**1. Жизненный цикл ПО. Понятие жизненного цикла ПО, этапы, их описание в терминологии различных стандартов (примеры).**

Жизненный цикл ПО — это последовательность этапов и активностей, через которые проходит ПО, начиная от его создания и разработки, заканчивая его выводом из эксплуатации. Различные стандарты и модели определяют свои термины и этапы в жизненном цикле ПО.

Жизненный цикл ПО обычно включает в себя следующие основные этапы: концепция, планирование, разработка, тестирование, развертывание, эксплуатация и вывод из эксплуатации.

Примеры этапов в терминологии двух популярных стандартов.

1. Модель "Waterfall" (Водопад):

- Требования: Определение функциональных и нефункциональных требований к ПО.

- Проектирование: Разработка архитектуры, дизайна интерфейса и базы данных.

- Разработка: Создание исходного кода и программирование ПО.

- Тестирование: Проверка ПО на соответствие требованиям и выявление ошибок.

- Внедрение: Установка и настройка ПО в рабочей среде.

- Эксплуатация: Работа ПО в реальной среде, выполнение задач и обслуживание.

- Обслуживание: Внесение исправлений, обновлений и поддержка во время эксплуатации.

- Вывод из эксплуатации: Завершение использования ПО, удаление или замена его другим.

2. Методология "Agile" (Гибкий метод):

- Запланировать: Определение целей и требований для текущего этапа разработки.

- Разработать: Создание инкрементальных версий ПО, фокусируясь на наиболее значимых функциях.

- Протестировать: Проверка и верификация функциональности и качества каждой инкрементальной версии.

- Оценить: Анализ результатов тестирования и обратная связь для корректировки и улучшения.

- Повторить: Повторение цикла планирования, разработки, тестирования и оценки для каждой версии.

- Завершить: Выпуск окончательной версии ПО и завершение проекта.

**2. Процессы создания ПО. Управление процессами, 5 групп процессов, их характерные особенности.**

Процессы создания ПО являются набором шагов и методологий, используемых для разработки, тестирования и внедрения ПО. Управление процессами включает планирование, координацию и контроль всех этапов разработки ПО для обеспечения достижения поставленных целей и требований проекта.

Управление процессами создания ПО может быть осуществлено с использованием различных методологий, таких как классическое управление проектами (например, модель PMBOK), Agile-методологии (например, Scrum, Kanban) или DevOps-подходы для автоматизации и ускорения процесса разработки и внедрения ПО.

Процесс создания ПО включает в себя различные этапы и подходы к разработке ПО. Вот общая структура процесса создания ПО и основные характеристики пяти групп процессов:

1. Линейные процессы:

- Водопадная модель: Линейный процесс, где каждый этап (анализ, проектирование, разработка, тестирование и внедрение) выполняется последовательно и не возвращается к предыдущему этапу. Отличается высокой степенью планирования и документирования.

- Последовательная модель: известна как простая линейная модель, где разработка ПО выполняется в строгой последовательности этапов, но с возможностью возврата на предыдущие этапы.

2. Итеративные процессы:

- Разработка по прототипу: Процесс, основанный на создании прототипов, которые позволяют уточнить требования и функциональность ПО в тесном взаимодействии с заказчиком.

- Разработка по инкрементам: подразумевает поэтапную разработку, при которой новые возможности и функциональность добавляются в ПО на протяжении нескольких циклов разработки.

3. Эволюционные процессы:

- Спиральная модель: Процесс, который объединяет элементы водопадной модели и итеративной разработки, где разработка осуществляется через циклы, но каждый цикл проходит через этапы планирования, риска, анализа и инженерии.

- RUP: Процесс, основанный на использовании итеративного и инкрементного подхода к разработке, включающий фазы, такие как начало, разработка, внедрение и поддержка.

4. Agile-процессы:

- Scrum: Итеративный и инкрементальный процесс, основанный на ролевом распределении (Scrum-мастер, владелец продукта, команда разработчиков) и коротких циклах разработки, называемых спринтами.

- Extreme Programming (XP): процесс, который акцентирует внимание на обратной связи клиента, непрерывном тестировании, парном программировании, маленьких релизах и непрерывном улучшении кодовой базы.

5. DevOps-процессы:

- CI/CD: Процесс, включающий непрерывное интегрирование изменений в код, непрерывное тестирование и автоматическое развертывание ПО для обеспечения быстрой и надежной доставки продукта.

- SRE: Методология, которая объединяет аспекты разработки и операций для создания стабильных и высокопроизводительных систем, где управление и поддержка приложения являются неотъемлемой частью разработки ПО.

Каждая группа процессов имеет свои характеристики, преимущества и недостатки. Выбор конкретного процесса зависит от требований проекта, командной структуры и других факторов.

**3. Проект. Управление проектами. Понятие проектного офиса, типы проектных офисов. Программы и портфели.**

Проект – это временное предприятие, направленное на создание уникального продукта, услуги или результата. Он характеризуется определенными целями, ограничениями по времени, бюджету и ресурсам. Проекты имеют четкие начало и конец, отличаются от повседневных операций и требуют уникальных навыков и управления.

Управление проектами – это процесс планирования, организации, контроля и координации ресурсов с целью достижения конкретных результатов в рамках ограничений проекта. Оно включает в себя определение целей проекта, планирование деятельностей, оценку рисков, управление коммуникациями, координацию команды и мониторинг прогресса проекта.

Проектный офис – это централизованная структура или отдел в организации, который обеспечивает управление и поддержку проектами. PMO обычно предоставляет методологии, стандарты, инструменты и ресурсы для эффективного выполнения проектов. Он может выполнять функции планирования, контроля, управления рисками, координации ресурсов и обучения.

Типы проектных офисов:

1. Супервизионный (Supportive) PMO: Основная функция - предоставление советов, тренингов, общей поддержки и методологической экспертизы проектам и проектным менеджерам в организации.

2. Контролирующий (Controlling) PMO: Он контролирует и стандартизирует процессы управления проектами, предоставляет руководство, отчетность и мониторинг проектов, а также выполняет аудиты и обеспечивает соответствие стандартам.

3. Директивный (Directive) PMO: Он принимает активное участие в управлении проектами, предоставляет ресурсы, назначает проектных менеджеров, контролирует исполнение и принимает ключевые решения в проектах.

4. Комбинированный (Hybrid) PMO: это комбинация различных функций и ролей PMO в одном офисе, в зависимости от потребностей и характеристик организации.

Программа – это набор взаимосвязанных проектов, целью которых является достижение общей стратегической цели организации. Программа обычно имеет управленческий фокус и требует координации между проектами.

Портфель – это набор программ, проектов и других работ, которые совместно управляются для достижения бизнес-целей организации. Портфель включает в себя оценку, приоритизацию и выбор проектов и программ, а также управление ресурсами и рисками на уровне портфеля.

Управление программами и портфелями предоставляет организации средства для эффективного планирования, выделения ресурсов, контроля и достижения стратегических целей через связанные проекты и программы.

**4. Требования к ПО. Виды требований. Техники сбора и описания.**

Требования к ПО — это описание функциональных и нефункциональных характеристик, которые ПО должно удовлетворять, чтобы соответствовать потребностям пользователей и бизнес-целям. Требования играют ключевую роль в процессе разработки ПО и служат основой для создания, тестирования и внедрения системы.

Виды требований:

1. Функциональные требования: описывают функциональность, которую должно предоставлять ПО. Это конкретные действия, которые система должна выполнять или результаты, которые должны быть получены.

2. Нефункциональные требования: описывают качественные атрибуты ПО, такие как производительность, безопасность, надежность, доступность, масштабируемость и удобство использования.

3. Бизнес-требования: описывают бизнес-цели, потребности и ожидания, которые должны быть удовлетворены разрабатываемым ПО. Они обычно формулируются на уровне организации или заказчика.

Техники сбора требований:

1. Интервью: Проведение бесед и обсуждений с представителями пользователей, заказчиками и другими заинтересованными сторонами для выявления требований.

2. Фокус-группы: Организация групповых дискуссий с представителями различных групп пользователей для получения обратной связи и выявления требований.

3. Наблюдение за работой: Изучение и наблюдение за текущими бизнес-процессами и работой пользователей для выявления потребностей и проблем, которые должны быть решены с помощью ПО.

4. Прототипирование: Создание прототипов или образцов ПО, которые позволяют пользователям и заказчикам визуализировать и оценить функциональность и интерфейс системы.

5. Воркшопы: Организация совместных сессий и рабочих групп с различными заинтересованными сторонами для обмена идеями, выявления требований и принятия решений.

Техники описания требований:

1. Use Case-диаграммы: Графическое представление взаимодействия актеров и системы через сценарии использования, которые описывают основные функциональные требования.

2. User Stories: Краткие описания требований, сфокусированные на потребностях и целях конкретных пользователей. Они формулируются в виде "Как пользователь, я хочу..., чтобы..."

3. Функциональные и нефункциональные спецификации: Документы, которые подробно описывают требования к функциональности и нефункциональным атрибутам ПО.

4. Протоколы испытаний: Описание процедур и критериев тестирования, которые используются для проверки соответствия ПО требованиям.

5. Диаграммы классов, последовательности, активности и другие.

