ETH zürich

HS 2024

Software Design Specifications

for

Dominion

Version 1.0

Prepared by Nicola Bruhin, Mathieu Martin, Aaron Mazzetta, Philipp Obrist, Nicholas Schaller, Caspar Schucan

doMINIONS

16.10.2024

Table of Contents

Revision History

Name	Date	Release Description	Version
Felix Friedrich	10/15/24	Template for Software Engineering Course in ETHZ.	0.2
doMINIONS	10/25/24	SDS for Dominion	1.0

1. Introduction

1.1 Purpose

<Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SDS, particularly if this SDS describes only part of the system or a single subsystem.>

1.2 Document Conventions

<Describe any standards or typographical conventions that were followed when writing this SDS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

1.3 Intended Audience and Reading Suggestions

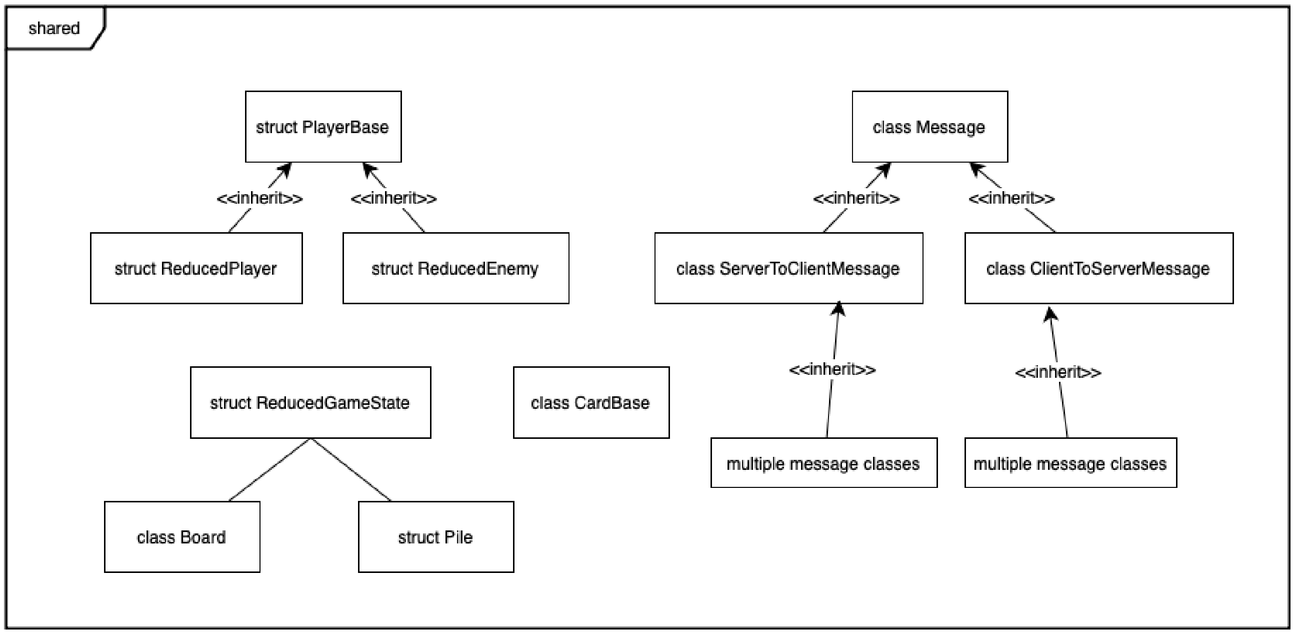
<Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SDS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.>

1.4 Product Perspective

<Describe the context and origin of the product being specified in this SDS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SDS defines a component of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.>

2. Static Modelling

2.1 Package Shared



2.1.1 Class Message

The class attributes are:4
- game_id: string
- message_id: string

The class operations are:
- to_json(): string
- from_json(string): Message

2.1.2 Class ClientToServerMessage : Message

Each message type that is sent from client to server has a corresponding class that inherits from this base class and has the attributes specified in the client-server interface section of this document. Specifically, those are
- GameStateRequestMessage
- CreateLobbyRequestMessage
- JoinLobbyRequestMessage
- StartGameRequestMessage
- ActionDecisionMessage

The class attributes are:
- PlayerBase::id_t: PlayerBase::id_t

2.1.3 Class ServerToClientMessage : Message

Each message type that is sent from server to client has a corresponding class that inherits from this base class and has the attributes specified in the client-server interface section of this document. Specifically, those are

- GameStateResponseMessage
- CreateLobbyResponseMessage
- JoinLobbyResponseMessage
- StartGameResponseMessage
- EndGameBroadcastMessage
- ResultResponseMessage

The class attributes are:

- PlayerBase::id_t: PlayerBase::id_t

2.1.4 struct ReducedGameState

The ReducedGameState functions as a storage for all important information concerning the current state of the game. This can then later be used by the GUI to draw the frames.

The class members are:

- board: Board
- player: ReducedPlayer
- enemies: vector<ReducedEnemy>
- active_player: PlayerBase::id_t

2.1.5 class Board

This class functions as a storage of the board for the client, as well as a parent class for the server side implementation of the board. It reflects the board that's used for the current game.

The class members are:

- victory_cards: vector<Pile>
- treasure_cards: vector<Pile>
- kingdom_cards: vector<Pile>
- trash: vector<CardBase>

The class operations are:

- sold_out_piles(): int

2.1.6 struct Pile

This class is used to implement the different piles of cards that are present on the board.

The class members are:

- card: CardBase::id_t (string)
- count: int

2.1.7 struct CardBase

This class functions as the basis for cards. It is used as storage on the client and used as parent class for the card implementations on the server.

using id_t = string
The class members are:
- id: const CardBase::id_t
- type: const card_type
- cost: const int

The member functions are:
- CardBase(id_t, Card_Type, int): CardBase
- isAction const: inline bool
- isAttack const: inline bool
- isTreasure const: inline bool
- isReaction const: inline bool
- isVictory const: inline bool
- getCost const: inline int
- getType const: inline CardType
- getId const: inline id_t
- Virtual toString const: string

2.1.8 Enumeration CardType

- Action: 0b100
- Attack: 0b110
- Reaction: 0b101
- Treasure: 0b001
- Victory: 0b010

2.1.9 struct PlayerBase

This is a base class for players that is both used as a parent class for the server side player implementation as well as the client side storage implementations of both players and enemies.

using id_t = string
The class members are:
- PlayerBase::id_t: id_t
- victory_points: int
- played_cards: vector<CardBase::id_t>
- gained_cards: vector<CardBase::id_t>
- available_actions: int
- available_buys: int

- available_treasure: int
- current_cards: CardBase::id_t

2.1.9.1 struct ReducedEnemy : public PlayerBase

This is a simplified version of a player that allows the client of a certain player to have some important information about the other players.

The class attributes are:
- hand_cards: int
- discard_pile: pair<CardBase::id_t, int> // CardBase::id_t is top card, int is pile size
- draw_pile_size: int

2.1.9.2 struct ReducedPlayer : public PlayerBase

This is a simplified version of the player that is used to store important information about the Player on the client.

