités mises en oeuvre par la première version du logiciel de gestion de prêts de livres.

Pour comprendre les exemples qui suivent, il faut savoir que la base de données gérée par le logiciel est une base de données *textuelle*, où chaque ligne représente les informations sur un livre emprunté (les différents champs étant séparés par un caractère «|»), et où le fichier contenant la base de données se nomme «`livredb.txt`»[1]. Dans ces exemples, où les commandes d'utilisation du logiciel sont soulignées et où les réponses et résultats ne le sont pas, nous allons donc illustrer le fonctionnement du logiciel en montrant l'effet de certaines commandes sur le contenu de ce fichier (illustré, entre autres, avec «`cat livredb.txt`»). Soulignons que lorsqu'une commande reçoit plusieurs arguments, ceux-ci sont séparés par «^».

Les exemples d'utilisation sont présentés dans les figures 1 et 2. La figure 1 illustre les fonctionnalités pour `emprunter` un livre et pour ensuite le `rapporter`. La figure illustre aussi le fonctionnement de l'opération `lister`, qui permet de présenter, de façon succincte et simplifiée, le contenu de la base de données — les emprunts sont présentés en ordre croissant des noms des emprunteurs et les adresses de courriel sont omises.

La figure 2, quant à elle, illustre le fonctionnement des opérations permettant d'identifier tous les `emprunts` effectués par une personne, d'identifier l'`emprunteur` d'un livre spécifié par son titre (lequel doit être le titre *exact et complet*, donc tel que spécifié lors de l'emprunt), de même qu'une opération qui permet de `trouver` le titre exact et complet d'un livre qui contient un certain mot (si plusieurs livres contiennent ce mot, tous les titres de ces livres seront retournés comme résultat). En outre, la figure 2 illustre aussi l'utilisation des deux opérations qui permettent d'envoyer des courriels pour demander à des emprunteurs de rapporter les livres empruntés : (*i*) `rappeler-livre`, qui envoie un courriel à une seule personne, à savoir celle désignée par le dernier appel à `emprunteur` ; (*ii*) `rappeler-tous-les-livres` qui, pour chacun des livres empruntés, envoie un courriel à l'emprunteur pour lui demander de retourner le livre.

[1] Dans la nouvelle version du logiciel, le nom a été changé et est défini à l'aide d'une constante symbolique dans un fichier de `configuration`.

```
% cat livredb.txt
% emprunter ^ Tremblay ^ tremblay.guy@uqam.ca ^ Code complet ^ McConnell

Emprunter un livre:
------------------

Livre ajoute a la liste des livres pretes :

    Titre      : "Code complet"
    Auteur     : McConnell
    Emprunteur : Tremblay
    Courriel   : tremblay.guy@uqam.ca

% cat livredb.txt
Code complet | McConnell | Tremblay | tremblay.guy@uqam.ca
% emprunter ^ Malenfant ^ malenfant.bruno@uqam.ca ^ La programmation en pratique ^ Kernighan et Pike

Emprunter un livre:
------------------

Livre ajoute a la liste des livres pretes :

    Titre      : "La programmation en pratique"
    Auteur     : Kernighan et Pike
    Emprunteur : Malenfant
    Courriel   : malenfant.bruno@uqam.ca


% cat livredb.txt
Code complet | McConnell | Tremblay | tremblay.guy@uqam.ca
La programmation en pratique | Kernighan et Pike | Malenfant | malenfant.bruno@uqam.ca
% lister
 Malenfant  :: [ Kernigha ] "La programmation en pratique "
 Tremblay   :: [ McConnel ] "Code complet "
% rapporter La programmation en pratique

Retour de livres :
------------------

- La programmation en pratique :  livre rapporte

% cat livredb.txt
Code complet | McConnell | Tremblay | tremblay.guy@uqam.ca

% emprunter ^ Tremblay ^ tremblay.guy@uqam.ca ^ The pragmatic programmer ^ Hunt et Thomas

Emprunter un livre:
------------------

Livre ajoute a la liste des livres pretes :

    Titre      : "The pragmatic programmer"
    Auteur     : Hunt et Thomas
    Emprunteur : Tremblay
    Courriel   : tremblay.guy@uqam.ca


% cat livredb.txt
Code complet | McConnell | Tremblay | tremblay.guy@uqam.ca
The pragmatic programmer | Hunt et Thomas | Tremblay | tremblay.guy@uqam.ca
% lister
 Tremblay   :: [ Hunt et  ] "The pragmatic programmer "
 Tremblay   :: [ McConnel ] "Code complet "
```

Figure 1: Exemples d'utilisation du logiciel (première version) : $1^{ère}$ partie

```
% cat livredb.txt
Code complet | McConnell | Tremblay | tremblay.guy@uqam.ca
The pragmatic programmer | Hunt et Thomas | Tremblay | tremblay.guy@uqam.ca
% emprunts Tremblay

Resultat de la recherche :

Titre      : Code complet
Emprunteur : Tremblay
Courriel   : tremblay.guy@uqam.ca

Titre      : The pragmatic programmer
Emprunteur : Tremblay
Courriel   : tremblay.guy@uqam.ca

% rappeler-livre

Le fichier de resultats ne contient pas les donnees requises apres une
execution normale de la commande emprunteur.

Avant d'envoyer un courriel a une personne lui demandant de rapporter un livre
vous devez tout d'abord executer la commande "emprunteur" et
le resultat de la commande doit confirmer que ce livre a ete prete.

% trouver pragmatic
The pragmatic programmer
% emprunteur The pragmatic programmer

Resultat de la recherche :

Titre      : "The pragmatic programmer"
Emprunteur : Tremblay
Courriel   : tremblay.guy@uqam.ca

% rappeler-livre

Un courriel a ete envoye a : Tremblay
A l'adresse suivante : tremblay.guy@uqam.ca
A propos du livre : "The pragmatic programmer"

% cat livredb.txt
Code complet | McConnell | Tremblay | tremblay.guy@uqam.ca
The pragmatic programmer | Hunt et Thomas | Tremblay | tremblay.guy@uqam.ca
% rappeler-tous-les-livres

Des courriels on ete envoyes a tous ceux qui ont emprunte un livre
et qui ont une adresse de courriel.
```

Figure 2: Exemples d'utilisation du logiciel (première version) : $2^{ième}$ partie

2.2 Aperçu des principaux problèmes

Après un examen rapide du code de la première version du logiciel de gestion de prêts de livres, les faiblesses suivantes ont été constatées :

1. Grande redondance du code, i.e., beaucoup de code qui se répète (code dupliqué). Entre autres, tous les programmes `main` se ressemblent.
2. Commentaires peu clairs, informels, qui ne reflètent pas toujours correctement le code associé, avec un mélange d'anglais et de français.
3. Présence de nombreuses instructions redondantes (par exemple, `seek` au début d'un fichier après un `open`, initialisation inutile de variables), montrant une plus ou moins bonne connaissance du langage `C`.
4. Structures de contrôle très (trop!) imbriquées, i.e., comptant un grand nombre de niveaux d'imbrication, rendant leur compréhension difficile.
5. Manque de cohésion de plusieurs routines, qui effectuent souvent des tâches trop complexes, qui consistent en fait en plusieurs tâches distinctes regroupées, sans cohésion, à l'intérieur d'une même routine. De façon générale, donc, absence de modularité.
6. Nombreux traitements et messages d'erreur éparpillés un peu partout dans le code, diminuant de façon importante sa lisibilité.
7. Présence de plusieurs *nombres magiques*, parfois contradictoires, qui auraient plutôt dû être définies par des constantes symboliques (`#define XXX ...`).
8. Présentation et indentation du code non consistante, mélangeant des styles différents d'indentation et de présentation des accolades.
9. Mauvais choix des noms d'identificateur pour les variables et routines, mélangeant parfois des mots anglais et français.
10. Utilisation d'une syntaxe inhabituelle pour la spécification des arguments sur la ligne de commande, les arguments devant être séparés par le caractère «^», par exemple :

```
emprunter ^ Guy ^ guy@bidon.ca ^ Code complet ^ McConnell
```

Certains de ces aspects sont examinés plus en détail dans les sections qui suivent.

2.3 Redondance du code

Un défaut majeur de la solution initiale était l'extrême redondance du code. Par exemple, en examinant le code, on pouvait constater que chaque commande avait son propre fichier `.c` :

- `emprunter.c`
- `rapporter.c`
- `emprunts.c`
- `emprunteur.c`
- `rappeler-livre.c`
- `rappeler-tous-les-livres.c`

Or, aucun fichier `.h` ou `.c` permettant de regrouper les parties communes de code n'avait été défini.

Par exemple, les fichiers `emprunter.c` et `emprunteur.c` comptaient respectivement 644 et 583 lignes de code. Or, après analyse du code de ces fichiers (avec `diff`), on a pu conclure qu'il suffisait de changer 300 lignes du fichier `emprunter.c` pour obtenir le fichier `emprunteur.c`. En d'autres mots, ces deux fichiers avaient en commun près de 50 % de leurs lignes de code — ce qui va à l'encontre du principe *DRY = Don't Repeat Yourself*!

Autre statistique intéressante : la version de départ du logiciel dans son ensemble comptait près de 5800 lignes de code. Après modularisation du code et, donc, *factorisation* des parties communes, le code résultant n'en compte plus qu'environ 2900, une réduction de 50 % — et ce même en comptant *et* les commentaires, *et* les programmes de tests unitaires qui ont été ajoutés!

