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

ОТЧЕТ

по лабораторной работе №6
по дисциплине «Объектно-ориентированное программирование»
Тема: сериализация, исключения

Студент гр.0382 _____ Литягин С.М.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2021

Цель работы.

Реализовать сохранение в определенном виде состояния программы с возможностью последующего его восстановления даже после закрытия программы.

Задание.

Сериализация - это сохранение в определенном виде состоянии программы с возможностью последующего его восстановления даже после закрытия программы. В рамках игры, это сохранения и загрузка игры.

Требования:

- Реализовать сохранения всех необходимых состояний игры в файл
- Реализовать загрузку файла сохранения и восстановления состояния игры
- Должны быть возможность сохранить и загрузить игру в любой момент
- При запуске игры должна быть возможность загрузить нужный файл
- Написать набор исключений, который срабатывают если файл с сохранением некорректный
- Исключения должны сохранять транзакционность. Если не удалось сделать загрузку, то программа должна находиться в том состоянии, которое было до загрузки. То есть, состояние игры не должно загружаться частично

Потенциальные паттерны проектирования, которые можно использовать:

- *Снимок (Memento) - получение и восстановления состояния объектов при сохранении и загрузке*

Выполнение работы.

Нужно реализовать сохранение и загрузку игры. Чтобы это сделать в рамках данной игры, достаточно сохранять изменяемые параметры объектов игры, т.е. игра, противников, вещей и игрового поля.

Для сохранения и загрузки был написан класс SaveAndLoad. У него всего два метода Save(HeroAction *heroAct, ObjectAction *objAct, Command command) и Load(HeroAction *heroAct, ObjectAction *objAct, Command command). Их аргументы – это указатели на объекты классов, в которых содержатся указатели на все нужные нам объекты для сохранения; а также команда, по которой Manager вызвал метод; метод Load вызывается в блоке try/catch (для отлавливания ошибки LoadError). Передача команды идет по следующей причине: была сделана возможность сохранить в/загрузить файл с названием, что вводится в консоль, а также возможность быстрого сохранения в/загрузки файл/файла стандартного сохранения (имеет название “save1”); для первого варианта это команды save/load, для второго – save1/load1.

Поговорим подробнее о сохранении. У каждого объекта есть какие-то параметры. Поле, например, состоит из клеток определенного типа; игрок имеет тип, здоровье, координату, максимальное здоровье, наносимый урон и количество монеток; вещи имеют тип, координату (поле с данными, что изменяют параметры игрока, константны, поэтому их сохранять не нужно); противники имеют тип, здоровье, координату (остальные параметры также константны). Эти данные мы можем записать числами (даже типы, поскольку те являются элементами перечислений).

Когда вызывается метод сохранения, то выбирается файл, куда сохранить данные (при команде save1 -> save1.txt; save -> просим ввести название в консоль). В файл с указанным названием записываются подряд данные, что получают при помощи метода объектов objAct и heroAct (что мы передавали в аргументах) GetData(). Данный метод возвращает нам строку, содержащую нужные нам цифры. Если говорить о objAct, то он вернет нам строку, в который

записано: а)тип каждой клетки поля; б) количество вещей (далее информация о каждой вещи согласно пункту “в”); в) тип объекта/координата х/координата у; г) количество противников (далее информация о каждом противнике согласно пункту “д”); д) тип объекта/координата х/координата у/здоровье. Метод heroAct же вернет нам: е) информация об игроке: тип объекта/координата х/координата у/здоровье/наносимый урон/максимальное здоровье/монетки. Каждый указанный параметр записывается в строгом порядке с новой строки, что поможет нам отлавливать ошибки при чтении данных для загрузки.

Поговорим теперь о загрузке. Суть идеи, по которой она реализована, проста. Мы считываем данные, в идеальном случае без ошибок. По ним создаем объекты. А потом заменяем их везде, где это нужно (в данном случае, достаточно поменять указатели на них в объектах классов HeroAction и ObjectAction).

