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

ОТЧЁТ

по лабораторной работе №1

по дисциплине «Объектно-ориентированное программирование»

Тема: Создание классов, конструкторов классов, методов классов; наследование

Студент гр. 8303 _____ Парфентьев Л.М.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы

Научиться создавать классы, добавлять в них конструкторы и методы, создавать классы, наследующиеся от других классов.

Задание

Разработать и реализовать набор классов:

- Класс игрового поля
- Набор классов юнитов

Игровое поле является контейнером для объектов представляющим прямоугольную сетку. Основные требования к классу игрового поля:

- Создание поля произвольного размера
- Контроль максимального количества объектов на поле
- Возможность добавления и удаления объектов на поле
- Возможность копирования поля (включая объекты на нем)
- Для хранения запрещается использовать контейнеры из stl

Юнит является объектов, размещаемым на поля боя. Один юнит представляет собой отряд. Основные требования к классам юнитов:

- Все юниты должны иметь как минимум один общий интерфейс
- Реализованы 3 типа юнитов (например, пехота, лучники, конница)
- Реализованы 2 вида юнитов для каждого типа(например, для пехоты могут быть созданы мечники и копейщики)
- Юниты имеют характеристики, отражающие их основные атрибуты, такие как здоровье, броня, атака.
- Юнит имеет возможность перемещаться по карте

Ход выполнения работы

- Класс, управляющий памятью прямоугольного поля – `RectMap`. Ячейки поля представлены классом `Cell`. Класс `Cell` используется только классами `RectMap`, `Map` и их итераторами.

- Класс, представляющий поле – `Map`. Поле содержит в себе объект класса `RectMap`, а само занимается управлением юнитами – их добавлением и снятием с поля, а также следит за ограничением на количество юнитов.

- Для классов `RectMap` и `Map` определены классы итераторов – соответственно `RectMapIter` и `MapIter`. Для простоты они *не* удовлетворяют интерфейсам итераторов из стандартной библиотеки C++.

- `MapIter` реализован через `RectMapIter`. Этот класс предоставляет доступ к содержимому ячеек поля. Итератор можно „перемещать“, чтобы он указывал на другую ячейку.

- Класс `Placeable` представляет объект, расположенный на поле. Он содержит свою позицию. Класс `Cell` хранит указатель на `Placeable`.

- Класс юнитов – `Unit`. Он наследуется от `Placeable`. Юнит содержит 3 „политики“ (стратегии) – стратегию передвижения (`MovePolicy`), стратегию атаки (`AttackPolicy`) и стратегию защиты (`DefensePolicy`). Поведение юнита полностью определяется этими стратегиями.

- Для создания различных подклассов юнитов были определены специальные подклассы стратегий (в основном стратегий атаки). Стратегии нужных классов создаются в конструкторах классов юнитов.

- Созданы следующие классы юнитов: `BasicMeleeUnit`, `BasicRangedUnit` и `BasicCatapultUnit`. Для них создано множество подклассов (в пространстве имён `units`).

Выводы

В ходе выполнения лабораторной работы было изучено создание классов, добавление в них методов, а также наследование.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: point.hpp

```
#ifndef _H_POINT_HPP
#define _H_POINT_HPP

class Vec2 {
    int _x, _y;

public:
    Vec2() :Vec2{0, 0} {}
    Vec2(int x, int y) :_x{x}, _y{y} {}

    int x() const { return _x; }
    int y() const { return _y; }

    bool operator==(const Vec2 &pt) const
    {
        return _x == pt._x && _y == pt._y;
    }
    bool operator!=(const Vec2 &pt) const
    {
        return !(*this == pt);
    }

    Vec2 delta(const Vec2 &o) const
    {
        return Vec2{_x - o._x, _y - o._y};
    }

    double length() const;
    double distance(const Vec2 &pt) const;

    bool unit() const;
    bool adjacent(const Vec2 &pt) const;

    Vec2 shifted(const Vec2 &dxy) const
    {
        return Vec2{_x + dxy._x, _y + dxy._y};
    }
};

#endif
```

