Рефакторирање на кодот

Design Pattern: **Strategy Pattern**

Што е Strategy Pattern?

Strategy Design Pattern е патерн за дизајн на однесување кој дозволува објект да избере однесување во текот на извршувањето. Тој дефинира семејство на алгоритми, ги енкапсулира секој од нив и ги прави меѓусебно заменливи. Шаблонот Strategy дозволува клиентот динамички да избере кој алгоритам ќе го користи, во зависност од контекстот.

Зошто да се имплементира Strategy Pattern?

Во контекстот на нашата апликација, има различни типови на предвидувања: **предвидувања засновани на сигнали** и **предвидувања засновани на вести**. Овие два метода на предвидување имаат различна логика, и примената на нив наизменично би можела да создаде пофлексибилен и одржлив систем.

Како е применет Strategy Pattern?

1. **Интерфејс PredictionStrategy**: Овој интерфејс дефинира заедничка метода (getPrediction()) која сите конкретни стратегии мора да ја имплементираат. Тој претставува семејството на алгоритми за предвидување.
2. **Конкретни стратегии**:
    - **SignalPredictionStrategy**:Го имплементира интерфејсот PredictionStrategy и обезбедува логика за сигнални предвидувања (на пример, пресметување на проценти за купување, продажба, задржување).
    - **NewsPredictionStrategy**:Го имплементира интерфејсот PredictionStrategy и ја обработува логиката за предвидувања засновани на вести (на пример, пресметување на броеви за сентимент и давање на препораки за акции).
3. **PredictionContext**: Оваа класа делува како контекст кој дозволува динамично избирање на стратегија. Контекстот се инјектира со соодветната стратегија (SignalPredictionStrategy или NewsPredictionStrategy), и ја пренесува на стратегијата методата getPrediction().
4. **Контролер**: Контролерот ги повикува резултатите од соодветната стратегија преку PredictionContext, што го поедноставува и прави пофлексибилен.

Рефакторирање на кодот со Strategy Pattern

- **PredictionContext**: Содржи мапа на стратегии и ја извршува избраната на основа на барањето.
- **Конкретни стратегии** (SignalPredictionStrategy, NewsPredictionStrategy): Овие класи ја енкапсулираат различната логика за предвидување.

- **Контролер**: Поедноставен со тоа што ја пренесува логиката на PredictionContext и дозволува динамично избирање на стратегија.

1. Интерфејс за стратегија (PredictionStrategy)

Ова е интерфејсот што сите стратегии ќе го имплементираат.

```
package mk.ukim.finki.mkstockexchange.stockexchangeapp.strategy;

import java.util.Map;

7 usages  2 implementations
public interface PredictionStrategy {
    1 usage  2 implementations
    Map<String, Object> getPrediction(String stockCode, String timeFrame);
}
```

2. Конкретни стратегии

Секој вид на предвидување (сигнал или вести) ќе биде енкапсулиран во своја конкретна стратегија.

SignalPredictionStrategy

```
package mk.ukim.finki.mkstockexchange.stockexchangeapp.strategy.impl;

import mk.ukim.finki.mkstockexchange.stockexchangeapp.model.Predictions;
import
mk.ukim.finki.mkstockexchange.stockexchangeapp.repository.PredictionReposit
ory;
import
mk.ukim.finki.mkstockexchange.stockexchangeapp.strategy.PredictionStrategy;
import org.springframework.stereotype.Component;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Component("signalPredictionStrategy")
public class SignalPredictionStrategy implements PredictionStrategy {

    private final PredictionRepository predictionRepository;

    public SignalPredictionStrategy(PredictionRepository
predictionRepository) {
        this.predictionRepository = predictionRepository;
    }
```

```
    @Override
    public Map<String, Object> getPrediction(String stockCode, String
timeFrame) {
        List<Predictions> predictions =
predictionRepository.findByCodeAndTimeframe(stockCode, timeFrame);

        if (predictions.isEmpty()) {
            return null;
        }

        Map<String, Long> signalCounts = predictions.stream()
                .collect(Collectors.groupingBy(Predictions::getSignal,
Collectors.counting()));

        long totalPredictions = predictions.size();

        Map<String, Object> result = new HashMap<>();
        result.put("buy", (signalCounts.getOrDefault("Buy", 0L) * 100.0) /
totalPredictions);
        result.put("sell", (signalCounts.getOrDefault("Sell", 0L) * 100.0)
/ totalPredictions);
        result.put("hold", (signalCounts.getOrDefault("Hold", 0L) * 100.0)
/ totalPredictions);

        return result;
    }
}
```

