**Федеральное государственное автономное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
университет
информационных технологий, механики и оптики»**

Факультет Программной Инженерии и Компьютерной Техники

Лабораторная работа №2

по дисциплине

«Архитектура программных систем»:

Выполнили:
Ястребов-Амирханов Алекси
Группа: P3332
ПРЕПОДАВАТЕЛЬ: Перл И.А.

Санкт-Петербург,

2025

Оглавление

Задание:

Из списка шаблонов проектирования GoF и GRASP выбрать 3–4 шаблона и для каждого из них придумать 2–3 сценария, для решения которых могут быть применены выбранные шаблоны. Сделать предположение о возможных ограничениях, к которым может привести использование шаблона в каждом описанном случае. Обязательно выбрать шаблоны из обоих списков.

Для выполнения задания выбраны следующие шаблоны:

- Стратегия (Strategy) – поведенческий шаблон из GoF.
- Наблюдатель (Observer) – поведенческий шаблон из GoF.
- Контроллер (Controller) – шаблон распределения обязанностей из GRASP.
- Эксперт по информации (Information Expert) – шаблон из GRASP.

Стратегия (Strategy, GoF)

Шаблон «Стратегия» применяется, когда объекту необходимо выбирать и менять алгоритм поведения во время выполнения программы. Его суть заключается в том, что различные варианты алгоритмов инкапсулируются в отдельные классы, реализующие общий интерфейс, а основной объект (контекст) делегирует выполнение нужному алгоритму. Это позволяет избавиться от громоздких условных операторов и легко изменять логику работы во время выполнения программы, не изменяя код контекста.

Сценарии применения (например, в играх):

- **Тактики NPC (искусственный интеллект):**

Контекст: `NPC` или `AIController`.
Интерфейс: `IBehaviourStrategy { void Execute(NPCContext ctx); }`.
Стратегии: `AggressiveStrategy`, `DefensiveStrategy`, `FleeStrategy`, `PatrolStrategy`, `AmbushStrategy`.
Применение: в зависимости от состояния (здоровье ниже 25%, численность противников > 3, цель далеко и т.д.) менеджер состояний подставляет нужную стратегию. Можно включать «переопределяемые» комбинации — например, `Patrol + Ambush` через композицию стратегий.

- **Режимы оружия/способностей:**

Контекст: `Weapon` или `Ability`.
Интерфейс: `IFireMode { void Fire(User u, Target t); }`.
Стратегии: `SingleShot`, `Burst`, `Auto`, `ChargedShot`
Применение: переключение режимов по нажатию кнопки; настраиваемые апгрейды добавляют новые стратегии без правки `Weapon`.

- **Алгоритмы навигации/движения:**

Контекст: `MovementController`.
Стратегии: `AStarNavigation`, `FlowFieldNavigation`, `SteeringBehaviors`, `PredictionDrivenMove`.
Применение: для больших карт можно подставлять лёгкий эвристический алгоритм на дальних дистанциях, а точный — вблизи целей; переключать при загрузке уровней под ограничения по CPU.

- **Графика и звук:**

Интерфейс: `IAnimationSmoothing`, `ISoundFilter`.
Стратегии: «линейная», «инерционная», «адаптивная» анимация; «компрессия-ACL», «low-latency» аудио-обработка.
Применение: выбирать профиль на лету в зависимости от производительности устройства или настроек пользователя.

Возможные ограничения:

Число классов — множество почти однотипных стратегий усложняет навигацию и сопровождение.
Сложность логики выбора — правила подстановки стратегий растут и дублируются, появляются тяжёлые условные конструкции.
Производительность и аллокации — частое создание инстансов стратегий (каждый кадр) вызывает нагрузку на сборщик мусора и падение FPS.
Проблемы со состоянием и потокобезопасностью — stateful-стратегии трудно переключать и безопасно использовать в многопоточном окружении.

Когда применять

- Когда поведение реально варьируется и ожидается добавление новых алгоритмов.
- Когда нужно подменять алгоритм в рантайме (режимы/апгрейды/профили производительности).
- Когда хочется отдельно тестировать и реиспользовать алгоритмы.

Когда не применять

- Если вариантов мало и они редко меняются — проще if/else.
- Если критична каждая аллокация и стратегия создаётся очень часто.
- Если стратегии должны держать сложное состояние, которое тяжело мигрировать.

Наблюдатель (Observer, GoF)

Шаблон «Наблюдатель» предназначен для оповещения множества объектов об изменениях состояния другого объекта без жесткой связи между ними. Объект-субъект хранит список наблюдателей и при изменении состояния уведомляет их. Это позволяет системе реагировать на события без прямых зависимостей между компонентами.

Сценарии применения

- **Система достижений:**
Игровые подсистемы (физика, бой, квесты) генерируют события о действиях игрока. Система достижений подписывается на эти события и анализирует их, принимая решение о выдаче награды. Источник события не зависит от логики достижений, что упрощает добавление новых условий и наград.

- **Обновление пользовательского интерфейса:**
При изменении состояния персонажа (здоровье, мана, очки) объект игрока уведомляет подписанные элементы интерфейса. HUD автоматически обновляет отображаемые значения, не вмешиваясь в игровую логику и не опрашивая состояние вручную.

- **Звуковые и визуальные эффекты:**
Одно игровое событие (взрыв, попадание, победа) может одновременно запускать несколько реакций: воспроизведение звука, показ анимации, запуск частиц. Все эти системы подписываются на событие и реагируют независимо друг от друга.

