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

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Объектно-ориентированное программирование»
Тема: Интерфейсы, полиморфизм

Студент гр. 0304 Крицын Д.Р.
Преподаватель Шевская Н.В.

Санкт-Петербург
2021

Цель работы.

Реализовать основные классы действующих объектов в игре — интерфейсы для игрока, врага, и вещи. Реализовать три класса врага и три класса вещи с уникальными свойствами. В данном процессе познакомиться с принципами полиморфизма и наследования классов, созданием и использованием (реализацией) интерфейсов.

Задание.

Могут быть три типа элементов располагающихся на клетках:

1. Игрок — объект, которым непосредственно происходит управление. На поле может быть только один игрок. Игрок может взаимодействовать с врагом (сражение) и вещами (подобрать).
2. Враг — объект, который самостоятельно перемещается по полю. На поле врагов может быть больше одного. Враг может взаимодействовать с игроком (сражение).
3. Вещь — объект, который просто располагается на поле и не перемещается. Вещей на поле может быть больше одной.

Требования:

- Реализовать класс игрока. Игрок должен обладать собственными характеристиками, которые могут изменяться в ходе игры. У игрока должна быть прописана логика сражения и подбора вещей. Должно быть реализовано взаимодействие с клеткой выхода.
- Реализовать три разных типа врагов. Враги должны обладать собственными характеристиками (например, количество жизней, значение атаки и защиты, и.т.д. Желательно, чтобы у врагов были разные наборы характеристик). Реализовать логику перемещения для каждого типа врага. В случае смерти врага он

должен исчезнуть с поля. Все враги должны быть объединены своим собственным интерфейсом.

- Реализовать три разных типа вещей. Каждая вещь должна обладать собственным взаимодействием на ход игры при подборе. *(например, лечение игрока).* При подборе, вещь должна исчезнуть с поля. Все вещи должны быть объединены своим собственным интерфейсом.
- Должен соблюдаться принцип полиморфизма.

Выполнение работы.

1. Дополнение интерфейса *cell_object* из предыдущей лабораторной работы: добавление поля — родительской клетки, и геттеров\сеттеров для него.

2. Создание интерфейса «действующего лица» — *actor*, наследника *cell_object.* Данный интерфейс имеет поля — текущее количество здоровья, максимальное количество здоровья, защитные характеристики, номер команды. Для каждого из этих параметров прописаны гетерры\сеттеры. Также актёр может реагировать на такие события, как начало своего хода (где актёр выполняет определённые действия), получение или излечение урона (начальные методы для этих событий обрабатывают получение урона в зависимости от защитных характеристик, и ограничивают получение урона до 0 единиц здоровья, а лечение — до максимального здоровья), смерть, на попытку войти в клетку, где находится данный актёр, другим актёром, при перемещении на клетку.

3. Создание интерфейса действия *action.* Данный интерфейс требует реализации методов — *getAPCost()* для получения информации о стоимости действия в «очках действий», *canExecute()* для получения информации о том, возможно ли выполнить действие, и *execute()* для реализации логики выполнения самого действия.
4. Реализация интерфейса действия в виде каких-то базовых действий. *action_combat* представляет собой абстрактный класс, для которого не реализован метод получения информации о стоимости действия, но который уже реализует метод *execute()* для нанесения урона цели. *action_shoot_ballistic* представляет собой реализацию *action_combat* и позволяет задать конкретный урон, дальность атаки и фиксированную стоимость в очках действий. *action_move* представляет собой примитивное действие перемещения, выполняемое до заданной клетки последовательно, по одной клетке за раз, параллельно осям координат. При этом действие завершается, если на пути актёра возникнет препятствие.
5. Создание интерфейса врага *enemy*. В данном классе присутствует поле — текущее и максимальное количество очков действий, а также дальность обнаружения, по одному действию для нападения на игрока и перемещению ближе к игроку. При это для демонстрации реализована базовая логика сражения — враг сначала тратит очки действия на возможные атаки на ближайшего противника, затем оставшиеся очки действия

уходят на перемещение ближе к ближайшему противнику, если такой существует.

