Лекции по Технологиям программирования

Сверстал: Кузякин Никита Александрович

По лекциям ФПМИ

Плейлист с лекциями — тут

СОДЕРЖАНИЕ

1 Принципы SOLID

- **S** (Single Responsibility Principle) — один класс одна ответсвенность. Одна задача выполняется одним классом, и один класс выполняет одну задачу.
- **O** (Open-Closed Principle) — програмные сущности должны быть открыты для расширения и закрыты для изменнеия.
- **L** (Liskov Substitution Principle) — объекты в программе должны быть заменяемы их подтипами (дочерними классами) поведение не должно нарушиться.
- **I** (Interface Segregation Principle) — много различных интерфейсов для разных клиентов (к примеру, десктоп и мобильный клиет) лучше, чем один общий интерфейс.
- **D** (Dependency Inversion Principle) — зависимости нужно строить на абстракциях, а не на реализациях. Если есть объект A, он реализовывает интерфейс A', и есть объект B и он реализует интерфейс B',то объекты A и B должны взаимодействовать по средствам интерфейсов A' и B'. К примеру: есть светлая и темная цветовая тема, а также операционные системы windows и linux. У нас должно быть 2 класса тема и операционная система. Интерфейс темы должен возвращать цвета соответствующей темы (темной или светлой), а интерфейс операционной системы должен возвращать фигуры, которые должны отрисовываться для иконок. Эти иконки будут получать цвет из интерфейса темы.

  Если бы мы строили зависимость на реализациях, то нам бы пришлось писать 4 варианта реализации: светлая тема windows, светлая тема linux, темная тема windows, темная тема linux.

Теорема 1. **Закон Деметры** — у нас есть объекты A, B, C. A имеет доступ к B, а B к C. Тогда объект A не должен иметь доступ к объекту C на прямую. Если объекту A нужна информация об объекте C, он должен обратиться к объекту B и получить от него информацию о C.

2 Архитектура ПО

Критерии архитектуры ПО

- Надежность — востребованей программа, которая более стабильна. К примеру, одна программа работает за 2 секунды, но раз в 10 секунд выдает неправильный результат, и есть другая программа, которая работает 5 секунд, но всегда выдает верный результат. Вторая программа будет востребованей первой.
- Безопасность — чем сложнее получить чужие пользовательские данные из ПО, тем оно лучше. Важно разделять важные данные и неважные, не нужно делать оптимизации ради оптимизаций. К примеру, не нужно шифровать информацию о том, сколько человек провел времени в приложении.
- Производительность — чем производительней ПО, тем лучше.
- Масштабируемость — ПО должно быть легко масштабируемой. Те легко расширяемой для большего числа пользователей, по числу запросов, по числу хранимой информации и тд.
- Гибкость — чем проще вносить изменения и устранять ошибки в системе (меньше модифицировать смежные подсистемы), тем лучше.

Критерии хорошей проектировки архитектуры ПО

- Масштабируемость процесса разработки — если вы пишете ПО не один, то чем лучше будет организован план разработки и совместных релизов, тем продуктивней будет написание кода.
- Тестируемость — нужно заранее придумать как будет тестироваться ПО. К примеру, позаботиться о тестах с большими данными или тестах на пользователях из разных регионов.
- Возможность повторного использования — хорошо написанная подсистема должна легко интегрироваться в другие проекты, при этом для ее работы не нужно переносить смежные блоки в новый проект.
- Сопровождаемость — перед написанием ПО нужно задуматься о том: как оно будет обновляться, как вы будете получать bug report (отчет об ошибках) от пользователя, как быстро вы сможете исправить ошибки у пользователя.

Критерии хорошей проектировки архитектуры ПО

- Масштабируемость процесса разработки — если вы пишете ПО не один, то чем лучше будет организован план разработки и совместных релизов, тем продуктивней будет написание кода.
- Тестируемость — нужно заранее придумать как будет тестироваться ПО. К примеру, позаботиться о тестах с большими данными или тестах на пользователях из разных регионов.
- Возможность повторного использования — хорошо написанная подсистема должна легко интегрироваться в другие проекты, при этом для ее работы не нужно переносить смежные блоки в новый проект.
- Сопровождаемость — перед написанием ПО нужно задуматься о том: как оно будет обновляться, как вы будете получать bug report (отчет об ошибках) от пользователя, как быстро вы сможете исправить ошибки у пользователя.

Критерии плохой архитектуры

- Жесткость — тяжело изменить
- Хрупкость — изменения нарушают работу других систем.
- Неподвижность — тяжело извлечь(удалить) модули(данные).

Определение 1. **Сильная сопряженность внутри модуля (High Cohesion)** — внутри модуля должны находиться только сильно сопряженные подсистемы. Слабо сопряженную подсистему лучше всего вынести за модуль, благодаря этому при обновление не придется затрагивать ее или модуль.

Определение 2. **Low Coupling** — Модули независимые друг от друга слабо связаны. Можно сказать, что модули представляют собой компоненты сильной связанности. Тогда принцип Low Coupling, говорит о том, что между этими компонентами должно быть как можно меньше мостов(связей).

Замечание 1. **Обещать что-то пользователю нужно аккуратно, а то он и напомнить может.**

3 Методы отладки ПО

Категории ошибок:

- критичные
- срочные
- несрочные
- "особенности реализации"

Что нужно делать после отладки? Проверить, что ошибка исправлена у вас, проверить точно ли ошибка исправлена у клиента.

Критерии критичности ошибок:

- какой процент пользователей пострадал
- на сколько большая часть приложения не работает
- происходит ли порча (потеря) данных
- сложность обхода

3.1 Воспроизведение дефекта

Суть данного этапа заключается в том, что нужно поймать точно такую же ошибку, которая возникла у пользователя,а не просто завершить программу с ошибкой.

Порядок воспроизведение дефекта

1. Проверка соответствия версия ПО с ошибкой и ваша.
2. Проверка соответствуют ли настройки ПО с ошибкой и ваша.
3. Проверка соответствуют ли данные ПО с ошибкой и ваша.
4. Точное воспроизведение действий пользователя (сценария ошибки).

Для быстрой проверки этих данных нужно тщательно собирать логи ошибок.

Замечание 2. По умолчанию пользователь не хочет обновляться.

3.2 Анализ дефекта

Определение 3. **Root Cause (суть проблемы)** — нужно исправлять не следствия ошибки, а саму суть. К примеру, если у вас выводится неправильно сумма 10 + 15, то нужно исправлять функцию суммирования, а не конкретно сумму для этих двух чисел.

Условия возникновения — когда происходит ошибка, нужно проверить все смежные сценарии.

Область повреждения — ошибка в одном месте может сломать огромное количество сценариев.

Кто привнес ошибку — имея такие данные можно понять, какие именно системы требуют более тщательного контроля, каких тестов нехватает и тд.

3.3 Дизайн исправления дефекта

Технический дизайн — как ошибка будет исправлена в коде.

Архитектурный дизайн — внедрение новых модулей, которые смогут предотвратить появление ошибки.

Технологический дизайн — stack технологий устарел, и для исправления требуется переход на новый stack технологий.

После исправления важно провести ревью и согласование, ведь вы могли затронуть подсистемы, которые используют / редактируют другие люди.

3.4 Исправление дефекта

Теорема 2. **Главное правило.** При исправление дефекта, не привнеси новые дефекты.

Не всегда ожидаемое поведение для вас и для пользователя совпадает. Поэтому важно получить обратную связь с пользователем, что его ошибка исправлена.

Очень важно документировать все изменение в системе, особенно исправление ошибок.

Перед загрузкой новой версии важно проверить все, даже те вещи, которые не менялись, даже на старых тестах и тд. Те происходит полноценное тестирование системы.

Также важно проверить корректность обновления.

Замечание 3. **Иногда сложность исправления ошибки на столько велика, что проще сказать пользователю путь обхода ошибки.**

3.5 Техники отладки

Запуск в отладчике

Самый простой способ это запуск программы в отладчике.

Виды отладкиков:

- софтверный (как в visual studio, pycharm).

- "железный— подключается напрямую к системе, на которой выполняется процесс и выводит данные на ваше устройство.
- Удаленный дебагер — посылает данные на ваше устройство с устройства, на котором происходит задача (к примеру, сервер посылает данные на локальную машину).

Анализ кода, без запуска программы

Данный метод заключается в анализе кода, без запуска программы. Нужно самому понять к чему приводят вызовы процедур, в каком порядке они выполняются и тд.

Анализ поведения программы

Данный метод заключается в построение причинно следственных связей для поиска ошибки. Делается предположение о причине ошибки, отталкиваясь от него делается предположение о сути ошибки и тд, пока вы не добираетесь для корня проблемы. Также в данном методе применяют упрощение сценариев тестирования и уменьшения данных, на которых тестируется ПО.

Данный метод помогает локализовать ошибку в крупных системах.

UNIT - тестирование

Предполагает тестирование каждой отдельной функции или каждого элемента системы.

Определение 4. **Процент покрытия кода** — процент функций, на которые написаны тексты. Чем больше покрытие кода, тем лучше.

Определение 5. **Self test** — после установки приложение запускает тест самого себя. Что гарантирует корректность установки.

Прототипирование

Метод, при котором для тестировки, модуль выносится в отдельное пространство и для него дописывается минимальный набор функций для запуска.

Данный метод хорошо подходит в случае, когда невозможно произвести полное тестирование системы вместе с данным модулем.

Отадка с помощью дампов

Определение 6. **Файл дампа** — это моментальный снимок, показывающий выполнявшийся процесс и загруженные для приложения модули в определенный момент времени. Дамп со сведениями о куче также содержит моментальный снимок памяти приложения на этот момент.