Важно выбирать и комбинировать различные техники сбора и описания требований в зависимости от особенностей проекта, заинтересованных сторон и доступных ресурсов.

**5. Проектное управление. Ограничения и жизненный цикл проекта**

Проектное управление — это процесс планирования, организации, контроля и координации ресурсов с целью достижения конкретных результатов в рамках ограничений проекта. Оно включает в себя управление временем, бюджетом, качеством, рисками, коммуникациями и другими аспектами проекта.

Ограничения проекта:

1. Временные ограничения: Проект имеет определенный срок выполнения, и его успех зависит от соблюдения этого срока. Задача проектного управления - управлять расписанием и ресурсами таким образом, чтобы проект был завершен в срок.

2. Бюджетные ограничения: Проект имеет определенный бюджет, и его выполнение не должно превышать этот бюджет. Планирование и контроль затрат, чтобы обеспечить эффективное использование ресурсов.

3. Ограничения качества: Проект должен удовлетворять определенным требованиям к качеству, установленным заказчиком или организацией. Планирование и контроль качества, чтобы обеспечить соответствие требованиям.

4. Ограничения области: Проект имеет определенную область применения, и его результаты должны соответствовать этой области. Определение и управление областью проекта, чтобы обеспечить достижение желаемых результатов.

Жизненный цикл проекта:

Жизненный цикл проекта — это последовательность фаз, через которые проходит проект от его начала до завершения. Жизненный цикл может варьироваться в зависимости от методологии и характеристик проекта.

1. Инициация: В этой фазе определяются цели проекта, его обоснование, заинтересованные стороны и ресурсы, а также проводится предварительная оценка выполнимости проекта.

2. Планирование: разрабатывается подробный план проекта, включающий определение задач, распределение ресурсов, составление графика, управление рисками и коммуникационные планы.

3. Выполнение: осуществляется фактическая реализация проекта. Задачи выполняются, ресурсы используются, коммуникация осуществляется в соответствии с планом проекта.

4. Мониторинг и контроль: в течение всего жизненного цикла проекта проводится контроль за выполнением плана, ресурсами, качеством и рисками.

5. Завершение: По достижении целей проекта фаза завершения включает оценку результатов, документацию и закрытие проекта. Ресурсы освобождаются, а полученные результаты передаются заказчику или организации.

Жизненный цикл проекта обеспечивает структурированный подход к управлению проектом и помогает в достижении поставленных целей, контроле ограничений и достижении желаемых результатов.

**6. Проектное управление. Методологии. PMBOOK и PRINCE2**

Проектное управление — это процесс, который охватывает планирование, организацию, управление и контроль проектом для достижения его целей в ограничениях, таких как время, бюджет, качество и ресурсы. Существует несколько методологий проектного управления: PMBOK предоставляет общие принципы, процессы и практики, которые должны быть применены для успешного выполнения проекта; PRINCE2 определяет роли и ответственности, принципы и процессы, которые должны быть применены для эффективного управления проектами; Agile акцентирует внимание на адаптивности, коллаборации и поэтапном развитии продукта; Lean стремится создать более эффективные и потокоориентированные процессы.

PMBOOK — это набор стандартов и лучших практик, разработанных Проектным управлением института. Он описывает основы проектного управления и включает стандарты и рекомендации по планированию, выполнению, контролю и завершению проектов. PMBOOK определяет процессы, входы, выходы и инструменты/техники, которые могут быть использованы при управлении проектом. Он охватывает девять областей знаний: интеграционное управление, управление содержанием проекта, управление временем, управление стоимостью, управление качеством, управление человеческими ресурсами, управление коммуникациями, управление рисками и управление закупками.

PRINCE2 — это методология управления проектами, разработанная в Великобритании. Она представляет собой структурированный подход к управлению проектом и является основным фреймворком для управления проектами во многих организациях. PRINCE2 определяет семь принципов, семь тематик и семь процессов, которые охватывают все аспекты проектного управления, включая организацию проекта, планирование, контроль, риски, качество и коммуникации. Она акцентирует внимание на управлении стадиями, включая их запуск, их выполнение и их завершение. PRINCE2 предоставляет методологические инструменты и шаблоны для эффективного управления проектом.

Обе методологии представляют установленные стандарты и лучшие практики для проектного управления. Они предоставляют фреймворки, инструменты и подходы для эффективного планирования, организации и управления проектами, независимо от их размера и сложности.

**7. Гибкие принципы управление. Agile – идеи и принципы.**

Гибкие принципы управления, также известные как Agile (гибкий), представляют собой набор идей и принципов, направленных на эффективное управление проектами и разработкой программного обеспечения. Agile подразумевает итеративный и инкрементальный подход к работе, при котором проект разделяется на небольшие части, называемые итерациями или спринтами, и каждая часть выполняется небольшими командами.

Вот основные идеи и принципы Agile:

1. Основаются на людях и взаимодействии: Agile ставит акцент на командную работу и сотрудничество между разработчиками, заказчиками и заинтересованными сторонами. Взаимодействие и коммуникация являются ключевыми для успешного выполнения проекта.

2. Работающее ПО как основной показатель прогресса: Agile сосредотачивается на доставке рабочего программного обеспечения на регулярной основе. Работающее ПО является первоочередным показателем прогресса и предоставляет ценность заказчику.

3. Гибкость и реагирование на изменения: Agile признает, что требования и условия могут изменяться в течение проекта. Гибкость и способность быстро реагировать на изменения являются важными аспектами Agile подхода.

4. Коллективная ответственность: Agile поощряет коллективную ответственность команды. Все члены команды делятся ответственностью за достижение целей проекта и успех проекта в целом.

5. Итеративность и инкрементальность: Agile разделяет проект на короткие итерации, в рамках которых команда выполняет работу и доставляет результаты. Каждая итерация добавляет новый функционал или улучшение к существующей системе.

6. Постоянное совершенствование и обратная связь: Agile поддерживает постоянное совершенствование путем регулярной обратной связи и ретроспективных собраний. Команда анализирует свою работу, выявляет проблемы и находит способы улучшить процесс и качество продукта.

7. Прозрачность и видимость: Agile ставит целью прозрачность и видимость всей работы. Все заинтересованные стороны имеют доступ к информации о проекте и его текущем статусе.

Идеи и принципы Agile помогают командам быть более гибкими, адаптивными и способствуют достижению более успешных результатов в разработке программного обеспечения и управлении проектами.

**8. Гибкие технологии. Scrum**

Гибкие технологии, также известные как Agile-технологии, представляют собой набор подходов, методик и инструментов, используемых в разработке программного обеспечения и управлении проектами. Они основаны на гибком и итеративном подходе к работе, при котором акцент делается на адаптивности, сотрудничестве и быстрой доставке ценности.

Одной из самых популярных Agile-методологий является Scrum. Scrum представляет собой гибкую методологию управления проектами, которая обеспечивает фокус на быстрой доставке ценности, коллаборации команды и способствует управлению изменениями. Вот некоторые основные аспекты Scrum:

1. Роли в Scrum:

- Product Owner: Отвечает за управление бэклогом продукта, определение требований и приоритетов.

- Scrum Master: Обеспечивает правильную реализацию Scrum-процесса, помогает команде и устраняет преграды.

- Development Team: Команда разработчиков, ответственных за создание и доставку работающего продукта.

2. Бэклог продукта:

Бэклог продукта представляет собой список требований, функциональностей и задач, которые нужно выполнить. Он поддерживается Product Owner'ом и регулярно обновляется.

3. Спринт:

Спринт представляет собой фиксированный период времени (обычно от 1 до 4 недель), в течение которого команда разрабатывает и доставляет работающий продукт. В начале спринта определяются цели и выбираются задачи для выполнения.

4. Ежедневное совещание (Daily Scrum):

Ежедневное совещание проводится каждый день в течение спринта. Команда обсуждает свой прогресс, задачи, преграды и планы на ближайшее время.

5. Спринтовый обзор (Sprint Review):

По окончании спринта проводится спринтовый обзор, на котором команда демонстрирует достигнутые результаты и получает обратную связь от заинтересованных сторон.

6. Ретроспектива (Sprint Retrospective):

Ретроспектива проводится после спринтового обзора. Команда анализирует свою работу, выявляет улучшения и планирует изменения в следующем спринте.

Scrum обеспечивает быструю доставку ценности, эффективное управление изменениями, повышение прозрачности и сотрудничество в команде. Он подходит для проектов, где требования могут изменяться и где важно быстро реагировать на изменения рынка или клиентских потребностей.

**9. Гибкие технологии. Kanban**

Гибкие технологии, также известные как Agile-технологии, являются подходами к разработке программного обеспечения и управлению проектами, которые акцентируются на адаптивности, коллаборации и быстрой доставке ценности. Гибкие технологии помогают командам эффективно работать в переменных и динамичных условиях.

Одним из примеров гибкой методологии является Kanban — это система визуализации и управления потоком работы. Она позволяет команде контролировать и оптимизировать процесс разработки или выполнения задач.

Вот некоторые основные аспекты Kanban:

1. Визуальная доска:

В канбан-методологии используется визуальная доска, которая разделена на колонки, представляющие различные стадии выполнения работы (например, "Запланировано", "В процессе", "Готово"). Каждая задача представлена карточкой, которая перемещается по колонкам в зависимости от своего состояния.

