ESAT INNOVATION SCHOOL | ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

BlackJack
MEMORIA

Nombre Asignatura:Programación Avanzada
Profesor: Arnau Roselló
Curso: 2025/2026
Autor/es:Pau Alvarez Belenguer

Índice/Index

1. TECHNICAL REPORT

1.1 Main Algorithm Definitions

The Blackjack game system is based on several key algorithms that manage the game flow, hand value calculation, and player decision-making. These algorithms form the computational core of the implementation and ensure correct game mechanics according to the established rules.

Hand Value Calculation Algorithm

The most critical algorithm is the hand value calculation, implemented in CalculateHandValue within both PABPlayer.cc and PABTable.cc files. This algorithm must handle the unique characteristic of Aces in Blackjack, which can be worth either 1 or 11 points depending on the context of the hand. The algorithm begins by initializing a total value counter to zero and a separate counter specifically for tracking Aces. As it iterates through each card in the hand, it applies different rules depending on the card type: face cards such as Jacks, Queens, and Kings contribute a value of 10 points, while Aces increment the dedicated Ace counter, and all other cards add their numeric face value to the total.
Once all cards have been processed, the algorithm enters a second phase where it determines the optimal value for each Ace. For each Ace encountered, the algorithm evaluates whether adding 11 points would cause the total to exceed the winning point threshold defined by the current rule set. If adding 11 keeps the total at or below the winning point, the Ace is counted as 11; otherwise, it is counted as 1. This intelligent evaluation ensures that hands are always calculated to their maximum beneficial value without busting. The algorithm finally returns the computed total value, which represents the strongest possible interpretation of the hand within the game rules.

Round Management Algorithm

The main game flow is orchestrated through a comprehensive round management algorithm implemented in the PlayGame method within PABGame.cc. This algorithm operates as a continuous loop that controls the entire lifecycle of each game round. The round begins with a verification of game-over conditions, checking whether the dealer or all players have been eliminated due to bankruptcy. If the game continues, the algorithm initiates a new round by shuffling the deck and dealing the initial cards to all participants according to the configured number of initial cards in the active rule set.
Following the initial deal, the algorithm enters the initial betting phase, where each player must place their wager for the round. If the dealer's visible card is an Ace, the algorithm then proceeds to an insurance betting phase, offering players the opportunity to place a side bet against the dealer achieving Blackjack. Subsequently, the algorithm manages each player's turn sequentially, allowing them to make strategic decisions about their hands through actions such as hitting, standing, doubling down, or splitting pairs. Once all players have completed their turns, the algorithm executes the dealer's play according to the established rules, automatically drawing cards until the dealer's hand value reaches or exceeds the dealer stop threshold. Winners are then determined by comparing final hand values, payouts are distributed accordingly, and the table is cleaned and prepared for the next round by clearing all hands and resetting bets.

Decision-Making Algorithm

The player strategy implementation relies on a sophisticated decision-making algorithm based on a pre-calculated optimal strategy table stored in the actiontable data structure within PABPlayer.cc. This algorithm, implemented in the DecidePlayerAction method, determines the statistically optimal action for any given game situation by consulting this comprehensive lookup table. The decision process takes into account multiple factors that characterize the current game state: the dealer's visible card value provides crucial information about the dealer's potential hand strength, while the player's hand total value represents their current position in the game.

The algorithm also considers the specific composition of the player's hand, distinguishing between hard hands, soft hands containing Aces, and pairs that could potentially be split. Additional game state information, such as the number of hands currently being played by the player and whether certain actions like doubling down remain available, further refines the decision-making process. The strategy table itself is organized as a two-dimensional matrix with 26 rows representing different player hand situations and 10 columns corresponding to the dealer's possible visible cards. This comprehensive mapping ensures that every conceivable combination of player hand and dealer upcard has a pre-determined optimal action, resulting in gameplay that adheres to mathematically sound Blackjack basic strategy principles.

For more information please check de doxygen documentation.

1.2 Interface Explanation and Implementation

The system architecture is built upon four fundamental interfaces that define the contracts and responsibilities of different components within the game. These interfaces enable extensibility and polymorphism, facilitating the implementation of different game variants while maintaining a clean separation of concerns and allowing components to interact through well-defined protocols.

IGame Interface

The IGame interface defines the contract for game execution and serves as the highest-level abstraction in the system. This interface contains a single pure virtual method called PlayGame, which represents the main entry point for executing the complete game loop from initialization through multiple rounds to final completion. The concrete implementation of this interface, PABGame, takes responsibility for orchestrating the entire game flow by coordinating interactions between the table, players, and rules. This orchestration includes managing the sequence of rounds, handling user interface output, controlling the betting phases, and determining when the game should terminate based on bankruptcy conditions.