В основном дампы используются для отладки проблем на компьютерах, к которым у разработчиков нет доступа. Если вы не можете воспроизвести на своем компьютере аварийное завершение или зависание программы, возникшие на компьютере клиента, вы можете записать файл дампа с его компьютера. Дампы также создаются тест-инженерами, чтобы сохранить данные для дополнительного тестирования.

Отладка с помощью перехватов

В данном методе, если есть предположение, что конкретная функция работает не правильно, то существуют специальные ПО, которые при вызове этой функции перехватят все передаваемые значения и тд.

Данный метод используется для ускорения тестировки, тк ПО в режиме дебагера работает гораздо дольше.

Определение 7. **Профилирование кода** — Сбор характеристик работы программы, таких как время выполнения отдельных фрагментов, число верно предсказанных условных переходов, число кэш-промахов, количество затрачиваемой оперативной памяти и т. д.

Определение 8. **Кэш-промах (cash- miss)** — Некорректная логика работы кэша.

Трансляция кода

Метод, при котором код переносится на другой язык. К примеру с C++ на ассемблер для того, чтобы увидеть как это работает на более низком уровне. Или наоборот с ассемблера на C++, те на более высокий уровень, для того чтобы логика процесса была более очевидной.

Обратное движение по алгоритму

Данный подход полезен, когда сценарий достаточно сложный, тк уровень сложности и вероятности ошибки растет ближе к концу кода. Соответственно, если начать отладку с конца, то вероятнее вы быстрее доберетесь до ошибки.

4 Проектирования ПО

4.1 Этапы проектирование ПО

1. формирование требований
2. Разработка концепции
3. Техническое задание
4. Эскизный проект
5. Технический проект
6. Рабочая документация
7. Поставка и ввод в действие
8. Сопровождение

Формирование требований — общение с клиентом (чаще всего сам клиент не будет пользоваться системой), общение с пользователем, анализ прикладной области, формирование оценок требуемой производительности. Важно понимать: для кого этот софт и какую проблему он решает.

Разработка концепции — изучение объекта автоматизации, проведение необходимых научно-исследовательских работ, разработка вариантов концепции. **Важно, подготовить отчетность по данному этапу**. Также тут уточняется: формат поставки (приложение андроид, веб сайт и тд), целевое оборудование (оборудование на котором должно работать ПО), выбор стека технологий.

Техническое задание — описание системы, описание функциональности, описание требований к системе и функциональности, описание сценариев использования, условия сдачи. **Техническое задание очень важно как для программиста, так и для заказчика**. Тк заказчику оно дает гарантию того, что ПО будет выполнять все, что прописано в ТЗ. А программисту дает гарантию того, что если выполнено все ТЗ, то ПО будет принято заказчиком.

Эскизный проект — разработка прототипов частей системы (не всех, а самых важных и сложных), тестирование, оценка производительности и качества, MVP.

Технический проект — разработка частей системы, написание документации пользователя, тестирование, разработка заданий на проектирование и реализацию смежных систем, оценка производительности.

Рабочая документация — описание всех сценариев использования, описание логики работы, описание производительности, примеры использования, обучающие мероприятия.

Ввод и действие — подготовка объекта автоматизации, подготовка персонала, проведение предварительных испытаний, опытная эксплуатация, приемочные испытания (проверка всех ограничений из ТЗ).

4.2 UML

Определение 9. **UML** — унифицированный язык моделирования (Unified Modeling Language). Это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования Его можно использовать для визуализации, спецификации конструирования и документирования программных систем.

Основные цели дизайна UML:

- Предоставить пользователям готовый, выразительный язык визуального моделирования, чтобы они могли разрабатывать и обмениваться осмысленными моделями.
- Обеспечить механизмы расширяемости и специализации для расширения основных понятий.
- Быть независимым от конкретных языков программирования и процессов разработки.
- Обеспечить формальную основу для понимания языка моделирования.
- Поощрять рост рынка объектно-ориентированных инструментов.
- Поддержка высокоуровневых концепций разработки, таких как совместная работа, структуры, шаблоны и компоненты.
- Интегрировать лучшие практики.

Диаграммы UML подразделяют на два типа — это структурные диаграммы и диаграммы поведения.

Структурные диаграммы показывают статическую структуру системы и ее частей на разных уровнях абстракции и реализации, а также их взаимосвязь. Элементы в структурной диаграмме представляют значимые понятия системы и могут включать в себя абстрактные, реальные концепции и концепции реализации.

Диаграммы поведения показывают динамическое поведение объектов в системе, которое можно описать, как серию изменений в системе с течением времени.

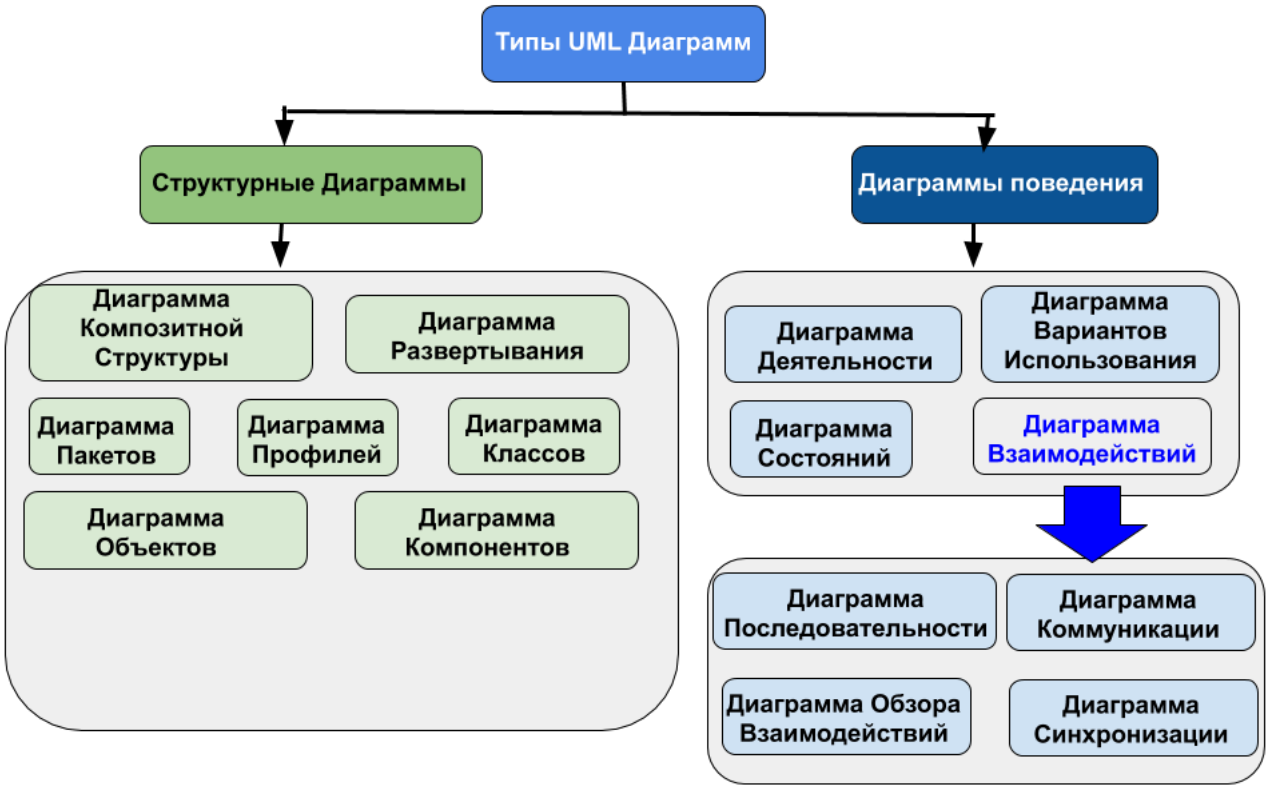

Типы UML Диаграмм
Структурные Диаграммы
Диаграммы поведения
Диаграмма Композитной Структуры
Диаграмма Развертывания
Диаграмма Пакетов
Диаграмма Профилей
Диаграмма Классов
Диаграмма Объектов
Диаграмма Компонентов
Диаграмма Деятельности
Диаграмма Вариантов Использования
Диаграмма Состояний
Диаграмма Взаимодействий
Диаграмма Последовательности
Диаграмма Коммуникации
Диаграмма Обзора Взаимодействий
Диаграмма Синхронизации

Диаграмма классов — включает в себя три компонента: Ассоциация, которая представляет отношения между экземплярами типов, наследование, и агрегация, которая представляет из себя форму композиции объектов в объектно-ориентированном дизайне.

