**Государственное автономное профессиональное учреждение**

**«Краснокаменский горно-промышленный техникум»**

09.02.07 «Информационные системы и программирование

Наименование дисциплины: МДК 02.01

Разработка программного обеспечения  
ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 2, 3

Выполнил студент гр 303:Медянцева Е.Э.Дата: 17.12.2021Проверил преподаватель:Воробьёва Наталья АнатольевнаДата проверки\_\_\_\_\_\_\_\_\_\_Оценка\_\_\_\_\_\_\_\_\_

**Г. Краснокаменск, 2021 г.**

**Практическая работа №2 «Построение архитектуры программного обеспечения»**

*Цель:*

• Изучение типовых шаблонов технического задания;

• Изучение методов разработки технического задания;

Вопросы для самоподготовки к освоению данной темы:

1. Что такое архитектура программного средства?

2. Что включает в себя архитектура программного средства?

Архитектура программного обеспечения (англ. software architecture) — совокупность важнейших решений об организации программной системы. Архитектура включает:

• выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов;

• соединение выбранных элементов структуры и поведения во всё более крупные системы;

• архитектурный стиль, который направляет всю организацию — все элементы, их интерфейсы, их сотрудничество и их соединение.

Шаблон — это решение задачи в определенном контексте.

Часто разработчики не до конца понимают разницу между архитектурными шаблонами, а иногда вообще мало что о них знают.

Что ж, давайте разбираться!

Многоуровневая архитектура

Каналы и фильтры

Клиент — сервер

Модель — представление — контроллер

Управляемая событиями архитектура

Архитектура на основе микросервисов

Многоуровневая архитектура

Самый распространенный архитектурный шаблон — многоуровневая архитектура (или «n-уровневая»). Она хороша известна большинству архитекторов, проектировщиков и разработчиков. Ограничений по количеству и типу уровней никаких нет, однако в большинстве случаев такая архитектура состоит из четырех уровней: представление данных, бизнес-логика, хранение данных и база данных.

Популярный пример n-уровневой архитектуры

Популярный пример n-уровневой архитектуры

Контекст

В любых сложных системах необходимо обеспечивать независимую разработку и развитие отдельных частей. Потому разработчикам необходимо понятное и хорошо задокументированное разделение задач, чтобы модули системы можно было разрабатывать и поддерживать независимым образом.

**Задача**

Программное обеспечение необходимо разделить так, чтобы модули можно было разрабатывать и развивать отдельно — с минимальным взаимодействием между частями, обеспечивая переносимость, модифицируемость и повторное использование.

**Решение**

Чтобы добиться такого разделения, при использовании многоуровневого шаблона программное обеспечение разделяется на сущности, называемые уровнями. Каждый уровень — группа модулей, предоставляющих взаимосвязанный набор сервисов. Их применение должно быть однонаправленным. Уровни полностью разделяют ПО, причем каждая часть доступна через публичный интерфейс.

Первая идея заключается в том, что у каждого из уровней есть определенные роль и ответственность. Например, уровень представления данных отвечает за обработку всего пользовательского интерфейса. Такое разделение задач в многоуровневой архитектуре упрощает формирование эффективных ролей и ответственности.

Вторая идея состоит в том, что шаблон многоуровневой архитектуры представляет собой техническое разделение — в отличие от разделения по доменам: то есть, это группы компонентов, а не домены.

И, наконец, третья идея: каждый уровень помечается как закрытый или открытый. Запрос, перемещаясь с уровня на уровень и минуя закрытый уровень, должен обязательно пройти через него — пропустить закрытый уровень нельзя.

Закрытые уровни и движение запроса

**Недостатки**

Наличие уровней снижает производительность. Этот шаблон не подходит для высокопроизводительных приложений: проходить несколько уровней архитектуры для выполнения бизнес-запроса — это неэффективно.

Кроме того, добавление уровней увеличивает полную стоимость и усложняет систему.

**Применение**

Использовать этот подход следует для небольших, простых приложений и веб-сайтов. Также этот шаблон хорошо подходит, если бюджет и время ограничены.

Многоярусный шаблон

**Контекст**

При распределенном развертывании часто необходимо разделить инфраструктуру системы на отдельные подмножества.

**Задача**

Как разделить систему на ряд независимых в вычислительном отношении исполнительных структур — групп программного и аппаратного обеспечения, объединенных каким-нибудь средством связи?

