СБЕРБАНК ТЕХНОЛОГИИ

Шаблоны проектирования

ОБСУДИМ

СБЕРБАНК ТЕХНОЛОГИИ

- observer
- chain of responsibility
- iterator
- state
- strategy
- command

OBSERVER (НАБЛЮДАТЕЛЬ)

OBSERVER (НАБЛЮДАТЕЛЬ)

СБЕРБАНК ТЕХНОЛОГИИ

Один класс должен уметь получать/реагировать на события от другого класса.

Не должна создаваться сильная связанность между классами.

ЗАДАЧА

СБЕРБАНК ТЕХНОЛОГИИ

У нас есть класс, который как-то обрабатывает данные

```
public interface Processor {
    void process(List<Data> data);
}
```

Мы хотим уметь реагировать на обработку каждого элемента.

ЗАДАЧА

СБЕРБАНК ТЕХНОЛОГИИ

У нас есть класс, который как-то обрабатывает данные

```
public interface Processor {
    void process(List<Data> data);
}
```

Мы хотим уметь реагировать на обработку каждого элемента.

Например, мы хотим отображать каждый обработанный элемент на графический интерфейс или отправлять уведомление об этом на почту (сразу после обработки этого элемента).

СОВСЕМ ПЛОХОЕ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class ProcessorImpl implements Processor {
    @Override
    public void process(List<Data> datas) {
        for (Data data : datas) {
            ProcessedData result = doProcess(data);

            paint(result);
            send(result);
        }
    }

    private ProcessedData doProcess(Data data) {….}
    …
}
```

РЕШЕНИЕ ЧУТЬ-ЧУТЬ ЛУЧШЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class ProcessorImpl implements Processor {
    private final DataPainter dataPainter = new DataPainter();
    private final EmailSender sender = new EmailSender();

    @Override
    public void process(List<Data> datas) {
        for (Data data : datas) {
            ProcessedData result = doProcess(data);

            dataPainter.paint(result);
            sender.send(result);
        }
    }

    private ProcessedData doProcess(Data data) {….}

}
```

ХОРОШЕЕ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class ObserverableProcessor implements Processor {
    private final DataListener listener;

    public ObserverableProcessor(DataListener listener) {
        this.listener = listener;
    }

    @Override
    public void process(List<Data> datas) {
        for (Data data : datas) {
            ProcessedData result = doProcess(data);
            listener.onEvent(result);
        }
    }
```

ИЛИ ТАК

СБЕРБАНК ТЕХНОЛОГИИ

```
public class ObserverableProcessor implements Processor {
    @Override
    public void process(List<Data> datas,
                        DataListener listener) {
        for (Data data : datas) {
            ProcessedData result = doProcess(data);
            listener.onEvent(result);
        }
    }
```

Абстракция, которая реагирует на события

```
public interface DataListener {
    void onEvent(ProcessedData result);
}
```

РЕАЛИЗАЦИИ ЛИСТЕНЕРА

СБЕРБАНК ТЕХНОЛОГИИ

```
public class EmailListener implements DataListener {
    private final EmailSender emailSender;

    public EmailListener(EmailSender emailSender) {
        this.emailSender = emailSender;
    }

    @Override
    public void onEvent(ProcessedData result) {
        emailSender.send(result);
    }
}
```

РЕАЛИЗАЦИИ ЛИСТЕНЕРА

СБЕРБАНК ТЕХНОЛОГИИ

```
public class PainterListener implements DataListener {
    private final DataPainter dataPainter;

    public PainterListener(DataPainter dataPainter) {
        this.dataPainter = dataPainter;
    }

    @Override
    public void onEvent(ProcessedData result) {
        dataPainter.paint(result);
    }
}
```

JAVA8 ДЕЛАЕТ СОЗДАНИЕ ЛИСТЕНЕРОВ КОРОЧЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
public static DataListener asListener(EmailSender s) {
    return s::send;
}

public static DataListener asListener(DataPainter p) {
     return p::paint;
}
```

ИСПОЛЬЗОВАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
Processor p1 =
      new ObserverableProcessor(newEmailListener());


Processor p2 =
      new ObserverableProcessor(newPainterListener());
```

Что если надо передать несколько листенеров, а не один?

Что если надо передать несколько листенеров, а не один?

Использовать паттерн Composite

РАСШИРЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
Processor p3 = new ObserverableProcessor(
        composite(newEmailListener(), newPainterListener())
);
```

COMPOSITE LISTENER

СБЕРБАНК ТЕХНОЛОГИИ

```
public class CompositeListener implements DataListener {
    private final List<DataListener> listeners;

    public CompositeListener(List<DataListener> list) {
        this.listeners = listeners;
    }

    @Override
    public void onEvent(ProcessedData result) {
        for (DataListener listener : listeners) {
            listener.onEvent(result);
        }
    }
}
```

COMPOSITE LISTENER JAVA8

СБЕРБАНК ТЕХНОЛОГИИ

```
public DataListener composite(DataListener... list) {
    return data -> Stream.of(list)
            .forEach(listener -> listener.onEvent(data));
}
```

СБЕРБАНК
ТЕХНОЛОГИИ

CHAIN OF RESPONSIBILITY (ЦЕПОЧКА ОБЯЗАННОСТЕЙ)

Предназначенный для организации в системе уровней ответственности.

Сообщения в системе обрабатываются по схеме «обработай сам либо перешли другому», то есть одни сообщения обрабатываются на том уровне, где они получены, а другие пересылаются объектам иного уровня

Нам надо обрабатывать входящий запрос.
Изначально не понятно кто именно должен обработать запрос.

```
public interface Handler {
    Response handle(Request request);
}
```

В ЛОБ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class HandlerImpl implements Handler {
    @Override
    public Response handle(Request request) {
        log(request);
        if (!isAuthorized(request)) throw new SecurityException();
        if (isCached(request))  return getFromCache(request);
        if (...) ..

        ..
```

СБЕРБАНК ТЕХНОЛОГИИ

```
public class LogHandler implements Handler {
    private final Handler next;

    public LogHandler(Handler next) {
        this.next = next;
    }

    @Override
    public Response handle(Request request) {
        log(request);
        return next.handle(request);
    }
```

СБЕРБАНК ТЕХНОЛОГИИ

```
public class SecurityHandler implements Handler {
    private final Handler next;

    public SecurityHandler(Handler next) {
        this.next = next;
    }

    @Override
    public Response handle(Request request) {
        if (!isAuthorized(request))
                    throw new SecurityException();
        return next.handle(request);
    }
```

СБЕРБАНК ТЕХНОЛОГИИ

```
public class CacheHandler implements Handler {
    private final Handler next;

    public CacheHandler(Handler next) {
        this.next = next;
    }

    @Override
    public Response handle(Request request) {
        return isCached(request) ? getFromCache(request)
                                 : next.handle(request);

    }
```

ИСПОЛЬЗОВАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
Handler handler = new LogHandler(
                      new SecurityHandler(new CacheHandler()));
```

ITERATOR

Позволяет получить последовательный доступ к внутреннему состоянию класс, не раскрывая деталей реализации

ЧТО ПЛОХОГО В ЭТОМ КОДЕ?

СБЕРБАНК ТЕХНОЛОГИИ

```
for (int i = 0; i < list.size(); i++) {
    //some logic
}
```

ЧТО ПЛОХОГО В ЭТОМ КОДЕ?

СБЕРБАНК ТЕХНОЛОГИИ

```
for (int i = 0; i < list.size(); i++) {
    //some logic
}
```

Для LinkedList'а работает за $O(n^2)$

ЧТО ПЛОХОГО В ЭТОМ КОДЕ?

СБЕРБАНК ТЕХНОЛОГИИ

```
for (int i = 0; i < list.size(); i++) {
    //some logic
}
```

Для LinkedList'а работает за $O(n^2)$.

