Софтверски шаблони искористени во crypto-info

1. **Фасада (Facade Pattern):** Имплементиран е преку класите **SantimentFacade** и **CryptoDataAggregator**.
    - **Зошто е искористен:** Оригиналниот код содржеше комплексна логика за комуникација со Santiment GraphQL API-то, вклучувајќи справување со пагинација (делење на временски интервали), "retry" логика при грешки и специфично парсирање на JSON одговори. Процесот на спојување на TVL и Hashrate податоци вклучува координација на повеќе колектори, чистење на симболи и комплексен outer join на DataFrames.

    - **Бенефит:** Овој шаблон ја крие таа комплексност зад едноставен интерфејс (fetch_metrics_for_slug). Ова овозможува главната бизнис логика да остане чиста и независна од техничките детали на надворешниот сервис. Доколку API-то се промени во иднина, промените ќе бидат изолирани само во оваа класа. Ја крие комплексноста на спојувањето податоци. Клиентот само повикува aggregate() и добива чист CSV/DataFrame, без да знае за внатрешната логика на спојување.

2. **Singleton:** Имплементиран е преку класата **DatabaseManager** **SentimentModelSingleton**.
    - **Зошто е искористен:** Конекциите кон базата на податоци се ресурсно скапи операции. Не е ефикасно да се креира нова конекција при секое барање. FinBERT моделот за процесирање на јазик е голем и одзема многу време и меморија за вчитување. Не е ефикасно да се иницијализира при секое повикување.

    - **Бенефит:** Овој шаблон гарантира дека ќе постои само една инстанца од Engine објектот за конекција во текот на извршувањето на апликацијата. Ова спречува преоптоварување на базата со непотребни конекции и овозможува централизирано конфигурирање на параметрите за базата. Гарантира дека моделот се вчитува во меморија само еднаш. Ова драстично ги намалува ресурсите и времето на извршување на скриптата.

3. **Template Method Pattern** Имплементиран е преку апстрактната класа **ETLPipeline** и конкретната класа **OnChainMergerPipeline**, апстрактната класа **DataCollector** и методот **collect**. Како и преку идентична структура на скриптите во **oscilators**, **moving-averages** и **филтрите**.
    - **Зошто е искористен:** Процесот на обработка на податоци секогаш следи иста структура: Екстракција (земање податоци), Трансформација (чистење и спојување) и Вчитување (зачувување во база) — популарно познато како

ETL. Оригиналниот код ги мешаше овие чекори во една голема функција, што го правеше тежок за одржување и тестирање. Кога повеќе класи/модули следат ист општ алгоритам, но со различна имплементација на чекорите, Template Method обезбедува конзистентност. Кога ќе ја разберете едната скрипта, автоматски ги разбирате и сите останати. Конзистентна структура го прави кодот предвидлив. Секој филтер треба да има исто однесување во однос на logging и error handling. И TVL колекторот и Security колекторот го следат истиот алгоритам: логирање почеток -> преземање податоци -> логирање крај. Само логиката за преземање е различна.

- **Бенефит:** Овој шаблон дефинира "скелет" на алгоритмот во методот run(), додека имплементацијата на чекорите ја остава на подкласите. Ова овозможува лесно додавање на нови pipeline-и во иднина (на пример, за други типови на податоци) кои ќе ја следат истата строга структура, без дуплирање на логиката за оркестрација. Кодот е предвидлив и лесен за навигација. Истите имиња на функции, исти параметри, ист тек. Ова го олеснува одржувањето и додавањето нови типови на анализа. Предвидливо однесување на филтрите. Лесно е да се разбере што прави секој филтер бидејќи сите следат ист шаблон.

4. **Шаблон Adapter:** Имплементиран е преку додавање на методот **fetch_data_as_dataframe** во сервисот **OnChainDataService** и негово повикување во новиот pipeline.
    - **Зошто е искористен:** Постоечката класа OnChainDataService беше дизајнирана да функционира како независен процес кој зачувува податоци директно во датотека (CSV), без да враќа резултат (void). Новиот ETLPipeline имаше потреба од директен пристап до податоците во меморија (како DataFrame) за да може да ги спои со други извори пред зачувување.

    - **Бенефит:** Наместо да креираме привремени фајлови и да ги читаме повторно (што е бавно и неефикасно), го адаптиравме интерфејсот на сервисот за да може да комуницира директно со pipeline-от. Ова овозможи лесна интеграција на постоечки код во нов контекст без да се наруши неговата оригинална функционалност.