Когда вызывается метод загрузки, то выбирается файл, откуда загрузить данные (при команде load1 -> save1.txt; load -> просим ввести название в консоль). Если файл не существует, то выбрасываем ошибку LoadError с соответствующими аргументами. Если же файл открыт удачно, то считываем оттуда информацию. Для этого мы используем оператор >> (файл открыт в потоке). Поскольку мы точно знаем размер поля Size (поле квадратное), то мы точно знаем количество чисел отвечающих за тип клетки. Сначала создаем новый объект поля new_field. Затем, начинаем по очереди считывать типы клеток в заранее объявленную переменную type. Если чтение не удалось или тип не соответствует доступному типу из перечисления типов клетки, то удаляем созданный объект поля и то выбрасываем ошибку LoadError с соответствующими аргументами; если тип считывается верно -> указываем данный тип соответствующей клетке поля методом SetType(). По завершении чтения типов, согласно описанному ранее порядку сохранения, далее идет информация о вещах. Заранее объявляем переменные Thing** new_thing (указатель на новый массив вещей), int new_THING (количество новых вещей)

и int x,y (для координаты объекта). Сначала считывается число новых вещей в переменную new_THING. Если чтение не удалось или считанное число вне диапазона [0, 1/3*Size), где Size – размер поля, то удаляем созданное поле и то выбрасываем ошибку LoadError с соответствующими аргументами. Иначе – создаем новый массив указателей на объекты Thing в количестве new_THING. Затем new_THING раз считывает параметры вещей (сначала тип в переменную type, затем координату x, y). Если считывание какого-то параметра не удалось, или тип не соответствует типам вещей, или координата находится вне позволенного диапазона, или по этой координате уже есть какой-то объект на новом поле – удаляем все объекты, что успел создать метод, и то выбрасываем ошибку LoadError с соответствующими аргументами. Если же все хорошо -> создаем вещь, согласно типу, помещаем его в new_thing[i], устанавливаем его на координату x,y (если она равна (-1;-1), то вызываем метод GetData() для объекта, это поменяет поле is_available на false; координата (-1;-1) в данной игре говорит о недоступности объекта); если координата не (-1;-1) и по ней не стоит уже объект на новом поле – устанавливаем в соответствующую клетку вещь. По завершении чтения и создания вещей, согласно описанному ранее порядку сохранения, далее идет информация о противниках. Заранее объявляем переменные Unit** new_enemy (указатель на новый массив противников), int new_ENEMY (количество новых противников) и int health(для здоровья противника). Сначала считывается число новых противников в переменную new_ENEMY. Если чтение не удалось или считанное число вне диапазона [0, 1/3*Size), где Size – размер поля, то удаляем созданное поле, созданные вещи и то выбрасываем ошибку LoadError с соответствующими аргументами. Иначе – создаем новый массив указателей на объекты Unit в количестве new_ENEMY. Затем new_ENEMY раз считывает параметры вещей (сначала тип в переменную type, координату в x, y и здоровье в health). Если считывание какого-то параметра не удалось, или тип не соответствует типам противников, или координата находится вне позволенного диапазона, или по этой координате уже