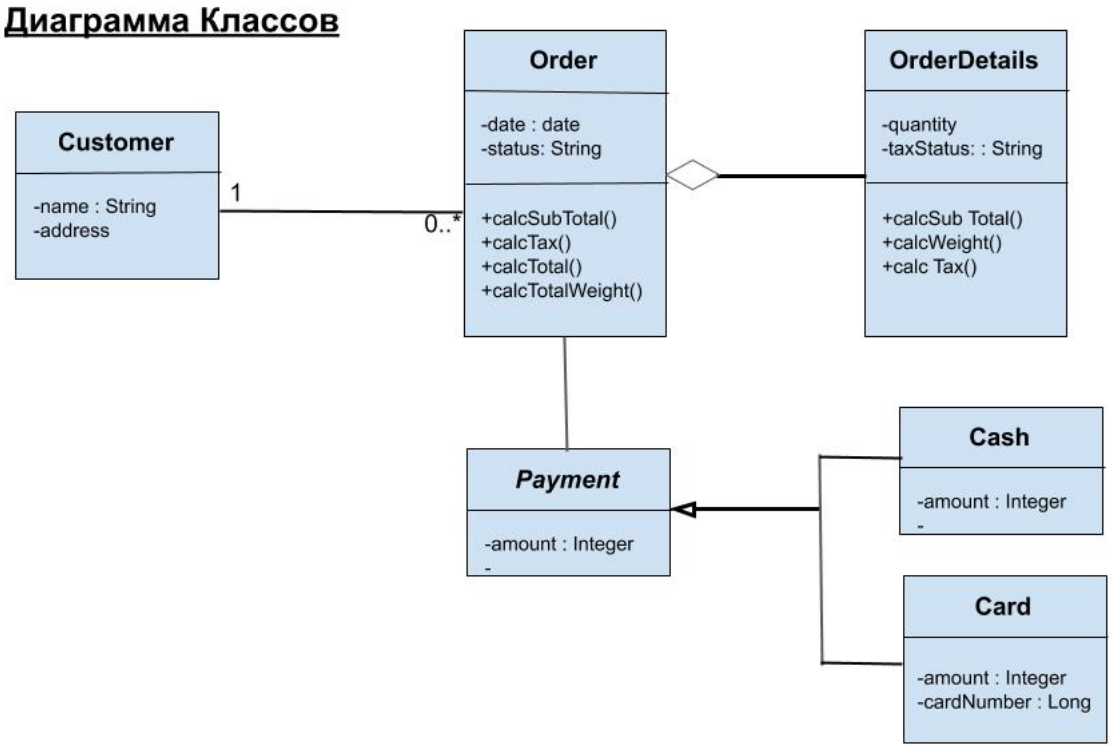


Диаграмма компонентов — показывает, как компоненты соединяются вместе для формирования более крупных компонентов или программных систем.

Диаграмма Компонентов

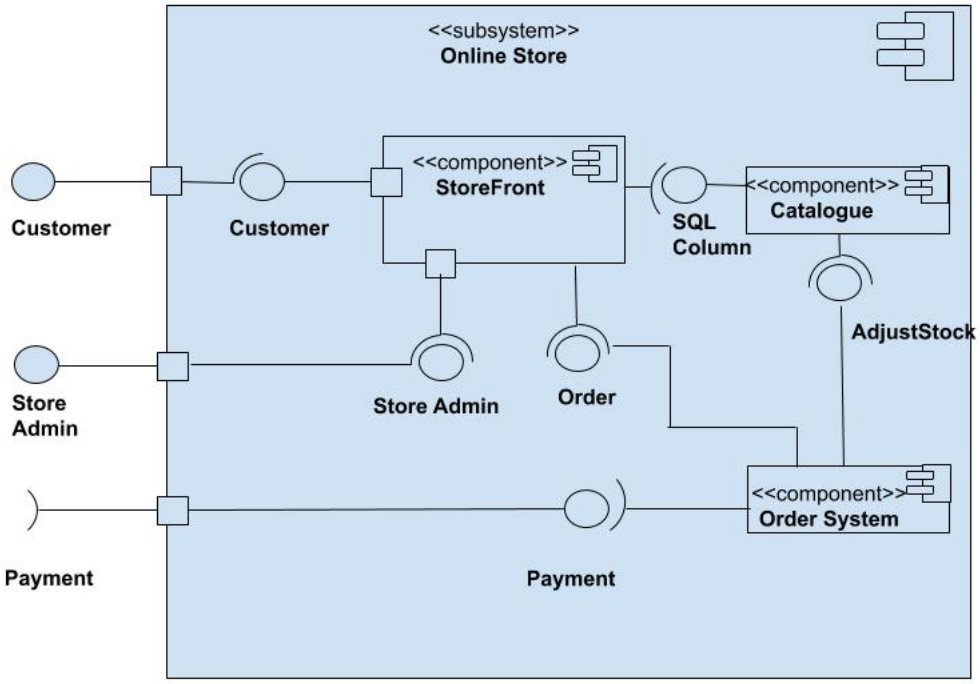


Диаграмма развертывания — показывает архитектуру системы, как развертывание (дистрибуции) программных элементов в физическом мире, которые являются результатом процесса разработки.

Диаграмма Развертывания

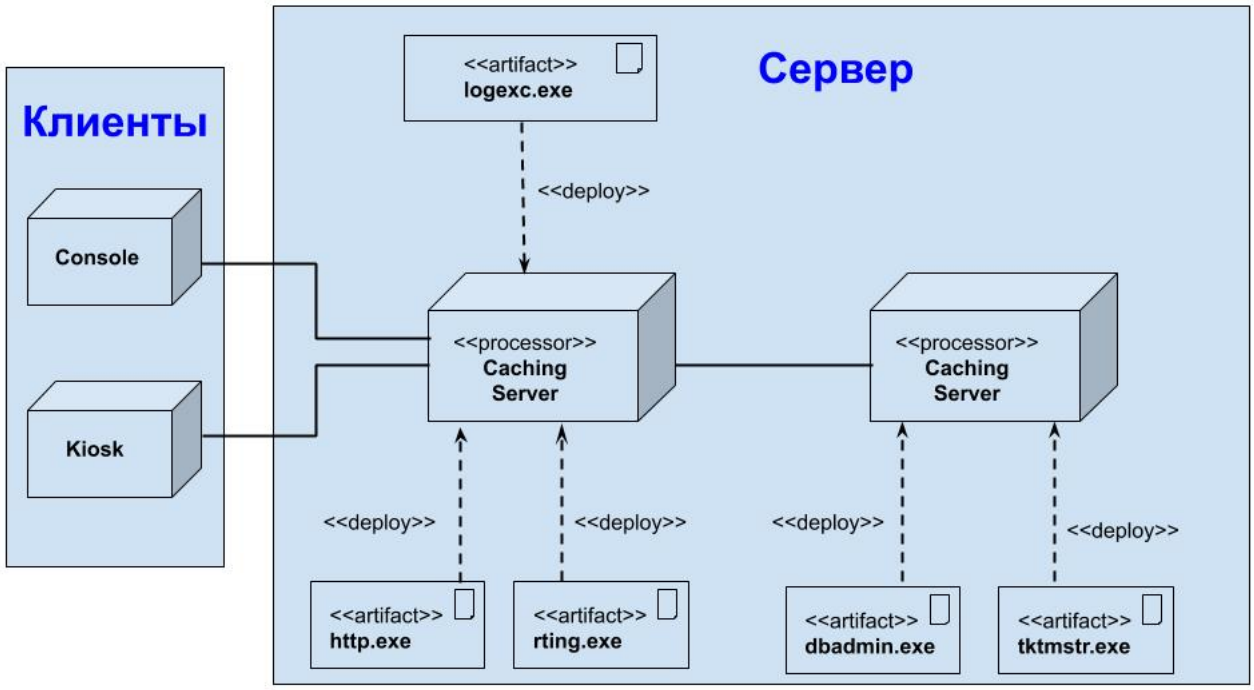


Диаграмма объектов — является экземпляром диаграммы класса. Она показывает снимок подробного состояния системы в определенный момент времени.

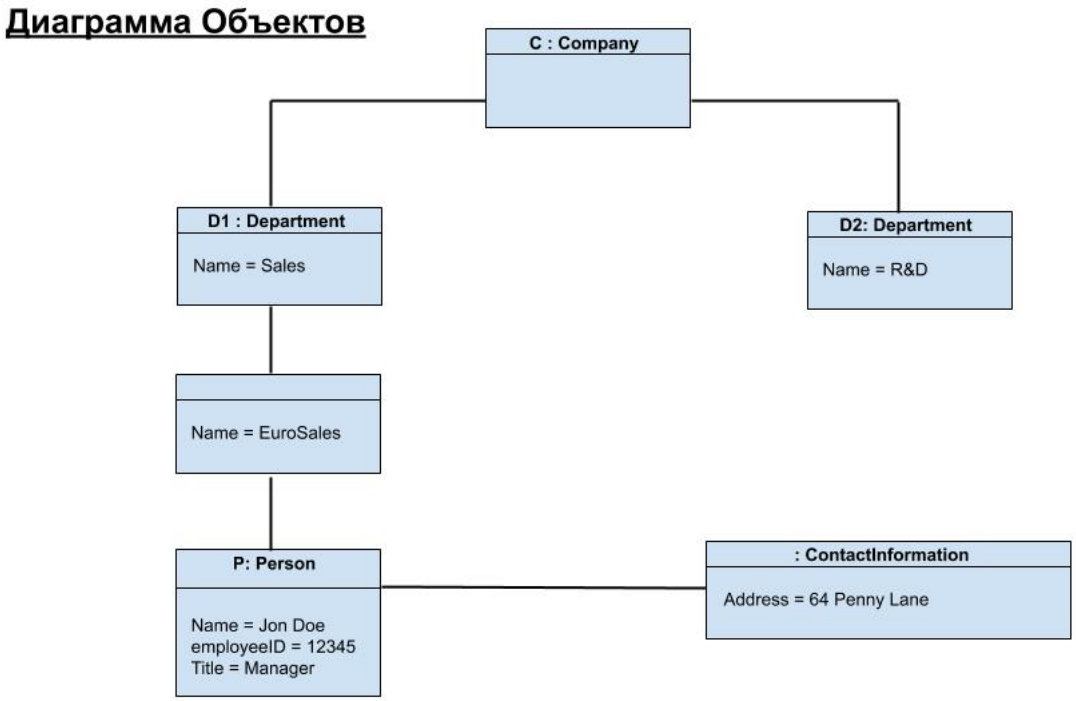


Диаграмма пакетов — это структурная схема, которая показывает пакеты и зависимости между ними.