2.4 Commandes utiles non définies

Certaines commandes utiles n'avaient pas été mises en oeuvre. Lors de livraison initiale du système, des *shell* scripts avaient donc dû être définis par le professeur Tremblay pour réaliser les fonctionnalités désirées :

- `trouver` : retourne le ou les titres, exacts et complets, des livres empruntés qui contiennent un certain mot spécifié en argument.
- `lister` : retourne (sur `stdout`) le contenu de la base de données des emprunts, mais sous une forme compacte et simplifiée : les emprunts sont indiqués en ordre croissant des noms des emprunteurs, les adresses de courriel sont omises, et seuls les premiers caractères des noms des auteurs sont indiqués.

Divers autres fichiers avaient été définis pour permettre de réaliser une interface *Web*, mais n'ont jamais été utilisés car ils n'étaient pas fonctionnels.

2.5 Absence de jeux d'essai

Un défaut majeur du logiciel dans sa première forme était l'absence complète de tests, c'est-à-dire l'absence totale de jeux d'essais permettant de vérifier le bon fonctionnement du logiciel. La vérification initiale du bon fonctionnement du logiciel a donc dû être faite de façon informelle, «à la main». Certains problèmes ont alors été corrigés (de façon pas toujours très élégante, à cause de l'absence de modularité du programme) pour rendre le logiciel véritablement fonctionnel.

Après cette validation initiale, on avait dès lors sous la main une version fonctionnelle d'un logiciel de gestion de livres. Toutefois, à cause de l'absence de tests, si des modifications ou améliorations devaient être apportées au logiciel, il n'y avait aucune façon de vérifier que ces changements n'introduisaient pas d'autres problèmes. En d'autres mots, il n'y avait pas de *tests de non régression*, un outil fondamental pour permettre l'évolution d'un logiciel.

Chapitre 3

État de BIBLIO, la version actuelle du système de gestion de prêts de livres

Dans ce qui suit, nous décrivons les divers travaux qui ont été effectués pour améliorer le logiciel de gestion de prêts de livres, décrivant par la même occasion son état actuel.

3.1 Développement de tests de régression de niveau système

La première tâche qui fut effectuée à consister à définir des tests de régression, de niveau système, pouvant être exécutés de façon automatique. Des tests de niveau système servent à vérifier le bon fonctionnement du logiciel dans son ensemble, en termes des fonctionnalités attendues de celui-ci. Des tests de régression permettent de s'assurer, lorsqu'une ou plusieurs modifications ont été effectuées à un module ou au programme, que rien n'a été brisé, que ce qui fonctionnait correctement auparavant fonctionne toujours après que les modifications ont été faites.[1]

En gros, un script (*C-shell script*) et un `Makefile` ont été définis pour automatiser l'exécution des tests, l'exécution de ces tests étant lancée simplement à l'aide de la commande «`make tests`». Le rôle du script est d'exécuter le programme sur différents cas de tests, chacun d'entre eux étant défini par un groupe de quatre fichiers :

- Un fichier contenant une série de commandes qu'on veut exécuter et dont on veut vérifier le bon fonctionnement.
- Un fichier décrivant l'état de la base de données avant l'exécution de la série de commandes.
- Un fichier décrivant les résultats produits sur `stdout` suite à l'exécution des commandes qu'on a voulu tester.
- Un fichier décrivant l'état de la base de données après l'exécution des diverses commandes.

Pour chacun des cas de test, les commandes spécifiées sont exécutées par le programme. Les résultats obtenus et l'état résultant de la base de données, tous des informations textuelles, sont alors comparés à l'aide de la commande `diff` avec les résultats attendus et des erreurs sont signalés si des différences existent. L'annexe B présente de façon plus détaillée cette stratégie de tests.

Addendum : Signalons que, par rapport à la version initiale décrite dans ce document, des modifications ont été effectuées pour les tests de niveau système. Ainsi, deux fichiers additionnels doivent maintenant être définis pour chacun des cas de tests :

[1] Certains auteurs disent plutôt «*tests de non régression*».

- Un fichier décrivant l'état, *avant* l'exécution des commandes, du *journal des courriels envoyés* ;
- Un fichier décrivant l'état de ce même journal *après* l'exécution des commandes.

Plus précisément, ce journal contient une copie de tous les courriels envoyés suite à des commandes `rappeler-livre` ou `rappeler-tous-les-livres`.

3.2 Mise sous contrôle de `CVS`

> Le contrôle de version (ou le contrôle de révision) consiste à maintenir des informations sur l'évolution d'un projet de façon à pouvoir retrouver des versions antérieures de fichiers, suivre les modifications et (souvent le plus important) coordonner les efforts d'une équipe de développement.
>
> (G.N. Purdy, «CVS précis & concis», Éditions O'Reilly, 2004)

De façon à pouvoir suivre l'évolution du logiciel de gestion de prêts de livres tout en permettant à plusieurs personnes de travailler en même temps à l'amélioration du logiciel, les divers fichiers définissant le logiciel, y compris les scripts de tests et les jeux d'essai, ont été mis sous le contrôle de `CVS` — *Concurrent Versions System* — un outil de gestion de configuration.

CVS, contrairement à d'autres outils semblables, permet à plusieurs développeurs de travailler sur les mêmes fichiers en parallèle : les changements qui ne créent pas de conflit sont automatiquement acceptés, les autres doivent être approuvés par l'usager.

Avec CVS, la «version maître» du logiciel est conservée dans un dépôt (une archive, en anglais «*a repository*») centralisé. Chaque développeur travaille donc sur une copie locale, sans interférence avec les autres développeurs, sauf aux moments de la sauvegarde des modifications (`commit`).

Pour plus de détails sur `CVS` et son utilisation, voir les notes de cours sur le site *Web* du cours INF3135 :

```
http://www.info2.uqam.ca/~tremblay/INF3135
```

3.3 Restructuration du logiciel et découpage en modules

Le logiciel de gestion de prêts de livres a été décomposé en plusieurs modules relativement indépendants. Ces divers modules peuvent être regroupés en différentes catégories, tel que présenté dans les sections qui suivent.

La figure 3.1 présente une vue d'ensemble de ces modules, et ce à l'aide d'un diagramme qui indique les dépendances existant entre ces divers modules : un module `M1` dépend d'un module `M2` si `M1`, soit dans son fichier d'interface (extension `.h`), soit dans son fichier de mise en oeuvre (extension `.c`), importe le module `M2` — en d'autres mots, si `M1.h` ou `M1.c` contient une inclusion de la forme «`#include "M2.h"`».

Signalons que la documentation détaillée (dans le format `DOC++` décrit à la section 3.4) de ces divers modules peut être consultée en ligne à l'adresse suivante :

```
http://www.info2.uqam.ca/~tremblay/INF3135/Biblio
```

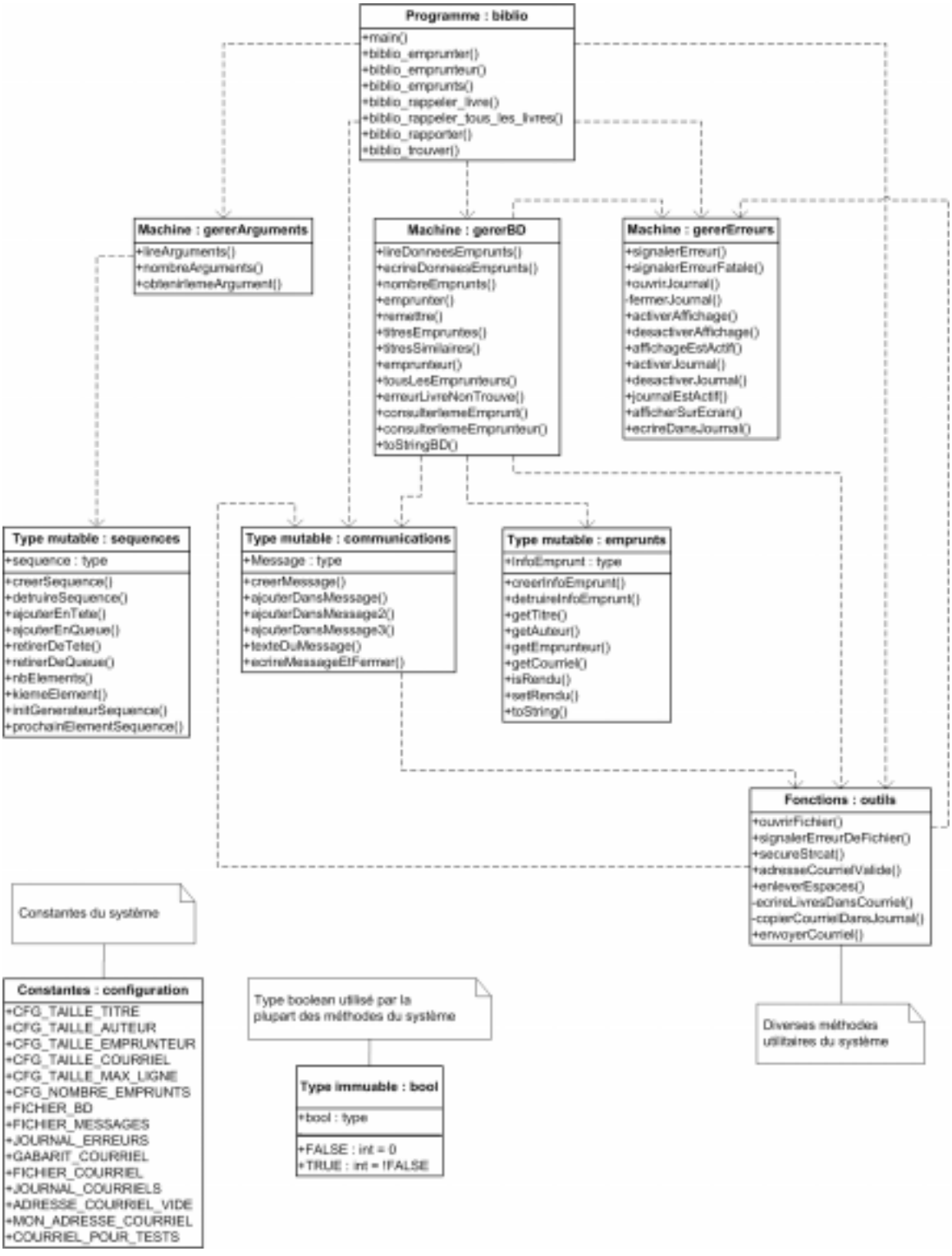

Figure 3.1: Diagramme illustrant l'organisation générale des modules du système de gestion des livres (diagramme de dépendances)

3.3.1 Programme principal

- Programme (`main`) `biblio` : Programme principal (répartiteur) pour le logiciel de gestion des prêts de livres.

