Дизајн и архитектура на софтвер

Домашна 4

Рефакторирање

Во овој документ ќе бидат објаснети имплементациите на неколку дизајн шаблони што се применети во развојот на нашата апликација. Искористени се повеќе шаблони, но сепак главниот фокус ќе биде ставен на примената на следниве шаблони:

1. **Strategy Pattern**: Користен за флексибилно менување на начинот на вчитување податоци.
2. **Template Method Pattern**: Применет за дефинирање на основна структура за вчитување податоци, со можност за специфична имплементација на одделни чекори.
3. **Singleton Pattern**: Искористен за да обезбедиме дека е создадена само една инстанца од класата, што е особено корисно, на пример, при работа со бази на податоци.
4. **Factory Pattern:** Дефинира интерфејс за креирање на објекти, но дозволува на подкласи да одлучат кој тип на објект ќе се искористи

Овој пристап овозможува лесно одржување и проширување на апликацијата. Преку објаснувањето на секој од дизајн шаблоните, ќе биде прикажано како тие придонесуваат за флексибилноста, одржливоста и разбирањето на системот.

Измените и дополнувањата беа насочени исклучиво кон деловите развиени со Python, поврзани со Analyzer и Scraper функционалностите. Овие компоненти беа надоградени за да обезбедат поефикасно собирање, обработка и анализа на податоци, притоа внимавајќи на примената на релевантни дизајн шаблони за овие модули.

Со овој пристап и користењето на соодветните шаблони, обезбедена е конзистентност во развојот на целиот систем, при што основниот бекенд и неговите дизајн решенија останаа недопрени, а дополнителните функционалности беа развиени како независни, но комплементарни компоненти.

I. Backend – Spring Boot

Бекенд делот на нашата апликација, развиен со користење на Spring Boot рамката, е веќе дизајниран и имплементиран следејќи го **Model-View-Controller (MVC)** дизајн патернот. Кодот од бекендот остана непроменет, со што се зачувaа оригиналните принципи и интегритетот на имплементацијата.

II. Analyzer

i. DataStorage

- Singleton Pattern (DatabaseConfig):
  - Обезбедува единствена инстанца за конфигурација на базата на податоци
  - Управува со поставките за поврзување со базата на податоци
- Strategy Pattern (DatabaseOperation):
  - Дефинира апстрактен интерфејс за операции со база на податоци
  - Овозможува имплементација на различни бази на податоци
  - PostgresOperation ја спроведува конкретната стратегија
- Factory Pattern (DatabaseOperationFactory):
  - Создава соодветни инстанци за работа со база на податоци
  - Овозможува лесно додавање на нови типови на бази на податоци
- Facade Pattern (DataStorage):
  - Го поедноставува сложениот потсистем за клиентите
  - Се справува со целото управување со грешки
- Template Method Pattern:
  - Основната апстрактна класа ја дефинира структурата на операцијата
  - Конкретните часови спроведуваат специфични однесувања

```
import os
from datetime import datetime
from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Any, Tuple
import psycopg2
from psycopg2.extensions import connection

# Singleton Pattern for Database Configuration
class DatabaseConfig:
    _instance = None

    def __new__(cls, db_url: Optional[str] = None):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
```

```python
            cls._instance.db_url = db_url or os.getenv('DB_URL',
"postgresql://dians:dians123@localhost:9555/diansdb")
        return cls._instance

    def get_connection(self) -> connection:
        return psycopg2.connect(self.db_url)

# Strategy Pattern for Database Operations
class DatabaseOperation(ABC):
    @abstractmethod
    def initialize_tables(self, conn: connection) -> None:
        pass

    @abstractmethod
    def load_issuer_dates(self, conn: connection) -> Dict[str, datetime]:
        pass

    @abstractmethod
    def update_issuer(self, conn: connection, issuer: str, last_date:
Optional[datetime]) -> None:
        pass

    @abstractmethod
    def get_issuer_date(self, conn: connection, issuer: str) ->
Optional[datetime]:
        pass

    @abstractmethod
    def save_issuer_data(self, conn: connection, data_rows: List[Tuple]) -> None:
        pass

    @abstractmethod
    def get_all_data(self, conn: connection) -> List[Tuple]:
        pass

    @abstractmethod
    def count_issuer_data_rows(self, conn: connection) -> int:
        pass

    @abstractmethod
    def get_by_issuer(self, conn: connection, issuer: str) -> List[Tuple]:
        pass

# Concrete Strategy for PostgreSQL
class PostgresOperation(DatabaseOperation):
```

```
    def initialize_tables(self, conn: connection) -> None:
        with conn.cursor() as cursor:
            cursor.execute("""
                CREATE TABLE IF NOT EXISTS issuer_dates (
                    issuer TEXT PRIMARY KEY,
                    last_date DATE
                )
            """)
            cursor.execute("""
                CREATE TABLE IF NOT EXISTS issuer_data (
                    date DATE,
                    issuer TEXT,
                    avg_price TEXT,
                    last_trade_price TEXT,
                    max_price TEXT,
                    min_price TEXT,
                    percent_change TEXT,
                    turnover_best TEXT,
                    total_turnover TEXT,
                    volume TEXT,
                    PRIMARY KEY (date, issuer)
                )
            """)
            conn.commit()

    def load_issuer_dates(self, conn: connection) -> Dict[str, datetime]:
        with conn.cursor() as cursor:
            cursor.execute("SELECT issuer, last_date FROM issuer_dates")
            return {row[0]: row[1] for row in cursor.fetchall()}

    def update_issuer(self, conn: connection, issuer: str, last_date:
Optional[datetime]) -> None:
        with conn.cursor() as cursor:
            date_str = last_date.strftime('%Y-%m-%d') if last_date else None
            cursor.execute("""
                INSERT INTO issuer_dates (issuer, last_date)
                VALUES (%s, %s)
                ON CONFLICT (issuer) DO UPDATE
                SET last_date = EXCLUDED.last_date
            """, (issuer, date_str))
            conn.commit()

    def get_issuer_date(self, conn: connection, issuer: str) ->
Optional[datetime]:
        with conn.cursor() as cursor:
```

```
            cursor.execute("""
                SELECT last_date FROM issuer_dates
                WHERE issuer = %s
            """, (issuer,))
            row = cursor.fetchone()
            return row[0] if row and row[0] else None

    def save_issuer_data(self, conn: connection, data_rows: List[Tuple]) -> None:
        with conn.cursor() as cursor:
            formatted_rows = [
                (datetime.strptime(row[0], "%d.%m.%Y").strftime("%Y-%m-%d"),
*row[1:])
                for row in data_rows
            ]
            cursor.executemany("""
                INSERT INTO issuer_data (
                    date, issuer, avg_price, last_trade_price, max_price,
min_price,
                    percent_change, turnover_best, total_turnover, volume
                ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
                ON CONFLICT (date, issuer) DO NOTHING
            """, formatted_rows)
            conn.commit()

    def get_all_data(self, conn: connection) -> List[Tuple]:
        with conn.cursor() as cursor:
            cursor.execute("SELECT * FROM issuer_data")
            return cursor.fetchall()

    def count_issuer_data_rows(self, conn: connection) -> int:
        with conn.cursor() as cursor:
            cursor.execute("SELECT COUNT(*) FROM issuer_data")
            return cursor.fetchone()[0]

    def get_by_issuer(self, conn: connection, issuer: str) -> List[Tuple]:
        with conn.cursor() as cursor:
            cursor.execute("""
                SELECT * FROM issuer_data WHERE issuer = %s
            """, (issuer,))
            return cursor.fetchall()

# Factory Pattern for Database Operations
class DatabaseOperationFactory:
    @staticmethod
    def create_operation(db_type: str = "postgres") -> DatabaseOperation:
```

```python
        if db_type.lower() == "postgres":
            return PostgresOperation()
        raise ValueError(f"Unsupported database type: {db_type}")

# Facade Pattern
class DataStorage:
    def __init__(self, db_url: Optional[str] = None):
        self.db_config = DatabaseConfig(db_url)
        self.db_operation = DatabaseOperationFactory.create_operation("postgres")
        self._initialize_db()

    def _initialize_db(self) -> None:
        try:
            with self.db_config.get_connection() as conn:
                self.db_operation.initialize_tables(conn)
        except psycopg2.Error as e:
            print(f"Error initializing database: {e}")

    def load_data(self) -> Dict[str, datetime]:
        try:
            with self.db_config.get_connection() as conn:
                return self.db_operation.load_issuer_dates(conn)
        except psycopg2.Error:
            return {}

    def update_issuer(self, issuer: str, last_date: Optional[datetime]) -> None:
        try:
            with self.db_config.get_connection() as conn:
                self.db_operation.update_issuer(conn, issuer, last_date)
        except psycopg2.Error as e:
            print(f"Error updating issuer: {e}")

    def get_issuer_date(self, issuer: str) -> Optional[datetime]:
        try:
            with self.db_config.get_connection() as conn:
                return self.db_operation.get_issuer_date(conn, issuer)
        except psycopg2.Error as e:
            print(f"Error retrieving issuer date: {e}")
            return None

    def save_issuer_data(self, data_rows: List[Tuple]) -> None:
        try:
            with self.db_config.get_connection() as conn:
                self.db_operation.save_issuer_data(conn, data_rows)
        except ValueError as ve:
```

```python
            print(f"Date format error: {ve}")
        except psycopg2.Error as e:
            print(f"Error saving issuer data: {e}")

    def get_all_data(self) -> List[Tuple]:
        try:
            with self.db_config.get_connection() as conn:
                return self.db_operation.get_all_data(conn)
        except psycopg2.Error as e:
            print(f"Error retrieving data: {e}")
            return []

    def count_issuer_data_rows(self) -> int:
        try:
            with self.db_config.get_connection() as conn:
                return self.db_operation.count_issuer_data_rows(conn)
        except psycopg2.Error as e:
            print(f"Error counting rows: {e}")
            return 0

    def get_by_issuer(self, issuer: str) -> List[Tuple]:
        try:
            with self.db_config.get_connection() as conn:
                return self.db_operation.get_by_issuer(conn, issuer)
        except psycopg2.Error as e:
            print(f"Error retrieving data for issuer {issuer}: {e}")
            return []
```