IPlayer Interface

The IPlayer interface defines the behavioral contract for all player implementations in the system. This interface establishes three essential methods that encapsulate player decision-making processes. All methods receive a const reference to the table, allowing them to query the current game state without modifying it, thus maintaining proper encapsulation and preventing unintended side effects. The DecidePlayerAction method determines which action the player should take given the current situation, choosing among options such as hitting, standing, doubling down, or splitting pairs based on the visible dealer card, the player's hand composition, and other contextual factors.
The DecideInitialBet method establishes the wagering strategy by determining the amount of money the player commits at the start of each round, taking into account the player's available funds and risk tolerance. The DecideUseSafe method implements the insurance decision logic, determining whether the player should place a side bet when the dealer shows an Ace as their visible card. The concrete PABPlayer implementation utilizes an optimal strategy table to make these decisions, providing competent and realistic gameplay that adheres to mathematically sound Blackjack principles while demonstrating how the interface enables different player implementations ranging from simple rule-based AI to potential human player interfaces or advanced card-counting strategies.

ITable Interface

The ITable interface represents the most complex component in the system, managing the complete state of the game table and all interactions with cards, bets, and player hands. This interface divides its methods into two distinct categories that reflect different types of operations. The query methods, all marked as const, provide read-only access to the current game state without modifying it. These include GetHand for retrieving a specific player's cards, GetNumberOfHands for determining how many hands a player currently controls after potential splits, GetPlayerMoney for checking available funds, GetPlayerCurrentBet for examining the wager on a specific hand, GetPlayerInitialBet for accessing the original bet amount, and GetDealerCard for viewing the dealer's visible upcard.

The mutation methods modify the game state and drive the progression of gameplay. DealCard adds a card from the deck to a specified player's hand, PlayInitialBet processes the primary wager at the start of a round, PlaySafeBet handles insurance bet placement, and ApplyPlayerAction executes the chosen action whether it be hitting, standing, doubling, or splitting. The StartRound method initializes a new round by shuffling the deck and dealing initial cards, FinishRound executes the dealer's turn according to the rules and determines winners and payouts, and CleanTable resets all hands and bets to prepare for the next round. The PABTable implementation maintains all game state including player hands organized in a vector to support split scenarios, individual bet tracking for each hand, the complete card deck with shuffling capability, and the monetary balance for each participant including the dealer.

BaseRules Class

While technically not a pure interface due to its default implementations, BaseRules serves as the foundation for defining and customizing the configurable parameters that govern gameplay. This class provides virtual methods that return the fundamental game constants and can be overridden by derived classes to create rule variants. The GetWinPoint method determines the target score that constitutes a winning hand, defaulting to the traditional value of 21. NumberOfDecks specifies how many standard 52-card decks are combined and shuffled together for play, with a default of 1 representing the classic single-deck variant. InitialCards defines how many cards each player receives at the start of a round, typically 2 in standard Blackjack.
The InitialPlayerMoney and InitialDealerMoney methods establish the starting bankrolls for players and the house respectively, providing the economic foundation for the betting system. MinimumInitialBet and MaximumInitialBet set the acceptable range for wagers, preventing bets that are too small to be meaningful or too large relative to available funds. Finally, the DealerStop method specifies the threshold at which the dealer must cease drawing cards, with the default value of 17 representing the standard house rule where the dealer stands on all 17s. This design allows game variants like ExtremeRules and Simplified to override specific parameters while inheriting sensible defaults for unchanged values, demonstrating the power of inheritance in creating diverse gameplay experiences from a common foundation.

For more information please check de doxygen documentation.

1.3 Inheritance System and Rule Variations

The inheritance system serves as a powerful mechanism for enabling different game variants while maximizing code reuse and maintaining consistency across implementations. The system is primarily implemented through the BaseRules class hierarchy, which provides a flexible foundation for defining game parameters that can be selectively overridden to create distinct gameplay experiences. Three concrete rule variants have been implemented to demonstrate this extensibility.

BaseRules: Original Blackjack

The BaseRules class represents the classic Blackjack configuration with traditional parameters that most players would recognize from casino gameplay. In this variant, the winning point threshold is set to the standard value of 21, requiring players to achieve this target without exceeding it to maximize their hand strength. The game utilizes a single standard 52-card deck, and each participant receives two cards during the initial deal. The dealer must continue drawing cards until reaching a total of at least 17 points, at which point the dealer stands regardless of the players' hands. This configuration provides the baseline gameplay experience and serves as the parent class from which all variants derive their default behaviors.

ExtremeRules: Extreme Blackjack

The ExtremeRules variant introduces significant modifications designed to create longer, more complex games with higher variance and strategic depth. The winning point threshold is elevated to 25, substantially increasing the target players must reach and fundamentally altering optimal strategy calculations. To accommodate this higher target and increase the number of possible hand combinations, the game employs two complete decks shuffled together rather than one. Each player begins with three cards instead of the traditional two, providing more information from the outset and creating a richer initial strategic landscape. The dealer's stopping threshold is raised to 21, meaning the dealer will continue drawing cards until achieving at least this value, which aligns more closely with the elevated winning point and creates more dramatic showdowns between players and the house.

