Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Волгоградский государственный технический университет»

Факультет электроники и вычислительной техники
Направление 09.03.04 «Программная инженерия»
Кафедра «Программное обеспечение автоматизированных систем»

Дисциплина «Объектно-ориентированный анализ и программирование»

Утверждаю
и.о. зав. кафедрой _____ Сычев О.А.

ЗАДАНИЕ
на курсовую работу

Студент: Матвеев С.А.
Группа: ПрИн-368

1. Тема: «Проектирование и реализация программы с использованием объектно-ориентированного подхода» (индивидуальное задание – вариант №10_02)
Утверждена приказом от «05» февраля 2025г. № 183-ст

2. Срок представления работы к защите « 06 » июня 2025 г.

3. Содержание пояснительной записки:
формулировка задания, требования к программе, структура программы, типовые процессы в программе, человеко-машинное взаимодействие, код программы и модульных тестов

4. Перечень графического материала:
_____

5. Дата выдачи задания «14» февраля 2025 г.

Руководитель проекта: _____ Литовкин Д.В.

Задание принял к исполнению: _____ Матвеев С.А.
«14» февраля 2025 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Волгоградский государственный технический университет»

Факультет электроники и вычислительной техники
Кафедра «Программное обеспечение автоматизированных систем»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе

по дисциплине «Объектно-ориентированный анализ и программирование»
на тему: «Проектирование и реализация программы с использованием объектно-ориентированного подхода»
(индивидуальное задание – вариант №10_2)

Студент: Матвеев С.А.
Группа: ПрИн-368

Работа зачтена с оценкой _____ «___» _____ 2025 г.

Руководитель проекта, нормоконтроллер _____ Литовкин Д.В.

Волгоград 2025 г.

Содержание

1 Формулировка задания **3**
2 Нефункциональные требования **4**
3 Первая итерация разработки **4**
3.1 Формулировка упрощённого варианта задания 5
3.2 Функциональные требования (сценарии) 5
3.3 Словарь предметной области 7
3.4 Структура программы на уровне классов 10
3.5 Типовые процессы в программе 13
3.6 Человеко-машинное взаимодействие 16
3.7 Реализация ключевых классов 17
3.8 Реализация ключевых тестовых случаев 26
4 Вторая итерация разработки **37**
4.1 Функциональные требования (сценарии) 37
4.2 Словарь предметной области 39
4.3 Структура программы на уровне классов 42
4.4 Типовые процессы в программе 43
4.5 Человеко-машинное взаимодействие 46
4.6 Реализация ключевых классов 49
4.7 Реализация ключевых тестовых случаев 51
Перечень замечаний к работе **55**

1 Формулировка задания

Правила игры «Забавы богов»:

- «жизнь» разыгрывается на бесконечном клеточном поле (плоской сфере);
- у каждой клетки 8 соседних клеток;
- в каждой клетке может жить существо;
- существо с двумя или тремя соседями выживает в следующем поколении, иначе погибает от одиночества или перенаселённости;
- в пустой клетке с тремя соседями в следующем поколении рождается существо.
- первоначально каждый игрок-человек (бог) порождает K существ своей расы в своей половине поля, т.е. свою колонию;
- далее боги "забывают" о своей колонии на T эпох;
- в каждой эпохе существа рождаются и умирают в соответствии с правилами игры «Жизнь», но со следующими отличиями: порождается существо той расы, представителей которых больше вокруг данной клетки; если существ одинаковое количество, то действует вероятностный выбор;
- через T эпох игрок может породить или уничтожить N существ своей расы на своей половине поля;
- цель игры — на поле должны остаться существа одной расы.

Дополнительные требования:

- предусмотреть в программе точки расширения, используя которые можно реализовать вариативную часть программы (в дополнение к базовой функциональности).

Вариативность:

- предусмотреть возможность создания создание плоских клеточных полей разной формы. НЕ изменяя ранее созданные классы, а используя точки расширения, реализовать: поле в форме ромба.

Реализовать:

- Игровое поле.
- Игровую модель.

- Игровой контроллер.

2 Нефункциональные требования

1. Программа должна быть реализована на языке c++ 20 с использованием библиотеки SFML 3
2. Форматирование исходного кода программы должно соответствовать Google c++ Code Style Gyde

3 Первая итерация разработки

3.1 Формулировка упрощённого варианта задания

Правила игры «Сломанный робот»:

- «жизнь» разыгрывается на бесконечном клеточном поле (плоской сфере);
- у каждой клетки 8 соседних клеток;
- в каждой клетке может жить существо;
- существо с двумя или тремя соседями выживает в следующем поколении, иначе погибает от одиночества или перенаселённости;
- в пустой клетке с тремя соседями в следующем поколении рождается существо.
- первоначально каждый игрок-человек (бог) порождает K существ своей расы в своей половине поля, т.е. свою колонию;
- далее боги "забывают" о своей колонии на T эпох;
- в каждой эпохе существа рождаются и умирают в соответствии с правилами игры «Жизнь», но со следующими отличиями: порождается существо той расы, представителей которых больше вокруг данной клетки; если существ одинаковое количество, то действует вероятностный выбор;
- через T эпох игрок может породить или уничтожить N существ своей расы на своей половине поля;
- цель игры — на поле должны остаться существа одной расы.

3.2 Функциональные требования (сценарии)

1) Сценарий «Игра завершается победой одного из Игроков».

1. Пользователем выбирается количество игроков.
2. **По указанию** Пользователя**,** Игра стартует.
3. **По указанию** Игры, Поле создаёт ячейки.
4. **По указанию** Игры, Пул Игроков создает Сущности игроков и присваивает им соответствующие им уникальные типы Существ.
5. **По запросу** Пула Игроков, Игра присваивает каждому Игроку уникальную область поля.
6. **Делать {**
   6.1. **Исполнить** дочерний сценарий «Подготовка поля игроком».
   6.2. **Исполнить** дочерний сценарий «Вычисление модели».
   6.3. **} Пока** на поле есть Существа **И** на поле есть Существа двух типов.
7. Игра считает победителем Игрока, чьи Существа остались на поле единственными.
8. **Сценарий завершается**.

1.1) Альтернативный сценарий «Досрочное завершение игры пользователем». Сценарий **выполняется в любой точке** главного сценария.

1. **По указанию пользователя,** программа завершается без определения победителя.
2. **Сценарий завершается.**

1.2) Альтернативный сценарий «Завершение Игры ничьей». Сценарий **выполняется после главного цикла** главного сценария.

1. Если на Поле не осталось Существ, то игра завершается ничьей.

2. Сценарий завершается.

1.3) Дочерний сценарий «Подготовка полем игроком»

1. **Делать {**

1. **Делать** Пока не выберутся все Игроки из Пула {
    1. Игра выбирает идущего по порядку Игрока и разрешает ему порождение и уничтожение его Существ в его области поля (*пользователь при нажатии на соответствующую область поля будет размещать/удалять соответствующий текущему игроку тип Существ*).
    2. Игрок создает Существо нужного типа в Разрешенной Клетке, **Если** нужно.
    3. **По запросу** Игрока, Область Поля размещает или уничтожает Существо на нужной позиции.
    4. Игра ждет пока Игрок не породит К Существ.

}

1.4) Дочерний сценарий «Вычисление модели»

9. Делать Пока не пройдет Т эпох {

9.1. **По запросу** Игры, Поле **сообщает** Игре позиции всех Существ.

9.2. **Поле** проходиться по всем Существам и соседним позициям

9.2.1.1. **Либо**

9.3. Выбирает уничтожить Существо, **Если** соседей меньше 2 или больше 3.

9.3.1. **Либо**

9.4. Выбирает породить Существо определенного типа в соответствующий позиции, **Если** соседей больше или 3 данного.

9.5. **По запросу** Игры, Поле размещает или уничтожает Существо на соответствующей позиции.

}

3.3 Словарь предметной области

Игра - знает о Поле и Пуле Игроков. Игра инициирует создание. Игра определяет очередного активного Игрока, набор существ на Поле и окончание игры (и победителя).

знает	● о Поле ● о Пуле Игроков ● о Существе
умеет	● инициировать создание Поля. ● инициализировать создание Пула Игроков ● определять очередного Игрока, который может обрабатывать запросы пользователя ● определять окончание игры (и победителя)
предназначение	● Организация общего игрового цикла

Поле - область заданной формы, состоящая из ячеек. Знает о Существах, находящихся на Поле.

знает	● свои размеры и форму ● Ячейки ● о Существах
умеет	● создавать себя из Ячеек ● предоставлять доступ к Ячейкам ● выдавать Существ
предназначение	● Контейнер Ячеек и Сущностей, которые располагаются внутри Ячеек

Ячейка - квадратная область Поля. Знает о четырёх соседних Ячейках и граничащих с ней Стенах. На ней может располагаться сущность.

знает	● соседние Ячейки ● о наличии в себе Существа
умеет	● устанавливать соседство с другой Ячейкой ● предоставляет доступ к размещенному в себе Существу
предназначение	● Контейнер для Существа

Существо - сущность необходимая для вычисления текущего состояния игры. Знает об игроке-хозяине и своем типе.

знает	● своего Игрока ● свой Тип
умеет	● Возвращать свой тип и игрока
предназначение	● Маркировать Игрока на поле

Пул Игроков - контейнер игроков. Умеет создавать игроков, инициализируя их типом Существа и Областью Поля, и выдавать очередного Игрока.

знает	● о Игроках ● о Существах ● о Областях Поля
умеет	● создавать Игроков ● выдавать очередного Игрока
предназначение	● Хранить игроков

Игрок - знает об Области Поля и Существе. Умеет создавать свой тип существ и размещать их на Поле посредством Области Поля.

знает	● свою Область поля ● свое Существо
умеет	● создавать Существо ● размещать его на Поле
предназначение	● абстракция игрока для пользователя и Игры

Область поля - знает о своей форме, размерах и Поле. Необходима для ограничения прав Игрока на размещение Существ до определенной области.

знает	● свои форму и размер ● Поле
умеет	● размещать существо в определенном пространстве поля
предназначение	● ограничение прав Игрока на размещение Сущностей