  Contrairement à la version initiale, on a maintenant un unique programme principal, le premier argument du programme étant l'opération à effectuer. Au niveau de la ligne de commande, les appels pour l'utilisation du programme ont donc maintenant l'allure suivante — on notera que l'utilisation du caractère «^» pour séparer les arguments n'est plus nécessaire, les arguments composés de plusieurs mots pouvant simplement être mis entre `"..."` ; on notera aussi que la commande `rappeler-livre` ne dépend plus de la commande `emprunteur` mais reçoit directement en argument le titre du livre à rappeler :

  ```
  biblio emprunter Tremblay tremblay.guy@uqam.ca "Code complet" McConnell
  ...
  biblio emprunts Tremblay
  ...
  biblio emprunteur "Code complet"
  biblio rappeler-livre "Code complet"
  ...
  biblio rapporter "Code complet"
  ...
  biblio rappeler-tous-les-livres
  ```

  Dans la nouvelle version, le programme principal est devenu essentiellement un «répartiteur» (un *dispatcher*) : il lit les arguments spécifiés sur la ligne de commande (voir le module `gererArguments`, section 3.3.4), puis appelle la routine qui va effectuer le travail demandé en lui spécifiant où les messages d'erreurs devront être transmis (`fluxMessages`). Les diverses procédures auxiliaires (privées au fichier `biblio.c`) qui mettent en oeuvre les diverses opérations du logiciel sont alors les suivantes :

  ```
  void biblio_emprunter               ( Message fluxMessages );
  void biblio_rapporter               ( Message fluxMessages );
  void biblio_emprunts                ( Message fluxMessages );
  void biblio_emprunteur              ( Message fluxMessages );
  void biblio_rappeler_livre          ( Message fluxMessages );
  void biblio_rappeler_tous_les_livres( Message fluxMessages );
  ```

  Chacune de ces routines a essentiellement pour tâche d'effectuer les vérifications sur les arguments. Si les arguments spécifiés sont valides, la sous-routine qui fait le véritable travail (par exemple, accès ou modification à la base de données) est alors appelée ; dans le cas contraire, un message d'erreur est signalé. Par exemple, la routine `biblio_emprunteur` a l'allure indiquée dans l'extrait de code C 1 (certains éléments ont été supprimés pour simplifier la présentation) — dans ce cas, c'est la routine `emprunteur`, du module `gererBD` (section 3.3.2), qui effectue l'accès effectif aux structures internes modélisant la base de données.

3.3.2 Machine et type liés à la gestion des emprunts et de la base de données

- Machine `gererBD` : Module (machine abstraite) pour gérer la base de données (fichier texte).

  C'est ce module qui dissimule tous les détails liés à la lecture et à l'écriture la base de données (textuelle). La base de données doit tout d'abord être chargée en mémoire (`lireDonnees-Emprunts`). Une fois cela fait, on peut alors utiliser les diverses opérations fournies par le module permettant d'examiner ou modifier les informations sur les emprunts actifs — par exemple,

```
static void biblio_emprunteur( Message fluxMessages ) {
  char *messageErreur;
  char *citoyenX;

  if (nombreArguments() != 2) {
     // Construction du message dans messageErreur.
     ...
     signalerErreurFatale( messageErreur );
  }

  // Pas d'erreur dans les arguments: on accede à la base de données
  // pour déterminer l'emprunteur.
  citoyenX = emprunteur( obtenirIemeArgument(1) );

  // On genere un message pour afficher l'emprunteur.
  ajouterDansMessage( fluxMessages, "L'emprunteur de \"" );
  ajouterDansMessage( fluxMessages, obtenirIemeArgument(1) );
  ajouterDansMessage( fluxMessages, "\" est " );
  ajouterDansMessage( fluxMessages, citoyenX );
}
```

Extrait de code C 1: Extrait du code pour la routine `biblio_emprunteur`

`nombreEmprunts`, `emprunter`, `remettre`, `titresEmpruntes`, `emprunteur`, `tousLesEmprunteurs`, etc. Finalement, à la fin du programme, on doit ensuite mettre à jour le fichier de la base de données à l'aide de `ecrireDonneesEmprunts`.

- Type `emprunts` : Module (type abstrait pour classe d'objets) pour représenter des emprunts (de livres).

  Ce module définit un type abstrait `InfoEmprunt` et l'opération à utiliser pour créer une instance de ce type est `creerInfoEmprunt`, où l'on spécifie les différents attributs caractérisant un emprunt (titre, auteur, emprunteur, adresse de courriel). Une fois une instance créée, on peut obtenir les attributs associés à l'aide de divers observateurs : `getTitre`, `getAuteur`, `getEmprunteur`, `getCourriel` et `isRendu`. On peut aussi modifier l'état de l'emprunt à l'aide de `setRendu` — les autres champs sont immuables.

3.3.3 Machine et type liés à la gestion des erreurs et des communications entre opérations

- Machine `gererErreurs` : Module (machine abstraite) pour gérer les erreurs.

  Ce module encapsule tous les détails liés à l'affichage des messages d'erreur, tant l'affichage à l'écran (`stdout`) que l'écriture dans le *journal des erreurs* (*error log file*). L'affichage des erreurs et l'écriture de ces erreurs dans le journal peuvent être activés/désactivés (`activerAffichage`, `desactiverAffichage`, `activerJournal`, `desactiverJournal`).

  L'opération principale pour signaler une erreur non fatale est `signalerErreur`. L'opération `signalerErreurFatale`, quant à elle, permet de signaler une erreur fatale, c'est-à-dire qui signale une erreur puis qui avorte l'exécution du programme.

- Type `communications` : Module (type abstrait pour classe d'objets) pour la gestion des messages servant à la communication des résultats.

  Lorsqu'une commande est exécutée, un flux pour les messages, tant pour les résultats que pour les erreurs, est créé. Toutes les communications d'information avec l'usager se font alors via ce

flux. Une fois la commande complétée, le contenu du flux est transmis sur `stdout`, *et* est aussi transmis dans un fichier (`resultats.txt`), lequel permet donc de conserver les détails de la dernière commande exécutée, ce qui peut être utile pour permettre aux différentes opérations de s'échanger de l'information. L'allure générale pour l'utilisation d'un flux de message aura donc l'allure suivante :

```
Message fluxMessages = creerMessage(FICHIER_MESSAGES);
...
ajouterDansMessage( fluxMessages,  texte_a_ajouter );
ajouterDansMessage( fluxMessages,  texte_a_ajouter );
...
printf( "%s\n", texteDuMessage(fluxMessages) );
ecrireMessageEtFermer( fluxMessages );
```

3.3.4 Modules auxiliaires

- Bibliothèque `configuration` : Module (bibliothèque de constantes) pour la configuration du logiciel. Ces constantes se regroupent essentiellement en deux catégories :
  - Des constantes utilisées pour spécifier les tailles maximales pour l'allocation statique de diverses structures de données.
  - Des constantes définissant divers noms de fichier utilisés par le logiciel, par exemple, `FICHIER_BD` qui indique le nom du fichier pour la base de données (textuelle).
  - Une constante, `COURRIEL_POUR_TESTS`, utilisée comme adresse de courriel durant les tests.

    Plus précisément, dans un environnement de tests — par opposition à un environnement de production, où le logiciel est réellement utilisé —, cette constante doit être définie de façon à ce que les adresses de courriel indiquées dans la base de données *soient ignorées* lors de l'exécution de commandes entraînant l'envoi de courriels. Les courriels seront plutôt envoyés à l'adresse indiquée par `COURRIEL_POUR_TESTS`, et ce peu importe ce qui est indiqué dans la base de données. Il est donc important qu'un développeur qui veut exécuter des tests définisse cette adresse de façon *à ce que les courriels générés lors de l'éxécution de tests lui soient envoyés.*[2] Un développeur peut aussi utiliser le symbole `ADRESSE_COURRIEL_VIDE` comme adresse de courriel pour les tests, auquel cas les courriels ne seront pas vraiment envoyés, même s'ils apparaîtront dans le journal des courriels envoyés. Dans un environnement de production, il suffira simplement *de ne pas définir cette constante.*

- Machine `gererArguments` : Module (machine abstraite) pour gérer les arguments transmis sur la ligne de commande, i.e., pour obtenir ces divers arguments.