Simplified: Simplified Blackjack

The Simplified variant takes the opposite approach from ExtremeRules, creating a streamlined version designed for faster gameplay and quicker rounds. The winning point is reduced to 20, lowering the barrier for achieving strong hands and increasing the frequency of wins and pushes. This variant maintains the single-deck configuration and two-card initial deal from the original rules but modifies the dealer behavior by setting the stop threshold at 16. This lower dealer threshold means the dealer will stand earlier than in traditional Blackjack, creating a mathematical advantage for players who can reach higher totals while increasing the dealer's risk of busting when forced to draw on hands totaling 16 or less.
Advantages of the Inheritance System
This inheritance-based design provides numerous architectural and practical advantages that validate its implementation. The system demonstrates excellent extensibility, as new game variants can be introduced simply by creating a new class that inherits from BaseRules and overrides only the specific parameters that differ from the base configuration. This selective override mechanism means that a new variant focusing solely on deck count could override just the NumberOfDecks method while automatically inheriting sensible defaults for all other parameters. The polymorphic nature of the design allows all components throughout the system to work with BaseRules references, enabling runtime rule selection without requiring any changes to the game logic, table management, or player strategy code.

The centralization of rules in this class hierarchy significantly enhances maintainability by eliminating magic numbers and configuration values scattered throughout the codebase. When a parameter needs adjustment, developers need only modify the appropriate method in the relevant rules class rather than hunting for hardcoded values across multiple files. This centralization also ensures consistency, as all components necessarily query the same rule methods to obtain their operational parameters.
An important lesson emerged during development regarding the proper implementation of this pattern. Initially, the PABTable class declared its rules member as a value type, which caused object slicing when a derived rules object was assigned to it. This meant that when an ExtremeRules instance was passed to PABTable, only the BaseRules portion was copied, losing all the overridden methods from the derived class. This manifested as a critical bug where the dealer would stand at 17 instead of the expected 21 in ExtremeRules mode. The solution required changing the declaration from 'BaseRules rules;' to 'BaseRules& rules;' to maintain a reference rather than a copy, and initializing this reference properly in the constructor's initialization list using the syntax 'PABTable::PABTable(BaseRules& game_rules) : rules(game_rules)'. This correction ensures that virtual method dispatch works correctly and all derived class behaviors are preserved, demonstrating the importance of using references or pointers when working with polymorphic class hierarchies in C++.

1.4 Design Decision Justification

The architectural decisions made throughout this project reflect careful consideration of software engineering principles, practical implementation concerns, and long-term maintainability. Each major design choice was made to address specific challenges while adhering to established best practices in object-oriented programming.

Separation of Responsibilities

The architecture strictly adheres to the single responsibility principle, with each class having a clearly defined and focused purpose. The PABGame class serves as the high-level orchestrator, managing the overall game flow, coordinating interactions between components, handling user interface output, and controlling the progression through multiple rounds. This class does not concern itself with the internal mechanics of card management or hand evaluation, delegating these responsibilities to more specialized components.
The PABTable class assumes responsibility for maintaining all game state, including the deck of cards, player hands, betting amounts, and money balances. It implements the logic for validating player actions, ensuring that moves are legal given the current game state, and executing the dealer's turn according to the established rules. The PABPlayer class focuses exclusively on decision-making strategy, implementing the algorithms that determine when to hit, stand, double, or split based on the current situation. Finally, the PABRules hierarchy defines all configurable game parameters, centralizing values that might otherwise be scattered throughout the codebase as magic numbers. This clear separation makes the codebase easier to understand, test, and modify, as changes to one aspect of the system rarely require modifications to unrelated components.

Interface Usage and Abstraction

The decision to build the system around well-defined interfaces provides numerous benefits that justify the additional abstraction layer. Interfaces enable the system to change implementations without affecting code that depends on those implementations, as long as the contract defined by the interface remains satisfied. This means that the PABPlayer implementation could be replaced with an entirely different strategy, or even a human player interface accepting keyboard input, without requiring any changes to PABGame or PABTable.
The interface-based design also facilitates unit testing by allowing the creation of mock objects that implement the same interfaces but provide controlled, predictable behavior for testing purposes. A test could create a mock ITable that returns specific predetermined cards, enabling systematic testing of player decision logic under known conditions. Additionally, the interfaces make the system inherently more flexible and extensible, as new implementations can be added without modifying existing code, adhering to the open-closed principle of software design.