3.4 Структура программы на уровне классов

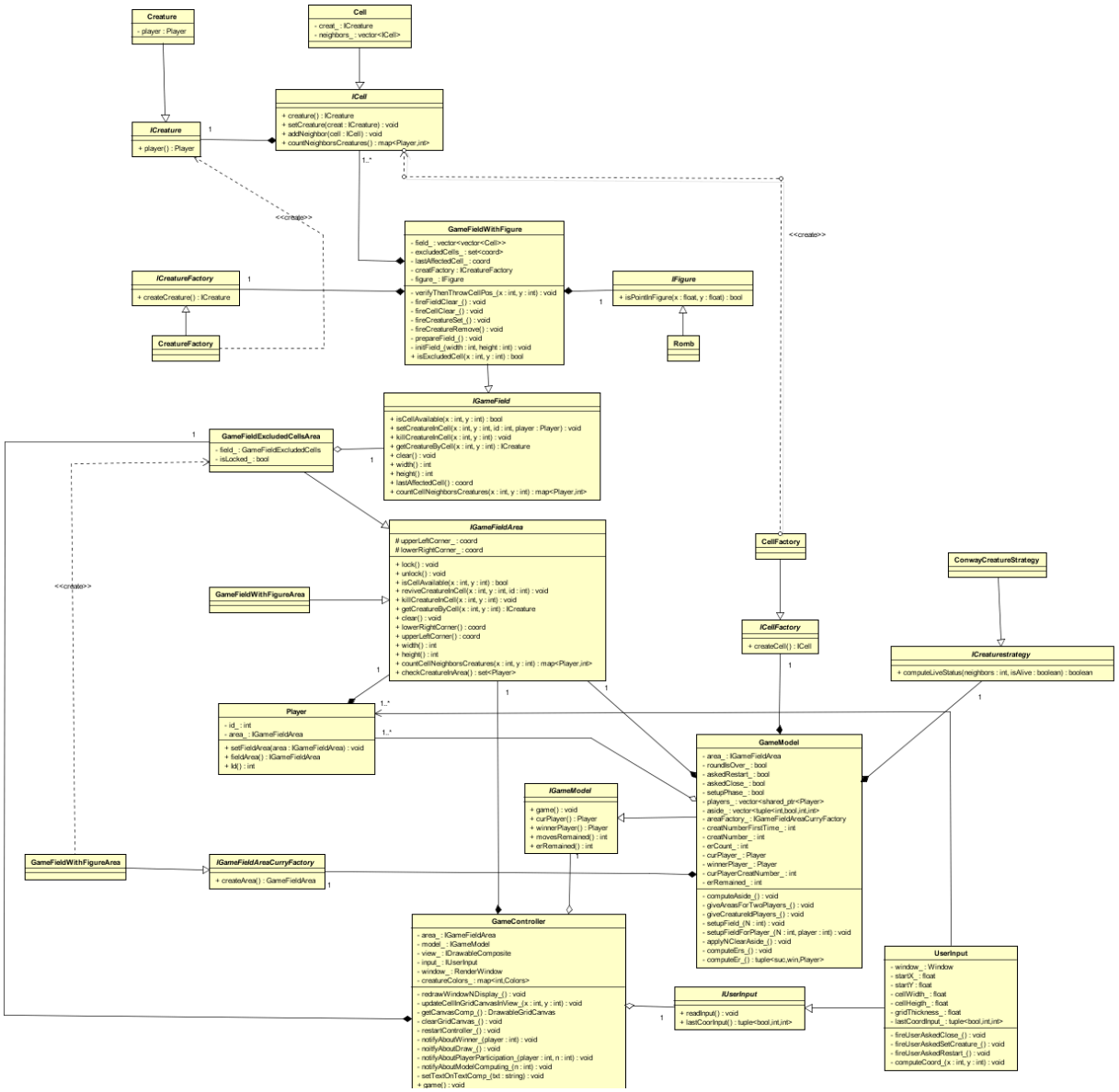

Рисунок 1 - Диаграмма классов вычислительной модели

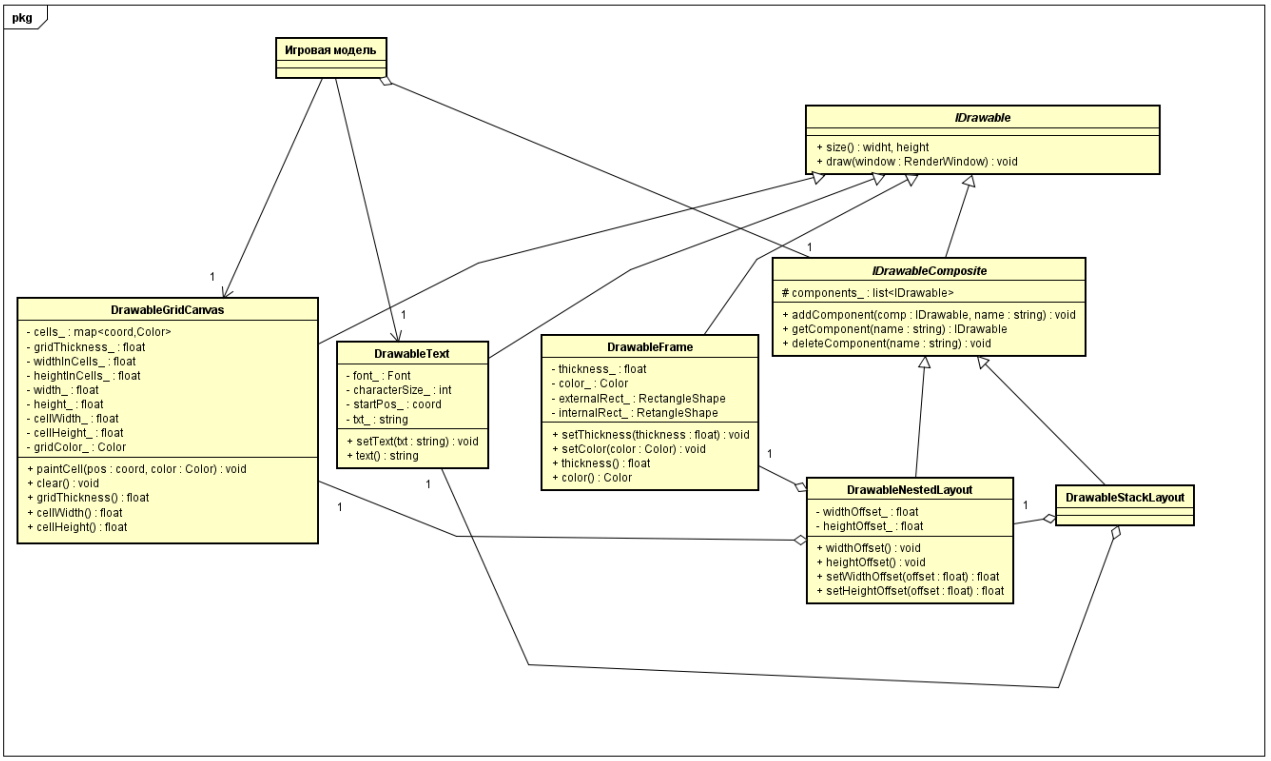

Рисунок 2 - Диаграмма классов представления

3.5 Типовые процессы в программе

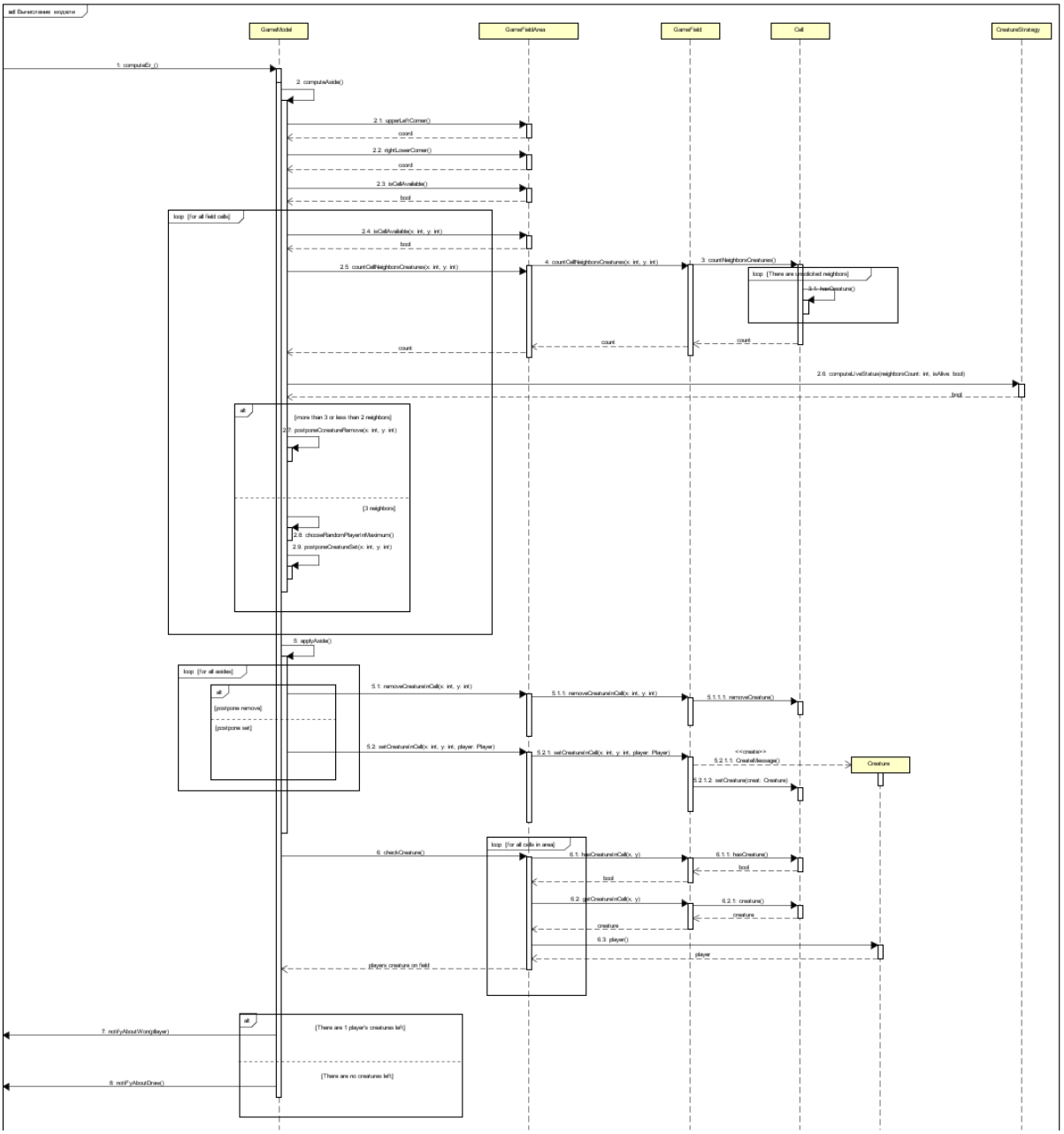

Рисунок 3 - Диаграмма последовательности для вычислительной модели. Вычисление одной эры.

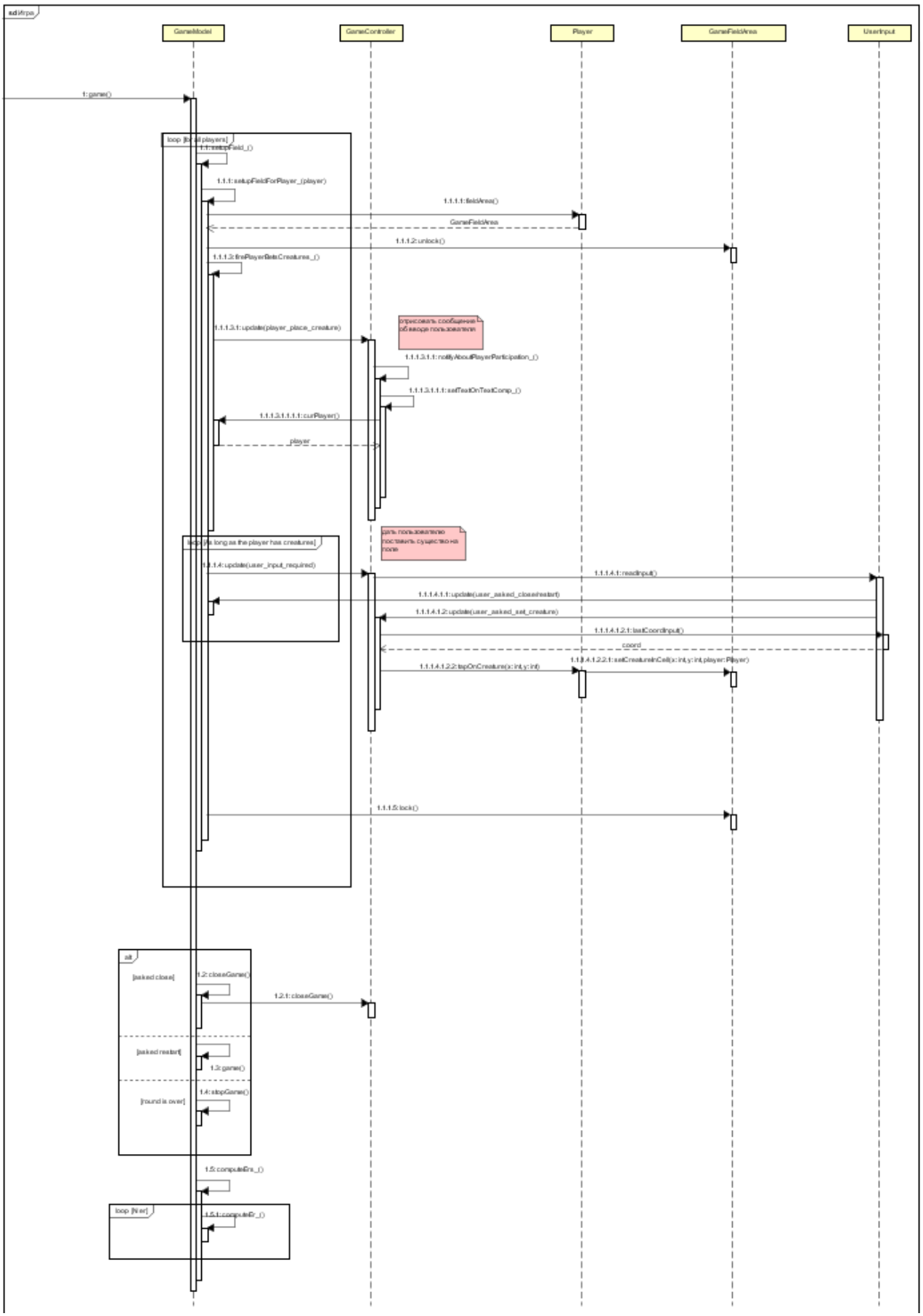

Рисунок 4 – Диаграмма последовательности для вычислительной модели. Обработка выигрыша и ничьей.

