chess

第一阶段

围棋和五子棋基本规则

围棋参考 Tromp-Taylor 规则：

- 棋盘被分为黑白色和空三种状态。
- 一个点 P 被称为到达点 C，如果存在一条路径，由 P 的颜色相邻（垂直或者水平）的路径从 P 到达 C。
- 清除点是将所有未达到空的该颜色点位清空的过程。
- 一个回合可以是跳过或者移动，但不能重复之前的格子着色（superko）。
- **一次移动包括将一个空点涂上自己的颜色；然后清除对手的颜色，最后清除自己的颜色。**
- 游戏在连续两次跳过后结束。
- 一个玩家的得分是他颜色的点数加上只能被他颜色到达的空白点数。

五子棋规则：垂直、水平或者斜线五子成线（或者大于五子）

其它额外的规则：支持悔棋、重开、认输、自动判负

BaseBoardGame

从上面的描述抽象出基类 BaseBoardGame，包括游戏名称、棋盘大小、轮次、历史记录等属性，实现了投降、悔棋、重开等方法，同时需要子类实现每一轮操作的抽象类 move。

```python
class BaseBoardGame(ABC):
    def __init__(self, size: int):
        self.name = ""
        self.size = size
        self.board = [[Color.EMPTY for _ in range(size)] for _ in range(size)]
        self.round = 0
        self.game_over = False
        self.winner = ""
        self.history: list[Memento] = []  # Use a list of Mementos for the history

    def cur_player(self) -> Color:
        return Color.BLACK if self.round % 2 == 0 else Color.WHITE

```

```
    @abstractmethod
    def move(self, coord: tuple[int, int] | None = None):
        raise NotImplementedError

    def surrender(self):
        self.game_over = True
        self.winner = "White" if self.cur_player() == Color.BLACK else "Black"
        logger.info(f"{self.winner} wins.")

    def regret(self):
        if len(self.history) > 0:
            self.restore_from_memento(self.history.pop())
            logger.info("Move undone.")
        else:
            self.restart()

    def restart(self):
        self.__init__(self.size)

    @abstractmethod
    def create_memento(self) -> Memento:
        raise NotImplementedError

    @abstractmethod
    def restore_from_memento(self, memento: Memento):
        raise NotImplementedError

    @abstractmethod
    def save_to_file(self, file_path: str):
        raise NotImplementedError

    @abstractmethod
    def load_from_file(self, file_path: str):
        raise NotImplementedError
```

由于还需要支持局面的保存与读取功能，因此采用备忘录模式，Memento 类作为快照，BaseBoardGame 作为 Originator 和 Caretaker。

```
class Memento:
    def __init__(self, state: dict):
        self.__state = state

    def get_saved_state(self):
        return self.__state
```

