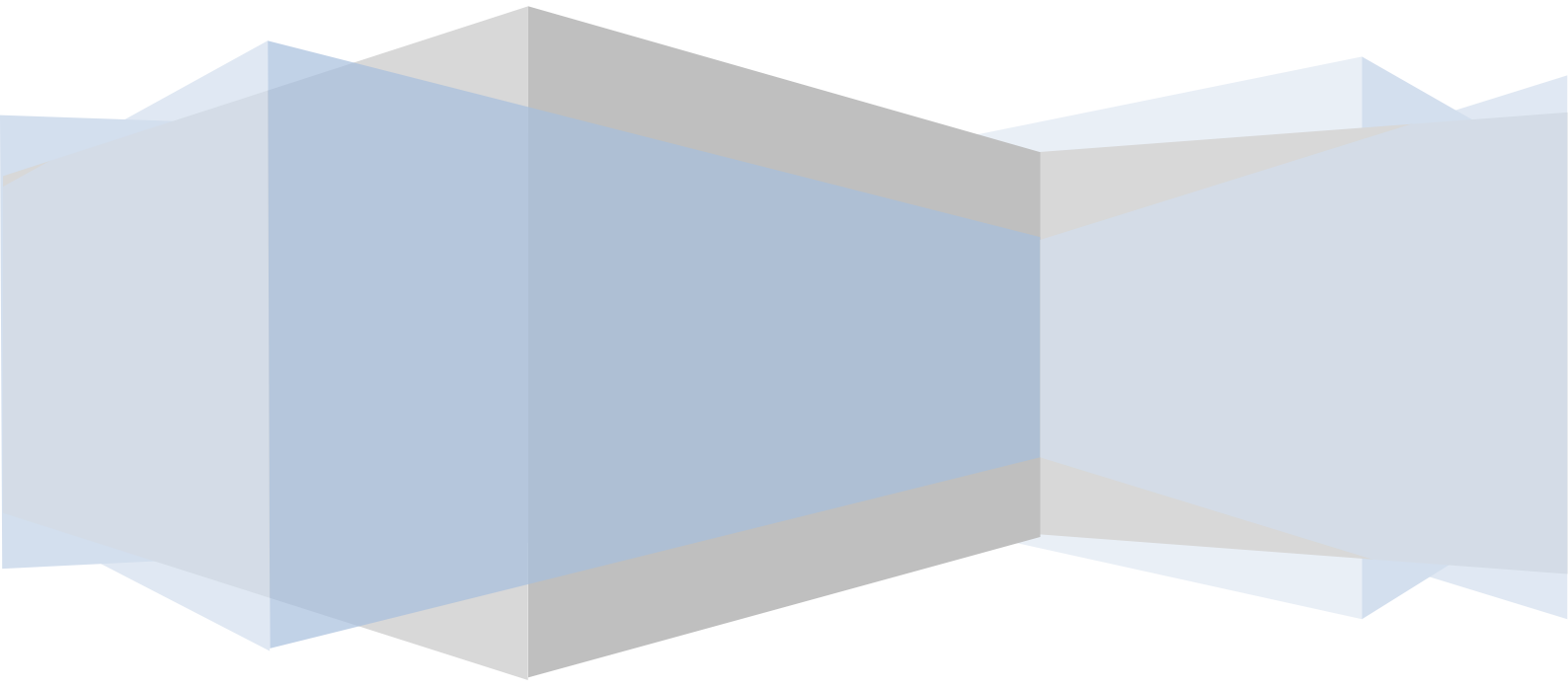
BTS SN-IR 1 Année

Lycée Diderot, Paris

TP Méthodes

Programmation C++ sous Linux

Gilles Dalles



SOMMAIRE

1. Introduction

L'objectif ici est de se familiariser avec les méthodes et les notions de programmation orientée objet.

2. Fonctions et Methodes

2.1. fonction

Une fonction permet de regrouper une séquence d'instructions et d'automatiser certaines tâches à réaliser. Le but est de la faire la plus générique possible de manière à pouvoir l'utiliser dans plusieurs programmes différents. Par exemple, nous avons utilisé la fonction rand() de la bibliothèque standard C dans les programmes sur l'euromillions et le sudoku.

Plus simplement, nous utilisons dans tous nos programmes la fonction main().

Une fonction peut prendre des données en entrée (pour les traiter) et fournir un résultat en sortie. La fonction time() par exemple prend un paramètre et retourne le nombre de secondes écoulé depuis le 01/01/1970.

2.2. Methode

Une méthode, c'est une fonction. Mais c'est une fonction en orienté objet.

Le concept objet, comme beaucoup de choses en informatique, est directement inspiré de la vie courante. Par exemple, dans votre vie de tous les jours, la fonction téléphoner() n'est pas accessible sans l'objet auquel elle est rattachée: un téléphone. Sans téléphone, pas d'appel possible.

