**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: **Полиморфизм**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3384 |  | Горский К.Д. |
| Преподаватель |  | Шестопалов Р.П. |

Санкт-Петербург

2024

## Цель работы.

Изучить полиморфизм в объектно ориентированном программировании, применить полученные знания на практике, реализовав систему способностей для игры «Морской бой» из предыдущей лабороторной работы.

## Задание.

**1.** Создать класс-интерфейс способности, которую игрок может применять. Через наследование создать 3 разные способности:

a. Двойной урон - следующая атак при попадании по кораблю нанесет

сразу 2 урона (уничтожит сегмент).

b. Сканер - позволяет проверить участок поля 2х2 клетки и узнать, есть ли там сегмент корабля. Клетки не меняют свой статус.

c. Обстрел - наносит 1 урон случайному сегменту случайного корабля. Клетки не меняют свой статус.

**2.** Создать класс менеджер-способностей. Который хранит очередь способностей, изначально игроку доступно по 1 способности в случайном порядке. Реализовать метод применения способности.

**3.** Реализовать функционал получения одной случайной способности при уничтожении вражеского корабля.

**4.** Реализуйте набор классов-исключений и их обработку для следующих ситуаций (можно добавить собственные):

a. Попытка применить способность, когда их нет

b. Размещение корабля вплотную или на пересечении с другим кораблем

c. Атака за границы поля

Примечания:

* Интерфейс события должен быть унифицирован, чтобы их можно было единообразно использовать через интерфейс
* Не должно быть явных проверок на тип данных

## Основные теоретические положения.

Полиморфизм — это одно из ключевых понятий в объектно-ориентированном программировании (ООП). Он позволяет объектам разных классов обрабатывать данные с одинаковым интерфейсом, что повышает гибкость и расширяемость кода. Полиморфизм в С++ часто реализуется с помощью переопределения методов классов. Класс-наследник может переопределить (или определить, если у класса-родителя эти методы не определены впринципе) методы класса-родителя, тем самым задав задав другую логику, обрабатывающую точно такое же сообщение. Таким образом, чтобы сделать несколько классов, обрабатывающих одинаковые запросы по-разному, они все должны наследоваться от общего класса и переопределять его методы.

Полиморфизм бывает статическим и динамическим. При использовании статического полиморфизма определяется, как именно обработать запрос, еще во время компиляции. Это не всегда позволяет достичь желаемого результата, поэтому существует динамический полиморфизм: программа во время исполнения знает класс, экземпляром которого является объект и соответственно может выбрать корректный метод для обработки запроса. Это достигается с помощью виртуальных таблиц, каждый такой объект хранит указатель на виртуальную таблицу, которая содержит адреса переопределённых у класса, экземпляром которого является этот объект, методов. Эта таблица может также хранить прочую информацию, такую как название класса и его ID.

## Выполнение работы.

Система сообщений: для взаимодействия двух объектов, которые не должны знать друг о друге была создана система сообщений. В дальнейшем я планирую от нее избавиться или хотя бы упростить. К условию данной лабороторной она никак не относится, но для понимания происходящего необходимо ее прояснить. Существует абстрактный виртуальный класс Message. Он представляет собой сообщение. От него наследуются различные сообщения, например LogMessage — сообщение для вывода текста на экран (в GUI и CLI версиях вывод производится по разному, совершенно разными объектами). Или например, сообщение RenderField можно использовать, чтобы изменить размер курсора. Способность Scanner посылает это сообщение (чтобы изменить размер курсора на 2х2) вверх по иерархии классов. Класс игры, когда встретит это сообщение, пошлёт его отрисовщику игры, и тот уже выполнит обработку сообщения. MessageGenerator — абстрактный класс, способный генерировать сообщения, а MessageReciever — абстрактный класс, способный их принимать. MessageTransformer может одновременно вести себя и как MessageGenerator, и как MessageReciever. Его можно привести к нужному типу с помощью например static\_cast. Его задача — принять сообщение, обработать, и, если требуется, сгенерировать новые сообщения. MessagePipe — это практически MessageTransformer, но за небольшим исключением: если ему не удается обработать сообщение (нет соответствующего обработчика), то он не выдает ошибку, а пропускает это сообщение дальше по иерархии. Таким образом например способность может послать сообщение отрисовки и оно пройдет через класс игры напрямую в класс-отрисовщик.

Абстрактный класс Ability представляет собой способность с абстрактнмым методом use(). Все способности наследуются от него.