Go Game

- 一次 move 需要做的事情是
  - 判断游戏是否结束，如果结束则不能继续下棋。
  - 是否是重复上一次格点染色（superko），通过一个 self.ko_point 记录
  - 将当前状态加入到历史记录，以便实现悔棋功能
  - 对格点着色
  - 对于着色点周围不同颜色判断是否需要清除（clear 函数）
  - 对自身进行清除
  - 记录是否是 ko_point
- clear 函数通过 string 函数找到相连的同色的 group，通过 liberties 函数判断是否要清除
- string 函数通过 BFS 找到同色的相连的 group
- liberties 通过遍历每个点的邻居点是否是空判断是否有“气”
- score 分数是点数+领地数量，领地数量通过洪水算法判断是否是只被一种颜色到达

```python
class GoGame(BaseBoardGame):
    # Rule 1: Go is played on a 19x19 square grid of points, by two players called Black and White.
    def __init__(self, size: int):
        # Rule 2: Each point on the grid may be colored black, white or empty.
        super().__init__(size)
        self.name = "Go Game"
        self.komi = 6.5
        self.ko_point: tuple[int, int] | None = None
        self.last_move_captured = None
        self.abstention = 0
        self.final_score = ""

    def move(self, coord: tuple[int, int] | None = None):
        if self.game_over:
            return
        if coord is not None:
            # Rule 6: A turn is either a pass or a move that doesn't repeat an earlier grid coloring (superko).
            if self.ko_point == coord:
                logger.info("Cannot recapture ko immediately.")
                return

            self.history.append(self.create_memento())

            # Rule 5: Starting with an empty grid, the players alternate turns, starting with Black.
            current_color = self.cur_player()
            opposite_color = Color.WHITE if current_color == Color.BLACK else Color.BLACK
            x, y = coord
```

```
            # Rule 7: A move consists of coloring an empty point one's own color; then clearing the opponent color, and then clearing one's own color.
            self.board[x][y] = current_color
            opponent = [nbr for nbr in self.neighbors(coord) if self.board[nbr[0]][nbr[1]] == opposite_color]
            captured = self.clear(opponent)

            # set ko point
            if len(captured) == 1:
                self.ko_point = captured.pop()
            else:
                self.ko_point = None

            self.clear(coord)
        else:  # pass
            self.history.append(self.create_memento())
            self.abstention += 1
            # Rule 8: The game ends after two consecutive passes.
            if self.abstention == 2:
                # Rule 10: The player with the higher score at the end of the game is the winner. Equal scores result in a tie.
                black_score, white_score = self.score()
                logger.info(f"Game over. Black: {black_score}, White: {white_score}")
                self.game_over = True
                self.winner = f"Black" if black_score > white_score else f"White" if black_score < white_score else f"Tie"
                self.final_score = f"Black: {black_score}, White: {white_score}"

        self.round += 1

    def create_memento(self) -> Memento:
        # Save the current state in a memento
        state = {
            "name": self.name,
            "size": self.size,
            "board": copy.deepcopy(self.board),
            "round": self.round,
            "game_over": self.game_over,
            "winner": self.winner,
            "final_score": self.final_score,
            # "history": copy.deepcopy(self.history),
            "komi": self.komi,
            "ko_point": self.ko_point,
            "last_move_captured": self.last_move_captured,
            "abstention": self.abstention,
        }
        return Memento(state)

    def restore_from_memento(self, memento: Memento):
        # Restore the state from the memento
        state = memento.get_saved_state()
```

```
        self.name = state["name"]
        self.size = state["size"]
        self.board = state["board"]
        self.round = state["round"]
        self.game_over = state["game_over"]
        self.winner = state["winner"]
        self.final_score = state["final_score"]
        # self.history = state["history"]
        self.komi = state["komi"]
        self.ko_point = state["ko_point"]
        self.last_move_captured = state["last_move_captured"]
        self.abstention = state["abstention"]

    def save_to_file(self, file_path: str):
        with open(file_path, "wb") as file:
            pickle.dump(self.create_memento(), file)
            logger.info("Game saved to file.")

    def load_from_file(self, file_path: str):
        with open(file_path, "rb") as file:
            self.restore_from_memento(pickle.load(file))
            logger.info("Game loaded from file.")

    def neighbors(self, coord: tuple[int, int]) -> list[tuple[int, int]]:
        x, y = coord
        return [(x, ny) for ny in (y - 1, y + 1) if 0 <= ny < self.size] + [(nx, y) for nx in (x - 1, x + 1) if 0 <= nx < self.size]

    # Rule 4: Clearing a color is the process of emptying all points of that color that don't reach empty.
    def clear(self, points: tuple[int, int] | list[tuple[int, int]]) -> set[tuple[int, int]]:
        if isinstance(points, tuple):
            points = [points]
        captured = set()
        for pt in points:
            str_points = self.string(pt)
            if not self.liberties(str_points):
                captured.update(str_points)
        for cap in captured:
            self.board[cap[0]][cap[1]] = Color.EMPTY
        return captured

    # Rule 3: A point P, not colored C, is said to reach C if there is a path of (vertically or horizontally) adjacent points of P's color from P to a point of color C.
    def string(self, coord: tuple[int, int]) -> list[tuple[int, int]]:
        visited = set()
        visited.add(coord)

        queue = deque([coord])