En programmation orientée objet, le principe est le même: une fonction se doit d'être attachée à un objet. L'appel (l'utilisation) de la méthode se fait ainsi:

```
objet.methode(); //ou objet->methode();
```

3. Classes et Objets

3.1. Objet

Un objet se caractérise par 3 choses:

- son état
- son comportement
- son identité

L'état est défini par les valeurs des attributs de l'objet à un instant t. Par exemple, pour un téléphone, certains attributs sont variables dans le temps comme allumé ou éteint, d'autres sont invariants comme le modèle de téléphone.

Le comportement est défini par les méthodes de l'objet: en résumé, les méthodes définissent à quoi sert l'objet.

L'identité est définie à la déclaration de l'objet (instanciation) par le nom choisi, tout simplement.

3.2. Classe

Pour créer un objet, il faut d'abord établir son plan de fabrication. En développement logiciel, on appelle ces plans des classes. Avec une classe, on définit donc un nouveau type de "variable": ces "variables" sont structurées par opposition aux variables de types simples comme les entiers ou les réels. On les appelle dès lors objet ou instance de classe.

Dans une classe, on trouvera donc les déclarations des attributs (des objets ou des variables simples) et des méthodes (afin de définir le comportement des futurs objets). Une méthode peut utiliser les attributs et/ou les autres méthodes de la classe.

4. QT

QT est à l'origine une bibliothèque graphique développé par une entreprise norvégienne: **Trolltech**. Cela a depuis été revendu à **Nokia**. C'est sous la tutelle de Nokia que certains outils sont apparus, notamment **QTCreator**, l'**IDE** (Integrated development environment) que nous allons désormais utiliser ensemble. Depuis 2011, c'est la société finlandaise **Digia** qui a récupéré la fameuse bibliothèque.

4.1. Pourquoi QT?

QT possède un avantage certain: il s'agit d'une bibliothèque complète, même les types de bases tels que char ou int ont été redéfinis. L'objectif a toujours été de rendre l'utilisation de la bibliothèque libre de l'environnement. Cela signifie que quel que soit l'OS sous lequel une application **QT** est développée, le code source pourra être récupéré et compilé sur des plates formes différentes. Ce qui fait la force de **QT** est ce que l'on appelle sa portabilité: un programme développé sous **Linux** pourra être compilé et exécuté sous **Windows** ou sous **MacOS** sans changer une ligne de code.

On retrouve ici l'un des gros avantages du langage **Java**, la machine virtuelle en moins.

Autre avantage, la bibliothèque **QT** n'est pas cantonnée a être utilisée uniquement avec **QTCreator**: elle peut être associée à **Visual Studio** ou encore **Netbeans**.

Nous allons utiliser désormais **QTCreator** pour une simple question de simplicité.

4.2. QTCreator

Il s'agit donc d'un IDE, comprendre un environnement de développement intégré. Ce qui veut dire que si vous vous y prenez bien, vous n'aurez plus à écrire de fastidieuses lignes de commandes pour compiler vos sources, que vous aurez à disposition un débogueur efficace, un manuel d'aide ultra complet, des exemples d'applications, des facilités pour créer des classes, de nouveaux fichiers etc…

Vous en utilisez déjà un sous Windows avec Visual Studio.

4.2.1. Creation d'un projet

La création d'un projet est relativement simple: au démarrage de **QTCreator**, vous pouvez soit cliquer sur le bouton "*Créer un projet*", soit aller dans le menu "*Fichier->Nouveau Fichier ou Projet*". La fenêtre ci-dessous apparait:

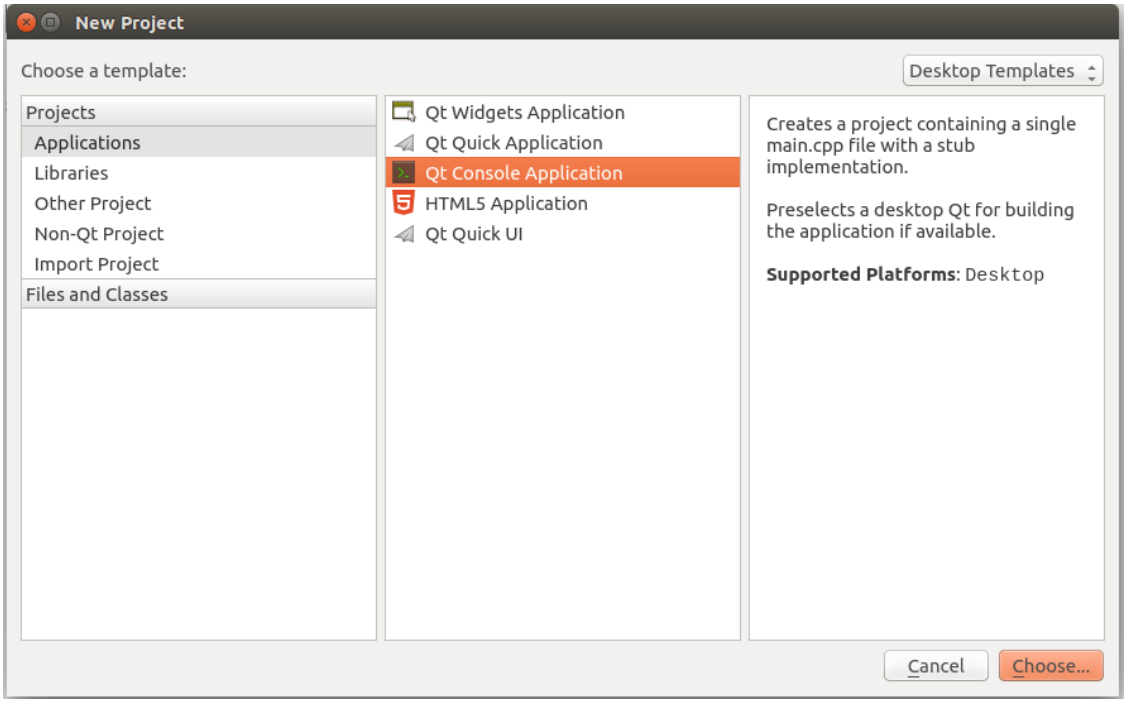


Figure 1: Creation de projet

Vous allez choisir un modèle "*Applications*" puis "*QT console Application*". On ne passe pas encore le cap des interfaces graphiques aujourd'hui. Ce type de projet permet d'utiliser la bibliothèque **QT** sans pour autant avoir forcément besoin d'une **GUI** (Graphic User Interface). Une fois le choix fait, vous devez avoir cette fenêtre:

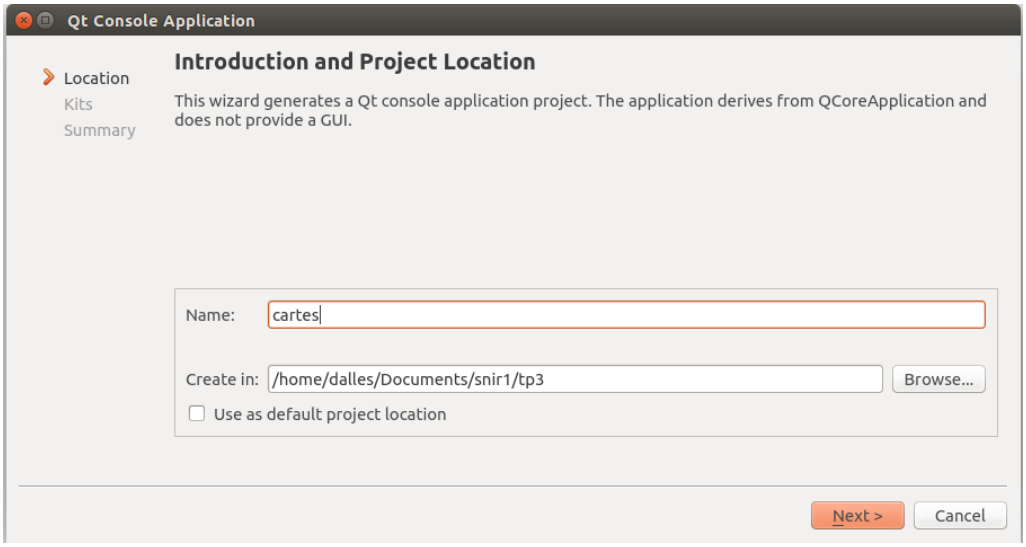


Figure 2: Choix du nom du projet et emplacement

Vous placez le projet où vous vous voulez dans votre répertoire personnel, il devra cependant se nommer **cartes**. Vous devriez maintenant arriver sur cette fenêtre:

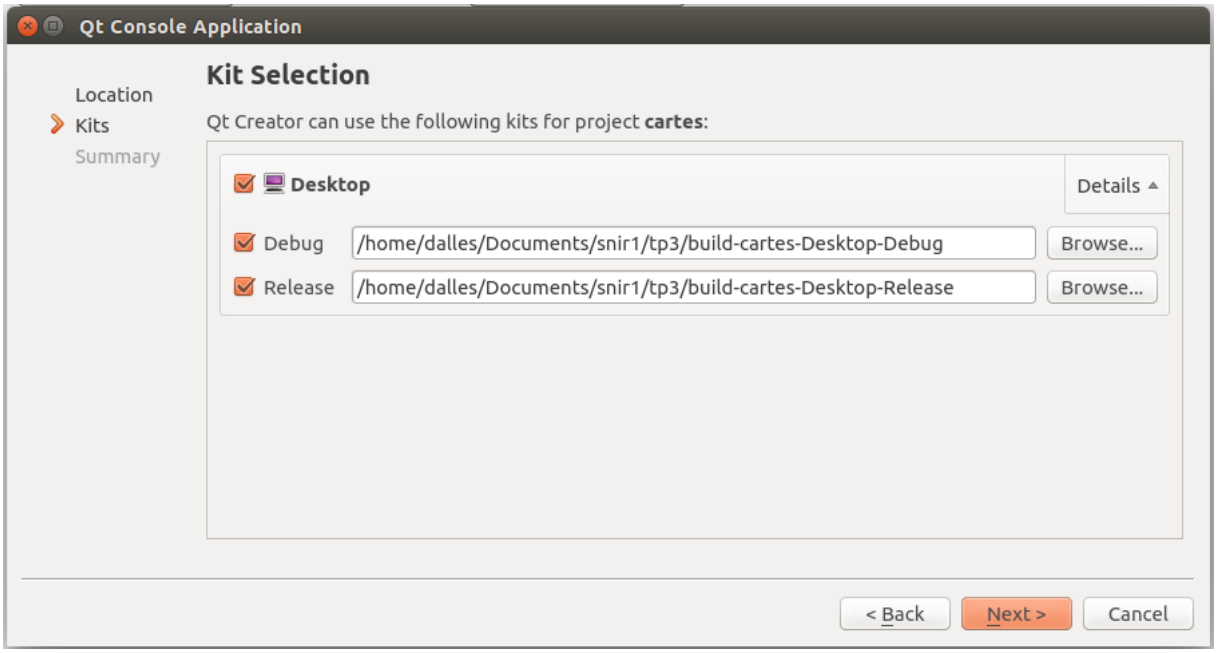


FIGURE 3: INSTALLATION DE LA CIBLE

Ne changez rien aux options et cliquez sur suivant. Vous devez arriver sur la fenêtre de gestion de projet: il est intéressant de se pencher sur cette question dès lors qu'un projet devient ambitieux et qu'il peut être amené à évoluer dans le temps. Pour nous, il n'y a aucune raison de se compliquer encore plus les choses, on restera en l'état:

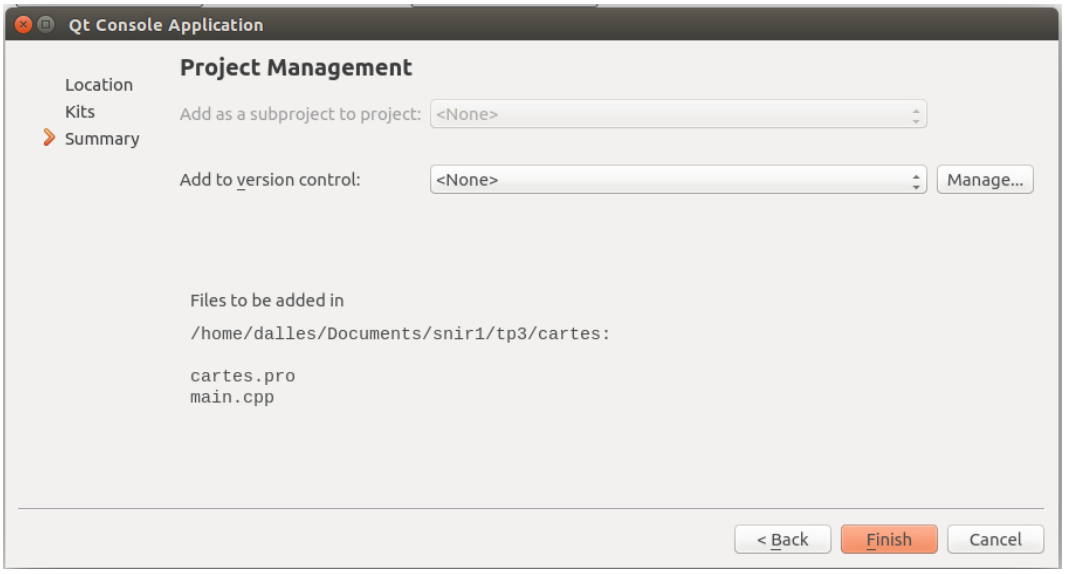


FIGURE 4: GESTION DU PROJET

Vérifiez seulement que vous avez bien 2 fichiers dans votre projet une fois le bouton *Terminer* cliqué: un source **main.cpp** et un fichier de projet **cartes.pro**.

