ESAT | ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

> **Programación Avanzada**
> Arnau Roselló
> Curso 2025/2026

BlackJack

Technical Documentation and Development Process

AUTOR/ES: Manuel Pablo Golpe Meizoso

Index

1.- Introduction

1.1 Project Objectives

The primary objective of this project is to develop a fully functional BlackJack game using object oriented C++ programming. This implementation serves as a comprehensive exercise shows proficiency the following key areas:

- Implementation of clean, maintainable and portable code.
- Design and implementation of abstract interfaces used for game components.
- Development of a hierarchy based flexible rule system.
- Creation of various AI player behaviours using different strategy patterns.
- Integration of terminal-based UI using the FTXUI library.
- Application of proper build systems through the use of Premake5.

Some secondary objectives include gaining practical and theoretical experience with modern C++ features, understanding game state management and implementing a mathematically correct blackjack behaviour through a two dimensional vector.

1.2 General Game Description

Blackjack is a famous card game played between a dealer and a given number of players. The goal is to beat the dealer by obtaining cards that total close to 21 without exceeding it.

This implementation has this core gameplay features and mechanics:

- Support for 2 to 6 simultaneous players.
- Three different rule sets: Base Rules, Disruptive Rules and Fitfluencer Rules.
- Four distinct AI behaviour patterns selected at the beginning of the game: Mathematically Correct, Drunk, Fearful, and Degenerate Gambler.
- Full BlackJack game actions: Hit, Stand, Double and Split.
- Insurance/Safe betting when the dealer shows an Ace.
- Terminal-based graphical interface with colored card rendering.

The game continues until either all players run out of money or the dealer becomes bankrupt. Then the final game screen shows how much money every player went away, as some can be kicked from the table for doing illegal moves (mostly the drunk player as it's decision making is completely random) and this screen also shows the dealer’s final money amount.

2.- Algorithms and System Design

2.1 Definition of Main Algorithms

2.1.1 Deck Shuffling Algorithm

The deck shuffling algorithm utilizes the Fisher-Yates shuffle algorithm implemented through C++ Standard Library’s std::shuffle function with pseudo-random number generator through the use of std::mt19937. This provides a cryptographically correct randomness suited for a card game simulation:

```
void MGMTable::ShuffleDeck() {

    std::random_device seed;

    std::mt19937 rand(seed());

    std::shuffle(deck_.begin(), deck_.end(), rand);

}
```

This algorithm has O(n) time complexity with n being the number of cards. For a standard 52-card deck, this function provides efficient shuffling while maintaining a uniform distribution of card positions.

2.1.2 Hand Value Calculation Algorithm

The hand evaluation algorithm handles the dual nature of Aces (valued 1 or 11) optimally by a classical BlackJack approach managing soft and hard hands. The implementation uses the following approach:

1. Sum of all card values, taking all Aces as 11 initially.
2. Count the exact number of Aces in the hand.
3. While total exceeds the win point and Aces remain counted as 11, subtract 10 to set Ace value as 1.
4. Return the final total along with hand status managing soft and hard hand boolean parameters.

This algorithm runs in O(n) times where n is the number of cards in the hand. HandInfo structure stores: total value, whether the hand is soft (contains an Ace counted as 11), and whether it's a splittable pair.

2.1.3 Strategy Decision Matrix

The mathematically correct player behaviour implements a complete strategy lookup table organized into three sections:

• Hard totals (rows 0-9): Hands without Aces or with Aces counted as 1.
• Soft totals (rows 10-15): Hands with an Ace counted as 11.
• Pairs (rows 16-25): Hands with two cards of equal value.

Column indexes (0-9) represent the dealer's first shown card (2-ACE). The lookup operation is O(1), turning decision-making instantaneous regardless of the game's state complexity.

