Употреба на Singleton и Strategy модели

1. Singleton

Имплементиран е Singleton моделот за да се оптимизира пристапот на апликацијата до базата на податоци. Singleton моделот обезбедува апликацијата да има само една инстанца во даден момент, како и овозможува глобален пристап до таа инстанца. Овој пристап е особено корисен за управување со ресурси како што се поврзување со базата на податоци, каде што повеќекратните врски може да доведат до конфликти на ресурси, зголемена латентност или проблеми со заклучување на базата.

1.1. Предности

1.1.1. Со обезбедување дека постои само една врска со базата на податоци во текот на животниот циклус на апликацијата, Singleton моделот ги намалува трошоците при постојано отворање и затворање конекции. Тоа го минимизира ризикот од надминување на конекциски лимити.

1.1.2. Имање единствена конекција го олеснува управувањето со базата на податоци. На пример, врската може да се иницијализира, конфигурира и затвора на контролиран начин.

1.1.3. Централизираната структура гарантира дека сите промени во логиката за ракување со базата на податоци треба да се направат само на едно место, поедноставувајќи го дебагирањето и одржувањето.

1.2. Имплементација

1.2.1. Креирање нова инстанца од класата

```
def __new__(cls, *args, **kwargs):
    if not cls._instance:
        cls._instance = super(DatabaseConnection, cls).__new__(cls, *args,**kwargs)
       cls._instance._initialize_connection()
    return cls._instance
```

Ја имплементира Singleton шемата со тоа што се осигурува дека е креиран само еден примерок од класата.

1.2.2. Иницијализирање на врската со базата на податоци

```
def _initialize_connection(self):
    """Initialize the database connection."""
    base_dir = os.path.dirname(os.path.abspath(__file__))
    self.db_path = os.path.join(base_dir, 'data', 'stock_data.db')
    self.connection = sqlite3.connect(self.db_path, check_same_thread=False)
```

Ја пресметува патеката на базата на податоци динамички врз основа на локацијата на тековната датотека и отвара конекција до неа.

1.2.3. Враќање валидна и активна врска до базата

```
def get_connection(self):
    """Get the database connection, reopening it if necessary."""
    try:
        # Test the connection with a simple query
        self.connection.execute("SELECT 1")
    except (sqlite3.ProgrammingError, sqlite3.OperationalError):
        print("Reopening closed database connection...")
        self._initialize_connection()  # Reinitialize the connection
    return self.connection
```

Извршува едноставно барање (SELECT 1) за да тестира дали врската е сè уште отворена. Ако врската е затворена или е неважечка (sqlite3.ProgrammingError или

sqlite3.OperationalError), таа повторно ја иницијализира врската со повикување _initialize_connection.

1.2.4. Затворање на врската со базата на податоци

```
def close_connection(self):
        """Close the database connection."""
        if hasattr(self, 'connection'):
            self.connection.close()
            self.connection = None
```

Обезбедува чистење на ресурсите со експлицитно затворање на врската со базата на податоци кога повеќе не е потребна.

1.2.5. Тестирање на класата DatabaseConnection

```
def test_database_connection():
    """Check if the database connection is successful at startup."""
    try:
        db = DatabaseConnection().get_connection()
        cursor = db.cursor()
        cursor.execute("SELECT 1")  # Simple test query
        print("Database connection successful!")
        cursor.close()
    except Exception as e:
        print(f"Database connection error: {e}")
        raise
```

Презема врска од единечната инстанса и извршува барање (SELECT 1) за да тестира дали врската е оперативна. Ако барањето е успешно печати порака за успешно извршување, а ако конекцијата не е успешна ги пешати сите релевантни исклучоци.

2. Strategy

Имплементиран е Strategy шаблонот за справување со различни пресметки на технички индикатори. Шаблонот Strategy овозможува дефинирање на фамилија на алгоритми, инкапсулирање на секој од нив и правење нивно заменливи. Овој пристап промовира флексибилност и одржливост, овозможувајќи и на апликацијата да избира и префрла помеѓу различни алгоритми при извршување без да го менува кодот на клиентот.
Во апликацијата, логиката за техничката анализа е организирана во поединечни модули за секој индикатор. Секој модул имплементира специфичен индикатор, како што се Bollinger, Exponential Moving Average, Momentum, Relative Strength Index и Simple Moving Average, следејќи го заедничкиот интерфејс дефиниран во base.py. Овој пристап ја опфаќа пресметковната логика за секој индикатор, овозможувајќи им лесно да се заменуваат и повторно да се користат.

2.1. Предности

2.1.1. Модуларната организација на индикаторите го олеснува изборот и примената на специфичен индикатор.

2.1.2. Логиката на секој индикатор е изолирана во неговата соодветна датотека, што ги намалува меѓузависните и го прави дебагирањето поедноставно. На пример, ако логиката за пресметка за RSI треба да се ажурира, само rsi.py треба да се измени, без влијание врз останатите индикатори.

2.1.3. Додавањето нов индикатор вклучува создавање нов модул. Ниту еден постоечки код не бара модификација, зачувувајќи ја стабилноста на системот.

2.2. Имплементација

2.2.1. TechnicalIndicator

```
class TechnicalIndicator(ABC):
    @abstractmethod
    def calculate(self, data, **kwargs):
        pass
```

Класата TechnicalIndicator служи како план за сите класи на технички индикатори. Секоја подкласа (како што се BollingerBandsIndicator, RSI, итн.) ќе го имплементира методот на пресметување, осигурувајќи дека секоја класа на индикатор има конзистентен интерфејс за извршување на пресметките.