**Решение**

Исполнительные структуры многих систем организованы как набор логических групп компонентов. Каждая группа называется ярусом.

**Недостатки**

Значительные полная стоимость и сложность.

**Применение**

Используется в распределенных системах.

Каналы и фильтры

Один из часто используемых шаблонов в архитектуре ПО — шаблон каналов и фильтров.

Подход «каналы и фильтры»

**Контекст**

Часто в системах необходимо преобразовывать потоки дискретных элементов данных от ввода к выводу. На практике многие типы преобразований повторяются неоднократно, поэтому желательно сделать из них независимые, повторно используемые компоненты.

**Задача**

Такие системы необходимо разделять на повторно используемые слабо связанные компоненты с простыми обобщенными механизмами взаимодействия — таким образом их можно будет удобно сочетать друг с другом. Слабо связанные компоненты с обобщенным взаимодействием легко использовать повторно. А поскольку они независимы, их выполнение можно запускать параллельно.

**Решение**

В этой архитектуре фильтры связаны между собой коммуникационными каналами. Первая идея заключается в том, что каждый из каналов по соображениям производительности имеет тип «точка — точка» и является однонаправленным: принимает входные данные от одного источника и направляет выходные данные в приемник.

В таком подходе выделяют фильтры четырех видов:

генератор (источник) — отправная точка процесса;

преобразователь (сопоставление) — выполняет преобразование некоторых или всех данных;

испытатель (редуцирование) — проверяет один или несколько критериев;

потребитель (приемник) — конечная точка.

**Недостатки**

Не лучший выбор для интерактивных систем, поскольку такая архитектура ориентирована на преобразование данных.

Чрезмерное использование синтаксического анализа и синтеза снижает производительность и усложняет написание самих фильтров.

**Применение**

Архитектура каналов и фильтров применяется в самых разных приложениях, особенно при решении задач, обеспечивающих простую одностороннюю обработку — например, инструменты EDI (электронный обмен данными), ETL (извлечение, преобразование и загрузка).

Пример — компиляторы: последовательно расположенные фильтры выполняют лексический, синтаксический, семантический анализ и создание кода.

Клиент — сервер

**Контекст**

Есть общие ресурсы и сервисы, к которым нужно обеспечить доступ большого количества распределенных клиентов, и при этом необходимо контролировать доступ или качество обслуживания.

**Задача**

Управляя набором общих ресурсов и сервисов, можно обеспечить модифицируемость и повторное использование, для чего общие сервисы выносятся отдельно, чтобы их можно было изменять в одном месте или в небольшом количестве мест. Требуется улучшить масштабируемость и доступность к использованию за счет централизованного управления этими ресурсами и сервисами при одновременном распределении самих ресурсов между несколькими физическими серверами.

**Решение**

В подходе «клиент — сервер» компоненты и соединительные элементы обладают определенным поведением.

Компоненты, называемые «клиентами», отправляют запросы компоненту, называемому «сервер», и ждут ответа.

Компонент «сервер» получает запрос от клиента и отправляет ему ответ.

**Недостатки**

Сервер может быть узким местом в отношении производительности, а также единой точкой отказа.

Изменять принятое решение о размещении функциональных возможностей (на клиенте или на сервере) после создания системы — это обычно сложно и дорого.

**Применение**

Подход «клиент — сервер» можно применять в моделировании части системы, имеющей много компонентов, отправляющих запросы (это «клиенты») другому компоненту (это «сервер»), который обеспечивает работу сервисов, — например, онлайн-приложения (электронная почта, обмен документами и банковское дело).

Модель — представление — контроллер

**Контекст**

Обычно в интерактивном приложении чаще других изменяется пользовательский интерфейс. Пользователям нужно видеть данные в различном представлении — например, как линейчатую или круговую диаграмму, — и оба представления должны отражать текущее состояние данных.

**Задача**

Как можно отделить функциональность пользовательского интерфейса от функциональности приложения и при этом обеспечить быстрый отклик на действия пользователя и изменения в базовых данных приложения?

Как создавать, поддерживать и координировать несколько представлений пользовательского интерфейса при изменении базовых данных приложения?

**Решение**

Шаблон «модель — представление — контроллер» (MVC) разделяет функциональность приложения на компоненты трех видов:

Модель — содержит данные приложения.

Представление — отображает некоторую часть базовых данных и взаимодействует с пользователем.