ii. lstm

- **Strategy Pattern (DataPreprocessor):**
  - Дефинира апстрактен интерфејс за предобработка на податоци.
  - Овозможува имплементација на различни стратегии за обработка на податоци.
    - **DefaultDataPreprocessor** е конкретна имплементација на стратегијата која врши обработка и креирање технички индикатори и подготовка на податоците за LSTM модели.

- **Factory Pattern (ModelBuilder):**
  - Создава и конфигурира соодветни инстанци на LSTM модели со однапред дефинирани архитектури.
  - Обезбедува централизирано место за додавање нови архитектури или промени во постојните модели без промена на главната логика.

- Ги враќа и потребните callbacks за обука на моделот, како што се **EarlyStopping** и **ReduceLROnPlateau**.

- **Facade Pattern (LSTMAnalyzer):**
  - Ја упростува работата со сложениот подсистем кој вклучува предобработка на податоци, обука, зачувување модели и предвидување.
  - Обезбедува унифициран интерфејс за корисникот преку методи како train, predict_next_days, и perform_prediction.
  - Се справува со управување на модели, вчитување и зачувување на параметри, како и работа со податоци од различни извори.

```
from abc import ABC, abstractmethod
from typing import List, Dict, Any, Tuple, Optional
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import LSTM, Dense, Input, Dropout,
BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.regularizers import l1_l2
import matplotlib.pyplot as plt
from DataStorage import DataStorage
from LSTMModelStorage import ModelStorage
import category_encoders as ce

# Strategy Pattern for Data Preprocessing
class DataPreprocessor(ABC):
    @abstractmethod
    def prepare_data(self, data: pd.DataFrame) -> pd.DataFrame:
        pass

    @abstractmethod
    def create_feature_matrix(self, data: pd.DataFrame, training: bool) ->
Tuple[np.ndarray, np.ndarray]:
        pass

    @abstractmethod
    def prepare_sequences(self, data: pd.DataFrame, training: bool) ->
Tuple[np.ndarray, np.ndarray]:
        pass

class DefaultDataPreprocessor(DataPreprocessor):
    def __init__(self, config: Dict[str, Any]):
```

```
        self.columns = config['columns']
        self.numeric_cols = config['numeric_cols']
        self.price_features = config['price_features']
        self.volume_features = config['volume_features']
        self.tech_features = config['tech_features']
        self.n_lags = config['n_lags']

        self.price_scaler = StandardScaler()
        self.volume_scaler = StandardScaler()
        self.binary_encoder = ce.BinaryEncoder()

        self.feature_dims = None
        self.price_dims = len(self.price_features)
        self.volume_dims = len(self.volume_features)
        self.tech_dims = len(self.tech_features)
        self.issuer_dims = -1

    def prepare_data(self, data: pd.DataFrame) -> pd.DataFrame:
        df = pd.DataFrame(data, columns=self.columns) if not isinstance(data,
pd.DataFrame) else data.copy()

        # Original data preparation logic
        for col in self.numeric_cols:
            df[col] = df[col].str.replace('.', '', regex=False).str.replace(',',
'.', regex=False)
            df[col] = pd.to_numeric(df[col], errors='coerce')

        df['Date'] = pd.to_datetime(df['Date'])
        df = df.sort_values(['Date', 'Issuer'], ascending=True)
        df['unique_index'] = df['Date'].astype(str) + '_' + df['Issuer']
        df.set_index('unique_index', inplace=True)

        for issuer in df['Issuer'].unique():
            mask = df['Issuer'] == issuer
            # Technical indicators calculation
            df.loc[mask, 'MA5'] = df.loc[mask, 'Close'].rolling(window=5).mean()
            df.loc[mask, 'MA20'] = df.loc[mask,
'Close'].rolling(window=20).mean()
            df.loc[mask, 'Price_to_MA5'] = df.loc[mask, 'Close'] / df.loc[mask,
'MA5'] - 1
            df.loc[mask, 'Price_to_MA20'] = df.loc[mask, 'Close'] / df.loc[mask,
'MA20'] - 1

            # RSI calculation
            delta = df.loc[mask, 'Close'].diff()
```

```
            gain = (delta.where(delta > 0, 0)).rolling(window=10).mean()
            loss = (-delta.where(delta < 0, 0)).rolling(window=10).mean()
            rs = gain / loss
            df.loc[mask, 'RSI'] = 100 - (100 / (1 + rs))

            # MACD calculation
            exp1 = df.loc[mask, 'Close'].ewm(span=12, adjust=False).mean()
            exp2 = df.loc[mask, 'Close'].ewm(span=26, adjust=False).mean()
            df.loc[mask, 'MACD'] = exp1 - exp2

        return df.dropna()

    def create_feature_matrix(self, data: pd.DataFrame, training: bool) ->
Tuple[np.ndarray, np.ndarray]:
        # Original feature matrix creation logic
        for col in self.price_features[1:]:
            data[f'rel_{col}'] = data[col] / data['Close'] - 1

        if training:
            price_scaled =
self.price_scaler.fit_transform(data[self.price_features])
            volume_scaled =
self.volume_scaler.fit_transform(data[self.volume_features])
            encoded_issuer = self.binary_encoder.fit_transform(data[['Issuer']])
        else:
            price_scaled = self.price_scaler.transform(data[self.price_features])
            volume_scaled =
self.volume_scaler.transform(data[self.volume_features])
            encoded_issuer = self.binary_encoder.transform(data[['Issuer']])

        feature_matrix = np.hstack([
            price_scaled,
            volume_scaled,
            data[self.tech_features].values,
            encoded_issuer.values
        ])

        if training:
            self.feature_dims = feature_matrix.shape[1]
            self.issuer_dims = encoded_issuer.shape[1]

        return feature_matrix, price_scaled[:, 0]

    def prepare_sequences(self, data: pd.DataFrame, training: bool) ->
Tuple[np.ndarray, np.ndarray]:
```

```
        feature_matrix, targets = self.create_feature_matrix(data, training)
        return self._create_sequences(feature_matrix, targets,
data['Issuer'].unique(), training)

    def _create_sequences(self, feature_matrix: np.ndarray, targets: np.ndarray,
                         unique_issuers: np.ndarray, training: bool) ->
Tuple[np.ndarray, np.ndarray]:
        X, y = [], []
        for issuer in unique_issuers:
            issuer_mask = data['Issuer'] == issuer
            issuer_data = feature_matrix[issuer_mask]
            issuer_targets = targets[issuer_mask]

            if training:
                returns = np.diff(issuer_targets) / issuer_targets[:-1]
                for i in range(len(issuer_data) - self.n_lags - 1):
                    X.append(issuer_data[i:i + self.n_lags])
                    y.append(returns[i + self.n_lags])
            else:
                for i in range(len(issuer_data) - self.n_lags):
                    X.append(issuer_data[i:i + self.n_lags])
                    if i + self.n_lags < len(issuer_targets):
                        y.append(issuer_targets[i + self.n_lags])
                    else:
                        y.append(np.nan)

        return np.array(X), np.array(y)

# Factory Pattern for Model Building
class ModelBuilder:
    @staticmethod