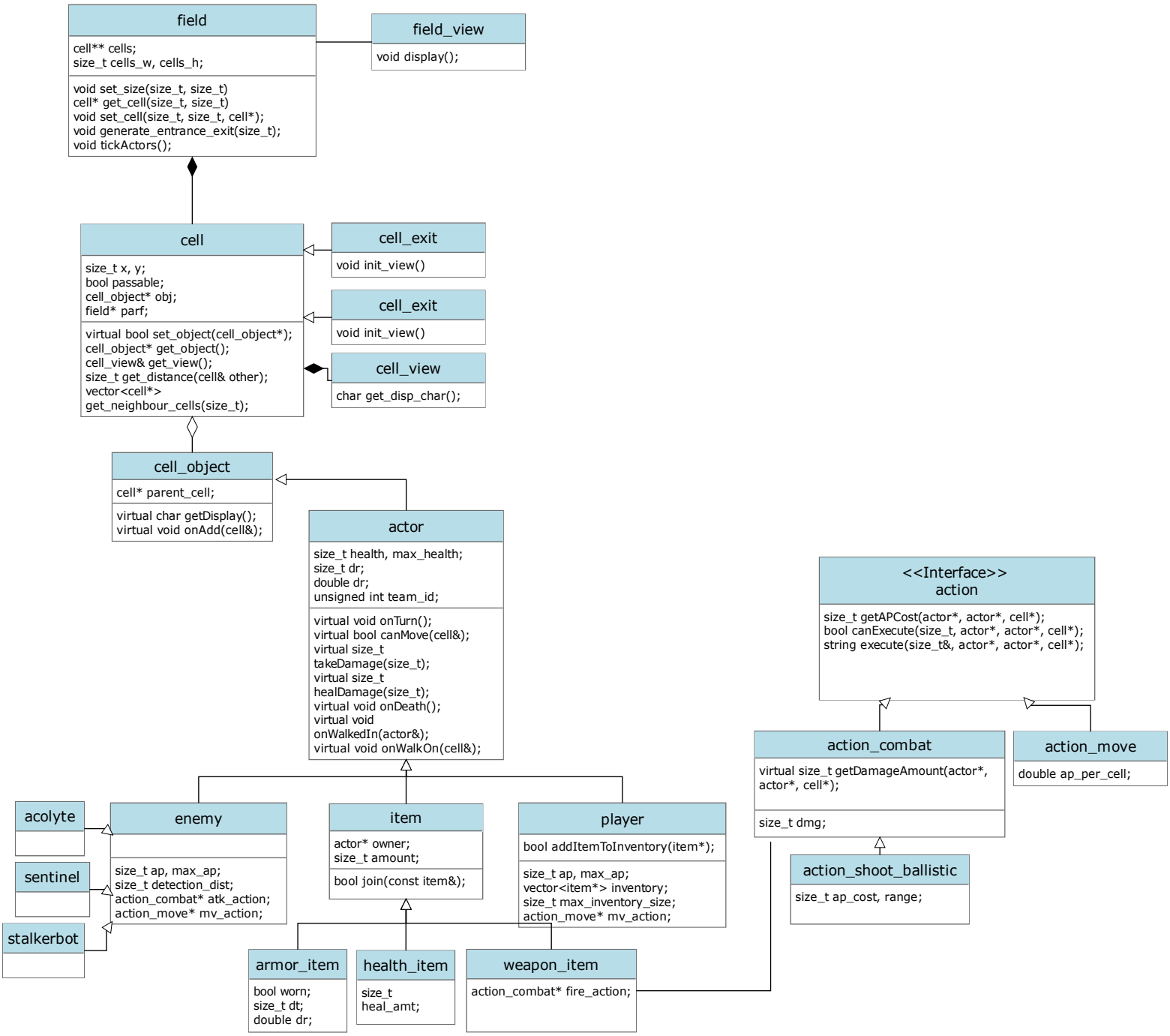
6. Создание базовых классов врагов — *acolyte, sentinel, stalkerbot*. Данные классы отличаются защитными характеристиками, дальностью обнаружения, а также дальностью, уроном и трудоёмкостью атак.
7. Создание интерфейса предмета — *item*. Данный класс реализует событие *onWalkedIn()* интерфейса *actor*, добавляясь в инвентарь игрока, если наступившим на клетку актёром был именно игрок. Также добавляется поле владельца предмета и его количества, и новые события *onUsed()* — событие при использовании предмета игроком, *join()* — попытка слияния двух предметов в один. По умолчанию, предметы одинаковых классов объединяются в один, складывая оба количества.
8. Реализация интерфейса предмета тремя классами — *armor_item, health_item, weapon_item*. Класс брони *armor_item* при использовании инвертирует своё состояние ношения — при ношении он увеличивает показатели устойчивости к урону и порогу урона, а при снятии уменьшает на ту же величину. Класс аптечки *healh_item* исцеляет игрока на фиксированную величину, уменьшая своё количество при каждом использовании. Класс оружия *weapon_item* хранит в себе действие, происходящее при стрельбе из оружия.

