C++ and Qt Project Style Guide and Project Management Guide

Team A – Baseball Lovers

# Coding Style and Formatting Standards

All source files must begin with a standardized header comment block that provides metadata about the file. This includes the file name, author, date of creation, and a brief description of the file's contents or purpose. This ensures traceability and quick identification when reviewing or updating the codebase.  
  
Example:  
/\*  
 \* File: loginwindow.cpp  
 \* Author: Breanna Bisuna  
 \* Created: March 10, 2025  
 \* Description: Implements the login window functionality and user authentication logic.  
 \*/

Code should be consistently formatted for readability and maintainability. Use four spaces for indentation — tabs should not be used. Opening braces should be placed on the same line as control statements or function declarations.  
  
Example:  
void authenticateUser(const QString& username, const QString& password) {  
 if (username.isEmpty() || password.isEmpty()) {  
 qDebug() << "Empty credentials!";  
 return;  
 }  
}

# Naming Conventions

Variables and function names must be descriptive and follow consistent casing rules. Use camelCase for variables and PascalCase for function names and class names. Constants should be named using UPPER\_SNAKE\_CASE.  
  
Examples:  
int totalTicketsSold;  
double calculateRevenue(double price, int quantity);  
const double TAX\_RATE = 0.0825;  
class TeamManager {  
public:  
 void addTeam(const QString& name);  
};

# Commenting and Documentation

Each function must be preceded by a comment block that briefly describes its purpose, input parameters, return value, and any side effects. Inline comments should be used to clarify complex logic but should be kept concise and relevant.  
  
Example:  
/\*\*  
 \* @brief Calculates the total cost including tax.  
 \* @param subtotal The subtotal amount before tax.  
 \* @return Total cost after applying tax.  
 \*/  
double computeTotalCost(double subtotal) {  
 return subtotal \* (1 + TAX\_RATE); // Apply tax rate  
}

# Header Files and Organization

Header files must be protected using either include guards or #pragma once to prevent multiple inclusions. Logical separation of code should be observed — place related declarations and definitions in appropriately named .h and .cpp files.  
  
Example (include guard):  
#ifndef LOGINWINDOW\_H  
#define LOGINWINDOW\_H  
  
#include <QDialog>  
  
class LoginWindow : public QDialog {  
 Q\_OBJECT  
public:  
 explicit LoginWindow(QWidget\* parent = nullptr);  
};  
  
#endif // LOGINWINDOW\_H

# Object-Oriented Practices

All classes should follow good object-oriented practices. Include appropriate constructors, destructors, and copy-control members if needed. Favor composition over inheritance unless a clear “is-a” relationship is present.  
  
Example:  
class Stadium {  
public:  
 Stadium(const QString& name, int capacity);  
 ~Stadium();  
 QString getName() const;  
  
private:  
 QString name;  
 int capacity;  
};

# GitHub Project Management and CI/CD

GitHub is the central tool for managing all aspects of the project. It acts as the version control system, code repository, and collaboration platform. The project uses GitHub’s Kanban board (under the Projects tab) to track user stories, tasks, bugs, and progress. Each task is added as a GitHub issue and categorized under columns like 'To Do', 'In Progress', 'In Review', and 'Done'. During sprint planning, the team performs task estimation using the Planning Poker technique to gauge the complexity and time required. **If a task is delayed or incomplete by the sprint deadline, it is reassigned to a team member who is ahead on their own tasks. If no such member is available, the Product Owner will manually reassign the task based on priority and team capacity.**

Development follows a structured branching strategy. The `main` branch contains production-ready, stable code, while the `feature` branch is the base for active development. Developers create individual branches from `feature` to work on tasks. To merge changes, they initiate a pull request (PR) on GitHub. This includes selecting the target (`main`) and source (feature/task) branches, writing a clear description, and requesting peer reviews. Before creating a PR, developers must synchronize with the latest remote changes using `git pull origin main` or `git fetch` followed by `git rebase`. Branch switching is done with `git checkout branch-name`. If merge conflicts arise, developers attempt to resolve them locally. If the conflict is complex or touches business logic, the Product Owner must be consulted before finalizing the resolution.

GitHub Actions is configured for CI/CD (Continuous Integration and Deployment). Each push or pull request triggers automated workflows that build the application, generate Doxygen documentation, and execute unit tests. Developers can check the status of a build under the 'Actions' tab or directly in the pull request view. If a build fails, the log provides specific error messages. Developers are expected to address build errors promptly. If the issue persists or is unclear, it should be escalated to the Product Owner after attempting a fix or consulting relevant team members.