2.2 System Architecture

The system follows a layered architecture with clear separation of matters:

- **Interface Layer:** Abstract base classes (IGame, IPlayer, ITable) define contracts for game components. These interfaces ensure loose coupling and enable polymorphic behavior.
- **Implementation Layer:** Concrete classes (MGMGame, MGMPlayer, MGMTable) implement the interfaces with specific game logic. The MGM prefix indicates the project's namespace.
- **Rules Layer:** BaseRules and derived classes (DisruptiveRules, FitfluencerRules) encapsulate game configuration through virtual method rule overrides.
- **Presentation Layer:** GameRenderer class handles all terminal UI rendering using the FTXUI library, completely separated from game logic only taking the necessary parameters to render.

This architecture enables easy extension of rules, player behavior, and UI elements without modifying existing code, adhering to the Open-Closed Principle.

3.- Programming Paradigm

3.1 OOP vs Procedural Comparison

This project implements the Object-Oriented Programming paradigm rather than procedural programming. The following table shows why OOP was the best choice for a program of this characteristics:

Aspect	OOP Approach	Procedural Approach
Game Rules	Inheritance hierarchy that allows adding new rule sets by creating derived classes.	Would require compulsory switch statements of function pointers throughout the program's implementation.
Player Behaviour	Strategy pattern implemented through polymorphic method DecidePlayerAction.	Global functions with behavioural type parameters passed everywhere.
State Management	Encapsulated MGMTable with controlled assets through different implemented methods.	Global variables or large structs passed around between functions.
Testing	Interfaces enable mock implementations for unit testing.	Difficult to isolate components for testing.
Code Reusability	BaseRules provides default ruleset with derived classes overwritten as needed.	Copy-paste with modification or complex conditional logic without portability.

3.2 Critical Evaluation

The OOP approach proved highly beneficial for this project due to a large number of factors:

- **Extensibility:** Adding the different ruleset variants required only creating new classes that override specific methods. No other code modification needed.
- **Encapsulation:** The MGMTable class maintains complete control over the game's state. External code cannot directly manipulate the deck, hands or money; all access goes through the validated interface methods.
- **Polymorphism:** The game loop, implemented through MGMGame, works with IPlayer pointers while being completely isolated from the behavioural implementation. This made adding the four player personalities really straightforward.

However, OOP introduced its own share of overhead: the virtual function calls have slight performance costs, and the abstraction layers require a stronger initial design. For a game of this type, with minimal performance requirements, this tradeoffs are completely negligible, making OOP the clear winner.

4.- Detailed Implementation

4.1 MGMPlayer Class

The MGMPlayer class implements the IPlayer interface, providing decision making for AI-controlled players. Key decisions include:

- **Behavior System:** The PlayerBehavour enum defines four distinct playing styles. SetRandomBehaivour() each player a playing style, creating dynamism and diverse table mechanics.
- **Strategy Matrix:** The mat_correct_behavour 2D made up from 26 rows and 10 columns, encoding optimal decisions for every possible hand against every dealer upcard. This is a direct implementation from a mathematical approach to a BlackJack strategy.
- **Decision Methods:** HardRowIdx(), SoftRowIdx(), and PairRowIdx() translate hand characteristics into matrix row indices. The column index derives from the dealer's first visible card value.

The four behavioral implementations:

1. **kPB_MatCorrect:** Consults the strategy matrix for every decision. Bets 25% of current funds.
2. **kPB_DrunkPlayer:** Returns random actions using rand()%4. Also bets random amounts.
3. **kPB_FearOfSuccess:** Always stands. Bets minimum money allowed.
4. **kPB_DegenerateGambler:** Always hits. Bets 50% of funds.

4.2 MGMTable Class

MGMTable is the central state manager, implementing ITable with clear and comprehensive game state tracking:

Data Structures:

- hands_[player_index][hand_index]: 2D vector supporting split hands.
- player_bets_[player_index][hand_index]: Parallel structure tracking bets per hand.
- total_player_money_: Vector tracking each player's available funds.
- deck_: Vector of Card structures, managed as a stack (back = top).

Key Methods:

- ApplyPlayerAction(): Validates and executes player actions, handling the complex split logic and the different player action logic options.
- FinishRound(): Implements dealer drawing rules, compares hands, and calculates payouts.
- CleanTable(): Resets state between rounds, removes bankrupt players and players trying an illegal move.

