# Developer documentation

## Basic information

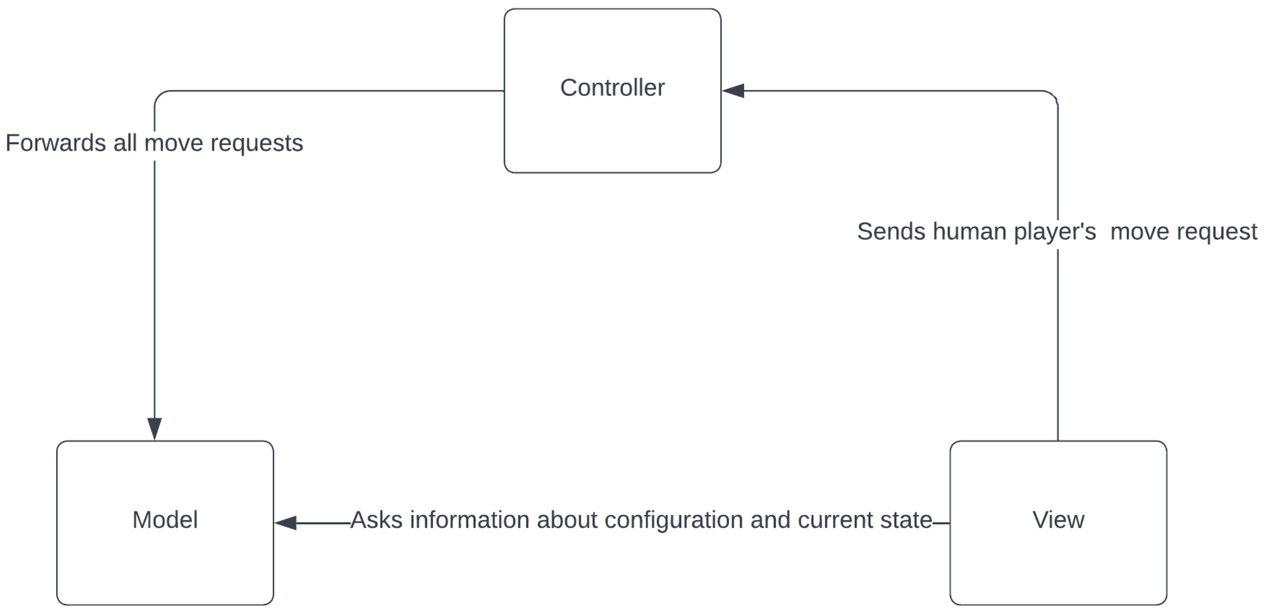
- The project is written in `C++` using the `C++20` standard version. The reason I chose the latest version of C++ is that it defines concepts that I am using to check at compile time whether certain classes fulfill the required template type (e.g. the local game card choosing).
- The game is visualized using the Gtkmm-3.0 graphical user interface library.
- The project supports the Windows and Linux operating systems. It was tested on both operating systems (Windows 11 and Linux - Ubuntu + using WSL).
- There are the following folders in the top level of the repository:
    - data - contains data that the game uses e.g. the patterns for Window pattern cards
    - doc - contains both developer documentation (= this document) and user documentation
    - images - contains pre-drawn images of the game that are non-changing during the game
    - include - contains the header files of the project
    - src - contains the source files of the project
    - subprojects - contains Meson packages for the `GoogleTest` testing framework and for the `Nlohmann JSON` parser
    - tests - contains the unit tests of the project including the testing data
- Building the project requires the Meson build system, the Ninja build system, the gtkmm-3.0 graphical user interface library and a C++ compiler that could compile C++20 features. The project is built using these commands:

```
$ meson build
$ ninja -C build
$ build/sagrada
```

- The following abbreviations are used frequently in the game's source code and this documentation:
    - Window pattern cards -> WPC
    - Tool cards -> TC
    - Public objective cards -> PuOC
    - Private objective cards -> PrOC The actual aliases used in the source code are listed in the `Typedefs.h` header file.