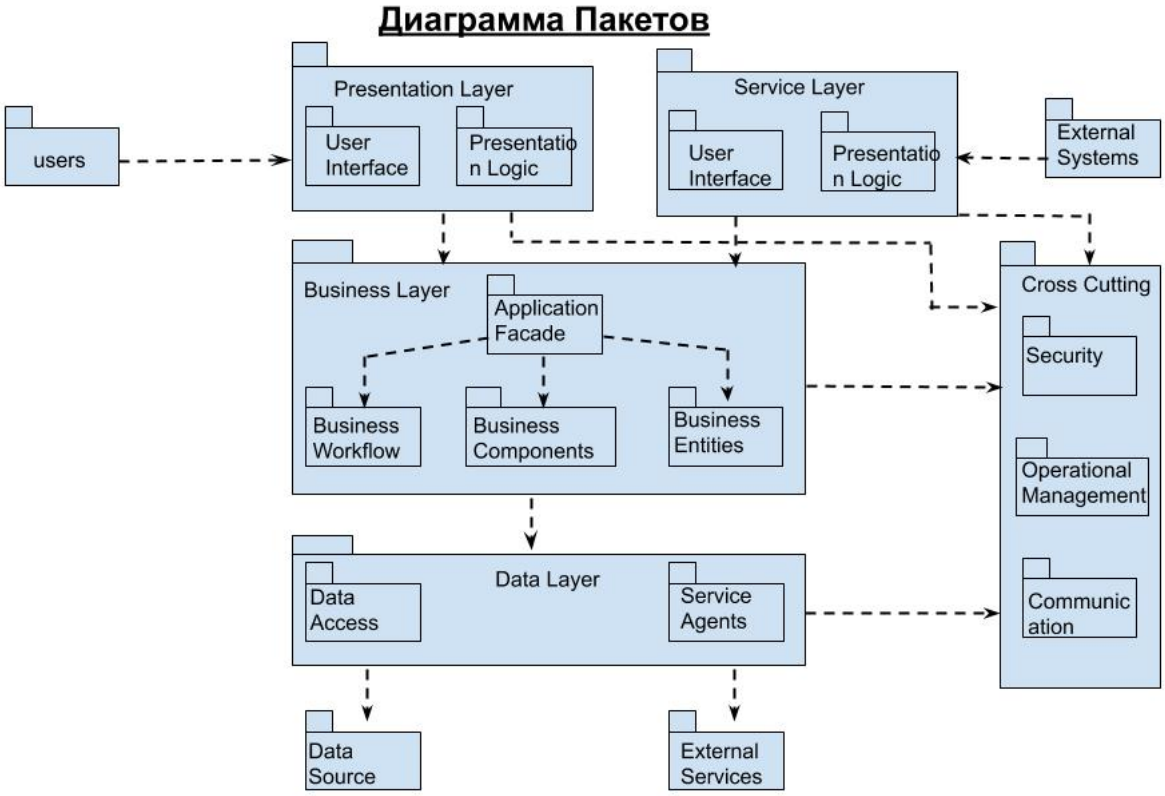


Диаграмма составной структуры — аналогична диаграмме классов и является своего рода диаграммой компонентов, используемой в основном при моделировании системы на микроуровне, но она изображает отдельные части вместо целых классов. Это тип статической структурной диаграммы, которая показывает внутреннюю структуру класса и взаимодействия, которые эта структура делает возможными.

Диаграмма Составной Структуры

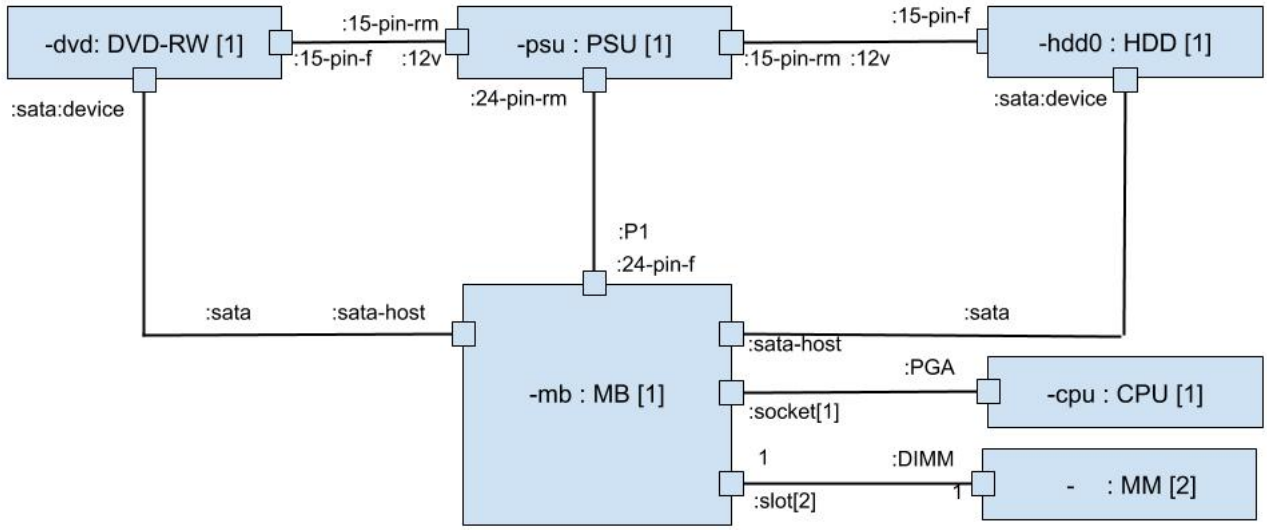


Диаграмма профилей — позволяет нам создавать специфичные для домена и платформы связи и определять отношения между ними.

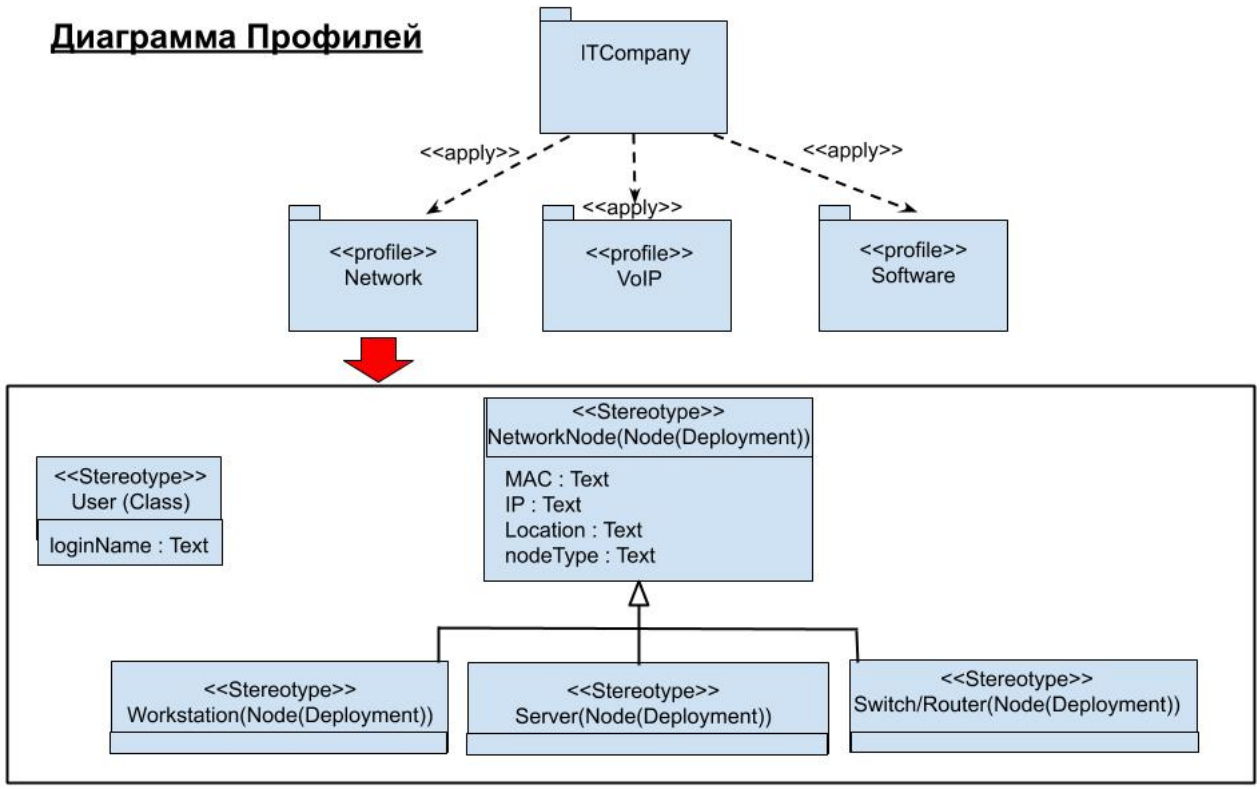


Диаграмма прецедентов — описывает функциональные требования системы с точки зрения прецедентов.