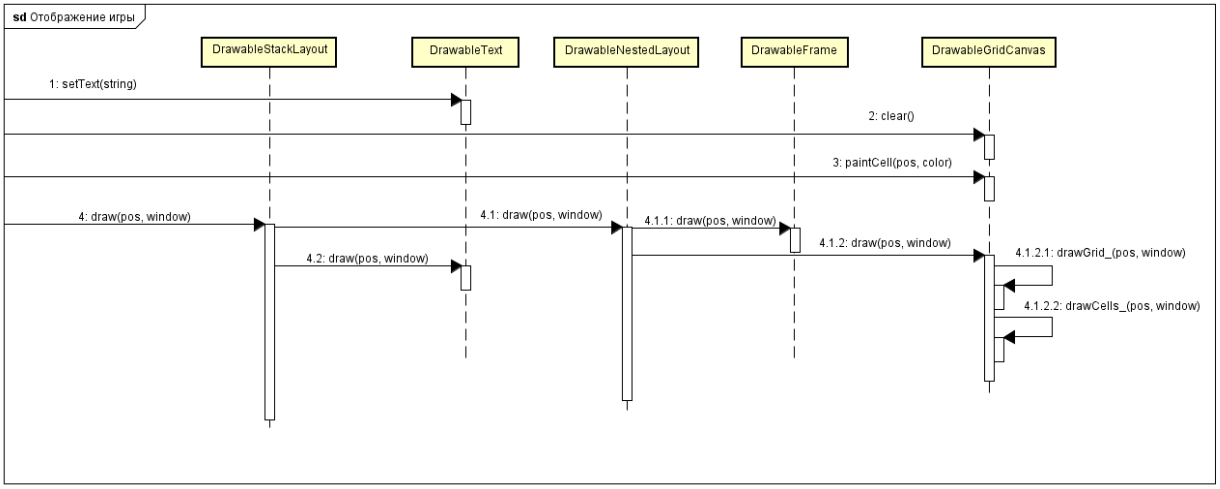

Рисунок 5 – Диаграмма последовательности представления.

3.6 Человеко-машинное взаимодействие

Общий вид главного экрана программы представлен ниже. На нём располагается игровое поле, на котором изображены существа двух игроков.