- Bibliothèque `outils` : Module (bibliothèque de fonctions) définissant diverses opérations auxiliaires utilisées par d'autres modules : `ouvrirFichier`, `signalerErreurDeFichier`, `secureStrcat`, `adresseCourrielValide`, `enleverEspaces`, `envoyerCourriel`.

- Type `bool` : Type et constantes pour des booléens, définissant le type `bool` et les deux constante `TRUE` et `FALSE`.

- Type `sequence` : Module (type abstrait définissant une classe d'objets) pour des séquences *génériques.*

[2] Dans les jeux de tests décrits à la section 3.1, la plupart des adresses indiquées sont des adresses *bidons*, donc vont générer des erreurs au niveau du programme `sendmail` si on tente de les transmettre.

3.4 Documentation avec DOC++

```
/** Module (machine abstraite) pour gérer les arguments de la ligne de
    commande.

    Créé à l'automne 2005 dans le cadre du projet FDP.

    @name gererArguments
    @author Pablo Zentilli et Guy Tremblay
*/


//@{

/** Lit et mémorise les arguments de la ligne de commande.
   @param argc Le nombre d'arguments de la ligne de commande
   @param argv Les arguments de la ligne de commande qu'il faut traiter
*/
void lireArguments( int argc, char **argv );


/** Retourne le nombre d'arguments memorisés.

   @precondition La procédure lireArguments() a été appelée
   @return nombreArguments() Le nombre d'arguments mémorisés
*/
int nombreArguments();


/** Retourne le ième argument.

   @precondition La procédure lireArguments() a été appelée et i < nombreArguments()
   @param i Position de l'argument cherché
   @return obtenirIemeArgument() Le ième argument spécifié sur la
        ligne de commande
*/
char *obtenirIemeArgument( int i );

//@}
```

Fichier d'interface 1: Fichier `gererArguments.h` documenté avec DOC++

De façon à bien documenter l'interface des divers modules, les principaux fichiers d'interface (i.e., les fichiers `*.h`) ainsi que les programmes exécutables (programme principal et programmes de tests) du logiciel de gestion de prêts de livres ont été documentés à l'aide de commentaires de type DOC++.

L'outil DOC++ est un outil semblable à JavaDoc, mais conçu pour les langages C et C++. Tel qu'on peut le voir dans le Fichier d'interface 1, pour une routine (procédure ou fonction), DOC++ permet, à l'aide d'un commentaire spécial qui apparaît *avant* la déclaration de l'en-tête de la routine et qui débute par «`/**`» et se termine par «`*/`», d'indiquer le rôle de la routine (la première phrase qui

apparaît immédiatement après le «/**») ainsi que divers autres éléments :

- `@param` *nomDuParamètre Description du paramètre*
- `@return` *nomDuRésultat Description du résultat*
- `@precondition` *Pré-condition requise pour le bon fonctionnement de la routine*
- `@postcondition` *Post-condition décrivant l'effet de la routine*

Pour le module dans son ensemble — où les routines sont encadrées par les commentaires spéciaux «//@{» et «//@}» —, DOC++ permet aussi d'inclure, au début du fichier, une brève description du rôle du module (première phrase après le «/**») ainsi que divers autres éléments :

- `@name` *Nom du module*
- `@author` *Auteur(s) du module*

```
gererArguments

    Module (machine abstraite) pour gérer les arguments de la ligne de commande.

o lireArguments
    Lit et mémorise les arguments de la ligne de commande.
o nombreArguments
    Retourne le nombre d'arguments mémorisés.
o obtenirIemeArgument
    Retourne le ième argument.

    Module (machine abstraite) pour gérer les arguments de la ligne de commande.

    Créé à l'automne 2005 dans le cadre du projet FDP.

Auteur :
    Pablo Zentilli et Guy Tremblay

Index alphabétique

This page was generated with the help of DOC++.
```

Figure 1: La page *Web* générée par DOC++ pour le fichier `gererArguments.h`

La figure 1 présente la page *Web* générée par DOC++ pour le fichier `gererArguments.h`. Les éléments soulignés sont des hyper-liens vers les descriptions des routines. L'annexe C (p. 34) présente aussi, cette fois en format LaTeX, les descriptions générées par DOC++ pour chacune de ces routines.

La documentation détaillée et complète des divers modules peut être consultée en ligne à l'adresse suivante :

`http://www.info2.uqam.ca/~tremblay/INF3135/Biblio`

L'allure de la page principale est présentée à la figure 3.

3.5 Développement de tests unitaires

À la section précédente, nous avons décrit de quelle façon le logiciel de gestion de prêts de livres a été découpé en modules indépendants. En plus de tests pour vérifier le bon fonctionnement de

Table des matières

General stuff

- `MiniCUnit` Mini cadre de tests unitaires pour le langage C.
- `biblio` (main) Programme principal (répartiteur) pour le logiciel de gestion des prêts de livres.
- `bool` Type et constantes pour des booléens.
- `communications` Module (type abstrait pour classe d'objets) pour la gestion des messages (communications entre modules et opérations).
- `configuration` Module (bibliothèque de constantes) pour la configuration du logiciel (pour l'allocation statique de diverses structures de données).
- `emprunts` Module (type abstrait pour classe d'objets) pour représenter des emprunts (de livres).
- `gererArguments` Module (machine abstraite) pour gérer les arguments de la ligne de commande.
- `gererBD` Module (machine abstraite) pour gérer la BD (fichier texte).
- `gererErreurs` Module (machine abstraite) pour gérer les erreurs
- `outils` Module (bibliothèque de fonctions) définissant diverses opérations auxiliaires utilisées par d'autres modules.
- `sequences` Module (type abstrait définissant une classe d'objets) pour des séquences génériques.
- `testCommunications` (main) Programme de tests (unitaires) pour le module de gestion des communications.
- `testEmprunts` (main) Programme de tests (unitaires) pour le module des emprunts.
- `testGererArguments` (main) Programme de tests (unitaires) pour le module de gestion des arguments.
- `testGererBD` (main) Programme de tests (unitaires) pour le module de gestion de la base de données.
- `testGererErreurs` (main) Programme de tests (unitaires) pour le module de gestion des erreurs.
- `testOutils` (main) Programme de tests (unitaires) pour le module des outils auxiliaires.

This page was generated with the help of DOC++.

Figure 3: La page *Web* générée par `DOC++` pour l'ensemble du logiciel

l'ensemble du logiciel (tests de niveau système), il est utile d'avoir des tests qui peuvent vérifier le bon fonctionnement de chacun des modules — ce qu'on appelle des *tests unitaires*.

Divers outils existent pour automatiser l'exécution des tests unitaires. Une particularité importante de ces outils est que le bon fonctionnement d'un module, comme pour les tests de niveau système définis à la section 3.1, est vérifié à l'aide d'*assertions*. Ceci implique que si le programme satisfait tous les tests, alors aucun résultat (ou presque) ne sera produit — en d'autres mots, c'est uniquement (ou presque) lorsqu'il y a des problèmes que les tests signalent quelque chose, autrement (presque) rien n'est signalé.

Des tests unitaires ont commencé à être développés (mais ils ne sont pas encore tous complétés) pour certains des modules du logiciel `biblio`, et ce à l'aide de programmes de tests écrits en `C` et utilisant un *mini cadre de tests unitaires* (`MiniCUnit`) que nous avons développé — et que nous expliquons brièvement dans les sections qui suivent. L'exécution de ces programmes de tests unitaires, comme pour les tests de niveau système, est lancée simplement à l'aide d'une commande `make`, soit «`make testsu`».

Plus précisément, les programmes suivants pour l'exécution de tests unitaires ont été définis, bien que certains de ces programmes restent encore à être complétés :

- `testCommunications` : Programme de tests (unitaires) pour le module de gestion des communications.
- `testEmprunts` : Programme de tests (unitaires) pour le module des emprunts.
- `testGererArguments` : Programme de tests (unitaires) pour le module de gestion des arguments.
- `testGererBD` : Programme de tests (unitaires) pour le module de gestion de la base de données.
- `testGererErreurs` : Programme de tests (unitaires) pour le module de gestion des erreurs.
- `testOutils` : Programme de tests (unitaires) pour le module des outils auxiliaires.

Dans les sections qui suivent, nous présentons quelques exemples illustrant le fonctionnement de ces programmes de tests unitaires. Mais tout d'abord, nous expliquons brièvement l'utilisation du cadre de tests `MiniCUnit`.

3.5.1 Utilisation du cadre de tests `MiniCUnit`

Le cadre de tests utilisé pour les tests unitaires a initialement été inspiré du cadre de tests `MinUnit` décrit à l'URL suivant :

```
http://www.jera.com/techinfo/jtns/jtn002.html
```

`MinUnit` est défini principalement à l'aide de quelques *macros*, qui utilisent un certain nombre de variables pour prendre note du nombre de tests exécutés et du nombre d'assertions évaluées. Il s'agit d'un cadre de tests extrêmement minimal et, surtout, assez peu robuste, puisque l'exécution des divers cas de tests se termine aussitôt qu'une assertion erronée est rencontrée.