Memory Management Strategy

The memory management approach prioritizes safety and correctness by preferring references over pointers whenever the semantics permit. The table parameter in PABGame is declared as 'ITable& table', using a reference type that guarantees a valid table object always exists and eliminates any possibility of null pointer dereferences. Similarly, the rules member in PABTable uses 'BaseRules& rules' to ensure that valid game rules are always available. References provide the additional benefit of making the code's intent clearer, as a reference cannot be reseated to refer to a different object after initialization, making the relationships between objects more explicit and predictable.
However, the player vector in PABGame is necessarily declared as 'std::vector<IPlayer*> Players', using pointers because vectors require copyable or moveable elements, and polymorphic behavior necessitates storing base class pointers rather than attempting to store derived class objects by value, which would cause slicing. The decision to create most instances on the stack in main.cc and pass them by reference throughout the system minimizes dynamic memory allocation and eliminates concerns about memory leaks or dangling pointers. The only exception is the BaseRules pointer created dynamically to

support runtime polymorphism in the menu selection system, where the specific rules variant cannot be determined until the user makes their choice.

Data Structure Design

The PlayerInfo structure in PABTable represents a carefully considered design that elegantly handles the complexities of Blackjack gameplay. By using vectors for hands, player_bet, and total_hand_value, the structure naturally accommodates the variable number of hands that result from split operations. When a player splits a pair, the system simply appends a new hand to the vector, duplicates the bet amount, and can then manage each hand independently through indexed access.
This design avoids the limitations of a fixed-size array, which would either impose arbitrary restrictions on the number of allowed splits or waste memory by allocating space for the maximum possible number of hands regardless of actual usage. The structure also encapsulates related data, keeping a player's hands, bets, and money together in a single cohesive unit rather than maintaining parallel data structures that must be carefully synchronized. The safe_bet member specifically handles insurance bets separately from regular wagers, maintaining clarity about the purpose and lifecycle of different bet types.
Strategy Table Implementation
The implementation of the strategy table as a static const matrix provides several important advantages. The static qualifier ensures that only a single instance of the table exists in memory regardless of how many PABPlayer objects are created, as the strategy data is identical for all players and requires no per-instance customization. This sharing reduces memory consumption and improves cache locality. The const qualifier guarantees immutability, preventing accidental modifications to the carefully constructed strategy table and enabling the compiler to place the data in read-only memory segments for additional safety.
The table-based lookup approach provides constant-time O(1) access to decisions, making the decision-making process extremely efficient regardless of the complexity of the underlying strategy. The two-dimensional array structure naturally maps to the conceptual organization of Blackjack strategy, where one dimension represents player hand situations and the other represents dealer upcards. This makes the code easier to understand and maintain compared to a complex web of conditional statements that would be required to encode the same decision logic procedurally.

2. DEVELOPMENT DOCUMENTATION

2.1 Code Writing Process Analysis

The development process followed a methodical, incremental approach that built the system from foundational abstractions to concrete implementations, ensuring that each component was well-understood before proceeding to dependent modules. This strategy minimized integration issues and allowed for early detection of architectural problems.

Phase 1: Interface Design and Analysis
Development began with a thorough examination of the four provided interfaces: IGame, IPlayer, ITable, and BaseRules. This initial phase was crucial for understanding the contracts that the system must fulfill and the specific responsibilities assigned to each component. By starting with interface analysis rather than immediately diving into implementation, the project established a clear architectural vision and identified the key interactions between components. This foundation proved invaluable throughout subsequent development phases, as it provided a reference point for resolving ambiguities and ensuring that implementations remained consistent with the intended design.

Phase 2: Table Implementation
The PABTable class was implemented first because it represents the central repository of game state and provides the foundation upon which other components depend. This phase began with the card deck management functionality, implementing the FillDeck method to populate the deck with the appropriate number of cards based on the rule configuration,

and the ShuffleDeck method to randomize card order before each round. The PlayerInfo data structure was carefully designed to encapsulate all information about each player, including their hands, bets, and available money.
The implementation then progressed to the query methods, which provide read-only access to game state. These getters were relatively straightforward to implement but required careful attention to const correctness to ensure they could be called from const contexts. The betting logic and validation methods came next, implementing checks to ensure that bets fall within acceptable ranges and that players have sufficient funds to cover their wagers. Finally, the round completion algorithm was implemented in FinishRound, incorporating the dealer's play logic and the winner determination calculations that compare final hand values and distribute payouts accordingly.

Phase 3: Player Strategy Implementation
With the table infrastructure in place, development shifted to implementing the PABPlayer class and its decision-making capabilities. The core of this work involved creating the strategy table based on established Blackjack basic strategy principles. This table required careful construction to ensure that all 26 player hand situations were correctly mapped against the 10 possible dealer upcards. The TableIndex function proved particularly challenging, as it needed to correctly identify whether a hand represented a pair that could be split, a soft hand containing an Ace, or a hard hand, and then map these situations to the appropriate row indices in the strategy table.
The betting decision logic was implemented to follow conservative principles, sizing bets appropriately based on available funds while respecting the minimum and maximum bet constraints defined by the rules. The insurance decision logic was kept simple but sound, generally declining insurance bets except in favorable situations as dictated by basic strategy principles.