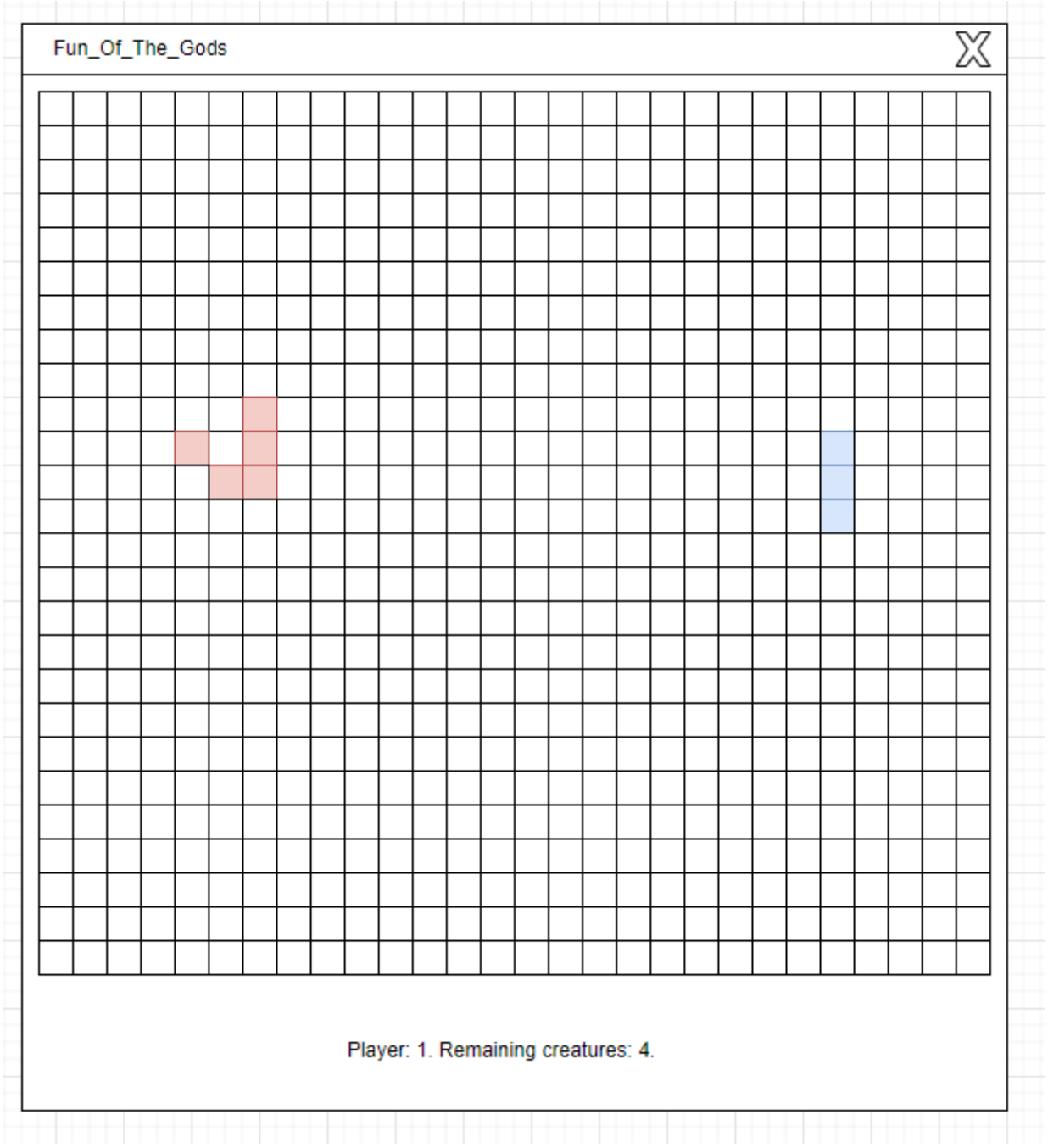

Рисунок 6 – Общий вид главного окна программы

3.7 Реализация ключевых классов

```
class GameModel :
    public IGameModel,
    public observer::IObserver,
    public subject::ISubject,
    public std::enable_shared_from_this<GameModel>
{
    using IGameFieldArea = game_field_area::IGameFieldArea;
    using IGameFieldAreaCurryFactory = factory::IGameFieldAreaCurryFactory;
    using ICreatureStrategy = creature_strategy::ICreatureStrategy;

public:
    GameModel(
        int creatNumberFirstTime,
        int creatNumber,
        int erCount,
        std::unique_ptr<IGameFieldArea> area,
        std::unique_ptr<IGameFieldAreaCurryFactory> areaFactory,
        const std::vector<std::shared_ptr<player::Player>>& players,
        std::unique_ptr<ICreatureStrategy> creatStrategy);

public:
    void attach(
        std::shared_ptr<observer::IObserver> obs, int event_t) override;
    void detach(
        std::weak_ptr<observer::IObserver> obs, int event_t) override;

private:
    void notify(int event_t) override;

public:
    void update(int event_t) override;

public:
    void game() override;
    std::shared_ptr<player::Player> curPlayer() const noexcept override;
    std::shared_ptr<player::Player> winnerPlayer() const noexcept override;
    int movesRemained() const noexcept override;
    int erRemained() const noexcept override;

private:
    void giveAreasForTwoPlayers_();
    void setupField_(int N);
    void setupFieldForPlayer_(int creatureNumber,
            std::shared_ptr<player::Player> player);
    void computeErs_(int erCount);
#ifdef TEST
public:
#endif
    std::tuple<bool, bool, std::shared_ptr<player::Player>>
        computeEr_();

#ifdef TEST
private:
#endif
    void computeAside_();
    void applyNClearAside_();
    void restartModel_();
    void fireWinnerDeterminate_();
    void fireThereWasDraw_();
    void firePlayerBetsCreatures_();
    void fireGameModelCalculatedEr_();
    void fireUserInputRequired();

private:
    const int creatNumberFirstTime_;
    const int creatNumber_;
    const int erCount_;
    std::unique_ptr<IGameFieldAreaCurryFactory> areaFactory_;
    std::unique_ptr<IGameFieldArea> area_;
    std::unique_ptr<ICreatureStrategy> creatStrategy_;
```

```
    // deferred field changes
    std::vector<
        std::tuple< std::shared_ptr<player::Player>, bool, int, int>> aside_;

    std::vector<std::shared_ptr<player::Player>> players_;

    bool roundIsOver_ = false;
    bool askedRestart_ = false;
    bool askedClose_ = false;
    bool setupPhase_ = false;

    std::shared_ptr<player::Player> curPlayer_;
    std::shared_ptr<player::Player> winnerPlayer_;
    int curPlayerCreatNumber_;
    int erRemained_;
};




GameModel::GameModel(
        int creatNumberFirstTime,
        int creatNumber,
        int erCount,
        std::unique_ptr<IGameFieldArea> area,
        std::unique_ptr<IGameFieldAreaCurryFactory> areaFactory,
        const std::vector<std::shared_ptr<player::Player>>& players,
        std::unique_ptr<ICreatureStrategy> creatStrategy) :
    creatNumberFirstTime_(creatNumberFirstTime)
    , creatNumber_(creatNumber)
    , erCount_(erCount)
    , area_(std::move(area))
    , areaFactory_(std::move(areaFactory))
    , players_(players)
    , creatStrategy_(std::move(creatStrategy))
{
    giveAreasForTwoPlayers_();
}

void GameModel::attach(
    std::shared_ptr<observer::IObserver> obs, int event_t)
{ subject::ISubject::attach(obs, event_t); }

void GameModel::detach(
    std::weak_ptr<observer::IObserver> obs, int event_t)
{ subject::ISubject::detach(obs, event_t); }

void GameModel::notify(int event_t)
{ subject::ISubject::notify(event_t); }

void GameModel::update(int event_t)
{
    using evt_t = game_event::event_t;
    auto evt = static_cast<evt_t>(event_t);
    switch (evt) {
        case evt_t::USER_ASKED_CLOSE: {
            askedClose_ = true;
            break;
        }
        case evt_t::USER_ASKED_RESTART: {
            askedRestart_ = true;
            restartModel_();
            break;
        }
        case evt_t::CREATURE_REMOVE_IN_FIELD: {
            if (setupPhase_) {
                ++curPlayerCreatNumber_;
            }
            break;
        }
        case evt_t::CREATURE_SET_IN_FIELD: {
```

```
            if (setupPhase_) {
                --curPlayerCreatNumber_;
            }
        }
    }
}

void GameModel::game() {
    while (!askedClose_) {
        askedRestart_ = false;
        setupField_(creatNumberFirstTime_);
        while (!askedClose_ && !askedRestart_ && !roundIsOver_) {
            computeErs_(erCount_);
            if (!roundIsOver_) setupField_(creatNumber_);
        }
        roundIsOver_ = false;
        while ((!askedRestart_) && (!askedClose_)) {
            fireUserInputRequired();
        }
    }
}

std::shared_ptr<player::Player>
GameModel::curPlayer() const noexcept {
    return curPlayer_;
}

std::shared_ptr<player::Player>
GameModel::winnerPlayer() const noexcept {
    return winnerPlayer_;
}

int GameModel::movesRemained() const noexcept {
    return curPlayerCreatNumber_;
}

int GameModel::erRemained() const noexcept  {
    return erRemained_;
}

void GameModel::giveAreasForTwoPlayers_() {
    std::pair<int, int> ul1 = {0, 0};
    std::pair<int, int> lr1 = {area_->width() / 2 - 1,
                               area_->height() - 1};
    auto area1 = areaFactory_->createArea(ul1, lr1);
    players_[0]->setFieldArea(std::move(area1));

    std::pair<int, int> ul2 = {area_->width() / 2, 0};
    std::pair<int, int> lr2 = {area_->width() - 1,
                               area_->height() - 1};
    auto area2 = areaFactory_->createArea(ul2, lr2);
    players_[1]->setFieldArea(std::move(area2));
}

void GameModel::setupField_(int creatureNumber) {
    for (auto&& p : players_) {
        setupFieldForPlayer_(creatNumber_, p);
    }
}

void GameModel::setupFieldForPlayer_(int creatureNumber,
    std::shared_ptr<player::Player> player)
{
    setupPhase_ = true;
    curPlayer_ = player;
    auto&& area = player->fieldArea();
    curPlayerCreatNumber_ = creatureNumber;
    area.unlock();
    while (curPlayerCreatNumber_ && !askedClose_ && !askedRestart_) {
        firePlayerBetsCreatures_();
        fireUserInputRequired();
    }
    area.lock();
```

```
    setupPhase_ = false;
}

void GameModel::computeErs_(int erCount) {
    while (!roundIsOver_ && erCount) {
        erRemained_ = erCount--;
        fireGameModelCalculatedEr_();
        auto [suc, win, player] = computeEr_();
        if (!suc) {
            roundIsOver_ = true;
            winnerPlayer_ = player;
            if (!win) {
                fireThereWasDraw_();
            } else {
                fireWinnerDeterminate_();
            }
        } else {
            std::this_thread::sleep_for(
                std::chrono::milliseconds(250));
        }
    }
}

std::tuple<bool, bool, std::shared_ptr<player::Player>>
GameModel::computeEr_()
{
    // рассчитать состояние поля в следующий момент и отложить его
    computeAside_();
    // применить отложенное состояние на поле
    applyNClearAside_();
    // получить всех список игроков, чьи существа еще есть на поле
    auto count = area_->checkCreatureInArea();
    if (count.size() < 2) {
        if (count.size() == 1) {
            return {false, true, *count.begin()};
        }
        return {false, false, nullptr};
    }
    return {true, false, nullptr};
}

void GameModel::computeAside_() {
    namespace views = std::ranges::views;
    auto luCorner = area_->upperLeftCorner();
    auto rdCorner = area_->lowerRightCorner();

    for (auto y = luCorner.second; y <= rdCorner.second; ++y) {
        for (auto x = luCorner.first; x <= rdCorner.first; ++x) {
            if (area_->isCellAvailable(x, y)) {
                auto ne = area_->countCellNeighborsCreatures(x, y);
                // посчитать количество существ всех игроков в соседях
                int neSum = std::accumulate(ne.begin(), ne.end(),
                                0, [] (int i, auto&& p) { return i + p.second; });
                // если существо есть в клетке - оно живо
                bool isAlive = area_->hasCreatureInCell(x, y);

                // принять решение через стратегию
                if (creatStrategy_->computeLiveStatus(neSum, isAlive)) {
                    if (!isAlive) {
                        // получить значение с максимальным количеством существ
                        auto max = std::max_element(ne.begin(), ne.end(),
                                                    [] (auto&& l, auto&& p)
                                                    { return l.second < p.second; });
                        // получить все максимумы
                        auto matchingMax = ne |
                            views::filter([&max](auto&& v)
                                    { return v.second == max->second; });
                        // получить количество максимумов
                        auto szMax = std::distance(matchingMax.begin(),
                                                   matchingMax.end());
                        // получить случайный максимум из равных
                        std::random_device r;
                        std::default_random_engine e1(r());
```

```
                        std::uniform_int_distribution<int> uniform_dist(0, szMax - 1);
                        int mean = uniform_dist(e1);
                        auto resMax = matchingMax.begin();
                        std::advance(resMax, mean);
                        // получить игрока из максимума
                        auto player = resMax->first;

                        // поставить существо (даже если оно там уже есть)
                        aside_.emplace_back(player, true, x, y);
                    }
                } else if (isAlive) {
                    // удалить существо (даже если его нет в клетке)
                    aside_.emplace_back(nullptr, false, x, y);
                }
            }
        }
    }
}

void GameModel::applyNClearAside_() {
    for (auto [player, set, x, y] : aside_) {
        if (set) {
            area_->setCreatureInCell(x, y, player);
        } else {
            area_->removeCreatureInCell(x, y);
        }
    }
    aside_.clear();
}

void GameModel::restartModel_() {
    area_->clear();
}

void GameModel::fireWinnerDeterminate_() {
    int evt = static_cast<int>(
        game_event::event_t::WINNER_DETERMINATE);
    notify(evt);
}

void GameModel::fireThereWasDraw_() {
    int evt = static_cast<int>(
        game_event::event_t::DRAW_DETERMINATE);
    notify(evt);
}

void GameModel::firePlayerBetsCreatures_() {
    int evt = static_cast<int>(
        game_event::event_t::PLAYER_BETS_CREATURES);
    notify(evt);
}

void GameModel::fireGameModelCalculatedEr_() {
    int evt = static_cast<int>(
        game_event::event_t::GAME_MODEL_CALCULATED_ER);
    notify(evt);
}

void GameModel::fireUserInputRequired() {
    int evt = static_cast<int>(
        game_event::event_t::USER_INPUT_REQUIRED);
    notify(evt);
}
```

```
class GameFieldWithFigure :
    public IGameField,
    public std::enable_shared_from_this<GameFieldWithFigure>,
    public subject::ISubject
{
private:
    using ISubject = subject::ISubject;
    using ICreatureFactory = factory::ICreatureFactory;
    using IFigure = figure::IFigure;

public:
    GameFieldWithFigure(
        int width, int height,
        std::unique_ptr<factory::ICreatureFactory> creatFactory,
        std::unique_ptr<factory::ICellFactory> cellFactory,
        std::unique_ptr<IFigure> figure);

public:
    const creature::ICreature& getCreatureByCell(int xidx, int yidx) const override;
    void setCreatureInCell(int xidx, int yidx,
            std::shared_ptr<player::Player> player) override;
    void removeCreatureInCell(int xidx, int yidx) override;
    bool hasCreatureInCell(int xidx, int yidx) const override;
    std::map<const std::shared_ptr<player::Player>, int>
        countCellNeighborsCreatures(int xidx, int yidx) const override;
    void clear() override;
    std::pair<int, int> lastAffectedCell() const noexcept override;
    int width() const noexcept override;
    int height() const noexcept override;

public:
    bool isExcludedCell(int xidx, int yidx) const;

public:
    void attach(std::shared_ptr<observer::IObserver> obs, int event_t) override;
    void detach(std::weak_ptr<observer::IObserver> obs, int event_t) override;

private:
    void notify(int event_t) override;

private:
    void verifyThenThrowCellPos_(int xidx, int yidx) const;
    void fireFieldClear_();
    void fireCreatureSet_();
    void fireCreatureRemove_();
    void initField_(int width, int height);
    void initCells_();

private:
    std::vector<std::vector<std::unique_ptr<cell::ICell>>> field_;
    std::pair<int, int> lastAffectedCell_ = {-1, -1};
    std::unique_ptr<factory::ICreatureFactory> creatFactory_;
    std::unique_ptr<factory::ICellFactory> cellFactory_;
    std::unique_ptr<IFigure> figure_;

    static constexpr std::array<const std::pair<int, int>, 8> neighborsPos_ {{
        {-1,  -1}, {0, -1}, {1, -1},
        {-1,   0},          {1,  0},
        {-1,   1}, {0,  1}, {1,  1}
    }};

};
```

```
GameFieldWithFigure::GameFieldWithFigure(
        int width, int height,
        std::unique_ptr<factory::ICreatureFactory> creatFactory,
        std::unique_ptr<factory::ICellFactory> cellFactory,
        std::unique_ptr<IFigure> figure) :
    creatFactory_(std::move(creatFactory))
    , cellFactory_(std::move(cellFactory))
    , figure_(std::move(figure))

{
    initField_(width, height);
    initCells_();
}

const creature::ICreature&
GameFieldWithFigure::getCreatureByCell(int xidx, int yidx) const {
    verifyThenThrowCellPos_(xidx, yidx);
    return field_.at(yidx).at(xidx)->creature();
}

void GameFieldWithFigure::setCreatureInCell(int xidx, int yidx,
    std::shared_ptr<player::Player> player)
{
    verifyThenThrowCellPos_(xidx, yidx);
    auto creat = creatFactory_->createCreature(player);
    field_.at(yidx).at(xidx)->setCreature(std::move(creat));
    lastAffectedCell_ = { xidx, yidx };
    fireCreatureSet_();
}

void GameFieldWithFigure::removeCreatureInCell(
    int xidx, int yidx)
{
    verifyThenThrowCellPos_(xidx, yidx);
    field_.at(yidx).at(xidx)->removeCreature();
    lastAffectedCell_ = { xidx, yidx };
    fireCreatureRemove_();
}

bool GameFieldWithFigure::hasCreatureInCell(
    int xidx, int yidx) const
{
    verifyThenThrowCellPos_(xidx, yidx);
    return field_.at(yidx).at(xidx)->hasCreature();
}
std::map<const std::shared_ptr<player::Player>, int>
GameFieldWithFigure::countCellNeighborsCreatures(int xidx, int yidx) const
{
    verifyThenThrowCellPos_(xidx, yidx);
    return field_.at(yidx).at(xidx)->countNeighborsCreatures();
}

void GameFieldWithFigure::clear() {
    int w = width();
    int h = height();
    field_.clear();
    initField_(w, h);
    initCells_();
    fireFieldClear_();
}

std::pair<int, int>
GameFieldWithFigure::lastAffectedCell() const noexcept
```

```
{ return lastAffectedCell_; }

int GameFieldWithFigure::width() const noexcept
{ return field_.back().size(); }

int GameFieldWithFigure::height() const noexcept
{ return field_.size(); }

bool GameFieldWithFigure::isExcludedCell(
    int xidx, int yidx) const
{ return !figure_->isPointInFigure(xidx, yidx); }

void GameFieldWithFigure::attach(
    std::shared_ptr<observer::IObserver> obs, int event_t)
{ ISubject::attach(obs, event_t); }

void GameFieldWithFigure::detach(
    std::weak_ptr<observer::IObserver> obs, int event_t)
{ ISubject::detach(obs, event_t); }

void GameFieldWithFigure::notify(int event_t) {
    ISubject::notify(event_t);
}

void GameFieldWithFigure::verifyThenThrowCellPos_(
    int xidx, int yidx) const
{
    if (isExcludedCell(xidx, yidx)) {
        throw std::logic_error("Accessing a forbidden cell.");
    }
}

void GameFieldWithFigure::fireFieldClear_() {
    int evt = static_cast<int>(
        game_event::event_t::FIELD_CLEAR);
    notify(evt);
}

void GameFieldWithFigure::fireCreatureSet_() {
    int evt = static_cast<int>(
        game_event::event_t::CREATURE_SET_IN_FIELD);
    notify(evt);
}

void GameFieldWithFigure::fireCreatureRemove_() {
    int evt = static_cast<int>(
            game_event::event_t::CREATURE_REMOVE_IN_FIELD);
    notify(evt);
}

void GameFieldWithFigure::initField_(int width, int height) {
    field_ = decltype(field_)();
    for (int i = 0; i < height; ++i) {
        std::vector<std::unique_ptr<cell::ICell>> r;
        for (int j = 0; j < width; ++j) {
            r.emplace_back(cellFactory_->createCell());
        }
        field_.emplace_back(std::move(r));
    }
}

void GameFieldWithFigure::initCells_() {
    for (int i = 0; i < height(); ++i) {
```

```
        for (int j = 0; j < width(); ++j) {
            auto&& cell = field_[i][j];
            for (auto [x, y] : neighborsPos_) {
                y += i;
                x += j;
                if (!isExcludedCell(x, y) &&
                    y >= 0 && y < height() &&
                    x >= 0 && x < width())
                {
                    auto&& ne = field_[y][x];
                    cell->addNeighbors(ne.get());
                }
            }
        }
    }
}
```

