SWEN30006

Assignment 2

[Mon16:00] Team 02:

Haoguang Zhou 1344871

Baimin PAN

Yudong Luan

Plain text pages <= 6 pages

Image == 11

(Format the pictures separately to display on the entire page)

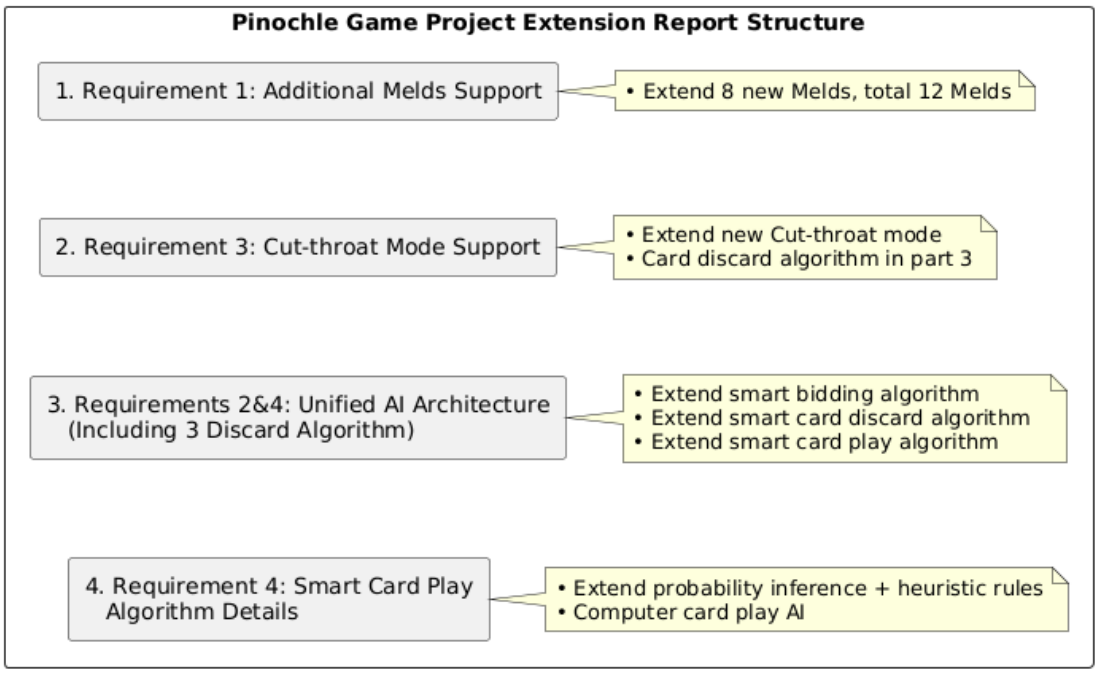
The domain/design class diagram is not here. It is in the folder

Assumptions for the Design Class Diagram:

1. This class diagram only illustrates the components that are added or modified compare with the original system. Therefore, it includes only the new classesand those existing classes that are changed and their relationships between each classes, which means relationships between unaffected classes in the original system are not contained.
2. Although dependency relationships are considered during the analysis process, UML design class diagrams only draw important dependency relationships (such as create) and hide dependency relationships that affect understanding.
3. Multiplicity with only one-way access permissions is also considered as two-way multiplicity in order to show the logical design.
4. The frequent use of composition relationships in this diagram is intended to clearly show the design intent and structural relationships of the entire system. It does not impose any limitations on the system's future scalability or modifications.
5. The symbol usage in the UML domain and design class diagrams focuses on conceptual design, allowing for the existence of appropriate distinctions from the code (such as the use of aggregation relationships).

Assumptions and Points to Note:

1. Considering that the expansion strategies of computer players are scattered across multiple project requirements, this article is not written in the order of business requirements but is classified and explained according to the following three dimensions: combination card expansion, game mode expansion, and AI/ computer decision-making expansion
2. Although some of the titles are "Application of GoF Design Pattern", this article also covers the non-GOF but equally appropriate design patterns that are actually adopted.
3. In order to better present our ideas, some UML has been simplified and expressed in an non-standard way



Task 1- Support for Additional Melds:

The original system's combination card logic uses hard-coded logic. If it is directly extended to twelve complete combinations in the main class, the addition of uncertain card types in the future will make the main class bloated and unclear in its responsibilities. Considering the project requirements, it is hoped that new combinations can be easily expanded in the future. An extremely flexible expansion method that does not require code modification (even hot-swappable) is worth Considering.

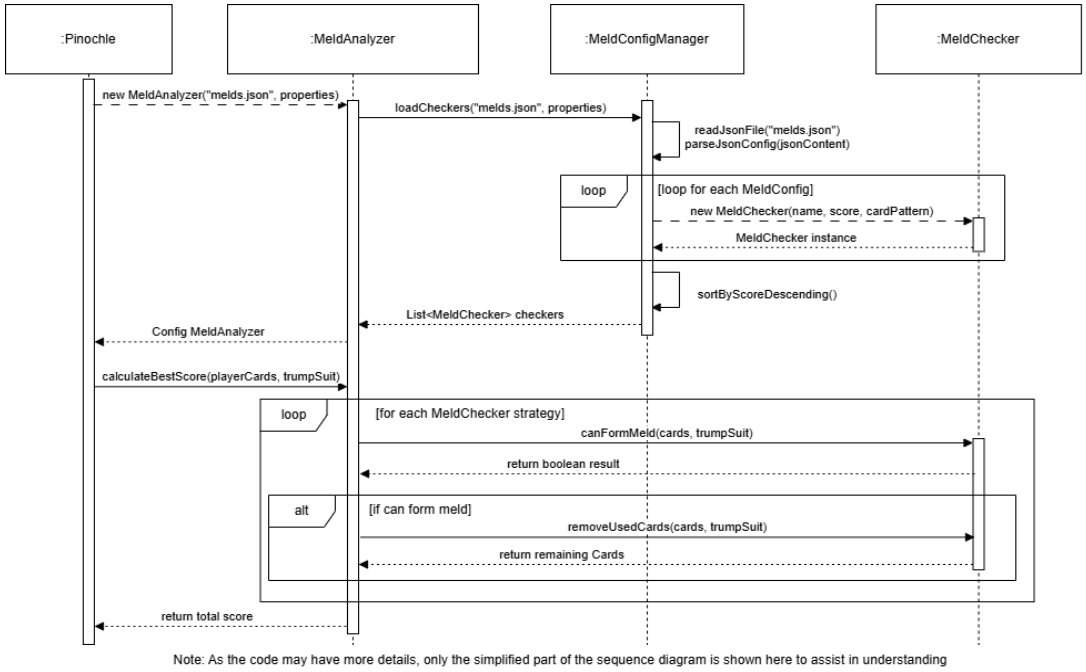


Figure 1. A brief sequence diagram showing how the card combination extension works