4.2.2. Execution du projet

Voila, votre projet est créé. Vous pouvez d'ores et déjà le compiler et l'exécuter en cliquant sur le triangle vert en bas à gauche de l'écran ou par le menu *Compiler*.

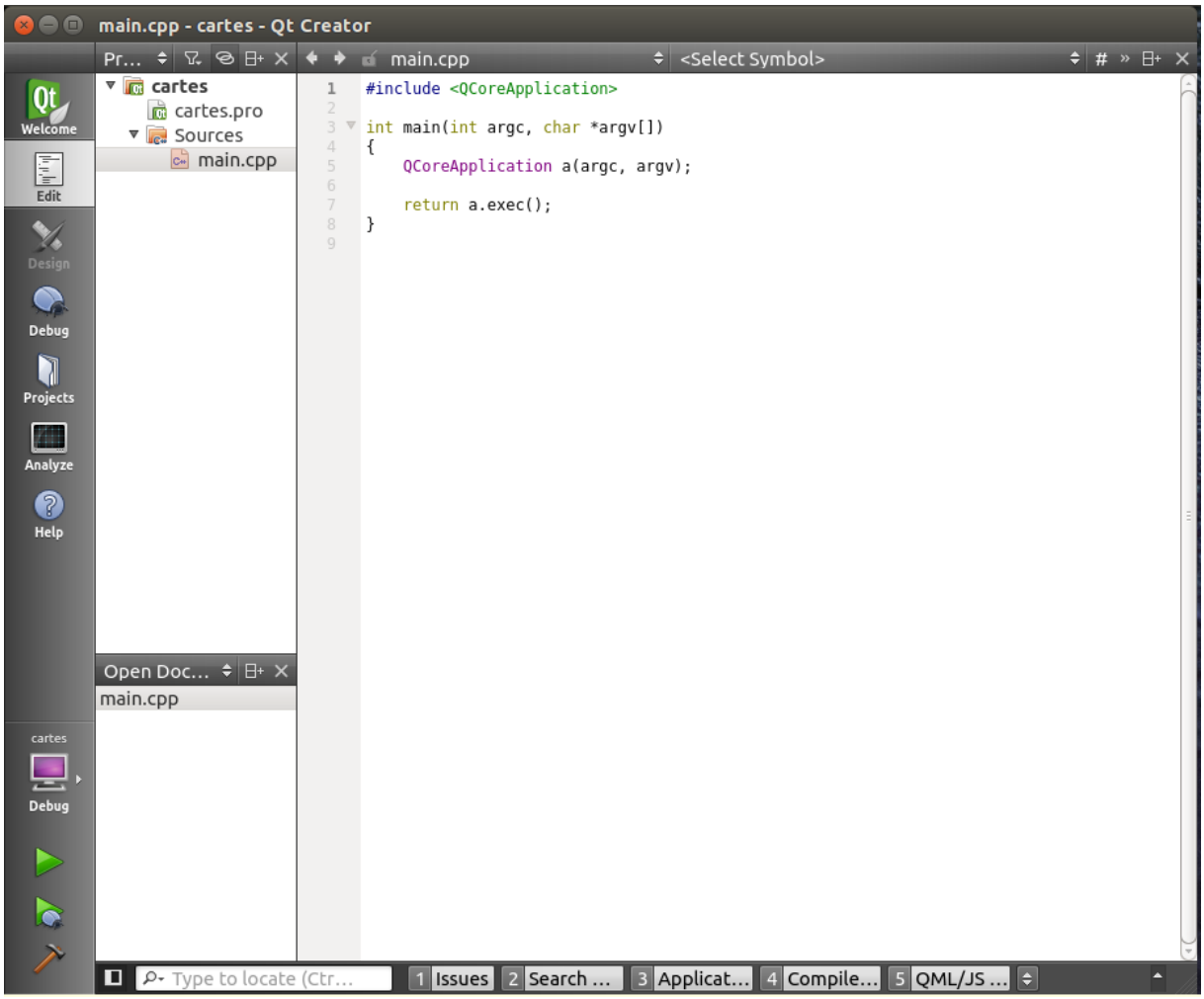


Figure 5: Mode projets

Mais cela ne suffit pas: **QTCreator** est capricieux et refuse de fonctionner correctement, sous **Ubuntu** du moins, en mode débogueur. Vous aurez systématiquement une erreur **ptrace: Operation not permitted.**

Il va vous falloir corriger un paramètre de configuration d'Ubuntu pour pouvoir utiliser le débogueur de QTCreator.

Le fichier **10-ptrace.conf** se trouve dans le répertoire **/etc/sysctl.d/**. Il faut modifier le fichier et remplacer la valeur 1 par 0 de **kernel.yama.ptrace_scope**.

