МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
“ЛЭТИ” ИМ. В. И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №5
по дисциплине «Объектно-ориентированное программирование»
Тема: управление, разделение на уровни абстракции

Студент гр. 0304 Крицын Д.Р.
Преподаватель Жангиров Т.Р.

Санкт-Петербург
2021

Цель работы.

Организовать необходимые слои абстракции, разделяющие управление игрой и игровую логику.

Задание.

Необходимо организовать управление игрой (номинально через CLI). При управлении игрой с клавиатуры должна считываться нажатая клавиша, после чего происходит перемещение игрока или его взаимодействие с другими элементами поля.

Требования:

- Реализовать управление игрой. Считывание нажатий клавиш не должно происходить в классе игры, а должно происходить в отдельном наборе классов.
- Клавиши управления не должны жестко определяться в коде. Например, это можно определить в отдельном классе.
- Классы управления игрой не должны напрямую взаимодействовать с элементами игры (поле, клетки, элементы на клетках)
- Игру можно запустить и пройти.

Выполнение работы.

1. Создание класса *input_manager*, в задачи которого входит получение ввода с клавиатуры (в конкретном случае через считывание вводимых символов со стандартного потока ввода в консоли). При этом режим работы консоли выставляется не требующим нажатия клавиши *enter* для получения очередного символа или группы симполов (внутренняя функция в *input_manager.cpp* ***static void change_terminal_icanon(bool)***,

используемая в конструкторе и деструкторе *input_manager*). Считывание символов происходит при помощи метода *int get_key_input(),* который возвращает код символа. В данном случае кодом символа являются до 4 символов типа char, упакованных в целое число типа int (это не имеет различия для идентификации уникальных символов).

2. Создание класса *key_bind_manager,* который преобразует считанные с клавиатуры клавиши в осмысленные назначения клавиш (т.е. «бинды», «binds») в методе *static std::string get_bind(int)*. При этом назначения клавиш не встроены в класс, но считываются с файла в статическом методе *void init(std::istream&)*. При этом в конфигурации могут фигурировать имена специальных клавиш по типу *enter, escape, shift* и т.д. — они обрабатываются методом *static int key_name_to_code(const std::string& name).*

3. Создание класса *game_view*, который отвечает за прорисовку игры в целом: при помощи метода *refresh* производится очистка экрана и повторный вывод информации об игре на экран (прорисовки самого игрового поля, буферизованных через *std::stringstream game_log_buf* логов об изменениях состояний объектов, доступного для класса *game* через методы *std::stringstream& getLogBuffer()* и *void clearLogBuffer()*), при помощи метода *clear_screen()* происходит очистка экрана терминала.

4. Создание интерфейса *ui_menu*, который представляет собой игровое меню, которое можно отрисовывать при помощи метода *virtual void draw(game_view& _gvw)* и с которым можно

взаимодействовать через метод *virtual void execute(game_view& _gvw, input_manager& inmngr)*, заверщающийся, когда игрок сделает определённый выбор либо выйдет из меню. При этом способ хранения и получения доступа к выбранным данным зависит от конкретной реализации интерфейса.

