객체지향설계 (00반)

Term Project : 오델로 게임 구현

조원

201701981 김기환

201802064 김성훈

201802120 윤지현

목차

1. 구현/미 구현 기능

(A) 오델로 게임의 규칙

처음에 판 가운데에 사각형으로 엇갈리게 배치된 돌 4 개를 놓고 시작한다.
돌은 반드시 상대방 돌을 양쪽에서 포위하여 뒤집을 수 있는 곳에 놓아야 한다.
돌을 뒤집을 곳이 없는 경우에는 차례가 자동적으로 상대방에게 넘어가게 된다.
아래와 같은 조건에 의해 양쪽 모두 더 이상 돌을 놓을 수 없게 되면 게임이 끝나게 된다.

64 개의 돌 모두가 판에 가득 찬 경우 (가장 일반적)
어느 한 쪽이 돌을 모두 뒤집은 경우
한 차례에 양 쪽 모두 서로 차례를 넘겨야 하는 경우

게임이 끝났을 때 돌이 많이 있는 플레이어가 승자가 된다. 만일 돌의 개수가 같을 경우는 무승부가 된다.

(B) 구현한 기능

오델로 게임의 기능을 아래와 같이 분할하였습니다.

[게임 시작 부분]

a. 오델로 게임을 시작하기 전 게임 보드의 사이즈 또는 게임 종료 명령어를 입력 받는 기능
b. 유저가 게임 보드의 사이즈를 입력할 경우 사이즈에 맞는 보드가 생성되며 오델로 게임이 시작되는 기능
c. 유저가 게임 종료 명령어를 입력하기 전까지 계속해서 새로운 오델로 게임을 진행

[게임 진행 부분]

a. 유저가 오델로 게임의 사이즈를 입력할 경우 하나의 오델로 게임이 진행

[게임 종료 부분]

a. 게임 시작 부분에서 유저가 게임 종료 명령어를 입력할 경우 프로그램이 종료

(C) 게임 진행 부분

게임 진행 부분의 기능들을 아래와 같이 나눌 수 있습니다.

1. 현재 유저가 보드에 돌을 둘 수 있는지 검사하는 기능
2. 유저로부터 보드에 둘 돌의 위치를 입력 받는 기능
3. 입력한 돌에 의해 오델로 규칙에 따라 보드의 돌들을 뒤집는 기능

4. 유저의 승패를 검사하는 기능
5. 현재 보드의 상태를 출력하는 기능
6. 다음 유저에게 입력 턴이 전환되는 기능
7. 현재 게임의 상태를 확인하고 하나의 오델로 게임이 종료되는 기능

2. 사용한 디자인 패턴

(A) Behavioral : state pattern

- state pattern 이란?
  상태 기계를 구현하는 디자인 패턴

- state pattern 의 구조
  Context

  concrete sate object 중 하나를 저장합니다.
  해당 state 에 작업을 위임합니다.
  context 는 state interface 를 통해 state object 의 기능을 사용합니다.
  context 는 새로운 state object 를 전달하기 위해 setter 를 사용합니다.

  State

  특정 상태에서 동작할 기능을 선언합니다.
  모든 concrete states 에서 공통되게 사용할 수 있는 동작

  Concrete states

  state-specific(특정-상태) method 를 각각의 concrete state 별로의 동작을 수행하는 기능입니다.
  state object 들은 context 객체에 대한 역참조(backreference)를 저장해 해당 reference 를 통해 state 는 context object 로부터 필요한 정보를 가지고와 동작합니다.

- state pattern 을 사용하는 이유
  1-c 의 게임 진행부분의 기능들을 통해 오델로 게임의 진행과정은 다음과 같습니다.

  유저가 돌을 둘 수 있는 지 확인합니다. 둘을 둘 수 있는 경우 보드에 돌의 위치를 입력받습니다. 이후 입력한 돌에 의해 오델로 규칙에 따라 보드의 돌들이 뒤집히게 되고 뒤집힌 보드를 통해 승패가 결정되었는지 확인하며 다음 유저에게 턴이 전환되거나 게임이 종료되게 됩니다.

  각각의 기능들이 서로 유기적으로 연결되어 있으며 현재 보드의 상태에 따라 다음 동작이 어떤

식으로 진행될지 결정됨을 알 수 있습니다. 따라서 각각의 상태 간의 관계를 표현하는데 효과적인 sate pattern 을 사용합니다.

현재 기능들을 각각의 상태에 따라 정리 하면 아래와 같습니다.