The class attributes are:
- hand_cards: vector<CardBase::id_t>
- discard_pile: pair<CardBase::id_t, int> // CardBase::id_t is top card, int is pile size
- draw_pile_size: int

2.2 Package Client

2.2.1 Class GUIInterface

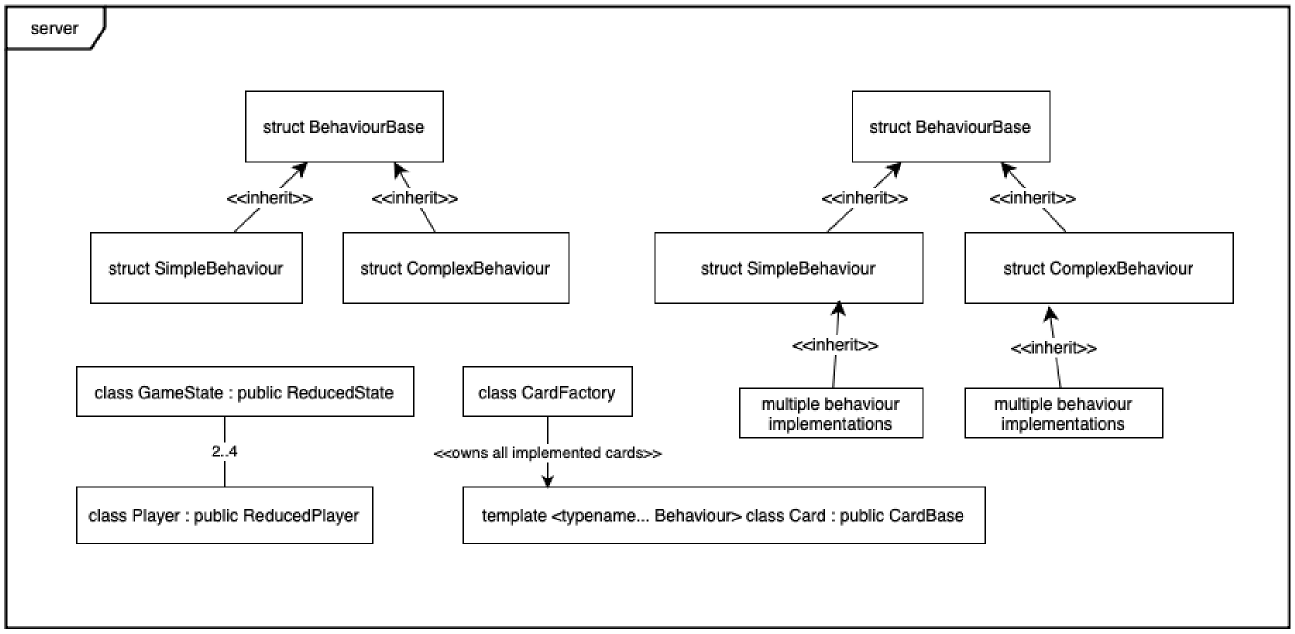
2.2.2 Class Controller

2.2.3 Class Model

2.2.4 Class NetworkManager

The class operations are:
- handle_message(ServerToClientMessage): void
- send_message(ClientToServerMessage): void

2.3 Package Server


server
struct BehaviourBase
<<inherit>>
<<inherit>>
struct SimpleBehaviour
struct ComplexBehaviour
struct BehaviourBase
<<inherit>>
<<inherit>>
struct SimpleBehaviour
struct ComplexBehaviour
<<inherit>>
<<inherit>>
multiple behaviour implementations
multiple behaviour implementations
class GameState : public ReducedState
2..4
class Player : public ReducedPlayer
class CardFactory
<<owns all implemented cards>>
template <typename... Behaviour> class Card : public CardBase

2.3.1 Class ServerNetworkManager

This class manages all communication with the client

The class attributes are:
- lobby_manager: LobbyManager

The class operations are:
- handle_message(string): void
- send_message(string): void
- get_message_interface(): MessageInterface

2.3.2 Class MessageInterface: public ServerToClientMessage, ClientToServerMessage

The class operations are:
- send_response_message(PlayerBase::id_t, ServerToClientMessage)
- broadcast_response_message(ServerToClientMessage)
- send_game_state(PlayerBase::id_t, ReducedGameState)
- broadcast_game_state(GameState)
- send_action_order(PlayerBase::id_t, ActionOrder)

2.3.3 Class LobbyManager

The class operations are:
- create_lobby(game_id: string, game_master: PlayerBase::id_t)
- join_lobby(game_id: string, PlayerBase::id_t: PlayerBase::id_t)
- start_game(game_id: string)
- receive_action(ActionDecisionMessage, MessageInterface)

- get_game_state(game_id: string, PlayerBase::id_t: PlayerBase::id_t): ReducedGameState

The class attributes are:

- lobbies: map<game_id, Lobby>
- messageInterface: MessageInterface

2.3.4 Class Lobby

The class operations are:

- join(PlayerBase::id_t: PlayerBase::id_t)
- start_game()
- receive_action(ActionDecisionMessage, MessageInterface)
- get_game_state(PlayerBase::id_t: PlayerBase::id_t)

The class attributes are:

- game_state: GameState

2.3.5 Class GameState

The GameState stores important information about the game and does all of the logic required for the game.

The class attributes are:

- players: vector<Player>
- board: ServerBoard
- current_player: PlayerBase::id_t

The class operations are:

- start_game(): void
- end_game(): void
- start_turn(): void
- end_turn(): void
- switch(Player): void
- try_buy(Player, CardBase::id_t): bool
- trash(Player, CardBase::id_t): bool
- discard(Player, CardBase::id_t): bool
- draw(Player, int): bool
- play(Player, CardBase::id_t): bool
- is_game_over(): bool
- new_action(ActionDecisionMessage): bool
- get_reduced_state() const: ReducedGameState

2.3.6 Class Player: public PlayerBase

The Player is a server class for Players that offers additional functionality compared to the base class in order to be able to execute moves.

The class attributes are:

- draw_pile: deque<CardBase::id_t>
- discard_pile: vector<CardBase::id_t>
- hand_cards: vector<CardBase::id_t>

- currently_playing_card: bool

The class operations are:

- try_buy(CardBase::id_t): bool
- trash(CardBase::id_t): bool
- discard(CardBase::id_t): bool
- draw(int): bool
- can_play(CardBase::id_t): bool
- add_points(int): void
- add_buys(int): void
- add_actions(int): void
- add_coins(int): void
- update_clientplayer(): ClientPlayer

2.3.7 Class ServerBoard : public Board

The ServerBoard offers additional functionality to be able to modify the board attributes.