Application of GoF Design Pattern

The expansion of combination cards mainly adopts the combination strategy Pattern and the factory method Pattern. The strategy Pattern is reflected in the Design of MeldChecker, although MeldChecker is a universal strategy framework. However, based on the rules of different hand types (defined in JSON), multiple specialized MeldCheckers representing different hand type checking strategies can be created (Created by MeldConfigManager using factory methods), while MeldAnalyzer combines multiple MeldChecker strategies (MeldChecker sets). And use strategies such as greedy selection based on scores or backtracking algorithms (optimal) to calculate the total score of the combination cards in the hand. (Note: This is also the idea of the configuration-driven mode)

GRASP Principle Analysis

Information Expert and Creator

The expansion of combo cards strictly adheres to the information expert principle. MeldChecker has all the information about the rules of each combo card type for each strategy and is thus responsible for handling specific combo cards. The MeldAnalyzer has all the strategy objects of the combinations, and thus is responsible for calculating the total score of the combination cards. The MeldConfigManager possesses the deck rule configuration information represented by JSON parsing (MeldChecker initialization information), and thus undertakes the responsibility of creating the business object MeldChecker

Low Coupling and High Cohesion

MeldAnalyzer does not rely on the implementation details of the combination cards at all and interacts through a unified MeldChecker object. The configuration file, on the other hand, significantly reduces code-level dependencies. changes to the combination card rules do not require modifying any source code or classes. From the perspective of high cohesion, MeldAnalyzer focuses on the algorithmic strategies and total score calculation of combo cards. MeldChecker focuses on the rule processing of a single deck type, while MeldConfigManager focuses on configuration parsing and object creation (the work is completely centered around JSON), which makes the responsibilities of the three new objects highly concentrated and almost completely independent.

Protected Variations

The general use of the combination strategy Pattern to parse and assemble JSON enables the most easily variable card rules (in future) and combination extensions to be isolated outside the program layer in a hot-swappable and JSON configuration file manner. This ensures that expanding or deleting card rules does not result in any internal system/code modifications.

Alternative strategy for task 1

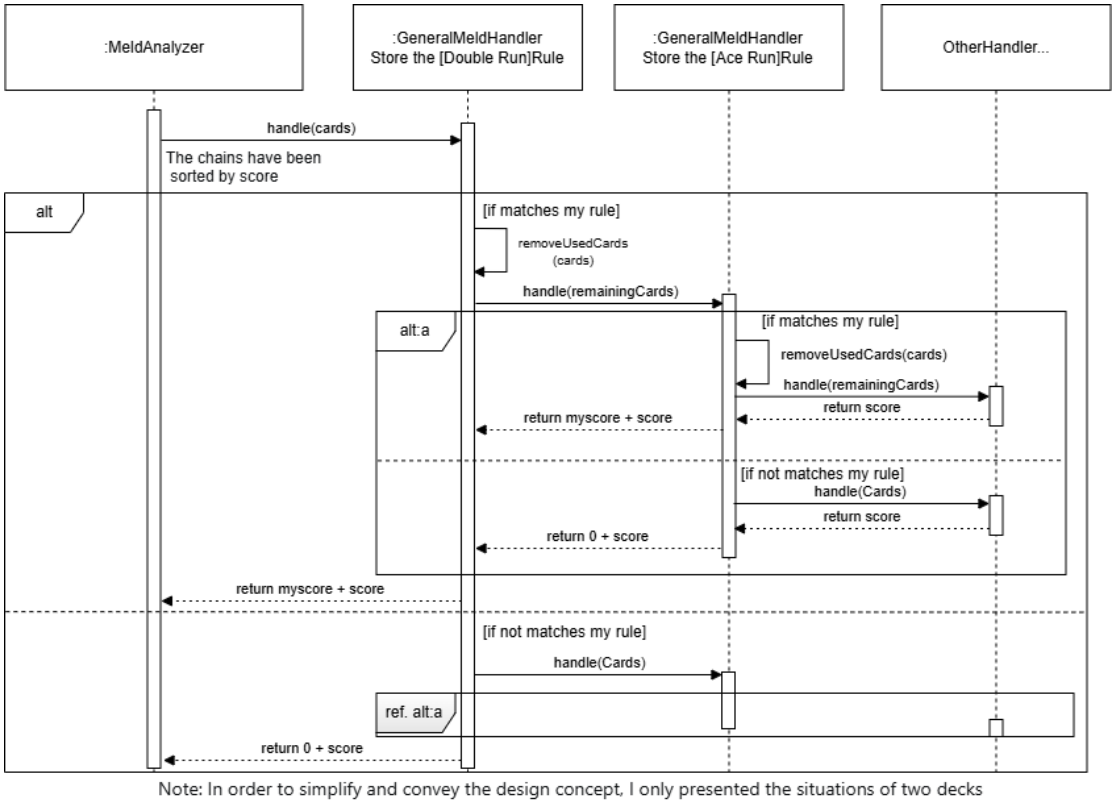


Figure 2. A simplified sequence diagram of the alternative to Requirement 1, the chain of responsibility solution

Initially, Project Design adopted the responsibility chain to expand and handle new combination cards, using common nodes or creating separate nodes for card types, and greedily forming calculation chains based on scores. According to the chain, the program can successively search for the matching hand type and continue to search for the next potential hand type after removal, and finally return the cumulative (similar to recursion) total score. The chain of responsibility can seemingly elegantly satisfy GRASP (for example, Low Coupling, High Cohesion and Protected Variations, etc.), especially performing well in Polymorphism

Reasons of the Alternative 1 is Not Selected

However, the reason for not adopting the chain of responsibility lies in the potential illegal GRASP risks brought about by business logic

1. Naive linear/greedy assumption: The chain of responsibility Pattern is inherently suitable for linear and early-stopping tasks. However, if there are slight changes in business requirements (such as calculating the global optimal combination, or if there are repetitions in combination rules or depends on melds sequences), the advantages of the chain of responsibility will rapidly weaken or even disappear. This kind of change will cause significant potential degradation risks of the Protected Variations principle and may lead to the deterioration of the Low Coupling and High Cohesion principles as well. However, the currently adopted strategy combination and configuration-driven scheme is not subject to the linear assumption and can elegantly use any card combination algorithm without being restricted by patterns