Контроллер — действует в качестве посредника между моделью и представлением и управляет уведомлениями об изменении состояния.

**Недостатки**

Для простых пользовательских интерфейсов такая сложность может быть чрезмерной.

Абстракции «модель», «представление» и «контроллер» могут не очень хорошо подходить в случае некоторых наборов инструментов для разработки пользовательского интерфейса.

**Применение**

Архитектурный шаблон MVC обычно используется в мобильных и веб-приложениях при разработке пользовательских интерфейсов.

**Управляемая событиями архитектура**

**Контекст**

Необходимо предоставить вычислительные и информационные ресурсы для обработки независимых асинхронных входящих событий, формируемых приложением, с возможностью масштабирования по мере потребности.

**Задача**

Создать распределенные системы, которые могут обслуживать поступающие асинхронные сообщения, связанные с событием, и масштабироваться в широком диапазоне величины и сложности.

**Решение**

Развернуть независимые процессы или обработчики для работы с событиями. Поступающие события ставятся в очередь. Планировщик извлекает события из очереди и передает их в соответствующий обработчик событий согласно политике планирования.

**Недостатки**

Возможные проблемные области — производительность и восстановление после ошибок.

**Применение**

Использующее такой подход приложение для электронной коммерции будет работать следующим образом:

Сервис «Заказы» создает Заказ в состоянии ожидания и публикует событие «Создан Заказ» OrderCreated.

Сервис «Покупатели» получает событие и пытается зарезервировать кредит для Заказа. Затем он публикует событие «Кредит Зарезервирован» CreditReserved или «Превышен Лимит Кредита» CreditLimitExceeded.

Сервис «Заказы» получает это событие от сервиса «Покупатели» и меняет состояние заказа на «Утвержден» или «Отменен».

**Архитектура на основе микросервисов**

**Контекст**

Развертывание серверных корпоративных приложений, поддерживающих различные браузеры и нативные мобильные клиенты. Приложение обрабатывает клиентские запросы, реализуя бизнес-логику, обращаясь к базе данных, обмениваясь сообщениями с другими системами и возвращая ответы. Приложение может предоставлять API для реализации третьими сторонами.

**Задача**

При необходимости оптимального использования распределенных ресурсов (например, в облачных средах) монолитные приложения могут оказаться слишком большими и сложными для эффективной поддержки и развертывания.

**Решение**

Создание приложений в виде наборов сервисов: каждый сервис развертывается и масштабируется независимо и имеет собственную «границу», обслуживаемую посредством API. Различные сервисы могут писаться на разных языках программирования, управлять собственными базами данных и разрабатываться разными командами.

**Недостатки**

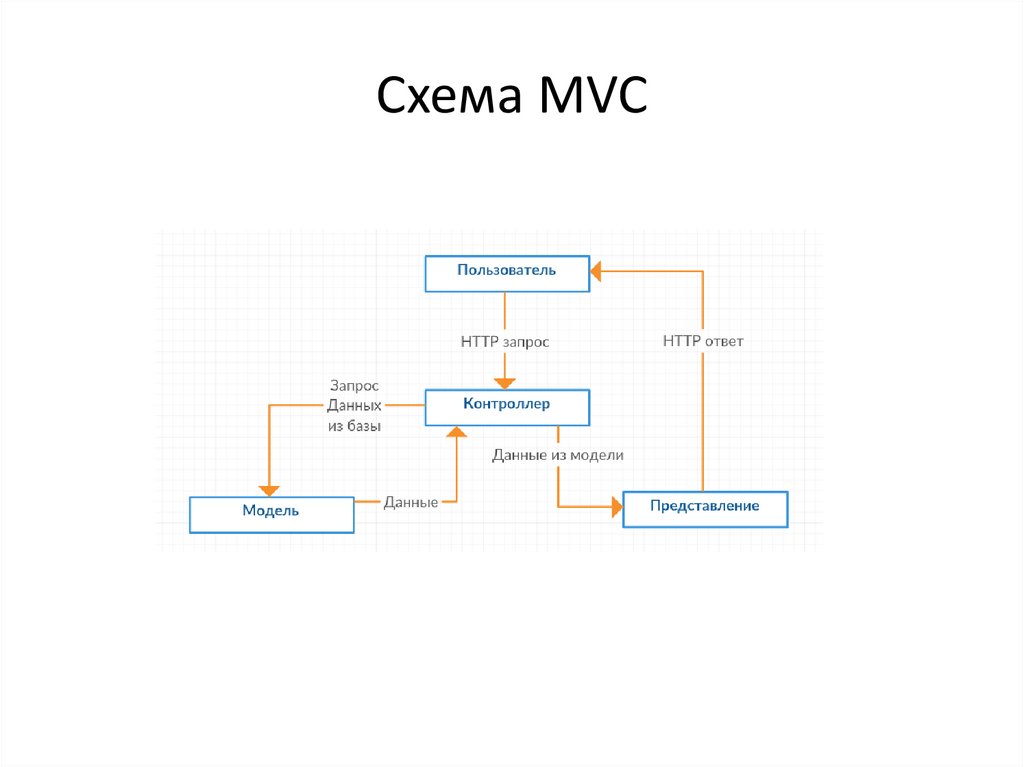
Системы должны быть спроектированы так, чтобы выдерживать сбои в работе сервисов, а это требует более тщательного мониторинга систем. Дополнительные расходы ресурсов на организацию работы сервисов и взаимодействия событий.