Ce bug est connu de longue date. Je vous ai épargné la peine de chercher comment le résoudre (ou plutôt passer outre) mais il existe nombre de forums pouvant vous aider dans de telles situations. Ceux qui dans le futur voudront utiliser une base de données **MySQL** sous **Windows** avec leur application **QT** développée avec **QTCreator** doivent se préparer mentalement à souffrir (problème ne se posant pas sous **Linux**).

5. JEU DE CARTES

On va réaliser une application permettant de gérer des jeux de cartes de 32 ou 52 cartes. On pourra mélanger et distribuer les cartes à un nombre de joueurs indéterminé. Par contre, aucune règle de jeu (poker, rami, bataille, etc..) ne sera établie. En gros, on ne joue pas encore.

On va donc réaliser une application orientée objet. Cela implique un fichier source (.cpp) et un fichier d'entête (.h) par classe utilisée.

Je vous rassure tout de suite, je vous fournis les 2 classes que nous allons devoir utiliser. Il faudra compléter les définitions des méthodes de l'une d'elle.

5.1. DIAGRAMME DE CLASSES

Un objet est en général composé d'autres objets. Par exemple, un téléphone a au moins un haut parleur et un micro qui le compose.

Ici, la philosophie est simple: un jeu de cartes est composé de cartes. Donc, nous allons utiliser une classe **c_carte** et une classe **c_jeuDeCartes**.

Un diagramme de classes UML (Unified Modeling Language) représente les associations entre les classes. Cela permet de savoir quelle classe contient les instances de quelle autre.

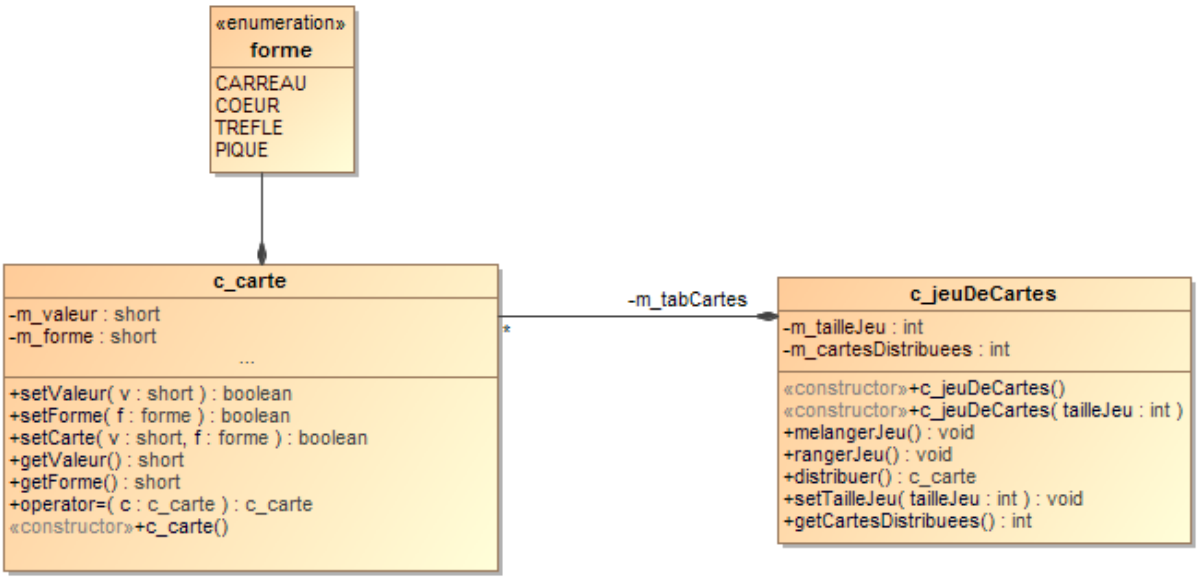


FIGURE 6: DIAGRAMME DE CLASSES

Dans ce diagramme, plusieurs informations:

- Une énumération forme est définie et appartient à la classe **c_carte**.
- La classe **c_carte** a plusieurs instances composant la classe **c_jeuDeCartes** regroupées sous le nom **m_tabCartes**

Ces associations sont déjà faites dans les sources fournis.

5.2. AJOUT DES SOURCES

Vous trouverez les sources dans le répertoire partagé de seacloud: Année1 / TP / Sources / TP3 /

Vous devez trouver 4 fichiers, 2 d'entêtes, 2 sources. Vous devez les copier dans votre répertoire de projet. Pour moi, dans mon exemple, ce serait ici (cf: Figure 2: Choix du nom du projet et emplacementFigure 2): /home/dalles/Documents/snir1/tp3/cartes.