1. 초기상태
   유저가 돌을 뒤집을 수 있는 위치를 모두 계산합니다.

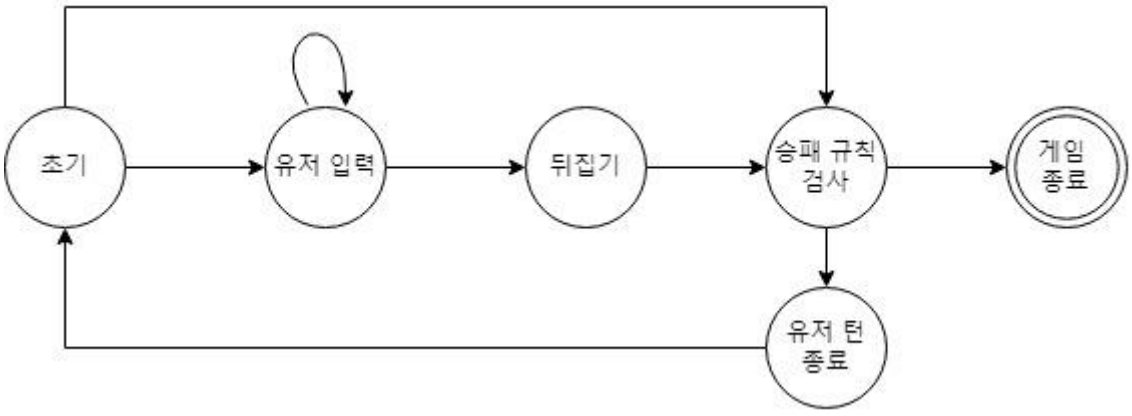
2. 유저 입력상태
   유저로부터 돌을 입력 받습니다.
   초기상태에서 구한 뒤집을 수 있는 돌의 위치에 해당하는지 확인합니다.

3. 뒤집기 상태
   입력 받은 돌의 위치를 통해 보드의 돌들을 뒤집어줍니다.

4. 승패 규칙 검사 상태
   현재 보드의 정보를 통해 승패가 결정이 되었는지 검사합니다.

5. 유저 턴 종료 상태
   유저 한 명의 턴(입력)이 종료되었기에 다음 유저에게 입력 권한을 부여합니다.

6. 게임 종료 상태
   각각의 상태들의 진행 과정을 flow chart 로 작성하면 아래와 같습니다.



위 그림을 통해 flow chart 가 하나의 유한 오토마타(FSM-유한 상태기계) 구조가 됨을 알 수 있습니다. 따라서 FSM 구조를 사용하는 state pattern 을 사용해 오델로 게임의 Behavioral pattern 을 구현합니다.

- Context

현재 게임의 상태를 관리하는 Context 객체에서 현재 상태 정보를 저장해 두었다가 각각의

상태에 따른 동작(Action) 을 수행합니다. 코드는 아래와 같습니다.

```
void Context::Action() {
    State *cur_state = nullptr;
    switch (this->state_) {
        case kStateUserInput:
            // 유저 입력 상태
            cur_state = &UserInputState::GetInstace(this);
            break;
        case kStateCoordCheck:
            // 초기 상태
            cur_state = &CoordCheckState::GetInstace(this);
            break;
        case kStateUpdateBoard:
            // 뒤집기 상태
            cur_state = &UpdateBoardState::GetInstace(this);
            break;
        case kStateWinnercheck:
            // 승패 규칙 검사 상태
            cur_state = &WinnerCheckState::GetInstace(this);
            break;
        case kStateFinal:
            // 현재 사용자 차례 종료 상태
            cur_state = &FinalState::GetInstace(this);
            break;
        case kStateGameEnd:
            // 게임 종료 상태
            cur_state = &GameEndState::GetInstace(this);
            break;
        default:
            cur_state = nullptr;
            break;
    }
    // 각 상태의 Action 실행
    if (cur_state != nullptr) {
        cur_state->Action();
    }
}
```

Context object 의 state_ 멤버 변수에서 현재 상태에 대한 정보를 가지고 있으며 state_ 멤버 변수의 값에 따라 상태의 Action() 멤버 함수가 실행됨을 알 수 있습니다.