3.8 Реализация ключевых тестовых случаев

```
TEST(GameModelTest, SingleCreature) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2"),
    };
    auto figure =
        std::make_unique<figure::DummyFigure>();
    auto creatureFactory
        = std::make_unique<CreatureFactory>();
    auto cellFactory =
        std::make_unique<CellFactory>();
    auto actualField
        = std::make_shared<GameFieldWithFigure>(
                3, 3,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    actualField->setCreatureInCell(1, 1, player[0]);

    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {2, 2};
    auto area = std::make_unique<
                GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();

    auto f =
        std::make_unique<
            factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy =
        std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();

    figure =
        std::make_unique<figure::DummyFigure>();
    creatureFactory
        = std::make_unique<CreatureFactory>();
    cellFactory =
        std::make_unique<CellFactory>();
    auto expectField
        = std::make_shared<GameFieldWithFigure>(
                3, 3,
                std::move(creatureFactory),
```

```
                std::move(cellFactory),
                std::move(figure)
            );

    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, SingleNeighbor) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;


    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2"),
    };
    auto figure =
        std::make_unique<figure::DummyFigure>();
    auto creatureFactory
        = std::make_unique<CreatureFactory>();
    auto cellFactory =
        std::make_unique<CellFactory>();
    auto actualField
        = std::make_shared<GameFieldWithFigure>(
                3, 3,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    actualField->setCreatureInCell(1, 1, player[0]);
    actualField->setCreatureInCell(0, 1, player[0]);

    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {2, 2};
    auto area = std::make_unique<
                GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();

    auto f =
        std::make_unique<
            factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy =
        std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();


    figure =
        std::make_unique<figure::DummyFigure>();
    creatureFactory
        = std::make_unique<CreatureFactory>();
    cellFactory =
```

```
        std::make_unique<CellFactory>();
    auto expectField
        = std::make_shared<GameFieldWithFigure>(
                3, 3,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, TwoNeighbors) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2"),
    };
    auto figure =
        std::make_unique<figure::DummyFigure>();
    auto creatureFactory
        = std::make_unique<CreatureFactory>();
    auto cellFactory =
        std::make_unique<CellFactory>();
    auto actualField
        = std::make_shared<GameFieldWithFigure>(
                3, 3,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    actualField->setCreatureInCell(1, 1, player[0]);
    actualField->setCreatureInCell(2, 1, player[0]);
    actualField->setCreatureInCell(1, 2, player[0]);

    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {2, 2};
    auto area = std::make_unique<
                GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();

    auto f =
        std::make_unique<
            factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy =
        std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();
```

```
    figure =
        std::make_unique<figure::DummyFigure>();
    creatureFactory
        = std::make_unique<CreatureFactory>();
    cellFactory =
        std::make_unique<CellFactory>();
    auto expectField
        = std::make_shared<GameFieldWithFigure>(
                3, 3,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    expectField->setCreatureInCell(1, 1, player[0]);
    expectField->setCreatureInCell(2, 1, player[0]);
    expectField->setCreatureInCell(1, 2, player[0]);
    expectField->setCreatureInCell(2, 2, player[0]);


    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, SingleCellField) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;


    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2"),
    };
    auto figure =
        std::make_unique<figure::DummyFigure>();
    auto creatureFactory
        = std::make_unique<CreatureFactory>();
    auto cellFactory =
        std::make_unique<CellFactory>();
    auto actualField
        = std::make_shared<GameFieldWithFigure>(
                1, 1,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    actualField->setCreatureInCell(0, 0, player[0]);

    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {0, 0};
    auto area = std::make_unique<
                GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();
```

```
    auto f =
        std::make_unique<
            factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy =
        std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();


    figure =
        std::make_unique<figure::DummyFigure>();
    creatureFactory
        = std::make_unique<CreatureFactory>();
    cellFactory =
        std::make_unique<CellFactory>();
    auto expectField
        = std::make_shared<GameFieldWithFigure>(
                1, 1,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, Stable) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2"),
    };
    auto figure =
        std::make_unique<figure::DummyFigure>();
    auto creatureFactory
        = std::make_unique<CreatureFactory>();
    auto cellFactory =
        std::make_unique<CellFactory>();
    auto actualField
        = std::make_shared<GameFieldWithFigure>(
                4, 4,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    actualField->setCreatureInCell(1, 1, player[0]);
    actualField->setCreatureInCell(2, 1, player[0]);
    actualField->setCreatureInCell(1, 2, player[0]);
    actualField->setCreatureInCell(2, 2, player[0]);
```

```
    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {3, 3};
    auto area = std::make_unique<
                GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();

    auto f =
        std::make_unique<
            factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy =
        std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();


    figure =
        std::make_unique<figure::DummyFigure>();
    creatureFactory
        = std::make_unique<CreatureFactory>();
    cellFactory =
        std::make_unique<CellFactory>();
    auto expectField
        = std::make_shared<GameFieldWithFigure>(
                4, 4,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    expectField->setCreatureInCell(1, 1, player[0]);
    expectField->setCreatureInCell(2, 1, player[0]);
    expectField->setCreatureInCell(1, 2, player[0]);
    expectField->setCreatureInCell(2, 2, player[0]);

    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, Glider) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2"),
    };
    auto figure =
        std::make_unique<figure::DummyFigure>();
    auto creatureFactory
        = std::make_unique<CreatureFactory>();
    auto cellFactory =
        std::make_unique<CellFactory>();
```

```
    auto actualField
        = std::make_shared<GameFieldWithFigure>(
                5, 5,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    actualField->setCreatureInCell(0, 2, player[0]);
    actualField->setCreatureInCell(1, 3, player[0]);
    actualField->setCreatureInCell(2, 1, player[0]);
    actualField->setCreatureInCell(2, 2, player[0]);
    actualField->setCreatureInCell(2, 3, player[0]);

    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {4, 4};
    auto area = std::make_unique<
                GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();

    auto f =
        std::make_unique<
            factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy =
        std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();


    figure =
        std::make_unique<figure::DummyFigure>();
    creatureFactory
        = std::make_unique<CreatureFactory>();
    cellFactory =
        std::make_unique<CellFactory>();
    auto expectField
        = std::make_shared<GameFieldWithFigure>(
                5, 5,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    expectField->setCreatureInCell(1, 1, player[0]);
    expectField->setCreatureInCell(2, 2, player[0]);
    expectField->setCreatureInCell(3, 2, player[0]);
    expectField->setCreatureInCell(1, 3, player[0]);
    expectField->setCreatureInCell(2, 3, player[0]);

    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, Oscillator) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
```

```
using namespace game_model;
using namespace creature_strategy;

std::vector<std::shared_ptr<player::Player>> player {
    std::make_shared<player::Player>(1, "player1"),
    std::make_shared<player::Player>(2, "player2"),
};
auto figure =
    std::make_unique<figure::DummyFigure>();
auto creatureFactory
    = std::make_unique<CreatureFactory>();
auto cellFactory =
    std::make_unique<CellFactory>();
auto actualField
    = std::make_shared<GameFieldWithFigure>(
            3, 3,
            std::move(creatureFactory),
            std::move(cellFactory),
            std::move(figure)
        );

actualField->setCreatureInCell(0, 1, player[0]);
actualField->setCreatureInCell(1, 1, player[0]);
actualField->setCreatureInCell(2, 1, player[0]);


std::pair<int, int> lu {0, 0};
std::pair<int, int> rl {2, 2};
auto area = std::make_unique<
            GameFieldWithFigureArea>(actualField, lu, rl);
area->unlock();

auto f =
    std::make_unique<
        factory::GameFieldWithFigureAreaCurryFactory>(actualField);
auto creatStrategy =
    std::make_unique<ConwayCreatureStrategy>();
GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
model.computeEr_();


figure =
    std::make_unique<figure::DummyFigure>();
creatureFactory
    = std::make_unique<CreatureFactory>();
cellFactory =
    std::make_unique<CellFactory>();
auto expectField
    = std::make_shared<GameFieldWithFigure>(
            3, 3,
            std::move(creatureFactory),
            std::move(cellFactory),
            std::move(figure)
        );

expectField->setCreatureInCell(1, 0, player[0]);
```

```
    expectField->setCreatureInCell(1, 1, player[0]);
    expectField->setCreatureInCell(1, 2, player[0]);

    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, Blinker2) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2")
    };

    // Initial setup (vertical blinker)
    auto figure = std::make_unique<figure::DummyFigure>();
    auto creatureFactory = std::make_unique<CreatureFactory>();
    auto cellFactory = std::make_unique<CellFactory>();
    auto actualField = std::make_shared<GameFieldWithFigure>(
        5, 5,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    actualField->setCreatureInCell(2, 1, player[0]);
    actualField->setCreatureInCell(2, 2, player[0]);
    actualField->setCreatureInCell(2, 3, player[0]);

    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {4, 4};
    auto area = std::make_unique<GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();

    auto f = std::make_unique<factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy = std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();

    // Expected result (horizontal blinker)
    figure = std::make_unique<figure::DummyFigure>();
    creatureFactory = std::make_unique<CreatureFactory>();
    cellFactory = std::make_unique<CellFactory>();
    auto expectField = std::make_shared<GameFieldWithFigure>(
        5, 5,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    expectField->setCreatureInCell(1, 2, player[0]);
    expectField->setCreatureInCell(2, 2, player[0]);
```

```
    expectField->setCreatureInCell(3, 2, player[0]);

    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, TwoPlayerCreaturesSpawnNewCreature) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2")
    };

    // Initial setup (vertical blinker)
    auto figure = std::make_unique<figure::DummyFigure>();
    auto creatureFactory = std::make_unique<CreatureFactory>();
    auto cellFactory = std::make_unique<CellFactory>();
    auto actualField = std::make_shared<GameFieldWithFigure>(
        5, 5,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    actualField->setCreatureInCell(0, 0, player[0]);
    actualField->setCreatureInCell(1, 0, player[1]);
    actualField->setCreatureInCell(0, 1, player[1]);

    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {4, 4};
    auto area = std::make_unique<GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();

    auto f = std::make_unique<factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy = std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();

    // Expected result (horizontal blinker)
    figure = std::make_unique<figure::DummyFigure>();
    creatureFactory = std::make_unique<CreatureFactory>();
    cellFactory = std::make_unique<CellFactory>();
    auto expectField = std::make_shared<GameFieldWithFigure>(
        5, 5,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    expectField->setCreatureInCell(0, 0, player[0]);
    expectField->setCreatureInCell(1, 0, player[1]);
    expectField->setCreatureInCell(0, 1, player[1]);
```

```
    expectField->setCreatureInCell(1, 1, player[1]);

    ASSERT_TRUE(eqFields(actualField, expectField));
}

TEST(GameModelTest, TwoPlayerCreaturesDisappear) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    std::vector<std::shared_ptr<player::Player>> player {
        std::make_shared<player::Player>(1, "player1"),
        std::make_shared<player::Player>(2, "player2")
    };

    // Initial setup (vertical blinker)
    auto figure = std::make_unique<figure::DummyFigure>();
    auto creatureFactory = std::make_unique<CreatureFactory>();
    auto cellFactory = std::make_unique<CellFactory>();
    auto actualField = std::make_shared<GameFieldWithFigure>(
        5, 5,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    actualField->setCreatureInCell(0, 0, player[0]);
    actualField->setCreatureInCell(1, 0, player[1]);

    std::pair<int, int> lu {0, 0};
    std::pair<int, int> rl {4, 4};
    auto area = std::make_unique<GameFieldWithFigureArea>(actualField, lu, rl);
    area->unlock();

    auto f = std::make_unique<factory::GameFieldWithFigureAreaCurryFactory>(actualField);
    auto creatStrategy = std::make_unique<ConwayCreatureStrategy>();
    GameModel model(0, 0, 0, std::move(area), std::move(f), player, std::move(creatStrategy));
    model.computeEr_();

    // Expected result (horizontal blinker)
    figure = std::make_unique<figure::DummyFigure>();
    creatureFactory = std::make_unique<CreatureFactory>();
    cellFactory = std::make_unique<CellFactory>();
    auto expectField = std::make_shared<GameFieldWithFigure>(
        5, 5,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    ASSERT_TRUE(eqFields(actualField, expectField));
}
```