Phase 4: Game Controller Development
The PABGame class implementation focused on orchestrating all previously developed components into a cohesive gameplay experience. The main game loop was structured to continuously cycle through rounds until termination conditions were met, checking after each round whether the dealer or all players had gone bankrupt. Player turn management required implementing logic to iterate through each player and, when splits occurred, through each of their hands, providing appropriate prompts and feedback through the console interface.
The user interface implementation, while textual, was designed to provide clear and informative output about the game state, including visible cards, current bets, and available actions. The split handling logic proved complex, requiring careful management of multiple hands for a single player, including proper bet duplication, independent decision-making for each hand, and correct tracking of which hand was currently being played.

Phase 5: Rule Variant Implementation
The final implementation phase involved creating the ExtremeRules and Simplified variants by inheriting from BaseRules. This phase validated the extensibility of the inheritance-based design, as each variant required only minimal code to override the specific parameters that differed from the base configuration. The implementation demonstrated that new gameplay variants could be added with very little effort, requiring no modifications to existing game logic, table management, or player strategy code.

Phase 6: Integration and Testing
The final development phase connected all components in main.cc and conducted comprehensive testing of the complete system. This testing phase involved playing numerous rounds with different rule configurations, deliberately triggering edge cases such as splits, insurance bets, and dealer blackjacks to verify that all code paths functioned correctly. Testing revealed the object slicing bug in the rules reference handling, which was then diagnosed and corrected, demonstrating the value of thorough end-to-end testing in catching issues that might not be apparent in isolated component testing.

2.2 Challenges Encountered and Solutions Applied

Throughout the development process, several significant technical challenges emerged that required careful analysis and systematic problem-solving. Each challenge provided valuable learning opportunities and resulted in important improvements to the codebase.

Challenge 1: Object Slicing in BaseRules
The most critical bug encountered during development involved object slicing in the handling of the BaseRules hierarchy. Initially, the PABTable class declared its rules member as a value type using the syntax 'BaseRules rules;'. When the constructor assigned the incoming game_rules parameter to this member using 'rules = game_rules;', C++ performed a copy operation that only copied the BaseRules portion of the object. If the actual parameter was an instance of a derived class such as ExtremeRules or Simplified, all the derived class-specific data and virtual function overrides were lost in this copy, a phenomenon known as object slicing.
This manifested as a subtle but serious bug where the dealer would stand at 17 points even when playing with ExtremeRules, which specifies that the dealer should stand at 21. The bug was particularly insidious because the code compiled without warnings and ran without crashes; it simply produced incorrect behavior. The solution required changing the member declaration to 'BaseRules& rules;' to use a reference instead of a value, and initializing this reference in the constructor's initialization list using the syntax 'PABTable::PABTable(BaseRules& game_rules) : rules(game_rules)'. This change ensures that PABTable maintains a reference to the actual derived class object, preserving polymorphic behavior and allowing virtual method calls to dispatch to the correct derived class implementations.

Challenge 2: Dynamic Ace Valuation
The peculiar nature of Aces in Blackjack, which can count as either 1 or 11 points depending on context, presented a significant algorithmic challenge. An incorrect implementation could lead to hands with Aces always being calculated with the Ace valued at 1, missing opportunities to reach higher totals, or always using 11 and causing unnecessary busts when the hand total would exceed the winning point.
The solution involved implementing a two-phase algorithm that processes Aces separately from other cards. In the first phase, the algorithm iterates through all cards in the hand, maintaining a separate counter for Aces while immediately adding the values of non-Ace cards to the running total. In the second phase, the algorithm processes each Ace individually, making a context-sensitive decision for each one. For each Ace, the algorithm checks whether adding 11 to the current total would exceed the winning point threshold defined by the active rules. If adding 11 keeps the total within bounds, the Ace counts as 11; otherwise, it counts as 1. This greedy approach ensures that hands are always evaluated at their maximum beneficial value while avoiding busts. The algorithm was implemented consistently in both PABPlayer and PABTable to ensure that hand values are calculated identically across all components.

Challenge 3: Split Handling and Multiple Hands
When a player splits a pair, the game must manage two independent hands with separate bets, cards, and outcomes. This requirement complicated both the data structures and the game flow logic, as code that previously assumed each player had exactly one hand needed to be generalized to handle multiple hands per player.
The solution leveraged C++ standard library containers to elegantly handle the variable number of hands. Within the PlayerInfo structure, the hands, player_bet, and total_hand_value members were declared as std::vector types rather than single values or fixed-size arrays. This design allows each player to have an arbitrary number of hands, naturally accommodating splits without imposing artificial limits or wasting memory. When a split action is executed in the ApplyPlayerAction method, the code creates a new hand vector, moves one card from the original hand to the new hand, duplicates the bet amount and adds it to the player_bet vector, and deducts the additional bet from the player's available money. Each hand can then be played independently, with indexed access to the vectors ensuring that the correct hand receives new cards and that bets and payouts are properly tracked for each separate hand.