5. **Шаблон Strategy:** Имплементиран е преку апстрактната класа **PredictionStrategy** и конкретната класа **RandomForestStrategy**. Имплементиран е во **combine_signals** каде различни стратегии за техничка анализа (осцилатори vs движечки проседи) се пресметуваат независно и се комбинираат. Имплементиран е во **scrapers_aggregator** преку **scrape_source** функцијата која третира различни scraping стратегии (requests vs Selenium) униформно. Имплементиран е преку тоа

што секој филтер е различна стратегија за процесирање на податоци. Pipeline-от може да користи различни комбинации на филтри. Имплементиран е преку апстрактната класа **PipelineStep** и конкретните чекори (**DataIngestionStep**, **SentimentAnalysisStep**, итн.). Имплементиран е преку класата **MetricStrategy** и нејзините подкласи (**HashRateStrategy**, **PoSMarketCapStrategy**).

- **Зошто е искористен:** Предвидувањето на цени е процес кој често бара експериментирање со различни алгоритми (Random Forest, XGBoost, Linear Regression, LSTM). Во оригиналниот код, логиката за Random Forest беше "хардкодирана" во главната функција. Во нашиот систем сега имаме два начини на предвидување: „Класично машинско учење“ (Random Forest) и „Длабоко учење“ (LSTM/Neural Networks). Овие два пристапи имаат многу различни процеси на тренирање (LSTM бара скалирање на податоци во опсег [-1, 1], креирање на тензори, и iterative training loop со epochs, додека Random Forest користи едноставен fit/predict. Различни методи на техничка анализа треба да можат да се пресметуваат и комбинираат без да се менува кодот за комбинирање. Треба да биде лесно да се додадат нови методи. Различни извори користат различни технологии за scraping. Агрегаторот не треба да знае за овие детали. Различни пристапи за процесирање на податоци треба да можат да се заменуваат или реорганизираат. Процесот на анализа се состои од многу независни чекори (читање, филтрирање, анализа, зачувување). Сакавме да избегнеме една огромна функција со измешана логика. Постојат различни начини за мерење на "сигурноста" на мрежата зависно од тоа дали е PoW или PoS.

- **Бенефит:** Овој шаблон овозможува лесна замена на алгоритмот за машинско учење. Класата PredictionService не зависи од специфичниот модел, туку од општиот интерфејс. Ако сакаме да додадеме нов модел во иднина, само креираме нова класа која наследува од PredictionStrategy, без да го менуваме постоечкиот код за процесирање на податоци или зачувување во база. Шаблонов ни овозможува да ја енкапсулираме целата комплексност на PyTorch (тензори, уреди, градиенти) внатре во една класа. Главниот PredictionService останува ист – тој само повикува predict(). Ова го прави системот мошне флексибилен; ако утре сакаме да користиме Transformer модел, само додаваме TransformerStrategy без да го менуваме остатокот од апликацијата. Лесно е да се додадат нови методи на анализа без да се модифицира постоечкиот код. Секој метод е изолиран и може да се тестира независно. Лесно е да се додадат нови извори на податоци без да се модифицира агрегаторот. Секој скрејпер е независен модул. Флексибилна композиција на pipeline-от. Може да се менува редоследот или да се прескокнат филтри по потреба. Овозможува лесно додавање или отстранување на чекори без да се менува главниот код (Open/Closed Principle). Овозможува лесна замена на алгоритмот за преземање метрики во runtime, зависно од типот на криптовалутата.

6. **Шаблон Composite:** Имплементиран е преку разделување на **save_df_to_db** во специјализирани функции.
    - **Зошто е искористен:** Големи datasets (>700K редови) се многу побавни со стандардниот **to_sql**. PostgreSQL COPY е многу побрз, но бара различна имплементација.

    - **Бенефит:** Автоматска оптимизација за големи datasets. Зачувува значително време при bulk import на податоци.

7. **Шаблон MVC:** Имплементиран е преку стандардната **Spring Boot** архитектура со јасна сепарација:

    - Model: Entity класи (Coin, OhlcvData, TextSentiment …) за репрезентација на податоци

    - View: REST API endpoints (/api/coins/, /api/ohlcv-data/ …) кои враќаат JSON податоци кои frontend-от ги прикажува на екранот.

    - Controller: @RestController класи (CoinController, OhlcvDataController …) за обработка на HTTP requests

    - **Зошто е искористен:** MVC обезбедува јасна сепарација на одговорности - бизнис логиката (Model), корисничкиот интерфејс (View) и контролата на текот (Controller) се независни компоненти. Ова овозможува паралелен развој и лесна промена на една компонента без да влијае на другите.

    - **Бенефит:** Апликацијата е полесна за одржување. Лесно тестирање на секоја компонента одделно. Промените во UI не бараат промени во бизнис логиката и обратно.

8. **Шаблон Factory:** Имплементиран е преку класата **StrategyFactory**.
    - **Зошто е искористен:** Системот мора динамички да одлучи дали за даден симбол (пр. BTC vs ETH) треба да преземе Hashrate или Market Cap.

    - **Бенефит:** Ја централизира логиката за креирање на објекти. Го елиминира потребата од долги if-else блокови во главниот код за колектирање.