2. Ограничение рабочего прогресса:

В каждой колонке на доске устанавливается ограничение по количеству задач, которые могут находиться в этой колонке одновременно. Это помогает предотвратить перегрузку работы и поддерживает более плавный поток выполнения задач.

3. Пул задач и приоритизация:

Задачи помещаются в пул задач, откуда команда берет новую задачу для выполнения, когда освобождаются ресурсы. Задачи могут быть приоритизированы с учетом их важности и срочности.

4. Постоянное улучшение и обратная связь:

Канбан подразумевает постоянное улучшение процесса работы и потока задач. Команда проводит периодические собрания, чтобы обсудить текущий статус работы, выявить проблемы и искать пути их решения.

5. Гибкость и адаптивность:

Канбан позволяет команде быть гибкой и адаптивной к изменениям. Задачи могут быть легко перепланированы или изменены в зависимости от новых требований или приоритетов.

Kanban акцентирует внимание на визуализации рабочего процесса, управлении потоком работы и оптимизации процесса. Он особенно полезен для команд, которые работают над повторяющимися задачами или проектами с непостоянным потоком требований.

**10. Масштабирование гибкой разработки. Проблемы и решения. SAFe как пример масштабируемого фреймворка**

Масштабирование гибкой разработки относится к применению принципов и практик гибкой методологии на уровне организации, где работают несколько команд над большими и сложными проектами. Масштабирование помогает справиться с вызовами, которые возникают при координации работы и высокой степени взаимодействия между командами.

Распространенные проблемы и их решения при масштабировании гибкой разработки:

Синхронизация и координация команд:

Проблема: когда работают несколько команд, важно синхронизировать и координировать их усилия для достижения общих целей. Возникает риск перекрытия работ. Решение: Внедрение методов координации, таких как ежедневные стендапы и совместное планирование, позволяет командам обмениваться информацией, решать проблемы и обеспечивать согласованность работы.

Управление зависимостями:

Проблема: Большие проекты часто имеют множество зависимостей между различными компонентами или командами. Если зависимости не управляются должным образом, возникают задержки и проблемы в качестве и доставке продукта. Решение: Использование инструментов визуализации зависимостей, установление ясных коммуникационных каналов, совместное планирование и координация работы.

Планирование и предсказуемость:

Проблема: при масштабировании гибкой разработки может быть сложно обеспечить предсказуемость и планируемость проекта, особенно когда требования меняются или возникают неожиданные проблемы. Решение: Использование гибких методик планирования, таких как итерационное планирование и планирование на основе ценности.

Согласование и высокий уровень прозрачности:

Проблема: При масштабировании гибкой разработки возникают сложности в согласовании и обеспечении прозрачности работы команд. Отсутствие общего видения и понимания может привести к конфликтам. Решение: Внедрение совместных сессий планирования, регулярного обмена информацией и создание механизмов для отчетности и прозрачности.

SAFe является примером масштабируемого фреймворка, который предоставляет подходы и рекомендации для масштабирования гибкой разработки на уровне организации. Он предлагает структуру и роли для эффективного сотрудничества между командами и высокой степени координации работы.

1. Портфель: SAFe предлагает структурированный подход к управлению портфелем проектов. Он помогает определить стратегические цели, принимать решения о приоритетах и выделять ресурсы на основе бизнес-значимости.

2. Потоки ценности: SAFe поддерживает организацию работы вокруг потоков ценности, которые выступают в качестве основных единиц доставки ценности для клиентов.

3. Роли и ответственности: SAFe определяет различные роли и ответственности на разных уровнях организации, такие как Agile-коучи, Product Owners, Release Train Engineers и другие. Эти роли помогают обеспечить эффективное управление проектами и координацию работы команд.

4. Инкрементальное планирование и инспекция: SAFe пропагандирует инкрементальное планирование и инспекцию работы команд. Это позволяет быстро адаптироваться к изменениям, снижает риск и обеспечивает непрерывную доставку ценности.

**11. Гибкое управление. Особенности организации работы дистанционно работающих команд**

Гибкое управление (Agile management) — это подход к управлению проектами и командами, основанный на принципах и ценностях гибкой методологии разработки ПО. Он акцентирует внимание на гибкости, адаптивности, сотрудничестве и непрерывном улучшении процессов работы.

Особенности организации работы дистанционно работающих команд являются важной темой в настоящее время, особенно с увеличением числа удаленных или распределенных команд. Вот некоторые из этих особенностей:

1. Коммуникация и сотрудничество:

В дистанционной работе коммуникация играет решающую роль. Команды должны использовать средства коммуникации, такие как видеоконференции, чаты, электронную почту и совместные инструменты для обмена информацией и сотрудничества. Важно установить ясные каналы связи и регулярно общаться для поддержания эффективного взаимодействия.

2. Управление задачами и контроль прогресса:

Дистанционные команды нуждаются в эффективных инструментах для управления задачами и контроля прогресса. Использование инструментов управления проектами, таких как доски задач и системы отслеживания прогресса, позволяет командам лучше организовывать работу, отслеживать выполнение задач и управлять приоритетами.

3. Доверие и автономия:

Дистанционные команды часто нуждаются в большей степени автономии и доверии. Они должны иметь возможность принимать решения и самостоятельно организовывать свою работу. Установление четких целей, общих ценностей и ожиданий помогает создать атмосферу доверия и поддерживает автономность команды.

4. Гибкость и адаптивность:

Дистанционные команды часто сталкиваются с неожиданными изменениями и вызовами. Гибкие методологии предоставляют командам инструменты и подходы для быстрой адаптации к изменяющимся условиям и требованиям.

5. Организация времени и управление рабочим процессом:

Дистанционные команды должны быть самоорганизованными и эффективно управлять своим временем. Установление четкого рабочего расписания, придерживание принципов приоритизации и установление ясных ожиданий по срокам помогает командам эффективно организовывать работу и достигать результатов.

Дистанционная работа может быть вызовом, но с правильным подходом и инструментами можно создать эффективные и сотрудничающие команды, работающие удаленно.

**12. Поток поставки ценности. DevOps – понятия и основные принципы**

Поток поставки ценности (Value Stream) является концепцией, которая описывает цепочку действий, необходимых для создания и доставки ценности клиенту или конечному пользователю. В контексте DevOps, поток поставки ценности относится к процессу разработки и развертывания программного обеспечения.

DevOps (Development and Operations) — это методология, объединяющая разработку (Development) и операционную деятельность (Operations) в единый, коллаборативный процесс. DevOps направлен на улучшение коммуникации, сотрудничества и автоматизации между разработчиками программного обеспечения и IT-операционными командами.

Ключевые принципы DevOps описаны в модели CALMS. Ее используют для оценки готовности компании к внедрению процессов DevOps и измерения успешности внедрения подхода. Аббревиатура CALMS расшифровывается, как:

*• культура (culture).* ИТ-специалисты по разработке, тестированию и эксплуатации образуют единую функциональную команду, которая отвечает за проект в целом. Поэтому они вовлечены в работу, активно взаимодействуют между собой и несут ответственность за качество конечного продукта;

*• автоматизация (automation).* Все процессы разработки максимально автоматизированы, чтобы снизить влияние человеческого фактора, повысить производительность команды и освободить время ИТ-специалистов на написание кода и генерацию новых идей;

*• бережливость (lean).* ИТ-специалисты работают по методике agile, в рамках которой они постоянно совершенствуют процессы, уделяют внимание экспериментам, оптимизируют использование ресурсов и увеличивают скорость работы над продуктом;

*• измерение (measurement).* Команды DevOps используют короткие циклы обратной связи с клиентами и конечными пользователями, мониторят и измеряют производительность программного обеспечения, чтобы использовать эту информацию для дальнейших улучшений и новых релизов;

*• обмен (sharing).* Быстрый выпуск релизов возможен, только когда у команд, которые работают над программным обеспечением, есть общая цель, доверие, открытая коммуникация и четкое понимание, на каком этапе сейчас находится конкретная задача.

**13. DevOps. Компоненты и процессы. Понятие безопасной разработки**

DevOps включает в себя различные компоненты и процессы, которые совместно обеспечивают эффективную разработку и доставку программного обеспечения. DevOps состоит из нескольких компонентов, которые взаимодействуют друг с другом:

*· Continuous Integration (CI)* — это процесс автоматической сборки и тестирования кода каждый раз, когда вносятся изменения в код. Это позволяет предотвратить конфликты и ошибки в коде, а также ускоряет процесс разработки.

*· Continuous Delivery (CD)* — это процесс автоматической доставки ПО в производственную среду после прохождения всех этапов тестирования. Это позволяет ускорить процесс развертывания и минимизировать риски, связанные с ручными ошибками.

*· Configuration Management (CM)* — это процесс автоматического управления конфигурацией системы. Это позволяет легко и быстро изменять настройки системы и управлять ее состоянием.

*· Monitoring and Logging* — это процесс автоматического мониторинга и логирования работы системы. Это позволяет быстро обнаруживать и устранять проблемы в работе системы.

*· Infrastructure as Code (IaC)* — это процесс автоматического управления инфраструктурой системы через код. Это позволяет легко и быстро создавать, изменять и удалять инфраструктуру системы.

