POLYTECH PARIS-UPMC

UPMC SORBONNE UNIVERSITÉS

EI-SE 4

COMPTE RENDU – PROJET C++

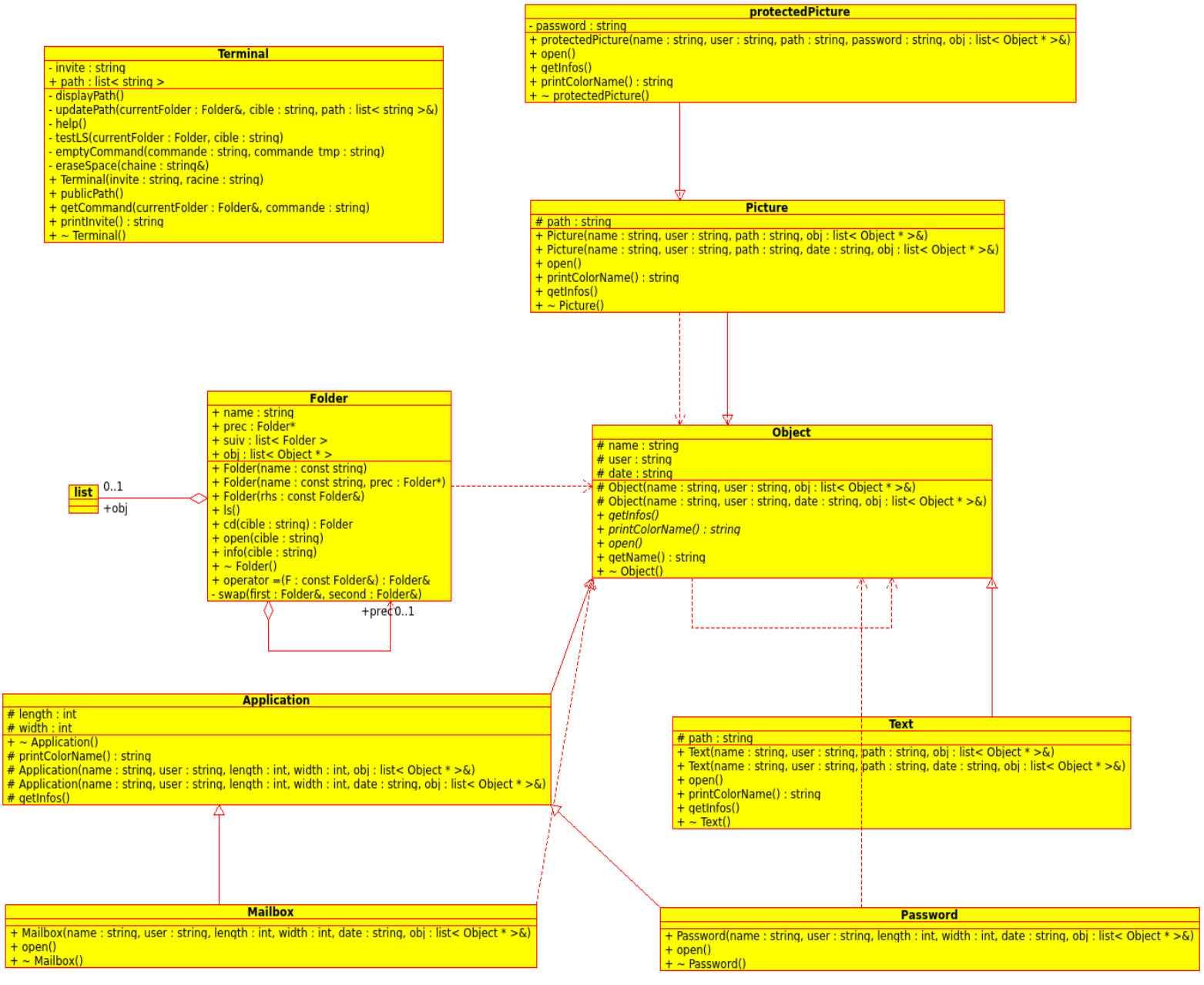
”Elections piège à cons”