    def build_model(input_shape: Tuple[int, int]) -> Model:
        model = Sequential([
            Input(shape=input_shape),
            LSTM(128, activation='tanh', return_sequences=True,
                 kernel_regularizer=l1_l2(l1=1e-6, l2=1e-5)),
            BatchNormalization(),
            Dropout(0.3),
            LSTM(64, activation='tanh',
                 kernel_regularizer=l1_l2(l1=1e-6, l2=1e-5)),
            BatchNormalization(),
            Dropout(0.3),
            Dense(32, activation='relu'),
            BatchNormalization(),
            Dense(1, activation='tanh')
```

```
        ])

        model.compile(optimizer='adam',
                     loss='huber',
                     metrics=['mae', 'mse'])
        return model

    @staticmethod
    def get_callbacks() -> List:
        return [
            EarlyStopping(monitor='val_loss',
                         patience=5,
                         restore_best_weights=True),
            ReduceLROnPlateau(monitor='val_loss',
                             factor=0.5,
                             patience=3,
                             min_lr=1e-6)
        ]

# Main Class with Facade Pattern
class LSTMAnalyzer:
    def __init__(self):
        # Configuration dictionary
        self.config = {
            'n_lags': 20,
            'columns': [
                'Date', 'Issuer', 'Avg Price', 'Close', 'High', 'Low', '%chg.',
                'Total turnover in denars', 'Turnover in BEST in denars',
'Volume'
            ],
            'numeric_cols': ['Close', 'High', 'Low', 'Avg Price', '%chg.',
                           'Turnover in BEST in denars', 'Total turnover in
denars', 'Volume'],
            'price_features': ['Close', 'High', 'Low', 'Avg Price', 'MA5',
'MA20'],
            'volume_features': ['Volume', 'Turnover in BEST in denars', 'Total
turnover in denars'],
            'tech_features': ['RSI', 'MACD', 'Price_to_MA5', 'Price_to_MA20']
        }

        self.preprocessor = DefaultDataPreprocessor(self.config)
        self.model = None
        self.model_storage = ModelStorage()
        self.data_storage = DataStorage()
```

```
    def load_model(self, price_scaler, volume_scaler, encoder, model, enc_len):
        self.preprocessor.price_scaler = price_scaler
        self.preprocessor.volume_scaler = volume_scaler
        self.preprocessor.binary_encoder = encoder
        self.model = model
        self.preprocessor.issuer_dims = enc_len

    def train(self, data, validation_split=0.2, epochs=30):
        df = self.preprocessor.prepare_data(data)
        df['target_return'] = df.groupby('Issuer')['Close'].pct_change()
        df = df[abs(df['target_return']) < 0.1]

        X, y = self.preprocessor.prepare_sequences(df, training=True)

        split_idx = int(len(X) * (1 - validation_split))
        X_train, X_val = X[:split_idx], X[split_idx:]
        y_train, y_val = y[:split_idx], y[split_idx:]

        time_weights = np.linspace(0.5, 1.0, len(X_train))

        self.model = ModelBuilder.build_model((X_train.shape[1],
X_train.shape[2]))

        history = self.model.fit(
            X_train, y_train,
            validation_data=(X_val, y_val),
            sample_weight=time_weights,
            epochs=epochs,
            batch_size=32,
            callbacks=ModelBuilder.get_callbacks(),
            verbose=1
        )

        self._save_trained_model()
        return history

    def _save_trained_model(self):
        additional_params = {
            'n_lags': self.config['n_lags'],
            'training_date': '2024-12-20',
            'model_version': '1.1',
            'enc_len': self.preprocessor.issuer_dims
        }

        self.model_storage.save_model(
```

```python
            self.model,
            self.preprocessor.volume_scaler,
            self.preprocessor.price_scaler,
            self.preprocessor.binary_encoder,
            model_name='stock_model_good',
            additional_params=additional_params
        )

    def predict_next_days(self, data, days=5):
        df = self.preprocessor.prepare_data(data)
        X, _ = self.preprocessor.prepare_sequences(df, training=False)

        predictor = PricePrediction(
            self.model,
            df,
            X[-1],
            self.preprocessor
        )
        return predictor.predict(days)

    def data_for_plotting(self, data, days=5):
        df = self.preprocessor.prepare_data(data)
        predictions, signal = self.predict_next_days(data, days)

        plot_data = PlotDataFormatter(df, predictions, signal, days)
        return plot_data.format()

    def perform_prediction(self, issuer, days=5):
        model_params = self.model_storage.load_model("stock_model_good")
        self.load_model(*model_params[:-1], model_params[-1]['enc_len'])

        data = self.data_storage.get_by_issuer(issuer)
        if len(data) < 100:
            print(f"Insufficient data for {issuer}")
            return

        return self.data_for_plotting(data, days)

# Helper class for price prediction
class PricePrediction:
    def __init__(self, model, df, last_sequence, preprocessor):
        self.model = model
        self.df = df
        self.last_sequence = last_sequence
        self.preprocessor = preprocessor
```

```
        self.last_known_price = df['Close'].iloc[-1]
        self._init_market_conditions()

    def _init_market_conditions(self):
        recent_prices = self.df['Close'].tail(20)
        recent_returns = recent_prices.pct_change().dropna()

        self.volatility = recent_returns.std()
        self.daily_volatility = self.volatility / np.sqrt(252)

        self.short_ma = recent_prices.tail(5).mean()
        self.long_ma = recent_prices.mean()
        self.trend_strength = (self.short_ma / self.long_ma - 1) * 100

        self.momentum = recent_returns.mean()

    def predict(self, days):
        predictions = []
        running_price = self.last_known_price
        running_momentum = self.momentum
        sequence = self.last_sequence.copy()

        for day in range(days):
            price, sequence, running_momentum = self._predict_single_day(
                day, sequence, running_price, running_momentum
            )
            predictions.append(price)
            running_price = price

        return self._format_predictions(predictions)

    def _predict_single_day(self, day, sequence, running_price,
running_momentum):
        predicted_return = self.model.predict(sequence[np.newaxis, :, :],
verbose=0)[0][0]

        blended_return = self._blend_predictions(
            predicted_return, day, running_momentum
        )

        next_price = self._calculate_next_price(
            running_price, blended_return
        )

        new_sequence = self._update_sequence(sequence, next_price)
```

```
        new_momentum = 0.7 * running_momentum + 0.3 * blended_return

        return next_price, new_sequence, new_momentum

    def _blend_predictions(self, predicted_return, day, running_momentum):
        random_factor = np.random.normal(0, self.daily_volatility)
        time_decay = np.exp(-0.2 * day)
```