```

```
        while queue:
            for nbr in self.neighbors(queue.popleft()):
                if self.board[nbr[0]][nbr[1]] == self.board[coord[0]][coord[1]] and nbr not in visited:
                    visited.add(nbr)
                    queue.append(nbr)

        return visited

    def liberties(self, group: list[tuple[int, int]]) -> bool:
        empty = [nbr for p in group for nbr in self.neighbors(p) if self.board[nbr[0]][nbr[1]] == Color.EMPTY]
        return len(empty) > 0

    def calculate_territory(self) -> tuple(set[tuple[int, int]], set[tuple[int, int]]):
        black_territory = set()
        white_territory = set()
        neutral_territory = set()
        visited = set()

        for x in range(self.size):
            for y in range(self.size):
                if (x, y) in visited or self.board[x][y] != Color.EMPTY:
                    continue

                territory, borders = self.flood_fill((x, y))
                visited.update(territory)

                # Determine the territory's ownership by its borders
                if all(self.board[x][y] == Color.BLACK for x, y in borders):
                    black_territory.update(territory)
                elif all(self.board[x][y] == Color.WHITE for x, y in borders):
                    white_territory.update(territory)
                else:
                    neutral_territory.update(territory)

        return black_territory, white_territory

    def remove_dead_stones(self) -> tuple[int, int]:
        return 0, 0

    def flood_fill(self, start: tuple[int, int]):
        queue = deque([start])
        territory = set([start])
        borders = set()

        while queue:
            x, y = queue.popleft()
            for nx, ny in self.neighbors((x, y)):
                if (nx, ny) in territory:
                    continue
```

```
                if self.board[nx][ny] == Color.EMPTY:
                    queue.append((nx, ny))
                    territory.add((nx, ny))
                else:
                    borders.add((nx, ny))

        return territory, borders

    def score(self):
        # Rule 9: A player's score is the number of points of her color, plus the number of empty points that reach only her color.
        black_captures, white_captures = self.remove_dead_stones()
        black_territory, white_territory = self.calculate_territory()

        black_score = len(black_territory) + white_captures
        white_score = len(white_territory) + black_captures + self.komi

        return black_score, white_score
```

Gomoku Game

一次 move 操作只包含染色以及判断垂直、水平和对角线方向是否大于五子成线。

```
class GomokuGame(BaseBoardGame):
    def __init__(self, size: int):
        super().__init__(size)
        self.name = "Gomoku Game"

    def move(self, coord: tuple[int, int] | None = None):
        if self.game_over:
            return
        if coord is None:
            logger.warning("Coord cannot be None.")
            return
        self.history.append(self.create_memento())
        x, y = coord
        self.board[x][y] = self.cur_player()
        if self.is_five(coord):
            self.game_over = True
            logger.info(f"{self.cur_player().value} wins.")
        self.round += 1

    def is_five(self, coord: tuple[int, int]) -> bool:
        directions = [(1, 0), (0, 1), (1, 1), (1, -1)]
        for d in directions:
            if self.count_in_direction(coord, d[0], d[1]) + self.count_in_direction(coord, -d[0], -d[1]) - 1 >= 5:
                return True
        return False

```

```
    def count_in_direction(self, start: tuple[int, int], dx: int, dy: int) -> int:
        count = 0
        x, y = start
        while 0 <= x < self.size and 0 <= y < self.size and self.board[x][y] ==
self.cur_player():
            count += 1
            x += dx
            y += dy
        return count

    def create_memento(self) -> Memento:
        # Save the current state in a memento
        state = {
            "name": self.name,
            "size": self.size,
            "board": copy.deepcopy(self.board),
            "round": self.round,
            "game_over": self.game_over,
            "winner": self.winner,
            # "history": copy.deepcopy(self.history),
        }
        return Memento(state)

    def restore_from_memento(self, memento: Memento):
        # Restore the state from the memento
        state = memento.get_saved_state()
        self.name = state["name"]
        self.size = state["size"]
        self.board = state["board"]
        self.round = state["round"]
        self.game_over = state["game_over"]
        self.winner = state["winner"]
        # self.history = state["history"]

    def save_to_file(self, file_path: str):
        with open(file_path, "wb") as file:
            pickle.dump(self.create_memento(), file)
            logger.info("Game saved to file.")

    def load_from_file(self, file_path: str):
        with open(file_path, "rb") as file:
            self.restore_from_memento(pickle.load(file))
            logger.info("Game loaded from file.")
```