Мы хотим создать **общий** механизм оптимального обхода элементов коллекции, не **раскрывая деталей реализации**

СБЕРБАНК ТЕХНОЛОГИИ

```
Iterator<Integer> iterator = list.iterator();
while (iterator.hasNext()) {
   ..
}
```

Или

```
for (Integer integer : list) {
   ..
}
```

STRATEGY (СТРАТЕГИЯ)

Позволяет инкапсулировать алгоритм в отдельный класс, иметь возможность менять конкретную стратегию в рантайме.

- есть несколько родственных классов, которые отличаются поведением;
- необходимо иметь несколько вариантов поведения;
- в классе есть данные которых не должен знать клиент;
- с помощью условных операторов в классе определено большое количество поведений.

МНОГО IF'ОВ. СЛОЖНО ДОБАВЛЯТЬ НОВЫЕ ТИПЫ

СБЕРБАНК ТЕХНОЛОГИИ

```
class Bird {
    //...
    public double getSpeed() {
        switch (type) {
            case EUROPEAN:
                return getBaseSpeed();
            case AFRICAN:
                return getBaseSpeed() - getLoadFactor() *
                                        numberOfCoconuts;
            case NORWEGIAN_BLUE:
                return isNailed ? 0 : getBaseSpeed(voltage);
        }
        throw new RuntimeException("Should be unreachable");
    }

}
```

```
class Bird {
    private final BirdType type;

    public Bird(BridType type) {
       this.type = type;
    }

    public double getSpeed() {
        return type.getSpeed();
    }

    …

}
```

STRATEGY

СБЕРБАНК ТЕХНОЛОГИИ

```
class EuropeanType implements BirdType {
    public double getSpeed() {
        return 10;
    }
}
class African implements BirdType {
    public double getSpeed() {
        return getBaseSpeed() - getLoadFactor() *
                                        numberOfCoconuts;
    }
}
class NorwegianBlue implements BirdType {
    public double getSpeed() {
        return isNailed ? 0 : getBaseSpeed(voltage);
    }


}
```

STATE (СОСТОЯНИЕ)

Используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.

- поведение объекта зависит от его состояния и изменяется во время выполнения;
- когда встречается большое количество условных операторов, когда выбор ветви зависит от состояния.

ЗАДАЧА

СБЕРБАНК ТЕХНОЛОГИИ

У нас есть трактор, который может двигаться вперед и поворачивать на 90 градусов. Движение вперед зависит от текущего направления(поворота)

```
public class Tractor {
    private Position position = new Position(0, 0);
    private Orientation orientation = Orientation.NORTH;

    public void moveForwards() {
        if (orientation == Orientation.NORTH) {
            position = position.changeY(1);
        } else if (orientation == Orientation.EAST) {
            position = position.changeX(1);
        // еще if
        }

    public void turnClockwise() {
        if (orientation == Orientation.NORTH) {
            orientation = Orientation.EAST;
        } else if (orientation == Orientation.EAST) {
            orientation = Orientation.SOUTH;
        }   // еще if
    }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

```
public class StatedTractor {
    private Position position = new Position(0, 0);
    private Orientation orientation = Orientation.NORTH;

    public void moveForwards() {
        position = orientation.move(position);
    }

    public void turnClockwise() {
        orientation = orientation.turn();

    }
```

```
enum Orientation {
    NORTH {
        public Orientation turn() {return EAST;}
        public Position move(Position p) {return p.changeY(1); }
    },
    WEST {
        public Orientation turn() {return NORTH;}
        public Position move(Position p) {return p.changeX(-1);}
    },
    SOUTH {
        public Orientation turn() {return WEST;}
        public Position move(Position p) {return p.changeY(-1);}
    },
    EAST {
        public Orientation turn() {return SOUTH;}
        public Position move(Position p) {return p.changeX(1);}