The class operations are:

- buy(CardBase::id_t): bool
- trash(CardBase::id_t): void
- update_clientboard(self): Board

2.3.8 struct BehaviourBase

- virtual apply(Player, GameState): void
- virtual isTrivial() const: bool
- virtual isComplex() const: bool
- virtual toString() const: string

2.3.9 struct SimpleBehaviour : public BehaviourBase

- isTrivial const override: bool

Some simple behaviours could be DrawCards<int N> or GainBuys<int N>

2.3.10 struct ComplexBehaviour : public BehaviourBase

- isComplex const override: bool

A complex behaviour could be DiscardAttack

2.3.11 class Card : public CardBase

The serverside representation of cards, allows to access all vital information as well as applying the effects (behaviours) of the card to the game state. Templated with an arbitrary list of Behaviours
The cards will be created and registered at compile time using macros. See the example in the diagram

using CardPtr = unique_ptr<Card>

- behaviours: tuple<Behaviour...>
- play(Player, GameState): void

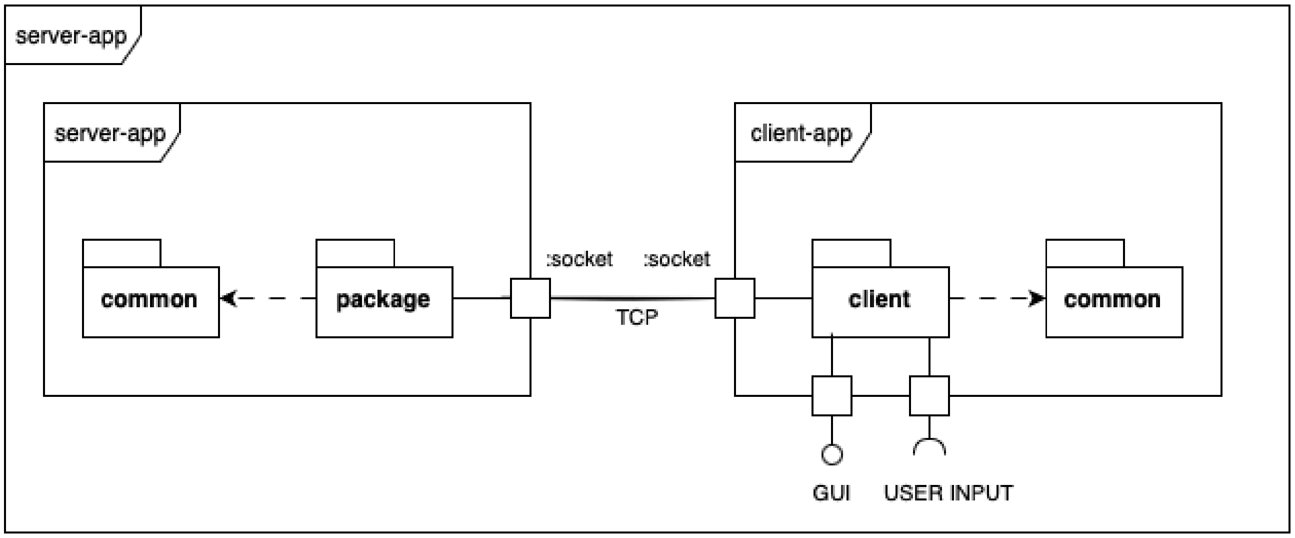
2.3.12 class CardFactory

The class used to instantiate and manage the cards during the game.

using map_type = map<string, Card::CardPtr>
- static card_map: map_type
- static get(string): Card::CardPtr
- static register(string, Card::CardPtr): void
- static getMap() const: map_type

2.4 Composite Structure Diagram

This is a composite structure diagram of our app. We provide this diagram to ease the visualisation and mental image of our package structure.



3. Class Diagrams

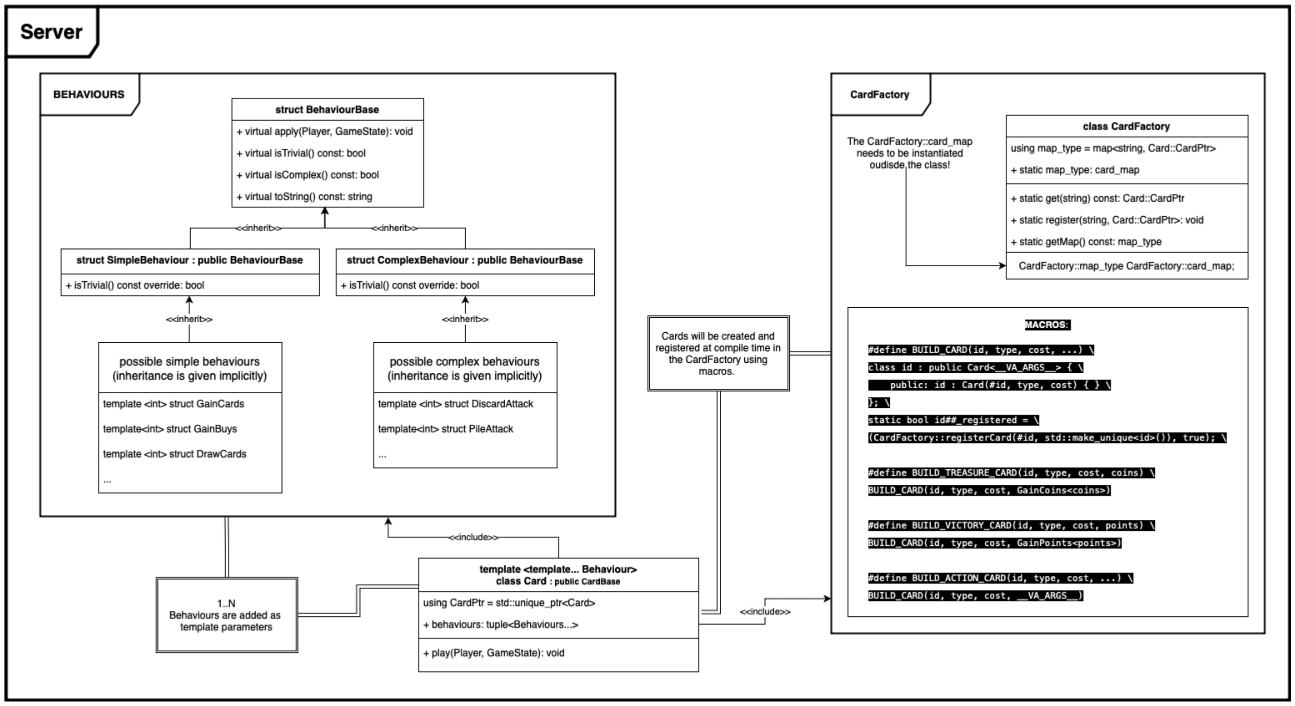
3.1 Card classes

The card and helper classes form the core of our game's behaviour management, encapsulating the relationships and functionalities that drive the game mechanics. The accompanying diagram focuses exclusively on classes and structures relevant to card interactions, omitting unrelated components for clarity.