iii. LSTMModelStorage

- **Singleton Pattern**:
    - ModelStorage сега е singleton, обезбедувајќи постоење на само една инстанца
    - Имплементиран со користење на методот __new__
- **Strategy Pattern**:
    - Воведена е основна апстрактна класа StorageStrategy
    - Имплементиран FileSystemStorageStrategy како конкретна стратегија
    - Овозможува лесно додавање на нови методи за складирање (cloud storage, database)
- **Factory Method Pattern**:
    - Додаден е статичен метод create_storage
    - Овозможува создавање на различни типови складирање додека се одржува singleton шемата

```
import os
import joblib
from tensorflow.keras.models import load_model
import json
from abc import ABC, abstractmethod
from typing import Any, Dict, List, Tuple, Optional

# Singleton Pattern for ModelStorage
class ModelStorage:
    _instance = None
```

```python
    def __new__(cls, base_path='models'):
        if cls._instance is None:
            cls._instance = super(ModelStorage, cls).__new__(cls)
            cls._instance._initialized = False
        return cls._instance

    def __init__(self, base_path='models'):
        if not self._initialized:
            self.base_path = base_path
            self._storage_strategy = FileSystemStorageStrategy()
            os.makedirs(base_path, exist_ok=True)
            self._initialized = True

    # Factory Method Pattern
    @staticmethod
    def create_storage(storage_type: str = 'filesystem', base_path: str =
'models') -> 'ModelStorage':
        storage = ModelStorage(base_path)
        if storage_type == 'filesystem':
            storage._storage_strategy = FileSystemStorageStrategy()
        # Can add more storage types here (e.g., cloud, database)
        return storage

    def save_model(self, model: Any, volume_scaler: Any, price_scaler: Any,
                  binary_encoder: Any, model_name: str = 'stock_prediction',
                  additional_params: Optional[Dict] = None) -> None:
        """Maintains the same interface but delegates to strategy"""
        self._storage_strategy.save_model(
            self.base_path,
            model,
            volume_scaler,
            price_scaler,
            binary_encoder,
            model_name,
            additional_params
        )

    def load_model(self, model_name: str = 'stock_prediction') -> Tuple:
        """Maintains the same interface but delegates to strategy"""
        return self._storage_strategy.load_model(self.base_path, model_name)

    def list_saved_models(self) -> List[str]:
        """Maintains the same interface but delegates to strategy"""
        return self._storage_strategy.list_saved_models(self.base_path)
```

```python
# Strategy Pattern
class StorageStrategy(ABC):
    @abstractmethod
    def save_model(self, base_path: str, model: Any, volume_scaler: Any,
                  price_scaler: Any, binary_encoder: Any, model_name: str,
                  additional_params: Optional[Dict]) -> None:
        pass

    @abstractmethod
    def load_model(self, base_path: str, model_name: str) -> Tuple:
        pass

    @abstractmethod
    def list_saved_models(self, base_path: str) -> List[str]:
        pass

class FileSystemStorageStrategy(StorageStrategy):
    def save_model(self, base_path: str, model: Any, volume_scaler: Any,
                  price_scaler: Any, binary_encoder: Any, model_name: str,
                  additional_params: Optional[Dict]) -> None:
        model_dir = os.path.join(base_path, model_name)
        os.makedirs(model_dir, exist_ok=True)

        # Save model
        model_path = os.path.join(model_dir, 'model.keras')
        model.save(model_path)

        # Save scalers and encoder
        joblib.dump(price_scaler, os.path.join(model_dir, 'price_scaler.pkl'))
        joblib.dump(volume_scaler, os.path.join(model_dir, 'volume_scaler.pkl'))
        joblib.dump(binary_encoder, os.path.join(model_dir, 'encoder.pkl'))

        # Save additional parameters
        if additional_params:
            params_path = os.path.join(model_dir, 'params.json')
            with open(params_path, 'w') as f:
                json.dump(additional_params, f)

    def load_model(self, base_path: str, model_name: str) -> Tuple:
        model_dir = os.path.join(base_path, model_name)

        # Load model
        model = load_model(os.path.join(model_dir, 'model.keras'))
```

```
        # Load scalers and encoder
        price_scaler = joblib.load(os.path.join(model_dir, 'price_scaler.pkl'))
        volume_scaler = joblib.load(os.path.join(model_dir, 'volume_scaler.pkl'))
        binary_encoder = joblib.load(os.path.join(model_dir, 'encoder.pkl'))

        # Load additional parameters
        additional_params = None
        params_path = os.path.join(model_dir, 'params.json')
        if os.path.exists(params_path):
            with open(params_path, 'r') as f:
                additional_params = json.load(f)

        return price_scaler, volume_scaler, binary_encoder, model,
additional_params

    def list_saved_models(self, base_path: str) -> List[str]:
        if not os.path.exists(base_path):
            return []
        return [d for d in os.listdir(base_path)
                if os.path.isdir(os.path.join(base_path, d))]
```

iv. technical_analysis

- **Singleton Pattern**:
    - Применет во класата TechnicalAnalyzer за да се осигура дека постои само една инстанца
    - Корисно за управување со споделени ресурси и одржување конзистентна состојба
- **Factory Pattern**:
    - Креирана е IndicatorFactory со специјализирани фабрики за различни типови индикатори
    - OscillatorFactory и MovingAverageFactory се справуваат со креирање на нивните соодветни индикатори
    - Ја одделува логиката за создавање индикатор од главната класа на анализаторот
- **Strategy Pattern**:
    - Имплементиран за генерирање сигнал со интерфејсот SignalStrategy
    - Одделни стратегии за осцилатори, moving averages и MACD