Название файла: point.cpp

```
#include <math.h>
```

```
#include "point.hpp"

double
Vec2::length() const
{
    return sqrt(_x*_x + _y*_y);
}

double
Vec2::distance(const Vec2 &pt) const
{
    return delta(pt).length();
}

bool
Vec2::unit() const
{
    return (_x || _y)
        && abs(_x) <= 1
        && abs(_y) <= 1;
}

bool
Vec2::adjacent(const Vec2 &pt) const
{
    return delta(pt).unit();
}
```

Название файла: rectmap.hpp

```
#ifndef _H_RECTMAP_HPP
#define _H_RECTMAP_HPP

#include "point.hpp"
#include "placeable.hpp"


class Cell {
    Placeable *_p = nullptr;

public:
    Cell() {}

    ~Cell()
    {
        delete _p;
    }

    Placeable *placeable() const { return _p; }
    void setPlaceable(Placeable *p) { _p = p; }
};
```

```
class RectMap;

class RectMapIter {
    RectMap *_map;
    Vec2 _pt;

public:
    RectMapIter(RectMap *map, Vec2 pt)
        :_map{map}, _pt{pt} {}
    RectMapIter(RectMap *map, int x, int y)
        :_map{map}, _pt{x, y} {}

    static RectMapIter makeNull() { return {nullptr, {0, 0}}; }

    bool operator==(const RectMapIter &o) const
    {
        return _map == o._map
            && _pt == o._pt;
    }
    bool operator!=(const RectMapIter &o) const
    {
        return !(*this == o);
    }

    int x() const { return _pt.x(); }
    int y() const { return _pt.y(); }
    Vec2 point() const { return _pt; }

    Cell &cell() const;

    bool null() const { return _map == nullptr; }
    bool valid() const;

    // Vec2 delta(const RectMapIter &o) const;

    void moveTo(Vec2 xy);
    RectMapIter otherAt(Vec2 xy) const;

    void advance(int d);
    RectMapIter advanced(int d) const;
};

class RectMap {
    const int _w, _h;
    Cell * const _storage;

public:
    RectMap(int w, int h)
        :_w{w}, _h{h}, _storage{new Cell [w * h]} {}

    int width() const { return _w; }
    int height() const { return _h; }
```

```
    Cell &at(Vec2 pt) { return _storage[pt.x() + pt.y()*_w]; }
    RectMapIter iterAt(Vec2 pt) { return RectMapIter{this, pt}; }

    ~RectMap()
    {
        delete[] _storage;
    }
};

#endif
```

Название файла: rectmap.cpp

```
#include "rectmap.hpp"


bool
RectMapIter::valid() const
{
    return x() >= 0
        && x() < _map->width()
        && y() >= 0
        && y() < _map->height();
}

Cell &
RectMapIter::cell() const
{
    return _map->at(point());
}

// Vec2
// RectMapIter::delta(const RectMapIter &o) const
// {
//     return o.point().delta(point());
// }

void
RectMapIter::moveTo(Vec2 xy)
{
    _pt = xy;
}

RectMapIter
RectMapIter::otherAt(Vec2 xy) const
{
    RectMapIter other = *this;
    other.moveTo(xy);
    return other;
}
```

```
void
RectMapIter::advance(int d)
{
    int nx = x() + d,
        w = _map->width();
    _pt = Vec2{nx % w, y() + nx / w};
}

RectMapIter
RectMapIter::advanced(int d) const
{
    RectMapIter other = *this;
    other.advance(d);
    return other;
}
```