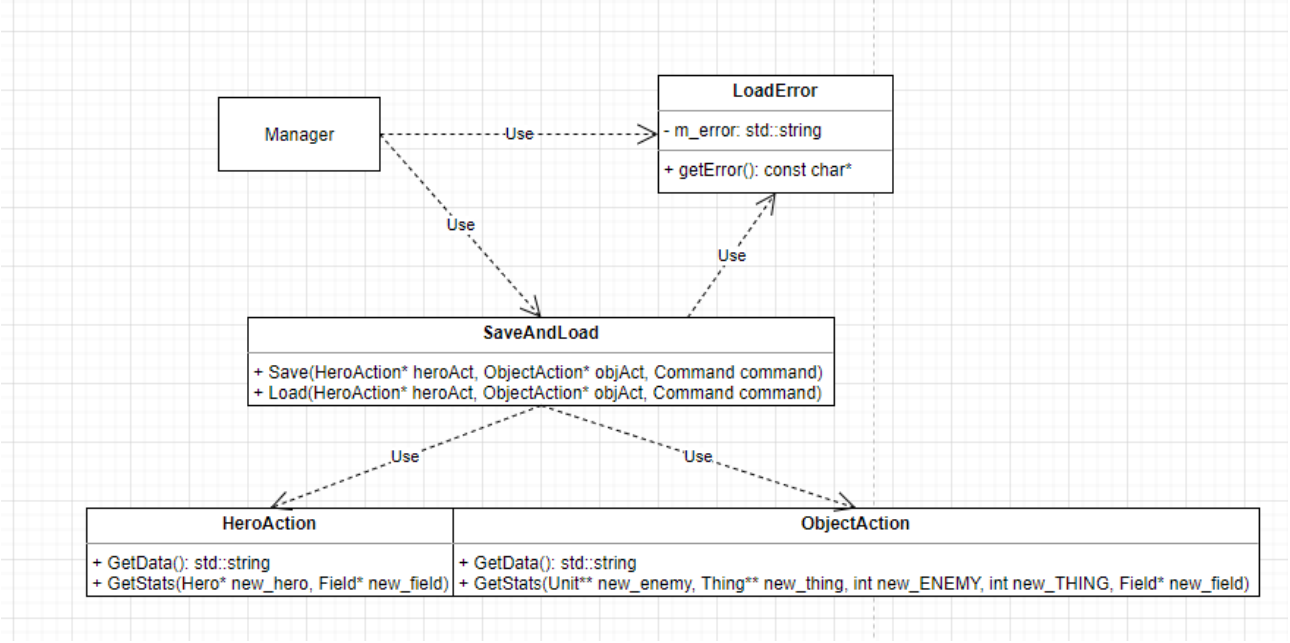
есть какой-то объект на новом поле (на здоровье ограничений не наложено) – удаляем все объекты, что успел создать метод, и то выбрасываем ошибку LoadError с соответствующими аргументами. Если же все хорошо -> создаем противника, согласно типу, помещаем его в new_enemy[i], устанавливаем его на координату x,y (если она равна (-1;-1), то вызываем метод SetHealth(-5) для объекта, это поменяет поле is_alive на false; координата (-1;-1) в данной игре говорит о недоступности объекта); если координата не (-1;-1) и по ней не стоит уже объект на новом поле – устанавливаем в соответствующую клетку противника. Устанавливаем методом SetHealth() здоровье health противнику, если тот жив (проверяется методом IsAlive()), затем вновь проверяем, если противник мертв – устанавливаем в указатель на объект в клетке, на которой находится противник, указатель на nullptr. После всего этого остается лишь считать параметры игрока. Объявляем переменные Hero* new_gamer (указатель на объект нового игрока), int force (для наносимого урона), int max_health (для максимального здоровья), int coins (для монеток). Затем происходит все аналогично предыдущим описаниям, лишь параметров чуть больше. Собственно, на этом все данные из сохранения считаны, новые объекты созданы. Осталось установить их в соответствующие поля объектов objAct и heroAct, вызывая методы соответственно.

Для этого были написаны методы GetStats(). Для класса ObjectAction передаются в аргументах new_enemy, new_thing, new_ENEMY, new_THING и new_field. В методе мы удаляем старые объекты, а в соответствующие поля (enemy, thing, ENEMY, THING, field) устанавливаем переданные. Также обновляем указатель на массив клеток в объекте draw класса Drawing (отвечает за отрисовку поля и объектов на нем). Для класса HeroAction передаются в аргументах new_gamer, new_field. Здесь мы удаляем уже лишь старый объект игрока (т.к. старое поле мы уже удалили при вызове метода ObjectAction); затем устанавливаем в соответствующие поля (gamer, field) переданные объекты. Процесс загрузки завершен.

Про отлавливание ошибок. Был написан класс LoadError. В его заголовочном файле определенно два перечисления: objError и Error с возможными объектами и ошибками (объекты ошибок: cellEr, heroEr, thingEr, enemyEr; сами ошибки: typeEr, posEr, hpEr, forcEr, mhpEr, coinEr, numbEr). Если при загрузке сохранения возникла ошибка, то нам выбрасывает объект этого класса. При его конструировании аргументами является пара переменных objError и Error соответственно. По ним, в поле std::string m_error собирается сообщение об ошибке вида “Not correct data for objError/Error”. Если при вызове метода загрузки в Manager мы отловили ошибку LoadError, то выводим в поток ошибок std::cerr строку с ошибкой, которую получаем с помощью метода getError() из выброшенного методом объекта.

UML-диаграмма классов представлена на рис. 1.

Рисунок 1 – UML-диаграмма классов.



Выводы.

В ходе работы было изучено сохранение и восстановление программы.