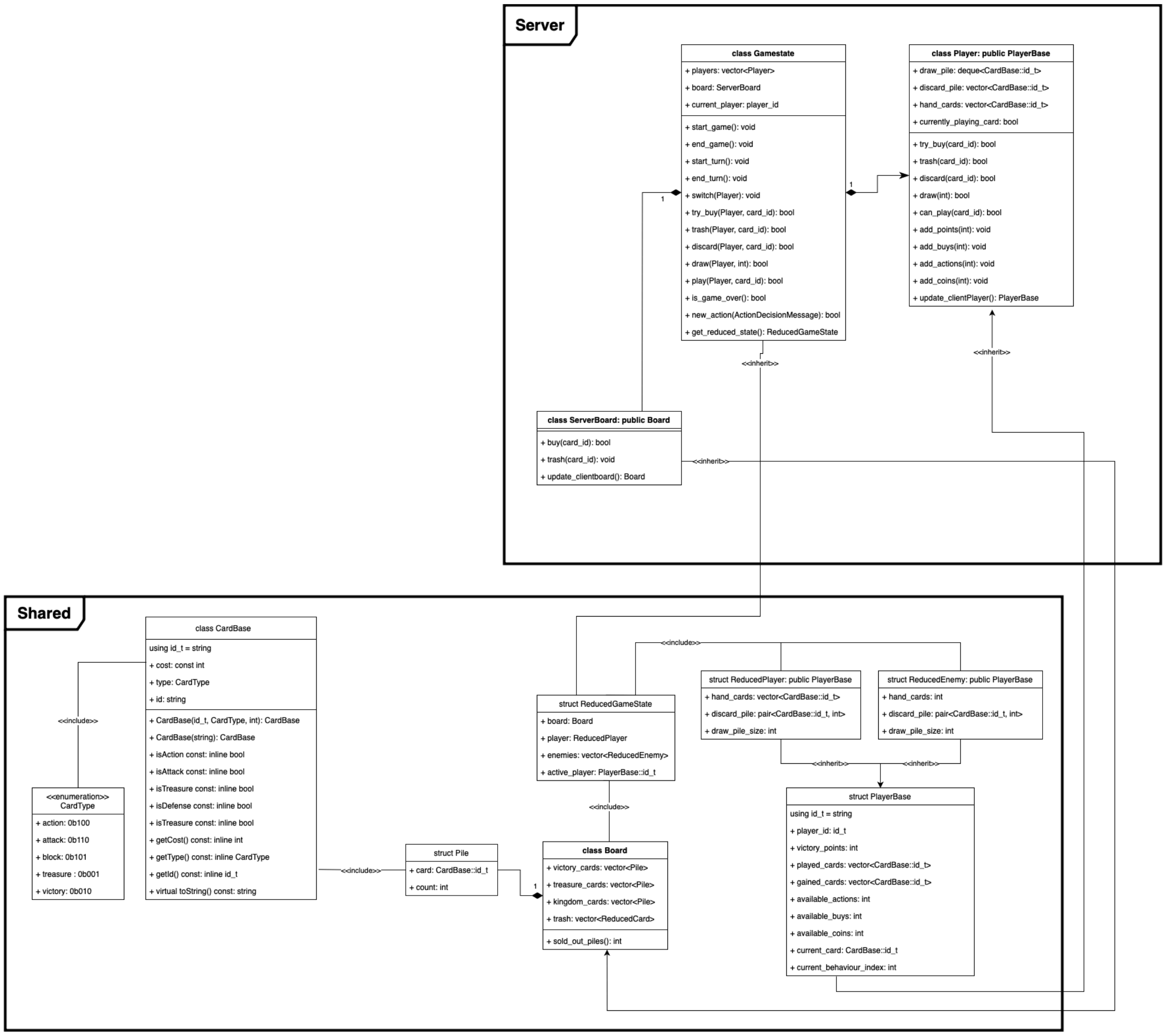
3.1.1 Concept Overview

Dominion features a vast array of cards, each with unique abilities, many of which exhibit overlapping behaviours. To promote code reusability and modularity, we have encapsulated these common behaviours within a dedicated `*Behaviour*` struct.

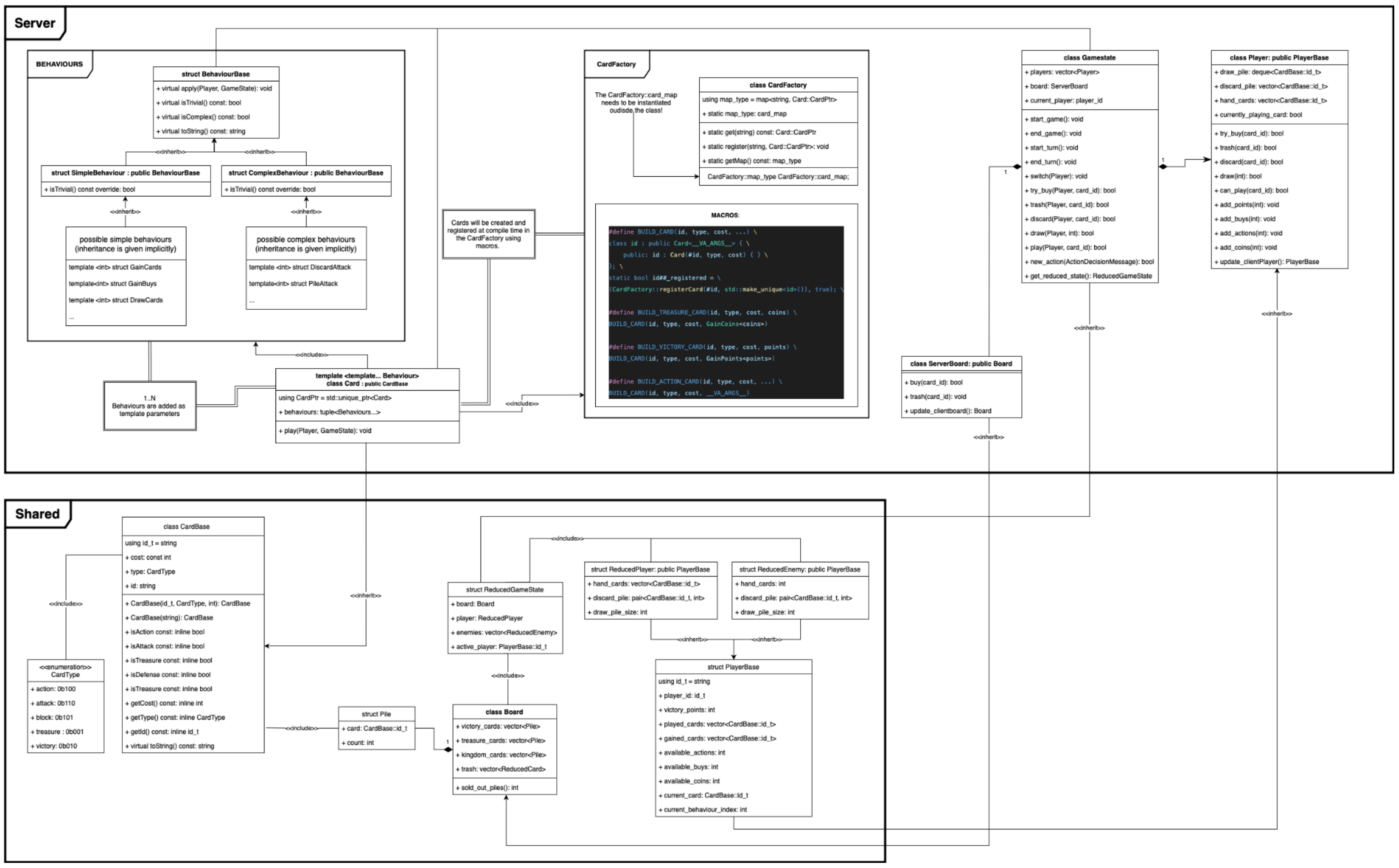
Individual cards are instantiated by specifying their behaviours as template parameters. During compilation, all `*apply*` functions from the behaviours are combined into a single cohesive function, ensuring efficiency and flexibility. This approach not only streamlines the creation of cards but also facilitates easy integration of future Dominion expansions. By introducing new behaviours, we can effortlessly generate new cards using macros, significantly minimising code duplication and simplifying the development process.