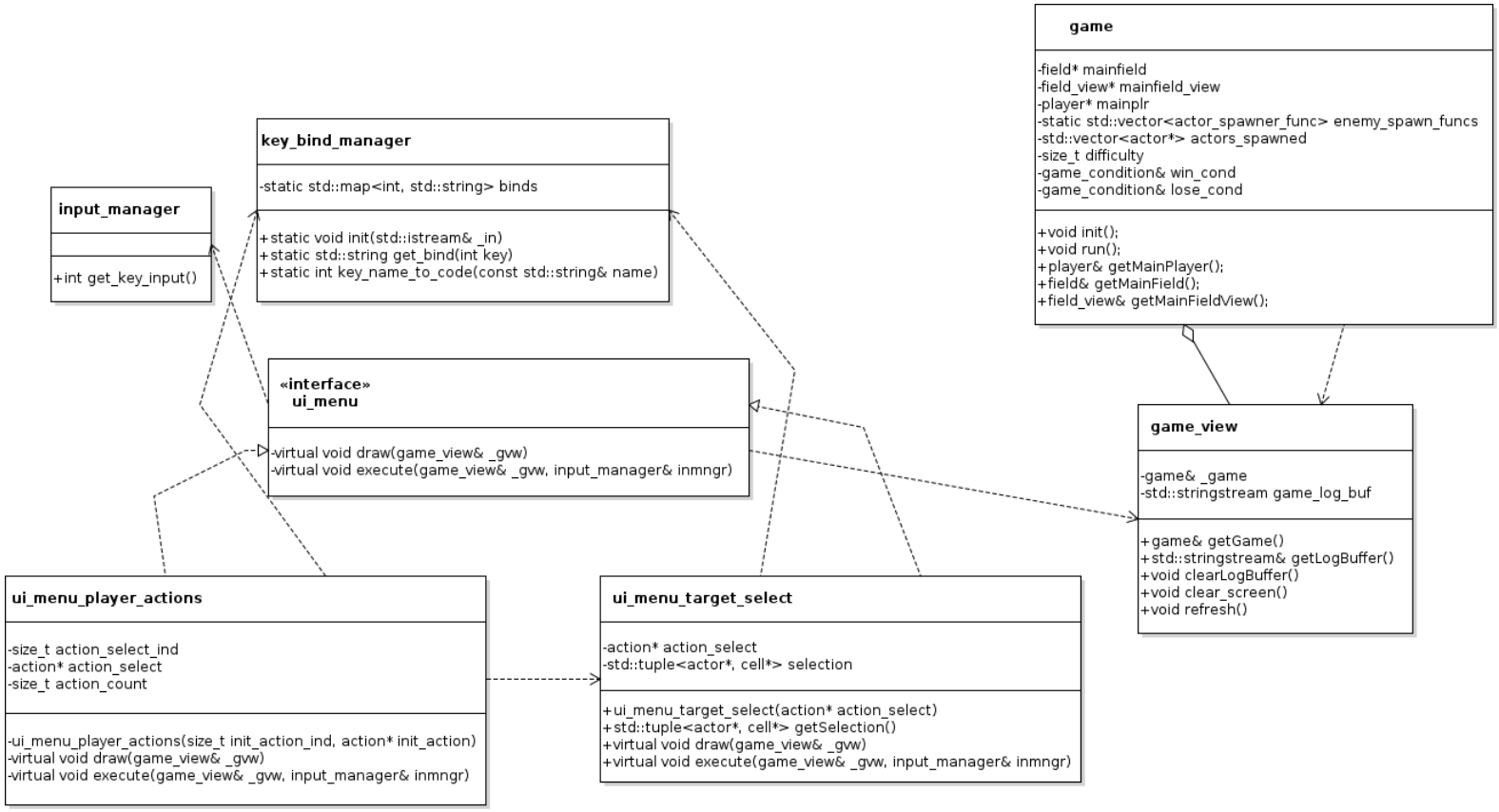
5. Реализация интерфейса *ui_menu* классом *ui_menu_target_select*, который представляет собой меню выбора цели. Для его инициализации небходимо передать в конструктор выбранное действие (*ui_menu_target_select(action* action_select)*), при этом по окончанию вызова метода *virtual void execute(game_view& _gvw, input_manager& inmngr)* можно получить выбранную игроком цель (клетку и/или актёра) при помощи геттера *std::tuple<actor*, cell*> getSelection()*.

6. Реализация интерфейса *ui_menu* классом *ui_menu_player_actions*, который представляет собой меню выбора действия игрока. Данный класс инициализируется начальным номером выбранного действия (обычно 0), и изначально выбранным действием (обычно действием перемещения игрока, которое ему всегда доступно): *ui_menu_player_actions(size_t init_action_ind, action* init_action)*. В методе *execute(game_view& _gvw, input_manager& inmngr)* данное меню ожидает от пользователя выбора действия при помощи навигации по меню нажатиями клавиш, и последующего подтверждения выбора по нажатию другой назначенной клавиши. При этом выбранное действие используется для вызова меню *ui_menu_target_select*, и последующего исполнения действия на выбранной цели, если такое возможно.

Выводы.

В ходе выполнения данной лабораторной работы были изучены различные способы организации прослойки ввода на примере считывания нажатий клавиш с консоли для организации простейших меню выбора и уточнения действия, а также различные приёмы работы с файлами конфигурации.