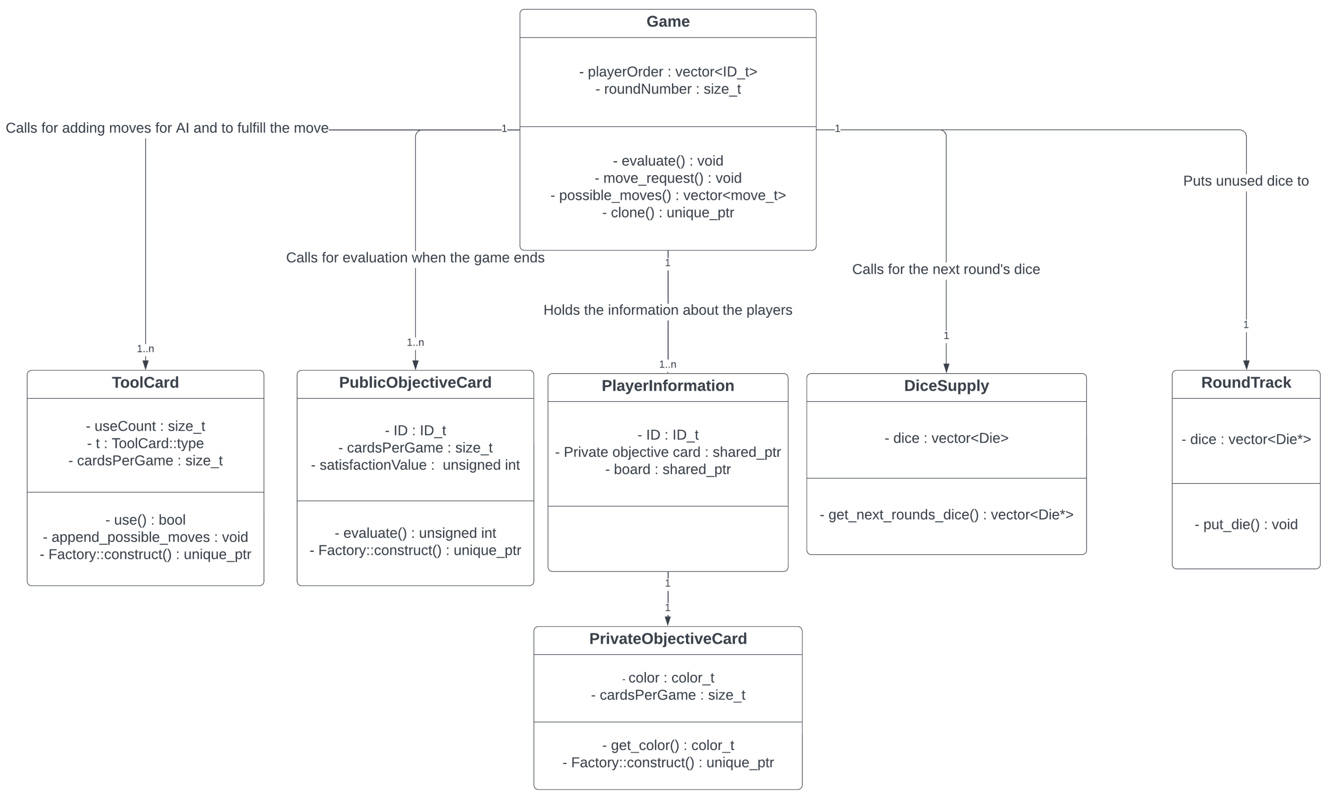
## Architecture

The project follows a typical Model - View - Controller architecture.


Controller
Forwards all move requests
Sends human player's move request
Model
Asks information about configuration and current state
View

## Model

The model component is present for holding the data about the game (e.g. the chosen cards) and the current state ( e.g. which player's turn it is). There are approximately 5000 lines of code in the Model component including comments. It is represented by the singleton Game class which has the following structure:


Game
- playerOrder : vector<ID_t>
- roundNumber : size_t
- evaluate() : void
- move_request() : void
- possible_moves() : vector<move_t>
- clone() : unique_ptr
Calls for adding moves for AI and to fulfill the move
Calls for evaluation when the game ends
Holds the information about the players
Calls for the next round's dice
Puts unused dice to
1
1..n
ToolCard
- useCount : size_t
- t : ToolCard::type
- cardsPerGame : size_t
- use() : bool
- append_possible_moves : void
- Factory::construct() : unique_ptr
PublicObjectiveCard
- ID : ID_t
- cardsPerGame : size_t
- satisfactionValue : unsigned int
- evaluate() : unsigned int
- Factory::construct() : unique_ptr
PlayerInformation
- ID : ID_t
- Private objective card : shared_ptr
- board : shared_ptr
DiceSupply
- dice : vector<Die>
- get_next_rounds_dice() : vector<Die*>
RoundTrack
- dice : vector<Die*>
- put_die() : void
PrivateObjectiveCard
- color : color_t
- cardsPerGame : size_t
- get_color() : color_t
- Factory::construct() : unique_ptr

## Board

The board class represents the data that describes a player's board. It consists of a 2D array of `BoardField` objects. Since it is a wrapper around a 2D array, it defines `operator[]` allowing the user to access board fields using double bracket syntax. The board objects are constructed from Window pattern cards.

The most important methods are mostly defined for often-used queries during the game. These are:

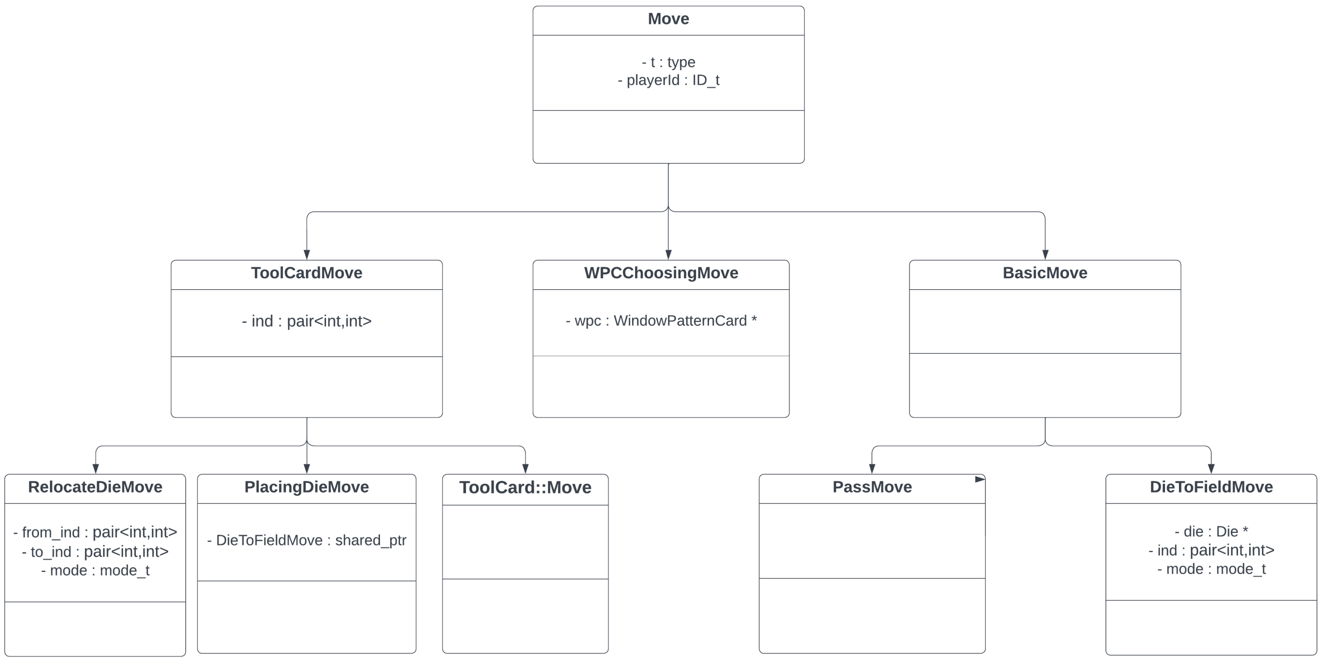
- **vector< pair<size_t, size_t>> get_direct_neighbor_indices(size_t rowInd, size_t colInd)**, **vector<pair<size_t, size_t>> get_corner_neighbor_indices(size_t rowInd, size_t colInd)** - these functions return a vector of absolute indices to the neighboring fields of a field
- **bool is_reachable(size_t rowIndex, size_t columnIndex)** - returns whether a field is reachable in the board = has a neighboring field that has a die on

## Moves

During the game, different types of moves could be made. The Game class distinguishes between these types using their enum class type variable `t`. Currently, it defines 3 different types of Moves: `BASIC_MOVE` , `WPC_CHOOSING`, `TOOL_CARD`. Each move has a playerId variable that indicates which player tries to make a move.