Диаграмма Прецедентов

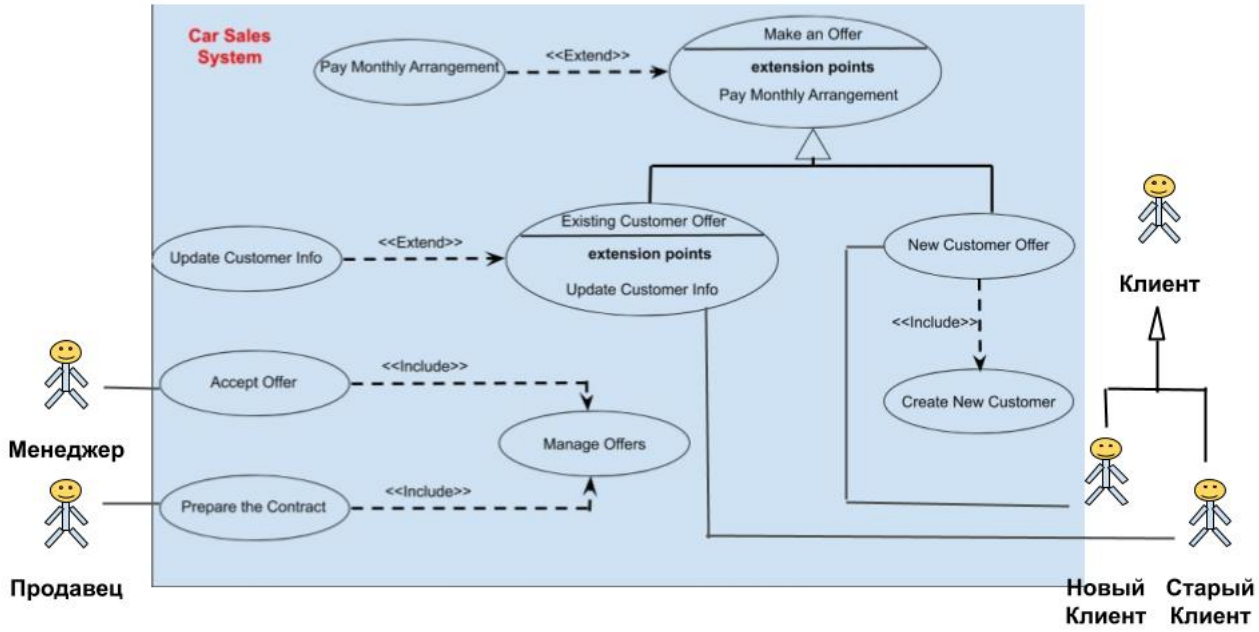


Диаграмма деятельности — представляет собой графическое представление рабочих процессов поэтапных действий и действий с поддержкой выбора, итерации и параллелизма.

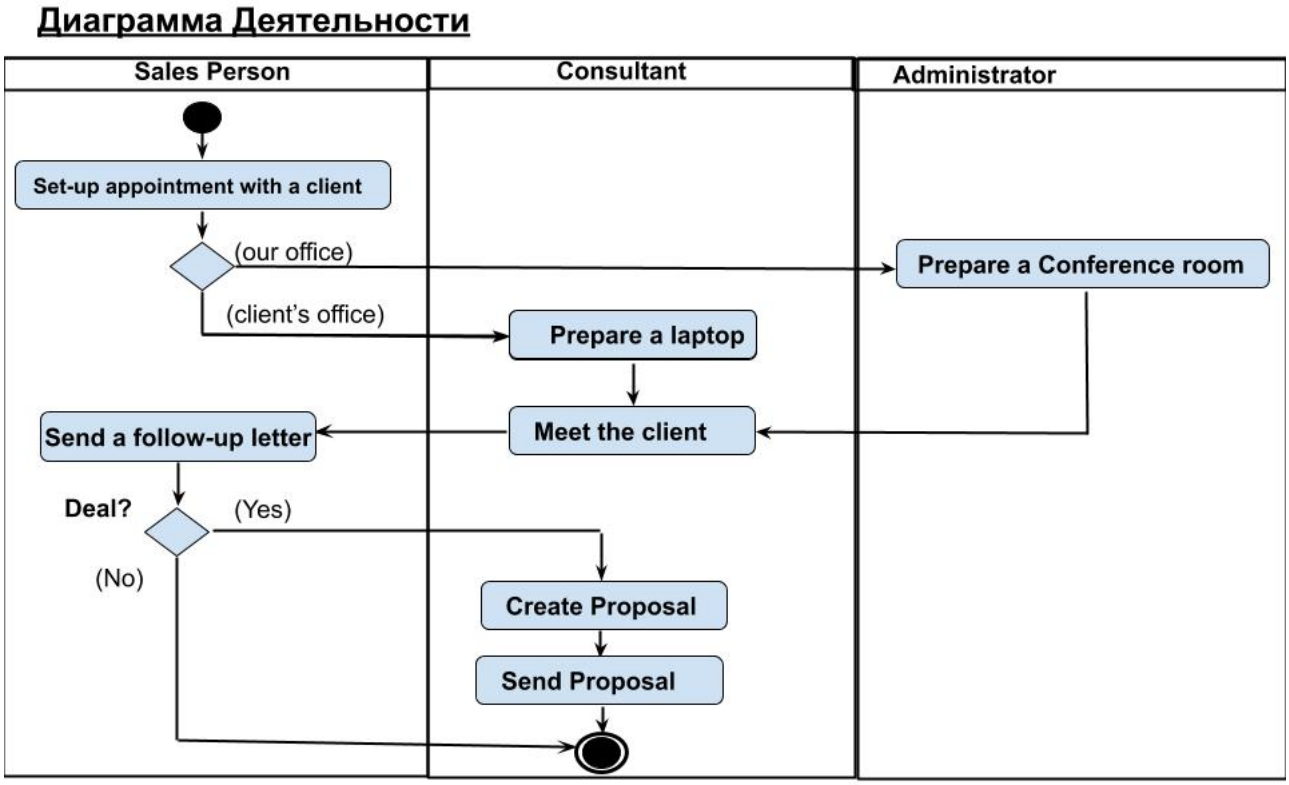


Диаграмма состояний — отображает разрешенные состояния и переходы, а также события, которые влияют на эти переходы. Она помогает визуализировать весь жизненный цикл объектов.

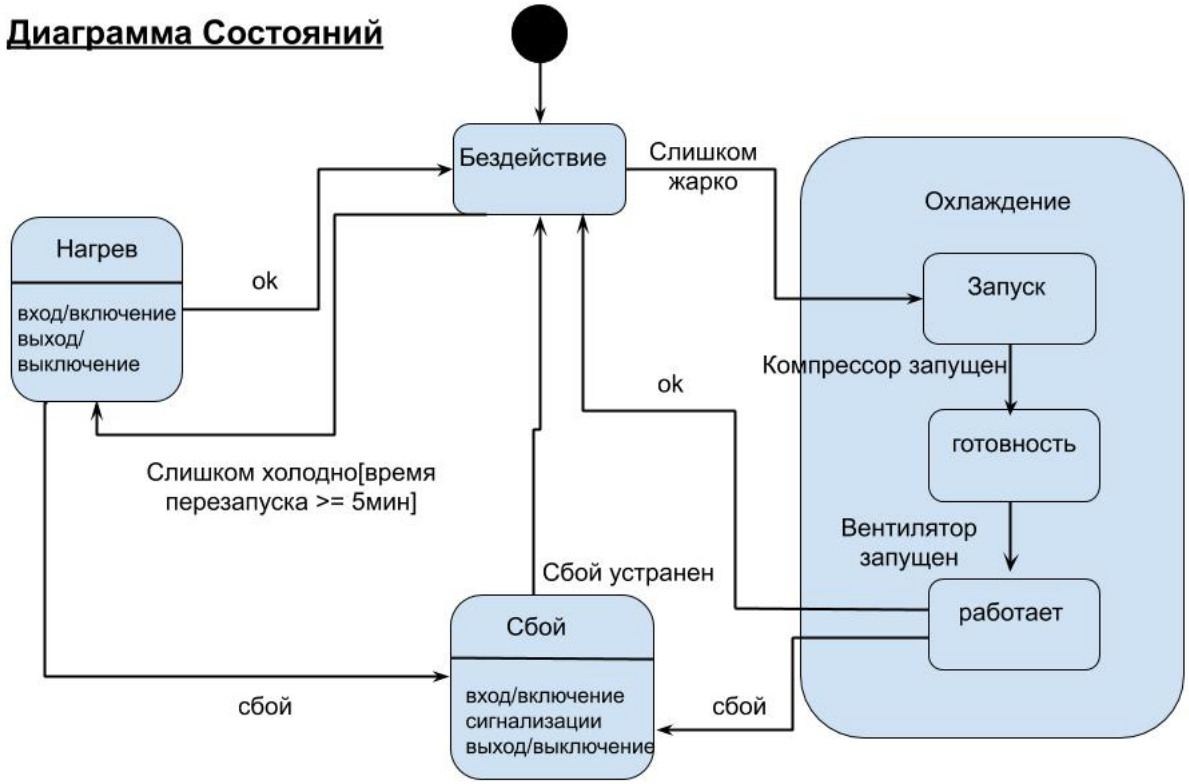


Диаграмма последовательности — моделирует взаимодействие объектов на основе временной последовательности.