Une fois fait, il faut les ajouter au projet QT. Rien de plus simple, il suffit dans QTCreator de faire un clic droit sur le nom du projet que vous venez de créer et de choisir dans le menu contextuel *ajouter fichiers existants*.

Les fichiers sources doivent apparaitre désormais dans votre projet.

5.3. METHODES A COMPLETER

La classe c_jeuDeCartes à 3 méthodes non définies:

- rangerJeu()
- melangerJeu()
- distribuer()

Il va vous falloir les définir. Je vais vous donner quelques indications.

5.3.1. RANGERJEU()

rangerJeu() est une méthode qui ne range pas vraiment le jeu. En fait, on remplit le tableau **m_tabCartes** avec les cartes dans l'ordre en fonction bien sur de la taille du jeu, 32 ou 52 cartes. On ne s'occupe donc pas des valeurs déjà stockées dans le tableau.

5.3.2. MELANGERJEU()

melangerJeu() permet de mélanger les cartes du jeu. En gros, on prend le tableau **m_tabCartes** et on échange les valeurs contenues dans ses cases.

5.3.3. DISTRIBUER()

distribuer() est une méthode retournant une seule carte du tableau **m_tabCartes**. Il est évident qu'à chaque appel de cette méthode, on progresse dans le tableau afin de ne pas toujours donner la première carte.

5.4. DIAGRAMMES DE SEQUENCE

Un diagramme de séquence permet de représenter les interactions entre les différents objets d'un projet.

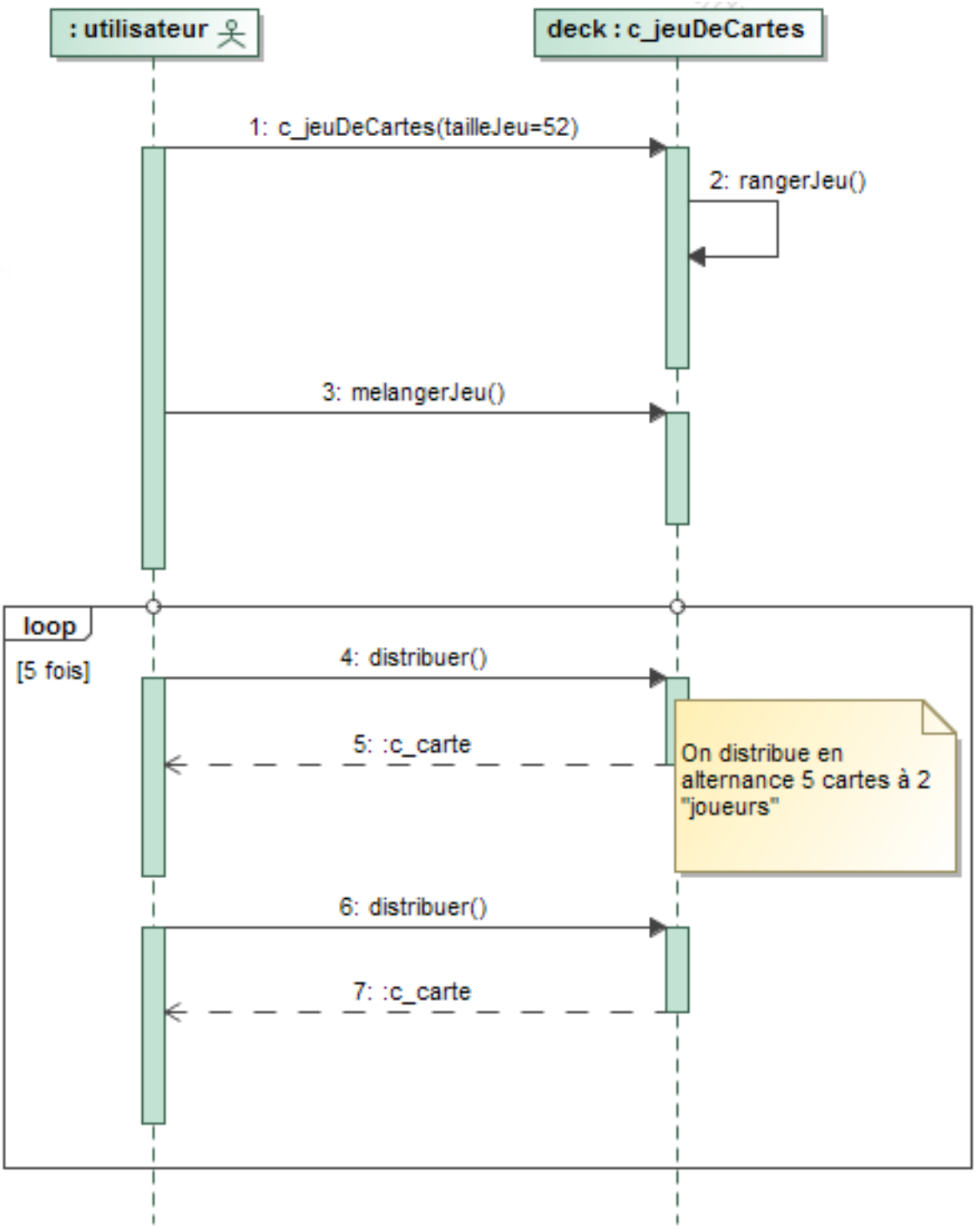


FIGURE 7: DIAGRAMME DE SÉQUENCE (MAIN)