4 Вторая итерация разработки

4.1 Функциональные требования (сценарии)

1) Сценарий «Игра завершается победой одного из Игроков».

1. Пользователем выбирается количество игроков.
2. **По указанию** Пользователя**,** Игра стартует.
3. **По указанию** Игры, Поле создаёт ячейки.
4. **По указанию** Игры, Пул Игроков создает Сущности игроков и присваивает им соответствующие им уникальные типы Существ.
5. **По запросу** Пула Игроков, Игра присваивает каждому Игроку уникальную область поля.
6. **Делать {**
   6.1. **Исполнить** дочерний сценарий «Подготовка поля игроком».
   6.2. **Исполнить** дочерний сценарий «Вычисление модели».
   6.3. **} Пока** на поле есть Существа **И** на поле есть Существа двух типов.
7. Игра считает победителем Игрока, чьи Существа остались на поле единственными.
8. **Сценарий завершается**.

1.1) Альтернативный сценарий «Досрочное завершение игры пользователем». Сценарий **выполняется в любой точке** главного сценария.

1. **По указанию пользователя,** программа завершается без определения победителя.
2. **Сценарий завершается.**

1.2) Альтернативный сценарий «Завершение Игры ничьей». Сценарий **выполняется после главного цикла** главного сценария.

1. Если на Поле не осталось Существ, то игра завершается ничьей.

2. Сценарий завершается.

1.3) Дочерний сценарий «Подготовка полем игроком»

1. **Делать {**

1. **Делать** Пока не выберутся все Игроки из Пула {
    1. Игра выбирает идущего по порядку Игрока и разрешает ему порождение и уничтожение его Существ в его области поля (*пользователь при нажатии на соответствующую область поля будет размещать/удалять соответствующий текущему игроку тип Существ*).
    2. Игрок создает Существо нужного типа в Разрешенной Клетке, **Если** нужно.
    3. **По запросу** Игрока, Область Поля размещает или уничтожает Существо на нужной позиции.
    4. Игра ждет пока Игрок не породит К Существ.

}

1.4) Дочерний сценарий «Вычисление модели»

9. Делать Пока не пройдет Т эпох {

9.1. **По запросу** Игры, Поле **сообщает** Игре позиции всех Существ.

9.2. **Поле** проходиться по всем Существам и соседним позициям

9.2.1.1. **Либо**

9.3. Выбирает уничтожить Существо, **Если** соседей меньше 2 или больше 3.

9.3.1. **Либо**

9.4. Выбирает породить Существо определенного типа в соответствующий позиции, **Если** соседей больше или 3 данного.

9.5. **По запросу** Игры, Поле размещает или уничтожает Существо на соответствующей позиции.

}

4.2 Словарь предметной области

Игра - знает о Поле и Пуле Игроков. Игра инициирует создание. Игра определяет очередного активного Игрока, набор существ на Поле и окончание игры (и победителя).

знает	● о Поле ● о Пуле Игроков ● о Существе
умеет	● инициировать создание Поля. ● инициализировать создание Пула Игроков ● определять очередного Игрока, который может обрабатывать запросы пользователя ● определять окончание игры (и победителя)
предназначение	● Организация общего игрового цикла

Поле - область заданной формы, состоящая из ячеек. Знает о Существах, находящихся на Поле.

знает	● свои размеры и форму ● Ячейки ● о Существах
умеет	● создавать себя из Ячеек ● предоставлять доступ к Ячейкам ● выдавать Существ
предназначение	● Контейнер Ячеек и Сущностей, которые располагаются внутри Ячеек

Ячейка - квадратная область Поля. Знает о четырёх соседних Ячейках и граничащих с ней Стенах. На ней может располагаться сущность.

знает	● соседние Ячейки ● о наличии в себе Существа
умеет	● устанавливать соседство с другой Ячейкой ● предоставляет доступ к размещенному в себе Существу
предназначение	● Контейнер для Существа

Существо - сущность необходимая для вычисления текущего состояния игры. Знает об игроке-хозяине и своем типе.

знает	● своего Игрока ● свой Тип
умеет	● Возвращать свой тип и игрока
предназначение	● Маркировать Игрока на поле

Пул Игроков - контейнер игроков. Умеет создавать игроков, инициализируя их типом Существа и Областью Поля, и выдавать очередного Игрока.

знает	● о Игроках ● о Существах ● о Областях Поля
умеет	● создавать Игроков ● выдавать очередного Игрока
предназначение	● Хранить игроков

Игрок - знает об Области Поля и Существе. Умеет создавать свой тип существ и размещать их на Поле посредством Области Поля.

знает	● свою Область поля ● свое Существо
умеет	● создавать Существо ● размещать его на Поле
предназначение	● абстракция игрока для пользователя и Игры

Область поля - знает о своей форме, размерах и Поле. Необходима для ограничения прав Игрока на размещение Существ до определенной области.

знает	● свои форму и размер ● Поле
умеет	● размещать существо в определенном пространстве поля
предназначение	● ограничение прав Игрока на размещение Сущностей

4.3 Структура программы на уровне классов

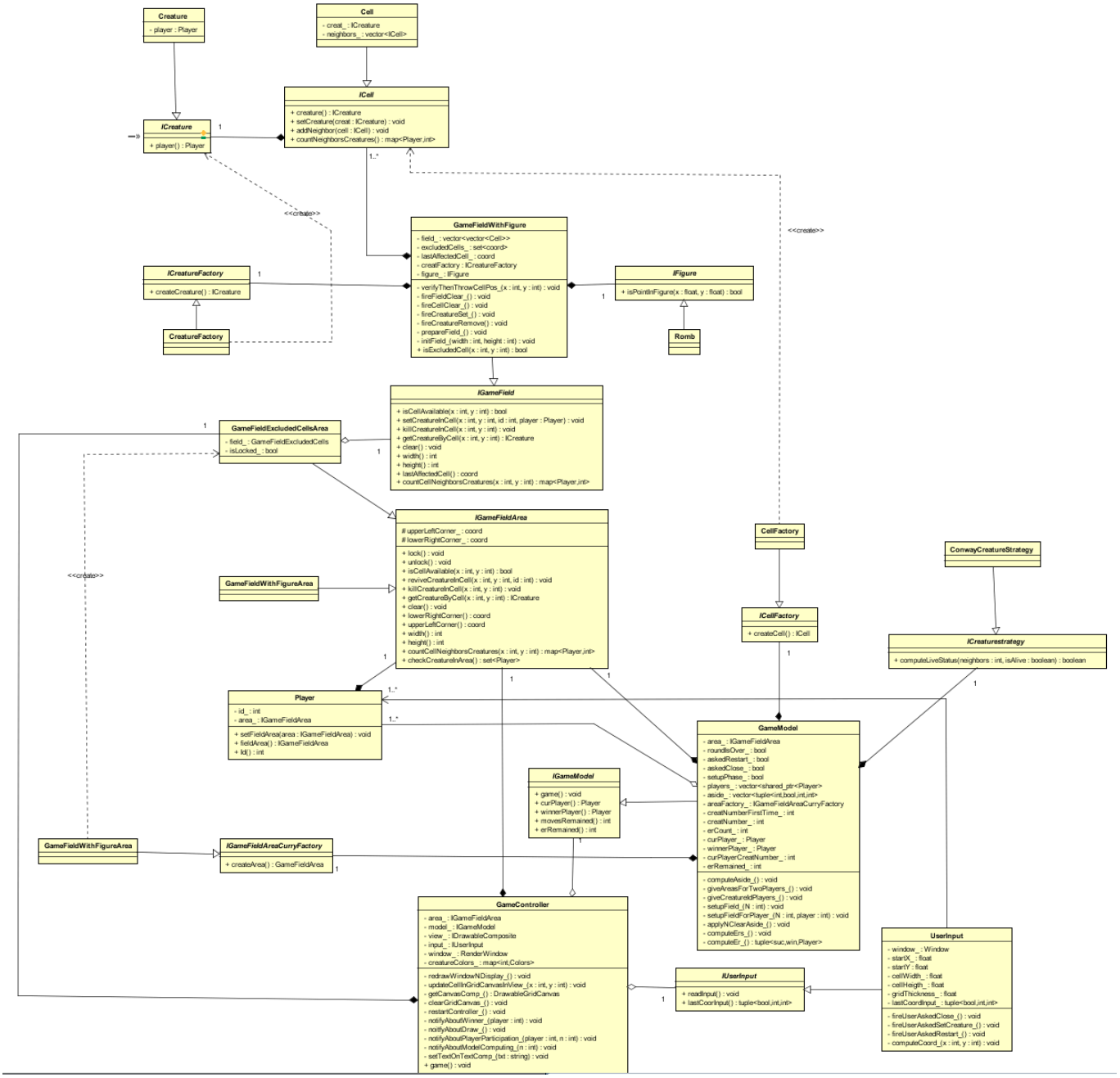

Рисунок 7 - Структура программы на уровне классов.