Название файла: map.hpp

```
#ifndef _H_MAP_HPP
#define _H_MAP_HPP

#include "rectmap.hpp"

// Map interface doesn't know about cells -- instead, it only cares
// about certain kinds of Placeables


class Map;

class Unit;
class Base;

class MapIter {
    RectMapIter _it;

    friend class Map;

    MapIter(RectMapIter r)
        :_it{r} {}

public:
    static MapIter makeNull()
    {
        return MapIter{RectMapIter::makeNull()};
    }

    bool operator==(const MapIter &o) const { return _it == o._it; }
    bool operator!=(const MapIter &o) const { return _it != o._it; }

    int x() const { return _it.x(); }
    int y() const { return _it.y(); }
    Vec2 point() const { return _it.point(); }
```

```
    bool null() const { return _it.null(); }
    bool valid() const { return _it.valid(); }

    void shift(Vec2 dxy) { _it.moveTo(point().shifted(dxy)); }
    MapIter shifted(Vec2 dxy) const
    {
        return MapIter{_it.otherAt(point().shifted(dxy))};
    }

    void moveTo(Vec2 xy) { _it.moveTo(xy); }
    MapIter otherAt(Vec2 xy) const
    {
        return MapIter{_it.otherAt(xy)};
    }

    void advance(int d) { _it.advance(d); }
    MapIter advanced(int d) const
    {
        return MapIter{_it.advanced(d)};
    }

    Unit *unit() const;
    bool occupied() { return unit() != nullptr; }
    // other placeable types in the future
};

class Map {
    RectMap _rm;
    int _units_count = 0;
    int _units_max = -1;

public:
    Map(int w, int h)
        :_rm{w, h} {}

    int width() const { return _rm.width(); }
    int height() const { return _rm.height(); }
    MapIter iterAt(Vec2 pt) { return MapIter{_rm.iterAt(pt)}; }
    MapIter iterAt(int x, int y) { return iterAt({x, y}); }

    MapIter begin() { return iterAt(0, 0); }
    MapIter end() { return iterAt(0, height()); }

    MapIter addUnit(Unit *u, Vec2 pt);
    Unit *removeUnitAt(Vec2 at);
    Unit *removeUnitAt(MapIter iter)
    {
        return removeUnitAt(iter.point());
    }

    int maxUnitsCount() const { return _units_max; }
```

```
    bool setMaxUnitsCount(int x)
    {
        if (_units_count > x)
            return false;
        _units_max = x;
        return true;
    }
    int unitsCount() const { return _units_count; }
};

#endif
```

Название файла: map.cpp

```
#include "point.hpp"
#include "unit.hpp"
#include "map.hpp"


MapIter
Map::addUnit(Unit *u, Vec2 pt)
{
    if (u->hasPosition())
        return MapIter::makeNull();

    if (_units_max >= 0
        && _units_count == _units_max)
        return MapIter::makeNull();

    RectMapIter rmiter = _rm.iterAt(pt);
    Cell &cell = rmiter.cell();

    if (cell.placeable())
        return MapIter::makeNull();

    cell.setPlaceable(u);
    u->setPosition(pt);

    ++_units_count;

    return MapIter{rmiter};
}

Unit *
Map::removeUnitAt(Vec2 at)
{
    RectMapIter rmiter = _rm.iterAt(at);
    Cell &cell = rmiter.cell();
    Unit *u = dynamic_cast<Unit *>(cell.placeable());

    if (u) {
        --_units_count;
```

```
        cell.setPlaceable(nullptr);
        u->unsetPosition();
    }

    return u;
}

Unit *
MapIter::unit() const
{
    return dynamic_cast<Unit *>(_it.cell().placeable());
}
```

Название файла: placeable.hpp

```
#ifndef _H_PLACEABLE_HPP
#define _H_PLACEABLE_HPP

class Placeable {
    bool _placed = false;
    Vec2 _pos;

public:
    bool
    hasPosition() const { return _placed; }

    const Vec2 &
    position() const
    {
        return _pos;
    }

    void
    setPosition(const Vec2 &pos)
    {
        _pos = pos;
        _placed = true;
    }

    void
    unsetPosition()
    {
        _placed = false;
    }

    virtual ~Placeable() {}
};

#endif
```