Challenge 4: Strategy Table Indexing
The strategy table implementation required sophisticated logic to map complex game situations to the correct table indices. The table must distinguish between pairs that can be split, soft hands containing Aces counted as 11, and hard hands, while also considering the dealer's upcard and various game state factors.
The TableIndex function solves this problem through a carefully ordered series of conditional checks. The function first examines whether the hand consists of a pair of cards with identical rank, as pair situations occupy specific rows in the strategy table and have unique optimal strategies. If the hand is not a pair, the function then checks whether it qualifies as a soft hand by determining if it contains an Ace that can be counted as 11 without busting. Soft hands occupy a different section of the strategy table because the presence of an Ace that can be valued at either 1 or 11 significantly affects optimal strategy. If the hand is neither a pair nor a soft hand, it is classified as a hard hand, and the function uses the total hand value to determine the appropriate table row. The function returns a vector containing the calculated indices, which are then used to look up the optimal action in the actiontable array. This systematic approach ensures that every possible game situation is correctly categorized and mapped to the appropriate strategic response.

2.3 Debugging Process Performed

The debugging process employed a combination of traditional print-based debugging, systematic testing of edge cases, and the sophisticated debugging tools provided by the Visual Studio IDE. Each identified issue was approached methodically, using a hypothesis-driven approach to isolate the root cause and verify that applied fixes resolved the problem completely.

Object Slicing Investigation
When testing revealed that the dealer was incorrectly standing at 17 points during ExtremeRules gameplay despite the rules specifying a dealer stop threshold of 21, a systematic investigation was initiated to identify the cause. Debug printf statements were strategically placed in the FinishRound method to output the value of dealer_stop at runtime, confirming that the value was indeed 17 rather than the expected 21. This observation led to an examination of how the rules object was being initialized and stored in PABTable.
Further investigation revealed that the constructor was using assignment rather than reference initialization, leading to the object slicing problem where only the BaseRules portion of ExtremeRules was being copied. Once the root cause was identified, the fix was straightforward: changing the member variable to a reference type and initializing it in the constructor's initialization list. After applying this correction, the debug output confirmed that dealer_stop now correctly reflected the value 21 when playing with ExtremeRules, validating that the polymorphic behavior was properly preserved.

Hand Calculation Verification
To ensure the correctness of the hand value calculation algorithm, particularly with respect to Ace handling, a comprehensive testing regime was implemented. Debug output was added to print each hand along with its calculated value, allowing visual verification that the algorithm was producing correct results. Special attention was paid to edge cases that historically cause problems in Blackjack implementations: hands containing multiple Aces, soft hands where an Ace is counted as 11, and hands with a mix of face cards and Aces.
A particularly important test case involved a hand of Ace-Ace-9 in ExtremeRules mode, which should be calculated as 21 (11 + 1 + 9) rather than 31 (11 + 11 + 9) or 11 (1 + 1 + 9). The testing confirmed that the algorithm correctly valued the first Ace as 11, recognized that valuing the second Ace as 11 would cause a bust, and therefore valued it as 1, then added the 9 to reach the optimal total of 21. This verification process built confidence that the Ace handling logic was functioning correctly across all scenarios.

Split Functionality Validation
The split functionality required thorough testing due to its complexity and the number of related operations that must execute correctly. Testing verified that when a player split a pair, a second hand was properly created with one card moved from the original hand to the new hand. The betting system was checked to ensure that the bet amount was correctly duplicated and that the player's available money was reduced by the amount of the second bet.
Each hand was then played through to completion to verify that they operated independently, with each hand receiving its own cards based on its own strategic decisions, and that payouts were calculated separately for each hand based on its individual outcome against the dealer. This independence is crucial for correct gameplay, as one hand might win while another loses, or both might push, and each result must be properly accounted for in the player's final balance.

Visual Studio Debugger Utilization
The integrated debugger in Visual Studio proved invaluable for investigating complex issues that were difficult to diagnose through print statements alone. Breakpoints were strategically placed at critical junctures in the code, such as immediately before and after key state modifications, allowing execution to be paused at precisely the right moments for inspection. The debugger's watch windows and variable inspection capabilities enabled real-time examination of data structure contents, particularly useful for verifying that the PlayerInfo vectors were being correctly maintained during split operations.
The step-by-step execution feature allowed careful tracing through complex algorithms such as the Ace valuation logic and the strategy table lookup process, making it possible to observe exactly how values changed as the algorithm progressed. The call stack view helped understand the sequence of method calls leading to particular execution points, which was especially helpful when investigating unexpected behavior that originated from interactions between multiple components.