Dans une première version de notre cadre de tests, nous avons utilisé essentiellement la même approche que `MinUnit`. Par la suite, nous avons développé une deuxième version, que nous avons appelée `MiniCUnit`, de façon à rendre le cadre de tests plus robuste, tant au niveau statique — utilisation limitée de macros, avec utilisation plutôt de fonctions et procédures — qu'au niveau dynamique — l'exécution des cas et suites de tests se poursuit même si une assertion est fausse. De plus, nous avons aussi ajouté diverses variantes d'opérations d'assertion (voir plus bas).

Le patron général d'utilisation est le suivant. Tout d'abord, un programme de test (fonction `main`) a typiquement la forme illustrée dans l'extrait de code C 2 : on fait un ou plusieurs appels à `executerSuiteDeTests`, suivi d'un appel à `sommaireDeTests` pour obtenir et imprimer une chaîne qui représente le sommaire de l'exécution des diverses suites et divers cas de tests. Ce sommaire indique différentes informations quant au nombre de suites de tests exécutées, au nombre de cas de tests, d'assertions, etc.

```
#include "MiniCUnit.h"

static SuiteDeTests suite();

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

  printf( "%s", sommaireDeTests() );
  return 0;
}

// Définition des divers cas de tests.
CAS_DE_TEST( test1 )
 ...
FIN_CAS_DE_TEST

  .
  .
  .

CAS_DE_TEST( testK )
 ...
FIN_CAS_DE_TEST


// Définition d'une suite de tests.
SUITE_DE_TESTS( suite )
  ajouterCasDeTest( test1 );
  ...
  ajouterCasDeTest( testK );
FIN_SUITE_DE_TESTS
```

Extrait de code C 2: Forme typique d'un programme de tests utilisant le cadre de tests `MiniCUnit`

Ensuite, le fichier définissant le programme de tests définit les différents cas de tests. Ces différents cas de tests peuvent utiliser l'une ou l'autre des routines suivantes pour exprimer des assertions :

- `void assertVrai( bool test, char* message )` : Vérifie qu'une expression booléenne est bien vraie.
- `void assertFaux( bool test, char* message )` : Vérifie qu'une expression booléenne est bien fausse.
- `void assertChainesEgales( char* ch1, char* ch2, char* message )` : vérifie que deux chaînes de caractères sont égales.
- `void assertEntiersEgaux( int v1, int v2, char* message )` : Vérifie que deux entiers sont égaux.

Dans tous les cas, rien n'est affiché si l'assertion est vérifiée. Dans le cas contraire, la chaîne indiquée comme dernier argument (`message`) est affichée sur `stdout`, avec dans certains cas des informations supplémentaires (dans les deux derniers cas, les valeurs des chaînes ou des entiers ayant été comparés sont affichées).

Finalement, toujours dans le programme de tests, la ou les suites de tests comme telles sont définies (`SUITE_DE_TESTS`), et ce en indiquant explicitement, avec `ajouterCasDeTest`, chacun des cas de tests qui devront être associés à cette suite. Signalons qu'il est effectivement possible de définir plusieurs suites de tests, auquel cas l'ensemble des informations pour ces diverses vont apparaître dans le sommaire.

3.5.2 Exemple tiré du programme `testGererArguments.c`

Trois cas de tests ont été définis pour la vérification du module `gererArguments` (voir section 3.4 pour la description de son interface). Voici l'un de ces tests :

```
CAS_DE_TEST( test1 )
  int argc = 3;
  char *argv[argc];
  argv[1] = "Le";
  argv[2] = "chien";

  lireArguments(argc, argv);
  assertEntiersEgaux( nombreArguments(), 2, "Nombre d'arguments incorrect" );
  assertChainesEgales( "Le", obtenirIemeArgument(0), "Zeroieme argument incorrect" );
  assertChainesEgales( "chien", obtenirIemeArgument(1), "Premier argument incorrect" );
FIN_CAS_DE_TEST
```

Lorsqu'aucune erreur n'est présente, l'exécution de la suite de tests dont fait partie ce cas de test génère alors le résultat suivant :

```
testGererArguments
*** Sommaire: 1 suites executees; 3 tests executes, 0 tests echoues; 9 assertions evaluees, 0 assertions echouees
```

3.5.3 Exemples tirés du programme `testEmprunts.c`

Cinq cas de tests ont été définis pour la vérification du module `emprunts`. Dans ce qui suit, nous présentons deux d'entre eux. Le premier cas de test vérifie que le constructeur et les observateurs associés fonctionnent correctement :

```
CAS_DE_TEST( testNewEmpruntEtGets1 )
  InfoEmprunt e1 = creerInfoEmprunt( "titre", "auteur", "emprunteur", "email@toto.com" );

  assertChainesEgales( getTitre(e1),      "titre",          "Titre incorrect" );
  assertChainesEgales( getAuteur(e1),     "auteur",         "Auteur incorrect" );
  assertChainesEgales( getEmprunteur(e1), "emprunteur",     "Emprunteur incorrect" );
  assertChainesEgales( getCourriel(e1),   "email@toto.com", "Courriel incorrect" );
FIN_CAS_DE_TEST
```

Le cas de test suivant, quant à lui, vérifie que l'attribut mutable indiquant si un emprunt est rendu ou non fonctionne correctement :

```
CAS_DE_TEST( testGetSetEffacerDeBD )
  InfoEmprunt e1 = creerInfoEmprunt( "titre", "auteur", "emprunteur", "toto.toto@toto.com" );

  assertFaux( isRendu(e1), "L'emprunt e1 est indique comme non rendu mais ne devrait pas l'etre" );

  setRendu(e1, TRUE);
  assertVrai( isRendu(e1), "L'emprunt e1 est indique comme non rendu mais ne devrait pas l'etre" );
FIN_CAS_DE_TEST
```

3.6 Ajout d'assertions

> *A useful technique is to add code to handle "can't happen" cases, situations where it is not logically possible for something to happen but (because of some failure elsewhere) it might anyway. [...] Defensive programming [is] making sure that a program protects itself against incorrect use or illegal data. Null pointers, out of range subscripts, division by zero, and other errors can be detected early and warned about or deflected.*
> *(B. Kernighan et R. Pike, «The practice of programming», Addison-Wesley, 1999)*

La spécification d'un service fourni par un composant logiciel peut s'exprimer à l'aide de pré-conditions et post-conditions (on dit aussi *antécédents* et *conséquents*). Durant l'exécution d'un programme, il arrive qu'une pré-condition ou qu'une post-condition ne soit pas vérifiée. Dans un tel cas, c'est donc qu'il y a une *erreur* (un *bogue*) *dans le programme*. Plus précisément :

- La violation d'une pré-condition est le signe d'une erreur dans le code *du client* : le client ne s'est pas assuré avant l'appel du service, comme il aurait dû le faire, que ses arguments étaient valides étant donné le service qu'il désirait utiliser. L'exécution du service demandé *ne devrait donc pas* se faire.
- La violation d'une post-condition est le signe d'une erreur dans le code *du fournisseur* : le fournisseur n'a pas été capable de fournir le service demandé. Si la situation est telle qu'il soit impossible de corriger cette erreur, alors l'exécution du programme devrait être avortée.

De façon à rendre le logiciel de gestion de prêts de livres plus robuste, des assertions ont été ajoutées à certains endroits (mais, faute de temps, pas partout) dans les fichiers de mise en oeuvre des divers modules (fichiers `*.c`).

3.6.1 Exemple tiré du fichier `gererArguments.h`

Dans le fichier `gererArguments.h`, on retrouve la pré-condition suivante pour la routine `nombreArguments()` :

```
/** Retourne le nombre d'arguments memorisés.
    @precondition La procédure lireArguments() a été appelée
    @return nombreArguments() Le nombre d'arguments mémorisés
*/
int nombreArguments();
```

Dans la mise en oeuvre du fichier `gererArguments.c`, les segments de code suivants ont été définis, qui permettent de s'assurer que la pré-condition est bien satisfaite — dans le cas contraire, l'exécution du programme est alors avortée :

```
static Sequence *ARG_LISTE_ARGUMENTS = NULL;

void lireArguments( int argc, char **argv ) {
  ...
  ARG_LISTE_ARGUMENTS = creerSequence();
  ...
}

int nombreArguments() {
  // L'assertion suivante sera fausse seulement si lireArguments() n'a pas été appelée.
  assert( ARG_LISTE_ARGUMENTS != NULL );
  return nbElements(ARG_LISTE_ARGUMENTS);
}
```

3.6.2 Exemple tiré du fichier `emprunt.h`

Le prochain exemple est tiré du fichier `emprunt.h`, où l'on retrouve la pré-condition suivante pour la routine `getTitre` :

```
/** Retourne le titre du livre de l'emprunt

    @precondition L'emprunt existe (a été créé par creerInfoEmprunt)
    @param emprunt L'emprunt dont on veut connaître le titre du livre
    @return Titre de l'emprunt
*/
char *getTitre( InfoEmprunt emprunt );
```

Dans la mise en oeuvre de `getTitre()`, on retrouve donc l'assertion suivante :

```
char *getTitre( InfoEmprunt emprunt ) {
  assert(emprunt != NULL);
  return emprunt->Titre;
}
```

Chapitre 4

Travaux futurs

Dans cette section, nous décrivons brièvement divers travaux futurs qui pourraient être effectués sur le logiciel de gestion de prêts de livres, soit pour améliorer ou rendre plus robuste les opérations déjà mises en oeuvre, soit pour ajouter de nouvelles fonctionnalités.[1]

- [LISTER] Mettre en oeuvre l'opération `lister` (décrite à la section 2.4).