2. Unclear division of responsibilities: The calculation of the combined score of the responsibility chain relies on the card meld structural information and rules of the combined cards. The responsibility chain nodes need to handle both the card meld rules and calculate the scores, and even the control of the links. The handler with the minimum granularity still has the risk of unclear responsibility division, which may lead to potential deterioration risks of GRASP (such as High Cohesion). However, the current approach adopted by this project strictly classifies the business responsibilities of different classes and objects

Task 3- Support a Cut-throat Mode

The original system adopts a fixed game process, and all game logics and GUI logics are directly hard coded in the main game class. If the complex logic of the Cut-throat pattern (such as card flipping selection, additional allocation, and 24→12 card selection) is directly added to the Pinochle main class, it will cause the already bloated main class (the God class) to continue to undertake more functions that are not its core responsibilities. To avoid refactoring the entire project at once, this extension adopts a progressive refactoring strategy: building a pluggable game mode framework, and first separating the Cut-throat mode from the core system to provide a unified extension basis for subsequent porting of other game phases (such as bid) and new modes. Furthermore, considering the extreme complexity of the original system, an abstraction layer that provides services to the outside is also worth considering.

Application of GoF Design Patterns

This extension builds a plug-in game mode management architecture similar to the modern game engine DLC/Workshop extension system.

1. The Plugin Pattern combined with the Observer Pattern variant: ExtensionManager is responsible for the registration, activation and lifecycle management of the game mode. When the game phase changes, the event is broadcast to all active modes using the notifyPhase method, and each mode decides whether to respond based on its own isApplicable logic. This mechanism makes it extremely simple to add new game modes. Just implement the GameMode interface and register.

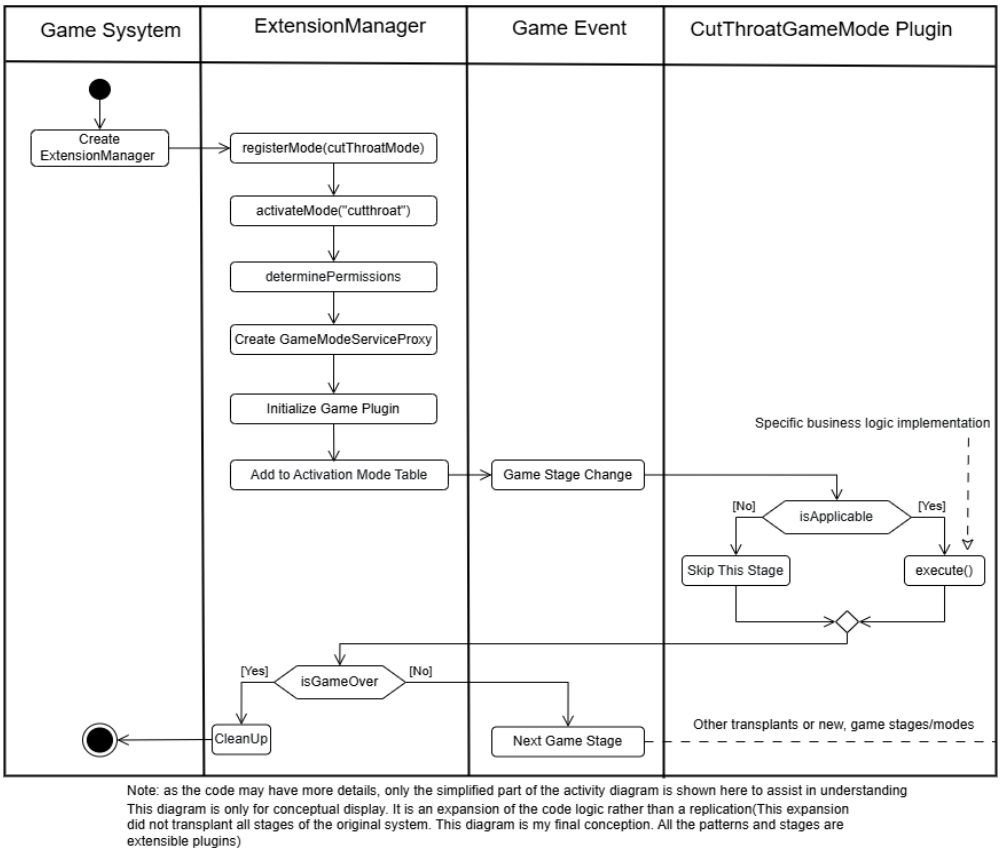


Figure 3. A brief activity diagram of the architecture for Requirement 3, demonstrating our design concept at the architectural level

2. Proxy Pattern: GameModeServiceProxy serves as the access proxy for the core system, providing standardized atomic service interfaces for extension modules and filtering out the complexity of the internal system. This enables extensions like CutThroatGameMode to focus on the implementation of business logic without having to deal with the details of the underlying system.