Some of the moves may use a so-called `mode` identifier which defines 3 types at the moment: `IGNORE_VALUE` , `IGNORE_COLOR`, `ACCEPT_UNREACHABLE`. These modes are used by the `move_request` evaluation function of the Game class.

The different types of moves have the following structure:



---

## Tool cards

Tool cards allow players to make special kinds of moves. Each tool card has a unique structure and a different influence on the game. They can modify different objects of the game. For this reason, there is a base class

`ToolCard`. It defines the types of Tool cards using an `enum class` object.

For easier construction, the `ToolCard` base class defines a `Factory` class having a single static `construct` method that receives the type of the tool card.

It defines the following interface to implement:

- **bool use(tool_card_move, Game&)** - receives the already constructed tool card move that contains data about the move to be made. After error checking it modifies the state according to the type of the tool card. Returns false when using a tool card that consists of more than one sub-moves and the final sub-move hasn't been sent yet. For some tool cards, it may happen that after a certain sub-move (that is not the last one) the player cannot continue using the tool card because there are no possible ways to continue. For example, the Flux Bush tool card (implemented by the RerollDie class), where the user first selects a die, rerolls it, and has to place the rerolled die. However, in some cases the user cannot place the rerolled die and the last (= placing) sub-move cannot be made.
- **size_t favor_tokens_to_use()** - according to the game rules, every tool card costs 1 favor token for the first use and 2 after. This function is made virtual so it could be overridden for a concrete tool card if decided so.
- **void append_possible_moves(moveContainer, playerId, Game&, toolCardInd, tool_card_move_t prevMove)** - when the game is collecting the possible moves to make for AI players, it calls this method to append the moves that could be made using the tool card. It receives as the last argument a `prevMove` argument which refers to the state of the tool card. This is important because there are tool cards using which consist of more than 1 move and the next move has to be appended according to this state.

---

## Public objective cards

Public objective cards are present to determine the underlying strategy of the game. Each game has the purpose of placing dice on the board, but every game is scored differently due to the randomness of the public objective cards.

For easier construction, the `PublicObjectiveCard` base class defines a Factory class having a single static `construct` method that receives the type of the tool card.

It defines the following interface to implement:

- **unsigned int evaluate(Board b)** - This is the function called after the game has ended and we want to evaluate a player's board. The implementation of this function has to be different for each type, but there is a big part of it that is the same for each of them. This part is looping through the board and tracking the things that have to be tracked according to the public objective card. This is handled by the **evaluate()** method of the **BoardEvaluator** class. This way this function only configures the evaluator according to the strategy of the given card.

#### BoardEvaluator

The **BoardEvaluator** class has a single static method **unsigned int evaluate(Board b, Tracker t)**. It defines the common part of the pattern matching for all the public objective cards. It receives a tracker instance that defines the pattern it is looking for. The evaluate function loops through the board and sends each field to the tracker. After this calls the tracker for the result and returns it.

A given tracker defines the perspective that we are looking through to the board. Each tracker has a unique structure that remembers the data it needs for evaluating a board field. The **AttributeTracker** class defines the interface that each tracker has to fulfill. As a template parameter, it receives the given class whose attributes it tracks. The most important functions are:

- **void track(std::unique_ptr& boardField, size_t rowInd, size_t colInd)** - this function is called for each field of a board and is responsible for saving the data that it needs for the evaluation process to produce a result for the given pattern
- **unsigned int get_result()** - called after all the fields of a board have been tracked to get the number of matches for the given pattern

---

### Private objective cards

Each player has a private objective card, which holds a color for which the player will receive bonus points. It is implemented as a wrapper around a `color_t` enum typed variable.

These card objects have to be hidden from almost everyone in the game because a player cannot be aware of the color of other players. For this reason, it's a private variable of the PlayerInformation objects, and the classes that have to know this color are defined as friend classes.