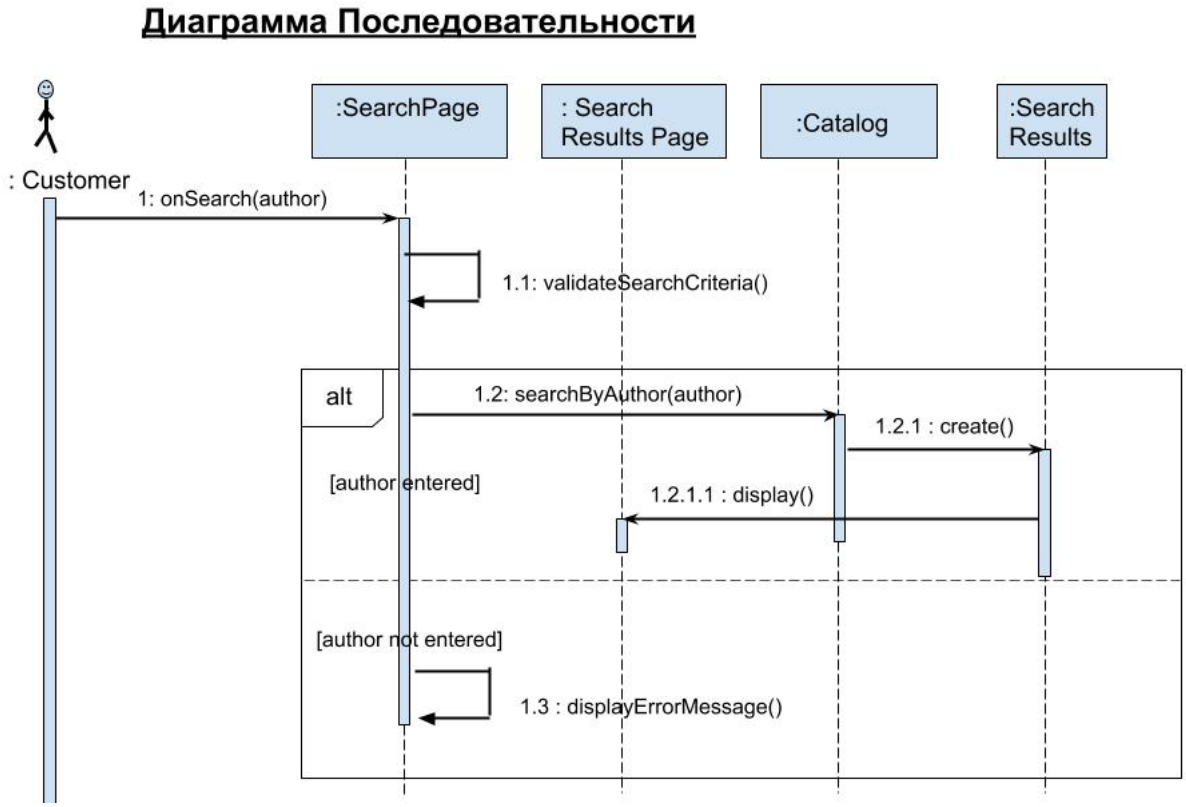


Диаграмма коммуникации — как и диаграмма последовательности, также используется для моделирования динамического поведения прецедента. Если сравнивать с диаграммой последовательности, Диаграмма коммуникации больше сфокусирована на показе взаимодействия объектов, а не временной последовательности.

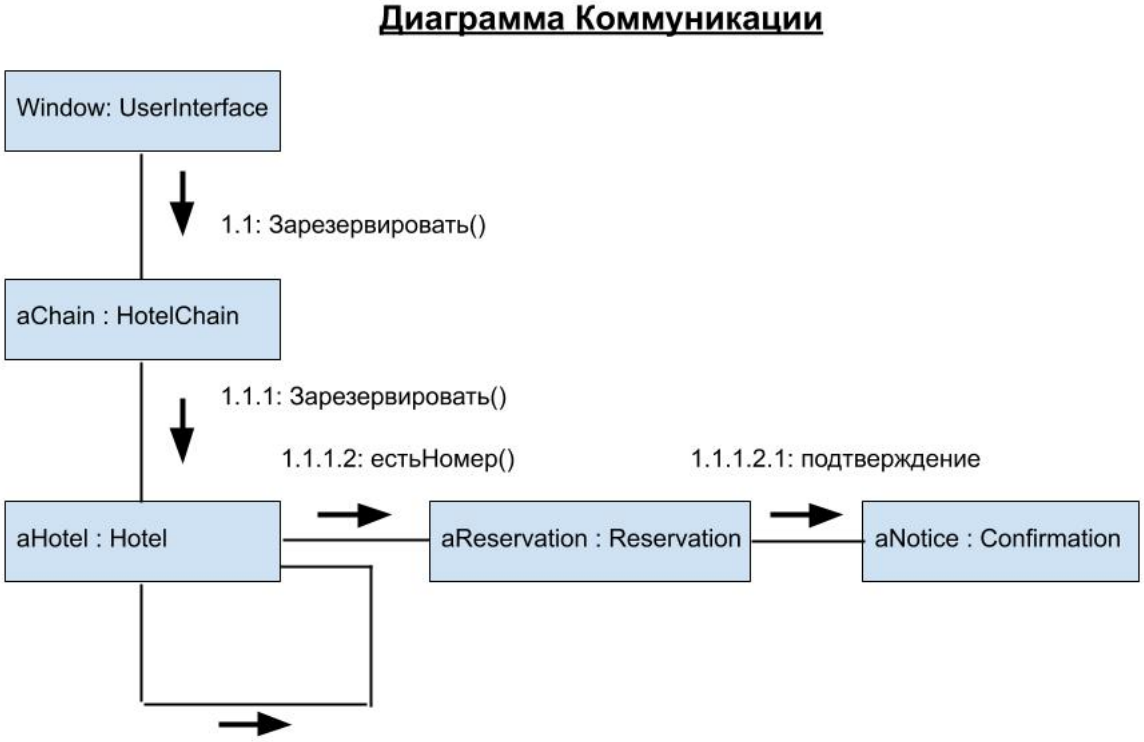
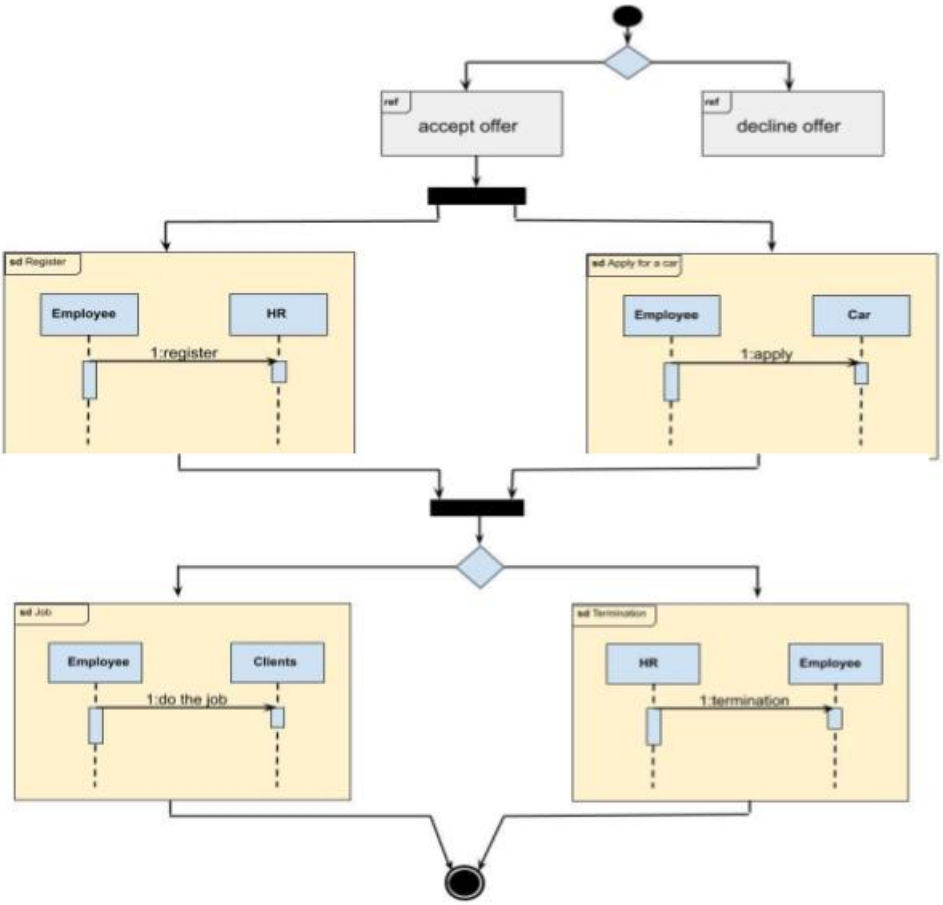
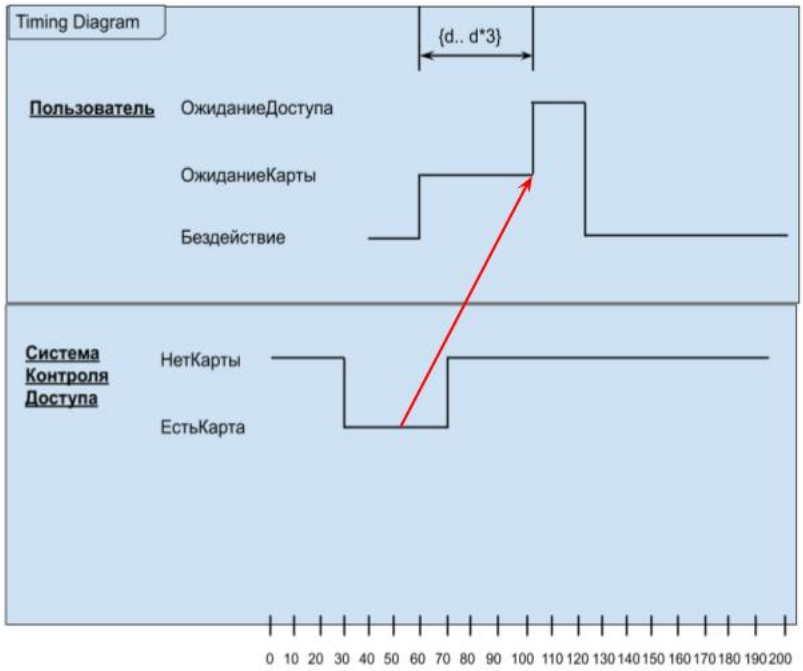


Диаграмма обзора взаимодействия — фокусируется на обзоре потока управления взаимодействиями. Это вариант Диаграммы деятельности, где узлами являются взаимодействия или события взаимодействия.



