Федеральное государственное автономное образовательное учреждение высшего образования Национальный исследовательский университет ИТМО

Факультет программной инженерии и компьютерной техники

Архитектура программных систем

Лабораторная работа №2

Выполнил: студент группы P3308,
Васильев Никита Алексеевич

Преподаватель: Перл Иван Андреевич

Санкт-Петербург 2025

Текст задания

Из списка шаблонов проектирования GoF и GRASP выбрать 3-4 шаблона и для каждого из них придумать 2-3 сценария, для решения которых могу применены выбранные шаблоны.

Сделать предположение о возможных ограничениях, к которым можем привести использование шаблона в каждом описанном случае. Обязательно выбрать шаблоны из обоих списков.

1. Strategy (Стратегия) — GoF

Strategy (Стратегия) — это паттерн, который позволяет выносить изменяющийся алгоритм в отдельные взаимозаменяемые реализации и переключать способ выполнения действия без переписывания основного кода.

Суть в двух идеях:

1. Есть контекст (место, где нужно выполнить действие: расчёт доставки, оплата, сортировка).
2. Есть набор стратегий (варианты алгоритма), у всех один общий интерфейс, но внутренняя логика разная.

В результате контекст не содержит кучи if/else под каждый случай; чтобы добавить новый вариант поведения, обычно достаточно добавить новую стратегию и подключить её в выбор.

Общие ограничения:

- **Рост числа классов/вариантов**: на каждый алгоритм своя стратегия → структура разрастается.
- **Сложность выбора стратегии**: нужна фабрика/резолвер/конфиг; иначе логика выбора превращается в if-else в другом месте.
- **Риск "толстого" интерфейса**: разные стратегии требуют разные данные → начинаются options, nullable-поля, усложнение контрактов.
- **Отладка и трассировка**: поведение зависит от выбранной стратегии → сложнее понять "почему так сработало", если выбор динамический.

Сценарий A: Расчёт доставки (курьер / ПВЗ / постамат / международная)

Проблема: в checkout каждый раз нужно посчитать стоимость+срок доставки, но правила разные:

- Курьер: зависит от адреса, зоны, времени, веса/габаритов.
- ПВЗ: зависит от выбранного пункта, графика, перевозчика.
- Постамат: ограничения по габаритам.
- Международная: таможня, другие сроки, другая тарификация.

Решение через Strategy:

- Вводим интерфейс `DeliveryPricingStrategy`:
  - `quote(order, destination, options) -> DeliveryQuote`
- Реализации:
  - `CourierDeliveryStrategy`
  - `PickupPointStrategy`
  - `LockerStrategy`
  - `InternationalStrategy`
- Выбор стратегии делает `DeliveryStrategyResolver` по типу доставки, региону, доступности.

Как выглядит поток:

1. Пользователь выбирает способ доставки «Курьером».
2. Сервер: `CheckoutService` просит `DeliveryStrategyResolver` подобрать стратегию.
3. Стратегия тянет нужные данные (тарифы, зоны, ограничения), считает: price, etaMin/etaMax, ограничения.
4. Сервер возвращает пользователю «Доставка 249₽, 2–3 дня».

Ограничения:

- **Большое количества стратегий**: если добавить, например, «курьер экспресс», «курьер вечерний», «ПВЗ стандарт/ускоренный», «международная эконом/стандарт», тогда стратегий станет 10–20. Это лечится композицией (стратегия + набор правил/валидаторов) или конфигурацией тарифов.
- **Сложный выбор стратегии**: «курьер доступен только в зоне + если вес < X + если время < 18:00». Если resolver будет содержать кучу `if`, это сложно нормально поддерживать. Тогда делают таблицу правил (конфиг), либо отдельный сервис `DeliveryEligibilityService`.
- **Разные стратегии требуют разные данные**: международной нужна таможенная стоимость и HS-коды, а ПВЗ — id пункта выдачи. Интерфейс становится слишком большим, или приходится пихать nullable-поля в `options`. Решение: вводят типизированные опции (sealed classes / разные DTO под тип доставки).