---

### Player Information

The PlayerInformation struct describes the information that the game holds about each player. The players can access each other's player information including the board and all the information except for the private objective cards.

---

### Dice

The main moves of the game are placing a die on a board. The dice are randomly initialized at the beginning of the game. Their values could be changed using some tool cards.

Since there are no new dice constructed in the game, the `DiceSupply` singleton object holds all the dice in a vector, and in any other place in the source code pointers are used for these dice instances. It makes it easier to compare two dice as well because there could be more dice having the same color and value, but each die has a unique address, and if we usually want to refer to a specific die.

---

### AI Players

The `AI_Player` class describes the interface that the concrete AI players have to implement. Currently, the single AI player is the `RandomPlayer` which makes random moves from all the possible moves.

This class defines the following interface:

- **move_t make_next_move()** - Returns a move that the player chooses according to the strategy it implements.

---

## View

There are approximately 5000 lines of code in the View component including the comments.

Visualizing the game is done using two different ways:

- drawing components that change during the game - boards, dice, round track, etc. These are drawn using the Cairomm graphics library.
- using pre-drawn images for the components that do not change - cards = tool cards, public and private objective cards and their icons

The `View` component defines a class of the same name. The **View** class serves as a background for the application. It is derived from Gtk::DrawingArea to be able to make any changes in the background. It constructs the home page. It defines a function **change_page_to()** which receives the page to change to as a parameter. It owns the Gtk::Fixed object that is used in the whole application to place Gtk::Widgets objects at specific positions.

### Pages

The different stages that occur from opening the application until closing it are divided into pages.

Currently, there are 3 pages present:

- **Home page** - this is the page that is displayed when opening the application
- **Game config page** - this is the page where you configure the game you are trying to play - e.g. the number of players
- **Game playing page** - this is the page where the whole game is played

The interface for pages:

- **show()**, **hide()** - virtual methods that define what widgets have to be shown/hidden when switching between pages

#### GamePlayingPage

- This is the main page of the application. To make the interaction with the human player smoother so the AI players' moves would be followable, the moves are handled by a timeout handler. If it's an AI player's move, it reschedules a timeout, otherwise, it returns.

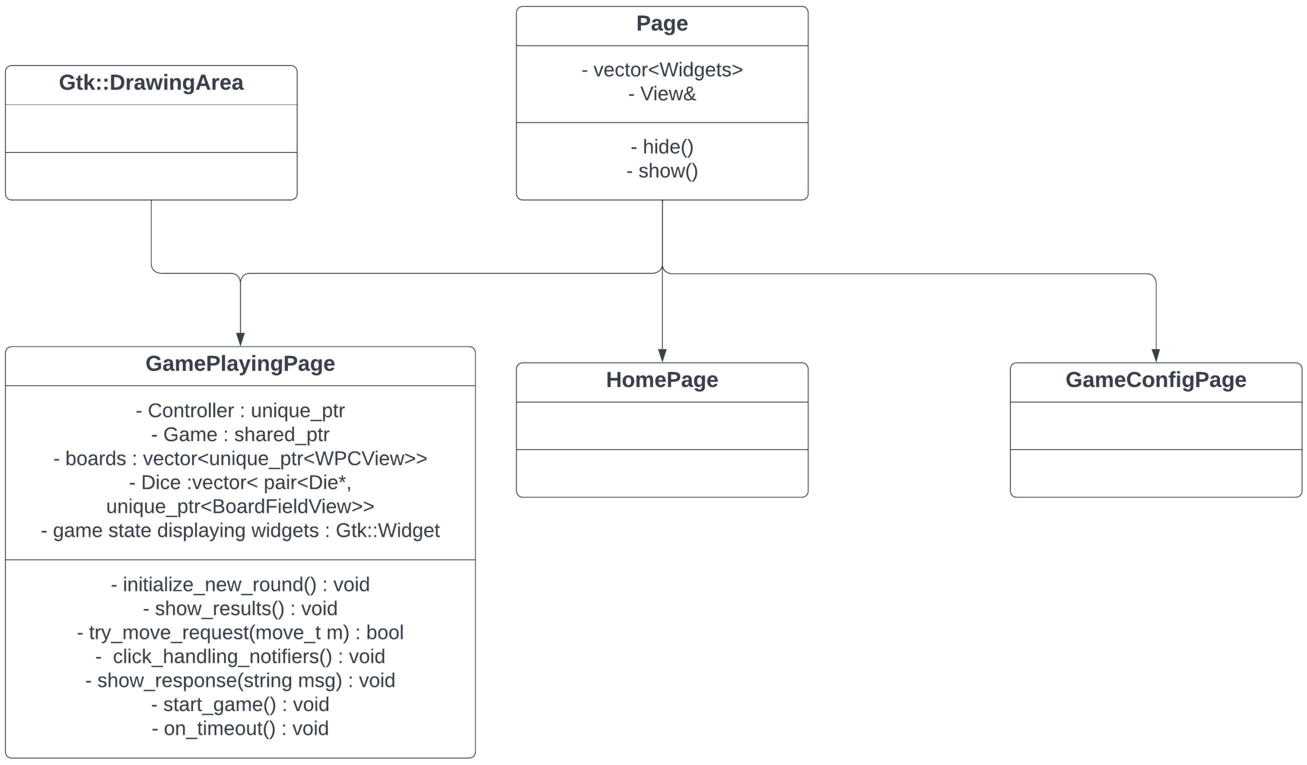
Using some of the tool cards requires extra settings. The reason is that they could consist of more than one move and after a given move, the tool card has to be used and there are no other options (e.g. placing any die on the board). This is handled by declaring flags whether the logic behind accepts clicks from given components. These flags are: `bool acceptDieClick`, `bool acceptBoardFieldClick`, `bool acceptRoundTrackClick`.