GUI 设计

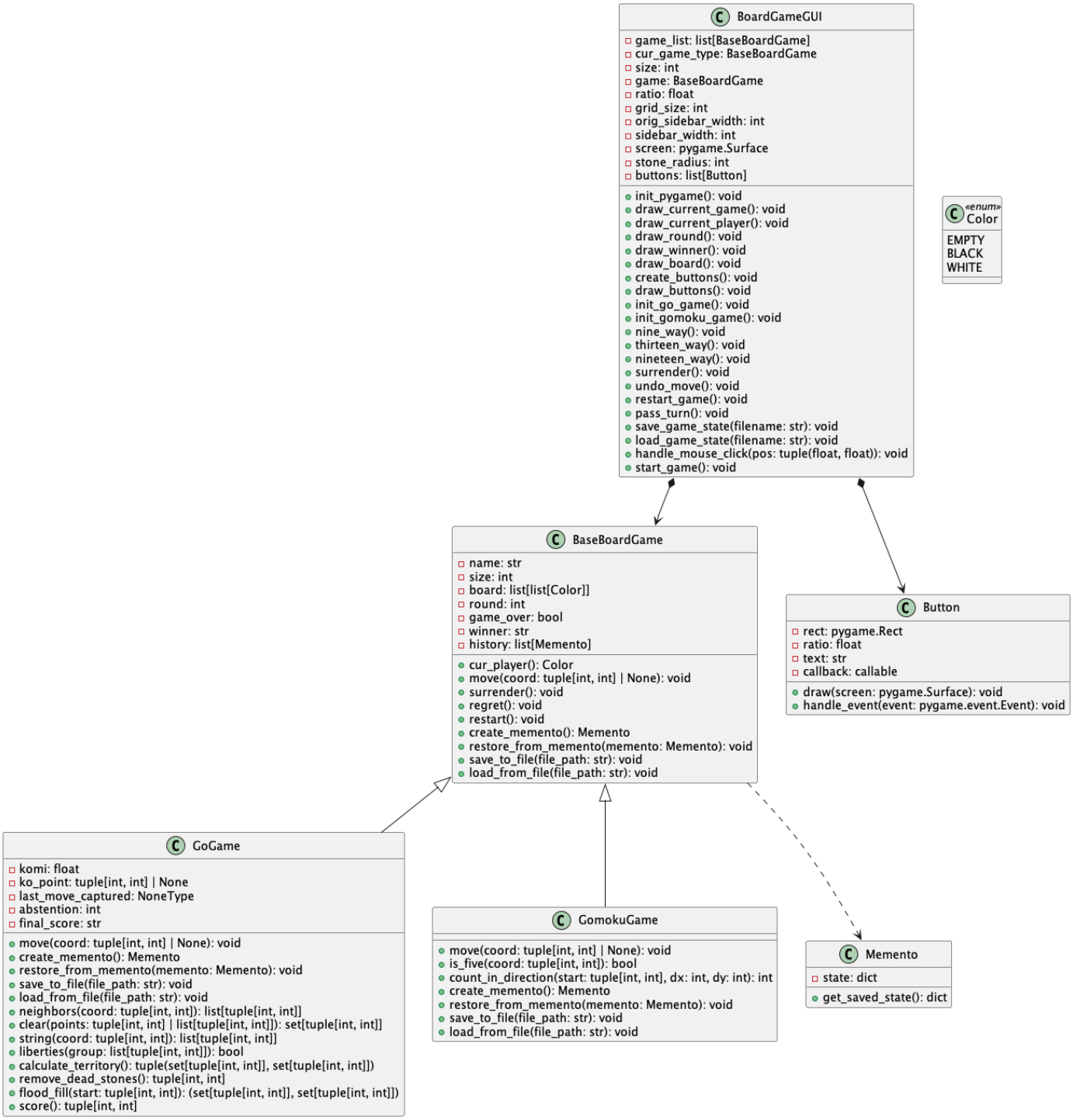
GUI和后端算法的分离采用策略模式，BaseBoardGame 基类定义了一系列的算法或行为，比如 move、regret、restart等等，这些都是抽象的操作。任何继承自 BaseBoardGame 的子类都必须提供这些抽象方法的具体实现。GUI 类则充当策略模式中的上下文（Context），它不关心具体的实现细节，只依赖于BaseBoardGame 基类的接口。