  Actuellement, cette opération est réalisée par un *shell script*, plutôt qu'être mise en oeuvre par le logiciel de gestion de prêts de livres. Une contrainte importante de mise en oeuvre à respecter devrait être que l'affichage n'est pas spécifié dans la base de données elle-même, puisque cela n'a rien à voir avec les données abstraites.

  Une généralisation possible pourrait être de pouvoir spécifier, en argument à la commande, le *format* dans lequel le listage des différents champs devrait s'effectuer. Par exemple, on pourrait spécifier un argument qui serait une chaîne de format dans le style suivant, exemple qui représente ici le format par défaut d'affichage :

  ```
  biblio lister " %-11e:: [ %-9a] \"%-20t \"\n"
  ```

  On aurait alors le choix parmi quatre indicateurs de format, l'élément associé étant toujours une chaîne :

  - `e` : Emprunteur.
  - `t` : Titre du document emprunté.
  - `a` : Auteur(s) du document emprunté.
  - `c` : Courriel de l'emprunteur.

- [RAPPORTER_LIVRES] Ajouter une commande `rapporter-livres` qui permet d'indiquer soit qu'un emprunteur donné a rapporté tous les livres qu'il avait empruntés, soit que plusieurs livres ont été rapportés, par exemple :

  ```
  biblio rapporter-livres --emprunteur Tremblay
  biblio rapporter-livres --livres "La programmation en pratique" "Code complet"
  ```

- ⋆ Mettre en oeuvre l'opération `trouver` (décrite à la section 2.4).

  Actuellement, cette opération est réalisée par un *shell script*, plutôt qu'être mise en oeuvre par le logiciel de gestion de prêts de livres.

[1] Signalons que certains des items, notés par ⋆, ont déjà été mis en oeuvre dans le cadre des travaux pratiques réalisés à des trimestres antérieurs.

- Pour l'opération `trouver`, il serait *intéressant* que le résultat retourné, lorsqu'il dénote un titre unique, puisse être utilisé comme argument à d'autres commandes. Par exemple, si le livre «`La programmation en pratique`» a été emprunté et que ce livre est le seul avec le mot «`pratique`» dans son titre, alors l'un ou l'autre des appels suivants devrait pouvoir permettre de signaler le retour de ce livre — rappelons que ‘`commande`‘ est une expression qui a pour effet d'exécuter la commande indiquée :

  ```
  biblio rapporter "La programmation en pratique"
  biblio rapporter ‘biblio trouver pratique‘
  ```

- [`RAPPORTER`] Une possibilité intéressante pour `rapporter` serait de permettre que le titre puisse n'être que *partiellement spécifié*. Si un seul emprunt contient le morceau de titre spécifié, alors l'emprunt est supprimé. Par contre, si plusieurs emprunts contiennent l'élément spécifié, alors un message d'erreur est signalé et aucun emprunt n'est supprimé.

  De plus, dans le cas, où plusieurs emprunts contiennent le même titre de volume (complet ou partiel), il serait aussi possible de spécifier le nom de l'emprunteur, par exemple :

  ```
  biblio rapporter "Modal and temporal" Tremblay
  ```

  Finalement, il pourrait aussi être possible d'indiquer que les minuscules/majuscules peuvent être ignorées lors de la recherche :

  ```
  biblio -i rapporter "modal AND Temporal" TrEmbLay
  ```

- Une possibilité intéressante pour `trouver` serait de permettre une forme limitée d'expressions régulières, par exemple :

  - `trouver ".*the.*programmer.*"` : Retourne tous les titres de livres contenant le mot "`the`" suivi, plus loin, du mot "`programmer`".
  - `trouver "^[A-Z].*[0-9]$"` : Retourne tous les titres de livres débutant par une lettre majuscule et se terminant par un chiffre

  Des formes plus simples d'expressions, comme celles permises au niveau du *shell*, pourraient aussi être utilisées à la place d'expressions plus générales comme les précédentes :

  - `trouver "*the*programmer*"`.
  - `trouver "[A-Z]*[0-9]"`.

  De plus, si une forme d'expressions régulières est supportée, `rapporter`, `rapporter-livres` et `rappeler` pourraient aussi recevoir en argument de telles expressions

- [`SIGNALER_RETARDS`] Ajouter, dans la base de donnée, la date où le prêt a été effectué.

  Permettre de trouver et rappeler tous les livres ayant été prêtés depuis une certaine période, par exemple, depuis plus de deux (2) mois, par exemple, à l'aide de la commande suivante :

  ```
  biblio signaler-retards
  ```

  Ajouter aussi les jeux d'essai (de niveau système) appropriés.

- Modifier la structure de données utilisée pour les emprunts (dans `gererBD`) pour utiliser une liste chaînée (taille variable) plutôt qu'un tableau (taille fixe spécifiée à la compilation dans le fichier `configuration.h`).

  Définir un *itérateur* pour obtenir les différents emprunteurs (actuellement générés avec `tousLes-Emprunteurs`).

- Compléter les tests, principalement les tests unitaires.

- [LISTE_DE_COMMANDES] Traiter de façon spéciale le cas où la ligne de commande est vide : lorsqu'il n'y aucun argument spécifié, alors indiquer les diverses commandes possibles.
- Gérer correctement les majuscules/minuscules dans les entrées de l'utilisateur.
- Examiner le code pour identifier les «destructeurs», puis les mettre en oeuvre et les utiliser.
- Éliminer les diverses «valeur magiques» restantes, y compris les noms des commandes à exécuter, noms qui devraient eux aussi pouvoir être facilement modifiables.
- Ajouter des pré-conditions à différents endroits dans le code où il en manque (par exemple, dans le module `Sequences`).
- Définir des «patrons» pour les messages, tant pour les erreurs que pour les résultats, de façon à pouvoir facilement changer ces messages sans devoir aller jouer dans les détails du code pour `biblio.c`.

  Actuellement, la plupart des messages de sortie (tant messages d'erreurs que messages pour indiquer le résultat correct d'une commande) se retrouvent éparpillés un peu partout dans le fichier `biblio.c`.

  Note : Cette modification permettrait aussi de faciliter «l'internationalisation» du programme, i.e., faire en sorte de garder le même code, mais avec des messages dans une langue différente.
- [RAPPELER_LIVRES] Définir une opération de rappel de livres (par courriel) qui prend explicitement en argument le nom de l'emprunteur, le courriel demandant alors le rappel de *tous les livres empruntés* :

  ```
  biblio rappeler-livres --emprunteur Tremblay
  ```
- [RAPPELER_TOUS_LES_LIVRES] L'opération `rappeler-tous-les-livres`, quant à elle, ne permet que d'envoyer des courriel à tous les emprunteurs, sans exception, et ce sans indiquer de façon explicite dans le courriel les livres devant être rapportés. Il pourrait être intéressant de la modifier pour que les livres prêtés soient indiqués dans le courriel transmis.
- ⋆ [BIBLIO+DISPATCHER] Le style du programme principal (`biblio.c`) pourrait être amélioré de différentes façons, entre autres :
  - Certaines constantes «magiques» sont utilisées dans le code pour allouer des tableaux. Or, ceci pourrait faire en sorte que, dans certains cas, le programme ne fonctionne pas correctement (par exemple, si un usager possède un très grand nombre d'emprunts).
  - Il faudrait éviter d'initialiser des variables lorsque la valeur initiale n'est jamais utilisée.
  - Certaines utilisation de `secureStrcat` pourraient être simplifiées, en passant directement comme arguments les deux chaînes à concaténer (donc sans passer par une chaîne initialement vide).
  - Certaines variables sont utilisées localement dans un bloc, mais sont définies (inutilement) au niveau global de la procédure.
  - La convention pour les noms de variable ne semble pas uniforme. Ainsi, certains noms de variables débutent par des minuscules, alors que d'autres débutent par des majuscules.
  - La procédure `ajouterDansMessage` ne prend qu'un seul et unique argument. Or, à plusieurs endroits, ceci alourdit inutilement le code du programme. Il pourrait donc être intéressant d'introduire des procédures additionnelles (par exemple, `ajouterDansMessage2`, `ajouterDansMessage3`) qui prennent deux ou trois arguments plutôt qu'un seul, ce qui allégerait un peu le code.

– Il serait intéressant d'utiliser un répartiteur (*dispatcher*) plus élégant que celui avec des `if` multiples. Plus précisément, une approche *table-driven* pourrait être utilisée qui permettrait d'associer à chaque nom de commande la procédure à exécuter pour cette commande. Une structure de données ayant l'allure suivante serait donc définie au début du programme et le répartiteur ne ferait que fouiller la structure (à l'aide d'une boucle) pour trouver la commande et la procédure associée, puis appeler cette procédure :

```
typedef struct {
  char *nomCommande;                        // Nom de la commande.
  void (*procedurePourCommande)(Message);  // Procedure qui traite la commande.
} CommandeEtProcedure;

static CommandeEtProcedure commandes[] = {
  {"nomDeCommande1", procedureTraitantLaCommande1},
  ...
};
```

- Le diagramme de dépendance entre modules (Figure 3.1) montre qu'il existe une dépendance «cyclique» entre deux modules. Bien que cette dépendance soit *indirecte* (ce ne sont évidemment pas les fichiers d'interface qui dépendent l'un de l'autre, mais bien les fichiers de mise en oeuvre qui dépendent d'une opération exportée par l'interface de l'autre module), il est préférable d'éviter de telles dépendances cycliques, puisqu'elles augmentent inutilement le couplage, tout en étant généralement un signe de mauvaise cohésion des modules. Il serait donc bon d'organiser les modules autrements, de façon à éviter une telle dépendance cyclique.