This class handles all the user interaction.

For each game, a new GamePlayingPage object is constructed.

Most important methods:

- **void board_field_clicked()**, **void handle_round_track_die_click()** - Since the objects such as the round track and the boards are derived from Gtk::DrawingArea, they receive the button_press event on their own and have to notify the page for getting an event.
- **bool try_move_request(move_t m)** - called by ToolCardView objects when a user move is detected. It sends the request to the **controller** object and handles the result by showing a response ( **show_response()** function) on failure and resetting the state in case of success.
- **void show_results()** - This method is called when the game has ended and the results of the game have to be shown.


Gtk::DrawingArea
Page
- vector<Widgets>
- View&
- hide()
- show()
GamePlayingPage
- Controller : unique_ptr
- Game : shared_ptr
- boards : vector<unique_ptr<WPCView>>
- Dice :vector< pair<Die*, unique_ptr<BoardFieldView>>
- game state displaying widgets : Gtk::Widget
- initialize_new_round() : void
- show_results() : void
- try_move_request(move_t m) : bool
- click_handling_notifiers() : void
- show_response(string msg) : void
- start_game() : void
- on_timeout() : void
HomePage
GameConfigPage

---

## BoardFieldView

This class is used to represent both fields on the board and dice for different purposes such as the round track or the current rounds dice. An instance of this object is constructed with a position and size.

---

## CardDisplayer

The different cards are displayed using this class. Each type of card (Tool cards, Public/Private objective cards) displays its type of icon. When the user clicks on the icon, the cards are displayed. This class is derived from Gtk::EventBox because they display the cards by pre-drawn images and they have to accept events that Gtk::Image is not capable of.

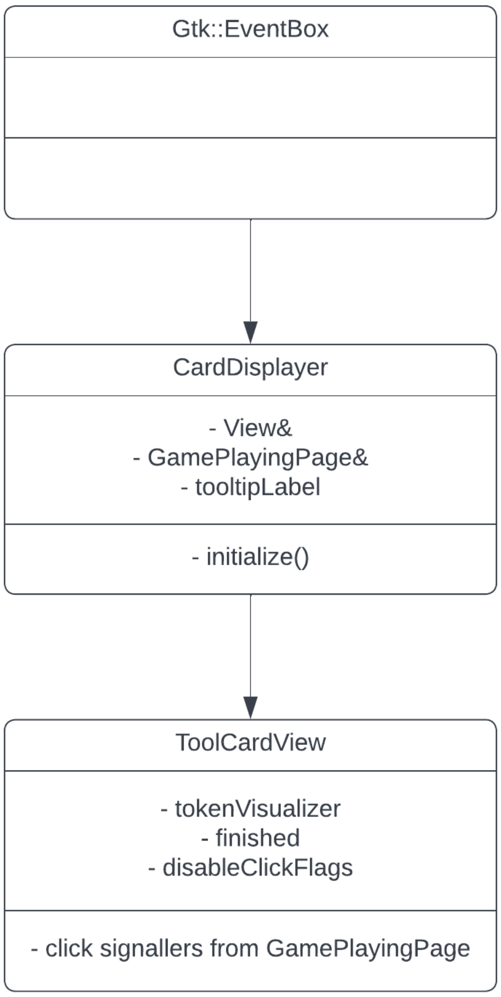
Most important methods:

- **on_enter_notify_event()**, **on_leave_notify_event()** - GTK event handlers that are invoked when the user moves the mouse over the icon or leaves it. It displays a tooltip that displays what kind of cards the icon refers to.
- **on_button_press_event()** - A handler for a GTK event that occurs when the user clicks the icon. It displays all the cards that they refer to.
- **initialize()** - according to the type of cards that the icon refers to, this method receives those cards as an argument from the Game class

**ToolCardView**

Each tool card has a View class that represents how it reacts to user interaction. If a tool card is selected, all the board/round track/dice clicks are forwarded to the tool card's View handler and it handles what happens when these events occur.

These classes have the following inheritance structure:



---

# Controller

## GameFlowController

This class serves as a middle layer between the `Model` and `View` components. It is called by View's timeout handler when the human player has made a move to forward it to the `Game` class and when it's an AI player's turn to call it to make a move. There are approximately 1500 lines of code in the controller component including the `Logger` and the `Command parser` libraries.

Most important methods:

- **void make_move_request(move_t m)** - called with human player's move
- **void make_ai_player_move()** - called when its an AI player's turn
- **shared_ptr<Game> start_local(size_t AIPlayerCount)** - called when the user has selected the last configuration setting before starting the game

## Tests

The unit tests of the project could be found in the tests directory. The project uses the `GoogleTest` testing framework.

Since the game uses a random configuration for each run, the tests include random tests as well. The game instances that the tests are run on are randomly initialized by default. However, testing specific parts of the game requires constructing the game from a configuration. This is provided by defining support for constructing game objects from **JSON** configuration.

### Configuration

The optional configuration file's name could be passed as a command-line argument after the `config_filename` keyword. The **bool SagradaTest::Init(int argc, char** argv)** function processes the configuration file and constructs a `GameConfig` object from it. If the configuration file uses incorrect format, the **Init()** function returns false and the tests are not run.

Example of running the tests with a configuration file named `MyTestConfig.json`

```
build/tests --config_filename tests/MyTestConfig.json
```

The configuration file must be in the following format:

- Public objective cards - receives an array of PuOC identifiers. The array is defined for key "puocIDs". It may contain any number of identifiers that could be used in the game (i.e. at most PublicObjectiveCards::cardsPerGame) including 0. The remaining IDs are filled randomly. Correctness of the IDs is checked in the **SagradaTest::Init()** method returning false for invalid IDs. The IDs must be in range [0, PublicObjectiveCards::totalCardCount).
- Tool cards - the same rules apply to Tool cards as for the Public objective cards with the correct constants and with the JSON key "tcIDs".
- Players - receives an array of PlayerInformation objects. The key for defining the player configuration array is "players". Each defined player must contain the PrivateObjectiveCard object defined inside the configuration. The **SagradaTest::Init()** function returns false if there is a player configuration without the PrivateObjectiveCard object defined. An additional
- Starting player ID - it could be defined using the key "playerStartingFirstRoundID". It accepts a single non-negative integer indicating which player will start the first round.

An example configuration that defines two Public objective cards (meaning that one will be chosen randomly) and two players having their Private objective cards color purple and red. The players have IDs 0 and 1. The configuration doesn't mention the Tool cards so they are chosen randomly.

```
{
    "puocIDs": [
        0,
        8
    ],

    "players": [
        {
            "ID": 0,
            "PrivateObjectiveCard": {
                "Color": "Purple"
            }
        },
        {
            "ID": 1,
            "PrivateObjectiveCard": {
                "Color": "Red"
            }
        }
    ]
}
```