- State

각각의 상태에 대한 base class 가 되며 특정 상태에 따른 동작을 수행할 Action() 멤버 함수를 pure virtual function 으로 선언해 abstract class 로 만들어 줍니다. 상태들이 State class 를 상속받아 Action() 함수를 overriding 해 각각의 상태에 따른 동작을 구현하게 됩니다.

```
class State {
 public:
    explicit State(Context *, StateType);

/*
```

```
**      각 상태 별로 해야하는 작업들, derived class 에서 각각의 Action 함수를 override
**      Context 의 Action 기능을 호출 했을 때 State 들에 맞게 Action 이 호출 됩니다.
*/
    virtual void Action() = 0;
    ~State();

 protected:
    Context *context_;
    StateType type_;

 private:
    /* data */
};
```

- Concrete states

  State class 를 상속받아 각각의 상태에 맞는 동작(Action()) 을 수행하도록 overriding 함을 알 수 있습니다.

```
class CoordCheckState : public State {
 public:
    static CoordCheckState &GetInstace(Context *);

    // 가능한 좌표와 그 좌표에서 바꿀수 있는 범위들을 모두 계산하고
    // context 에 저장
    void Action() override;

 private:
    static std::array<std::array<int, 2>, 8> mover_;
    ~CoordCheckState();
    explicit CoordCheckState(Context *);
    struct ValidCoord  CalculateValidCoord(int, int);
};
```

(B) Creational : Singleton pattern

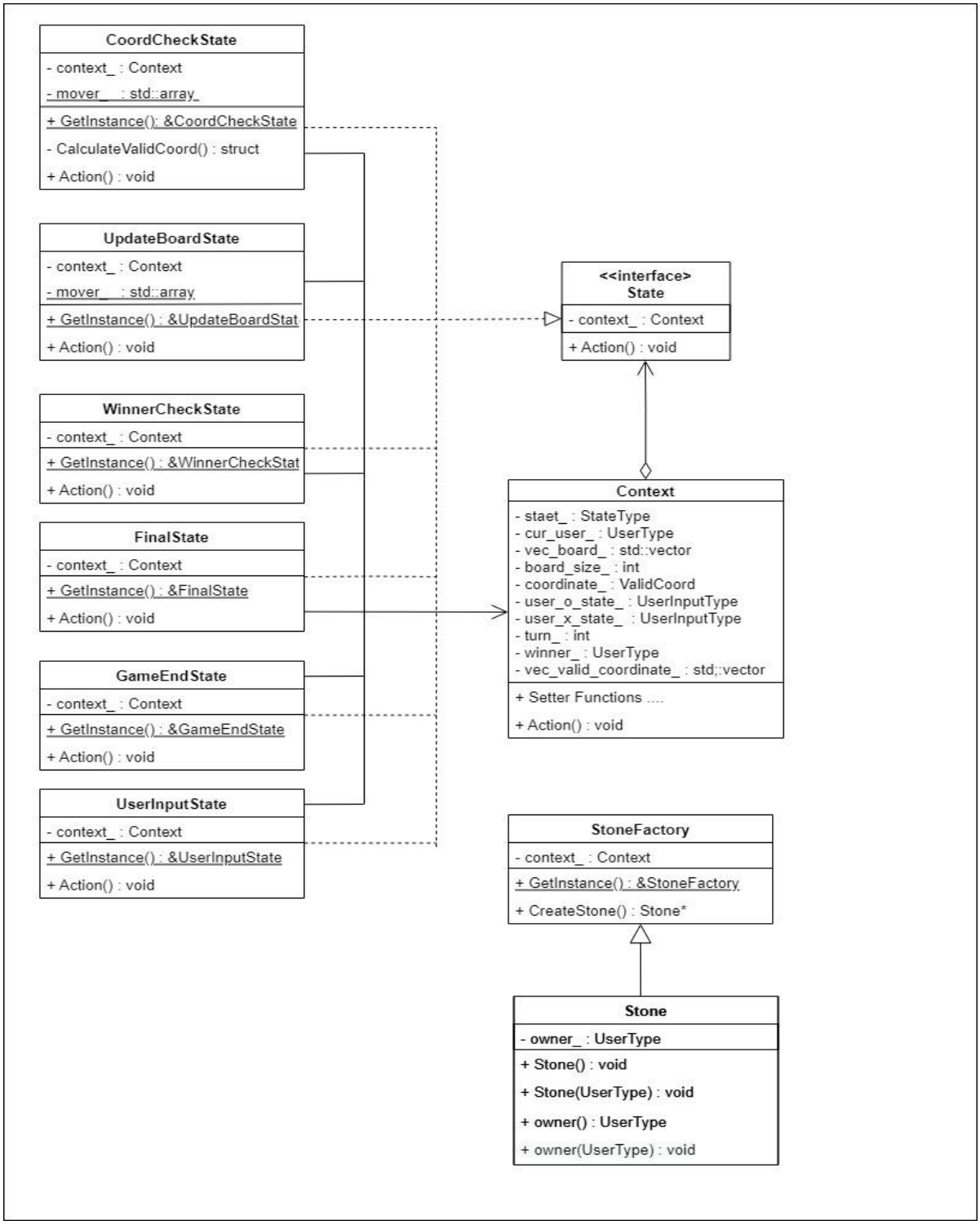
State pattern 에서 사용되는 상태들은 게임 내에서 유일한 상태이어야 합니다. 따라서 프로그램 내에서 하나의 object 만 존재하도록 구현하는 singleton pattern 을 사용해 각각의 상태들을 구현합니다. singleton pattern 을 적용해 상태를 구현한 코드는 Concrete states 항목과 같습니다.

