Федеральное государственное бюджетное образовательное учреждение
высшего образования
«АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Архитектура программных систем

“Секретные короткие переписки”

Исполнитель:
студент гр. № 474б
Сердюк Егор

2020 г.

Постановка задачи

Егор хотел спросить «как дела?» у своего друга Ильи, но понял, что их прослушивают. Егор решил не мириться с этим и подошел серьезно к вопросу – нужно обмениваться зашифрованными (неважно каким шифрованием, предположим, оба друга могут на бумажке расшифровать сходу шифры) сообщениями, чтобы не было ясно третьим лицам, что в них написано. Так же он хотел иметь возможность отслеживать отправку сообщений и возможность вернуть незашифрованный вариант сообщения (вдруг в нем совершены грамматические ошибки).

Описание примененных паттернов

1. Итератор: поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.
2. Наблюдатель: поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.
3. Снимок: поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

Вывод

Егор подумал, раз фразы короткие – не нужно шифровать каждое отдельное слово, сделав шифрование по целой фразе, благо он с Ильей может сразу понять написанное. Так же приходят оповещения, когда происходит отправка сообщения Илье. Чтобы не забывать, о чем разговор, был реализован механизм сохранения сообщения до отправки – всегда можно откатиться до незашифрованного варианта.

Приложение

```
# This is one of the variations of the three design patterns in document processi
ng
from __future__ import annotations
from collections.abc import Iterable, Iterator
from abc import ABC, abstractmethod
from random import randrange, sample
from typing import Any, List
from datetime import datetime
from string import ascii_letters, digits
import hashlib
```

```
class Message:
    """
    Самописный класс, реализующий простую структуру сообщения
    """
    def __init__(self, name, text, recipient):
        self.name = name
        self.text = text
        self.recipient = recipient

    def __repr__(self):
        return f"Send by {self.name} to {self.recipient}\nText: {self.text}"

    def update(self, status: MessageObserver):
        if not status.get_state(): return f"Send by {self.name} to {self.recipien
t}\nText: {self.text}"
        else: return f"Error"

    def text_list(self):
        return list(self.text.split(" "))
    """
    Механизмы сохранения данных сообщения и восстановления
    """
    def save(self):
        return Memento(self)

    def restore(self, memento) -> None:
        message = memento.get_message()
        self.name = message.name
        self.text = message.text
        self.recipient = message.recipient

# Iterator implementation
"""
Используем абстрактный класс Iterator из модуля collections
Реализуем метод __iter__() в итерируемом объекте и метод __next__() в итераторе
"""
class TextCryptor(Iterator):
    """
    Конкретные итераторы реализуют разные алгоритмы обхода, поэтому они хранят по
стоянно текущее положение обхода
    Здесь атрибут _position хранит положение обхода
    """
    _position: int = None

    def __init__(self, collection: Words) -> None:
        self._collection = collection
        self._position = 0

    def __str__(self):
        return list(self.value)
```

```
    def __next__(self):
        """
        Метод __next __() должен вернуть следующий элемент в последовательности
        В процессе происходит шифрование поступившей фразы в формате sha1
        При достижении конца коллекции и в последующих вызовах должно вызываться
        исключение StopIteration
        """
        try:
            value = self._collection[self._position]
            value = hashlib.sha1(value.encode('utf-8')).hexdigest()
            self._position += 1
        except IndexError:
            raise StopIteration()

        return value

class Words(Iterable):
    """
    Конкретные Коллекции предоставляют один или несколько методов для получения
    новых экземпляров итератора, совместимых с классом коллекции
    """
    def __init__(self, collection: List[Any] = []) -> None:
        self._collection = collection

    def __iter__(self) -> TextCryptor:
        """
        Метод __iter__() возвращает объект итератора
        """
        return TextCryptor(self._collection)

    def add_item(self, item: Any):
        self._collection.append(item)

# Observer implementation
class MessageObserver:
    """
    Здесь происходит хранение некоторого важного состояния и оповещает наблюдател
ей о его
    изменениях
    """
    _state: bool = None
    """
    Для удобства в этой переменной хранится состояние
    """
    _observers = []
    """
    Список наблюдателей
    """

    def attach(self, observer) -> None:
```

```
        print("Subject: Attached an observer.")
        self._observers.append(observer)

    def detach(self, observer) -> None:
        self._observers.remove(observer)

    """
    Методы управления подпиской
    """
    def notify(self) -> None:
        """
        Запуск обновления в каждом наблюдателе
        """
        print("Subject: Notifying observers...")
        for observer in self._observers:
            print("Message sent")
            observer.update(self)

    def send(self, message_state: bool) -> None:
        """
        Отслеживание отправки сообщения
        """
        self._state = message_state

        print("Message on the way") if message_state else print("WTF, error")
        self.notify()

    def get_state(self):
        return self._state

# Memento implementation
class Memento:
    """
    Интерфейс Снимка предоставляет способ извлечения метаданных снимка
    """
    def __init__(self, message):
        self.message = Message(name = message.name, text = message.text, recipien
t = message.recipient)

    def get_message(self):
        """
        Создатель использует этот метод, когда восстанавливает своё состояние
        """
        return self.message

class Caretaker:
    """
    Опекун не зависит от класса Конкретного Снимка
    Он работает со всеми снимками через базовый интерфейс Снимка
    """
    def __init__(self, message) -> None:
```

```
        self.mementos = []
        self.message = message

    def backup(self) -> None:
        self.mementos.append(self.message.save())

    def undo(self) -> None:
        if not len(self.mementos):
            return

        memento = self.mementos.pop()
        try:
            self.message.restore(memento)
        except Exception:
            self.undo()

if __name__ == "__main__":
    notify = MessageObserver()

    letter = Message("Egor", "How do you?", "Ilya")
    print(letter)

    notify.attach(letter)

    caretaker = Caretaker(letter)
    caretaker.backup()

    notify.send(True)

    collection = Words()
    collection.add_item(letter.name)
    collection.add_item(letter.text)
    collection.add_item(letter.recipient)

    name = "\n".join(collection)[0:40]
    text = "\n".join(collection)[41:81]
    recipient = "\n".join(collection)[82:122]

    letter.name = name
    letter.text = text
    letter.recipient = recipient
    print(letter)

    caretaker.undo()
    print(letter)
```