ГУАП

КАФЕДРА № 42

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| старший преподаватель |  |  | |  | С. Ю. Гуков |
| должность, уч. степень, звание |  | | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 2 |
| ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ |
| по курсу: |
| ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ гр. № | 4326 |  |  |  | Г. С. Томчук |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2025

1. Цель работы

Цель работы: ознакомиться с основными паттернами проектирования и понять их назначение для решения общих задач проектирования в конкретном контексте, научиться применять паттерны на практике при проектировании и разработке программного обеспечения.

1. Задание

Необходимо придумать контекст и разработать программу, используя предложенные в вариантах два паттерна проектирования. Реализация должна быть сделана в одном связанном проекте (контексте). Также требуется нарисовать UML диаграмму классов реализуемой программы.

Каждый студент берет два варианта заданий: один – свой вариант по списку, второй – любой на выбор с паттерном другой группы. То есть, например, один из паттернов должен быть структурным или порождающим, а второй должен быть поведенческим – это обеспечит баланс между архитектурой и поведением. Не разрешается реализовывать в каждом конкретном паттерне такой же контекст, как был в примерах на лекциях.

Проект может быть выполнен либо в качестве консольного приложения (тогда обязателен командно-текстовый интерфейс), либо иметь графический пользовательский интерфейс (User Interface, UI), а также может быть написан на любом языке программирования.

1. Краткое описание хода разработки и назначение используемых технологий

В рамках лабораторной работы был выбран контекст разработки простого консольного редактора уровня для условной 2D-игры. Программа реализована на языке Python, что позволило сосредоточиться на архитектуре и паттернах, а не на низкоуровневых деталях. Для хранения состояния уровня и удобного обмена данными используется формат JSON, так как он является простым и широко применяемым стандартом сериализации.

При разработке программа была разделена на несколько логических компонентов:

* Модель сущностей уровня (игрок, враг, сокровище),
* Фабрика для их создания,
* Механизм команд для действий пользователя,
* Менеджер истории для undo/redo,
* Командно-текстовый интерфейс, позволяющий добавлять, перемещать и удалять сущности, сохранять и загружать уровень.

Такой подход обеспечил модульность кода и дал возможность гибко расширять функциональность.

1. Описание использованных паттернов и аргументация их выбора в придуманных контекстах

В проекте реализованы два паттерна из разных групп — один порождающий, второй поведенческий:

1. Factory Method (Фабричный метод). Данный паттерн был использован для создания игровых объектов уровня: Player, Enemy, Treasure. Все сущности создаются через единый интерфейс EntityFactory.create(kind, x, y, \*\*kwargs), что избавляет клиентский код от знания о конкретных конструкторах и их параметрах. Такой подход оправдан в контексте редактора уровня: в будущем можно будет легко добавлять новые типы сущностей (например, двери, ловушки, NPC) без изменения клиентской логики. Достаточно будет зарегистрировать новый класс в фабрике.
2. Command (Команда). Этот паттерн применён для реализации действий пользователя: добавления, перемещения и удаления сущностей. Каждое действие представлено отдельным классом (AddEntityCommand, MoveEntityCommand, DeleteEntityCommand), поддерживающим методы execute() и undo(). Использование Command позволило организовать систему undo/redo, что крайне важно для редактора любого типа — пользователь всегда должен иметь возможность отменить или повторить свои действия. Кроме того, такая архитектура упрощает расширение: например, добавление новой команды CloneEntityCommand или ChangePropertyCommand не потребует изменения существующего кода.

Выбор именно этих паттернов объясняется их естественным применением в контексте редактора: Factory Method решает задачу удобного и расширяемого создания сущностей, а Command — задачу управления действиями пользователя и истории изменений. На рисунке 1 изображена диаграмма классов.

A screenshot of a computer program

Description automatically generated

Рисунок 1 – UML-диаграмма классов программы

1. Результаты работы программы с примерами разных сценариев

На рисунке 2 изображен пример использования программы.