Server
BEHAVIOURS
struct BehaviourBase
+ virtual apply(Player, GameState): void
+ virtual isTrivial() const: bool
+ virtual isComplex() const: bool
+ virtual toString() const: string
<<inherit>>
<<inherit>>
struct SimpleBehaviour : public BehaviourBase
+ isTrivial() const override: bool
struct ComplexBehaviour : public BehaviourBase
+ isTrivial() const override: bool
<<inherit>>
<<inherit>>
possible simple behaviours (inheritance is given implicitly)
template <int> struct GainCards
template<int> struct GainBuys
template <int> struct DrawCards
...
possible complex behaviours (inheritance is given implicitly)
template <int> struct DiscardAttack
template<int> struct PileAttack
...
<<include>>
1..N Behaviours are added as template parameters
template <template... Behaviour> class Card : public CardBase
using CardPtr = std::unique_ptr<Card>
+ behaviours: tuple<Behaviours...>
+ play(Player, GameState): void
Cards will be created and registered at compile time in the CardFactory using macros.
<<include>>
CardFactory
The CardFactory::card_map needs to be instantiated oudisde the class!
class CardFactory
using map_type = map<string, Card::CardPtr>
+ static map_type: card_map
+ static get(string) const: Card::CardPtr
+ static register(string, Card::CardPtr>: void
+ static getMap() const: map_type
CardFactory::map_type CardFactory::card_map;
MACROS:

```
#define BUILD_CARD(id, type, cost, ...) \
class id : public Card<__VA_ARGS__> { \
    public: id : Card(#id, type, cost) { } \
}; \
static bool id##_registered = \
(CardFactory::registerCard(#id, std::make_unique<id>()), true); \

#define BUILD_TREASURE_CARD(id, type, cost, coins) \
BUILD_CARD(id, type, cost, GainCoins<coins>)

#define BUILD_VICTORY_CARD(id, type, cost, points) \
BUILD_CARD(id, type, cost, GainPoints<points>)

#define BUILD_ACTION_CARD(id, type, cost, ...) \
BUILD_CARD(id, type, cost, __VA_ARGS__)
```


Server
class Gamestate
+ players: vector<Player>
+ board: ServerBoard
+ current_player: player_id
+ start_game(): void
+ end_game(): void
+ start_turn(): void
+ end_turn(): void
+ switch(Player): void
+ try_buy(Player, card_id): bool
+ trash(Player, card_id): bool
+ discard(Player, card_id): bool
+ draw(Player, int): bool
+ play(Player, card_id): bool
+ is_game_over(): bool
+ new_action(ActionDecisionMessage): bool
+ get_reduced_state(): ReducedGameState
class Player: public PlayerBase
+ draw_pile: deque<CardBase::id_t>
+ discard_pile: vector<CardBase::id_t>
+ hand_cards: vector<CardBase::id_t>
+ currently_playing_card: bool
+ try_buy(card_id): bool
+ trash(card_id): bool
+ discard(card_id): bool
+ draw(int): bool
+ can_play(card_id): bool
+ add_points(int): void
+ add_buys(int): void
+ add_actions(int): void
+ add_coins(int): void
+ update_clientPlayer(): PlayerBase
1
1
<<inherit>>
<<inherit>>
class ServerBoard: public Board
+ buy(card_id): bool
+ trash(card_id): void
+ update_clientboard(): Board
<<inherit>>
Shared
class CardBase
using id_t = string
+ cost: const int
+ type: CardType
+ id: string
+ CardBase(id_t, CardType, int): CardBase
+ CardBase(string): CardBase
+ isAction const: inline bool
+ isAttack const: inline bool
+ isTreasure const: inline bool
+ isDefense const: inline bool
+ isTreasure const: inline bool
+ getCost() const: inline int
+ getType() const: inline CardType
+ getId() const: inline id_t
+ virtual toString() const: string
<<include>>
<<enumeration>> CardType
+ action: 0b100
+ attack: 0b110
+ block: 0b101
+ treasure : 0b001
+ victory: 0b010
<<include>>
struct Pile
+ card: CardBase::id_t
+ count: int
1
<<include>>
struct ReducedGameState
+ board: Board
+ player: ReducedPlayer
+ enemies: vector<ReducedEnemy>
+ active_player: PlayerBase::id_t
<<include>>
class Board
+ victory_cards: vector<Pile>
+ treasure_cards: vector<Pile>
+ kingdom_cards: vector<Pile>
+ trash: vector<ReducedCard>
+ sold_out_piles(): int
struct ReducedPlayer: public PlayerBase
+ hand_cards: vector<CardBase::id_t>
+ discard_pile: pair<CardBase::id_t, int>
+ draw_pile_size: int
struct ReducedEnemy: public PlayerBase
+ hand_cards: int
+ discard_pile: pair<CardBase::id_t, int>
+ draw_pile_size: int
<<inherit>>
<<inherit>>
struct PlayerBase
using id_t = string
+ player_id: id_t
+ victory_points: int
+ played_cards: vector<CardBase::id_t>
+ gained_cards: vector<CardBase::id_t>
+ available_actions: int
+ available_buys: int
+ available_coins: int
+ current_card: CardBase::id_t
+ current_behaviour_index: int