    };
```

ЗАДАЧА

СБЕРБАНК ТЕХНОЛОГИИ

Вы пишите AI танка для пошаговой стратегии.
Поведение танка должно меняться в зависимости от внешних условий.

В ЛОБ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class TankAiImpl implements TankAi {
    @Override
    public void move(World world, List<Enemy> alive,int round) {
        if (round < 10) {
            //some start logic
        } else if (alive.size() > 2) {
            //attack code
        } else if (alive.size() == 1) {
            //duel mode code
        }
    }

}
```

ПОВЕДЕНИЕ ТАНКА ЗАВИСИТ ОТ СОСТОЯНИЯ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class StatedTankAi implements TankAi {
    private Strategy strategy = new StartedStrategy();

    @Override
    public void move(World world, List<Enemy> alive, int round) {
        strategy = strategy.move(world, alive, round);
    }

}
```

ПОВЕДЕНИЕ ТАНКА ЗАВИСИТ ОТ СОСТОЯНИЯ

СБЕРБАНК ТЕХНОЛОГИИ

```
public class StartedStrategy implements Strategy {
    @Override
    public Strategy move(World world, List<Enemy> alive, int round) {
        //hide logic
        return round == 10 ? new AttackStrategy() : this;
    }
}
```

```
public class AttackStrategy implements Strategy {
    @Override
    public Strategy move(World world, List<Enemy> alive, int round) {
        //attack logic
        return alive.size() == 1 ? new DuelStrategy() : this;
    }
}
```

COMMAND (КОМАНДА)

Действие инкапсулируется в объект.

Объект команды заключает в себе само действие и его параметры.

- параметризация объектов выполняемым действием;
- определять запрос, ставить его в очередь или выполнять его в разное время.

ЗАДАЧА

СБЕРБАНК ТЕХНОЛОГИИ

У нас есть текстовый редактор, который работает с документами.

Мы хотим иметь возможность развивать его, добавлять новые команды.

Уметь выполнять эти команды, и уметь отменять эти команды (redo cntl+z)

БАЗОВЫЙ КЛАСС

СБЕРБАНК ТЕХНОЛОГИИ

```
public interface Editor {
    void moveCursor(int line, int pos);

    void setTest(Color color);

    //...
}
```

```
public interface Editor {
    void moveCursor(int line, int pos);

    void setTest(Color color);

    //еще 500 методов
}
```

Можно добавлять в Editor новые методы и реализации этих методов.

Со временем он будет разрастаться и станет плохо поддерживаемым.

А как сделать отмену последних действий?

```
public interface Editor {
    void execute(Command command);

    void undo();

    void redo();
}
```

```
public interface Command {
    void execute(Document document);

    void undo(Document document);
}
```

COMMAND

СБЕРБАНК ТЕХНОЛОГИИ

```
public class EditorImpl implements Editor2 {
    private Document document = ..;
    private List<Command> last =..;

    @Override
    public void execute(Command command) {
        command.execute(document);
        last.add(command);
    }

    @Override
    public void undo() {
        Command last = last.remove(last.size() - 1);
        last.undo(document);

    }
```

COMMAND

СБЕРБАНК ТЕХНОЛОГИИ

```
class RemoveAllTextCommand implements Command {
    private String removed;

    @Override
    public void execute(Document document) {
        text = document.getText();
        document.setText('');

    }

    @Override
    public void undo(Document document) {
        document.setText(removed);
    }

}
```

СБЕРБАНК ТЕХНОЛОГИИ

```
class PastCommand implements Command {
//
}


class ChangeColorCommand implements Command {
    public ChangeColorCommand(Color color) {
      //
    }
//
}
```

COMMAND ПЛЮСЫ

СБЕРБАНК ТЕХНОЛОГИИ

Каждое действие описано в своем классе.

Легко добавлять новые действия не меняя существующие классы

Легко реализовать отмену действий.

ЗАКЛЮЧЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Паттерны позволяют развивать код путем добавления новых классов, а не добавлением логики в существующие классы.

Один класс должен делать ровно одно дело, и делать это хорошо