Название файла: unit.hpp

```
#ifndef _H_UNIT_HPP
#define _H_UNIT_HPP

#include <utility>
#include <random>
#include <math.h>

#include "map.hpp"


extern std::default_random_engine global_random;


class MovePolicy {
public:
    virtual bool canMove(const Unit *u, MapIter to) =0;
    virtual ~MovePolicy() {}
};

enum class AttackKind {
    sword, spear, cavalry, arrow, stone, rock, bolt,
};

// NOTE: can't do area damage
class AttackPolicy {
public:
    virtual bool canAttackTo(const Unit *u, MapIter to) =0;

    virtual MapIter actualPosition(const Unit *, MapIter to)
    {
        return to;
    }

    // returns kind and base damage
    virtual std::pair<AttackKind, int>
    baseAttack(const Unit *u, MapIter to) =0;

    virtual ~AttackPolicy() {}
};

struct DamageSpec {
    int base_damage, damage_spread;

    int evaluate() const
    {
        std::uniform_int_distribution<>
            dist {-damage_spread, damage_spread};

        return base_damage + dist(global_random);
    }
};
```

```
class DefensePolicy {
protected:
    static DamageSpec
    make_spec(double base, double spread)
    {
        return DamageSpec{(int)base, (int)spread};
    }

    static DamageSpec
    defense_level(double k, int dmg)
    {
        return make_spec(round(1.0*dmg/k),
                         round(0.25*dmg/k));
    }

    static DamageSpec
    normal_defense(double dmg)
    {
        return defense_level(1.0, dmg);
    }

public:
    // returns base damage and spread
    virtual DamageSpec
    actualDamage(const Unit *u, AttackKind kind, int base) =0;

    virtual ~DefensePolicy() {}
};

class Unit: public Placeable {
    // Controls the lifetime of policies once they are given to the unit
    // through the constructor.
    MovePolicy *_move_policy;
    AttackPolicy *_attack_policy;
    DefensePolicy *_defence_policy;

    int _health, _base_health;

public:
    Unit(MovePolicy *move,
         AttackPolicy *attack,
         DefensePolicy *defense,
         int base_health)
        :_move_policy{move},
         _attack_policy{attack},
         _defence_policy{defense},
         _health{base_health},
         _base_health{base_health} {}

    ~Unit()
    {
        delete _move_policy;
```

```
        delete _attack_policy;
        delete _defence_policy;
    }

    int
    health() const { return _health; }
    int
    baseHealth() const { return _base_health; }
    double
    relativeHealth() const { return _health / (double)_base_health; }
    bool
    alive() const { return health() > 0; }

    void
    takeDamage(int dmg) { _health -= dmg; }

    bool
    canMove(MapIter to) const
    {
        return _move_policy->canMove(this, to);
    }

    bool
    canAttackTo(MapIter to) const
    {
        return _attack_policy->canAttackTo(this, to);
    }

    MapIter
    actualPosition(MapIter to) const
    {
        return _attack_policy->actualPosition(this, to);
    }

    std::pair<AttackKind, int>
    baseAttack(MapIter to) const
    {
        return _attack_policy->baseAttack(this, to);
    }

    DamageSpec
    actualDamage(AttackKind kind, int base) const
    {
        return _defence_policy->actualDamage(this, kind, base);
    }
};

#endif
```

Название файла: unit.cpp

```
#include <random>
```

```
#include "unit.hpp"

std::default_random_engine global_random {};
```

Название файла: common_policies.hpp

```
#ifndef _H_COMMON_POLICIES_HPP
#define _H_COMMON_POLICIES_HPP

#include "map.hpp"
#include "pathfinder.hpp"


class BasicMovement: public MovePolicy {
    int _steps_per_turn;

public:
    BasicMovement(int n)
        :_steps_per_turn{n} {}

    virtual bool
    canMove(const Unit *u, MapIter to) override
    {
        MapIter from = to.otherAt(u->position());
        PathFinder pf {from, to, _steps_per_turn};
        return pf.run();
    }
};

class BasicDefense: public DefensePolicy {
    double _lvl;

public:
    explicit BasicDefense(double level=1.0)
        :_lvl{level} {}

    virtual DamageSpec
    actualDamage(const Unit *, AttackKind, int base) override
    {
        return normal_defense(base);
    }
};

class DefenseLevelDeco: public DefensePolicy {
    // Controls nested policy lifetime
    DefensePolicy *_p;
    AttackKind _kind;
    double _lvl;

public:
    DefenseLevelDeco(DefensePolicy *p,
```

```
                    AttackKind kind,
                    double level)
        :_p{p}, _kind{kind}, _lvl{level} {}

    ~DefenseLevelDeco()
    {
        delete _p;
    }

    virtual DamageSpec
    actualDamage(const Unit *u, AttackKind kind, int base) override
    {
        if (kind == _kind)
            return defense_level(_lvl, base);
        return _p->actualDamage(u, kind, base);
    }

    static DefenseLevelDeco *
    defense_level_deco(AttackKind kind, double lvl, DefensePolicy *p)
    {
        return new DefenseLevelDeco {p, kind, lvl};
    }

    static DefenseLevelDeco *
    good_defense_deco(AttackKind kind, DefensePolicy *p)
    {
        return defense_level_deco(kind, 2.0, p);
    }

    static DefenseLevelDeco *
    vulnerability_deco(AttackKind kind, DefensePolicy *p)
    {
        return defense_level_deco(kind, 0.5, p);
    }
};

#endif
```