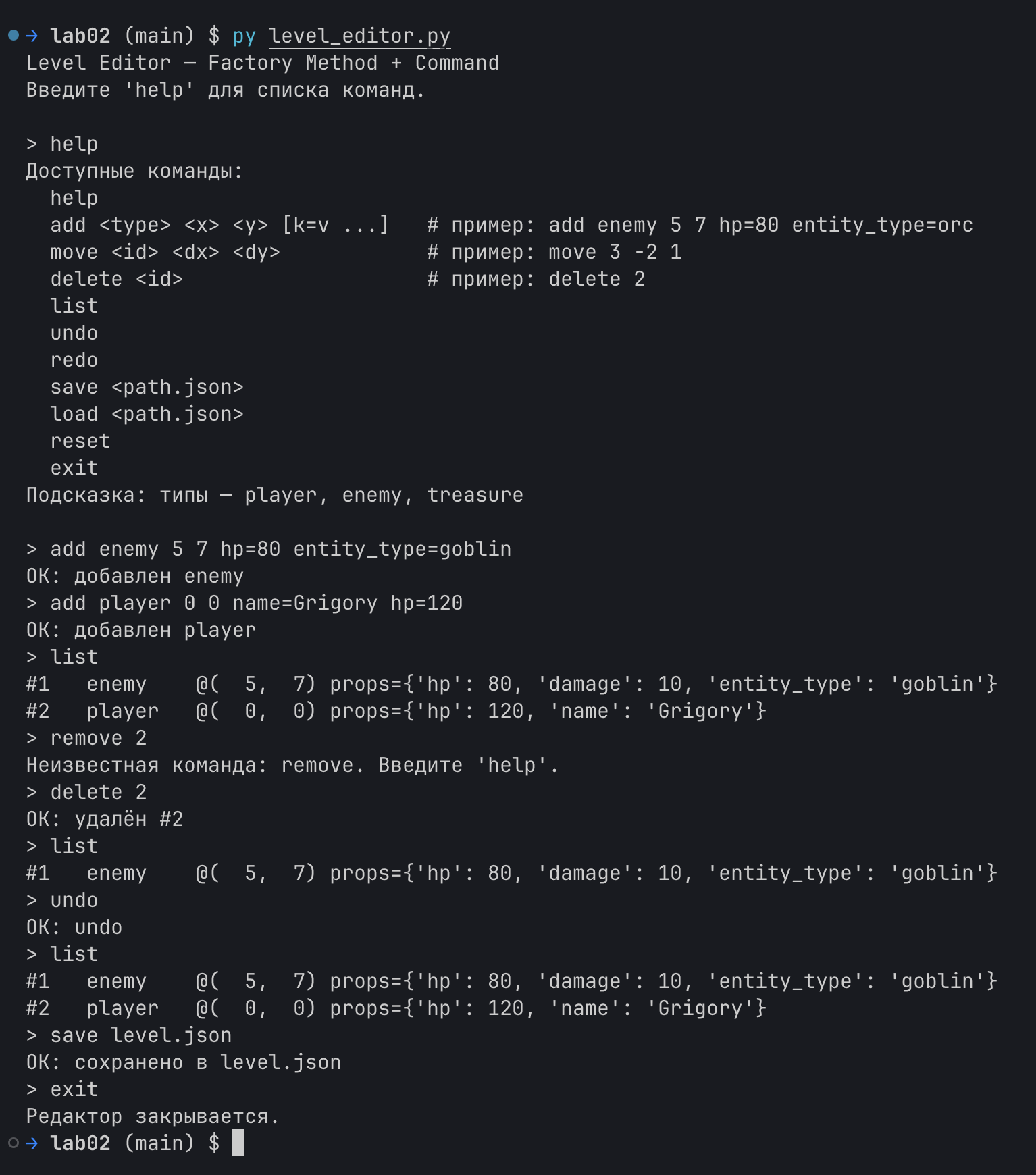


Рисунок 2 – Пример использования программы

1. Выводы

В ходе выполнения лабораторной работы были изучены и применены два паттерна проектирования — Factory Method и Command. Их совместное использование в едином проекте позволило построить простую, но модульную архитектуру редактора уровня.

Реализация продемонстрировала преимущества применения паттернов:

* снижение связности между компонентами,
* расширяемость системы за счёт фабрики,
* удобное управление историей изменений благодаря командам.

Таким образом, работа показала, что использование паттернов проектирования значительно упрощает разработку программного обеспечения, делает код более гибким и сопровождаемым.

ПРИЛОЖЕНИЕ А

from \_\_future\_\_ import annotations

import json

import shlex

from dataclasses import asdict, dataclass, field

from typing import Any, Dict, List, Optional, Tuple, Type

# Domain Model (Entities)

@dataclass

class Entity:

id: int

type: str

x: int

y: int

props: Dict[str, Any] = field(default\_factory=dict)

def move(self, dx: int, dy: int) -> Tuple[int, int]:

old = (self.x, self.y)

self.x += dx

self.y += dy

return old

class Player(Entity):

def \_\_init\_\_(self, id: int, x: int, y: int, \*\*kwargs):

defaults = {"hp": 100, "name": "Hero"}

defaults.update(kwargs)

super().\_\_init\_\_(id=id, type="player", x=x, y=y, props=defaults)

class Enemy(Entity):

def \_\_init\_\_(self, id: int, x: int, y: int, \*\*kwargs):

defaults = {"hp": 50, "damage": 10, "entity\_type": "goblin"}

defaults.update(kwargs)

super().\_\_init\_\_(id=id, type="enemy", x=x, y=y, props=defaults)

class Treasure(Entity):

def \_\_init\_\_(self, id: int, x: int, y: int, \*\*kwargs):

defaults = {"value": 100, "currency": "gold"}

defaults.update(kwargs)

super().\_\_init\_\_(id=id, type="treasure", x=x, y=y, props=defaults)

# Factory Method

class EntityFactory:

"""Factory Method: создаёт сущности по строковому типу."""