4.4 Типовые процессы в программе

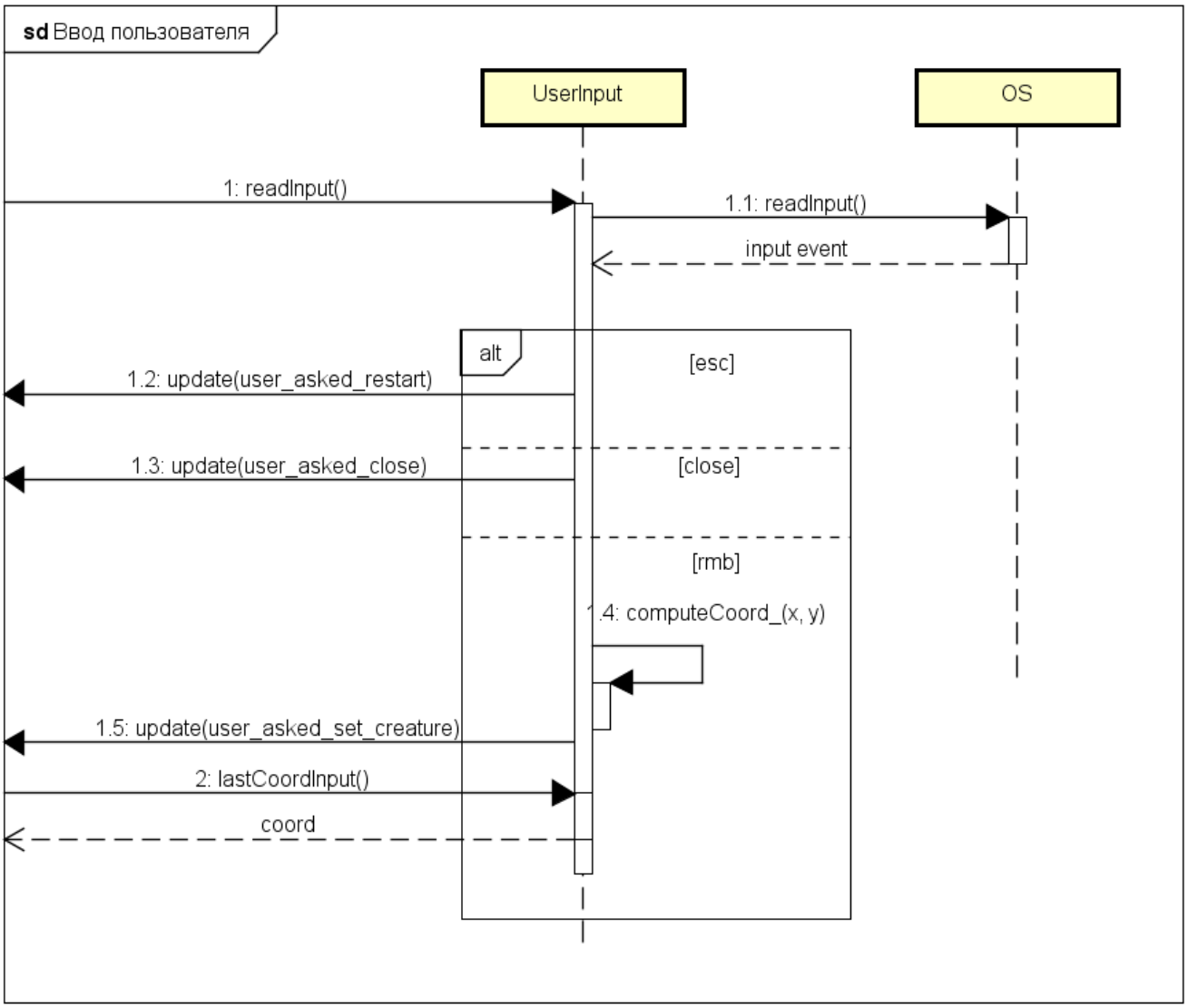

Рисунок 8 - Ввод пользователя.

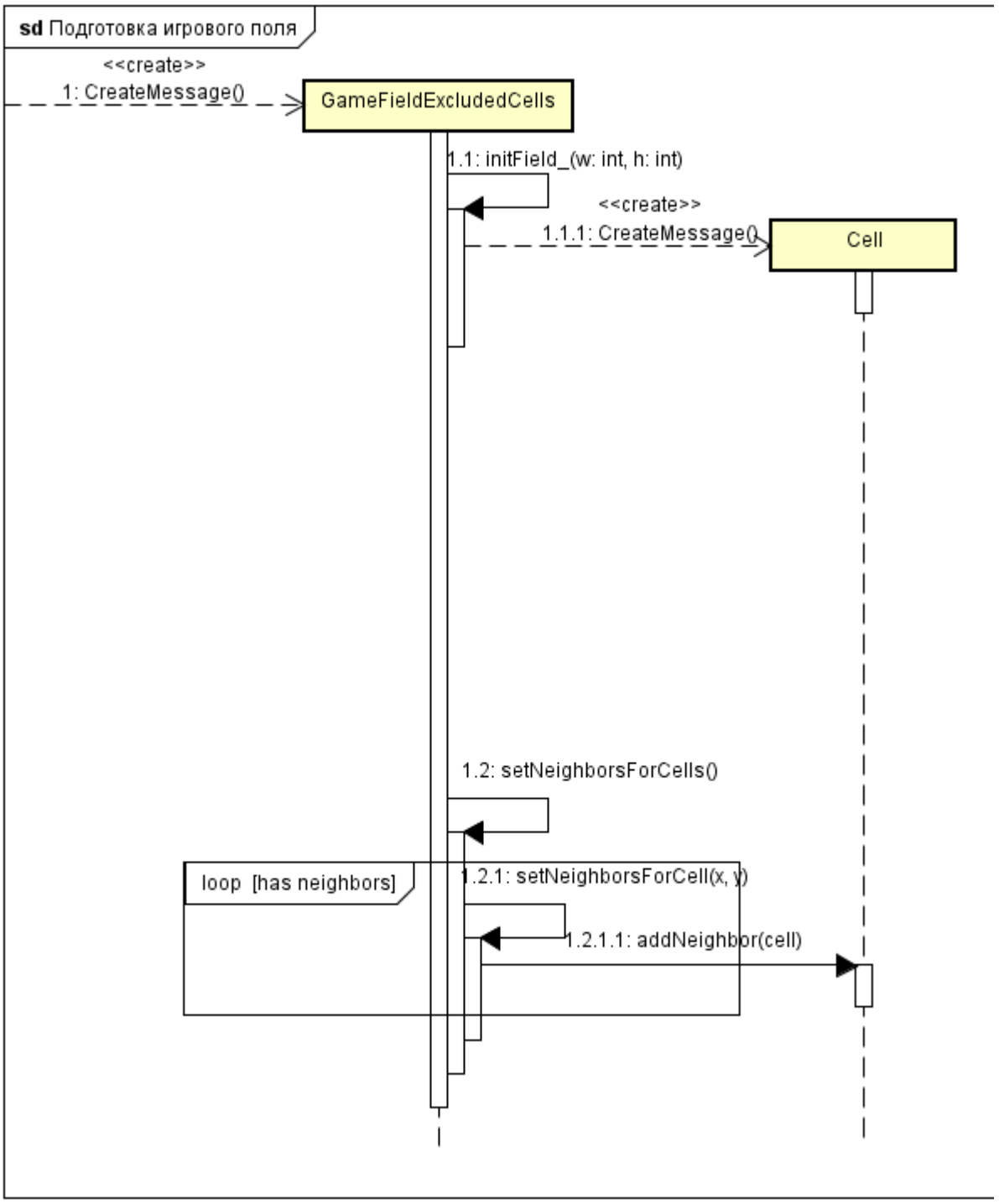

Рисунок 9 - Подготовка игрового поля.

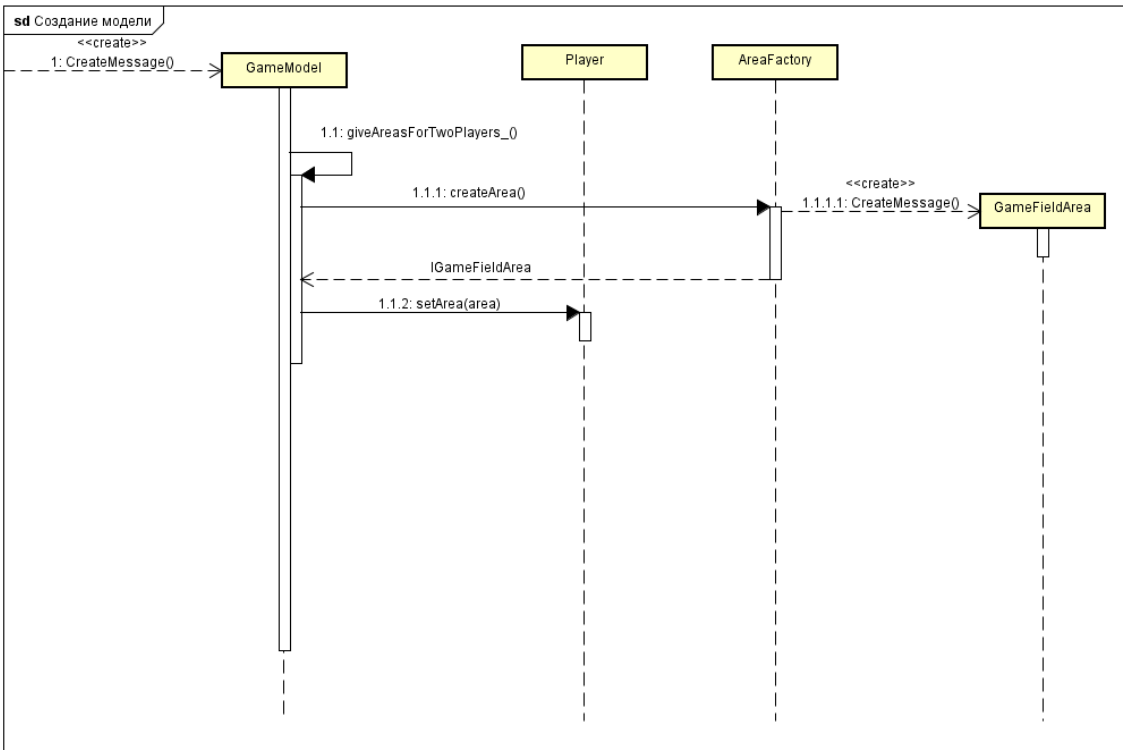

Рисунок 10 - Создание модели.

4.5 Человеко-машинное взаимодействие

1. **Запуск приложения**

   1.1. Пользователь запускает исполняемый файл `Fun_Of_The_Gods.exe` на одном устройстве.

   1.2. Открывается одно окно, содержащее:
   - графическое поле (в верхней части),
   - текстовое поле сообщений от игры (в нижней части).

   1.3. Программа переходит в режим размещения существ игроком 0.

2. **Размещение существ игроками**

   2.1. **Игрок 0:**
   - Размещает существ на своей половине поля.
   - Управление осуществляется через **ПКМ (правая кнопка мыши)**:
     - ПКМ по свободной клетке своей половины поля — размещение существа (если остались в запасе).
     - ПКМ по уже размещённому своему существу — удаление существа, возвращение его в пул.
   - В текстовом поле отображается количество оставшихся существ.
   - Нажатие клавиши **Esc**:
     - Полностью очищает поле.
     - Возвращает управление игроку 0.
     - Перезапускает этап размещения.

3. 2.2. **Игрок 1:**
   - После завершения размещения игрока 0, управление переходит к игроку 1.
   - Размещение и удаление существ осуществляется по тем же правилам.
   - Также отображается количество оставшихся существ.
   - Доступен перезапуск с помощью Esc (см. п. 2.1).