Разберем способности на примере Scanner: в конструкторе она принимает Player &target — цель, игрок, на которого эта способность используется. В конструкторе Scanner берет от игрока только то, что нужно для выполнения этой способности, а именно ссылку на поле (Field). Таким образом, существует единый интерфейс создания способностей, при этом есть возможность хранить внутри способности только то, что действительно необходимо для нее. Это может быть полезно, чтобы случайно не изменить то, к чему эта способность иметь доступа не должна (под «случайно» я понимаю неаккуратность программиста, который реализует код этой способности).

Класс AbilityRegistry представляет собой реестр всех способностей, существует только один глобальный экземпляр этого класса. Он в себе хранит записи вида (название способности; функция). Эта функция имеет сигнатуру std::unique\_ptr<Ability>(Player &user, Player &target) и возвращает новый объект заданной способности, используемой игроком user на игрока target. Для регистрирования способности в реестре способностей реализован метод add. Каждую способность можно зарегистрировать в своем .cpp файле, что очень удобно. Также реестр способностей представляет единый список способностей для всех, кому он может понадобится. Например, при создании менеджера способностей тот просто итерируется по реестру. Это позволяет избежать повторения кода (Похожий код в будущем может появится как минимум в методе загрузки игры).

Класс AbilityManager представляет собой менеджер способностей. Он хранит в себе очередь функций-генераторов способностей (из реестра способностей). В конструкторе он итерируется по реестру способностей, забирает из него по одной способности каждого типа, после чего перемешивает их. Метод addRandomAbility позволяет создать новую способность, берется случайная способность из реестра способностей. Метод top возвращает функцию-генератор на вершине очереди. Если очередь пустая, выбрасывает исключение. Метод pop удаляет функцию-генератор из вершины очереди. Она вызывается, если способность была успешно использована (неуспешно значит например игрок выбрал некорректные координаты).

Класс Player представляет собой одного из двух игроков. У игрока есть поле (класс Field), набор кораблей (класс ShipManager) и текущее положение его курсора. Поскольку по правилам игры не бывает игрока без поля и не бывает игрока без кораблей, а также не бывает поля без игрока и кораблей без игрока, было принято решение сделать такой класс.