9. Создание класса игрока *player*. В нём ещё не реализовано управление игроком, но реализован инвентарь и управление очками действий (а также действие перемещения).

Выводы.

В ходе выполнения данной лабораторной работы были изучены такие принципы парадигмы ООП, как наследование, интерфейсы и их реализация, полиморфизм, на примере создания классов, реализующих логику компьютерной игры.

ПРИЛОЖЕНИЕ А
UML-ДИАГРАММА КЛАССОВ



ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ

field.h

```
#ifndef FIELD_H
#define FIELD_H

#include <cstdlib>

#include "cells/cell.h"

class field
{
    public:

    field(const field& other);
    field(field&& other);
    field& operator=(const field& other);
    field& operator=(field&& other);

    field(size_t w, size_t h);

    // Basic manipulation methods

    size_t get_w() const;
    size_t get_h() const;
    void set_size(size_t w, size_t h);

    cell* get_cell(size_t x, size_t y) const;
    void set_cell(size_t x, size_t y, cell* c);

    // Display

    void print();

    // Generation

    void generate_entrance_exit(size_t min_dist = 3);

    private:

    cell** cells;
    size_t cells_w, cells_h;
};

#endif
```

actor.h

```
#ifndef ACTOR_H
#define ACTOR_H

#include "../cell_object.h"
#include "../cells/cell.h"

class actor : public cell_object
{
    public:

    // Properties
    virtual size_t getHealth();
    virtual void setHealth(size_t hp);
    virtual size_t getMaxHealth();
    virtual void setMaxHealth(size_t max_hp);
    virtual size_t getDamageThreshold();
    virtual void setDamageThreshold(size_t dt);
    virtual double getDamageResistance();
    virtual void setDamageResistance(double dr);

    virtual unsigned int getTeamID();
    virtual void setTeamID(unsigned int tid);

    // Events
    virtual void onTurn() = 0;

    virtual bool canMove(cell& to_where);

    virtual size_t takeDamage(size_t dmg);
    virtual size_t healDamage(size_t heal);
    virtual void onDeath() = 0;

    virtual void onWalkedIn(actor& by_who) = 0;
    virtual void onWalkOn(cell& to_where) = 0;


    private:

    size_t health, max_health;
    size_t dt;
    double dr;

    unsigned int team_id;
};
```

```
#endif
```