2.2.2. BollingerBandsIndicator

```
from .base import TechnicalIndicator
class BollingerBandsIndicator(TechnicalIndicator):
    def calculate(self, data, window=10):
        ma = data.rolling(window=window).mean()
        std = data.rolling(window=window).std()
        upper_band = ma + (2 * std)
        lower_band = ma - (2 * std)
        return ma.fillna(0), upper_band.fillna(0), lower_band.fillna(0)
```

Класата bollinger_bands.py ја воведува класата BollingerBandsIndicator, која ги пресметува подвижниот просек, горниот опсег и долниот опсег, каде што std е стандардното отстапување. Вредностите што недостасуваат се пополнуваат со 0 за да се обезбеди непречена интеграција во понатамошните анализи.

2.2.3. EMAIndicator

```
from .base import TechnicalIndicator
class EMAIndicator(TechnicalIndicator):
    def calculate(self, data, window=10):
        return data.ewm(span=window, adjust=False).mean().fillna(0)
```

Класата EMAIndicator наследува од TechnicalIndicator и го имплементира пресметковниот метод за пресметување на Exponential Moving Average (EMA) на дадените податоци. Ја користи функцијата ewm со одреден временски период и ги пополнува сите NaN вредности со 0.

2.2.4. MomentumIndicator

```
from .base import TechnicalIndicator
class MomentumIndicator(TechnicalIndicator):
    def calculate(self, data, window):
        result = ((data - data.shift(window)) / data.shift(window) * 100)
        return result.fillna(0)
```

Класата MomentumIndicator наследува од TechnicalIndicator и го имплементира метод за пресметување на моментумот со споредување на тековните податоци со податоците поместени од одреден временски период. Ја пресметува процентуалната промена и ги пополнува сите NaN вредности со 0.

2.2.5. RSIIndicator

```
from .base import TechnicalIndicator
class RSIIndicator(TechnicalIndicator):
    def calculate(self, data, window=14):
        delta = data.diff()
        gain = delta.where(delta > 0, 0).rolling(window=window).mean()
        loss = -delta.where(delta < 0, 0).rolling(window=window).mean()
        rs = gain / loss
        rsi = 100 - (100 / (1 + rs))
        return rsi.fillna(0)
```

Relative Strength Index (RSI) го пресметува односот на просечните добивки и просечните загуби во одреден временски период, што укажува на условите за прекумерно купување или препродажба. Ги враќа вредностите на RSI, пополнувајќи ги сите NaN вредности со 0.

2.2.6. SMIndicator

```
from .base import TechnicalIndicator
class SMAIndicator(TechnicalIndicator):
    def calculate(self, data, window=10):
```

```
        return data.rolling(window=window).mean().fillna(0)
```

Simple Moving Average (SMA) се пресметува со просек на податоците, обезбедувајќи линија на трендови. Ги враќа вредностите на SMA, пополнувајќи ги сите NaN вредности со 0.

2.2.7. StochasticOscillatorIndicator

```
from .base import TechnicalIndicator

class StochasticOscillatorIndicator(TechnicalIndicator):
    def calculate(self, data, high, low, window):
        highest_high = high.rolling(window=window).max()
        lowest_low = low.rolling(window=window).min()
         stochastic_oscillator = ((data - lowest_low) / (highest_high -
  lowest_low))* 100
        return stochastic_oscillator.fillna(0)
```

Оваа класа го пресметува Стохастичкиот осцилатор споредувајќи ја тековната цена со највисоката и најниската во текот на дефинираниот временски период, обезбедувајќи процентуална вредност за означување на моментумот. Го враќа пресметаниот осцилатор, пополнувајќи ги сите NaN вредности со 0.

2.2.8. UltimateOscillatorIndicator

```
class UltimateOscillatorIndicator(TechnicalIndicator):
    def calculate(self, data, high, low, window1=7, window2=14, window3=28):
        buying_pressure = data - low
        true_range = high - low
        avg1 = buying_pressure.rolling(window=window1).sum() /
true_range.rolling(window=window1).sum()
        avg2 = buying_pressure.rolling(window=window2).sum() /
true_range.rolling(window=window2).sum()
        avg3 = buying_pressure.rolling(window=window3).sum() /
true_range.rolling(window=window3).sum()

        ultimate_oscillator = 100 * ((4 * avg1) + (2 * avg2) + avg3) / 7
        return ultimate_oscillator.fillna(0)
```

UltimateOscillator се пресметува со користење на три различни временски рамки за да се процени притисокот за купување и вистинскиот опсег, при што конечниот резултат е просек од трите периоди. Ја враќа вредноста на осцилаторот, пополнувајќи ги сите NaN вредности со 0.

2.2.9. WilliamsPercentRangeIndicator

```
class WilliamsPercentRangeIndicator(TechnicalIndicator):
    def calculate(self, data, high, low, window):
        highest_high = high.rolling(window=window).max()
        lowest_low = low.rolling(window=window).min()
        result = (-100 * (highest_high - data) / (highest_high -
lowest_low)).fillna(0)
        return result
```

Оваа класа го пресметува WilliamsPercentRange индикаторот, кој ги мери условите за прекумерно купување и препродажба преку споредување на моменталната цена со највисоката и најниската цена во одредна временска рамка. Го враќа пресметаниот резултат, пополнувајќи ги сите NaN вредности со 0.