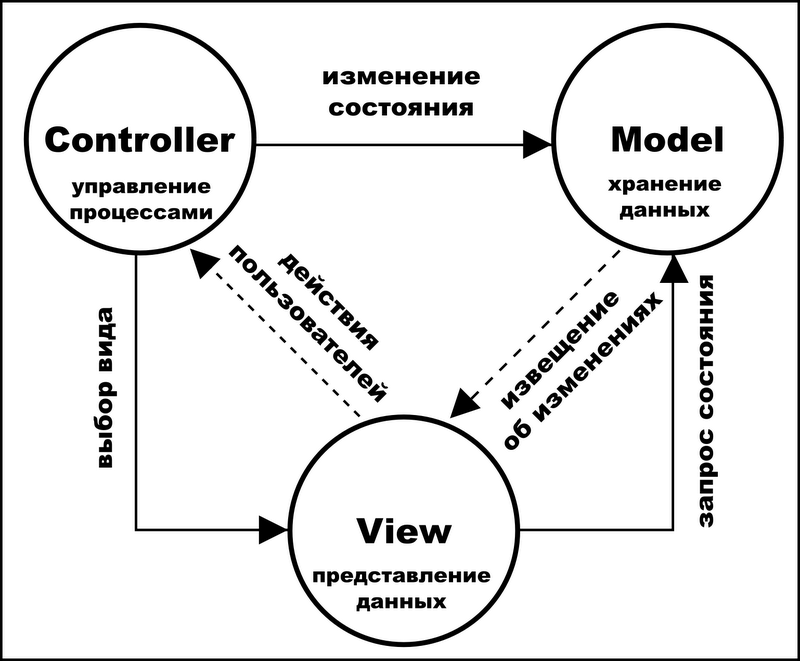
Кроме того, потребуется больше памяти.

**Применение**

Архитектура микросервисов применима во многих случаях, особенно когда используется большой конвейер данных. Например, микросервисная архитектура — отличный выбор для системы отчетности о продажах в розничных магазинах компании. Каждый шаг в процессе подготовки данных будет обрабатываться микросервисом: сбор, очистка, нормализация, обогащение, агрегирование данных, отчетность и т. д.

Модель — представление — контроллер схема:





**Практическая работа №3 «Изучение работы в системе контроля версий»**

*Цель:*

• Изучение типовых шаблонов технического задания;

• Изучение методов разработки технического задания;

Вопросы для самоподготовки к освоению данной темы:

1. Что такое система контроля версий?
2. Что включает в себя архитектура программного средства?

**Система управления версиями (от англ. Version Control System, VCS или Revision Control System)** — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.



*Рисунок 1 – схема локальной СКВ*

**

*Рисунок 2 – схема централизованного контроля версий*

**

*Рисунок 3 – схема распределённой системы контроля версий*

**Git (произн. «гит»)** — распределённая система управления версиями. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. На сегодняшний день его поддерживает Джунио Хамано.

Примерами проектов, использующих Git, являются: *Ядро Linux, Swift, Android, Drupal, Cairo, GNU Core Utilities, Mesa, Wine, Chromium, Compiz Fusion, FlightGear, jQuery, PHP, NASM, MediaWiki, DokuWiki, Qt и некоторые дистрибутивы Linux.*