NewsPredictionStrategy

```
package mk.ukim.finki.mkstockexchange.stockexchangeapp.strategy.impl;

import
mk.ukim.finki.mkstockexchange.stockexchangeapp.model.NewsPredictions;
import
mk.ukim.finki.mkstockexchange.stockexchangeapp.repository.NewsPredictionsRe
pository;
import
mk.ukim.finki.mkstockexchange.stockexchangeapp.strategy.PredictionStrategy;
import org.springframework.stereotype.Component;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Component("newsPredictionStrategy")
public class NewsPredictionStrategy implements PredictionStrategy {

    private final NewsPredictionsRepository newsPredictionsRepository;

    public NewsPredictionStrategy(NewsPredictionsRepository
newsPredictionsRepository) {
        this.newsPredictionsRepository = newsPredictionsRepository;
    }

    @Override
    public Map<String, Object> getPrediction(String stockCode, String
timeFrame) {
        List<NewsPredictions> newsPredictions =
```

```
newsPredictionsRepository.findByCompanyName(stockCode);

        Map<String, Long> sentimentCounts = newsPredictions.stream()

.collect(Collectors.groupingBy(NewsPredictions::getSentiment,
Collectors.counting()));

        long positiveCount = sentimentCounts.getOrDefault("POSITIVE", 0L);
        long neutralCount = sentimentCounts.getOrDefault("NEUTRAL", 0L);
        long negativeCount = sentimentCounts.getOrDefault("NEGATIVE", 0L);

        String recommendation = getRecommendation(positiveCount,
negativeCount, neutralCount);

        Map<String, Object> result = new HashMap<>();
        result.put("company_name", stockCode);
        result.put("recommendation", recommendation);

        return result;
    }

    private String getRecommendation(long positiveCount, long
negativeCount, long neutralCount) {
        if (positiveCount > Math.max(negativeCount, neutralCount)) {
            return "The stock will rise";
        } else if (negativeCount > Math.max(positiveCount, neutralCount)) {
            return "The stock will fall";
        } else {
            return "The stock will remain stable";
        }
    }
}
```

3. Context (PredictionContext)

Ова е класа која ја управува стратегијата и ја извршува во зависност од контекстот.

```
package mk.ukim.finki.mkstockexchange.stockexchangeapp.strategy;

import org.springframework.stereotype.Component;

import java.util.Map;

@Component
public class PredictionContext {

    private final Map<String, PredictionStrategy> strategies;

    public PredictionContext(Map<String, PredictionStrategy> strategies) {
        this.strategies = strategies;
    }

    public Map<String, Object> executeStrategy(String strategyKey, String
stockCode, String timeFrame) {
        PredictionStrategy strategy = strategies.get(strategyKey);
        if (strategy == null) {
            throw new IllegalArgumentException("Strategy not found: " +
strategyKey);
        }
        return strategy.getPrediction(stockCode, timeFrame);
```

```
    }
}
```

4. Контролер (PredictionsController)

Контролерот сега само ја повикува стратегијата преку контекстот.

```
package mk.ukim.finki.mkstockexchange.stockexchangeapp.web;

import
mk.ukim.finki.mkstockexchange.stockexchangeapp.strategy.PredictionContext;
import org.springframework.web.bind.annotation.*;

import java.util.Map;

@CrossOrigin(origins = "http://localhost:3000")
@RestController
@RequestMapping("/api/predictions")
public class PredictionsController {

    private final PredictionContext predictionContext;

    public PredictionsController(PredictionContext predictionContext) {
        this.predictionContext = predictionContext;
    }

    @GetMapping("/signal-percentages")
    public Map<String, Object> getSignalPercentages(
            @RequestParam String stockCode,
            @RequestParam String timeFrame) {
        return
predictionContext.executeStrategy("signalPredictionStrategy", stockCode,
timeFrame);
    }

    @GetMapping("/news-prediction")
    public Map<String, Object> getNewsPrediction(@RequestParam String
stockCode) {
        return predictionContext.executeStrategy("newsPredictionStrategy",
stockCode, null);
    }
}
```