Название файла: melee_units.hpp

```
#ifndef _H_MELEE_UNITS_HPP
#define _H_MELEE_UNITS_HPP

#include <utility>

#include "point.hpp"
#include "unit.hpp"
#include "common_policies.hpp"


class MeleeAttack: public AttackPolicy {
    AttackKind _kind;
```

```
    int _base_damage;

public:
    MeleeAttack(AttackKind kind, int base_dmg)
        :_kind{kind}, _base_damage{base_dmg} {}

    virtual bool
    canAttackTo(const Unit *u, MapIter to) override
    {
        return to.unit() != nullptr
            && to.point().adjacent(u->position());
    }

    virtual std::pair<AttackKind, int>
    baseAttack(const Unit *u, MapIter)
    {
        return std::make_pair(
            _kind,
            int(_base_damage * u->relativeHealth()));
    }
};

class BasicMeleeUnit: public Unit {
public:
    BasicMeleeUnit(int speed,
                   AttackKind attack_kind,
                   int base_dmg,
                   DefensePolicy *def,
                   int base_health)
        :Unit{new BasicMovement {speed},
              new MeleeAttack {attack_kind, base_dmg},
              def, base_health} {}
};

namespace units {
    class Swordsman: public BasicMeleeUnit {
    public:
        Swordsman() :BasicMeleeUnit{
            2,
            AttackKind::sword, 40,
            DefenseLevelDeco::good_defense_deco(
                AttackKind::spear,
                DefenseLevelDeco::vulnerability_deco(
                    AttackKind::cavalry,
                    new BasicDefense {})),
            100} {}
    };

    class Spearsman: public BasicMeleeUnit {
    public:
        Spearsman() :BasicMeleeUnit{
            2,
```

```
            AttackKind::spear, 75,
            DefenseLevelDeco::good_defense_deco(
                AttackKind::cavalry,
                DefenseLevelDeco::vulnerability_deco(
                    AttackKind::spear,
                    new BasicDefense {})),
            75} {}
    };

    class Cavalry: public BasicMeleeUnit {
    public:
        Cavalry() :BasicMeleeUnit{
            3,
            AttackKind::cavalry, 50,
            DefenseLevelDeco::good_defense_deco(
                AttackKind::sword,
                DefenseLevelDeco::vulnerability_deco(
                    AttackKind::spear,
                    new BasicDefense {})),
            75} {}
    };
}

#endif
```

Название файла: ranged_units.hpp

```
#ifndef _H_RANGED_UNITS_HPP
#define _H_RANGED_UNITS_HPP

#include <utility>
#include <math.h>

#include "point.hpp"
#include "unit.hpp"
#include "common_policies.hpp"


class RangedAttack: public AttackPolicy {
    AttackKind _kind;
    int _base_damage;
    double _min_distance, _max_distance;
    double _dist_pow;

    static double
    distance(const Unit *u, MapIter to)
    {
        return to.point().distance(u->position());
    }

public:
    RangedAttack(AttackKind kind,
```

```
                int base_dmg,
                double min_dist,
                double max_dist,
                double dist_pow)
        :_kind{kind},
         _base_damage{base_dmg},
         _min_distance{min_dist},
         _max_distance{max_dist},
         _dist_pow{dist_pow} {}

    virtual bool
    canAttackTo(const Unit *u, MapIter to) override
    {
        double dist = distance(u, to);
        return dist >= _min_distance
            && dist <= _max_distance;
    }

    virtual std::pair<AttackKind, int>
    baseAttack(const Unit *u, MapIter to) override
    {
        double dist = distance(u, to);
        return std::make_pair(
            _kind,
            int(_base_damage
                * u->relativeHealth()
                / pow(dist, _dist_pow)));
    }
};

class BasicRangedUnit: public Unit {
public:
    BasicRangedUnit(int speed,
                    AttackKind attack_kind,
                    int base_dmg,
                    double max_dist,
                    double dist_pow,
                    DefensePolicy *def,
                    int base_health)
        :Unit{new BasicMovement {speed},
              new RangedAttack {attack_kind, base_dmg,
                                1., max_dist, dist_pow},
              def, base_health} {}
};

namespace units {
    class Archer: public BasicRangedUnit {
    public:
        Archer() :BasicRangedUnit{
            2,
            AttackKind::arrow, 50, 5., .20,
            new BasicDefense {0.9},
```

```
            40} {}
    };

    class Slinger: public BasicRangedUnit {
    public:
        Slinger() :BasicRangedUnit{
            2,
            AttackKind::stone, 60, 3., .30,
            new BasicDefense {.09},
            50} {}
    };
}

#endif
```