3.2 Full Class Diagram


Server
BEHAVIOURS
struct BehaviourBase
+ virtual apply(Player, GameState): void
+ virtual isTrivial() const: bool
+ virtual isComplex() const: bool
+ virtual toString() const: string
<<inherit>>
struct SimpleBehaviour : public BehaviourBase
+ isTrivial() const override: bool
struct ComplexBehaviour : public BehaviourBase
+ isTrivial() const override: bool
possible simple behaviours (inheritance is given implicitly)
template <int> struct GainCards
template<int> struct GainBuys
template <int> struct DrawCards
...
possible complex behaviours (inheritance is given implicitly)
template <int> struct DiscardAttack
template<int> struct PileAttack
...
<<include>>
1..N Behaviours are added as template parameters
template <template... Behaviour> class Card : public CardBase
using CardPtr = std::unique_ptr<Card>
+ behaviours: tuple<Behaviours...>
+ play(Player, GameState): void
Cards will be created and registered at compile time in the CardFactory using macros.
CardFactory
The CardFactory::card_map needs to be instantiated oudisde the class!
class CardFactory
using map_type = map<string, Card::CardPtr>
+ static map_type: card_map
+ static get(string) const: Card::CardPtr
+ static register(string, Card::CardPtr>: void
+ static getMap() const: map_type
CardFactory::map_type CardFactory::card_map;
MACROS:
class Gamestate
+ players: vector<Player>
+ board: ServerBoard
+ current_player: player_id
+ start_game(): void
+ end_game(): void
+ start_turn(): void
+ end_turn(): void
+ switch(Player): void
+ try_buy(Player, card_id): bool
+ trash(Player, card_id): bool
+ discard(Player, card_id): bool
+ draw(Player, int): bool
+ play(Player, card_id): bool
+ new_action(ActionDecisionMessage): bool
+ get_reduced_state(): ReducedGameState
1
class Player: public PlayerBase
+ draw_pile: deque<CardBase::id_t>
+ discard_pile: vector<CardBase::id_t>
+ hand_cards: vector<CardBase::id_t>
+ currently_playing_card: bool
+ try_buy(card_id): bool
+ trash(card_id): bool
+ discard(card_id): bool
+ draw(int): bool
+ can_play(card_id): bool
+ add_points(int): void
+ add_buys(int): void
+ add_actions(int): void
+ add_coins(int): void
+ update_clientPlayer(): PlayerBase
class ServerBoard: public Board
+ buy(card_id): bool
+ trash(card_id): void
+ update_clientboard(): Board
Shared
class CardBase
using id_t = string
+ cost: const int
+ type: CardType
+ id: string
+ CardBase(id_t, CardType, int): CardBase
+ CardBase(string): CardBase
+ isAction const: inline bool
+ isAttack const: inline bool
+ isTreasure const: inline bool
+ isDefense const: inline bool
+ isTreasure const: inline bool
+ getCost() const: inline int
+ getType() const: inline CardType
+ getId() const: inline id_t
+ virtual toString() const: string
<<enumeration>> CardType
+ action: 0b100
+ attack: 0b110
+ block: 0b101
+ treasure : 0b001
+ victory: 0b010
struct Pile
+ card: CardBase::id_t
+ count: int
struct ReducedGameState
+ board: Board
+ player: ReducedPlayer
+ enemies: vector<ReducedEnemy>
+ active_player: PlayerBase::id_t
class Board
+ victory_cards: vector<Pile>
+ treasure_cards: vector<Pile>
+ kingdom_cards: vector<Pile>
+ trash: vector<ReducedCard>
+ sold_out_piles(): int
struct ReducedPlayer: public PlayerBase
+ hand_cards: vector<CardBase::id_t>
+ discard_pile: pair<CardBase::id_t, int>
+ draw_pile_size: int
struct ReducedEnemy: public PlayerBase
+ hand_cards: int
+ discard_pile: pair<CardBase::id_t, int>
+ draw_pile_size: int
struct PlayerBase
using id_t = string
+ player_id: id_t
+ victory_points: int
+ played_cards: vector<CardBase::id_t>
+ gained_cards: vector<CardBase::id_t>
+ available_actions: int
+ available_buys: int
+ available_coins: int
+ current_card: CardBase::id_t
+ current_behaviour_index: int

```
#define BUILD_CARD(id, type, cost, ...) \
class id : public Card<__VA_ARGS__> { \
    public: id : Card(#id, type, cost) { } \
}; \
static bool id##_registered = \
(CardFactory::registerCard(#id, std::make_unique<id>()), true); \

#define BUILD_TREASURE_CARD(id, type, cost, coins) \
BUILD_CARD(id, type, cost, GainCoins<coins>)

#define BUILD_VICTORY_CARD(id, type, cost, points) \
BUILD_CARD(id, type, cost, GainPoints<points>)

#define BUILD_ACTION_CARD(id, type, cost, ...) \
BUILD_CARD(id, type, cost, __VA_ARGS__)
```

4. Sequence Diagrams

<Sequence Diagrams are linked to the scenarios and the functional requirements of the SRS document. In this diagrams we are trying to show how the modelling can be used to implement this scenarios and requirements.>

4.1 Sequence Create Lobby

Player goes from the welcome screen to the lobby screen

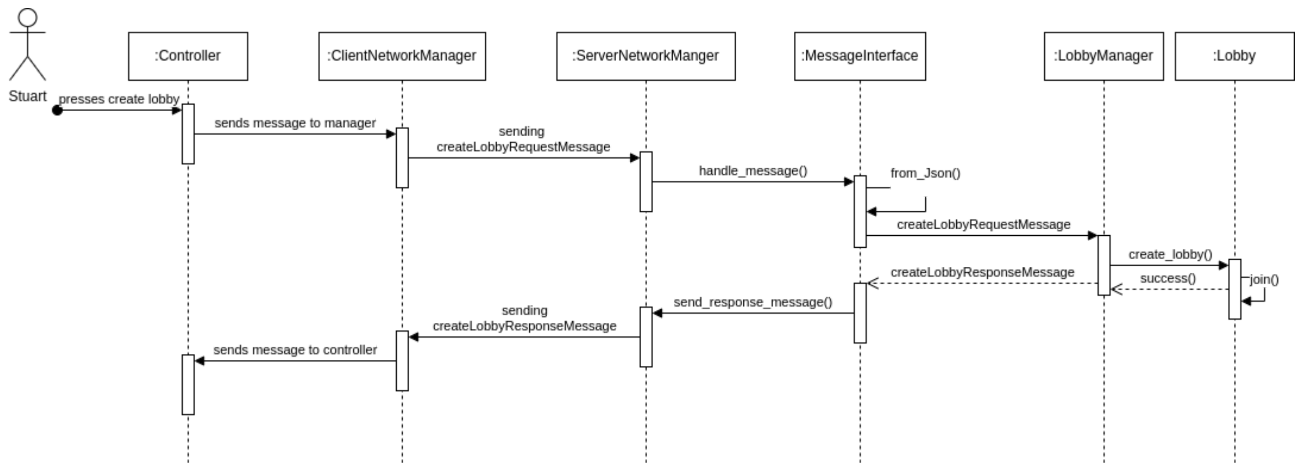
The functional requirements related to this sequence are:

FREQ-S-01: Game session management
FREQ-S-02: Player authentication

The scenarios which are related to this sequence are:

SCN-1: Setting up a lobby

Scenario Narration:
Stuart presses create lobby and a createLobbyRequestMessage is sent to the server which creates a lobby and adds Stuart and then sends back a createLobbyResponseMessage.