Dans ce diagramme, on voit les interactions entre l'utilisateur du programme via la fonction main() et l'objet issu de la classe **c_jeuDeCartes**.

Il s'agit d'une séquence, ce qui veut dire que les actions décrites sont faites les unes après les autres. Le diagramme ci-dessus décrit les actions réalisées tel que je le souhaite. Le diagramme ci-dessous décrit les actions à réaliser si on appelle la méthode **rangerJeu()** dans la fonction main. Il décrit donc les interactions de l'instance de **c_jeuDeCartes** avec celles de **c_carte**. Cela devrait être utile pour définir cette méthode.

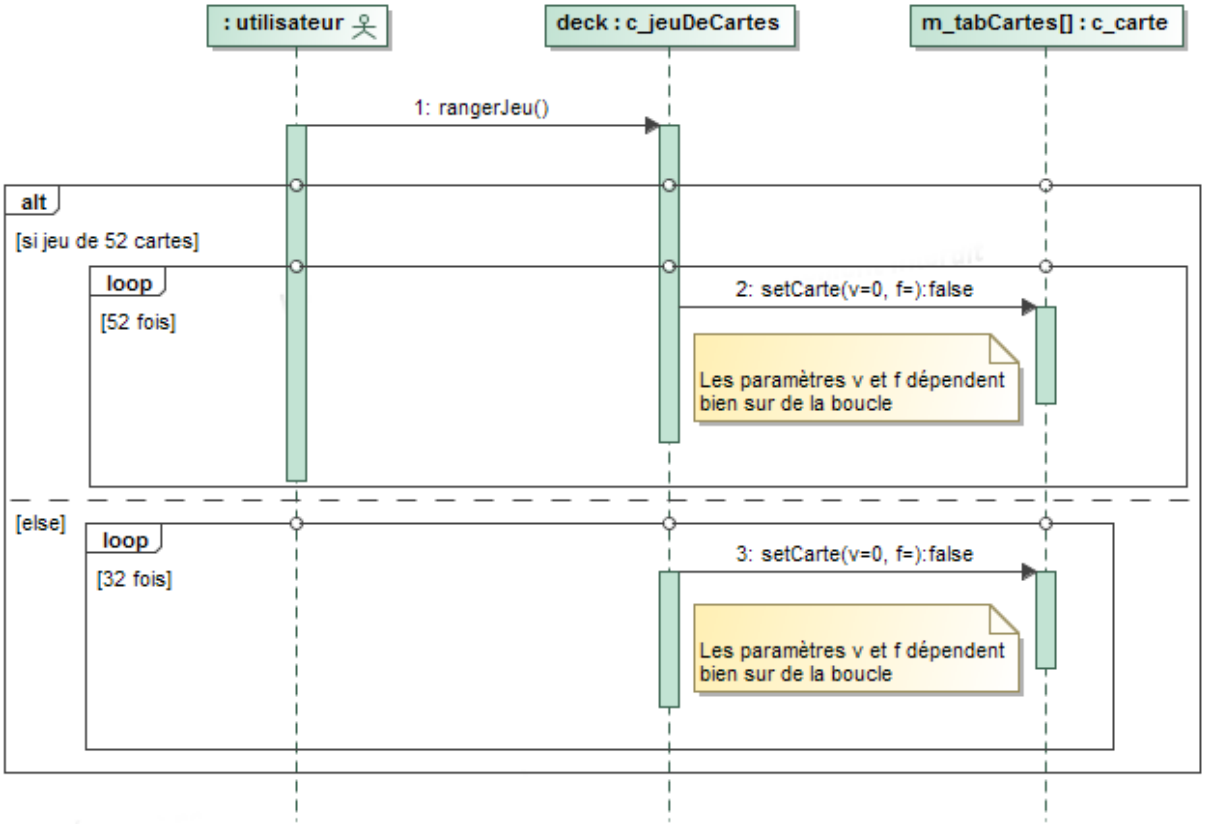


FIGURE 8: DIAGRAMME DE SÉQUENCE (RANGERJEU)