    ○ Го олеснува менувањето или додавањето нови алгоритми за генерирање сигнали

- **Single Responsibility Principle**:
    ○ Одвоено претпроцесирање на податоци во сопствена класа на DataPreprocessor
    ○ Секоја класа има единствена, добро дефинирана одговорност
- **Interface Segregation**:
    ○ Создадени посебни интерфејси за различни типови индикатори
    ○ Овозможува полесно проширување и модификација на типовите на индикатори

```python
from abc import ABC, abstractmethod
import pandas as pd

class SignalStrategy(ABC):
    @abstractmethod
    def generate_signal(self, data, **kwargs):
        pass

class OscillatorSignalStrategy(SignalStrategy):
    def __init__(self, thresholds):
        self.thresholds = thresholds

    def generate_signal(self, value, indicator_name):
        if pd.isna(value):
            return 'Hold'

        thresholds = self.thresholds[indicator_name]
        if value <= thresholds['buy']:
            return 'Buy'
        elif value >= thresholds['sell']:
            return 'Sell'
        return 'Hold'

class MovingAverageSignalStrategy(SignalStrategy):
    def generate_signal(self, data, price_col, ma_col):
        signals = pd.Series('Hold', index=data.index)
        buffer = data[ma_col] * 0.01
        signals[data[price_col] > (data[ma_col] + buffer)] = 'Buy'
```

```python
        signals[data[price_col] < (data[ma_col] - buffer)] = 'Sell'
        return signals

class MACDSignalStrategy(SignalStrategy):
    def generate_signal(self, row):
        macd = row['MACD']
        macd_signal = row['MACD_Signal']

        if pd.isna(macd) or pd.isna(macd_signal):
            return 'Hold'

        threshold = abs(macd_signal) * 0.15
        if macd > (macd_signal + threshold):
            return 'Buy'
        elif macd < (macd_signal - threshold):
            return 'Sell'
        return 'Hold'


class DataPreprocessor:
    @staticmethod
    def preprocess_data(data):
        numeric_cols = [
            'Close', 'High', 'Low', 'Avg. Price', '%chg.',
            'Turnover in BEST in denars', 'Total turnover in denars', 'Volume'
        ]

        for col in numeric_cols:
            data[col] = (
                data[col]
                .str.replace('.', '', regex=False)
                .str.replace(',', '.', regex=False)
                .astype(float)
            )

        data['Volume'] = data['Volume'].astype(int)
        data['Date'] = pd.to_datetime(data['Date'])
        data = data.sort_values('Date', ascending=True)
        data.set_index('Date', inplace=True)
        return data


class TechnicalAnalyzer:
    _instance = None
```

```python
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(TechnicalAnalyzer, cls).__new__(cls)
            cls._instance._initialized = False
        return cls._instance

    def __init__(self):
        if self._initialized:
            return

        self.storage = DataStorage()
        self.thresholds = {
            'RSI': {'buy': 25, 'sell': 75},
            'Stoch_%K': {'buy': 15, 'sell': 85},
            'Williams_R': {'buy': -85, 'sell': -15},
            'PPO': {'buy': -1.5, 'sell': 1.5},
            'ROC': {'buy': -3, 'sell': 3},
            'CCI': {'buy': -150, 'sell': 150}
        }

        self.oscillator_factory = OscillatorFactory()
        self.ma_factory = MovingAverageFactory()
        self.oscillator_strategy = OscillatorSignalStrategy(self.thresholds)
        self.ma_strategy = MovingAverageSignalStrategy()
        self.macd_strategy = MACDSignalStrategy()
        self._initialized = True

    def generate_oscillator_signal(self, row, indicator_name):
        return self.oscillator_strategy.generate_signal(row[indicator_name],
indicator_name)

    def generate_moving_average_signal(self, data, price_col, ma_col):
        return self.ma_strategy.generate_signal(data, price_col, ma_col)

    def generate_macd_signal(self, row):
        return self.macd_strategy.generate_signal(row)

    def preprocess_data(self, data):
        return DataPreprocessor.preprocess_data(data)

    def compute_indicators(self, data):
        oscillators = self.oscillator_factory.create_indicator(data)
        moving_averages = self.ma_factory.create_indicator(data)

        for indicator_name, values in oscillators.items():
```

```
            data[indicator_name] = values

        for indicator_name, values in moving_averages.items():
            data[indicator_name] = values

        return data

    def analyze_stock(self, issuer):
        db = self.storage.get_by_issuer(issuer)
        columns = [
            'Date', 'Issuer', 'Avg. Price', 'Close', 'High', 'Low', '%chg.',
            'Total turnover in denars', 'Turnover in BEST in denars', 'Volume'
        ]
        data = pd.DataFrame(db, columns=columns)

        data = self.preprocess_data(data)
        data = self.compute_indicators(data)

        oscillator_indicators = ['RSI', 'Stoch_%K', 'Williams_R', 'PPO', 'ROC',
'CCI']
        for indicator in oscillator_indicators:
            data[f'{indicator}_Signal'] = data.apply(
                self.generate_oscillator_signal, indicator_name=indicator, axis=1
            )

        ma_indicators = [('Close', 'SMA_20'), ('Close', 'EMA_20'),
                         ('Close', 'WMA_20'), ('Close', 'TRIX')]
        for price_col, ma_col in ma_indicators:
            data[f'{ma_col}_Signal'] = self.generate_moving_average_signal(data,
price_col, ma_col)

        data['MACD_Signal'] = data.apply(self.generate_macd_signal, axis=1)

        data_weekly = data.resample('W').last()
        data_monthly = data.resample('ME').last()

        latest_daily_signal = data.iloc[-1].filter(like='_Signal')
        latest_weekly_signal = data_weekly.iloc[-1].filter(like='_Signal')
        latest_monthly_signal = data_monthly.iloc[-1].filter(like='_Signal')

        return {
            'daily': latest_daily_signal.to_dict(),
            'weekly': latest_weekly_signal.to_dict(),
            'monthly': latest_monthly_signal.to_dict()
        }
```