BIASUTTI Valentin – SÉBERT Antoine

Introduction

Fortement inspirés par la série Mr. Robot, nous avons souhaité créer un jeu de type « Found Phone Game » dans lequel l'utilisateur est amené à explorer un téléphone perdu afin de reconstituer l'histoire de son propriétaire. Nous avons ainsi cherché à recréer l'arborescence de fichiers et de dossiers d'un certain Tronald Dump (dont l'existence n'est aucunement inspirée d'une quelconque personnalité réelle) que l'utilisateur serait absolument libre d'explorer (via le terminal) afin de récolter des indices et de débloquer un fichier protégé par un mot de passe. Nous justifions ainsi le thème « Élections, piège à cons » par le fait que l'utilisateur est totalement libre de ses choix sans que cela influence le dénouement du jeu.

Diagramme de classes



Réalisation

La réalisation de ce projet a donc tout d'abord nécessité la création d'une arborescence de dossiers. Pour ce faire, nous avons donc commencé par définir une classe *Folder*. A la manière d'une liste chaînée, chaque dossier ainsi construit est doté d'un attribut père de type pointeur sur *Folder*. Le dossier initial (intitulé Bureau) sera quant à lui affilié à un pointeur NULL. De plus, chaque dossier sera également doté de 2 conteneurs :

- une liste de *Folder*, qui représente en fait tous les sous dossier présent dans chaque répertoire ;
- une liste de pointeurs sur *Object* (la classe *Object* étant abstraite, nous n'avions d'autre choix que de créer une liste de pointeurs), qui représente tous les autres éléments présents dans un dossier (images et applications).

Dans un second temps, il a fallu implémenter une façon de permettre à l'utilisateur d'ouvrir les objets qui contiendraient les différents indices à récupérer, telles que les images ou les applications. Nous avons donc décrit une classe abstraite *Object*. Celle-ci est dotée d'une méthode open() dont le comportement sera spécifique à chaque instance des différentes classes filles.

Dans un premier temps, une classe *Picture* servira à afficher les images fournies à l'aide d'un attribut de type string qui indiquera le chemin vers l'image en question (située dans le dossier src). Celle-ci sera également mère d'une autre classe : *ProtectedPicture*, qui correspond en fait à l'image que l'utilisateur doit trouver. Sa fonction open() diffère donc de la classe *Picture* puisqu'une première fenêtre demande un mot de passe à l'utilisateur.

Dans un second temps, un autre classe *Application* héritera de la classe *Object*. Chacun des différents programmes que l'utilisateur va pouvoir lancer sera alors défini avec son comportement dans les différentes classes filles *(PaswordGenerator* et *MailBox*).

Finalement, c'est la classe *Terminal* qui sera en charge d'assurer le lien entre tous les objets précédemment décris et d'interpréter toutes les commandes que l'utilisateur entrera.

Nous avons réellement trouvé intéressant le fait de simuler le comportement d'un système de fichiers. Toutefois, certaines fonctionnalités (telles que l'utilisation des flêches ou de la tabulation) ne sont pas implémentées, ce qui peut s'avérer perturbant pour les initiés.

Dans l'ensemble, nous avons essayé de faire en sorte que l'utilisateur puisse trouver le mot de passe relativement facilement, sans pour autant que cela soit trivial : nous voulions avant tout développer un jeu de réflexion sur fond de hacking. Une commande help a également été implémentée afin d'afficher une fenêtre (à la façon d'une page man) expliquant le scénario et les différentes commandes à disposition.

Enfin, beaucoup de fonctions et de méthodes ont été implémentées dans chacune des classes et dans le fichier main.hh afin de fournir un affichage qui soit attractif et le plus intuitif possible.

Ainsi tous les différents dossiers et fichiers sont créés et linkés dans le début du main et une boucle infinie va simplement traiter les lignes de commandes entrées par l'utilisateur. La commande exit permet toutefois de sortir convenablement de la boucle et ainsi de faire appel aux différents destructeurs.