Сценарий B: Оплата (карта / СБП / рассрочка / кошелёк)

Проблема: процесс оплаты похож, но интеграции разные:

- Карта → 3DS, холд/капчер
- СБП → QR, статусы от банка
- Рассрочка → скоринг, договор
- Кошелёк → внутренние списания

Решение через Strategy:

Интерфейс `PaymentStrategy`:

- `initiatePayment(order, amount) -> PaymentSession`
- `handleCallback(payload) -> PaymentResult`

Реализации:

- `CardPaymentStrategy`
- `SBPPaymentStrategy`
- `InstallmentPaymentStrategy`
- `WalletPaymentStrategy`

Как выглядит поток:

- Пользователь выбирает оплату «СБП».
- Сервер создаёт `PaymentSession` через `SBPPaymentStrategy` → получает ссылку на QR/платёж.
- UI показывает пользователю QR.
- Платёжная система дергает callback → `handleCallback` фиксирует «paid».

- Сервер переводит заказ в «Оплачен», запускает доставку.

Ограничения:

- **Разная модель жизненного цикла**: у карты может быть “authorize → capture”, у СБП “pending → paid”, у рассрочки — “approved/declined”. Если попытаться сделать «одинаковый» интерфейс, часть методов будет не применима. Решение: разбивать на capability-интерфейсы (`Authorizable`, `Capturable`) или описывать state machine отдельно.
- **Сложность идемпотентности**: callback может прийти дважды. Стратегии должны быть идемпотентны (по `paymentId`), иначе будут двойные списания/двойное подтверждение заказа.
- **Тестирование**: в моках нужно симулировать разные статусы/коллбеки. Без контрактных тестов интеграции легко ломаются.

Сценарий C: Промо-логика (купон, скидка продавца, акции платформы)

Проблема: скидки комбинируются, но по правилам:

- Купон платформы нельзя вместе с купоном продавца
- Скидка по категории действует только на часть товаров
- «1+1=3» меняет формулу расчета корзины

Решение через Strategy:

Интерфейс `DiscountStrategy`:

- `apply(cart, context) -> DiscountResult`

Примеры:

- `CouponDiscountStrategy`
- `SellerDiscountStrategy`
- `CategoryPromoStrategy`
- `BundlePromoStrategy`

Ограничения:

- **Порядок применения важен**: разные стратегии дают разные итоги.
  Нужен `DiscountPipeline` с явным приоритетом, иначе баги «почему сумма такая».
- **Конфликты правил**: придётся вводить «арбитр» (правила совместимости), иначе пользователь сможет ломать математику скидок.

2. Observer (Наблюдатель) — GoF

Observer (Наблюдатель) — это паттерн для ситуации одно изменилось — многих надо уведомить.

Суть:

1. Есть издатель (Subject/Publisher), у которого происходит событие (сменился статус заказа, изменилась цена, пришёл трекинг).
2. Есть подписчики (Observers/Subscribers), которым важно об этом узнать (уведомления, логирование, аналитика, обновление UI).
3. Подписчики подписываются на издателя, и когда событие происходит — издатель рассылает уведомление всем подписанным.

В результате получаем слабую связанность — издатель не знает конкретно, кто и что будет делать с событием и возможность легко добавлять новые реакции (добавил нового подписчика — и всё), не меняя основной код издателя.

Общие ограничения:

- **Шторм событий и нагрузка**: много подписчиков / частые изменения → всплески CPU/очередей/уведомлений.
- **Порядок и консистентность**: события могут прийти в другом порядке или с задержками → нужны версии/таймстемпы/правила обработки.
- **Дубли/повторная доставка**: в асинхронной доставке события часто приходят повторно → нужна идемпотентность и дедупликация.
- **Скрытые зависимости**: сложно увидеть кто на что подписан, система становится менее прозрачной.
- **Утечки ресурсов** (в синхронном варианте): если не отписывать слушателей, можно держать лишние ссылки/память.

Сценарий А: Изменение статуса заказа → уведомления + история + аналитика

Проблема: когда заказ меняет статус («создан», «оплачен», «отправлен», «доставлен»), надо:

- обновить экран пользователя,
- отправить push/email,
- записать событие в историю заказа,
- отправить событие в аналитику.

Если всё делать прямо в `OrderService` — он станет комбайном.