actor.cpp

```
#include "actor.h"

size_t actor::getHealth() { return health; }
void actor::setHealth(size_t hp) { health = hp; }
size_t actor::getMaxHealth() { return max_health; }
void actor::setMaxHealth(size_t max_hp) { max_health =
max_hp; }
size_t actor::getDamageThreshold() { return dt; }
void actor::setDamageThreshold(size_t dt) { this->dt = dt; }
double actor::getDamageResistance() { return dr; }
void actor::setDamageResistance(double dr) { this->dr =
dr; }

unsigned int actor::getTeamID() { return team_id; }
void actor::setTeamID(unsigned int tid) { this->team_id =
tid; }

bool actor::canMove(cell& to_where)
{
    if(!to_where.is_passable())
        return false;
    if(to_where.has_object())
        return false;

    return true;
}

size_t actor::takeDamage(size_t dmg)
{
    size_t new_dmg = (dmg >= getDamageThreshold() ? dmg -
getDamageThreshold() : 0)
             * (1.0 - (getDamageResistance() >= 1.0 ? 1.0 :
getDamageResistance()));
    setHealth(new_dmg < getHealth() ? getHealth() -
new_dmg : 0);
    return new_dmg;
}
size_t actor::healDamage(size_t heal)
{
    setHealth(getMaxHealth() - getHealth() <= heal ?
getHealth() + heal : getMaxHealth());
```

```
    return getMaxHealth() - getHealth() <= heal ? heal :
getMaxHealth() - heal;
}
```

enemy.h

```
#ifndef ENEMY_H
#define ENEMY_H

#include "actor.h"
#include "../actions/action_combat.h"
#include "../actions/action_move.h"

#include <vector>

class enemy : public actor
{
    public:

    // Properties
    virtual char getDisplay(cell& holder);
    size_t& getAP();

    // Events
    virtual void onAdd(cell& holder);

    virtual void onTurn();
    virtual void onDeath();

    virtual void onWalkedIn(actor& by_who);
    virtual void onWalkOn(cell& to_where);

    protected:

    size_t ap, max_ap;

    size_t detection_dist;

    action_combat* atk_action;
    action_move* mv_action;
};

#endif
```

enemy.cpp

```
#include "enemy.h"

#include <iostream>
```

```
// Properties

char enemy::getDisplay(cell& holder) { return '%'; }

size_t& enemy::getAP() { return ap; }

// Events

void enemy::onAdd(cell& holder){}

void enemy::onTurn()
{
    ap = max_ap;

    std::vector<cell*> nc =
getParentCell().get_neighbour_cells(detection_dist);
    actor* last_attack_target = nullptr;

    for(size_t i = 0; i < nc.size(); ++i)
    {
        cell* c = nc[i];
        if(!c->has_object())
            continue;
        actor* a = dynamic_cast<actor*>(&c->get_object());
        if(a == nullptr)
            continue;
        if(a->getTeamID() == getTeamID())
            continue;

        while(a != nullptr && a->getHealth() > 0
            && atk_action->canExecute(ap, this, a, c)){
            std::cout << atk_action->execute(ap, this, a, c)
<< std::endl;
            last_attack_target = a;
        }
    }

    if(last_attack_target == nullptr)
    {
        cell* target_cell = nullptr;
        actor* target = nullptr;
        size_t target_dist = (size_t)-1;

        for(size_t i = 0; i < nc.size(); ++i)
```

```
        {
            cell* c = nc[i];
            if(!c->has_object())
                continue;
            actor* a = dynamic_cast<actor*>(&c-
>get_object());
            if(a == nullptr)
                continue;
            if(a->getTeamID() == getTeamID())
                continue;

            if(a->getHealth() > 0 && c->get_distance(this-
>getParentCell()) < target_dist){
                target_cell = c;
                target = a;
                target_dist = c->get_distance(this-
>getParentCell());
            }
        }

        if(target && mv_action->canExecute(ap, this, target,
target_cell))
            mv_action->execute(ap, this, target,
target_cell);
    }
}

void enemy::onDeath()
{
    getParentCell().set_object(nullptr);
    delete this;
}


void enemy::onWalkedIn(actor& by_who){}
void enemy::onWalkOn(cell& to_where){}
```

item.h

```
#ifndef ITEM_H
#define ITEM_H

#include "actor.h"

class item : public actor
{
    public:
```

```
    // events
    void onWalkedIn(actor& by_who);
    virtual void onUsed(actor& p) = 0;

    // properties
    char getDisplay(cell& holder);

    size_t getAmount() const;
    void setAmount(size_t amount);

    actor& getOwner() const;
    void setOwner(actor* o);

    bool join(const item& other);


    protected:

    actor* owner;
    size_t amount;
};

#endif
```