- La mémoire allouée dynamiquement (avec `malloc` ou autres opérations auxiliaires) n'est pas correctement libérée (avec `free`). Bien que ce ne soit pas un problème majeur — puisque chaque exécution d'une commande lance une nouvelle instance du programme, donc avec de la mémoire «fraîche» — il serait quand même plus élégant et «propre» d'effectuer correctement la gestion de la mémoire.

- La fonction `secureStrcat` n'est pas entièrement sécure, puisqu'elle n'effectue aucune vérification du résultat retourné par `calloc`...

- Des améliorations au niveau performances pourraient être apportées à la fonction `enleverEspaces`. Toutefois, il faudrait s'assurer au préalable que cette fonction est toujours utile.

Appendice A

Les spécifications d'origine

Les spécifications d'origine du système de gestion de prêts de livres étaient les suivantes :

> Il arrive fréquemment que je prête des livres à des étudiants gradués ou des collègues. J'aimerais avoir un logiciel qui me permettrait de gérer ces emprunts. Les principaux cas d'utilisation seraient les suivants :
>
> a. Indiquer le prêt d'un ou plusieurs livres (à la fois) à quelqu'un.
> b. Indiquer le retour d'un ou plusieurs livres.
> c. Déterminer si un livre a été prêté et si oui à qui.
> d. Envoyer un courriel demandant à la personne à qui j'ai prêté un livre de me le rapporter (action faisant suite à l'action précédente c).
> e. Envoyer des courriels demandant aux personnes à qui j'ai prêté des livres de me les rapporter, s'ils n'en ont plus besoin.
> f. Identifier tous les livres prêtés à une personne donnée.
>
> Les personnes (emprunteurs) seront identifiées simplement par leur prénom et nom.
>
> Les livres seront identifiés simplement par leur titre et par un ou plusieurs auteurs.
>
> Une contrainte importante est que j'aimerais avoir un logiciel utilisable par l'intermédiaire de deux (2) interfaces personne–machine distinctes :
>
> 1. Interface textuelle utilisable à partir du *shell* Unix/Linux.
> 2. Interface *Web* utilisable à partir d'un navigateur.

Appendice B

Stratégie de tests pour la vérification du bon fonctionnement et de la non régression du logiciel

Un problème typique de maintenance consiste à améliorer et restructurer un logiciel tout en s'assurant qu'il fonctionne toujours correctement.

Dans son état initial, le logiciel de gestion de prêts de livres semblait fonctionner de façon minimalement correcte. Toutefois, le logiciel était plus ou moins bien conçu et structuré, pas documenté, sans aucun test pour vérifier son bon fonctionnement, etc. Bref, il souffrait donc du syndrome du "*stinking code*". Or, on désirait améliorer ce logiciel, lui ajouter de nouvelles fonctionnalités, le rendre plus fiable et plus propre.

De façon à s'assurer que, dans un premier temps, les travaux de restructuration du code n'aient aucun impact sur le bon fonctionnement du logiciel, nous avons donc développé un ensemble de tests avec *exécution automatique*, tests qui assurent la *non régression* du logiciel lorsqu'on le modifie.

Comme il s'agit d'un programme qui gère une base de données (textuelle), et comme certaines des commandes peuvent avoir un effet sur cette base de données (par exemple, `emprunter` ou `rapporter`), il n'est donc pas suffisant de définir des données et les résultats attendus pour ces données. Soit `X` un cas de test spécifique — dans ce qui suit, chaque cas de test sera simplement identifié par un numéro unique. Chaque cas de tests doit alors être défini par les informations suivantes, spécifiées dans quatre fichiers distincts : [1]

1. `test-X.commandes` : fichier contenant une série de commandes qu'on veut exécuter et dont on veut vérifier le bon fonctionnement.

2. `test-X.avant` : fichier décrivant l'état de la base de données avant l'exécution de la série de commandes.[2]

[1] Rappelons que, par rapport à la version initiale décrite dans ce document, deux fichiers additionnels doivent maintenant être définis pour chacun des cas de tests — le script de test a donc été adapté en conséquence :

- `test-X.journal-courriels-avant`, qui décrit l'état, *avant* l'exécution des commandes, du *journal des courriels envoyés* ;
- `test-X.journal-courriels-apres`, qui décrit l'état de ce même journal *après* l'exécution des commandes.

[2] Dans ce qui suit, pour tous les cas de tests, nous partons toujours d'une base de données ne contenant aucun élément, donc d'un fichier vide.

3. `test-X.resultats` : fichier décrivant les résultats produits sur `stdout` suite à l'exécution des commandes qu'on a voulu tester.

4. `test-X.apres` : fichier décrivant l'état de la BD après l'exécution des diverses commandes.

Pour chacun des cas de test, disons `X`, nous effectuons donc les opérations suivantes :

- On exécute tout d'abord les commandes contenues dans le fichier `test-X.commandes`, en conservant les résultats émis sur `stdout` dans un fichier temporaire — l'exécution des commandes s'effectue par un appel à l'interpréteur de commandes `csh`, avec comme argument le fichier des commandes :

```
csh $repertoireTests/$casDeTest.commandes >$casDeTest.resObtenu
```

- On vérifie ensuite que les résultats produits sont bien ceux attendus et, si des différences sont rencontrées, on en prend note. Pour faire la comparaison, nous utilisons la commande `diff`, qui permet de comparer le contenu de deux fichiers, et ce ligne par ligne. Toutefois, comme les blancs qui apparaissent dans le fichier de résultat ne sont pas significatifs, nous utilisons les options `t`, `w` et `b` de `diff` pour les ignorer :

```
diff -twb $repertoireTests/$casDeTest.resultats $casDeTest.resObtenu | more
if ($status != 0) then
    # Des differences existent: on en prend note.
    set differences  = "$differences $casDeTest"
    set estDifferent = 1
    echo "$casDeTest" >> $fichErreurs
    set casDeTestOk = 0
endif
```

- Ensuite, on doit s'assurer que la base de données textuelle a été correctement modifiée. Pour rendre le script plus générique, le nom du fichier texte contenant cette base de données est reçue en argument, qu'on aura affecté préalablement à la variable `BD`. La vérification de l'état de la base de données se fait donc comme suit — on ne veut pas noter qu'un cas de test incorrect a été exécuté si une erreur pour ce cas de test avait déjà été notée, d'où l'utilisation de l'indicateur booléen `casDeTestOk` :

```
diff -twb $repertoireTests/$casDeTest.apres $BD | more
if ( ($casDeTestOk != 0) && ($status != 0) ) then
    # Des differences existent: on en prend note.
    set differences  = "$differences $casDeTest"
    set estDifferent = 1
    echo "$casDeTest" >> $fichErreurs
endif
```

Encore une fois, si des différences sont détectées, on en prend note

- Finalement, après avoir exécuté chacun des cas de tests, on indique si des différences ont été rencontrées ou non — une ligne finale sommaire indiquant le nombre total d'erreurs est aussi imprimée :

```
if ($estDifferent == 0) then
  echo ""
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++"
  echo "--- OK (aucun fichier n'a produit de difference)."
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++"
else
  echo ""
  echo "----------------------------------------------------"
  echo "*** Les fichiers suivants ont produit des differences:"
  echo "  $differences"
  echo "----------------------------------------------------"
endif

set nbErreurs = `wc -l <$fichErreurs`
echo "*** Sommaire: $nbErreurs cas de test errone(s) ***"
```

Le script 1 présente le script complet. La figure 4 présente le résultat produit par l'exécution du script 1 via l'appel à la commande "`make tests`" dans le cas où tous les tests fonctionnent correctement (et dans le cas où la compilation des divers fichiers définissant le programme a été effectuée au préalable). Pour cet exemple, nous avons produit sept (7) cas de tests différents, dont les noms sont indiqués dans la figure 5

Les différents fichiers pour le cas de test `1` (`test-1.*`) sont présentés à la figure 6.

Que se passe-t-il maintenant si on désire modifier le programme pour l'améliorer? Par exemple, dans la version initiale du logiciel, lorsqu'on désirait déterminer les emprunts effectués par quelqu'un et que cette personne n'avait emprunté aucun livre, le message suivant était produit :

```
Resultat de la recherche :


Cette persone n'a emprunter aucun livre.
```

Or, ce message contient une faute d'orthographe... qu'on aimerait corriger.

Supposons donc qu'on effectue la correction, dans le fichier `emprunts.c`, et qu'on exécute ensuite les tests. Dans ce cas, on obtiendrait les résultats indiqués à la figure 7, où l'on constate la présence de différences dans les résultats obtenus par rapport à ceux attendus. Or, ici, ce n'est pas le programme qui est erroné ... mais bien la description des résultats attendus pour l'un des cas de tests qui n'est plus correcte! Il faut donc *modifier les cas de tests* en conséquence — dans le cas présent, il faut modifier le fichier `test-4.resultats` pour tenir compte de la modification par rapport aux résultats attendus.

Des telles modifications conjointes du code et des cas de tests sont tout à fait normales et inévitables, puisque les cas de tests sont un livrable aussi important que le code lui-même, et il faut donc s'assurer de bien intégrer les deux. Or, les intégrer veut aussi dire les faire évoluer conjointement, lorsque nécessaire.