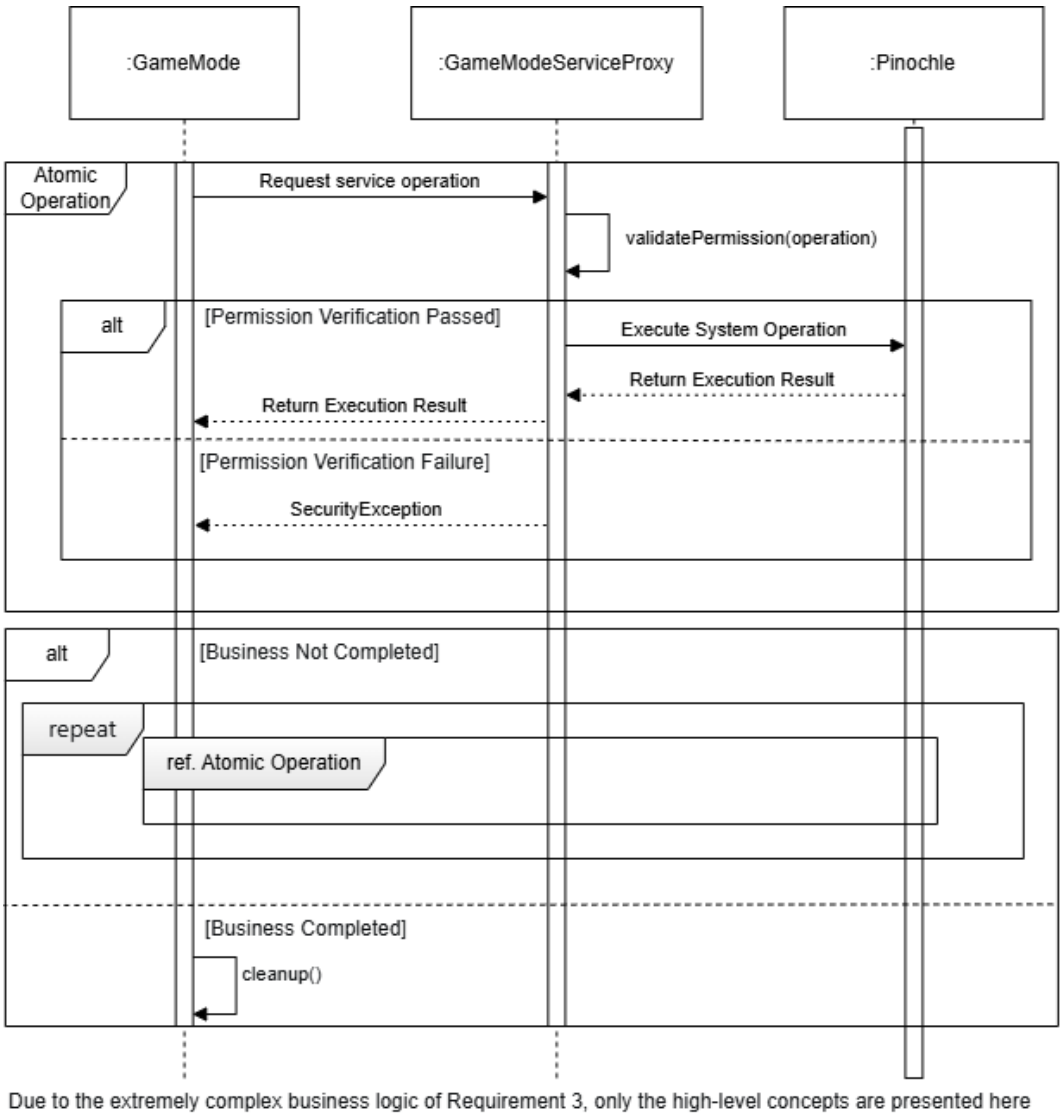


Figure 4. A sequence diagram of how our proxy pattern works (simplified version), showing how our proxy pattern abstracts atomic methods, isolates internal systems, and enables the extended pattern to focus on business logic

3. Access Control Pattern: *Considering that different extensions should have different access permissions, the determinePermissions method in ExtensionManager creates the corresponding permission set based on the Pattern type. GameModeServiceProxy verifies permissions each time a method is called to ensure that the extension module can only access the functions it is authorized to.*

GRASP Principle Analysis

Controller, Indirection and Pure Fabrication

ExtensionManager is responsible for coordinating the lifecycle management, event distribution and permission allocation of all game modes. It undertakes the control responsibilities at the system level and avoids the dispersion of control logic among various modes. ExtensionManager and GameModeServiceProxy (which also

provide an indirect access mechanism) are both fictional classes created to address specific design issues. They do not directly correspond to real-world domain concepts but are designed to meet the technical requirements of low coupling and separation of responsibilities.

Low Coupling and High Cohesion

The core game system and specific game modes are completely decoupled through the standardized GameMode interface. The addition, removal or modification of modes does not affect the main game process. GameModeServiceProxy further isolates the dependency between the extension and the core system through the mechanism of broadcasting events rather than direct invocation for communication. The internal functions of each component are closely related and have clear goals. ExtensionManager focuses on pattern management, CutThroatGameMode focuses on the business logic of harsh patterns, and GameModeServiceProxy focuses on permission control and service proxy.

Polymorphism and Protected Variations

Polymorphism is achieved through the GameMode interface, and the system can uniformly handle different types of game modes without the need to understand the specific implementation. ExtensionManager uses the GameMode interface to extend games and supports dynamic switching of game modes at runtime. The plugin architecture effectively isolates the changes of the extension game mode at the extension layer, protecting the core system from the synchronous modifications of internal code and logic caused by the expansion (or transplantation and addition) of the game mode.

Alternative strategy for task 3

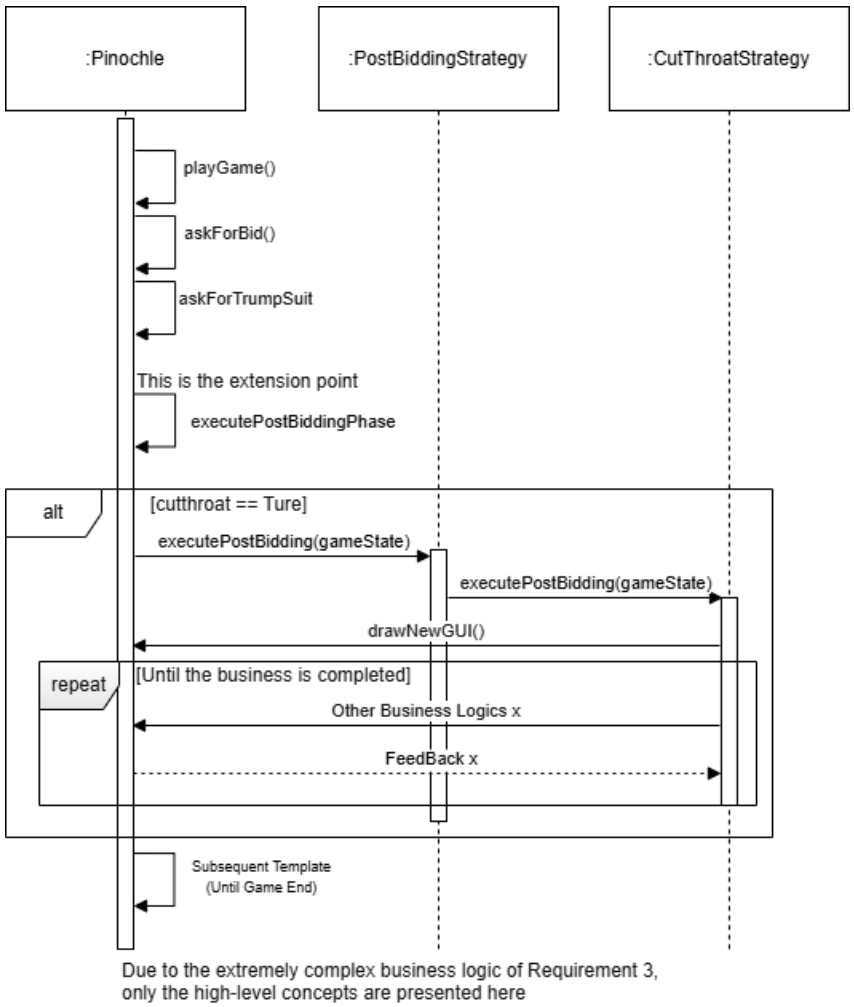


Figure 5. An alternative to Requirement 3, a brief sequence diagram showing how the template + strategy pattern works