III. Scraper

i. data_storage

- **Singleton Pattern**:
  - Имплементирано во DatabaseConfig за да се осигураме дека постои само една конфигурациска инстанца
  - Централно управува со поставките за поврзување со базата на податоци
- **Factory Pattern**:
  - DatabaseConnectionFactory се справува со креирање на поврзувањето со базата на податоци
  - Ја енкапсулира логиката на поврзувањето и го олеснува менувањето или проширувањето
- **Strategy Pattern**:
  - Создадена е апстрактна класа DatabaseStrategy со конкретни имплементации
    - QueryStrategy за SELECT операции
    - UpdateStrategy за INSERT/UPDATE операции
    - BatchInsertStrategy за сериски вметнувања
  - Го олеснува додавањето на нови стратегии за работа со бази на податоци
- **Repository Pattern**:
  - Класата DataRepository обезбедува чист интерфејс за пристап до податоци
  - Ги опфаќа сите операции со податоци и детали за нивната имплементација

```python
import os
from datetime import datetime
from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Tuple, Any
import psycopg2
from psycopg2.extensions import connection, cursor

# Configuration class using Singleton pattern
class DatabaseConfig:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
```

```
        return cls._instance

    def __init__(self):
        if not hasattr(self, 'initialized'):
            self.db_url = os.getenv('DB_URL',
"postgresql://dians:dians123@localhost:9555/diansdb")
            self.initialized = True

# Database connection factory
class DatabaseConnectionFactory:
    @staticmethod
    def create_connection() -> connection:
        config = DatabaseConfig()
        return psycopg2.connect(config.db_url)

# Abstract strategy for database operations
class DatabaseStrategy(ABC):
    @abstractmethod
    def execute_query(self, conn: connection, query: str, params: tuple = None) -
> Any:
        pass

# Concrete strategies
class QueryStrategy(DatabaseStrategy):
    def execute_query(self, conn: connection, query: str, params: tuple = None) -
> List[tuple]:
        with conn.cursor() as cur:
            cur.execute(query, params or ())
            return cur.fetchall()


class UpdateStrategy(DatabaseStrategy):
    def execute_query(self, conn: connection, query: str, params: tuple = None) -
> None:
        with conn.cursor() as cur:
            cur.execute(query, params or ())
            conn.commit()

class BatchInsertStrategy(DatabaseStrategy):
    def execute_query(self, conn: connection, query: str, params: List[tuple]) ->
None:
        with conn.cursor() as cur:
            cur.executemany(query, params)
            conn.commit()
```

```
class IssuerDate:
    def __init__(self, issuer: str, last_date: datetime):
        self.issuer = issuer
        self.last_date = last_date

class IssuerData:
    def __init__(self, date: datetime, issuer: str, avg_price: str,
last_trade_price: str,
                 max_price: str, min_price: str, percent_change: str,
turnover_best: str,
                 total_turnover: str, volume: str):
        self.date = date
        self.issuer = issuer
        self.avg_price = avg_price
        self.last_trade_price = last_trade_price
        self.max_price = max_price
        self.min_price = min_price
        self.percent_change = percent_change
        self.turnover_best = turnover_best
        self.total_turnover = total_turnover
        self.volume = volume

# Repository pattern for data access
class DataRepository:
    def __init__(self):
        self.connection_factory = DatabaseConnectionFactory()
        self.query_strategy = QueryStrategy()
        self.update_strategy = UpdateStrategy()
        self.batch_strategy = BatchInsertStrategy()
        self._initialize_db()

    def _initialize_db(self) -> None:
        create_tables_query = """
            CREATE TABLE IF NOT EXISTS issuer_dates (
                issuer TEXT PRIMARY KEY,
                last_date DATE
            );
            CREATE TABLE IF NOT EXISTS issuer_data (
                date DATE,
                issuer TEXT,
                avg_price TEXT,
                last_trade_price TEXT,
                max_price TEXT,
                min_price TEXT,
```

```
                percent_change TEXT,
                turnover_best TEXT,
                total_turnover TEXT,
                volume TEXT,
                PRIMARY KEY (date, issuer)
            );
        """
        with self.connection_factory.create_connection() as conn:
            self.update_strategy.execute_query(conn, create_tables_query)

    def load_issuer_dates(self) -> Dict[str, datetime]:
        query = "SELECT issuer, last_date FROM issuer_dates"
        with self.connection_factory.create_connection() as conn:
            results = self.query_strategy.execute_query(conn, query)
            return {row[0]: row[1] for row in results}

    def update_issuer_date(self, issuer_date: IssuerDate) -> None:
        query = """
            INSERT INTO issuer_dates (issuer, last_date)
            VALUES (%s, %s)
            ON CONFLICT (issuer) DO UPDATE
            SET last_date = EXCLUDED.last_date
        """
        date_str = issuer_date.last_date.strftime('%Y-%m-%d') if
issuer_date.last_date else None
        with self.connection_factory.create_connection() as conn:
            self.update_strategy.execute_query(conn, query, (issuer_date.issuer,
date_str))

    def get_issuer_date(self, issuer: str) -> Optional[datetime]:
        query = "SELECT last_date FROM issuer_dates WHERE issuer = %s"
        with self.connection_factory.create_connection() as conn:
            results = self.query_strategy.execute_query(conn, query, (issuer,))
            return results[0][0] if results and results[0][0] else None

    def save_issuer_data(self, data_list: List[IssuerData]) -> None:
        formatted_rows = [
            (
                data.date.strftime("%Y-%m-%d"),
                data.issuer,
                data.avg_price,
                data.last_trade_price,
                data.max_price,
                data.min_price,
                data.percent_change,
```

```
                data.turnover_best,
                data.total_turnover,
                data.volume
            )
            for data in data_list
        ]

        query = """
            INSERT INTO issuer_data (
                date, issuer, avg_price, last_trade_price, max_price, min_price,
                percent_change, turnover_best, total_turnover, volume
            ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
            ON CONFLICT (date, issuer) DO NOTHING
        """

        with self.connection_factory.create_connection() as conn:
            self.batch_strategy.execute_query(conn, query, formatted_rows)

    def get_all_data(self) -> List[IssuerData]:
        query = "SELECT * FROM issuer_data"
        with self.connection_factory.create_connection() as conn:
            results = self.query_strategy.execute_query(conn, query)
            return [
                IssuerData(
                    date=row[0],
                    issuer=row[1],
                    avg_price=row[2],
                    last_trade_price=row[3],
                    max_price=row[4],
                    min_price=row[5],
                    percent_change=row[6],
                    turnover_best=row[7],
                    total_turnover=row[8],
                    volume=row[9]
                )
                for row in results
            ]

    def count_issuer_data_rows(self) -> int:
        query = "SELECT COUNT(*) FROM issuer_data"
        with self.connection_factory.create_connection() as conn:
            results = self.query_strategy.execute_query(conn, query)
            return results[0][0] if results else 0

    def get_by_issuer(self, issuer: str) -> List[IssuerData]:
```

```
    query = "SELECT * FROM issuer_data WHERE issuer = %s"
    with self.connection_factory.create_connection() as conn:
        results = self.query_strategy.execute_query(conn, query, (issuer,))
        return [
            IssuerData(
                date=row[0],
                issuer=row[1],
                avg_price=row[2],
                last_trade_price=row[3],
                max_price=row[4],
                min_price=row[5],
                percent_change=row[6],
                turnover_best=row[7],
                total_turnover=row[8],
                volume=row[9]
            )
            for row in results
        ]
```

ii. stock_data_scraper

- **Strategy Pattern**:
    - Се користи преку апстрактните класи DataParser и DataFetcher
    - Ни овозможува да замениме различни стратегии за парсирање (како HTML, JSON, XML) без да го менуваме главниот scraper код
    - Слично на тоа, можеме да го промениме начинот на преземање податоци (HTTP requests, локални датотеки и др.) со создавање на нови имплементации за преземање (fetch)