Название файла: catapult_units.hpp

```
#ifndef _H_CATAPULT_UNITS_HPP
#define _H_CATAPULT_UNITS_HPP

#include <utility>
#include <random>
#include <math.h>

#include "point.hpp"
#include "unit.hpp"
#include "common_policies.hpp"
#include "ranged_units.hpp"


class CatapultAttack: public RangedAttack {
    double _spread_tang, _spread_normal;

    struct FVec2 {
        double x, y;

        explicit FVec2(const Vec2 &v)
            :x{(double)v.x()}, y{(double)v.y()} {}

        FVec2(double x, double y)
            :x{x}, y{y} {}

        operator Vec2() const
        {
            return Vec2{int(round(x)), int(round(y))};
        }

        FVec2
        orthogonal() const { return {y, -x}; }

        FVec2 &
        operator*=(double a)
```

```
        {
            x *= a;
            y *= a;
            return *this;
        }
        FVec2
        operator*(double a) const
        {
            FVec2 tmp {*this};
            return tmp *= a;
        }

        FVec2
        normalized() const { return (*this) * (1/sqrt(x*x + y*y)); }

        FVec2 &
        operator+=(const FVec2 &dxy)
        {
            x += dxy.x;
            y += dxy.y;
            return *this;
        }
        FVec2
        operator+(const FVec2 &dxy) const
        {
            FVec2 tmp{*this};
            return tmp += dxy;
        }

        FVec2
        apply(double t, double n) const
        {
            return normalized() * t
                + orthogonal().normalized() * n;
        }
    };

public:
    CatapultAttack(AttackKind kind,
                   int base_dmg,
                   double min_dist,
                   double max_dist,
                   double dist_pow,
                   double spread_t,
                   double spread_n)
        :RangedAttack{kind, base_dmg, min_dist, max_dist, dist_pow},
         _spread_tang{spread_t},
         _spread_normal{spread_n} {}

    virtual MapIter
    actualPosition(const Unit *u, MapIter to) override
    {
```

```
        Vec2 dest = to.point();
        Vec2 delta = dest.delta(u->position());
        FVec2 fdelta {delta};

        std::uniform_real_distribution<>
            t_dist {-_spread_tang, _spread_tang},
            n_dist {-_spread_normal, _spread_normal};

        double
            t = t_dist(global_random),
            n = n_dist(global_random);

        FVec2 result = fdelta.apply(t, n);
        return to.shifted(Vec2{result});
    }
};

class BasicCatapultUnit: public Unit {
public:
    BasicCatapultUnit(AttackKind attack_kind,
                      int base_dmg,
                      double min_dist,
                      double max_dist,
                      double dist_pow,
                      double spread_t,
                      double spread_n,
                      int base_health)
        :Unit{new BasicMovement {1},
              new CatapultAttack {attack_kind,
                                  base_dmg, min_dist, max_dist,
                                  dist_pow, spread_t, spread_n},
              DefenseLevelDeco::good_defense_deco(
                  AttackKind::arrow,
                  new BasicDefense {0.75}),
              base_health} {}
};

namespace units {
    class Onager: public BasicCatapultUnit {
    public:
        Onager() :BasicCatapultUnit{
            AttackKind::rock, 90,
            3, 10, 0.05,
            0.2, 0.1,
            30} {}
    };

    class BoltThrower: public BasicCatapultUnit {
    public:
        BoltThrower() :BasicCatapultUnit{
            AttackKind::bolt, 110,
            2, 6, 0.15,
```

```
            0.05, 0.05,
            20} {}
    };
}

#endif
```