La classe *protectedPicture* contient le dénouement du jeu, il est donc déconseillé de l'étudier si vous souhaitez découvrir le contenu du fichier.

Pistes de résolution (spoiler alert)

L'application PasswordGenerator présente 4 champs qu'il faut remplir afin de générer un mot de passe. C'est ce mot de passe qui empêche l'ouverture de l'image protégée. L'utilisateur va ainsi devoir explorer les fichiers de Tronald Dump dans le but de récolter ces 4 indices :

- Le premier est le nom de sa conjointe (qui n'est évidemment pas Mélania). Un dossier « Babe », situé dans « Documents » regroupe les fichiers de la femme de Tronald. C'est grâce à la fonction info() que l'utilisateur va pouvoir récupérer le nom de sa femme ;
- Le deuxième concerne la date d'anniversaire de mariage de Tronald. C'est dans les mails que se trouvera la réponse ;
- Le troisième concerne son année de naissance. Dans un document texte, Tronald relate une anecdote de sa jeunesse en faisant un lien avec un évènement historique ; permettant ainsi d'en déduire l'indice en question ;
- Enfin, le dernier indice concerne la dernière destination de vacances de Tronald. Sans grande surprise, c'est dans le dossier « Holidays » que se trouvera la réponse. Il va alors falloir découvrir la date de création de chaque image et la comparer à la date de téléchargement de PasswordGenerator.

Le mot de passe est finalement : **GreAUTronaldia_Gce221950**

Installation et compilation

3 bibliothèques sont nécessaires pour la compilation et l'exécution de ce jeu :

- La première est *figlet*, qui permet simplement d'afficher du texte de manière esthétique dans le terminal, elle est simplement utilisée lors de l'introduction :

sudo apt-get install figlet

- La deuxième est ncurse.h qui est utilisée lors de l'affichage de la page d'aide (commande help) :

sudo apt-get install libncurses5-dev libncursesw5-dev

- La dernière permet l'affichage de toutes les fenêtres graphiques et la lecture des fichiers sonores : c'est la bibliothèque Gtkmm :

sudo apt-get install libbtkmm-3.0-dev

Gtkmm et fuites mémoires

Après de nombreux essais, nous avons remarqué que la bibliothèque Gtkmm génère énormément de fuites mémoires (celles-ci peuvent aller jusqu'à 4Mo pour un simple printf(« Hello World ») qui ne nécessite même pas l'utilisation de la lib ; son inclusion seule est déjà source de fuites mémoires). Il semblerait en fait que la bibliothèque génère des objets statiques lors de son initialisation, qui ne seront pas désalloués à la fin du programme.

Il semblerait que cette bibliothèque ne puisse tout simplement pas être utilisée sans générer de fuites mémoire. Une utilisation « conventionnelle » de la bibliothèque consisterait à produire l'affichage d'une fenêtre dont la fermeture causerait la fin du programme (réduisant ainsi l'impact des fuites mémoires). Or, nous souhaitons ici faire en sorte que l'utilisateur puisse choisir d'ouvrir une fenêtre à n'importe quel moment, nous avons donc cherché à résoudre, ou du moins limiter ce problème, en utilisant des fichiers de suppressions.

Le répertoire utile à la création de ces fichiers a directement été importé depuis GitHub et est présent dans le commit final. A l'aide du terminal, il faut se rendre dans le dossier GNOME.supp et exécuter la commande make.

Après s'être replacé dans le dossier contenant les fichiers .cc et .hh, on exécute alors le programme de la façon suivante :

./main --suppression=/GNOME.supp/build/gtk3.supp

Toutefois, il semblerait que cette méthode limite les fuites de seulement quelques centaines (voire quelques milliers) d'octets, ce qui reste très largement négligeable devant les centaines de milliers perdus mais il semblerait qu'elle reste la seule solution connue à ce jour.

Finalement, on notera qu'une simple exécution valgrind (sans autre option concernant les fuites telle que *–leak-check = full*) ne révèle toutefois aucune erreur.