4.3 MGMGame Class

MGMGame manages the complete game flow through its PlayGame() method:

1. Initialize players with random behaviors and display introductions.
2. Main game loop (while dealer and players have money).
3. Round phases:

   Clean table → Start round → Collect bets → Deal cards → Safe bet offers → Player actions → Dealer reveal → Finish round

4. Display round results and wait for user input.
5. Display final game results.

The GameRenderer nested class handles all FTXUI rendering, keeping presentation completely separate from game logic. This separation enables future UI changes without touching any of the game's logic code.

4.4 Rules System

The rules system shows effective use of inheritance and polymorphism:

BaseRules: Provides standard Blackjack configuration: win point 21, single deck, 2 initial cards, dealer stands on 17.

DisruptiveRules: Increases difficulty and makes the game more "radical": win point 25, 2 decks, 3 initial cards, dealer stands on 21. This creates a more volatile variant.

FitfluencerRules: Decreases target to 20, implements soft 17 rule (dealer hits on soft 17). This favors the house slightly while maintaining a fairly standard structure.

All rule classes share the same interface, enabling the game to work with any ruleset through a BaseRules reference. The rules reference is stored in MGMTable, selected at the beginning of the game through the game's interface and is queried throughout gameplay.

5.- IDE Usage

5.1 Development with Visual Studio

This project was developed using Microsoft Visual Studio 2022 on Windows 10/11. The development environment was configured as shown next:

Project Generation: Premake5 generates Visual Studio solution files from premake5.lua. This ensures consistent project configuration across all development machines.

Configuration: x64 architecture with Debug and Release configurations. C++17 standard enabled for modern language features.

Library Integration: FTXUI library installed in ftxui/install/ subdirectory. Include paths and library directories configured through Premake.

Build Process: compile.bat script invokes MSBuild directly, enabling command-line builds without opening the IDE.

Key Visual Studio features utilized:

- IntelliSense for code completion and syntax error detection
- Integrated debugger with breakpoints and watch windows
- Solution Explorer for project navigation
- Error List window for compilation error tracking

5.2 Evaluation of IDE Usage

Visual Studio proved highly effective for this C++ project:

Strengths:

- Excellent debugging capabilities, essential for tracking issues
- Strong IntelliSense support for C++ including STL containers
- Easy external library integration through project properties

Challenges:

- Initial FTXUI setup required manual path configuration as it was meant to be compiled with CMake directly.
- Solution file changes when using Premake require regeneration for each code iteration.

Overall, the IDE significantly improved the development speed through its comprehensive tools and debugging support. The Premake integration ensured reproducible builds across different development environments.

6.- Debugging Process

6.1 Applied Debugging Techniques

Several debugging techniques were used throughout the development:

- **Breakpoint Debugging:** Strategic breakpoints in ApplyPlayerAction() and FinishRound() allowed step-through analysis of game logic. Conditional breakpoints triggered only when specific hands occurred.

- **Watch Windows:** Monitoring hands_, player_bets_, and deck_ vectors revealed state errors and memory leaks. The ability to inspect std::vector contents directly was of great help.

- **Printf Debugging:** Temporary print statements in MGMPlayer::DecidePlayerAction() helped trace strategy matrix lookups: "is pair", "is soft", "is hard" outputs to check for correct access to the decision matrix and correct HandData management.

- **Memory Inspection:** Visual Studio's memory viewer helped identify uninitialized HandInfo structures that caused incorrect decisions and bad memory accesses.

6.2 Debugging for Safe Applications

Several safety-critical debugging practices were implemented:

Bounds Checking: All vector accesses validated indices before access. The GetHand() and GetNumberOfHands() methods prevent out-of-bounds reads.

Null Pointer Safety: The players_ vector initializes to nullptr; all player accesses go through GetPlayer() in order to be validated.

State Validation: ApplyPlayerAction() returns Result::Illegal for invalid states rather than allowing undefined and out of the ruleset behavior.

Money Integrity: Bet placement validates sufficient funds before deduction, preventing negative balances and returning kicking the bot from the table otherwise.

