# *RAPPORT DE PROJET :*

# *DÉMINEUR (C++ / Qt)*

**Réalisé Par :** SOUFI Soufiane

**Université de Bourgogne, IUT Le Creusot**

**Génie Electrique et Informatique Industrielle**

**Date : 07 janvier 2026**

**Projet : Développement d'un jeu avec Interface Graphique (Qt)**

# I. Introduction & Objectifs

Ce projet avait pour ambition principale la conception intégrale d'un jeu de Démineur fonctionnel en exploitant la puissance du langage C++ couplée aux bibliothèques graphiques du framework Qt. Au-delà de la simple implémentation des règles ludiques standard, ce travail pratique constituait un exercice de synthèse technique visant à maîtriser plusieurs concepts clés de l'ingénierie logicielle. Il s'agissait d'apprivoiser la programmation événementielle propre aux interfaces graphiques et d'intégrer des ressources visuelles externes via la classe QPixmap, mais surtout de conceptualiser une logique mathématique matricielle capable de représenter fidèlement l'état d'un jeu en temps réel.

Le défi majeur de ce développement ne résidait pas tant dans l'écriture du code lui-même que dans la compréhension de l'architecture sous-jacente nécessaire à son fonctionnement. Il a fallu assimiler le fait que l'interface visible par l'utilisateur n'est qu'une projection graphique d'une mémoire informatique abstraite. Cette dualité entre le modèle de données en mémoire et la vue affichée à l'écran a guidé l'ensemble de ma démarche, depuis la phase de réflexion initiale jusqu'aux phases de débogage et d'optimisation.

# II. Architecture Technique

Pour répondre à ces exigences de robustesse et de maintenabilité, j'ai structuré le programme autour d'une séparation claire entre les objets logiques et le gestionnaire de l'interface. La brique fondamentale de cette architecture est la classe Tile, que j'ai conçue pour agir comme une unité de mémoire autonome. Chaque instance de Tile ne se préoccupe pas de son affichage, mais conserve uniquement son état interne. Elle sait si elle piège une bombe grâce au booléen isBomb, si le joueur a posé un drapeau via isFlagged, et connaît son statut de

visibilité grâce à l'entier tileState. Cette encapsulation permet de manipuler des concepts abstraits sans se soucier immédiatement de la complexité graphique.

### a. La Classe Tile

À une échelle plus large, le plateau de jeu est modélisé par une matrice à deux dimensions, déclarée sous la forme QVector<QVector<Tile>> M. C'est véritablement le cerveau du programme. J'ai compris que pour rendre le jeu fonctionnel, il ne fallait pas manipuler des pixels ou des boutons graphiques, mais opérer directement sur cette grille mathématique. Chaque case de la matrice correspond à une coordonnée précise définie par les indices de lignes et de colonnes, et c'est sur ces indices que s'appliquent tous les algorithmes de vérification et de modification. L'interface graphique, quant à elle, ne fait que consulter cette matrice pour savoir quelle image afficher à quel endroit, garantissant ainsi une cohérence parfaite entre la logique du jeu et le rendu visuel.

## III. Chronologie et Logique de Développement

Le processus de création ne s'est pas fait de manière linéaire, mais a suivi une approche itérative découpée en trois phases distinctes, alternant entre la conception théorique, l'implémentation du code et la résolution de problèmes techniques complexes.

### Phase 1 : Analyse Préliminaire et Conception

Avant d'écrire la moindre ligne de code graphique, j'ai consacré une première phase à l'analyse des besoins. C'est durant cette étape critique que j'ai défini les structures de données nécessaires et les algorithmes fondamentaux. J'ai par exemple anticipé le besoin d'une fonction capable d'inverser l'état d'un drapeau, ou encore la logique mathématique pour calculer le nombre de bombes autour d'une case donnée. Cette préparation en amont a permis de créer les fichiers

d'en-tête avec une vision claire des variables requises, évitant ainsi de nombreuses refontes ultérieures.

### Phase 2 : Implémentation du Cœur et Résolution des Conflits

La seconde phase a été la plus dense techniquement car elle a confronté la théorie à la pratique du framework Qt. J'ai rapidement rencontré une difficulté majeure liée au cycle de vie de l'application, spécifiquement la confusion entre la gestion des clics et le dessin. Initialement, j'avais tenté d'intégrer la logique de jeu directement dans la boucle de dessin paintEvent. J'ai alors réalisé que cette fonction est appelée en permanence, environ 60 fois par seconde, pour rafraîchir l'écran, ce qui rendait impossible toute gestion d'état stable car les variables changeaient continuellement. J'ai donc dû restructurer le code pour que la fonction mousePressEvent gère exclusivement la logique et les changements d'état dans la matrice, tandis que paintEvent se contente de lire la matrice pour afficher le résultat, séparant ainsi strictement le traitement de l'affichage.

Une autre réflexion importante a concerné la génération aléatoire des mines. Lors de l'écriture de la fonction plantBombs, je me suis interrogé sur la portée de la variable aléatoire. J'ai conclu qu'il était bien plus propre, selon les principes de qualité du code, de déclarer cette variable localement à l'intérieur de la fonction plutôt que d'en faire un membre de la classe, puisque son utilité est éphémère et limitée à l'initialisation du plateau.

### Phase 3 : Affinement, Rigueur Mathématique et Optimisation

La dernière phase a été consacrée à la stabilisation du programme et à l'adoption d'une rigueur mathématique accrue. J'ai notamment dû résoudre une confusion persistante entre le monde visuel utilisant des pixels et le monde logique utilisant des indices de matrice. Pour pallier ce problème qui causait des erreurs de placement, j'ai mis en place une conversion systématique dès l'entrée des fonctions de clic, et j'ai unifié la nomenclature des variables en utilisant strictement r pour les lignes et c pour les colonnes dans tout le projet.

Enfin, j'ai opéré un changement de paradigme fondamental. Au lieu de gérer le jeu en fonction du type de clic physique, j'ai réorienté la logique autour de l'état de la tuile. Ce passage d'une vue événementielle à une vue basée sur l'état de la grille a grandement simplifié l'algorithme de propagation "effet domino". Cela a permis d'écrire une fonction récursive neighbourState propre et efficace qui révèle automatiquement les zones vides sans erreur de débordement mémoire.

## Conclusion

En conclusion, ce travail pratique a été une opportunité d'aller au-delà de la simple syntaxe C++ pour toucher aux concepts d'architecture logicielle. J'ai appris qu'un programme graphique robuste repose avant tout sur une séparation stricte des responsabilités : la mémoire ne doit pas dépendre de la vue, et la boucle de dessin ne doit jamais effectuer de calculs logiques. Le résultat final est un jeu de Démineur stable, utilisant QPixmap pour une interface soignée, et dont le code reflète une compréhension claire de la manipulation matricielle.