Решение через Observer:

- Есть событие `OrderStatusChanged(orderId, from, to, time)`
- `OrderService` публикует событие.
- Подписчики:
  - `NotificationListener` (push/email)
  - `OrderHistoryListener` (аудит)
  - `AnalyticsListener` (метрики)

- `UIRealtimeGateway` (websocket/longpoll)

Как выглядит поток:

1. Payment callback подтвердил оплату.
2. `OrderService` меняет статус на “PAID”.
3. Публикует `OrderStatusChanged`.
4. Подписчики параллельно выполняют своё:
   - пуш «Оплата успешна»
   - история: «оплачено в 12:31»
   - аналитика: конверсия оплат
   - UI: обновить страницу

Ограничения:

- **Дубли событий**: брокер/ретраи → событие может прилететь 2 раза.
  Тогда `NotificationListener` отправит 2 пуша. Решение: дедупликация по `(eventId)` или “outbox + exactly-once на уровне приложения”.
- **Порядок**: если «Отправлен» обработался раньше «Оплачен» (из-за задержек), UI может мигнуть. Решение: хранить «версию статуса/sequence», игнорировать устаревшее.
- **Нагрузка**: пик распродажи → миллионы событий. Нужно асинхронить и масштабировать listeners отдельно.

Сценарий B: Отслеживание доставки (tracking updates)

Проблема: внешняя доставка шлёт события: «Принят», «В сортировочном центре», «Передан курьеру», «Доставлен».
На каждое событие надо обновлять:

- статус в карточке заказа,
- таймлайн,
- уведомления.

Решение через Observer:

- событие `DeliveryTrackingUpdated(orderId, trackingStatus, location, time)`
- подписчики:
  - `OrderStatusSyncListener`
  - `CustomerNotificationListener`
  - `SupportDashboardListener`

Ограничения:

- **Низкое качество данных от партнёра**: могут быть скачки статусов, пропуски. Listener должен быть устойчивым: «не понижать статус», «добавлять как событие, но не ломать state».
- **Согласованность**: если хранить «главный статус» в разных местах (заказ и доставка) — легко рассинхронизировать. Нужен один источник истины + правила синка.

Сценарий C: Изменение цены/наличия → подписка пользователя

Проблема:

В маркетплейсе пользователь часто видит:

- товара нет в наличии (или нет в его регионе),
- цена слишком высокая.

Он жмёт:

- «Уведомить, когда появится»
- «Уведомить, когда цена упадёт ниже X»

Дальше система должна:

- отслеживать изменения в остатках и/или цене,
- при выполнении условия отправить уведомление (push/email/внутренний inbox),
- сделать это масштабируемо: у одного SKU могут быть десятки тысяч подписчиков, а SKU миллионы.

Решение через Observer:

1. Источник событий (Publisher):
   - `InventoryService` (остатки/наличие)
   - `PricingService` (изменение цены, скидки, распродажи)
   - иногда ещё видимость по региону (ограничение доставки)
2. Они публикуют события:
   - `StockChanged(skuId, regionId, oldQty, newQty, timestamp)`
   - `PriceChanged(skuId, regionId, oldPrice, newPrice, timestamp)`
3. Подписчик (Subscriber) — сервис уведомлений:
   - `WatchlistNotificationListener` (или `SubscriptionProcessor`)
     Он слушает события и решает:
   - «Какие подписки удовлетворились?»
   - «Кого уведомлять?»
   - «Как не заспамить и не умереть по бюджету?»
4. Хранилище подписок:
   - таблица/коллекция `ProductSubscription`:
     - `userId, skuId, regionId, type (stock/price), threshold, channel, createdAt, lastNotifiedAt, state`

Как выглядит поток:

1. Пользователь в карточке товара нажимает «Уведомить, когда появится».
2. UI отправляет запрос на backend: создать подписку:

    - `POST /subscriptions { skuId, regionId, type=IN_STOCK }`
3. Backend сохраняет запись подписки:
    - `state=ACTIVE`
    - `lastNotifiedAt=null`
4. Через некоторое время поставщик/склад обновляет остатки: было `0`, стало `25`.
5. `InventoryService` публикует событие `StockChanged(skuId, regionId, 0 -> 25)`.
6. `WatchlistNotificationListener` получает событие и делает шаги:
    - Проверяет: переход из «нет» в «есть» (а не просто «25 -> 26»).
    - Находит все активные подписки на этот `skuId+regionId`.
    - Для каждой подписки решает: можно ли сейчас уведомлять (антиспам).
7. Уведомления уходят в push/email/внутренние уведомления.
8. Подписка либо:
    - помечается как `FULFILLED` (одноразовое уведомление «появилось»), либо
    - остаётся `ACTIVE` (если нужна многократная логика «снова появится»)