2.4 OOP vs Procedural Comparison

This project serves as an excellent demonstration of the advantages that object-oriented programming provides for complex systems with multiple interacting components and requirements for extensibility. A comparative analysis reveals why the OOP approach is particularly well-suited for this type of application.

Object-Oriented Programming Advantages
Encapsulation provides one of the most significant benefits in this implementation. Each class maintains complete control over its internal state and exposes only the operations that other components legitimately need to perform. PABTable, for instance, keeps all details of card deck management, player hand storage, and bet tracking completely private, exposing only the well-defined methods specified in the ITable interface. This encapsulation means that the internal representation of these data structures could be completely redesigned without affecting any code outside of PABTable itself, as long as the public interface behavior remains consistent.
Polymorphism enables the system to work with abstractions rather than concrete implementations, providing tremendous flexibility. PABGame interacts with the table through the ITable interface and with players through the IPlayer interface, without any knowledge of or dependency on the specific concrete classes being used. This means that entirely different implementations of these interfaces could be substituted into the system without modifying PABGame's code. A network-based multiplayer table implementation or an AI player using advanced machine learning techniques could seamlessly integrate into the existing framework. This polymorphic design also facilitates comprehensive testing, as mock objects implementing the same interfaces can provide controlled, predictable behavior for systematic unit testing.
Inheritance allows the BaseRules class hierarchy to maximize code reuse while supporting variations. Each variant class overrides only the specific methods that differ from the base implementation, automatically inheriting sensible defaults for all other parameters. This selective override mechanism eliminates code duplication and ensures that improvements or bug fixes to shared behavior automatically benefit all variants. The reusability extends beyond individual classes to entire components; replacing PABPlayer with a different

implementation, whether a human player interface or an alternative AI strategy, requires no changes to PABGame or PABTable, demonstrating the loose coupling that OOP enables.

Hypothetical Procedural Implementation
A procedural implementation of the same Blackjack system would face significant challenges. Without encapsulation, all data structures would likely be global or passed extensively between functions, creating tight coupling where changes to one part of the system ripple through many functions. The lack of polymorphism would make it difficult to support different player strategies or rule variants without extensive conditional logic checking which variant is active and branching accordingly. This conditional branching would be scattered throughout the codebase rather than being cleanly separated into different classes, making the code harder to understand and maintain.
Adding new rule variants would require modifications to existing functions rather than simply creating a new derived class, violating the open-closed principle and increasing the risk of introducing bugs into previously working code. Code duplication would likely be more prevalent, as similar operations for different game components couldn't be easily unified under common interfaces. Testing would be more difficult without the ability to create mock objects that implement standard interfaces.
While a procedural approach might offer marginal simplicity for a minimal implementation, any system of this scope and complexity benefits substantially from OOP's organizational principles. The small overhead of virtual function calls is completely negligible for an application where the bottleneck is human interaction rather than computation. For a system with multiple components, state management requirements, variant support, and extensibility needs, object-oriented design is unequivocally the superior choice.

2.5 IDE Usage Evaluation

Development was conducted using Microsoft Visual Studio 2022, which provided a comprehensive integrated development environment specifically tailored for C++ development. The IDE's extensive feature set significantly enhanced productivity and code quality throughout the project lifecycle.

Key Features and Their Impact
The integrated debugger represented perhaps the most critical tool for development and problem-solving. Its ability to set breakpoints, step through code execution line by line, and inspect variable contents in real-time made it possible to understand complex behaviors and identify subtle bugs that would be extremely difficult to diagnose through other means. The watch windows allowed monitoring of specific variables or expressions throughout execution, while the call stack view provided essential context about how the current execution point was reached. The debugger was particularly valuable for investigating the object slicing issue and verifying the correct operation of the split functionality.
Project management features streamlined the organization of the substantial codebase comprising multiple header and implementation files. The Solution Explorer provided a clear hierarchical view of all project files, while the IDE automatically managed dependencies and compilation order. Changes to header files automatically triggered recompilation of dependent implementation files, ensuring that the built executable always reflected the current state of the code. Build configurations allowed easy switching between debug and release modes, optimizing for either debugging capability or execution performance as needed.
Real-time error detection highlighted compilation errors and warnings immediately as code was written, often before the file was even saved. This immediate feedback loop dramatically reduced the time between introducing an error and recognizing it, preventing the accumulation of multiple related errors that can occur when problems go undetected for extended periods. Warnings about potential issues, such as unused variables or implicit type conversions, helped maintain code quality and catch potential bugs before they manifested as runtime problems.

.
Potential Enhancements
While the IDE proved highly effective, several additional capabilities could have further improved the development process. Integration with version control systems like Git directly within the IDE would have enabled easier tracking of changes and experimentation with

alternative implementations through branching. Static code analysis tools could have identified potential issues such as memory leaks, uninitialized variables, or inefficient algorithms before runtime testing. The ability to configure and easily switch between multiple build configurations for different rule variants might have streamlined testing of the various game modes. Nevertheless, even with these potential enhancements unconsidered, the IDE significantly contributed to project success and development efficiency.