(C) Creational : Factory pattern

유저의 입력에 따라 보드의 크기와 생성되는 돌의 개수 또한 결정됩니다. 유저의 입력에 따라 돌(object) 을 동적으로 할당해주어야 합니다. 따라서 factory pattern 을 사용해 돌(object) 의 동적 할당만을 담당하는 Factory를 생성해줍니다.

3. Class 다이어그램

디자인 패턴을 사용해 클래스의 다이어그램을 구성하면 아래와 같습니다.



4. 예외 상황과 예외처리 방법

(A) 메모리 누수 문제

a. 싱글턴 패턴 방식의 메모리 누수

기존의 싱글턴 패턴 방식의 경우 포인터를 활용했었습니다. 프로그램 실행과정에서 해당 메모리를 제거할 일이 없기 때문에 메모리 누수문제가 불가피하였습니다.

```
CoordCheckState &CoordCheckState::GetInstace(Context *context) {
    static CoordCheckState instance(context);
    return (instance);
}
```

위 코드와 같이 포인터를 활용해 메모리를 동적 할당하는 방식을 사용하지 않고 static 변수를 선언하는 코드를 GetInstance() 함수안에 작성하여 local static 변수가 되도록 구현해주었습니다.

b. 동적할당한 메모리 해제

프로그램에서 동적할당을 수행하는 부분은 board 의 구성요소인 stone 을 동적할당해 생성하는 부분 밖에 없습니다. board 는 context 의 멤버 변수이고 context 의 destructor 가 호출될 때 board 의 stone 들을 delete 해 메모리를 해제해주어 메모리 누수를 방지합니다.

valgrind 를 활용한 메모리 누수 체크 결과

```
valgrind --leak-check=full --show-reachable=yes ./othello
```

```
==6254==
==6254== HEAP SUMMARY:
==6254==     in use at exit: 0 bytes in 0 blocks
==6254==   total heap usage: 103 allocs, 103 frees, 77,492 bytes allocated
==6254==
==6254== All heap blocks were freed -- no leaks are possible
==6254==
==6254== For counts of detected and suppressed errors, rerun with: -v
==6254== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

(B) 유저의 비정상적인 입력처리

유저가 의도치 않은 문자열을 입력한 경우 프로그램 실행에 부정적인 영향을 줄 수 있습니다.

오델로 게임 프로그램에서 유저의 입력을 요구하는 부분은 아래와 같이 두가지가 있습니다.

a. 돌을 두기 위한 좌표 입력

입력 받은 문자열을 파싱하여 split 한 후 std::vector (=vec_cmd) 에 저장하여 문자열 유효성을 검사합니다.

b. 오델로 게임 시작을 위한 보드사이즈 입력 또는 게임 종료명령어 입력

[1번 항목 문자열 유효성 검사]

유저가 유효한 입력을 하지 못했을 경우 다시 UserInputState 상태가 되도록 설정해주어 유저의 입력을 다시 받도록 구현해줍니다
1번 항목의 입력에 대한 유효성을 검사하는 기능과 코드는 아래와 같습니다.

a. 유저가 두개의 좌표를 입력하였는지 확인

```
if (vec_cmd.size() != 2) {
    // error message
    std::cout << "[Invalid input] : wrong coordinate number!!\n";
    context->state(kStateUserInput);
    return;
}
```

b. 입력한 문자열이 숫자로만 이루어져 있는지 확인

```
for (auto &str : vec_cmd) {
    if (!UtilsStringIsNumber(str)) {
        // error message
        std::cout << "[Invalid input] : wrong coordinate format!!\n";
        context->state(kStateUserInput);
        return;
    }
}
```

UtilsStringIsNumber() 함수는 문자열이 숫자로만 이루어져 있는지 검사하는 util 함수입니다.

c. 유저가 입력한 좌표가 보드에 저장된 index 범위내에 있는지 확인