Un autre exemple de cette situation est la façon dont les arguments aux diverses commandes étaient et sont spécifiés. Dans la version initiale du programme, les arguments étaient spécifiés en les séparant par «^». Un appel pour emprunter un livre avait donc la forme suivante :

```
emprunter ^ Guy ^ guy@bidon.ca ^ Code complet ^ McConnell
```

```
#!/bin/csh

if ($#argv < 2) then
    echo "Script pour executer une serie de cas de tests pour la bibliotheque"
    echo "usage:"
    echo "  Argument 1: Nom du repertoire contenant les fichiers de tests"
    echo "  Argument 2: Nom de la base de donnees (textuelle) utilisee par le programme
    echo ""
    echo "Pour chaque cas de test X, on doit avoir les fichiers suivants:"
    echo "   test-X.commandes"
    echo "   test-X.avant"
    echo "   test-X.resultats"
    echo "   test-X.apres"
    exit -1
endif

set repertoireTests  = $1
set BD               = $2

set fichErreurs = fichAvecErreurs.txt
touch $fichErreurs

# Variables pour determiner si des differences existent et ce qu'elles sont.
set estDifferent = 0
set differences  = ""


# On execute le programme sur les divers cas de test dans le repertoire indique.
#
foreach f (`ls $repertoireTests/test*.commandes`)
    set casDeTestCommandes = $f:t
    set casDeTest = $casDeTestCommandes:r

    # On execute les commandes sur les donnees du cas de test.
    echo "**** Execution du cas $casDeTest"
    cp $repertoireTests/$casDeTest.avant $BD
    chmod +x $repertoireTests/$casDeTestCommandes
    csh $repertoireTests/$casDeTestCommandes >$casDeTest.resObtenu

    # On compare les resultats obtenus.
    set casDeTestOk = 1
    diff -twb $repertoireTests/$casDeTest.resultats $casDeTest.resObtenu | more
    if ($status != 0) then
        # Des differences existent: on en prend note.
        set differences  = "$differences $casDeTest"
        set estDifferent = 1
        echo "$casDeTest" >> $fichErreurs
        set casDeTestOk = 0
    endif
    diff -twb $repertoireTests/$casDeTest.apres $BD | more
    if ( ($casDeTestOk != 0) && ($status != 0) ) then
        # Des differences existent: on en prend note.
        set differences  = "$differences $casDeTest"
        set estDifferent = 1
        echo "$casDeTest" >> $fichErreurs
    endif
end


# On imprime les differences, si elles existent.
#
if ($estDifferent == 0) then
  echo ""
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
  echo "--- OK (aucun fichier n'a produit de difference)."
  echo "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
else
  echo ""
  echo "----------------------------------------------------------"
  echo "*** Les fichiers suivants ont produit des differences:"
  echo "  $differences"
  echo "----------------------------------------------------------"
endif

set nbErreurs = `wc -l <$fichErreurs`
echo "*** Sommaire: $nbErreurs cas de test errone(s) ***"

# On fait le menage.
rm -f *.resObtenu
rm -f $fichErreurs
```

Script 1: Le *shell script* `executer-tests` pour les différents cas de tests du programme de gestion de prêts de livre

```
% make tests
... compilation des fichiers ...
executer-tests Tests
**** Execution du cas test-1
**** Execution du cas test-2
**** Execution du cas test-3
**** Execution du cas test-4
**** Execution du cas test-5
**** Execution du cas test-6
**** Execution du cas test-7

++++++++++++++++++++++++++++++++++++++++++++++++++
--- OK (aucun fichier n'a produit de difference).
++++++++++++++++++++++++++++++++++++++++++++++++++
*** Sommaire: 0 cas de test errone(s) ***
```

Figure 4: Résultats de l'exécution du script `executer-tests` pour le programme de gestion de prêts des livres et où le programme produit les bons résultats

```
% ls Tests
Tests:
CVS/              test-3.apres      test-5.commandes*
mk-test*          test-3.avant      test-5.resultats
test-1.apres      test-3.commandes* test-6.apres
test-1.avant      test-3.resultats  test-6.avant
test-1.commandes* test-4.apres      test-6.commandes*
test-1.resultats  test-4.avant      test-6.resultats
test-2.apres      test-4.commandes* test-7.apres
test-2.avant      test-4.resultats  test-7.avant
test-2.commandes* test-5.apres      test-7.commandes*
test-2.resultats  test-5.avant      test-7.resultats
```

Figure 5: Les sept (7) cas de tests et leurs (quatre) fichiers associés

```
% cat Tests/test-1.commandes
emprunter ^ Guy ^ guy@bidon.ca ^ Code complet ^ McConnell
emprunts Guy
emprunteur Code complet
-----------------------------------------------------------------
% cat Tests/test-1.avant
-----------------------------------------------------------------
% cat Tests/test-1.apres
Code complet | McConnell | Guy | guy@bidon.ca
-----------------------------------------------------------------
cat Tests/test-1.resultats

Emprunter un livre:
------------------

Livre ajoute a la liste des livres pretes :

    Titre      : "Code complet"
    Auteur     : McConnell
    Emprunteur : Guy
    Courriel   : guy@bidon.ca



Resultat de la recherche :

Titre      : Code complet
Emprunteur : Guy
Courriel   : guy@bidon.ca


Resultat de la recherche :

Titre      : "Code complet"
Emprunteur : Guy
Courriel   : guy@bidon.ca
```

Figure 6: Les différents fichiers pour le cas de test 1 (version initiale du logiciel)

```
% make tests
executer-tests Tests livredb.txt
**** Execution du cas test-1
**** Execution du cas test-2
**** Execution du cas test-3
**** Execution du cas test-4
73c73
< Cette persone n'a emprunter aucun livre.
---
> Cette persone n'a emprunte aucun livre.
**** Execution du cas test-5
**** Execution du cas test-6
**** Execution du cas test-7

-------------------------------------------------------
*** Les fichiers suivants ont produit des differences:
   test-4
-------------------------------------------------------
*** Sommaire: 1 cas de test errone(s) ***
```

Figure 7: Résultat de l'exécution des divers cas de tests suite à la modification du fichier `emprunts.c` pour corriger une erreur d'orthographe dans le message d'erreur

Dans la nouvelle version du logiciel, les arguments sont simplement séparés par des espaces, les arguments formés de plusieurs mots tels les titres de livres étant simplement mis entre guillemets. De plus, il n'existe qu'un unique programme, `biblio`, lequel reçoit comme premier argument le nom de la commande à exécuter. Pour tenir compte de cette nouvelle façon d'exécuter les diverses commandes, il faut donc modifier les fichiers des cas de tests contenant des commandes. Le fichier contenant les commandes pour le cas de test 1 (`Tests/test-1.commandes`) devient donc le suivant :

```
biblio emprunter Guy guy@bidon.ca "Code complet" McConnell
biblio emprunts Guy
biblio emprunteur "Code complet"
```

Appendice C

Exemple de résultat produit avec DOC++ (version LaTeX) pour les opérations du module `gererArguments`

```
//@{

/** Lit et mémorise les arguments de la ligne de commande.

   @param argc Le nombre d'arguments de la ligne de commande
   @param argv Les arguments de la ligne de commande qu'il faut traiter
*/
void lireArguments( int argc, char **argv );


/** Retourne le nombre d'arguments memorisés.

   @precondition La procédure lireArguments() a été appelée
   @return nombreArguments() Le nombre d'arguments memorisés
*/
int nombreArguments();


/** Retourne le ième argument.

   @precondition La procédure lireArguments() a été appelée et i < nombreArguments()
   @param i Position de l'argument cherché
   @return obtenirIemeArgument() Le ième argument spécifié sur la
        ligne de commande
*/
char *obtenirIemeArgument( int i );

//@}
```

Figure 8: Extraits du fichier `gererArguments.h`, fichier documenté avec `DOC++`

La figure 8 présente des extraits du contenu du fichier `gererArguments.h`, lequel fichier, comme tous les autres fichiers `.h`, a été documenté avec les commentaires spéciaux de l'outil `DOC++`. Le résultat pour chacune des opérations est présenté, sous forme LaTeX, dans les pages qui suivent. Ce

résultat a été obtenu à l'aide de la commande suivante :

```
doc++ --tex --output arguments.tex gererArguments.h
Ajout dans le fichier arguments.tex de la ligne <<usepackage[T1]{fontenc}>>,
     pour le traitement correct des accents.
latex arguments.tex
latex arguments.tex
```

1

void **lireArguments** (int argc, char** argv)

Lit et mémorise les arguments de la ligne de commande.

Lit et mémorise les arguments de la ligne de commande.

Parameters:

`argc`	Le nombre d'arguments de la ligne de commande
`argv`	Les argument de la ligne de commande qu'il faut traiter

2

int **nombreArguments** ()

Retourne le nombre d'arguments mémorisés.

Retourne le nombre d'arguments mémorisés.

Return Value: `nombreArguments()` Le nombre d'arguments mémorisés
Preconditions: `La` procédure lireArguments() a été appelée

3

char* **obtenirIemeArgument** (int i)

Retourne le ième argument.

Retourne le ième argument.

Return Value:	`obtenirIemeArgument()`	Le ième argument spécifié sur laligne de commande
Parameters:	`i`	Position de l'argument cherché
Preconditions:	`La`	procédure lireArguments() a été appelée et i < nombreArguments()