5. Објаснување на промените

1. **PredictionStrategy**: Интерфејсот за стратегија кој ја дефинира методата getPrediction(). Оваа методата се користи во сите стратегии.
2. **SignalPredictionStrategy & NewsPredictionStrategy**: Овие класи ги имплементираат конкретните стратегии, со различна логика за предвидување. На пример, SignalPredictionStrategy се фокусира на проценти на сигналите, додека NewsPredictionStrategy обработува сентимент на вестите.
3. **PredictionContext**: Класа која овозможува динамично променување на стратегијата. Контролерот ја поставува стратегијата (сигнал или вести) преку оваа класа.

4. **PredictionsController**: Контролерот е поедноставен бидејќи сега само ги повикува стратегиите преку контекстот, наместо да ги содржи самите имплементации на логиката.

Заклучок

Strategy Pattern помага во правењето на системот флексибилен преку одвојување на алгоритмите за предвидување од контролерот. Тоа ја зголемува способноста за скалирање со додавање нови стратегии во иднина, го намалува дуплирањето на кодот и го прави системот поодржлив.

Design Pattern: **MVC**

1. Модел:

- **Што претставува:**
  Моделот ги содржи податоците и бизнис логиката на апликацијата.
- **Одговорности:**
  - Чување и ажурирање на информации за акциите (име, симбол, цена).
  - Управување со трансакции (купување/продавање акции).
  - Комуникација со базата на податоци.

2. View:

- **Што претставува:**
  View е слојот кој го прикажува корисничкиот интерфејс. Тој ги покажува податоците од моделот и овозможува интеракција со корисникот.
  Stock Exchange App:
  - Листа на акции со нивните тековни цени.
  - Предвидувач за купување или продавање акции.
  - Графикони за промена на цените.
- **Одговорности:**
  - Презентирање на податоците во разбирлива форма.
  - Обезбедување визуелни елементи за корисничка интеракција.

3. Controller (Контролер):

- **Што претставува:**
  Контролерот е посредникот меѓу моделот и прегледот. Тој ги прима барањата од корисникот, ги обработува и враќа соодветен одговор.
- **Одговорности:**
  - Обработување на кориснички барања (на пр., приказ на листа на акции, извршување трансакции).
  - Верификација и обработка на податоци од корисникот пред да ги испрати до моделот.
  - Управување со логиката на префрлување податоци помеѓу моделот и прегледот.

4. Заклучок

Дизајн патернот **MVC (Model-View-Controller)** овозможува организирање на апликацијата во јасно дефинирани слоеви, што носи неколку клучни предности:

- **Разделување на одговорностите:** Моделот се грижи за податоците и бизнис логиката, прегледот за прикажување на корисничкиот интерфејс, а контролерот ја координира интеракцијата меѓу нив. Ова ја прави апликацијата полесна за одржување и развој.
- **Флексибилност:** Промените во корисничкиот интерфејс (View) можат да се направат без влијание на моделот или контролерот. На пример, може да се создадат различни интерфејси за мобилни и десктоп корисници.
- **Повторна употреба:** Логиката од моделот може повторно да се користи во различни делови од апликацијата, без да се дуплира код.

Во контекст на Stock Exchange App, MVC помага за:

- Управување со комплексната логика за акции, трансакции и финансиски податоци.
- Обезбедување на динамичен и интерактивен интерфејс за корисниците.

Со примената на MVC, апликацијата станува поорганизирана, скалабилна и лесна за проширување, што е особено важно за систем како Stock Exchange App, кој бара флексибилност и сигурност.