An intuitive and effective approach is to use a combined design of the strategy pattern and the template method pattern. This scheme defines the template method skeleton of the game process in the Pinochle main class. At the same time, provide a strategy interface (CutThroatStrategy implements the PostBiddingStrategy interface) for specific stages that need to be extended (such as the Post-Bidding stage). The intuitive and consistent game flow and flexible extension interfaces enable this method to perform well in GRASP principles such as High Cohesion, Polymorphism and Indirection

Reasons of the Alternative 3 is Not Selected

1. Serious violations of the principles of Low Coupling and Protected Variations: Due to the excessive complexity of the original system, without reconstructing the overall game framework, the template method cannot truly decoupler the strategy implementation directly from the business logic of the core system. The implementation of the business logic of specific strategies such as CutThroatStrategy requires direct access to the internal state and methods of the Pinochle class, which means that the implementation of strategies must have a deep understanding of the internal structure of the core system. This tight coupling leads to an extremely high risk of change propagation: Any adjustment to the core system API (such as changes in method signatures, refactoring of internal data structures, or optimization of processing flows) will force all related strategies to be modified accordingly. More seriously, the implementation errors of the strategy may directly undermine the state consistency of the core system because the strategy has direct access to the key system objects. The currently adopted scheme, the atomic operation interface of the service agent layer effectively solves these coupling problems. Moreover, when refactoring, API adjustments or performance optimizations occur internally in the core system, as long as the interface of the agent layer remains stable, all extension modules do not need to be modified.

2. The blurring of the boundaries of the Information Expert principle: In the strategy + template scheme, the boundary between information ownership and processing responsibilities becomes potential blurred. The template method requires an understanding of the execution conditions, result processing and exception situations of the strategy, while the strategy requires an in-depth understanding of the state management, data structure and processing flow of the core system. This cross-dependence has led to the confusion in the application of the information expert principle. With the addition of more strategies, information dependencies will form complex networks, making the system behavior difficult to predict and maintain. In contrast, the current plugin mode clearly defines the information access boundaries through GameModeServiceProxy, ensuring a clear separation of responsibilities.

Task 2, 3(Discard Algorithm) & 4- Unified Intelligent Decision Making

Although the project requirements spread the functional expansions of computer players (intelligent bidding, intelligent discarding, and intelligent playing) among requirements 2, 3, and 4, essentially, these requirements are all aimed at expanding the decision-making functions of computer players. If independent AI strategy classes are implemented separately for each requirement, it will lead to dispersed responsibilities, repetitive code, and be difficult to maintain and scale when the business logic changes. Furthermore, considering that the activation of AI functions are highly bound to configuration files and there may be multiple demands for the expansion of AI strategies in the future, we need to build a modular, highly configurable and easily scalable unified intelligent decision-making architecture to centrally handle all decision-making requests from computer players. Since AI decision-making needs to be based on the internal state of the game system and at the same time maintain the independence of the AI system and core system, designing a data exchange protocol independent of the main system and a data message format without side effects is also a crucial consideration.

Application of GoF Design Patterns:

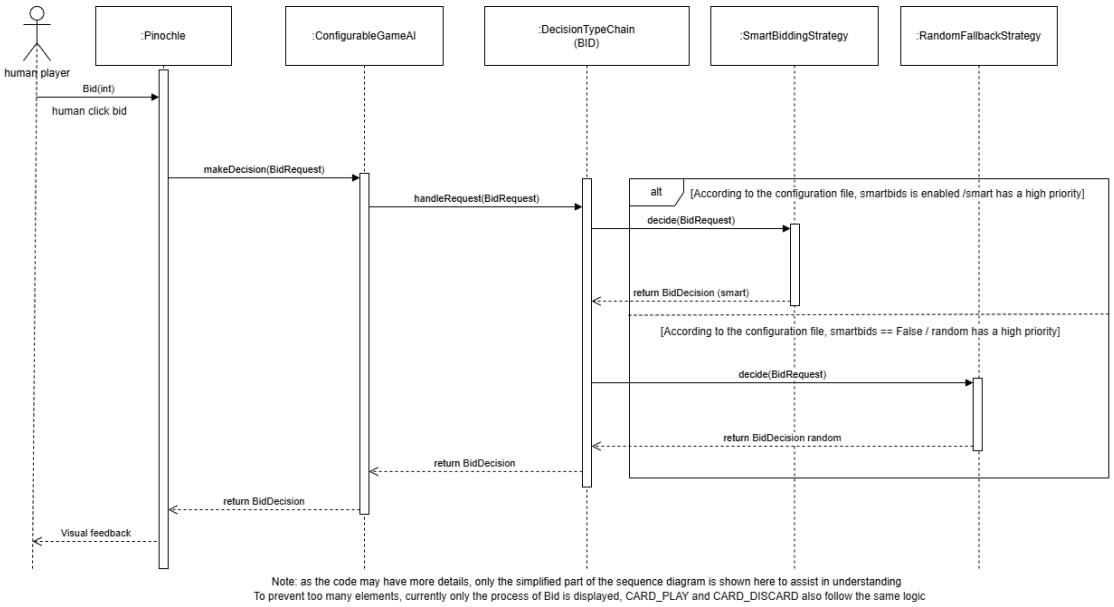


Figure 6. Considering the complexity of presenting all elements at once, this is a brief example of how our AI system works, demonstrating how BID requests are processed.

1. Two-Dimension Chain of Responsibility for AI Decision: This extension adopts an innovative two-dimensional chain of responsibility model (Chain of responsibility forest) architecture to uniformly handle all AI decision requests: The first-dimension routes based on the decision type DecisionType (such as BID, CARD_PLAY, etc.). The second dimension uses the responsibility chain combination strategy pattern (DecisionStrategy interface) and sorts the strategies according to their priority to form a processing chain, providing a RandomFallbackStrategy as a fallback.