```
if (0 > coord_x || context->board_size() <= coord_x || 0 > coord_y ||
    context->board_size() <= coord_y) {
    // error message
    std::cout << "[Invalid input] : wrong coordinate range!!\n";
    context->state(kStateUserInput);
    return;
}
```

d. 유저가 입력한 좌표가 둘 수 있는 돌의 목록안에 포함이 되는지 확인

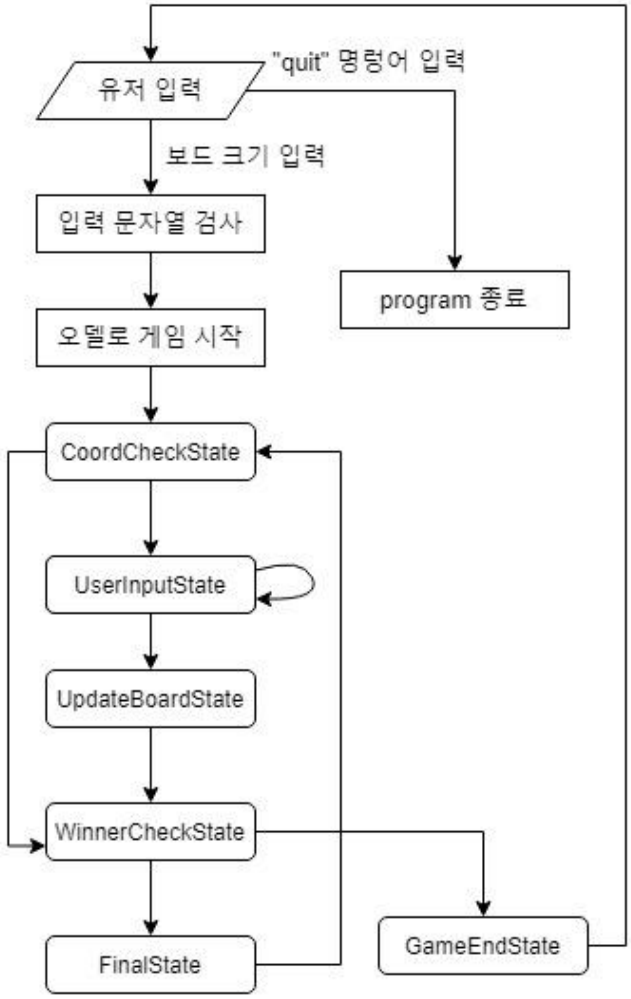
```
if (valid_coord_list.size() == 0) {
    std::cout << "[Change User] : No coordinate to input!!\n";
    // 현재 유저가 현재 턴에 아직 입력을 완료하지 않았다고 표시
    // 이후 WinerCheckState 에서 게임 종료상황인지 확인하기 위해 사용
    (context->cur_user() == kUser0) ? context->user_o_state(kUserInputInvalid)
                                    : context->user_x_state(kUserInputInvalid);
    context->state(kStateWinnercheck);
    return;
}
// 유저가 입력한 좌표가 가능한 목록에 있는지 확인
bool is_valid = false;
int idx = 0;
for (; idx < valid_coord_list.size(); ++idx) {
    if (coord_x == valid_coord_list[idx].x_ &&
        coord_y == valid_coord_list[idx].y_) {
        is_valid = true;
        break;
    }
}
```

[2번 항목 문자열 유효성 검사]

입력 받은 보드의 사이즈가 최소 사이즈(2x2) 보다 큰지 확인합니다.

```
// 입력받은 str(board size)의 유효성 검사
bool BoardSizeValidation(std::string str) {
    if (str == "quit") {
        kCmdIsQuit = true;  // quit 용 flag
        return false;
    } else if (UtilsStringIsNumber(str)) {
        int board_size = std::stoi(str);
        // minimum board size
        if (board_size < kMinBoardSize) {
            std::cout << "[Invalid input] : Wrong baord size (size >= 3)\n";
            return (false);
        }
        return (true);
    }
    return (false);
}
```

5. 실행 시나리오



state 설명
CoordCheckState : 초기 상태
UserInputState : 유저 입력 상태
UpdateBoardState : 뒤집기 상태
WinnerCheckState : 승패 규칙 검사 상태
FinalState : 유저 턴 종료 상태
GameEndState : 게임 종료 상태