- **Dependency Injection**:
    - Главната класа StockDataScraper прифаќа parser, fetcher и storage како зависности
    - Го прави кодот подобар за тестирање бидејќи можеме да инјектираме mock објекти
    - Овозможува флексибилна конфигурација

- **Iterator Pattern**:
    - Имплементиран во класата DateRangeIterator
    - Се справува со сложеноста на разложување на големи временски периоди на помали парчиња
    - Го прави кодот за преминување на временскиот опсег почист и поодржлив

```python
from dataclasses import dataclass
from datetime import date, timedelta
from typing import List, Dict, Optional
from abc import ABC, abstractmethod
import logging
import requests
from bs4 import BeautifulSoup, Tag
from requests.exceptions import RequestException

# Data Models
@dataclass
class ScrapingConfig:
    base_url: str = "https://www.mse.mk/mk/stats/symbolhistory"
    max_days_per_request: int = 364
    retry_attempts: int = 3
    timeout_seconds: int = 30

@dataclass
class ScrapingResult:
    data: List[Dict[str, str]]
    success: bool
    error_message: Optional[str] = None

# Abstract base classes for strategy pattern
class DataParser(ABC):
    @abstractmethod
    def parse(self, content: str, issuer: str) -> List[Dict[str, str]]:
        pass

class DataFetcher(ABC):
    @abstractmethod
    def fetch(self, url: str, params: Dict) -> Optional[str]:
        pass
```

```python
# Concrete implementations
class HTMLTableParser(DataParser):
    COLUMN_NAMES = [
        "Date", "Last trade price", "Max", "Min", "Avg. Price",
        "%chg.", "Volume", "Turnover in BEST in denars", "Total turnover in
denars"
    ]

    def parse(self, content: str, issuer: str) -> List[Dict[str, str]]:
        soup = BeautifulSoup(content, 'html.parser')
        return self._parse_table(soup, issuer)

    def _parse_table(self, soup: BeautifulSoup, issuer: str) -> List[Dict[str,
str]]:
        results = []
        table = soup.select_one('#resultsTable > tbody')

        if not table:
            return results

        for row in table.find_all('tr'):
            row_data = self._parse_row(row)
            if row_data:
                row_data['Issuer'] = issuer
                results.append(row_data)

        return results

    def _parse_row(self, row: Tag) -> Optional[Dict[str, str]]:
        row_data = {}
        cells = row.find_all('td')

        if len(cells) != len(self.COLUMN_NAMES):
            return None

        for td, column in zip(cells, self.COLUMN_NAMES):
            # Skip empty Max values as they indicate invalid rows
            if column == 'Max' and not td.text.strip():
                return None
            row_data[column] = td.text.strip()

        return row_data
```

```python
class RequestsFetcher(DataFetcher):
    def __init__(self, config: ScrapingConfig):
        self.config = config
        self.logger = logging.getLogger(__name__)

    def fetch(self, url: str, params: Dict) -> Optional[str]:
        for attempt in range(self.config.retry_attempts):
            try:
                response = requests.get(
                    url,
                    params=params,
                    timeout=self.config.timeout_seconds
                )
                response.raise_for_status()
                return response.text
            except RequestException as e:
                self.logger.warning(f"Attempt {attempt + 1} failed: {str(e)}")
                if attempt == self.config.retry_attempts - 1:
                    self.logger.error(f"All attempts failed for URL: {url}")
                    return None
        return None


# Date range iterator for handling large date ranges
class DateRangeIterator:
    def __init__(self, start_date: date, end_date: date, max_days: int):
        self.current = start_date
        self.end_date = end_date
        self.max_days = max_days

    def __iter__(self):
        return self

    def __next__(self) -> tuple[date, date]:
        if self.current >= self.end_date:
            raise StopIteration

        period_end = min(
            self.current + timedelta(days=self.max_days),
            self.end_date
        )
        result = (self.current, period_end)
        self.current = period_end + timedelta(days=1)
        return result
```

```python
# Main scraper class
class StockDataScraper:
    def __init__(
        self,
        storage: 'DataStorage',
        config: ScrapingConfig = ScrapingConfig(),
        parser: Optional[DataParser] = None,
        fetcher: Optional[DataFetcher] = None
    ):
        self.storage = storage
        self.config = config
        self.parser = parser or HTMLTableParser()
        self.fetcher = fetcher or RequestsFetcher(config)
        self.logger = logging.getLogger(__name__)

    def scrape_issuer_data(self, issuer: str, start_date: date) ->
ScrapingResult:
        """
        Scrapes stock data for a given issuer from start_date until today.
        Returns a ScrapingResult containing the scraped data and status
information.
        """
        url = f"{self.config.base_url}/{issuer}"
        all_data = []

        date_ranges = DateRangeIterator(
            start_date,
            date.today(),
            self.config.max_days_per_request
        )

        try:
            for period_start, period_end in date_ranges:
                params = {
                    "FromDate": self._format_date(period_start),
                    "ToDate": self._format_date(period_end),
                }

                content = self.fetcher.fetch(url, params)
                if not content:
                    return ScrapingResult(
                        data=[],
                        success=False,
                        error_message=f"Failed to fetch data for {issuer}"
                    )
```

```
                period_data = self.parser.parse(content, issuer)
                all_data.extend(period_data)

            self._log_results(issuer, all_data)
            return ScrapingResult(data=all_data, success=True)

        except Exception as e:
            error_msg = f"Unexpected error while scraping {issuer}: {str(e)}"
            self.logger.error(error_msg)
            return ScrapingResult(data=[], success=False,
error_message=error_msg)

    @staticmethod
    def _format_date(d: date) -> str:
        return d.strftime("%d.%m.%Y")

    def _log_results(self, issuer: str, data: List[Dict[str, str]]) -> None:
        if data:
            self.logger.info(f"Collected {len(data)} rows for {issuer}")
        else:
            self.logger.warning(f"No data collected for {issuer}")
```

iii. Pipeline

Останатите дефинирани класи во овој пакет *filter1, filter2, filter3, pipeline* и *run_ pipeline* останаа не променети. Истите се однесуваат на **Pipeline** архитектурениот модел и се користат во процесот на влечење податоци, креирени за потребите на Домашна 1.