6.3 Debugging Evaluation

The most challenging bugs encountered and their resolutions:

- **Split Hand Bug:** Initially, split hands shared the same bet vector entry.

Resolution: Resize player_bets_ alongside hands_ in split logic.

- **Ace Counting Bug:** HandData() initially failed to decrement ace count when converting from 11 to 1.

Resolution: Track ace_count separately from hand iteration.

- **Dealer Draw Bug:** FinishRound() drew cards after determining dealer_value, while not updating it.

Resolution: Move ace adjustment inside the draw loop.

The combination of IDE debugging tools and strategic printf statements provided effective bug isolation for their resolutions. The most valuable technique was setting breakpoints in FinishRound() and stepping through complete rounds to verify game logic.

7.- Coding Standard

7.1 Conventions Used

The project follows a consistent coding standard based on ESAT's own C++ Style Guide with project-specific adaptations:

Naming Conventions:

- Classes: PascalCase (MGMPlayer, MGMTable, GameRenderer)
- Methods: PascalCase (GetHand, PlayInitialBet, DecidePlayerAction)
- Member Variables: snake_case with trailing underscore (deck_, player_num_, behaviour)
- Constants: kPrefixPascalCase (kSuitNum, kValueNum, kMaxPlayers)
- Enums: PascalCase with k prefix for values (kPB_MatCorrect, kPB_DrunkPlayer)

File Organization:

- Header files (.h): Class declarations, inline implementations, documentation
- Source files (.cc): Method implementations
- Interface files: I-prefix (IGame.h, IPlayer.h, ITable.h)

Documentation:

- Doxygen-style comments for classes and public methods
- @brief, @param, @return tags for method documentation
- Inline comments for complex logic sections

7.2 Importance of the Standard

Maintaining a consistent coding standard provided several benefits:

- **Readability:** Consistent naming allows quickly identifying variable types and purposes. Member variables are immediately recognized.
- **Maintainability:** The separation between interface and implementation files enables changes without affecting one another. Header guards prevent multiple inclusion issues.
- **Collaboration:** A documented standard enables other developers to contribute without extensive onboarding. The Doxygen comments enable automatic documentation generation.
- **Debugging:** Consistent patterns make it easier to predict where specific functionality resides, speeding up bug location.

The coding standard proved particularly valuable when implementing the GameRenderer class, where consistent method naming (DrawCard, DrawHand, DrawGameScreen) clearly showed each method's purpose.

8.- Implementation Analysis

8.1 Challenges Found

Several significant challenges were faced and managed during implementation:

- **Split Hand Implementation:** Managing multiple hands per player required 2D vectors for different hands and bets. The challenge was ensuring all parallel data structures stayed synchronized during split operations and the whole game flow.
- **Strategy Matrix Indexing:** Mapping hand characteristics to matrix rows required careful analysis of the strategy table structure. Different indexing schemes for hard totals, soft totals, and pairs added complexity and generated bugs accessing through the matrix index handling functions.
- **FTXUI Integration:** Learning the FTXUI library's reactive paradigm required significant effort. The element-based composition model differs from traditional rendering approaches, but while having a html base was somewhat intuitive once I got the hang of it.
- **Ace Value Handling:** The dual nature of Aces (1 or 11) affected multiple systems: hand evaluation, strategy lookup, and dealer drawing logic. Ensuring consistent behavior across all three systems was challenging.
- **Memory Management:** The game creates dynamic player objects in main.cc. Ensuring proper cleanup with delete statements required careful tracking of ownership.

8.2 Code Evaluation

Critical evaluation of the final implementation:

Strengths:

- Clean interface separation enables easy testing and extension
- Complete basic strategy implementation for mathematically correct play
- Flexible rule system supporting multiple game variants
- Terminal UI rendering with color-coded cards and clear status displays

Areas for Improvement:

- Some methods in MGMTable are quite long; FinishRound() could be broken into smaller functions by using private helpers.
- Many behavioural related code have typos (should be named Behaviour not Behaivour)
- Raw pointers in main.cc could be replaced with smart pointers for automatic memory management

9.- Compilation and Execution

9.1 Compilation Process

The build system uses Premake5 for cross-platform project generation:

Step 1 - Generate Project Files:

```
premake5 vs2022
```

This creates Visual Studio 2022 solution files in the build/ directory.