\_registry: Dict[str, Type[Entity]] = {

"player": Player,

"enemy": Enemy,

"treasure": Treasure,

}

def \_\_init\_\_(self):

self.\_next\_id = 1

def create(self, kind: str, x: int, y: int, \*\*kwargs) -> Entity:

kind = kind.lower()

if kind not in self.\_registry:

raise ValueError(

f"Неизветсный тип сущности: '{kind}'. "

f"Доступны: {', '.join(self.\_registry)}"

)

cls = self.\_registry[kind]

ent = cls(self.\_alloc\_id(), x, y, \*\*kwargs)

return ent

def recreate(

self, type\_name: str, id\_: int, x: int, y: int, props: Dict[str, Any]

) -> Entity:

"""Используется при загрузке из JSON (восстановление точного подкласса)."""

type\_name = type\_name.lower()

if type\_name not in self.\_registry:

# fallback на базовый Entity

ent = Entity(id=id\_, type=type\_name, x=x, y=y, props=props)

else:

cls = self.\_registry[type\_name]

ent = cls(id\_, x, y, \*\*props)

# корректируем счётчик id, чтобы не пересекаться

if id\_ >= self.\_next\_id:

self.\_next\_id = id\_ + 1

return ent

def \_alloc\_id(self) -> int:

id\_ = self.\_next\_id

self.\_next\_id += 1

return id\_

# Level aggregate

class Level:

def \_\_init\_\_(self, factory: EntityFactory):

self.factory = factory

self.entities: Dict[int, Entity] = {}

def add\_entity(self, entity: Entity) -> None:

if entity.id in self.entities:

raise ValueError(f"Сущность с id {entity.id} уже есть")

self.entities[entity.id] = entity

def remove\_entity(self, entity\_id: int) -> Entity:

if entity\_id not in self.entities:

raise ValueError(f"Сущность с id {entity\_id} не найдена")

return self.entities.pop(entity\_id)

def get(self, entity\_id: int) -> Entity:

if entity\_id not in self.entities:

raise ValueError(f"Сущность с id {entity\_id} не найдена")

return self.entities[entity\_id]

def to\_json(self) -> str:

payload = [

{

"id": e.id,

"type": e.type,

"x": e.x,

"y": e.y,

"props": e.props,

}

for e in self.entities.values()

]

return json.dumps(payload, ensure\_ascii=False, indent=2)

def from\_json(self, data: str) -> None:

items = json.loads(data)

self.entities.clear()

for it in items:

ent = self.factory.recreate(

type\_name=it["type"],

id\_=it["id"],

x=it["x"],

y=it["y"],

props=it.get("props", {}),

)

self.entities[ent.id] = ent

# Command Pattern

class Command:

def execute(self) -> None:

raise NotImplementedError

def undo(self) -> None:

raise NotImplementedError

class AddEntityCommand(Command):

def \_\_init\_\_(self, level: Level, kind: str, x: int, y: int, \*\*kwargs):

self.level = level

self.kind = kind

self.x = x

self.y = y

self.kwargs = kwargs

self.created\_id: Optional[int] = None

self.\_entity\_snapshot: Optional[Entity] = None

def execute(self) -> None:

if self.\_entity\_snapshot is None:

ent = self.level.factory.create(self.kind, self.x, self.y, \*\*self.kwargs)

else:

ent = self.\_entity\_snapshot

self.level.add\_entity(ent)

self.created\_id = ent.id

def undo(self) -> None:

if self.created\_id is None:

return

ent = self.level.remove\_entity(self.created\_id)

# сохраняем для возможного повтора redo

self.\_entity\_snapshot = ent

class MoveEntityCommand(Command):

def \_\_init\_\_(self, level: Level, entity\_id: int, dx: int, dy: int):

self.level = level

self.entity\_id = entity\_id

self.dx = dx

self.dy = dy

self.\_prev\_pos: Optional[Tuple[int, int]] = None

def execute(self) -> None:

ent = self.level.get(self.entity\_id)

self.\_prev\_pos = (ent.x, ent.y)

ent.move(self.dx, self.dy)

def undo(self) -> None:

if self.\_prev\_pos is None:

return

ent = self.level.get(self.entity\_id)

ent.x, ent.y = self.\_prev\_pos

class DeleteEntityCommand(Command):