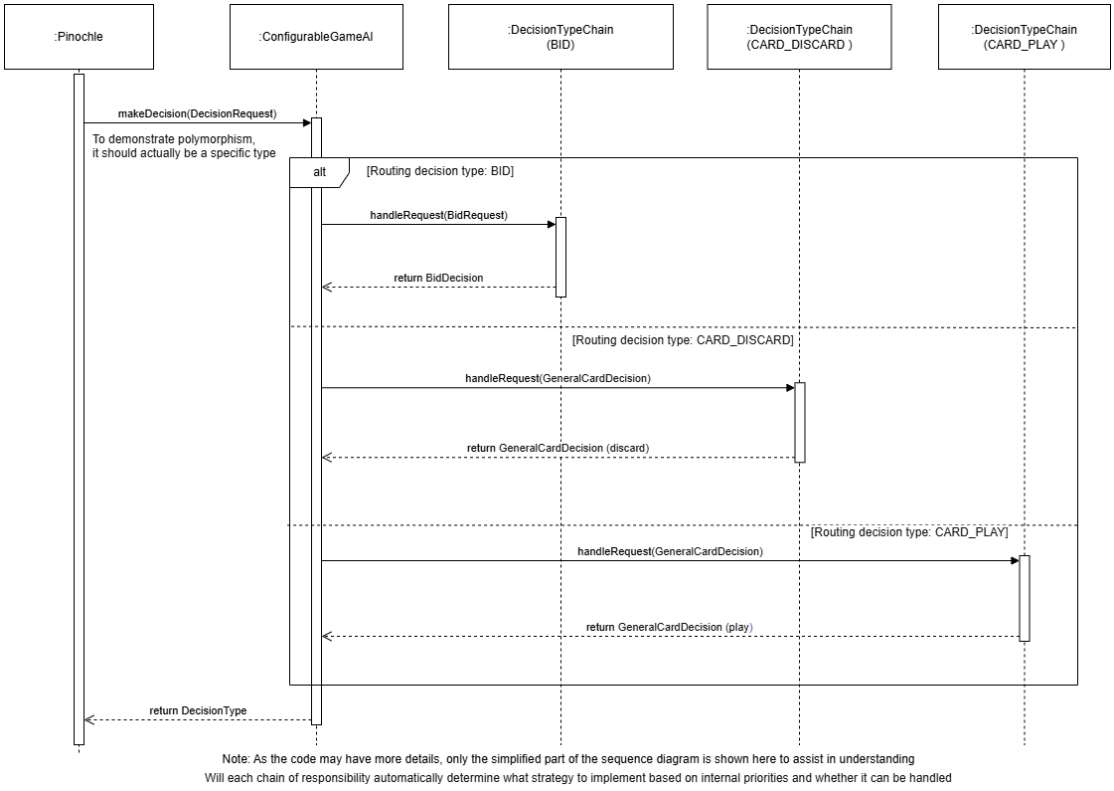


Figure 7. This is part of how our AI system works, mainly demonstrating how requests are routed to specific chains

2. AI Construction via Builder Pattern: Since the factory pattern is difficult to elegantly meet the complex AI construction process, this extension adopts the builder pattern. AIBuilder utilizes ChainBuilder to build a single decision chain and stream the construction of customized AI system ConfigurableGameAI (the builder will directly construct customized computer AI based on the configuration file).

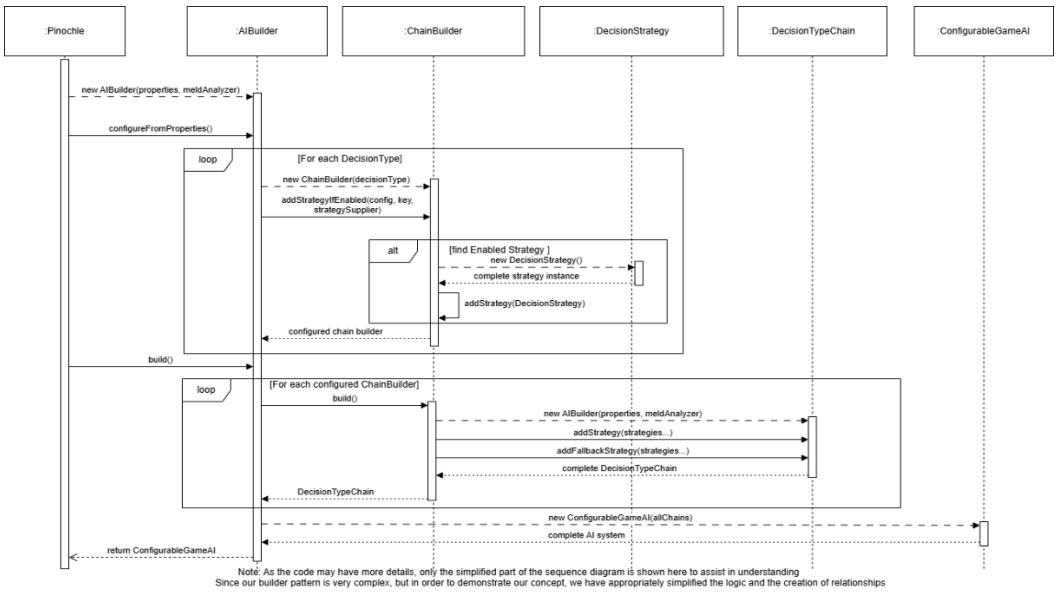


Figure 8. This is a brief conceptual sequence diagram showing how our Builder pattern builds, which can demonstrate the general process of our system building

3. Data exchange protocol based on DTO mode: To ensure the complete independence of the AI system and the data security of the main system, this extension designs a complete request and response data exchange protocol. DecisionRequest and its subclasses (BidRequest, GeneralCardDecisionRequest) encapsulation AI decisions needed to complete parameter information and immutable snapshot of the system state data (avoid modifying the state of the primary system accident). Finally, the AI will encapsulate the Decision result as Decision and its subclasses (BidDecision, GeneralCardDecision) and return it as the decision result. Since AI services are entirely based on request and response protocols, the AI module has a high degree of independence almost independent of the main system and even supports remote provision of AI services by extending the protocol format. It is important to note that only need GeneralCardDecisio and GeneralCardDecisionRequest protocol interfaces can satisfy all the card related requests and decisions, this is because the card operating essentially from the card set A mapping to the card set B operation.