DevOps также включает в себя несколько процессов, которые помогают улучшить и ускорить процесс разработки:

*· Agile Development* — это методология разработки ПО, основанная на гибком и итеративном подходе. Она позволяет быстро реагировать на изменения требований и максимально эффективно использовать ресурсы.

*· Continuous Testing* — это процесс автоматического тестирования ПО на каждом этапе разработки. Это позволяет обнаруживать и устранять ошибки в ранней стадии разработки и повышать качество ПО.

*· Continuous Deployment* — это процесс автоматической доставки ПО в производственную среду после прохождения всех этапов тестирования. Это позволяет ускорить процесс развертывания и минимизировать риски, связанные с ручными ошибками.

*· Continuous Monitoring* — это процесс автоматического мониторинга работы системы. Это позволяет быстро обнаруживать и устранять проблемы в работе системы.

*· Continuous Feedback* — это процесс обратной связи между разработчиками и пользователями. Он позволяет быстро реагировать на обратную связь, улучшать качество ПО и удовлетворять потребности пользователей.

Понятие безопасной разработки (Secure DevOps) относится к интеграции безопасности в процессы DevOps. Это включает в себя внедрение практик и инструментов для обнаружения и устранения уязвимостей, управления доступом, мониторинга безопасности и автоматизации безопасных процессов разработки и развертывания. Безопасная разработка ставит целью обеспечить безопасность приложений и защиту данных в рамках DevOps-цикла, сокращая риски и возможность возникновения уязвимостей.

**15. Тестирование ПО. Цели и принципы**

Тестирование (Testing) — процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта. Конечной целью тестирования является предоставление пользователю качественного программного обеспечения (ПО). Тестирование программного обеспечения (ПО) является важной частью DevOps-процесса, поскольку помогает обеспечить высокое качество и надежность разрабатываемого и доставляемого ПО.

Цели тестирования ПО в DevOps:

*1. Обеспечение качества:* Главная цель тестирования ПО в DevOps - это проверка функциональности, производительности, безопасности и других атрибутов ПО для обеспечения высокого уровня качества и соответствия требованиям.

*2. Раннее обнаружение дефектов:* Тестирование в DevOps проводится непрерывно на протяжении всего процесса разработки и доставки, что позволяет обнаруживать дефекты и проблемы как можно раньше, снижая затраты на их исправление.

*3. Быстрая обратная связь:* Тестирование в DevOps предоставляет быструю обратную связь разработчикам о проблемах и дефектах, что позволяет им оперативно реагировать и вносить исправления.

*4. Автоматизация:* Целью тестирования в DevOps является автоматизация тестовых процессов, чтобы снизить время и ресурсы, необходимые для проведения тестов, и обеспечить повторяемость и консистентность результатов.

Принципы тестирования ПО:

*1. Планомерность:* Тестирование должно быть организовано и планомерно проводиться на всех этапах разработки ПО.

*2. Раннее тестирование:* Тестирование должно начинаться как можно раньше в процессе разработки ПО, поскольку раннее обнаружение ошибок существенно уменьшает затраты на их исправление.

*3. Повторное тестирование:* После исправления ошибок важно провести повторное тестирование, чтобы убедиться в их полном устранении и отсутствии новых дефектов.

*4. Тестирование не может быть абсолютным:* Из-за огромного количества вариантов использования и состояний системы, невозможно протестировать каждый возможный сценарий. Тестирование стремится покрыть максимально возможное количество сценариев, но 100% покрытие тестами невозможно.

**14. Документация. Типы, назначение. Принципы создания, группы заинтересованных лиц, методики формирования пакетов документации.**

Характер архитектуры любой системы обусловливается предъявляемыми к ней требованиями; это утверждение справедливо и по отношению к документации системы — содержание документации зависит от предполагаемых вариантов ее применения. Документация ни при каких обстоятельствах не может быть универсальной. С одной стороны, она должна быть абстрактной и, следовательно, доступной для понимания новыми сотрудниками, но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа. Перечислим основные типы документации и их назначение:

*1. Техническая документация:* Включает в себя технические спецификации, архитектурные диаграммы, описания API и другую информацию, необходимую разработчикам для понимания системы и ее компонентов.

*2. Инструкции по развертыванию и управлению:* Описывают процессы и инструкции по развертыванию приложения, настройке окружения, управлению конфигурациями и другими операционными задачами.

*3. Руководства для пользователей:* Предоставляют информацию и инструкции для конечных пользователей приложения. Это может включать руководства по использованию, FAQ, решение проблем и т. д.

*4. Документация по безопасности:* Содержит руководства и рекомендации по безопасности, политики доступа, управлению учетными записями и другой информации, связанной с безопасностью системы.

Принципы создания документации в рамках DevOps:

*1. Простота и понятность:* Документация должна быть простой, легко понятной и ориентированной на конечного пользователя. Используйте понятный язык, примеры и диаграммы для упрощения понимания.

*2. Согласованность:* Соблюдайте единообразие и согласованность в структуре, формате и стиле документации. Это упрощает ее использование и понимание.

*3. Актуальность:* Документация должна быть постоянно обновляемой и отражать текущее состояние системы и процессов.

*4. Коллаборация:* Вовлекайте разработчиков, операционные команды и другие заинтересованные стороны в создание и обновление документации. Обмен знаниями и опытом помогает создать более полезную и точную документацию.

Группы заинтересованных лиц в создании документации:

*1. Архитектор и разработчики требований:* Обсуждение конфликтующих требований и поиск компромиссов.

*2. Проектировщики составляющих систему элементов:* Разрешение состязаний за ресурсы, составление бюджета производительности и других бюджетов потребления ресурсов периода прогона.

*3. Разработчики и проектировщики:* Установление жестких ограничений применительно к нисходящим операциям разработки (и допустимых вольностей).

*4. Тестировщики и сборщики:* Регламентация корректного поведения совмещаемых элементов при тестировании методом «черного ящика».

*5. Специалисты по сопровождению:* Выявление областей влияния предполагаемых изменений.

*6.* *Проектировщики сторонних систем, взаимодействующих с рассматриваемой системой:* Установление набора предоставляемых и требуемых операций, а также определение протокола их исполнения.

*7.* *Специалисты по атрибутам качества:* Разработка модели, выступающей в качестве основы для функционирования аналитических инструментов: частотно-монотонного анализа возможности планирования в реальном времени, моделирования и т. д. Информации, содержащейся в документации, должно быть достаточно для оценки различных атрибутов качества: безопасности, производительности, практичности, готовности и модифицируемости.

*8–9.* *Руководители, Группа контроля качества:* Формирование групп разработчиков согласно установленному распределению функций, планирование и распределение ресурсов проекта, контроль действий, выполняемых различными группами.

Методики формирования пакетов документации:

*1. Минимально необходимая документация:* Определите минимальный набор документов, необходимых для разработки, развертывания и использования системы. Это поможет избежать избыточной документации и сосредоточиться на ключевых аспектах.

*2. Итеративный подход:* Создавайте и обновляйте документацию в процессе разработки и доставки продукта. Начните с основных элементов, а затем постепенно дополняйте и расширяйте ее.

*3. Автоматизация генерации документации:* Используйте инструменты, такие как генераторы документации, чтобы автоматически создавать часть документации из исходного кода, комментариев и конфигурационных файлов. Это может сократить время и упростить процесс создания документации.

*4. Обратная связь и рецензирование:* Предоставьте возможность разработчикам, операционным командам и пользователям вносить свои предложения и корректировки в документацию. Это поможет улучшить ее качество и полезность.

**16. Тестирование. Уровни и типы тестирования.**

В рамках DevOps существуют различные уровни и типы тестирования, которые выполняются для обеспечения качества и надежности разрабатываемого программного обеспечения. Ниже перечислены основные уровни и типы тестирования в DevOps:

Уровни тестирования:

*1. Модульное тестирование (Unit testing):* Это тестирование отдельных модулей или компонентов программы. Оно проверяет правильность работы каждого модуля изолированно от остальных компонентов системы.

*2. Интеграционное тестирование (Integration testing):* На этом уровне проверяется взаимодействие между различными модулями или компонентами системы. Целью является обнаружение ошибок, возникающих при интеграции модулей.

*3. Системное тестирование (System testing):* Это тестирование всей системы в целом. Оно проверяет соответствие системы требованиям и ее работоспособность в целом. Включает тестирование функциональности, производительности, безопасности и других атрибутов системы.

*4. Приемочное тестирование (Acceptance testing):* Это тестирование, проводимое с целью проверки соответствия системы бизнес-требованиям и ожиданиям заказчика. Обычно проводится совместно с заказчиком или конечными пользователями.

Типы тестирования:

*1. Функциональное тестирование (Functional testing):* Это тестирование функциональности системы с целью проверки, выполняет ли она заданные функции в соответствии с требованиями.

*2. Нагрузочное тестирование (Load testing):* Проверяет производительность системы при нагрузке, чтобы убедиться, что она может обрабатывать требуемое количество запросов и поддерживать определенный уровень производительности.

*3. Тестирование безопасности (Security testing):* Это тестирование, направленное на выявление уязвимостей и проверку безопасности системы, включая защиту данных, аутентификацию и авторизацию.

*4. Автоматизированное тестирование (Automated testing):* Это использование инструментов и скриптов для автоматизации выполнения тестовых сценариев. Это позволяет повторять тесты, снижает время и ресурсы, необходимые для тестирования и обеспечивает более высокую точность и скорость выполнения тестов.