Временная диаграмма — показывает поведение объекта в данный период времени. По сути — это особая форма диаграммы последовательности и различия между ними состоят в том, что оси меняются местами так, что время увеличивается слева направо, а линии жизни отображаются в отдельных отсеках, расположенных вертикально.

Временная Диаграмма



4.3 Методологии разработки

Определение 10. **Методология разработки** — набор методов, принципов и правил, которые используются для постановки задачи, планирования, контроля и в конечном итоге — для достижения поставленной цели.

Водопадная модель

- все стадии проекта идут строго друг за другом
- следующая стадия не начинается, пока не закончится предыдущая
- план разрабатывается сразу на весь проект

Рисунок 1 – Водопадная модель разработки ПО

Главный минус — это возможное изменение требований, что может полностью обесценить уже сделанную работу.

Рисунок 2 – V модель разработки ПО

Данная модель состоит из двух главных компонентов верификация и валидация. Левая ветка это — верификация, а правая — валидация.

Инкрементная модель

В данной модели сначала создается базовая функциональность ПО (MVP), затем добавляются доп возможности, и тд проект разрастается. В ходе разработки каждого инкремента у нас получается отдельное работоспособное ПО. Каждый инкремент заранее определен. Подходит, когда есть четкий план развития продукта.

Рисунок 3 – Инкрементная модель разработки ПО

Итерационная модель

Каждая новая версия также работоспособная, как и в инкрементной. Но тут мы заранее не знаем, какой будет следующая итерация. Мы смотрим на отзывы пользователей и прочии параметры и отталкиваясь от них решаем, что будет разрабатываться на следующей итерации. Подходит, когда планируется очень долгий проект, который будет разрабатываться не один год.

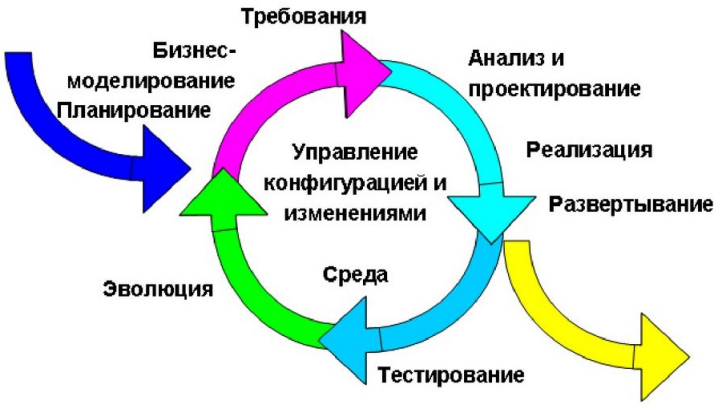


Рисунок 4 – Итерационная модель разработки ПО

Спиральная модель

Спиральная модель отличается тем, что на каждой итерации происходит анализ рисков. Также на любом этапе можно отказаться от разработки и выкинуть в релиз то, что есть сейчас. Хорошо подходит для разработки больших проектов, в которых не совсем понятно, к чему стремится продукт.

Спиральная модель

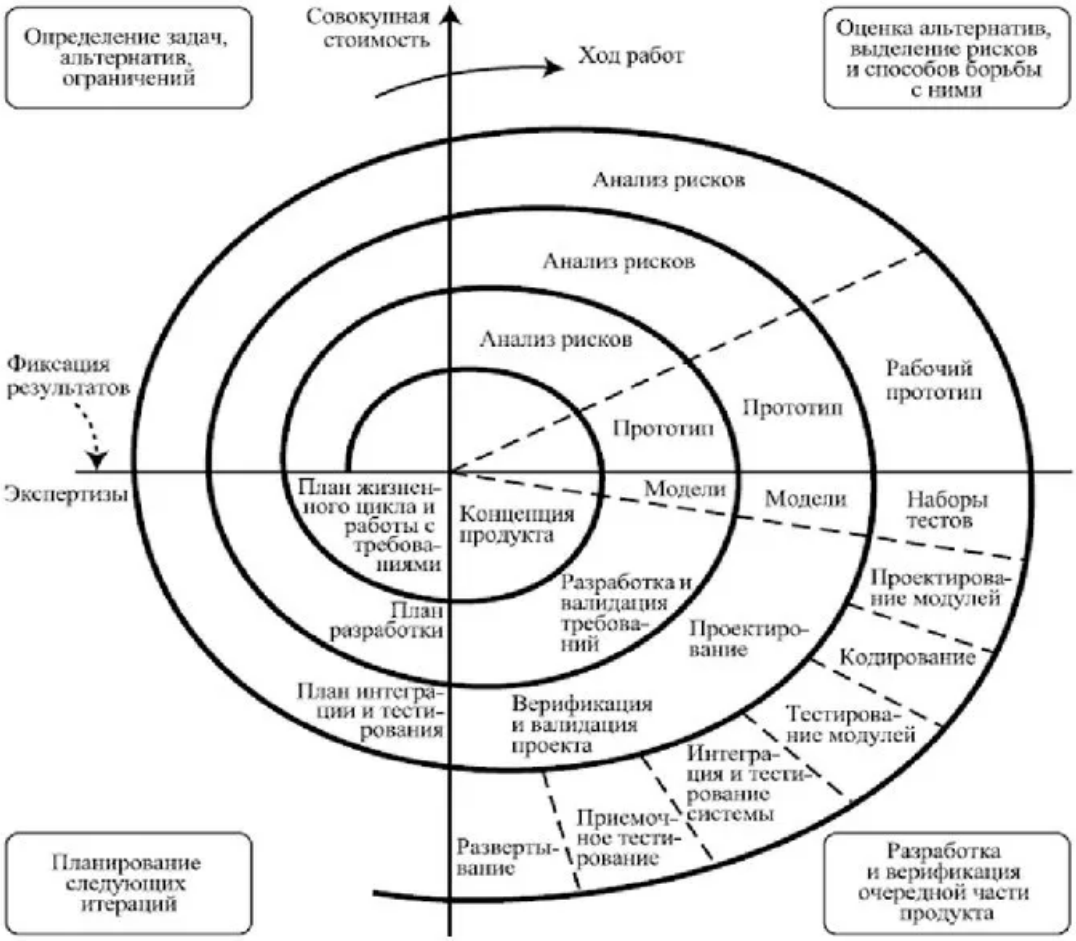


Рисунок 5 – Спиральная модель разработки ПО

RAD модель

Определение 11. **RAD (Rapid Application Development model)** — быстрая разработка предложений. Отличается тем, что различные модули разрабатываются различными командами, жесткое ограничение времени, различные модули интегрируются в один. Часто используется автоматическая сборка и генерация кода.

Этапы:

- Бизнес - моделирование
- Анализ и создание модели данных
- Анализ и создание процесса

- Автоматическая сборка приложения
- Тестирование

Agile модель

Это семейство гибких методологий разработок для которых характерно:

- Короткие итерации
- Разные метрики качества работы
- Много разных конкретных подходов

Данный подход рисково использовать для крупных проектов, тк сроки тут явно не указаны.

Примеры: Канбан, Scrum и тд.

Agile - манифест:

- люди и взаимодействие важнее процессов и инструментов
- работающий продукт важнее хорошей документации
- сотрудничество с заказчиком важнее условий контракта
- готовность к изменениям важнее первоначального плана

На разных этапах разработки могут использоваться различные методологии.

5 Continuous integration

Continuous Integration — это практика разработки программного обеспечения, которая заключается в слиянии рабочих копий в общую основную ветвь разработки несколько раз в день и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной.