Возможные ограничения:

- **Синхронная обработка уведомлений:**
Если уведомления рассылаются последовательно, медленный или ресурсоёмкий наблюдатель может задержать выполнение субъекта и повлиять на производительность, что критично для игровых циклов.

- **Неявные зависимости и сложность отладки:**
Логика реакции на событие распределена по множеству наблюдателей, из-за чего становится трудно отследить, какие именно действия будут выполнены при возникновении события.

- **Управление подписками:**
При динамическом создании и уничтожении объектов важно корректно подписывать и отписывать наблюдателей. Ошибки в этом процессе могут приводить к утечкам памяти или вызовам методов у уже уничтоженных объектов.

- **Рост количества уведомлений:**
При большом числе наблюдателей одно событие может инициировать множество обработок, что увеличивает нагрузку на систему и может негативно сказаться на производительности.

Контроллер (Controller, GRASP)

Шаблон «Контроллер» предполагает выделение специального класса, отвечающего за обработку системных событий и координацию взаимодействия между объектами. Контроллер принимает входные сигналы и распределяет ответственность между другими компонентами системы.

Сценарии применения (например, в играх):

- **Контроллер уровня или сцены:**
Класс контроллера управляет жизненным циклом уровня: инициализацией объектов, запуском игрового процесса, постановкой на паузу и завершением уровня. Он координирует взаимодействие между персонажами, камерами, звуком и интерфейсом, не вмешиваясь в их внутреннюю реализацию.

- **Обработка пользовательского ввода:**
Контроллер получает события от устройств ввода (клавиатура, мышь, геймпад) и преобразует их в игровые команды. Это позволяет изолировать обработку ввода от логики персонажей и легко изменять схему управления или добавлять новые устройства.

- **Сетевой контроллер:**
В многопользовательских играх контроллер принимает сетевые сообщения, обрабатывает их и передает данные соответствующим игровым объектам. Он служит связующим звеном между сетевым уровнем и игровой логикой.

Возможные ограничения:

- **Перегрузка обязанностями:**
При неосторожном проектировании контроллер может начать выполнять слишком много функций и превратиться в «толстый» класс, что усложняет сопровождение и тестирование.

- **Повышенная связанность:**
Контроллер часто взаимодействует с большим количеством объектов. Изменения в структуре системы могут потребовать правок в контроллере, что снижает гибкость архитектуры.

- **Снижение модульности:**
Сильная концентрация логики управления в одном классе может привести к ослаблению ответственности других компонентов и нарушению принципов распределения обязанностей.

- **Избыточность для простых сценариев:**
В небольших или простых системах введение отдельного контроллера может быть неоправданным и привести к лишнему усложнению архитектуры.

Эксперт по информации (Information Expert, GRASP)

Шаблон «Эксперт по информации» рекомендует назначать ответственность тому классу, который обладает всей необходимой информацией для выполнения операции. Это позволяет сделать код более логичным и уменьшить количество зависимостей.

Сценарии применения (например, в играх):

- **Подсчёт очков и уровня персонажа:**
Класс персонажа хранит количество очков опыта и текущий уровень, поэтому логично реализовать методы расчёта уровня и прогресса непосредственно в этом классе. Это упрощает логику и устраняет необходимость во внешних вычислительных классах.

- **Проверка столкновений объектов:**
Игровой объект содержит данные о своих координатах, размерах и форме. Метод проверки столкновения с другим объектом логично размещать в этом же классе, так как он располагает всей необходимой информацией для вычислений.

- **Сохранение состояния игры:**
Объект, представляющий состояние игры, хранит данные о текущем прогрессе, настройках и параметрах мира. Реализация методов сериализации и сохранения данных в этом классе упрощает процесс сохранения и восстановления игры.

Возможные ограничения:

- **Перегрузка обязанностями класса:**
Назначение слишком большого количества операций одному объекту может привести к снижению связности и превращению класса в трудно поддерживаемый «монолит».

- **Жесткая связь данных и логики:**
Изменение алгоритмов обработки данных требует изменения самого класса, что снижает гибкость и усложняет повторное использование.

- **Ограниченная масштабируемость:**
При росте функциональности класса может потребоваться его рефакторинг и выделение вспомогательных классов, что усложняет развитие системы.

- **Сложность тестирования:**
Класс, совмещающий хранение данных и сложную логику обработки, труднее изолировать при модульном тестировании.

Выводы

В ходе выполнения лабораторной работы были рассмотрены и проанализированы шаблоны проектирования из каталогов GoF и GRASP, применяемые при проектировании архитектуры программных систем. Для каждого выбранного шаблона были описаны основные идеи, приведены практические сценарии использования, преимущественно из области разработки компьютерных игр, а также рассмотрены возможные ограничения и проблемы, возникающие при их применении.

Анализ показал, что шаблоны проектирования позволяют повысить гибкость, расширяемость и поддерживаемость программных систем за счёт правильного распределения ответственности между компонентами. В частности, шаблоны GoF ориентированы на решение типовых архитектурных задач и организацию взаимодействия объектов, тогда как шаблоны GRASP помогают принимать обоснованные решения при назначении обязанностей классам.

При этом использование шаблонов требует осознанного подхода, так как их чрезмерное или неуместное применение может привести к усложнению архитектуры, снижению производительности и ухудшению читаемости кода. Таким образом, шаблоны проектирования следует рассматривать как инструмент, эффективность которого напрямую зависит от контекста задачи и требований к системе.