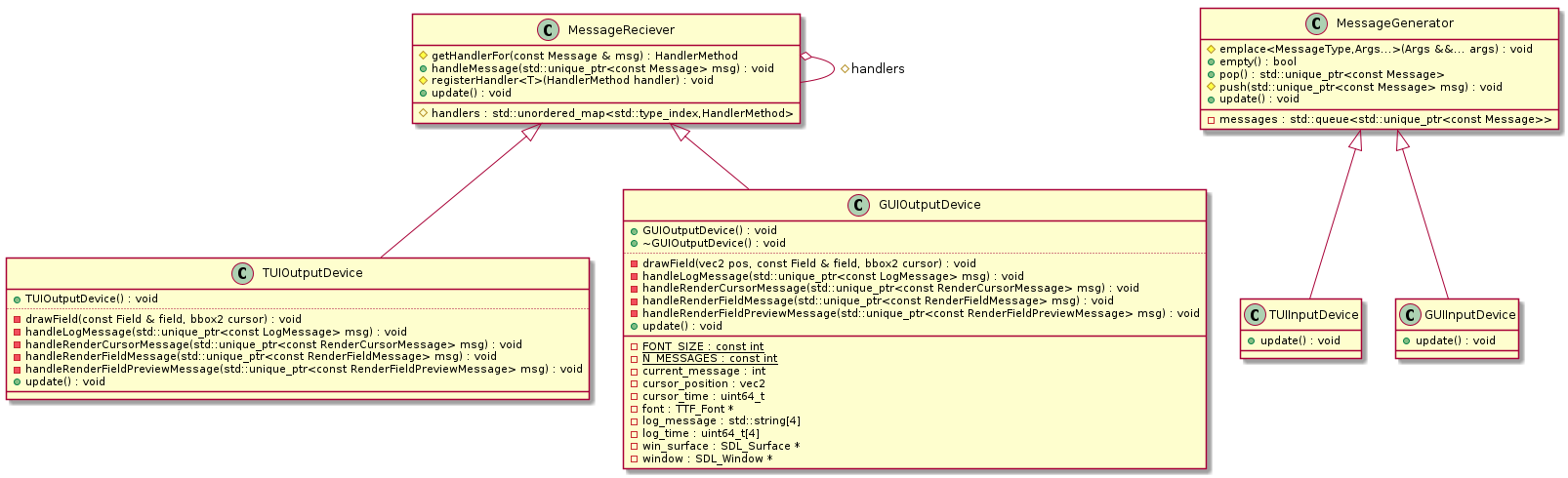
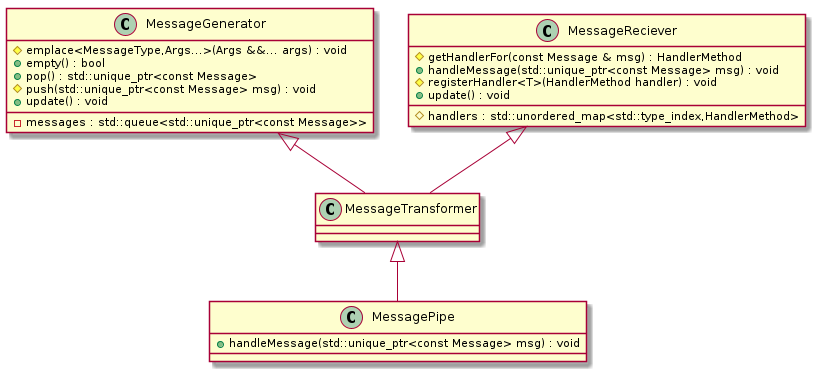
Класс HumanPlayer наследуется от Player и представляет собой игрока, контроллируемого человеком. По условию лабороторной, игрок может использовать способности, а его соперник (ИИ), не может. Поэтому у HumanPlayer`а есть свой AbilityManager. Решение отнаследовать HumanPlayer от Player может пригодиться в будущем для создания режима игры человек-против-человека: способности не могут изменить набор способностей игрока, поэтому имеет смысл в методах использования способности в качестве цели указывать Player &target. Таким образом, можно будет использовать способности как на ИИ, так и на реального игрока (В теории, можно даже создатть режим ИИ-против-ИИ). У этого класса есть метод attack, который производит обычную атаку c помощью метод Field::attack. Этот метод возвращает true, если сегмент корабля был поврежден и false — если нет. Если сегмент поврежден, то игрок для соответствующего корабля вызывает метод isDestroyed. Если корабль уничтожен, то в AbilityManager`е создается новая способность. Также у игрока есть метод useAbility, который использует способность: сначала из очереди способностей достается функцию-генератор, далее она вызывается, чтобы создать объект способности с пользователем this и заданной целью. Далее эта способность используется. Если было выброшено исключение, то работа функции заканчивается здесь. Если же всё прошло успешно, то далее эта функция удаляется из очереди способностей.

Также был реализован набор классов-исключений. Базовый абстрактный класс — SeabattleException (наследуется от std::exception). Имеет виртуальный метод const char \*what(). Все исключения определяют этот метод.