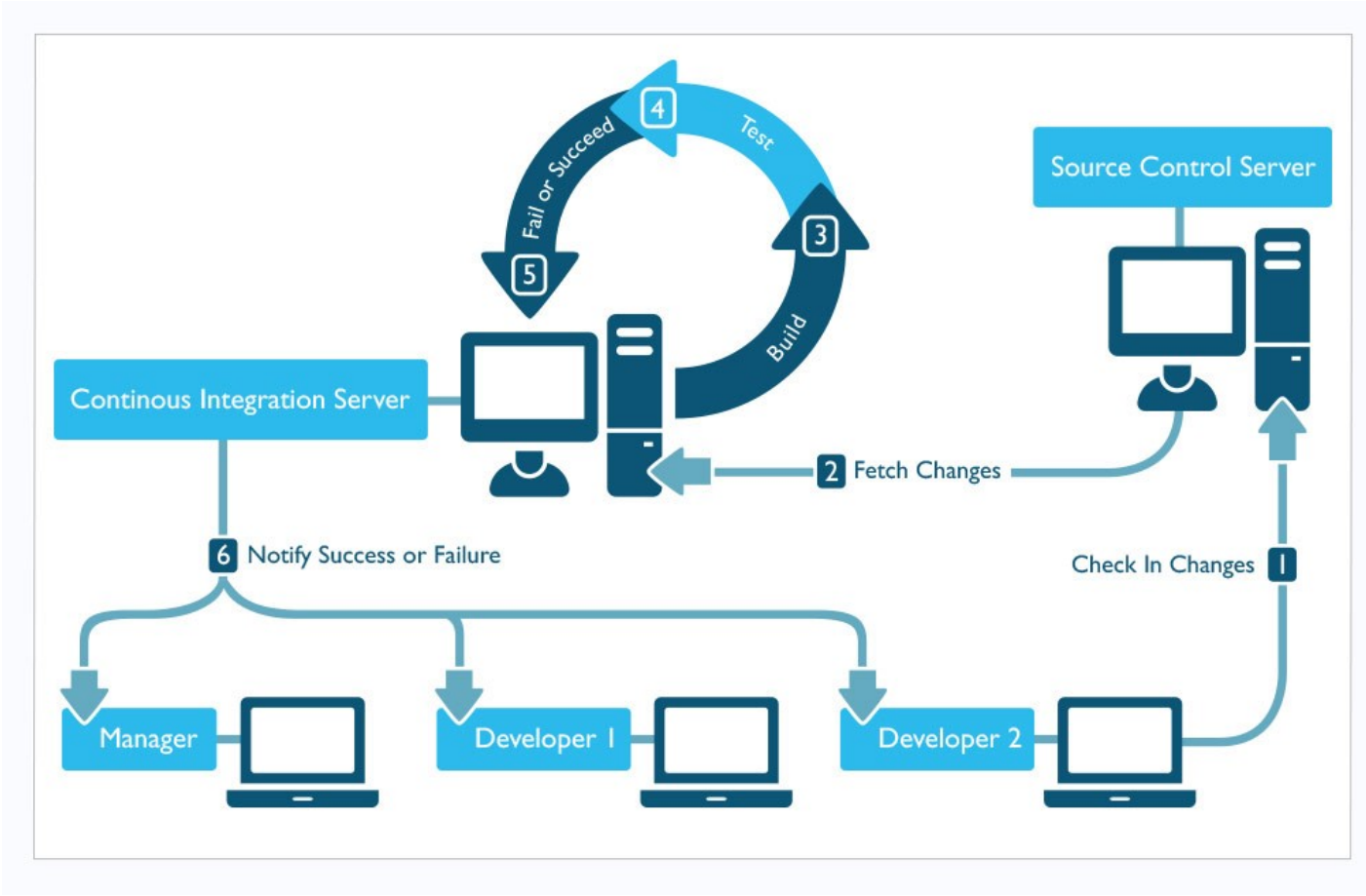


Рисунок 6 – Continuous integration

Проще говоря, девелоперы трудятся на своих ветках, после того как заливается комит на сервер или в определенное время (к примеру ночью, тк сборка проекта может занять много времени) все ветки собираются в одну, и происходит билд проекта и тестирование. Если билд или тестирование произошло с ошибкой девелоперам, которые делали комит или ответственным за качество сотрудникам отправляется оповещение и они сразу идут его исправлять.

Continuous Delivery — расширяет понятие Continuous Development тем, что помимо обычных тестов, проходит проверка по сценарным тестам.

Continuous Deployment — еще большее расширение, тк если комит прошел все тесты успешно, то сборка выкладывается в продакшен. (Достаточно рисковая практика)

Как выглядит этап сдачи задачи в проекте:

1. Код-ревью
2. Исправить имеющиеся замечания, вернуться в п.1
3. Подготовка краткой документации по задачи
4. Отправка задачи на независимую проверку (человек получает готовую сборку и краткую документацию пользователя)
5. Финальный "OK"на готовность влить задачу в ствол проекта
6. Вливание в ствол
7. Задача закрыта

Определение 12. **Мажорная версия** — версия, которая будет достаточно долго поддерживаться. (Будут исправляться ошибки, независимо от выхода новых версий)

6 Антипаттерны

Где встречаются антипаттерны:

- в ООП
- в кодировании
- в методологиях
- в управлении конфигурацией

Антипаттерны в ООП

Наследования функциональности от класса утилита, вместо ее делегирования.

Anemic Domain Problem — боязнь размещать логику в объектах предметной области (создавать движок, чтобы он крутил колеса машине, а не сделать так, чтобы колеса крутились сами).

Вызов предка — для реализации функциональности методу потомка приходится вызывать те же методы родителя.

Ошибка пустого подкласса — когда класс обладает различным поведением по сравнению с классом, который от него наследуется без изменений.

God object — концентрация слишком большого количества функциональности в одном классе / модуле / системе

Объектная клоака — переиспользование объектов находящихся в непригодном для переиспользования состояние.

Полтергейст — объект, чье единственное назначение передавать данные другим объектам.

Проблема йо-йо — чрезмерная размытость сильно связанного кода по иерархии классов.

Одиночка — неуместное использование паттерна Singleton.

Приватизация — чрезмерное скрытие функциональности, что затрудняет расширение при наследование.

Френд-зона — неуместное использование дружественных классов и функций

Каша из интерфейсов — Объединение нескольких интерфейсов, предварительно разделенных, в один.

Висячие концы — интерфейс, большая часть методов которого пустышки.

Заглушка — попытка натянуть малоподходящий интерфейс на класс.

Определение 13. **Singleton** — класс, для которого создается только один объект.

Антипаттерны в кодирование

Ненужная сложность кода.

Действие на расстояние — взаимодействие между широко разнесенными частями.

Накопить и запустить — установка параметров в глобальных переменных.

Слепая вера — недостаточная проверка корректности и полноты исправления ошибки или результата работы.

Лодочный якорь — сохранение неиспользуемой части кода.

Активное ожидание — потребление ресурсов в процессе ожидания запроса, путем выполнения проверок, чтения файлов и тд., вместо асинхронного программирования.

Кэширование ошибки — не сбрасывание флага ошибки после ее обработки.

Воняющий подгузник — сброс флага ошибки без ее обработки или передачи на уровень выше.

Проверка типа вместо интерфейса — проверка на специфический тип, вместо требуемого определенного интерфейса.

Инерация кода — избыточное ограничение из-за подразумевания постоянной ее работы в других частях системы (к примеру, убрать из всей системы поддержку 120гц, из-за того, что на телефонах не поддерживается 120гц)

Кодирование путем исключения — добавление нового кода для каждого нового особого случая.

Таинственный код — использование аббревиатур / сокращений вместо логичных имен.

Жесткое кодирование — внедрение предположений в слишком большое количество точек системы.

Мягкое кодирование — настраивается вообще все, что усложняет конфигурирование.

Поток лавы — сохранение нежелательного кода из-за боязни последствий его удаления / исправления.

Волшебные числа — использование числовых констант без объяснения их смысла.

Процедурный код — когда стоило отказаться от ООП.

Спагетти код — код с чрезмерно запутанным порядком выполнения.

Лазанья код — использование неоправданного большого уровней абстракций.

Равиоли код — объекты настолько склеены между собой, что невозможно провести рефакторинг.

Мыльный пузырь — объект, инициализированный мусор (неинициализированный) слишком долго ведет себя как корректный.

Мьютексный ад — внедрение слишком большого количества примитивов синхронизации в код.

Шаблонный рак — использование шаблонов везде, где можно, а не нужно.

Антипаттерны в кодирование

Использование паттернов — значит имеется недостаточный уровень абстракции.

Копирование / вставка — нужно делать более общий код.

Дефакторинг — процесс уничтожения функциональности и замена ее документацией.

Золотой молоток — использование любимого решения везде, где только получилось.

Фактор невероятности — гипотеза о том, что известная ошибка не случится.

Преждевременная оптимизация — оптимизация при недостаточной информации.

Метод подбора — софт разрабатываемый путем небольших изменений.

Изобретение велосипеда — создание с нуля того, для чего есть готовое решение.

Изобретение квадратного колеса — создание плохого решения, когда уже есть хорошее готовое.

Самоуничтожение — мелкая ошибка приводит к фатальной.

Два тоннеля — вынесение нового функционала в отдельное приложение.

Коммит убийца — внесение изменений без проверки влияния на другие части программы.

Ад зависимостей — разрастание зависимостей до такого уровня, что раздельная установка / удаление становится затруднительным / невозможным.

Дым и зеркала — демонстрация того, как будут работать еще ненаписанные функции.

Раздувание ПО — разрешение последующим версиям использовать все больше и больше ресурсов.

Функция для галочки — превращение программы в "сборную солянку"плохо работающих и не связанных между собой функций.

7 Методы защиты ПО

Определение 14. **Критическая секция** — часть кода, которую очень важно защитить. К примеру, модуль авторизации пользователей.

Атака: дизасемблирование — получение из исполняемого бинарного файла исходного кода программы.

Защита:

- обфускация исходного кода
- шифрование программного кода
- динамическое шифрование (шифрование разных частей кода)
- многопроходная расшифровка (те шифрование несколько раз)
- архивирование
- динамическое изменение кода программы
- использование нестандартной структуры программы

Атака: работа под контролем отладчика (точка останова, которые появляются путем внесения кода 0xCC, которое вызывает прерывание int 3).

Защита:

- подсчет контрольных сумм программного кода
- использование контрольного сумм для расшифровки
- многопроходная расшифровка
- использование корректирующего кода для поиска контрольного байта (проверка того, что все байты кода находятся на своих местах)
- контроль абсолютной времени выполнения
- контроль относительного времени выполнения
- самостоятельный перехват прерывания
- полностью забивать стек критическими данными

Атака: работа под трассировкой.

Защита: те же самые методы, как и в работе под контролем отладчика.

Защита от нелегального использования

Локальная программная защита:

- код на коробке с диском (легко обходится)

Сетевая программная защита:

- локальная (в пределах одной сети, если на двух устройствах один и тот же код, то он перестает работать)
- глобальная, то же самое, только с удаленным сервером контроля

Минусы в том, что никто не отменял создания нелегальных серверов.

Защита с помощью аппаратных средств:

- USB флешки
- компакт диски
- специальные аппаратные средства (HASH ключи)
- привязка лицензии к разнообразным параметрам пк.

8 Полезные утилиты

Google Test Framework — https://google.github.io/googletest/primer.html

gRPC — https://grpc.io/

ПО для Continuous Integration — https://www.jenkins.io/

ПО для Continuous Integration — https://www.travis-ci.com/