2.6 Critical Code Evaluation

A critical evaluation of the final implementation reveals both significant strengths that validate the architectural decisions made and areas where future improvements could enhance functionality, robustness, or user experience.

Architectural Strengths

The architecture demonstrates excellent organization with clear separation of responsibilities across well-defined components. Each class has a focused purpose and maintains proper encapsulation of its internal state and implementation details. The extensive use of interfaces facilitates understanding of component interactions and makes the codebase approachable for new developers, who can comprehend the system architecture by examining the interface definitions before diving into implementation specifics.

The polymorphic design enables genuine extensibility, as demonstrated by how easily the rule variants were implemented. The inheritance-based approach to game rules allows modifications to game parameters without touching any of the core game logic, validating that the abstraction boundaries are well-chosen. The code robustly handles complex edge cases, including multiple splits creating numerous hands for a single player, Aces with context-dependent values, and various rule configurations, all while maintaining correct game state and properly calculating outcomes.

The comprehensive Doxygen documentation provides valuable context and explanation throughout the codebase, making it significantly easier to understand both high-level architecture and low-level implementation details. This documentation investment will pay dividends if the code requires future maintenance or extension.

Areas for Improvement

Error handling currently relies primarily on assertions, which are valuable for catching programming errors during development but are disabled in release builds. A more robust approach would incorporate exception handling for runtime errors that might occur in production, such as the deck running out of cards due to an unexpectedly long sequence of splits, or file I/O errors if save game functionality were added. Graceful degradation and meaningful error messages would improve the user experience compared to assertion failures or undefined behavior.

The absence of unit tests represents a significant gap in code quality assurance. The interface-based architecture would facilitate comprehensive testing through dependency injection and mock objects, but this opportunity was not exploited during development. A proper test suite would verify that complex algorithms like hand value calculation handle all edge cases correctly and would catch regressions when code is modified. Automated testing could verify the strategy table produces expected decisions for known situations and validate that split handling, insurance betting, and payout calculations all function correctly.

The current text-based interface, while functional, provides a minimal user experience. A graphical interface would significantly enhance usability and appeal, displaying cards visually, providing intuitive controls for player decisions, and presenting game state more clearly than text output can achieve. Even within the constraint of a console interface, improvements such as color-coding and better formatting could enhance readability.

Some game parameters remain hardcoded despite the configurable rules system. The maximum number of players is defined as a constant kMaxRealPlayers rather than being derived from the rules configuration. While this is reasonable for preventing resource exhaustion, a more flexible approach might allow rules to specify player limits as well. The lack of persistence means that all game progress is lost when the application closes. Adding save game functionality would allow players to maintain bankrolls across sessions and resume interrupted games.

Future Extension Possibilities
Several natural extensions to the current implementation suggest themselves. Implementing interactive human players would transform the system from a simulator into an actual game, requiring minimal changes to PABGame thanks to the polymorphic design. Multiple AI difficulty levels could range from a random decision maker for beginners through the current basic strategy implementation for intermediate players to an advanced card counting strategy for expert-level play.
Statistical tracking would add value by recording win percentages, total money won or lost, longest winning streaks, and other metrics that players find engaging. Additional game variants beyond the three currently implemented could include popular casino variations like Spanish 21, Blackjack Switch, or Super Fun 21, each with its own rules class inheriting from BaseRules and overriding the appropriate parameters.

Overall Assessment
In conclusion, the implementation successfully fulfills all project requirements and demonstrates sophisticated application of object-oriented programming principles. The identified areas for improvement are largely enhancements that would extend functionality or improve user experience rather than corrections to fundamental architectural problems. The core design is sound, the code is well-structured and properly documented, and the system handles the complexities of Blackjack gameplay correctly. The project serves as a solid foundation that could support future enhancements while requiring minimal modification to existing code, validating the extensibility that was a primary design goal.

5. BIBLIOGRAFÍA

Capítulo 01

How to Play Blackjack | The Venetian Resort Las Vegas. (s.f.). Luxury Hotel in Las Vegas | The Venetian Resort Las Vegas. https://www.venetianlasvegas.com/resort/casino/table-games/how-to-play-blackjack.html#:~:text=Blackjack%20Rules&amp;text=Players%20receive%20all%20cards%20face,and%20the%20wager%20is%20lost.

Blackjack Cheat Sheet: Ultimate Blackjack Strategy Chart. (2024, 30 de octubre). ProfitDuel: Smart Sports Betting Tools, Calculators And Strategies | ProfitDuel. https://www.profitduel.com/blackjack-cheat-sheet

BlackJack | Play Online for Free | Washington Post. (s.f.). Arena Washington Post. https://games.washingtonpost.com/games/blackjack