4.2 Sequence Play Card

Player plays card

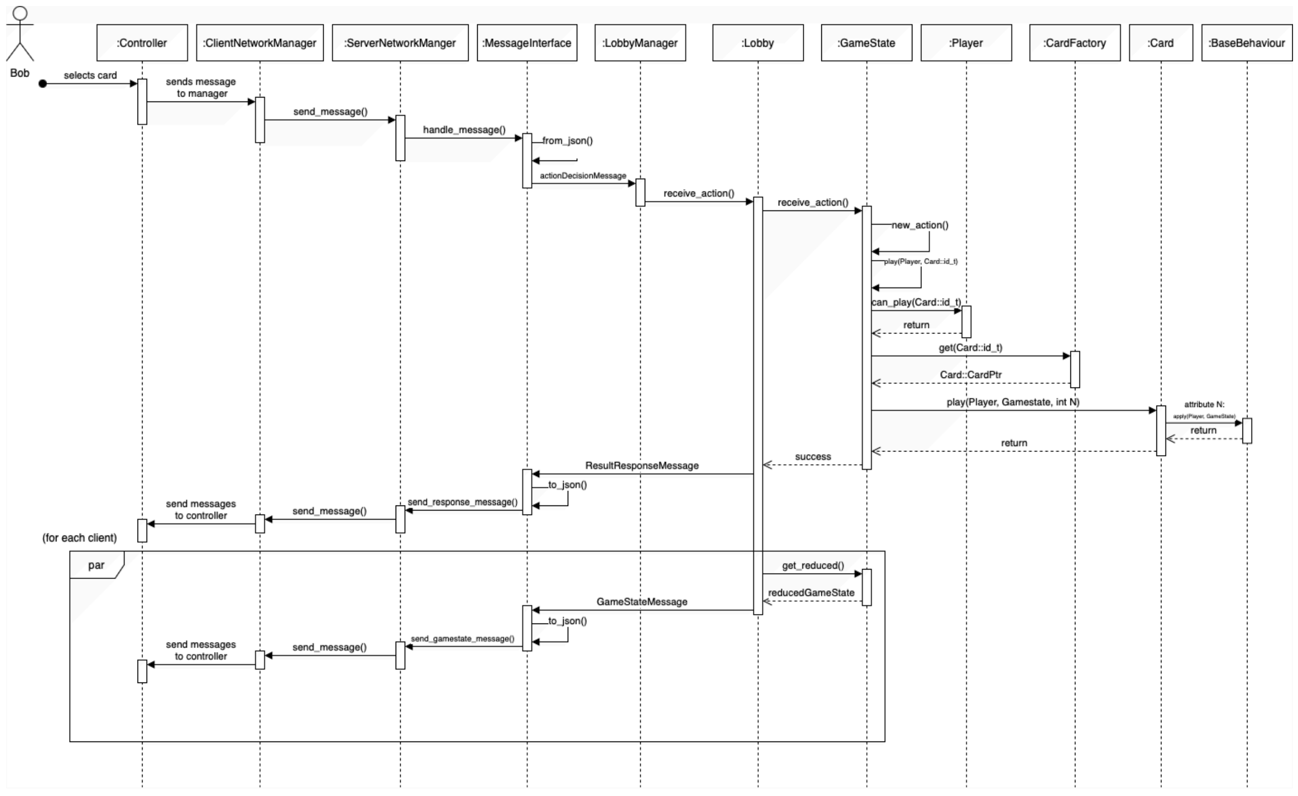
The functional requirements related to this sequence are:

FREQ-01: Implemented Cards
FREQ-S-03: Game state Synchronisation
FREQ-C-02: Server Communication
FREQ-C-03: Player Notification

The scenarios which are related to this sequence are:

SCN-3: Standard turn

Scenario Narration:
Bob plays a card. The client sends an according message to the server which registers that a card has been played and starts executing the behaviours of the card which edit the Game State. After successfully completing the behaviour it returns a success message and also notifies the other player that the card has been played.



4.3 Sequence End of Game

Then end of the game

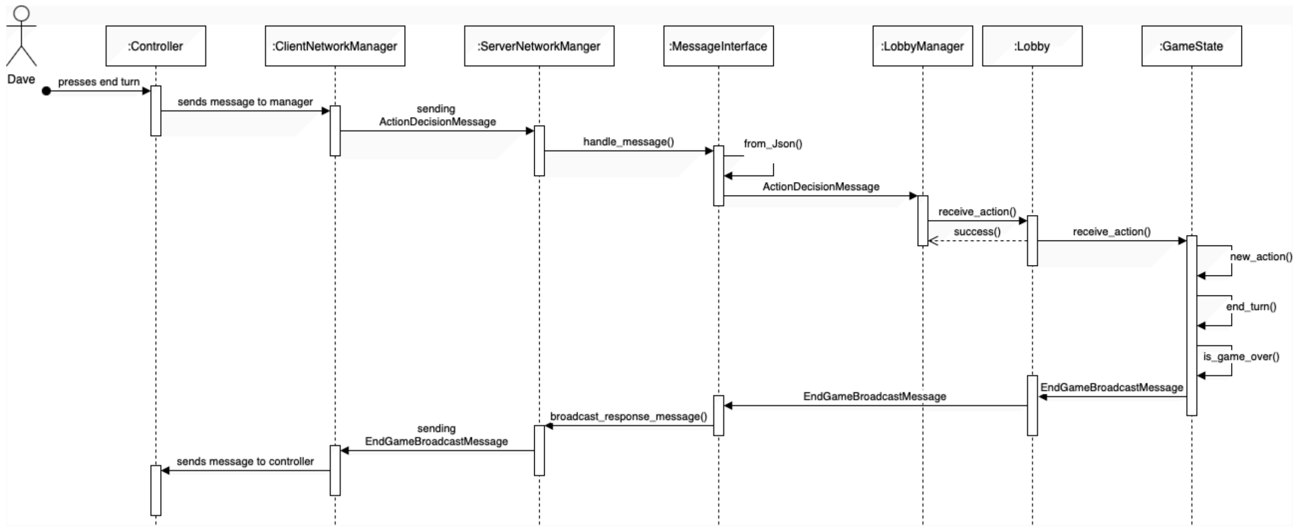
The functional requirements related to this sequence are:

FREQ-S-03: Game state Synchronisation
FREQ-C-02: Server Communication
FREQ-C-03: Player Notification
FREQ-G-04: Victory Screen

The scenarios which are related to this sequence are:

SCN-5: End of game

Scenario Narration:
Dave presses end turn. The client sends an ActionDecisionMessage to the server which ends Dave's turn and checks whether the game is over. After finding that it is, the EndGameMessage is broadcast.



5. Interface Modelling

5.1 Interface Client-Server

The purpose of this interface is the communication of the clients with the server. The interface allows performs the following primary functions:

- *Initialization of games*
- *Running games, that is, it provides the capabilities to transfer game state updates from the server to the client and moves from the client to the server.*
- *Termination of games (end_game_broadcast)*
- *Recovery in the case of disconnection (game_state_request)*

Communication between: Client and server, generally initiated by client, game state update broadcasts by the server

Protocol: TCP

Communication modes:

- Request-response during initialization
- Three-way-communication action_order->action_decision->game_state between the current player and the server during a running game
- Broadcasts from the server to all players of a game to update their game states

5.1.1 game_state

Purpose: Get the game state for the client

Direction: Server to client

Content:

- game_id: string (required)
- message_id: string (required)
- in_response_to: string (optional; only for game_state_request)
- type: “game_state” (required)
- game_state: array of objects describing the game state (required)

Format: JSON string

Example:
```
{
      "game_id": "Nicola’s Game",
      "message_id": "abc123",
      "type": "game_state",
      "game_state": <json game state encoding>
}
```

Expected response: *none*

5.1.2 ***game_state_request***

Purpose: Request to get the complete state of the game. This is used when a player reconnects after a loss of connection.