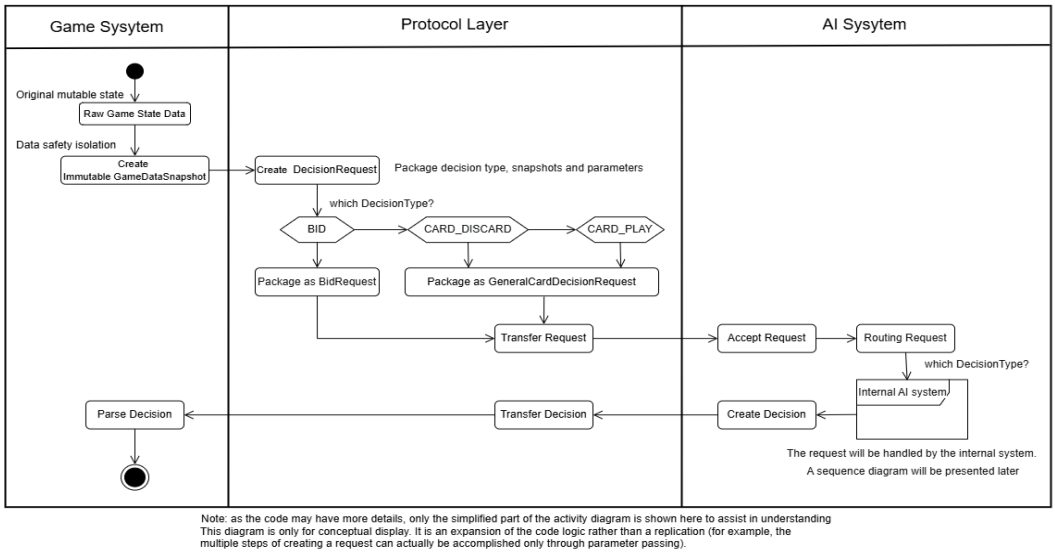


Figure 9. It mainly shows how our data exchange protocol conceptually works, a conceptual activity diagram

GRASP Principle Analysis

Controller and Pure Fabrication

Components such as DecisionTypeChain, AIBuilder, and ChainBuilder are not essentially directly mapped to business domain objects. Instead, they are created based on pure technical requirements (responsibility division, solving complex AI decision-making and control problems). While ConfigurableGameAI, as a unified decision controller, undertakes the responsibilities of receiving all AI decision requests, routing and distribution, etc. DecisionTypeChain, as a policy execution controller, manages policy selection, priority ranking, and fallback mechanisms within a specific decision type. This centralized control approach avoids the decision processing logic being scattered across various AI strategies or business classes

Low Coupling and High Cohesion

The complete decoupling between the AI system and the main game system (with almost no direct dependency) has been achieved through the DTO data exchange protocol (DecisionRequest/Decision). The AI decision-making process is entirely based on immutable data snapshots, eliminating cross-system state dependencies and side effect risks. Meanwhile, each component demonstrates extremely high hierarchical cohesion, and the strategy class (DecisionStrategy interface) is only responsible for business logic. ConfigurableGameAI and DecisionTypeChain focus on policy management and execution coordination. Builder focuses on creating AI and configuration parsing

Polymorphism and Protected Variations

The DecisionStrategy interface unifies the processing methods of different intelligent strategies, while the GameAI interface abstracts the unified behavior of intelligent AI. The application of multiple polymorphisms makes the expansion of future AI modules easy and effortless. The two-dimensional chain of responsibility architecture isolates the decision type and strategy algorithm at different logical layers respectively. As long as the interface requirements are met, modifying the code of the AI strategy will not have any impact on the intelligent AI architecture. The Request and response data exchange protocol isolates the AI module itself from the main system at different business layers and always completes it through stable interfaces and protocols. As long as the protocol is followed, any modification of the AI module and the main system code will not affect each other. The only thing that needs to be changed is the enumeration of the new decision type and the corresponding request-response protocol. Based on this, the extension meets the design concept of extension rather than modification in multiple aspects.

Alternative Design for AI System

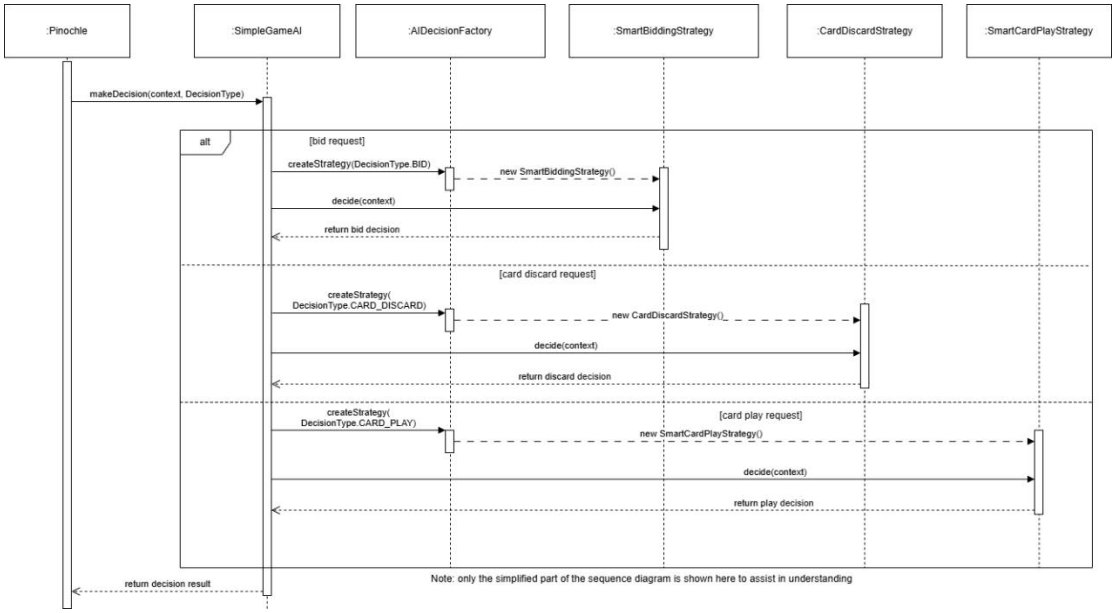


Figure 10. A brief sequence diagram showing how our other AI system/module was implemented and worked in the early days

A simple and direct approach is to abandon the multi-layer abstraction and intermediate structure of the current method and adopt the classic combination scheme of the simple factory pattern combined with the strategy pattern. AIDecisionFactory directly creates the corresponding policy instances (BID→SmartBiddingStrateg,

CARD_PLAY→SmartCardPlayStrategy, etc.) based on the DecisionType enumeration value. SimpleGameAI serves as a unified decision entry point to receive requests and invoke the policies created by the factory to execute decisions. This scheme performs well in GRASP principles such as Creator, Polymorphism, and Information Exper. The most important thing is that this solution is simple and intuitive enough with low technical complexity, yet it can meet the current project requirements.