def \_\_init\_\_(self, level: Level, entity\_id: int):

self.level = level

self.entity\_id = entity\_id

self.\_deleted: Optional[Entity] = None

def execute(self) -> None:

self.\_deleted = self.level.remove\_entity(self.entity\_id)

def undo(self) -> None:

if self.\_deleted is None:

return

self.level.add\_entity(self.\_deleted)

class CommandManager:

def \_\_init\_\_(self):

self.\_undo\_stack: List[Command] = []

self.\_redo\_stack: List[Command] = []

def do(self, cmd: Command) -> None:

cmd.execute()

self.\_undo\_stack.append(cmd)

self.\_redo\_stack.clear()

def undo(self) -> bool:

if not self.\_undo\_stack:

return False

cmd = self.\_undo\_stack.pop()

cmd.undo()

self.\_redo\_stack.append(cmd)

return True

def redo(self) -> bool:

if not self.\_redo\_stack:

return False

cmd = self.\_redo\_stack.pop()

cmd.execute()

self.\_undo\_stack.append(cmd)

return True

# CLI

BANNER = """\

Level Editor — Factory Method + Command

Введите 'help' для списка команд.

"""

HELP = """\

Доступные команды:

help

add <type> <x> <y> [k=v ...] # пример: add enemy 5 7 hp=80 entity\_type=orc

move <id> <dx> <dy> # пример: move 3 -2 1

delete <id> # пример: delete 2

list

undo

redo

save <path.json>

load <path.json>

reset

exit

Подсказка: типы — player, enemy, treasure

"""

def parse\_kv(pairs: List[str]) -> Dict[str, Any]:

out: Dict[str, Any] = {}

for p in pairs:

if "=" not in p:

raise ValueError(f"Bad arg '{p}', ожидается k=v")

k, v = p.split("=", 1)

# попытка привести к int/float/bool, иначе строка

if v.lower() in ("true", "false"):

out[k] = v.lower() == "true"

else:

try:

out[k] = int(v)

except ValueError:

try:

out[k] = float(v)

except ValueError:

out[k] = v

return out

def cmd\_list(level: Level) -> None:

if not level.entities:

print("(пусто)")

return

for e in sorted(level.entities.values(), key=lambda e: e.id):

print(f"#{e.id:<3} {e.type:<8} @({e.x:>3},{e.y:>3}) props={e.props}")

def main():

factory = EntityFactory()

level = Level(factory)

history = CommandManager()

print(BANNER)

while True:

try:

line = input("> ").strip()

except (EOFError, KeyboardInterrupt):

print("\nBye!")

break

if not line:

continue

try:

parts = shlex.split(line)

except ValueError as e:

print(f"Ошибка парсинга: {e}")

continue

cmd = parts[0].lower()

try:

if cmd == "help":

print(HELP)

elif cmd == "add":

if len(parts) < 4:

print("Использование: add <type> <x> <y> [k=v ...]")

continue

kind = parts[1]

x, y = int(parts[2]), int(parts[3])

kwargs = parse\_kv(parts[4:]) if len(parts) > 4 else {}

history.do(AddEntityCommand(level, kind, x, y, \*\*kwargs))

print(f"OK: добавлен {kind}")

elif cmd == "move":

if len(parts) != 4:

print("Использование: move <id> <dx> <dy>")

continue

eid, dx, dy = int(parts[1]), int(parts[2]), int(parts[3])

history.do(MoveEntityCommand(level, eid, dx, dy))

print(f"OK: перемещён #{eid} на ({dx},{dy})")

elif cmd == "delete":

if len(parts) != 2:

print("Использование: delete <id>")

continue

eid = int(parts[1])

history.do(DeleteEntityCommand(level, eid))

print(f"OK: удалён #{eid}")

elif cmd == "list":

cmd\_list(level)

elif cmd == "undo":

if history.undo():

print("OK: undo")

else:

print("Нечего отменять")

elif cmd == "redo":

if history.redo():

print("OK: redo")

else:

print("Нечего повторять")

elif cmd == "save":

if len(parts) != 2:

print("Использование: save <path.json>")

continue

path = parts[1]

with open(path, "w", encoding="utf-8") as f:

f.write(level.to\_json())

print(f"OK: сохранено в {path}")

elif cmd == "load":

if len(parts) != 2:

print("Использование: load <path.json>")

continue

path = parts[1]

with open(path, "r", encoding="utf-8") as f:

level.from\_json(f.read())

print(f"OK: загружено из {path}")

elif cmd == "reset":

level.entities.clear()

print("OK: уровень очищен")

elif cmd in ("exit", "quit"):

print("Редактор закрывается.")

break

else:

print(f"Неизвестная команда: {cmd}. Введите 'help'.")

except Exception as e:

print(f"Ошибка: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

main()