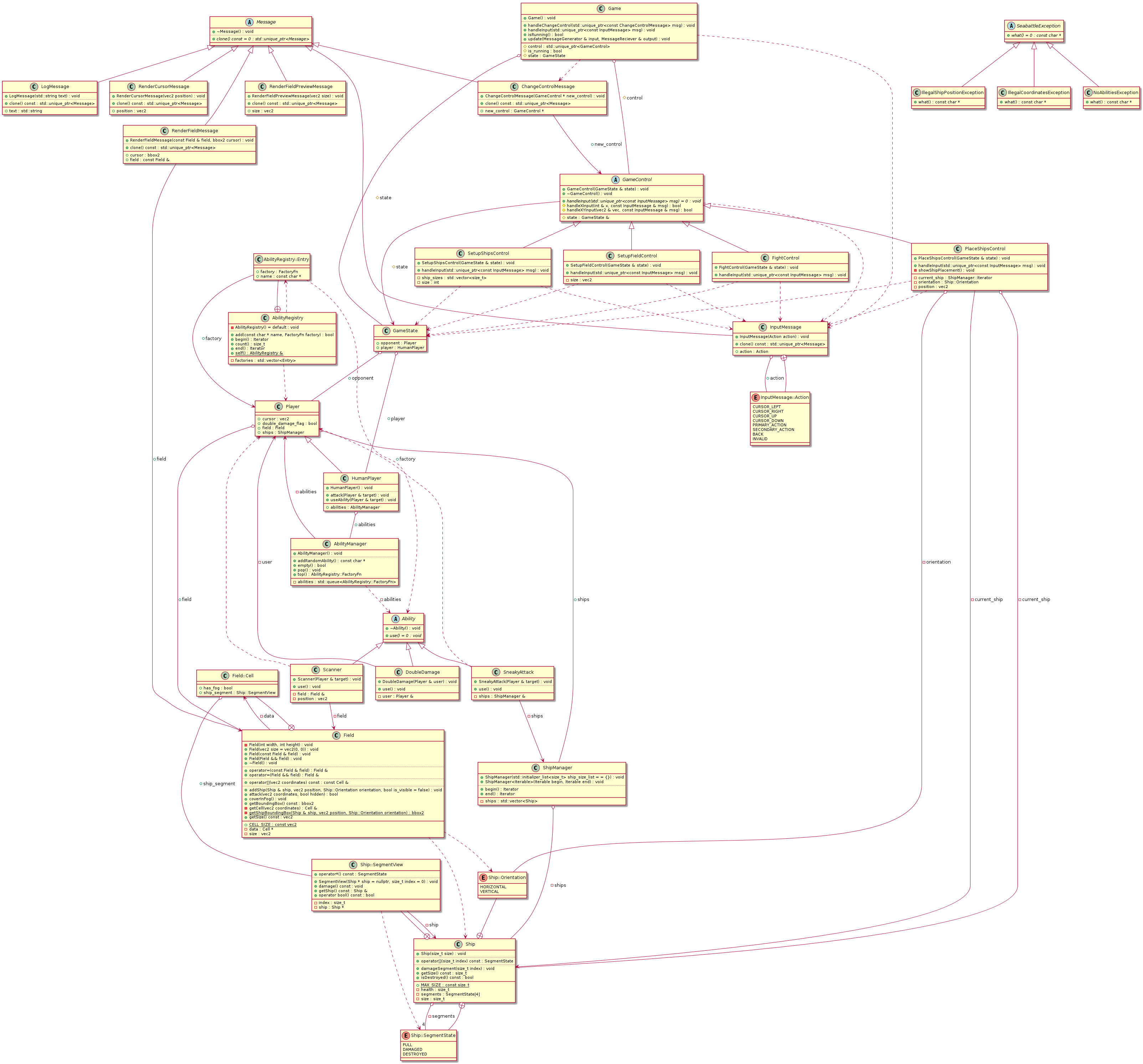
Исключение NoAbilitiesException выбрасывается, когда игрок хочет использовать способность, но очередь пуста. Исключение IllegalShipPositionException выбрасывается при неудачной попытке поставить корабль на поле (корабль находится за пределами поля или касается других кораблей).

**Диаграммы классов.**

Диаграмма системы сообщений:

Диаграмма устройств ввода/вывода:

Общая диаграмма:



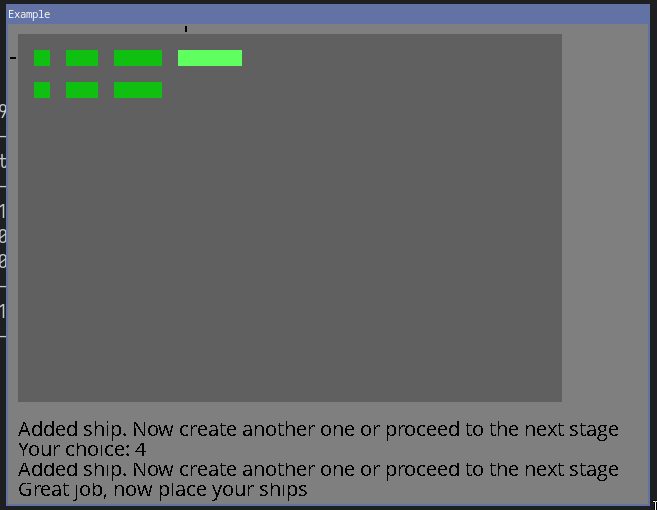
## Тестирование.

Результаты тестирования:

В случае возникновения исключения, обрабатывающее некорректность ввода, оно ловится и его сообщение выводится снизу-слева

Сканер вывел сообщение «Scanner didn’t find anything»

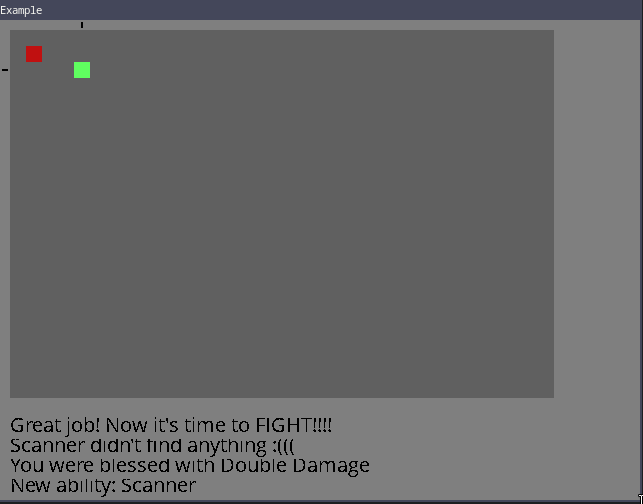
Расстановка кораблей:



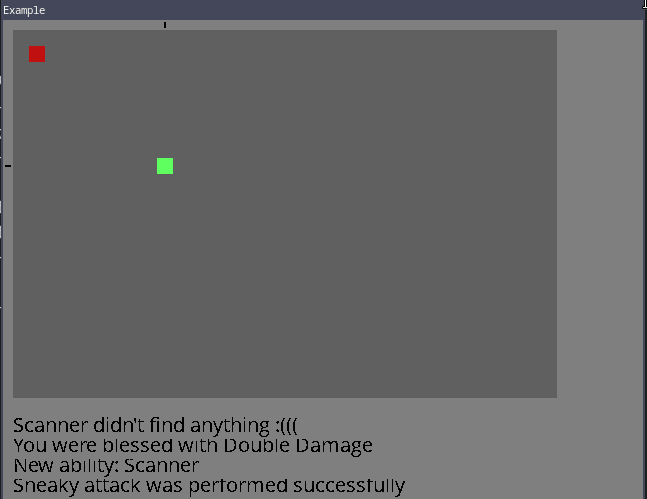
Использование сканера:



## Использование двойного урона (из сообщений видно, что после использования сканера был использован двойной урон, после чего игрок совершил атаку по ранее закрытой клетке и она сразу же уничтожилась, после чего игроку была выдана новая способность за уничтожение корабля):

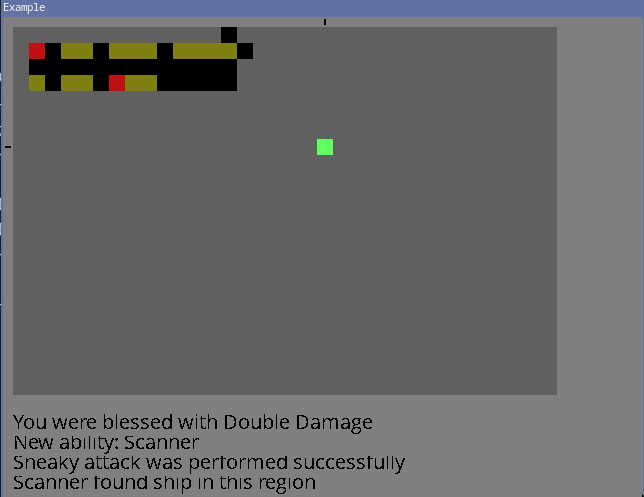


Использование обстрела (урон прошёл, но клета так и осталась скрытой):

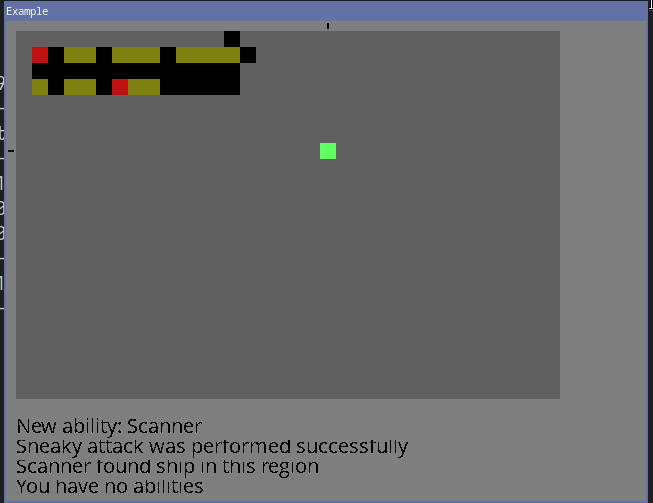


## Повторное использование сканера:

Проверка работоспособности обстрела (игрок, используя обычные атаки нанес 1 урон по каждому сегменту и оказалось, что один из сегментов уже был поврежден):



Сообщение при отсуствии способностей у игрока:



## Выводы.

Был изучен полиморфизм в объектно ориентированном программировании. Полученные знания были применены на практике: была реализована система способностей для игры «Морской бой» из предыдущей лабороторной работы. Также были создана собственная система классов исключений.