ПРИЛОЖЕНИЕ А
UML-ДИАГРАММА КЛАССОВ



ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ

game_ui.h

```
#ifndef GAME_UI_H
#define GAME_UI_H

#include "game.h"

class game_ui
{
 private:

 game& _game;

 size_t action_select_ind;
 action* action_select;
 size_t action_count;


 public:

 game_ui(game& _game, size_t init_action_ind, action*
    init_action);

 // output
 void draw_player_actions();

 // input
 void handle_action_menu(input_manager& inmngr);
 std::tuple<actor*, cell*>
    select_action_target(input_manager& inmngr);
};

#endif
```

game_ui.cpp

```
#include "game_ui.h"

#include "input/key_bind_manager.h"

game_ui::game_ui(game& _game, size_t init_action_ind,
    action* init_action)
 : _game(_game), action_select_ind(init_action_ind),
    action_select(init_action) {}

void game_ui::draw_player_actions()
```

```
{
 action_count = 1;
 std::cout << "AP: " << _game.getMainPlayer().getAP() << '\
    n';
 std::cout << "Avaliable actions:\n";

 if(action_select_ind == 0){
    std::cout << "** " <<
    _game.getMainPlayer().getMoveAction().getName() << " **\
    n";
    action_select = &_game.getMainPlayer().getMoveAction();
 }
 else
    std::cout <<
    _game.getMainPlayer().getMoveAction().getName() << '\n';

 for(auto i = _game.getMainPlayer().getInventoryBegin(); i !=
    _game.getMainPlayer().getInventoryEnd(); ++i){
    if(!*i) continue;
    for(auto j = (*i)->getActionsBegin(); j != (*i)-
    >getActionsEnd(); ++j){
        std::cout << (*i)->getName() << " | ";
        if(action_count == action_select_ind){
            std::cout << "** " << (*j)->getName() << " **\
    n";
            action_select = *j;
        }
        else
            std::cout << (*j)->getName() << '\n';
        ++action_count;
    }
 }
}

void game_ui::handle_action_menu(input_manager& inmngr)
{
 std::string bind;

 while(1){
 bind = key_bind_manager::get_bind(inmngr.get_key_input());

 if(bind == ""){
    _game.refresh(*this);
    std::cout << "This key is not bound to any action.\n";
    continue;
```

```
}
else if(bind == "menu_down"){
   if(action_select_ind + 1 < action_count)
       ++action_select_ind;
   _game.refresh(*this);
   continue;
}
else if(bind == "menu_up"){
   if(action_select_ind > 0)
       --action_select_ind;
   _game.refresh(*this);
   continue;
}
else if(bind == "end_turn"){
   _game.refresh(*this);
   break;
}
else if(bind == "confirm"){
   std::tuple<actor*, cell*> action_info =
   select_action_target(inmngr);
   if(!action_select-
   >canExecute(_game.getMainPlayer().getAP(),
   &_game.getMainPlayer(),
                        std::get<0>(action_info),
   std::get<1>(action_info))){
       std::cout << "Cannot execute this action.\n";
       continue;
   }
   action_select->tryExecute(_game.getMainPlayer().getAP(),
   &_game.getMainPlayer(),
                        std::get<0>(action_info),
   std::get<1>(action_info));
   _game.refresh(*this);
   continue;
}
else{
   _game.refresh(*this);
   continue;
}
break;}
}

std::tuple<actor*, cell*>
   game_ui::select_action_target(input_manager& inmngr)
{
```

```
size_t x = _game.getMainPlayer().getParentCell().get_x(), y
   = _game.getMainPlayer().getParentCell().get_y();

while(1){
_game.getMainFieldView().setHighlight(true, x, y);
_game.refresh(*this);

cell* c = &_game.getMainField().get_cell(x, y);
actor* passive = c->has_object() ? &dynamic_cast<actor&>(c-
   >get_object()) : nullptr;
if(action_select->canExecute(_game.getMainPlayer().getAP(),
   &_game.getMainPlayer(), passive, c))
   std::cout << "Cost: " << action_select-
   >getAPCost(&_game.getMainPlayer(), passive, c) << " AP\
   n";
else
   std::cout << "Cannot execute\n";

std::string bind =
   key_bind_manager::get_bind(inmngr.get_key_input());

if(bind == ""){
   std::cout << "This key is not bound to any action.\n";
   continue;
}
else if(bind == "menu_right"){
   if(y >= _game.getMainField().get_h()){
       std::cout << "Cannot advance outside of the field.\
   n";
   }
   else{
       y++;
   }
   continue;
} else if(bind == "menu_left"){
   if(y == 0){
       std::cout << "Cannot advance outside of the field.\
   n";
   }
   else{
       y--;
   }
   continue;
} else if(bind == "menu_up"){
   if(x == 0){
```

```
        std::cout << "Cannot advance outside of the field.\
    n";
    }
    else{
        x--;
    }
    continue;
 } else if(bind == "menu_down"){
    if(x >= _game.getMainField().get_w()){
        std::cout << "Cannot advance outside of the field.\
    n";
    }
    else{
        x++;
    }
    continue;
 }

 break;}
 _game.getMainFieldView().setHighlight(false);

 cell* c = &_game.getMainField().get_cell(x, y);
 return {c->has_object() ? &dynamic_cast<actor&>(c-
    >get_object()) : nullptr,
    c};
}
```