Step 2 - Build Solution:

```
compile.bat
```

The batch file invokes MSBuild with Debug configuration for x64 platform.

Project Structure: The premake5.lua defines three projects:

- MGMPlayer: Static library containing player logic
- MGMGame: Static library containing game flow logic
- MGMTable: Static library containing table management
- BlackJack: Console application linking both libraries plus FTXUI

Dependencies are managed through Premake's links directive using lua, ensuring correct build order.

9.2 Test Cases

Manual testing verified correct behavior across multiple scenarios:

Test Case 1 - Rule Variants: Started games with each ruleset, verified different win points (20, 21, 25) and initial card counts (2, 3).

Test Case 2 - Player Behaviors: Observed each behavior type: MatCorrect follows strategy matrix, DrunkPlayer acts randomly, FearOfSuccess always stands, DegenerateGambler always hits.

Test Case 3 - Split Functionality: Verified split creates two hands when a player is dealt a pair, with independent betting and card dealing.

Test Case 4 - Double Down: Verified bet doubling, single card draw, and correct payout calculation.

Test Case 5 - Ace Handling: Hands with multiple Aces correctly adjust values to avoid busting when possible (soft-hands).

Test Case 6 - Player Elimination: Verified players with insufficient funds are removed from subsequent rounds.

Test Case 7 - Game Termination: Game ends correctly when all players lose or dealer runs out of money.

10.- Conclusions

10.1 Objectives Accomplished

The project successfully achieved all primary objectives:

1. Implemented a complete, functional Blackjack game with various rulesets.
2. Created a flexible inheritance hierarchy for ruleset variations.
3. Implemented four distinct AI player behaviors including a mathematically correct strategy managed through a decision matrix.
4. Integrated FTXUI library for a terminal-based UI.
5. Configured reproducible builds using Premake5.

The secondary objectives were also met: the project demonstrates modern C++ practices, proper game state management, and correct implementation of a Blackjack strategy.

10.2 Main Learnings

Key lessons learned during development:

Interface Design: Learning to use and implement interfaces ended up giving the code a really strong internal structure and modularity. The clean ITable interface made MGMTable implementation straightforward.

State Management: Card games have deceptively complex states. The 2D vector approach for hands and bets proved essential for handling splits correctly.

Library Integration: External libraries like FTXUI require significant learning investment but notably improve output quality.

10.3 Possible Future Improvements

Several enhancements could improve the project:

- Add human player input mode alongside AI players.
- Implement card counting AI that adjusts bet sizes based on deck composition.
- Create a graphical UI version using a game framework.
- Add local statistics tracking and session persistence.
- Replace raw pointers with smart pointers throughout.

10.4 post-mortem

Throughout the game's implementation and being kind of the first big project I made solely in C++, coming from C, the change of paradigm from procedural to OOP was quite a challenge. The learning curve was more steep than I anticipated but it paid off a lot. This project has made me understand how OOP really works, its advantages and disadvantages and overall it helped me to get familiarized with a new way of working, programming and structuring the code. All throughout this project I encountered problems and bugs that were a lot easier to work on and solve rather than with a procedural approach.

10.5 Bibliography

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.

Meyers, S. (2014). *Effective Modern C++*. O'Reilly Media.

FTXUI Library Documentation. (n.d.). Retrieved from https://arthursonzogni.github.io/FTXUI/

Premake Documentation. (n.d.). Retrieved from https://premake.github.io/docs/

Wizard of Odds. (n.d.). *Blackjack Basic Strategy Engine*. Retrieved from https://wizardofodds.com/games/blackjack/strategy/calculator/

Microsoft. (n.d.). *Visual Studio Documentation*. Retrieved from https://docs.microsoft.com/en-us/visualstudio/

cppreference.com. (n.d.). *C++ Standard Library Reference*. Retrieved from https://en.cppreference.com/