GUI 的代码比较冗长无聊，主要是在 start_game 体现策略模式，这里都使用 BaseBoardGame 的抽象方法，这样可以轻松地更换不同的游戏逻辑（即 BaseBoardGame 的不同子类），而 GUI 类不需要做任何改变。这提高了代码的复用性和灵活性，同时也遵循了设计原则中的“依赖倒置原则”（Dependence Inversion Principle），即高层模块不应该依赖于低层模块，两者都应该依赖于抽象。

```
class BoardGameGUI:
    # ...

    def surrender(self):
        self.game.surrender()

    def undo_move(self):
        self.game.regret()

    def restart_game(self):
        self.game.restart()

    def pass_turn(self):
        self.game.move(None)

    def save_game_state(self, filename):
        self.game.save_to_file(filename)

    def load_game_state(self, filename):
        self.game.load_from_file(filename)

    def start_game(self):
        running = True
        while running:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False
                elif event.type == pygame.MOUSEBUTTONDOWN:
                    if event.button == 1:
                        for button in self.buttons:
                            button.handle_event(event)
                        pos = pygame.mouse.get_pos()
                        self.handle_mouse_click(pos)
                elif event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_u:
                        self.undo_move()
                    elif event.key == pygame.K_r:
                        self.restart_game()
                    elif event.key == pygame.K_p:
                        self.pass_turn()
                    elif event.key == pygame.K_s:
                        self.save_game_state("game_state.pickle")
                    elif event.key == pygame.K_l:
                        self.load_game_state("game_state.pickle")
                    elif event.key == pygame.K_q:
                        running = False
```

```
                self.create_buttons()
                self.draw_board()
                self.draw_current_game()
                self.draw_current_player()
                self.draw_round()
                self.draw_winner()
                self.draw_buttons()
            pygame.display.flip()

        pygame.quit()
        sys.exit()


if __name__ == "__main__":
    gui = BoardGameGUI()
    gui.start_game()
```

UML 图


BoardGameGUI
- game_list: list[BaseBoardGame]
- cur_game_type: BaseBoardGame
- size: int
- game: BaseBoardGame
- ratio: float
- grid_size: int
- orig_sidebar_width: int
- sidebar_width: int
- screen: pygame.Surface
- stone_radius: int
- buttons: list[Button]
+ init_pygame(): void
+ draw_current_game(): void
+ draw_current_player(): void
+ draw_round(): void
+ draw_winner(): void
+ draw_board(): void
+ create_buttons(): void
+ draw_buttons(): void
+ init_go_game(): void
+ init_gomoku_game(): void
+ nine_way(): void
+ thirteen_way(): void
+ nineteen_way(): void
+ surrender(): void
+ undo_move(): void
+ restart_game(): void
+ pass_turn(): void
+ save_game_state(filename: str): void
+ load_game_state(filename: str): void
+ handle_mouse_click(pos: tuple(float, float)): void
+ start_game(): void
«enum»
Color
EMPTY
BLACK
WHITE
BaseBoardGame
- name: str
- size: int
- board: list[list[Color]]
- round: int
- game_over: bool
- winner: str
- history: list[Memento]
+ cur_player(): Color
+ move(coord: tuple[int, int] | None): void
+ surrender(): void
+ regret(): void
+ restart(): void
+ create_memento(): Memento
+ restore_from_memento(memento: Memento): void
+ save_to_file(file_path: str): void
+ load_from_file(file_path: str): void
Button
- rect: pygame.Rect
- ratio: float
- text: str
- callback: callable
+ draw(screen: pygame.Surface): void
+ handle_event(event: pygame.event.Event): void
GoGame
- komi: float
- ko_point: tuple[int, int] | None
- last_move_captured: NoneType
- abstention: int
- final_score: str
+ move(coord: tuple[int, int] | None): void
+ create_memento(): Memento
+ restore_from_memento(memento: Memento): void
+ save_to_file(file_path: str): void
+ load_from_file(file_path: str): void
+ neighbors(coord: tuple[int, int]): list[tuple[int, int]]
+ clear(points: tuple[int, int] | list[tuple[int, int]]): set[tuple[int, int]]
+ string(coord: tuple[int, int]): list[tuple[int, int]]
+ liberties(group: list[tuple[int, int]]): bool
+ calculate_territory(): tuple(set[tuple[int, int]], set[tuple[int, int]])
+ remove_dead_stones(): tuple[int, int]
+ flood_fill(start: tuple[int, int]): (set[tuple[int, int]], set[tuple[int, int]])
+ score(): tuple[int, int]
GomokuGame
+ move(coord: tuple[int, int] | None): void
+ is_five(coord: tuple[int, int]): bool
+ count_in_direction(start: tuple[int, int], dx: int, dy: int): int
+ create_memento(): Memento
+ restore_from_memento(memento: Memento): void
+ save_to_file(file_path: str): void
+ load_from_file(file_path: str): void
Memento
- state: dict
+ get_saved_state(): dict