Reasons of this Alternative is Not Selected:

1. The defect of the low coupling and Protected Variations principle: The current method is vulnerable to the robustness of policy and main class changes. The strategy pattern used itself has not been completely decoupled from the bloated main class (for example, when capturing the system state), which can lead to changes and additions to the strategy or changes in the main class directly affecting the calling logic and interface format. This solution is sensitive to interface changes and has the risk of modifying interfaces from multiple files. Our current solution fundamentally eliminates the possibility of repeated interface modifications through a unified protocol and data snapshots.

2. The structural defect of scalability: The policy factory solution has the fundamental limitation of single policy binding at the architectural level. Each decision type can only be hard-coded to correspond to one policy instance, and it cannot support the dynamic combination and priority selection of policies. As business requirements evolve, for instance, when strategies need to be adjusted dynamically based on different game modes (such as whether the cruel mode is enabled, which may potentially change the subsequent intelligent card-playing strategy) or according to the game stages. All these changes carry the risk of refactoring the AI extension and incurring technical debt. In contrast, the current DecisionTypechain-composed responsibility chain forest, in conjunction with ConfigurableGameAI for decision type forwarding, can flexibly extend any strategy and make dynamic decision changes

About our AI strategy [AI graph at Next page]

Considering that this is not an algorithm competition, we adopt a simple combination strategy: namely, naive probability inference (1 - (the number of unknown cards that can defeat me ÷ the total number of unknown cards)) + ace heuristic rule. First, the algorithm will receive a card-playing request. It will calculate the expected return of each card and initially select the card with the highest return [(winning probability × points gained from winning) + ((1 - winning probability) × points lost from losing)]. Then the heuristic rule will correct this decision: 1. If it is determined that the probability of losing or winning is extremely low, select the non-ace card with the lowest score. 2. If the current card is a trump card with a low chance of winning, try choosing a non-trump card. This strategy is very simple but much better than random situations, and it can be achieved only with discard information and hand card information. Considering that the current AI architecture is flexible enough and easy to expand, it is possible to consider using the Monte Carlo Tree Search (MCTS) strategy or true Bayesian probability in the future and reduce the priority of this strategy as a fallback strategy.

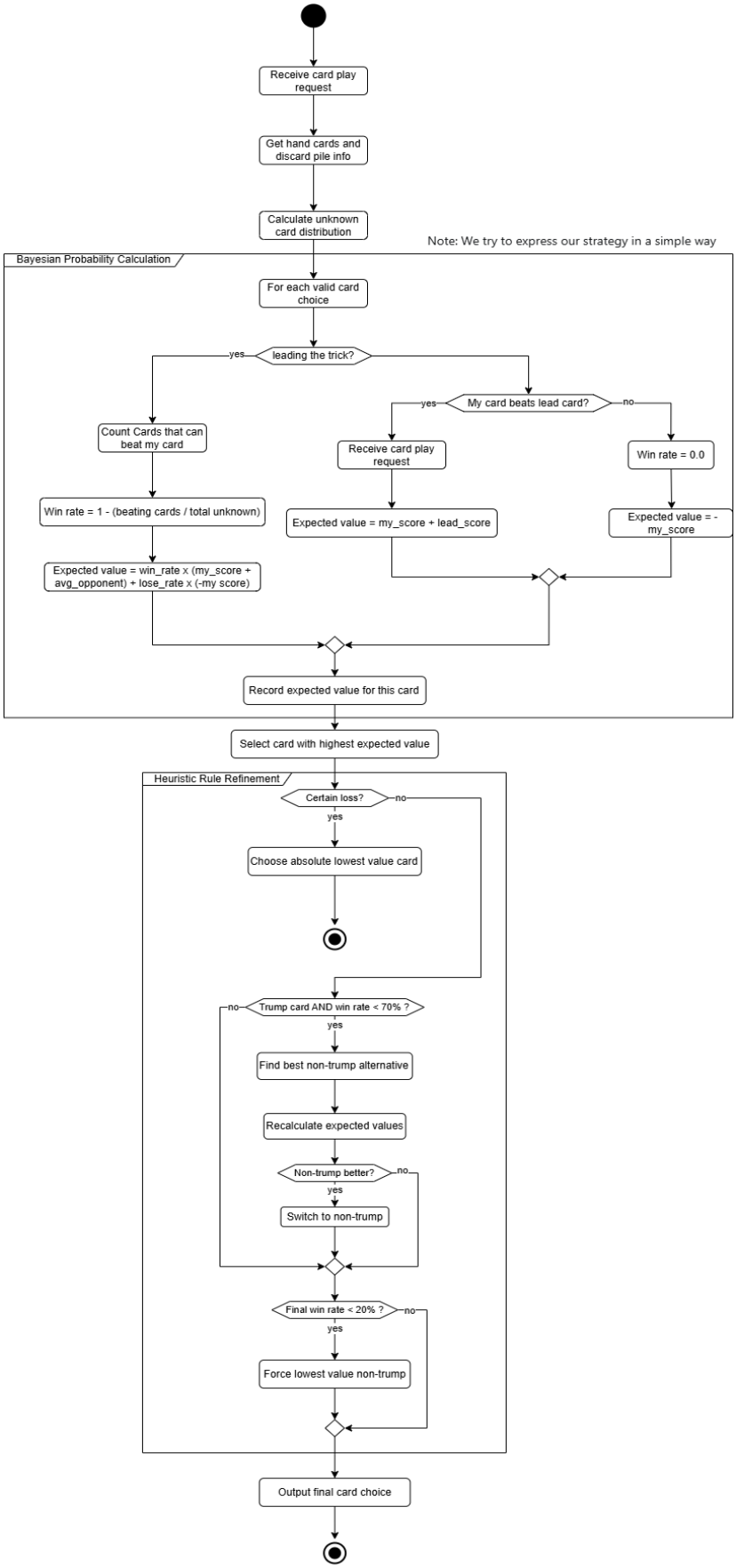


Figure 11. A brief description of our AI algorithm demonstrates the decision-making of relationships and the calculation of probabilities and expectations