*5. Регрессионное тестирование (Regression testing):* Проверяет, не возникли ли новые ошибки после внесения изменений в систему или исправления дефектов, и что существующая функциональность осталась работоспособной.

*6. Тестирование пользовательского интерфейса (UI testing):* Это тестирование взаимодействия пользователя с пользовательским интерфейсом системы, проверка его удобства, надежности и соответствия требованиям.

**18. Тестирование. Методы и инструменты. Классификация дефектов**

*Методы тестирования:*

• Модульное тестирование: Тестирование отдельных модулей (например, функций или классов) программы для проверки их корректности и работы в изоляции.

• Интеграционное тестирование: Тестирование взаимодействия и корректности работы различных модулей или компонентов программы вместе.

• Системное тестирование: Тестирование всей системы в целом для проверки ее соответствия требованиям и ожидаемому поведению.

• Приемочное тестирование: Тестирование программы с целью проверки ее готовности к принятию заказчиком или конечным пользователям.

*Инструменты тестирования:*

• Фреймворки для автоматизированного тестирования, такие как Selenium, Appium, JUnit, TestNG, PyTest, и другие. Они предоставляют средства для разработки, выполнения и анализа автоматизированных тестов.

• Инструменты для управления тестовыми случаями и требованиями, такие как TestRail, Zephyr, и другие. Они помогают организовать и управлять тестовой документацией, планами и результатами.

• Инструменты для непрерывного интеграционного и доставки (CI/CD), такие как Jenkins, Travis CI, GitLab CI/CD, и другие. Они автоматизируют процесс сборки, тестирования и доставки программного обеспечения.

• Инструменты для отслеживания дефектов и управления задачами, такие как JIRA, Trello, Redmine, и другие. Они позволяют командам управлять и отслеживать дефекты, задачи и исправления.

*Классификация дефектов:* Дефекты, или ошибки, в программном обеспечении могут быть классифицированы по разным признакам.

1. По типу дефекта:

• Функциональные дефекты: Ошибки, связанные с некорректной работой функциональности программы.

• Нефункциональные дефекты: Ошибки, связанные с некорректной работой аспектов, таких как производительность, безопасность, удобство использования и т. д.

2. По приоритету и серьезности:

• Критические дефекты: Дефекты, которые приводят к неработоспособности программы или серьезным сбоям.

• Высокоприоритетные дефекты: Дефекты, которые могут значительно повлиять на работу программы или представляют высокий риск.

• Низкоприоритетные дефекты: Дефекты, которые имеют небольшое влияние на работу программы или могут быть исправлены в последующих версиях.

3. По источнику дефекта:

• Дефекты проектирования: Ошибки, связанные с некорректным проектированием программы или архитектурой системы.

• Дефекты реализации: Ошибки, связанные с ошибками в коде программы или неправильной реализацией функциональности.

• Дефекты среды выполнения: Ошибки, связанные с проблемами окружения выполнения программы, такими как настройка операционной системы, зависимости и другие факторы.

**17. Тестирование. Методы черного и белого ящика. Статическое и динамическое тестирование**

Тестирование программного обеспечения может быть классифицировано на основе разных методов и подходов. Два основных метода тестирования, которые широко используются, — это методы черного ящика и белого ящика. Кроме того, тестирование может быть статическим или динамическим.

Белый ящик (White Box Testing) – тестирование, основанное на анализе внутренней структуры компонентов или системы (у тестировщика есть доступ к внутренней структуре и коду приложения). Черный ящик (Black Box Testing) – тестирование системы без знания внутренней структуры и компонентов системы (у тестировщика нет доступа к внутренней структуре и коду приложения либо в процессе тестирования он не обращается к ним).

*Методы черного ящика (Black-box testing):*

• Подход черного ящика предполагает тестирование программного обеспечения без знания внутренней структуры или реализации кода.

• Тестировщик сосредотачивается на входных данных и ожидаемых выходных результатах, а также на функциональных требованиях и спецификациях системы.

• В методе черного ящика используются различные техники, такие как эквивалентное разбиение классов, граничное значение, таблицы принятия решений и т. д.

• Тестирование черного ящика может быть проведено как ручным, так и автоматизированным способом.

*Методы белого ящика (White-box testing):*

• Подход белого ящика основывается на понимании внутренней структуры, логики и реализации кода программного обеспечения.

• Тестировщик активно изучает и анализирует код, чтобы разработать тесты, которые позволят проверить каждую ветвь, путь и условие выполнения внутри программы.

• Используемые методы включают тестирование покрытия кода (code coverage), тестирование пути выполнения (path testing), тестирование условий (condition testing) и другие.

• Тестирование белого ящика обычно проводится разработчиками или специалистами по тестированию, имеющим доступ к исходному коду.

*Статическое тестирование (Static testing):*

• Статическое тестирование выполняется без актуального выполнения программы.

• Основная цель статического тестирования - выявить ошибки, дефекты и проблемы в исходном коде или других артефактах программного обеспечения, таких как документация, спецификации и т. д.

• Типичные методы статического тестирования включают обзоры кода (code reviews), инспекции, анализ статического кода и т. д.

*Динамическое тестирование (Dynamic testing):*

• Динамическое тестирование выполняется во время актуального выполнения программного обеспечения.

• Оно включает запуск тестовых случаев, ввод тестовых данных и проверку результатов выполнения программы.

• Цель динамического тестирования - проверить, соответствует ли программное обеспечение ожидаемым требованиям и спецификациям, и обнаружить ошибки и дефекты.

• Примеры методов динамического тестирования включают модульное тестирование, интеграционное тестирование, системное тестирование и т. д.

Оба метода (черного и белого ящика) и статическое и динамическое тестирование имеют свои преимущества и недостатки и могут быть эффективно применены в зависимости от требований и характеристик программного обеспечения, которое тестируется.

**22. Процесс разработки. Архитектура решения. Назначение, принципы выбора. Анализ и оценка**

Архитектура решения (Solution Architecture) в ITIL является ключевым аспектом при проектировании и разработке IT-систем и сервисов. Её назначение состоит в определении структуры, компонентов, интерфейсов и отношений между ними, чтобы обеспечить достижение бизнес-целей и требований.

Принципы выбора архитектуры решения в ITIL включают следующие аспекты:

*1. Выравнивание с бизнес-требованиями:* Архитектура решения должна соответствовать бизнес-стратегии и целям организации, поддерживая их эффективное достижение.

*2. Гибкость и масштабируемость:* Архитектура решения должна быть гибкой и способной адаптироваться к изменяющимся требованиям и масштабироваться в соответствии с ростом бизнеса.

*3. Использование стандартов:* Архитектура решения должна основываться на существующих стандартах и лучших практиках, чтобы обеспечить совместимость, интеграцию и переиспользование компонентов.

*4. Эффективное использование ресурсов:* Архитектура решения должна оптимально использовать доступные ресурсы, такие как оборудование, программное обеспечение и персонал, чтобы достичь оптимальной производительности и эффективности.

*5. Управление рисками и безопасностью:* Архитектура решения должна учитывать требования безопасности и риски, связанные с конфиденциальностью, целостностью и доступностью данных и систем.

Анализ и оценка архитектуры решения включает процесс изучения и проверки архитектурных решений на соответствие бизнес-требованиям, стандартам и принципам выбора. Это включает оценку технических аспектов, взаимодействий между компонентами, принципов безопасности и рисков, а также выявление потенциальных улучшений и оптимизаций.

Анализ и оценка архитектуры решения помогают обеспечить качество, надежность, гибкость и эффективность IT-систем и сервисов, а также выявить возможности для их улучшения и совершенствования.

**19. Сопровождение и поддержка ПО. ITSM и ITIL общие понятия и принципы**

Сопровождение и поддержка программного обеспечения (ПО) являются важными аспектами в жизненном цикле ПО. Они включают в себя деятельности, связанные с обеспечением непрерывной работы и поддержкой программных систем после их внедрения. Два основных подхода, связанных с сопровождением и поддержкой ПО, — это ITSM (IT Service Management) и ITIL (IT Infrastructure Library). Давайте рассмотрим их общие понятия и принципы:

ITSM (IT Service Management) — это набор методологий и практик, направленных на эффективное управление предоставлением IT-услуг организации. Основная цель ITSM - обеспечить высокий уровень качества IT-услуг, соответствующий потребностям бизнеса. Он охватывает процессы, инструменты и людей, необходимых для оказания IT-услуг.

ITIL (IT Infrastructure Library) — это набор передовых практик в области ITSM, который описывает лучшие методы и процессы для управления IT-услугами. ITIL устанавливает набор рекомендаций и рамок, которые помогают организациям повысить эффективность и эффективность своих IT-сервисов. ITIL включает в себя многоэтапный жизненный цикл IT-услуги, который охватывает ее стратегию, проектирование, переход, операцию и улучшение.

Некоторые общие понятия и принципы ITSM и ITIL:

*1. IT-услуга:* ИТ-услуга — это средство, предоставляемое IT-организацией для удовлетворения потребностей бизнеса. Она может включать в себя различные компоненты, такие как программное обеспечение, оборудование, процессы и документацию.