Direction: Client to server

Content:

- game_id: string (required)
- message_id: string (required)
- type: “game_state_request” (required)

Format: JSON string

Example:
```
{
      "game_id": "Nicola’s Game",
      "message_id": "4",
      "type": "game_state_request"
}
```

Expected response: game_state

5.1.3 ***initiate_game_request***

Purpose: Request to create a new game lobby.

Direction: Client to server

Content:

- game_id: string (required)
- message_id: string (required)
- type: “initiate_game_request” (required)

Format: JSON string

Example:
```
{
      "game_id": "Nicola’s Game",
      "message_id": "abc123",
      "type": "initiate_game_request"
}
```

Expected response: initiate_game_response

5.1.4 *initiate_game_response*

Purpose: Response to creating a new lobby containing all available cards to select from.

Direction: Client to server

Content:

- game_id: string (required)
- message_id: string (required)
- in_response_to: string (required)
- type: “initiate_game_response” (required)
- available_cards: array of all card type UIDs (required)

Format: JSON string

Example:
```
{
      "game_id": "Nicola’s Game",
      "message_id": "abc123",
      "in_response_to": "xyz456",
      "type": "initiate_game_response",
      "available_cards": [
            "Cellar", "Market", "Merchant", "Militia", "Mine", "Moat,
            "Remodel", "Smithy", "Village", "Workshop", …, "Witch"
      ]
}
```

Expected response: none

5.1.5 *join_game_request*

Purpose: Request to join an existing game lobby

Direction: Client to server

Content:

- game_id: string (required)
- message_id: string (required)
- type: “join_game_request” (required)

Format: JSON string

Example:
```
{
      "game_id": "Nicola’s Game",
      "message_id": "2",
      "type": "join_game_request"
}
```

Expected response: result_response

5.1.6 *start_game_request*

Purpose: Request to start a game along with supplying a selection of 10 kingdom cards. Only the game master is allowed to do this action

Direction: Client to server

Content:

- game_id: string (required)
- message_id: string (required)
- type: “start_game_request” (required)
- cards: array of exactly 10 card type UIDs (required)

Format: JSON string

Example:

```
{
      "game_id": "Nicola’s Game",
      "message_id": "abc123",
      "type": "start_game_request",
      "cards": [
            "Cellar", "Market", "Merchant", "Militia", "Mine", "Moat,
            "Remodel", "Smithy", "Village", "Workshop"
      ]
}
```

Expected response: result_response, game_state

5.1.7 result_response

Purpose: Indicate the result (success/error) of a request.

Direction: Server to client

Content:

- game_id: string (required)
- message_id: string (required)
- in_response_to: string (required)
- type: “result_response” (required)
- success: bool (required)
- additional_information: string (required)

Format: JSON string

Example:
```
{
      "game_id":"Nicola’s Game",
      "message_id": "abc123",
      "in_response_to": "adf543",
      "type": "result_response",
      "success": true,
      "additional_information": "You selected too many cards"
}
```

Expected response: *none*

5.1.8 *action_order*

Purpose: Used during a running game to tell a client to ask the player to make a move. The parameter “phase” specifies which types of moves are allowed.

Direction: Server to client

Content:

- game_id: string (required)
- message_id: string (required)
- type: “action_order” (required)
- description: string (optional)
- phase: one of “action_phase”, “buy_phase”, “choose_n_cards_from_hand”
- params: array of parameters specific to the action_type

Format: JSON string

Example:
```
{
      "game_id": "Nicola’s Game",
      "message_id": "5",
      "type": "action_order",
      "phase": "choose_n_cards_from_hand",
      "params": {
            "n": "1"
      }
}
```

Expected response: action_decision

5.1.9 *action_decision*

Purpose: Answer to an “action_order”. Used to transmit the action of a player to the server.

Direction: Client to server

Content:

- game_id: string (required)
- message_id: string (required)
- in_response_to: string (required, this is the “message_id” of the corresponding “action_order”)
- PlayerBase::id_t: string (required)
- type: “action_decision” (required)
- action: one of “play_action_card”, “buy_card”, “end_turn”, “choose_n_cards_from_hand” (required)
- params: array of parameters specific to the action (optional)

Format: JSON string

Example 1 (play_action_card):
```
{
      "game_id": "Nicola’s Game",
      "message_id": "6",
      "in_response_to": "5",
      "PlayerBase::id_t": "Nicola",
      "type": "action_decision",
      "action": "play_action_card",
      "params": {
            "card": "Village"
      }
}
```

Example 2 (buy_card):
```
{
      "game_id": "Nicola’s Game",
      "message_id": "6",
      "in_response_to": "5",
      "PlayerBase::id_t": "Nicola",
      "type": "action_decision",
      "action": "buy_card",
      "params": {
            "card": "Province"
      }
}
```

Example 3 (end_turn):
```
{
      "game_id": "Nicola’s Game",
      "message_id": "6",
      "in_response_to": "5",
      "PlayerBase::id_t": "Nicola",
      "type": "action_decision",
      "action": "end_turn"
}
```

Example 4 (choose_n_cards_from_hand):
```
{
      "game_id": "Nicola’s Game",
      "message_id": "6",
      "in_response_to": "5",
      "PlayerBase::id_t": "Nicola",
      "type": "action_decision",
      "action": "choose_n_cards_from_hand",
      "params": {
            cards: [ "Curse", "Duchy" ]
      }
}
```

Expected response: game_state

5.1.10 *join_game_broadcast*

Purpose: Tell the clients that a new player joined the game lobby.

Direction: Server to client

Content:

- game_id: string (required)
- message_id: string (required)
- type: “join_game_broadcast” (required)
- player_name: string (required)

Format: JSON string

Example:
```
{
      "game_id": "Nicola’s Game",
      "message_id": "2",
      "type":"join_game_broadcast",
      "player_name": "Philipp"
}
```

Expected response: *none*

5.1.11 *end_game_broadcast*

Purpose: Tell the client that the game ended as well as the scores and who is on the podium.

Direction: Server to client

Content:

- game_id: string (required)
- message_id: string (required)
- type: “end_game_broadcast” (required)
- player_scores: array of objects containing PlayerBase::id_t and player_score (required)

Format: JSON string

Example:
```
{
      "game_id": "Nicola’s Game",
      "message_id": "2", "type":
      "start_game_broadcast", "player_scores": [
            { "PlayerBase::id_t": "Nicola", "player_score": 20 },
            { "PlayerBase::id_t": "Philipp", "player_score": 18 },
            { "PlayerBase::id_t": "Aaron", "player_score": 30 }
      ]
}
```

Expected response: *none*