item.cpp

```
#include "item.h"

#include "player.h"

// events

void item::onWalkedIn(actor& by_who)
{
    player* p = dynamic_cast<player*>(&by_who);
    if(p){
        if(p->addItemToInventory(this)){
            if(hasParentCell())
getParentCell().set_object(nullptr);
        }
    }
}

// properties

char item::getDisplay(cell& holder) { return '+'; }
```

```
size_t item::getAmount() const { return amount; }
void item::setAmount(size_t amount)
{
    this->amount = amount;
    if(this->amount == 0) {
        if(hasParentCell())
getParentCell().set_object(nullptr);
    }
}

actor& item::getOwner() const { return *owner; }
void item::setOwner(actor* o) { owner = o; }

bool item::join(const item& other)
{
    if(typeid(other).hash_code() !=
typeid(*this).hash_code())
        return false;

    amount += other.getAmount();
    return true;
}
```

player.h

```
#ifndef PLAYER_H
#define PLAYER_H

#include <vector>

#include "item.h"
#include "../actions/action_move.h"

class player : public actor
{
    public:

    // Properites
    virtual char getDisplay(cell& holder);

    size_t& getAP();

    // Inventory
    bool addItemToInventory(item* i);

    // Events
```

```
    virtual void onAdd(cell& holder);

    virtual void onTurn();
    virtual void onDeath();

    virtual void onWalkedIn(actor& by_who);
    virtual void onWalkOn(cell& to_where);


    protected:

    size_t ap, max_ap;

    std::vector<item*> inventory;
    size_t max_inventory_size = 10;

    action_move* mv_action;
};

#endif
```

player.cpp

```
#include "player.h"

#include "../cells/cell_exit.h"

// Properties

char player::getDisplay(cell& holder) { return '@'; }

size_t& player::getAP() { return ap; }


// Inventory

bool player::addItemToInventory(item* i)
{
    for(size_t j = 0; j < inventory.size(); ++j)
        if(inventory[j]->join(*i))
            return true;

    if(inventory.size() >= max_inventory_size)
        return false;

    inventory.push_back(i);
    return true;
```

```
}


// Events

void player::onAdd(cell& holder) {}

void player::onTurn()
{
    ap = max_ap;
}
void player::onDeath() {}

void player::onWalkedIn(actor& by_who) {}
void player::onWalkOn(cell& to_where)
{
    cell_exit* ce = dynamic_cast<cell_exit*>(&to_where);

    if(ce){

    }
}
```

acolyte.cpp

```
#include "acolyte.h"

acolyte::acolyte()
{
    max_ap = 8;
    detection_dist = 15;

    setMaxHealth(50);
    setDamageThreshold(5);
    setDamageResistance(0.2);

    atk_action = new action_shoot_ballistic(4, 3, 1/*20*/);
    mv_action = new action_move(5.0);
}
```

sentinel.cpp

```
#include "sentinel.h"

sentinel::sentinel()
{
    max_ap = 9;
    detection_dist = 25;
```

```
    setMaxHealth(30);
    setDamageThreshold(8);
    setDamageResistance(0.3);

    atk_action = new action_shoot_ballistic(3, 3, 18);
    mv_action = new action_move(0.7);
}
```

stalkerbot.cpp

```
#include "stalkerbot.h"

stalkerbot::stalkerbot()
{
    max_ap = 6;
    detection_dist = 20;

    setMaxHealth(80);
    setDamageThreshold(6);
    setDamageResistance(0.2);

    atk_action = new action_shoot_ballistic(6, 5, 3);
    mv_action = new action_move(0.8);
}
```

armor_item.h

```
#ifndef ARMOR_ITEM_H
#define ARMOR_ITEM_H

#include "../item.h"

class armor_item : public item
{
    public:

    armor_item(size_t dt, double dr);

    virtual void onUsed(actor& on_who);


    protected:

    bool worn;

    size_t dt; double dr;
};

#endif
```