*2. Процессы ITSM/ITIL:* ITSM и ITIL определяют ряд процессов, которые помогают организации управлять IT-услугами. Некоторые из них включают управление инцидентами, управление изменениями, управление проблемами, управление уровнем услуг, управление конфигурациями и др.

*3. Ключевые роли и ответственности:* ITSM и ITIL определяют роли и ответственности, связанные с управлением IT-услугами. Примеры ролей включают менеджера инцидентов, менеджера изменений, координатора релизов, специалиста по уровню услуг и др.

*4. Непрерывное улучшение:* ITSM и ITIL акцентируют внимание на непрерывном улучшении процессов и сервисов. Это осуществляется через процессы мониторинга, анализа данных, выявления проблем и внедрения улучшений.

*Пример сопровождения и поддержки ПО:* Представим, что вы разработали систему управления контентом (CMS) для веб-сайта компании. После ее внедрения ваши обязанности включают регулярное обновление системы для обеспечения ее безопасности, устранение обнаруженных ошибок, добавление новых функций по запросу пользователей и оптимизацию системы для повышения производительности.

*Пример ITSM:* Компания может иметь центр обслуживания клиентов, который отвечает за поддержку IT-сервисов, предоставляемых клиентам. Это может включать управление инцидентами (устранение проблем с IT-сервисами), управление проблемами (определение и устранение причин инцидентов) и управление изменениями (внедрение новых версий IT-сервисов).

*Пример ITIL:* Организация может применять ITIL для стандартизации своих IT-процессов. Например, она может использовать процесс управления инцидентами ITIL для структурированного и эффективного управления IT-инцидентами. Это может включать обнаружение и запись инцидентов, их классификацию и приоритезацию, расследование и диагностику, а также восстановление и затем закрытие инцидентов.

**25. Атрибуты качества и сценарии - подробное описание на примере одного из типа сценариев (Модифицируемость, Ремонтопригодность, Производительность, Безопасность...)**

Атрибуты качества систем определяют некоторые особенности, которые должны быть присутствующими в программном обеспечении или системе, чтобы удовлетворить требования пользователей и обеспечить их эффективное функционирование.

Сценарием атрибута качества называется требование, путем выполнения которого этот атрибут реализуется. Сценарии атрибутов качества используются для более детального определения и проверки атрибутов качества системы. Они описывают конкретные ситуации или условия, в которых система должна проявлять определенное поведение или демонстрировать определенные качества.

Рассмотрим атрибут качества "Производительность" и сценарий, связанный с ним.

Производительность — это временная характеристика. Когда фиксируются те или иные события (прерывания, сообщения, пользовательские запросы, временные промежутки), система должна на них реагировать. Характеристик поступления событий и реакций на них может быть много, по сути, вопрос заключается в том, за какое время система справляется с реакцией на то или иное событие. Стимулом для всех сценариев производительности являются поступающие события. Образец поступления может быть периодическим, непереодическим и случайным. Пример стимула: непереодическое инициирование 100 000 транзакций в минуту. Артефактом для всех сценариев производительности являются предоставляемые системой услуги.

Составление общего сценария производительности:

|  |  |
| --- | --- |
| **Элемент сценария** | **Возможные значения** |
| Источник | Ряд независимых источников, часть из которых может находиться внутри системы |
| Стимул | Периодическое, непериодическое либо случайное поступление событий |
| Артефакт | Система, ее функции (услуги) |
| Условия | Нормальный режим; Перегруженный режим |
| Реакция | Обработка стимулов; Изменения уровня обслуживания |
| Количественная мера реакции | Задержка, предельный срок, пропускная способность, неустойчивость, коэффициент неудач, потеря данных |

**20. Сопровождение и поддержка. ITIL. Базовые процессы**

ITIL (IT Infrastructure Library) определяет несколько базовых процессов, которые используются в сопровождении и поддержке IT-услуг. Эти процессы обеспечивают эффективное управление IT-сервисами и помогают достичь высокого уровня качества и удовлетворения потребностей бизнеса.

Ниже перечислены основные процессы ITIL:

*1. Управление инцидентами (Incident Management):* Процесс управления инцидентами отвечает за регистрацию, классификацию, приоритизацию, назначение ответственных лиц и восстановление нормальной работы в случае возникновения инцидентов или нарушений в работе IT-сервисов.

*2. Управление проблемами (Problem Management):* Процесс управления проблемами направлен на исследование корневых причин инцидентов, предотвращение их повторного возникновения и минимизацию отрицательного влияния на бизнес. Включает в себя анализ проблем, разработку и внедрение долгосрочных решений.

*3. Управление изменениями (Change Management):* Процесс управления изменениями отвечает за планирование, оценку, авторизацию, координацию и контроль изменений в IT-инфраструктуре. Целью процесса является минимизация рисков и обеспечение успешного внедрения изменений в среде эксплуатации.

*4. Управление уровнем услуг (Service Level Management):* Процесс управления уровнем услуг направлен на установление, согласование и мониторинг уровня предоставляемых IT-услуг, а также обеспечение соответствия уровня услуг требованиям бизнеса и клиентов.

*5. Управление конфигурациями и активами (Configuration and Asset Management):* Процесс управления конфигурациями и активами отвечает за управление информацией о конфигурациях IT-систем, компонентах и активах. Включает в себя учет, контроль, анализ и управление изменениями в конфигурации и активах.

*6. Управление выпуском и развертыванием (Release and Deployment Management):* Процесс управления выпусками и развертыванием отвечает за планирование, координирование и контроль выпуска и внедрения новых или измененных компонентов ПО и IT-сервисов в среде эксплуатации.

*7. Управление доступом (Access Management):* Процесс управления доступом отвечает за управление правами доступа пользователей к IT-системам и сервисам. Включает в себя управление идентификацией, аутентификацией, авторизацией и аудитом доступа.

Пример: Компания X ведет записи всех SLA, заключенных с ее клиентами, и регулярно проверяет свои IT-услуги, чтобы убедиться, что они соответствуют или превосходят эти SLA.

**21. Управление изменениями. ITIL и процесс управления изменениями**

Управление изменениями (Change Management) является одним из ключевых процессов в рамках фреймворка ITIL (IT Infrastructure Library). Он направлен на эффективное управление изменениями в IT-инфраструктуре и сервисах организации с целью минимизации рисков и обеспечения успешного внедрения изменений.

Процесс управления изменениями в ITIL включает следующие основные шаги и активности:

*1. Запрос на изменение (Request for Change, RFC):* Процесс начинается с поступления запроса на изменение, который может быть инициирован различными сторонами, включая пользователей, бизнес-заказчиков, команды разработки или поддержки. Запрос на изменение описывает необходимые изменения, их обоснование и ожидаемые выгоды.

*2. Оценка изменения:* Запрос на изменение проходит процесс оценки, где рассматривается его влияние на IT-инфраструктуру, сервисы, бизнес-процессы и риски. Важно определить, как изменение будет влиять на текущую среду и выявить связанные ресурсы и стейкхолдеров.

*3. Авторизация изменения:* после оценки запроса на изменение он должен быть авторизован. Это включает согласование изменения со всеми заинтересованными сторонами и принятие решения о дальнейшем выполнении изменения.

*4. Планирование и координация изменения:* В этом этапе процесса определяются необходимые шаги и ресурсы для реализации изменения. Важным аспектом является планирование времени, распределение ролей и обязанностей, а также согласование с другими процессами и командами.

*5. Тестирование и оценка изменения:* прежде чем изменение будет внедрено в рабочую среду, необходимо провести тестирование, чтобы проверить его работоспособность и минимизировать потенциальные риски. Оценка изменения включает анализ результатов тестирования и убеждение в том, что изменение соответствует требованиям и ожиданиям.

*6. Внедрение изменения:* на этом этапе изменение фактически внедряется в рабочую среду. Это может включать установку нового ПО, конфигурацию систем, обновление оборудования и другие необходимые шаги для реализации изменения.

*7. Оценка и управление изменением:* после внедрения изменения необходимо оценить его результаты и эффективность. Отслеживаются и анализируются данные о производительности системы и обратная связь от пользователей и заинтересованных сторон. Если необходимо, принимаются корректирующие действия и проводится дальнейшее управление изменением.

*8. Закрытие запроса*

Пример: Представим, что IT-компания хочет обновить свое серверное программное обеспечение. Это изменение начинается с запроса на изменение, который описывает, что именно нужно обновить, почему, как это будет сделано и какие потенциальные риски могут возникнуть. Затем команда управления изменениями оценивает RFC, рассматривая потенциальное влияние на бизнес, стоимость и риски. Если они решают, что изменение стоит реализовать, они авторизуют его. Затем планируется и проводится обновление серверного программного обеспечения. После того, как изменение было внедрено, оно проверяется на эффективность и, если все прошло успешно, изменение закрывается.

**23. Процесс разработки. Архитектура. Документирование и распространение сведений. Группы заинтересованных лиц.**

Процесс разработки программного продукта — это сложный процесс, который включает в себя множество этапов, начиная с идеи и заканчивая выпуском готового продукта. В рамках DevOps, архитектура является ключевым аспектом проектирования и разработки IT-систем и сервисов. Она определяет структуру, компоненты, интерфейсы и отношения между ними, чтобы обеспечить достижение бизнес-целей и требований.