Ограничения:

- **Шторм уведомлений (push-спам):** Товар то появляется, то исчезает: 0 -> 5 -> 0 -> 3 -> 0 каждые 5–10 минут (например, резервы, отмены, синхронизация складов)**.** Или цена скачет из-за динамического ценообразования / промо: 1999 -> 1899 -> 1999 -> 1899
- **Стоимость и масштаб (миллионы подписчиков):** У топового товара 500k подписчиков «сообщить о наличии». Остатки обновились → нужно обработать 500k записей и отправить 500k пушей. Если таких товаров много — сервис умрёт по CPU/DB/очередям.

3. Controller — GRASP

Controller — это принцип, который говорит: системные события (запросы пользователя/внешних систем) должен принимать специальный объект-контроллер, который становится точкой входа для конкретного прецедента, а не размазывает обработку по доменным сущностям.

Суть:

1. Контроллер получает.
2. Он не делает бизнес-логику внутри себя**,** а координирует выполнение сценария: вызывает нужные сервисы/доменные объекты в правильном порядке, собирает результат, возвращает ответ.
3. Часто контроллер соответствует одному use-case или группе близких use-case.

Зачем:

- отделить приём запроса и оркестрацию от самой бизнес-логики;
- избежать ситуации, когда UI/HTTP слой напрямую дёргает доменные сущности хаотично;
- упростить сопровождение: логика сценария в одном месте, но детали правил — в домене/сервисах.

Общие ограничения:

- **Риск God Controller**: контроллер начинает содержать бизнес-правила и превращается в комбайн.
- **Сильная связность с множеством сервисов**: контроллер тянет много зависимостей → сложнее тестировать и поддерживать.
- **Дублирование сценариев**: похожие use-case легко копипастятся в разные контроллеры вместо переиспользования use-case слоя.
- **Границы ответственности**: если плохо отделить “оркестрацию” от домена, логика расползается между слоями.

Сценарий A: CheckoutController для «Подтвердить заказ»

Задача: один use-case «Оформление заказа» включает много шагов:

- валидация корзины,
- расчёт суммы,
- применение промокода,
- создание заказа,
- запуск оплаты,
- обработка результата.

Как делать по GRASP Controller:

- `CheckoutController` принимает “ConfirmOrderRequest”.
- Получаем оркестр:
  - `CartService.validate()`
  - `PricingService.calculateTotal()`
  - `OrderService.createDraft()`

    - `PaymentService.initiate()`

Получаем, что точка входа под use-case, а бизнес-логика остаётся в доменных/сервисных объектах.

Ограничения:

- **Риск God Controller**: если начать писать в контроллере «как считать скидки, как валидировать доставку» — контроллер станет жирным и непереиспользуемым. Правило: контроллер только связывает шаги и обрабатывает ошибки.
- **Дублирование**: если появится «Оформить в 1 клик» и ты скопируешь половину кода — будет боль. Решение: выделить `CheckoutUseCase`/`ApplicationService`.

Сценарий B: ReturnController (оформление возврата)

Проблема:

Принять системное событие: «пользователь хочет возврат» (HTTP запрос / команда). Провалидировать вход: пользователь = владелец заказа, позиции действительно в заказе, количества корректны. Оркестрировать use-case: проверить право на возврат, создать заявку на возврат, инициировать возврат денег, при необходимости запланировать забор/доставку обратно. Сформировать ответ: вернуть пользователю статус

Как делать по GRASP Controller:

- ReturnController = тонкая точка входа под прецедент «Оформить возврат».
- Он не содержит правил («можно/нельзя») и не интегрируется напрямую с внешними API.
- Он вызывает Application/UseCase сервис (например, `CreateReturnUseCase`), который уже координирует доменные сервисы.