armor_item.cpp

```
#include "armor_item.h"

armor_item::armor_item(size_t dt, double dr) : dt(dt),
dr(dr) {}

void armor_item::onUsed(actor& on_who)
{
    if(worn){

    on_who.setDamageThreshold(on_who.getDamageThreshold() +
dt);

    on_who.setDamageResistance(on_who.getDamageResistance()
+ dr);
    }
    else{

    on_who.setDamageThreshold(on_who.getDamageThreshold() -
dt);

    on_who.setDamageResistance(on_who.getDamageResistance()
- dr);
    }

    worn = !worn;
}
```

health_item.h

```
#ifndef HEALTH_ITEM_H
#define HEALTH_ITEM_H

#include "../item.h"

class health_item : public item
{
    public:

    health_item(size_t heal_amt);

    virtual void onUsed(actor& on_who);


    protected:

    size_t heal_amt;
```

```
};

#endif
```

health_item.cpp

```
#include "health_item.h"

health_item::health_item(size_t heal_amt) :
heal_amt(heal_amt) {}

void health_item::onUsed(actor& on_who)
{
    size_t healed = on_who.healDamage(heal_amt);
    if(healed){
        setAmount(getAmount() - 1);
    }
}
```

weapon_item.h

```
#ifndef WEAPON_ITEM_H
#define WEAPON_ITEM_H

#include "../item.h"
#include "../../actions/action_combat.h"

class weapon_item : public item
{
    public:

    virtual bool fire(actor* target);


    protected:

    action_combat* fire_action;
};

#endif
```

weapon_item.cpp

```
#include "weapon_item.h"

#include "../enemy.h"
#include "../player.h"

bool weapon_item::fire(actor* target)
{
    if(!owner)
```

```
        return false;

    size_t* apref;
    if(dynamic_cast<player*>(owner))
        apref = &dynamic_cast<player*>(owner)->getAP();
    else if(dynamic_cast<enemy*>(owner))
        apref = &dynamic_cast<enemy*>(owner)->getAP();
    if(!fire_action->canExecute(*apref, owner, target,
&target->getParentCell()))
        return false;

    fire_action->execute(*apref, owner, target, &target-
>getParentCell());
    return true;
}
```

action.h

```
#ifndef ACTION_H
#define ACTION_H

#include "../actors/actor.h"
#include <string>

class action
{
    public:

    virtual size_t getAPCost(actor* active, actor* passive,
cell* c) = 0;
    virtual bool canExecute(size_t ap, actor* active, actor*
passive, cell* c) = 0;

    virtual std::string execute(size_t& ap, actor* active,
actor* passive, cell* c) = 0;
};

#endif
```

action_combat.h

```
#ifndef ACTION_COMBAT_H
#define ACTION_COMBAT_H

#include "action.h"

#include <cstddef>

class action_combat : public action
```

```
{
    public:

    action_combat(size_t dmg);

    virtual size_t getDamageAmount(actor* active, actor*
passive, cell* c);

    virtual bool canExecute(size_t ap_amt, actor* active,
actor* passive, cell* c);
    std::string execute(size_t& ap_amt, actor* active,
actor* passive, cell* c);

    protected:

    size_t dmg;
};

#endif
```

action_combat.cpp

```
#include "action_combat.h"

action_combat::action_combat(size_t dmg) : dmg(dmg) {}

size_t action_combat::getDamageAmount(actor* active, actor*
passive, cell* c) { return dmg; }
bool action_combat::canExecute(size_t ap, actor* active,
actor* passive, cell* c)
{
    if(ap < getAPCost(active, passive, c))
        return false;
    if(passive == nullptr)
        return false;
    return true;
}

std::string action_combat::execute(size_t& ap, actor*
active, actor* passive, cell* c)
{
    size_t dmgdealt = passive->takeDamage(dmg);
    ap -= getAPCost(active, passive, c);
    return "hit for " + std::to_string(dmgdealt) + "
damage";
}
```