Архитектура программного продукта включает в себя его структуру и компоненты, а также определяет, как эти компоненты взаимодействуют между собой и как они связаны. Хорошая архитектура обеспечивает легкость сопровождения и расширения продукта. Один из ключевых аспектов архитектуры — это разделение продукта на модули. Каждый модуль должен выполнять определенную функцию и быть легко переиспользуемым в других проектах. Кроме того, в архитектуре должны учитываться требования к производительности, масштабируемости и безопасности.

Документирование и распространение сведений об архитектуре является важной практикой в DevOps, чтобы обеспечить понимание и согласованность у всех заинтересованных сторон. Это позволяет снизить риск непонимания и улучшить коммуникацию и сотрудничество между различными командами и ролями в рамках разработки и эксплуатации системы.

Документирование является важной частью процесса разработки программного продукта. Он помогает убедиться, что все участники проекта понимают требования и процесс разработки. Распространение сведений также важно, чтобы гарантировать, что все участники проекта имеют доступ к необходимой информации. Документирование может включать в себя спецификации требований, техническую документацию, диаграммы и протоколы совещаний. Важно поддерживать документацию в актуальном состоянии и обеспечивать ее доступность для всех участников проекта.

Группы заинтересованных лиц (Stakeholders) в контексте архитектуры включают различные роли и команды, которые имеют интерес или влияние на архитектуру и функционирование IT-системы. Группы заинтересованных лиц включают:

*1. Архитектор и разработчики требований:* Обсуждение конфликтующих требований и поиск компромиссов.

*2. Проектировщики составляющих систему элементов:* Разрешение состязаний за ресурсы, составление бюджета производительности и других бюджетов потребления ресурсов периода прогона.

*3. Разработчики и проектировщики:* Установление жестких ограничений применительно к нисходящим операциям разработки (и допустимых вольностей).

*4. Тестировщики и сборщики:* Регламентация корректного поведения совмещаемых элементов при тестировании методом «черного ящика».

*5. Специалисты по сопровождению:* Выявление областей влияния предполагаемых изменений.

*6. Проектировщики сторонних систем, взаимодействующих с рассматриваемой системой:* Установление набора предоставляемых и требуемых операций, а также определение протокола их исполнения.

*7. Специалисты по атрибутам качества:* Разработка модели, выступающей в качестве основы для функционирования аналитических инструментов: частотно-монотонного анализа возможности планирования в реальном времени, моделирования и т. д. Информации, содержащейся в документации, должно быть достаточно для оценки различных атрибутов качества: безопасности, производительности, практичности, готовности и модифицируемости.

*8–9. Руководители, Группа контроля качества:* Формирование групп разработчиков согласно установленному распределению функций, планирование и распределение ресурсов проекта, контроль действий, выполняемых различными группами.

**24. Атрибуты качества систем и их связь с функциональными требованиями. Сценарии атрибутов качества**

Атрибуты качества систем определяют некоторые особенности, которые должны быть присутствующими в программном обеспечении или системе, чтобы удовлетворить требования пользователей и обеспечить их эффективное функционирование. Эти атрибуты качества описывают различные аспекты системы, такие как производительность, надежность, безопасность, поддерживаемость, удобство использования и др.

Архитектурно-экономический цикл, как правило, основан на требованиях к качеству, определяемыми прямыми коммерческими мотивами, то есть, функциональностью разрабатываемой программы или программного комплекса. Это предполагает включения атрибутов качества как элементов требований к функциональности – при описании возможностей и поведения системы и оказываемых ей услуг. Функциональность становится ведущим показателем качества и занимает в схеме разработки центральное, или даже единоличное положение. Проблема: Причины переработки систем часть кроются НЕ в недостатке функциональности. Причинами, потребовавшими изменения систем чаще всего, становятся: Трудности сопровождения; Проблемы масштабирования; Низкая производительность; Недостаточная стойкость к атакам; Неготовность к переносу.

Вопросы воплощения атрибутов качества решаются в периоды проектирования, реализации и развертывания. Не существует ни одного атрибута качества, который бы зависел только от какого-то одного этапа программного цикла. Примеры:

* Практичность имеет как архитектурные, так и не архитектурные аспекты. Примеры последних – обеспечение ясности и простоты применения пользовательского интерфейса. Что выбрать: переключатель или флажок? Какая схема размещения на экране наиболее интуитивна? Какая гарнитура шрифта отображается четче других? Эти вещи имеют большое значение для конечного пользователя, но при этом являются локальными решениями стадии реализации.
* Модифицируемость определяется тем, каким образом происходит разделение функциональности (архитектурный аспект) и методик кодирования в рамках отдельного модуля (аспект тактической реализации).
* Производительность также зависит от архитектурных и неархитектурных факторов. Она зависит от объемов информации, передающейся между компонентами, от распределения функциональности между компонентами, управления общими ресурсами – все это архитектурные аспекты. Но на производительность влияют и выбранные для реализации функциональности алгоритмы, и качество их реализации (кодирования) – а это неархитектурные аспекты.

Сценарием атрибута качества называется требование, путем выполнения которого этот атрибут реализуется. Сценарии атрибутов качества используются для более детального определения и проверки атрибутов качества системы. Они описывают конкретные ситуации или условия, в которых система должна проявлять определенное поведение или демонстрировать определенные качества. Сценарий состоит из шести элементов:

* *Источник стимула.* Это некий субъект (человек, вычислительная система), который порождает символ.
* *Стимул.* Стимулом называется наблюдаемое в системе явление, требующее к себе внимания.
* Условия. Стимул возникает при конкретных условиях. К примеру, система может находиться в состоянии перегрузки или исполняться в обычном режиме.
* *Артефакт.* Объектом воздействия стимула является некий артефакт. В этом качестве выступает система в целом или ее отдельный элемент.
* *Реакция.* Реакция – это действие, предпринятое в ответ на появление стимула.
* *Количественная мера реакции.* Предпринимаемые в ответ на стимул действия должны быть измеримы – только тогда соответствие требованию можно проверить.

Пример сценария готовности:

Изображение выглядит как текст, снимок экрана, диаграмма, дизайн

Автоматически созданное описание

**26. Тактики обеспечения атрибутов качества (на примере одного из типов Модифицируемость, Ремонтопригодность, Производительность, Безопасность...)**

Тактика — это проектное решение, которое влияет на управление реакцией по атрибуту качества. Совокупность тактик называется архитектурной стратегией (architectural strategy). Тактики предназначены для управления реакцией на стимулы.

Каждая тактика — это проектная альтернатива. Предположим, к примеру, что одна из возможных тактик повышает готовность системы за счет реализации резервирования. Повышение готовности — это лишь одна из многочисленных альтернатив, среди которых архитектор волен выбирать. Как правило, повышение готовности через резервирование сопровождается синхронизацией (которая нужна для того, чтобы в случае повреждения оригинала можно было обратиться к резервной копии). Из этого примера можно сделать два вывода.

1. Одна тактика может использоваться для уточнения другой. Мы назвали резервирование тактикой. В свою очередь, можно различать «резервирование данных» (в системе баз данных) или «резервирование вычислений» (во встроенной системе управления). И то и другое — тактики. Проектировщик может пойти еще дальше и путем уточнения максимально конкретизировать резервирование. В отношении всех атрибутов качества, которые нам предстоит рассмотреть, мы намерены структурировать тактики в рамках иерархических систем.

2. Упаковка тактик в образцы. Любой образец (паттерн), призванный обеспечивать готовность, вероятнее всего, предусматривает совместное применение тактик резервирования и синхронизации. Кроме того, они в нем, наверное, конкретизированы. Ближе к завершению данного раздела мы представим пример образца, описанного в категориях тактик.

Рассмотрим тактики обеспечения атрибута качества "Производительность":

После поступления события система приступает к его обработке; по тем или иным причинам обработка может быть заблокирована. Отсюда делаем вывод о двух основных слагаемых времени отклика — это потребление ресурсов и продолжительность блокирования.

1. Потребление ресурсов. Ресурсы — это центральный процессор, хранилища данных, пропускная способность сетевых соединений и память; кроме того, в категорию ресурсов попадают некоторые сущности, задаваемые конкретной проектируемой системой. К примеру, речь может идти об управлении буферами и обеспечении последовательного доступа к критическим секциям. Каждому из перечисленных типов событий соответствует собственный цикл обработки. К примеру, сообщение генерируется одним компонентом, а затем через сетевое соединение поступает другому компоненту. Затем оно размещается в буфере, каким-то образом преобразуется, обрабатывается по определенному алгоритму, опять преобразуется в выходную форму, помещается в выходной буфер и, наконец, отправляется очередному компоненту, системе или пользователю. Общая задержка обработки события складывается из задержек всех этапов.

2. Продолжительность блокирования. В ходе проведения вычислений доступ к ресурсу может быть заблокирован; поводами к блокированию могут быть состязание за ресурс, его неготовность или зависимость данного вычисления от результатов других вычислений, которые еще не завершены.

Состязание за ресурсы. События могут поступать в систему одним или несколькими потоками. Если ряд потоков или несколько событий в составе одного потока соперничают за право обращения к одному и тому же ресурсу, происходит задержка. Как правило, чем серьезнее состязание за ресурс, тем больше вероятность возникновения задержки. С другой стороны, это зависит от арбитража и способа обработки механизмом арбитража отдельных запросов.