input_manager.h

```
#ifndef INPUT_MANAGER_H
#define IMPUT_MANAGER_H

class input_manager
{
 public:

 input_manager();
 ~input_manager();

 int get_key_input();
};

#endif
```

input_manager.cpp

```
#include "input_manager.h"

#include <cstdio>
```

```
#include <termios.h>
#include <unistd.h>

static void change_terminal_icanon(bool set)
{
 static struct termios told, tnew;
 tcgetattr(STDIN_FILENO, &told);
 tnew = told;
 if(set)
    tnew.c_lflag |= ICANON;
 else
    tnew.c_lflag &= ~ICANON;
 tcsetattr(STDIN_FILENO, TCSANOW, &tnew);
}

input_manager::input_manager()
{
 change_terminal_icanon(false);
}
input_manager::~input_manager()
{
 change_terminal_icanon(true);
}

static inline char add_char_to_int(int& i, char c)
{
 i *= 256; i += c;
 return c;
}
int input_manager::get_key_input()
{
 int key = 0;
 if( (key = getchar()) == '\033'){
    if(add_char_to_int(key, getchar()) == '[')
        add_char_to_int(key, getchar());
 }
 return key;
}
```

key_bind_manager.h

```
#ifndef KEY_BIND_MANAGER_H
#define KEY_BIND_MANAGER_H

#include <istream>
#include <map>
```

```
class key_bind_manager
{
 private:

 static std::map<int, std::string> binds;


 public:

 static void init(std::istream& _in);

 // возвращает пустую строку в случае отсутствия назначения
    клавиши
 static std::string get_bind(int key);

 // преобразует имя клавиши в код клавиши.
 static int key_name_to_code(const std::string& name);
};

#endif
```

key_bind_manager.cpp

```
#include "key_bind_manager.h"

#include <algorithm>
#include <cstring>
#include <vector>
#include <string>

#include <iostream>

std::map<int, std::string> key_bind_manager::binds;

void key_bind_manager::init(std::istream& _in)
{
 int ch;
 std::vector<char> bind; size_t bind_i = 0;
 std::vector<char> key; size_t key_i = 0;
 const size_t buf_growth = 8;

 while(1)
 {
    while( (ch = _in.get()) != '=' ){
        if(ch == std::char_traits<char>::eof()) return;
        if(std::isspace(ch)) continue;
        if(bind_i >= bind.capacity())
```

```
            bind.resize(bind.size() + buf_growth);
        bind[bind_i++] = ch;
    }
    while( (ch = _in.get()) != ',' ){
        if(ch == std::char_traits<char>::eof()) break;
        if(std::isspace(ch)) continue;
        if(key_i >= key.capacity())
            key.resize(key.size() + buf_growth);
        key[key_i++] = ch;
    }

    bind[bind_i] = '\0'; key[key_i] = '\0';
    std::string _bind;
    for(auto i = bind.begin(); *i; ++i) _bind += *i;
    std::string _key;
    for(auto i = key.begin(); *i; ++i) _key += *i;

    binds.insert({key_name_to_code(_key), _bind});
    bind_i = 0; key_i = 0;
 }
}

std::string key_bind_manager::get_bind(int key)
{
 auto b = binds.find(key);
 if(b == binds.end())
    return "";
 return b->second;
}


static inline int multichar_to_int(const char* chars)
{
 int res = 0;
 for(; *chars; ++chars)
 { res *= 256; res += *chars; }
 return res;
}
int key_bind_manager::key_name_to_code(const std::string&
    name)
{
 static const std::map<std::string, int> transl_map {
 {"enter", '\n'}, {"backspace", '\b'},
 {"leftarrow", multichar_to_int("\033[D")},
 {"rightarrow", multichar_to_int("\033[C")},
```

```
{"uparrow", multichar_to_int("\033[A")},
{"downarrow", multichar_to_int("\033[B")}
};

auto kcode = transl_map.find(name);
if(kcode == transl_map.end())
   return name[0];

return kcode->second;
}
```