action_move.h

```
#ifndef ACTION_MOVE_H
#define ACTION_MOVE_H

#include "action.h"

#include <cstddef>

class action_move : public action
{
    public:

    action_move(double ap_per_cell);

    size_t getAPCost(actor* active, actor* passive, cell*
c);
    bool canExecute(size_t ap, actor* active, actor*
passive, cell* c);

    std::string execute(size_t& ap, actor* active, actor*
passive, cell* c);

    protected:

    double ap_per_cell;
};

#endif
```

action_move.cpp

```
#include "action_move.h"

#include "../field.h"

action_move::action_move(double ap_per_cell) :
ap_per_cell(ap_per_cell) {}

size_t action_move::getAPCost(actor* active, actor* passive,
cell* c)
{
    return 1;
}
bool action_move::canExecute(size_t ap, actor* active,
actor* passive, cell* c)
{
    if(ap < getAPCost(active, passive, c))
        return false;
```

```
    if(active == nullptr || c == nullptr)
        return false;
    return true;
}

#include <iostream>
std::string action_move::execute(size_t& ap, actor* active,
actor* passive, cell* c)
{
    double total_ap_cost = 0.0;
    cell* curc = &active->getParentCell();
    while(curc->get_x() != c->get_x() || curc->get_y() != c-
>get_y())
    {
        if(curc->get_x() < c->get_x()
        && curc->get_x() < curc->get_parent_field().get_w()-
1
        && active->canMove(curc-
>get_parent_field().get_cell(curc->get_x() + 1, curc-
>get_y())))
            curc = &curc->get_parent_field().get_cell(curc-
>get_x() + 1, curc->get_y());
        else if(curc->get_x() > c->get_x()
        && curc->get_x() > 0
        && active->canMove(curc-
>get_parent_field().get_cell(curc->get_x() - 1, curc-
>get_y())))
            curc = &curc->get_parent_field().get_cell(curc-
>get_x() - 1, curc->get_y());
        else if(curc->get_y() < c->get_y()
        && curc->get_y() < curc->get_parent_field().get_h()-
1
        && active->canMove(curc-
>get_parent_field().get_cell(curc->get_x(), curc->get_y() +
1)))
            curc = &curc->get_parent_field().get_cell(curc-
>get_x(), curc->get_y() + 1);
        else if(curc->get_y() > c->get_y()
        && curc->get_y() > 0
        && active->canMove(curc-
>get_parent_field().get_cell(curc->get_x(), curc->get_y() -
1)))
            curc = &curc->get_parent_field().get_cell(curc-
>get_x(), curc->get_y() - 1);
        else
```

```
            break;

        total_ap_cost += ap_per_cell;
        if((size_t)(total_ap_cost + ap_per_cell) > ap)
            break;
    }
    ap -= ((size_t)total_ap_cost);


    active->getParentCell().set_object(nullptr);
    active->setParentCell(curc);
    curc->set_object(active);
    return "ran";
}
```

action_shoot_ballistic.h

```
#ifndef ACTION_SHOOT_BALLISTIC_H
#define ACTION_SHOOT_BALLISTIC_H

#include "action_combat.h"

class action_shoot_ballistic : public action_combat
{
    public:

    action_shoot_ballistic(size_t dmg, size_t ap_cost,
size_t range);

    virtual size_t getAPCost(actor* active, actor* passive,
cell* c);
    virtual bool canExecute(size_t ap_amt, actor* active,
actor* passive, cell* c);


    protected:

    size_t ap_cost;
    size_t range;
};

#endif
```

action_shoot_ballistic.cpp

```
#include "action_shoot_ballistic.h"

action_shoot_ballistic::action_shoot_ballistic(size_t dmg,
size_t ap_cost, size_t range) : action_combat(dmg)
```

```
{
    this->ap_cost = ap_cost;
    this->range = range;
}

bool action_shoot_ballistic::canExecute(size_t ap_amt,
actor* active, actor* passive, cell* c)
{
    if(!action_combat::canExecute(ap_amt, active, passive,
c))
        return false;

    if(active->getParentCell().get_distance(*c) > range)
        return false;

    return true;
}
size_t action_shoot_ballistic::getAPCost(actor* active,
actor* passive, cell* c) { return ap_cost; }
```