Пример структуры вызовов:

- `ReturnController.createReturn(req)`
  → `CreateReturnUseCase.execute(cmd)`
  → `ReturnPolicyService.checkEligibility(...)`
  → `ReturnService.createRequest(...)`
  → async `RefundJob` (через очередь) / `PaymentService.refund...`
  → async `PickupJob` / `DeliveryService.schedulePickup(...)`

Поток:

1. Пользователь нажал «Оформить возврат»
2. `ReturnController` принимает запрос
3. вызывает:
    - `ReturnPolicyService.checkEligibility(order, items)`
    - `ReturnService.createRequest()`
    - `PaymentService.refundOrPartialRefund()`
    - `DeliveryService.schedulePickup()` (если нужно)

Ограничения:

- **Много интеграций** (платёжка + доставка). Если всё синхронно — контроллер будет ждать, а пользователь будет ждать. Нужны async задачи/очереди, иначе таймауты.
- **Контроллер легко превращается в «оркестратор всего мира»**, если не разделять возврат денег и логистику.

4. Information Expert — GRASP

Information Expert — это принцип распределения ответственности: делай так, чтобы задачу выполнял тот объект, у которого есть нужные данные для её выполнения.

Суть: если объект владеет информацией, необходимой для вычисления/проверки, то он и должен нести ответственность за это.

Зачем:

- меньше процедурного кода, где один сервис таскает данные туда-сюда и считает снаружи;
- логика ближе к данным → меньше ошибок и рассинхрона;
- обычно проще тестировать, потому что поведение сконцентрировано там, где данные.

Общие ограничения:

- **Раздувание сущностей**: эксперт может стать перегруженным (слишком много ответственности в одном классе).
- **Внешние зависимости рядом с доменом**: если эксперт для решения задачи вынужден ходить в БД/сервисы, модель теряет чистоту и тестируемость.
- **Данные распределены**: в реальных системах нужная информация может быть в разных сервисах/контекстах → эксперт не рядом, приходится вводить отдельные сервисы/агрегацию.
- **Сложно менять правила**: если эксперт содержит много бизнес-правил, изменения могут затрагивать ключевые доменные классы и вызывать каскад правок.

Сценарий A: Order как эксперт по итогу заказа

Идея**:** кто знает состав заказа и цены позиций? Сам `Order` (и `OrderItem`).
Значит `Order.total()` суммирует item totals, применяет скидки, добавляет доставку.

Плюсы:

- логика рядом с данными, меньше рассинхрона,
- проще тестировать «на уровне домена».

Ограничения на этом примере:

- Если `Order.total()` начинает делать внешние вызовы (курс валют, тарифы доставки, промо-сервис) — доменная модель перестаёт быть чистой и тестируемой.
    - Решение: `Order` считает то, что уже передано (prepared data), а внешние данные готовит `PricingService`.
- В микросервисах цены/скидки могут жить в другом сервисе → тогда `Order` не может быть «экспертом обо всём».

Сценарий B: Cart как эксперт по применению промокода к позициям

Идея: по GRASP Information Expert — ответственность отдаём тому объекту, у кого есть нужные данные. Корзина лучше всех знает состав позиций, количества, категории → значит именно Cart логично распределяет скидку «по строкам» (а не внешний сервис, который не видит деталей или начнёт дублировать логику).

Поток:

1. Пользователь вводит промокод
2. `PromoService` валидирует промокод (срок, пользователь, регион)
3. `Cart.applyPromo(validPromo)` распределяет скидку по позициям по своим правилам (например, только на категорию)

Ограничения:

- Если правила промо часто меняются маркетингом, «зашивать» их в `Cart` неудобно → придётся релизить код. Тогда лучше: правила в конфиге/правил-движке, а `Cart` лишь применяет готовые вычисленные скидки.
- Есть риск, что `Cart` станет монстром (и корзина, и скидки, и доставка, и налоги). Тогда надо делить ответственность (e.g., `PricingBreakdown`).

Сценарий C: Inventory как эксперт по доступности

Идея: доступность товара — это не «флаг», а набор правил (остатки, резервы, лимиты, региональные ограничения). Эти данные и правила естественно живут в Inventory/Stock, поэтому именно он должен отвечать на «можно ли зарезервировать X?».

Поток:

- `Inventory.canReserve(productId, qty)` знает остаток, резервы, лимиты, «1 в руки», «только для региона».
- `OrderService` при оформлении спрашивает `Inventory` и делает резерв.

Ограничения:

- Конкурентность: два пользователя одновременно → oversell. Эксперт должен поддерживать атомарный резерв (транзакции/optimistic lock/queues).
- Если складов много и распределённо, эксперт становится распределённым — сложнее обеспечить сильную консистентность.