4. **Моделирование эпох**

   3.1. После завершения размещения обоими игроками, управление полностью переходит к программе.

   3.2. Программа автоматически моделирует **T эпох**.

   3.3. В рамках каждой эпох происходит расчёт взаимодействия между существами на поле.

5. **Оценка результатов**

   4.1. По завершении T эпох программа анализирует состояние игрового поля:

- Если один из игроков одержал победу, в текстовом поле появляется сообщение:
  - «Победил игрок 0» или
  - «Победил игрок 1».
- Если достигнута ничья — выводится сообщение:
  - «Ничья».
- В случае победы или ничьей игра завершается.

6. **Дополнительное размещение существ**

5.1. Если победитель не определён и ничья не наступила, игра предоставляет игрокам возможность поочерёдно разместить **K дополнительных существ**.

5.2. **Игрок 0** размещает/удаляет существа на своей половине поля по тем же правилам (ПКМ, только свои клетки, отображение оставшегося количества, возможность отмены через Esc).

5.3. После него — **игрок 1** по аналогичной схеме.

7. **Повторение цикла**

6.1. После дополнительного размещения существ запускается очередной цикл из **T эпох**.

6.2. Производится повторная проверка условий победы или ничьей.

6.3. Этапы 5–6 повторяются до момента окончания игры (победа или ничья).

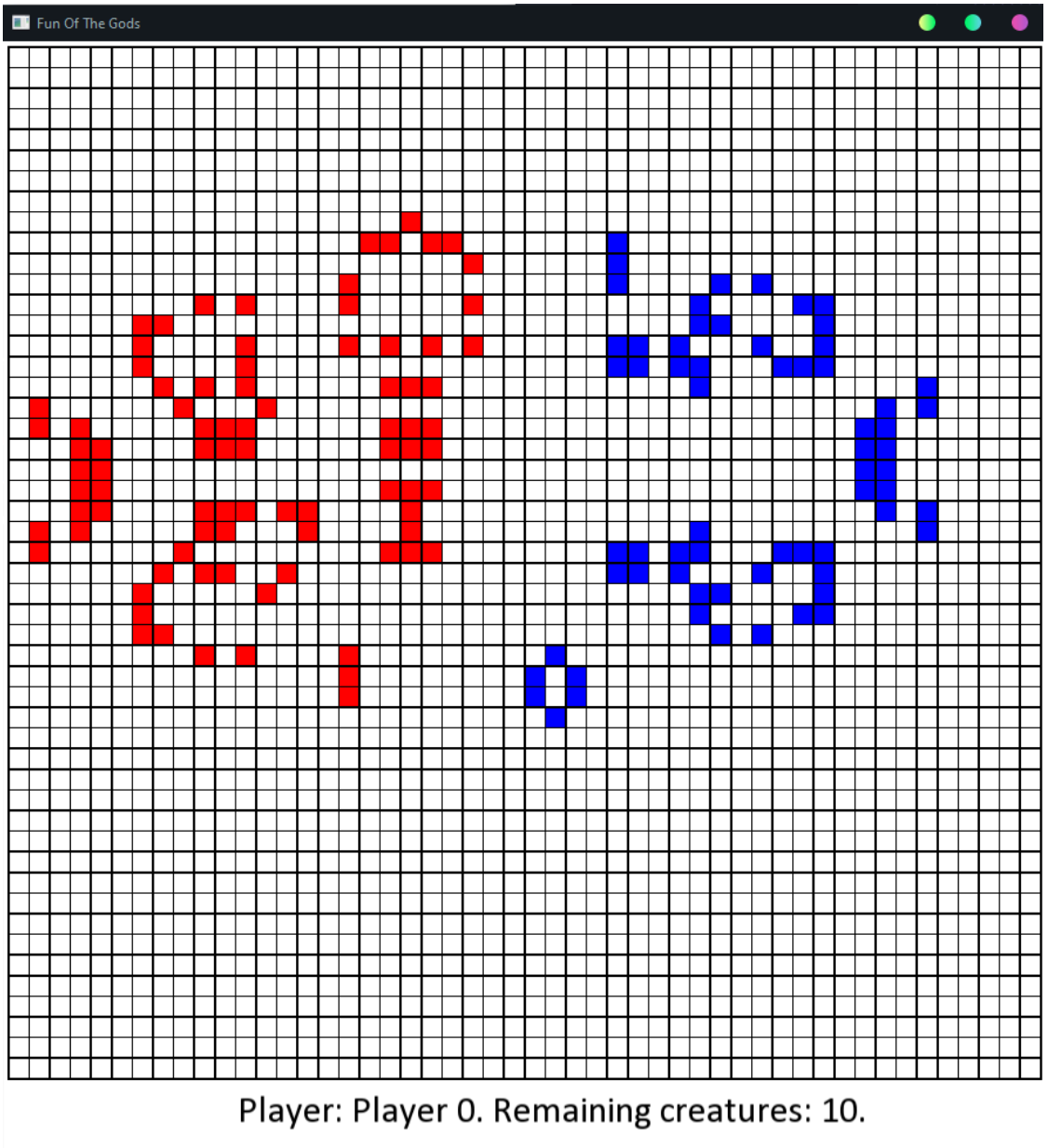

Рисунок 11 - Игровая сессия.

4.6 Реализация ключевых классов

```
class Cell : public ICell {
public:
    const creature::ICreature& creature() const override;
    creature::ICreature& creature() override;
    void setCreature(std::unique_ptr<creature::ICreature> creat) override;
    bool hasCreature() const override;
    void removeCreature() override;
    void addNeighbors(const ICell* ne) override;
    std::map<const std::shared_ptr<player::Player>, int>
        countNeighborsCreatures() const override;

private:
    std::unique_ptr<creature::ICreature> creat_;
    std::vector<const ICell*> neighbors_;
};

creature::ICreature& Cell::creature() {
    return *creat_;
}

const creature::ICreature& Cell::creature() const {
    return *creat_;
}

void Cell::setCreature(
    std::unique_ptr<creature::ICreature> creat) {
    creat_ = std::move(creat);
}

bool Cell::hasCreature() const {
    return static_cast<bool>(creat_);
}

void Cell::removeCreature() {
    creat_.release();
}



void Cell::addNeighbors(const ICell* ne) {
    neighbors_.push_back(ne);
}

std::map<const std::shared_ptr<player::Player>, int>
    Cell::countNeighborsCreatures() const
{
    std::map<const std::shared_ptr<player::Player>, int> res;
    for (auto c : neighbors_) {
        if (c->hasCreature()) {
            auto&& cr = c->creature();;
            auto it = res.find(cr.player());
            if (it != res.end()) {
                ++it->second;
            } else {
                res[cr.player()] = 1;
            }
        }
    }
    return res;
```

```
}

struct ICreature {
    virtual const std::shared_ptr<player::Player> player() const = 0;
    virtual ~ICreature() = default;
};

class Creature : public ICreature {
public:
    Creature(const std::shared_ptr<player::Player> player);

public:
    const std::shared_ptr<player::Player> player() const override;

public:
    bool operator<(const Creature& cr) const;

private:
    const std::shared_ptr<player::Player> player_;
};

Creature::Creature(
    const std::shared_ptr<player::Player> player) :
    player_(player)
{}

const std::shared_ptr<player::Player> Creature::player() const {
    return player_;
}

bool Creature::operator<(const Creature& cr) const {
    return player_->id() < cr.player_->id();
}
```

4.7 Реализация ключевых тестовых случаев

```
TEST(GameFieldSetupTest, SingleCellSize) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    auto player =
        std::make_shared<player::Player>(1, "player1");

    auto figure =
        std::make_unique<figure::DummyFigure>();
    auto creatureFactory
        = std::make_unique<CreatureFactory>();
    auto cellFactory =
        std::make_unique<CellFactory>();
    auto field
        = std::make_shared<GameFieldWithFigure>(
                1, 1,
                std::move(creatureFactory),
                std::move(cellFactory),
                std::move(figure)
            );

    ASSERT_EQ(field->height(), 1);
    ASSERT_EQ(field->width(), 1);
    ASSERT_FALSE(field->hasCreatureInCell(0, 0));

    field->setCreatureInCell(0, 0, player);
    ASSERT_TRUE(field->hasCreatureInCell(0, 0));
}

TEST(GameFieldSetupTest, MultiCellField) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    auto player1 = std::make_shared<player::Player>(1, "player1");
    auto player2 = std::make_shared<player::Player>(2, "player2");

    auto figure = std::make_unique<figure::DummyFigure>();
    auto creatureFactory = std::make_unique<CreatureFactory>();
    auto cellFactory = std::make_unique<CellFactory>();
    auto field = std::make_shared<GameFieldWithFigure>(
        3, 3,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    ASSERT_EQ(field->height(), 3);
    ASSERT_EQ(field->width(), 3);

    // Проверяем пустые клетки
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
```

```
            ASSERT_FALSE(field->hasCreatureInCell(i, j));
        }
    }

    // Заполняем две клетки
    field->setCreatureInCell(0, 0, player1);
    field->setCreatureInCell(2, 2, player2);

    ASSERT_TRUE(field->hasCreatureInCell(0, 0));
    ASSERT_TRUE(field->hasCreatureInCell(2, 2));
    ASSERT_FALSE(field->hasCreatureInCell(1, 1));
}

TEST(GameFieldSetupTest, CreatureOverwrite) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    auto player1 = std::make_shared<player::Player>(1, "player1");
    auto player2 = std::make_shared<player::Player>(2, "player2");

    auto figure = std::make_unique<figure::DummyFigure>();
    auto creatureFactory = std::make_unique<CreatureFactory>();
    auto cellFactory = std::make_unique<CellFactory>();
    auto field = std::make_shared<GameFieldWithFigure>(
        2, 2,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    field->setCreatureInCell(0, 0, player1);
    ASSERT_TRUE(field->hasCreatureInCell(0, 0));

    // Перезаписываем существо
    field->setCreatureInCell(0, 0, player2);
    ASSERT_TRUE(field->hasCreatureInCell(0, 0));
}

TEST(GameFieldSetupTest, InvalidCellAccess) {
    using namespace game_field;
    using namespace game_field_area;
    using namespace factory;
    using namespace game_model;
    using namespace creature_strategy;

    auto player = std::make_shared<player::Player>(1, "player1");
    auto figure = std::make_unique<figure::DummyFigure>();
    auto creatureFactory = std::make_unique<CreatureFactory>();
    auto cellFactory = std::make_unique<CellFactory>();
    auto field = std::make_shared<GameFieldWithFigure>(
        2, 2,
        std::move(creatureFactory),
        std::move(cellFactory),
        std::move(figure)
    );

    // Проверяем выход за границы
    EXPECT_THROW(field->hasCreatureInCell(-1, 0), std::out_of_range);
    EXPECT_THROW(field->hasCreatureInCell(0, -1), std::out_of_range);
    EXPECT_THROW(field->hasCreatureInCell(2, 0), std::out_of_range);
```

```
    EXPECT_THROW(field->hasCreatureInCell(0, 2), std::out_of_range);

    EXPECT_THROW(field->setCreatureInCell(-1, 0, player), std::out_of_range);
    EXPECT_THROW(field->setCreatureInCell(0, -1, player), std::out_of_range);
    EXPECT_THROW(field->setCreatureInCell(2, 0, player), std::out_of_range);
    EXPECT_THROW(field->setCreatureInCell(0, 2, player), std::out_of_range);
}
```

5 Список использованной литературы и других источников

1. Логинова, Ф.С. Объектно-ориентированные методы программирования. [Электронный ресурс] : учеб. пособие — Электрон. дан. — СПб. : ИЭО СПбУТУиЭ, 2012. — 208 с. — Режим доступа: http://e.lanbook.com/book/64040

Перечень замечаний к работе