Готовность ресурсов. Если состязания не наблюдается, но требуемый ресурс недоступен, вычисление останавливается. Причинами неготовности ресурса могут быть, к примеру, его неоперативное состояние или сбой в компоненте.

Зависимость от результатов других вычислений. Задержка вычисления иногда обусловливается необходимостью синхронизации с результатами других вычислений или ожиданием результатов инициированного вычисления. К примеру, если при получении информации из двух разных источников они считываются последовательно, задержка будет больше, чем если бы они считывались параллельно.

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

**27. Документирование программных систем. Заинтересованные лица, представления и нотации (с примерами).**

Документирование программных систем является важной частью процесса разработки программного обеспечения. Оно включает создание различных документов и записей, которые описывают систему, ее функциональность, архитектуру, требования и другую важную информацию. Характер архитектуры любой системы обусловливается предъявляемыми к ней требованиями; это утверждение справедливо и по отношению к документации системы — содержание документации зависит от предполагаемых вариантов ее применения. Документация ни при каких обстоятельствах не может быть универсальной. С одной стороны, она должна быть абстрактной и, следовательно, доступной для понимания новыми сотрудниками, но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа.

Заинтересованные в составлении документации лица преследуют разные цели — от представленной в ней они требуют разной направленности, разных уровней детализации и разных трактовок. Стремиться следует к тому, чтобы любое заинтересованное лицо смогло быстро найти необходимую информацию, при этом не тратя время на незначительные (с его точки зрения) сведения. В процессе документирования программных систем следующие заинтересованные стороны могут быть учтены:

*1. Архитектор и разработчики требований:* Обсуждение конфликтующих требований и поиск компромиссов.

*2. Проектировщики составляющих систему элементов:* Разрешение состязаний за ресурсы, составление бюджета производительности и других бюджетов потребления ресурсов периода прогона.

*3. Разработчики и проектировщики:* Установление жестких ограничений применительно к нисходящим операциям разработки (и допустимых вольностей).

*4. Тестировщики и сборщики:* Регламентация корректного поведения совмещаемых элементов при тестировании методом «черного ящика».

*5. Специалисты по сопровождению:* Выявление областей влияния предполагаемых изменений.

*6.* *Проектировщики сторонних систем, взаимодействующих с рассматриваемой системой:* Установление набора предоставляемых и требуемых операций, а также определение протокола их исполнения.

*7.* *Специалисты по атрибутам качества:* Разработка модели, выступающей в качестве основы для функционирования аналитических инструментов: частотно-монотонного анализа возможности планирования в реальном времени, моделирования и т. д. Информации, содержащейся в документации, должно быть достаточно для оценки различных атрибутов качества: безопасности, производительности, практичности, готовности и модифицируемости.

*8–9.* *Руководители, Группа контроля качества:* Формирование групп разработчиков согласно установленному распределению функций, планирование и распределение ресурсов проекта, контроль действий, выполняемых различными группами.

Вероятно, наиболее важным понятием документирования программной архитектуры является «представление» (view). С понятием представления, которое можно определить как способ фиксации структуры, связан основной принцип документирования программной архитектуры: Документирование архитектуры подразумевает документирование всех значимых представлений с последующей фиксацией сведений, относящихся одновременно к нескольким представлениям. Этот принцип полезен тем, что он помогает разбить проблему документирования архитектуры на ряд менее обширных элементов:

♦ выбор значимых представлений;

♦ документирование представления;

♦ документирование сведений, относящихся к нескольким представлениям.

Общие подходы к документированию программных систем: Типы нотаций, Нотации, «Лучшие практики», Инструменты.

Документирование программных систем может включать формальные, частично формальные и неформальные нотации. Пример нотаций: классические boxes-and-lines, например в PowerPoint (неформальные); UML, BPMN, ERD (частично формальные); AADL, ER (формальные). А так типичные представители нотаций это – UML, BPMN, C4, FlowChart/swimlane.

**28. Документирование программных систем. Неформальные нотации. Плюсы и минусы, области применения**

Документирование программных систем — это процесс создания и поддержки документации, которая описывает систему, ее компоненты, функциональность, атрибуты качества, требования, архитектуру и другую важную информацию. Характер архитектуры любой системы обусловливается предъявляемыми к ней требованиями; это утверждение справедливо и по отношению к документации системы — содержание документации зависит от предполагаемых вариантов ее применения. Документация ни при каких обстоятельствах не может быть универсальной. С одной стороны, она должна быть абстрактной и, следовательно, доступной для понимания новыми сотрудниками, но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа. Документирование программных систем может включать формальные, частично формальные и неформальные нотации. В данном случае рассмотрим неформальные нотации, их плюсы и минусы, а также области их применения.

Неформальные нотации в документировании программных систем – это диаграммы в свободной форме; семантика моделей на естественном языке (цвета, форма, надписи); информация диаграмм не может быть формально проанализирована; легко понятные, но при этом высок риск неправильного понимания. Пример неформальной нотации: классические boxes-and-lines, например в PowerPoint.

У диаграмм в неформальной нотации есть свои особенности:

* Для понимания требуется «ключ»
* Содержимое раскрывается с помощью фигур, цветов и стрелок
* Легко создавать
* Самые популярные
* Хороши для презентаций

Плюсы неформальных нотаций в документировании программных систем:

*1. Простота и доступность:* Неформальные нотации могут быть легко поняты и использованы различными заинтересованными сторонами, включая разработчиков, тестировщиков и пользователей. Они не требуют специальных знаний или обучения для их понимания.

*2. Гибкость и выразительность:* Неформальные нотации позволяют выразить идеи и концепции с большей гибкостью и свободой. Они могут быть адаптированы под конкретные потребности и контекст системы, а также могут включать примеры кода или сценариев использования для более наглядного представления.

*3. Сокращение времени и затрат:* Использование неформальных нотаций может быть более быстрым и экономически эффективным в сравнении с разработкой формальных документов или использованием специализированных инструментов. Это особенно полезно в небольших проектах или в ситуациях, когда требуется быстрое представление информации.

Минусы неформальных нотаций в документировании программных систем:

*1. Отсутствие строгой структуры:* В отличие от формальных нотаций, неформальные нотации могут не обладать строгой структурой и организацией, что может затруднить понимание и поиск нужной информации в документации.

*2. Разнообразие форм и стилей:* Использование неформальных нотаций может привести к разнообразию форм и стилей документации, что усложняет согласованность и сопровождение. Разработчики могут представлять информацию по-разному, что может затруднить обмен и понимание между разными участниками проекта.

Области применения неформальных нотаций в документировании программных систем:

*1. Описания функциональности и требований:* Неформальные нотации могут использоваться для описания функциональности системы, требований к ней, пользовательских сценариев и других аспектов, связанных с функциональностью системы.

*2. Комментарии к коду:* Неформальные нотации часто применяются для написания комментариев к коду, которые помогают разработчикам понять его структуру, функциональность и особенности.

*3. Заметки и руководства:* Неформальные нотации могут быть использованы для создания заметок, руководств, рекомендаций и других документов, которые помогают разработчикам и пользователям понять и использовать систему.

**29. Документирование программных систем. Частично формальные и формальные нотации. Плюсы и минусы, области применения**

Документирование программных систем — это процесс создания и поддержки документации, которая описывает систему, ее компоненты, функциональность, атрибуты качества, требования, архитектуру и другую важную информацию. Характер архитектуры любой системы обусловливается предъявляемыми к ней требованиями; это утверждение справедливо и по отношению к документации системы — содержание документации зависит от предполагаемых вариантов ее применения. Документация ни при каких обстоятельствах не может быть универсальной. С одной стороны, она должна быть абстрактной и, следовательно, доступной для понимания новыми сотрудниками, но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа. Документирование программных систем может включать формальные, частично формальные и неформальные нотации. В данном случае рассмотрим частично формальные и формальные нотации, их плюсы и минусы, а также области их применения. Частично формальные и формальные нотации в документировании программных систем используют более строгие правила и синтаксис для представления информации.

Частично формальные нотации – нотация стандартизована, есть наборы графических примитивов и правила их использования; полная семантика и возможные трактовки элементов диаграмм не зафиксированы; анализ семантики упрощен по сравнению с неформальными нотациями. Примеры частично формальной нотации: UML, BPMN, ERD. Особенности: 1. Требуется изучение нотации всеми участниками процесса; 2. Как правило, требуется дополнительное ПО; 3. Фокусировка на определенных областях, атрибутах.

Формальные нотации – семантика полностью стандартизована, смысл диаграмм определяется однозначно; возможен формальный анализ как синтаксиса, так и семантики модели; специализированное по с возможностями анализа полноты описания и генерации программного кода на целевых платформах. Примеры формальной нотации: AADL, ER. Особенности: 1. Избыточно сложны; 2. Требуются глубокие знания языка описания; 3. ВСЕГДА требуется дополнительное ПО; 4. Больше подходят для технического описания аппаратных комплексов.

Плюсы частично формальных и формальных нотаций в документировании программных систем:

*1. Однозначность и точность:* Формальные нотации предоставляют ясное и однозначное определение элементов и отношений в системе. Они позволяют избежать неоднозначностей и снижают вероятность ошибок в понимании требований и других аспектов системы.

*2. Автоматическая проверка:* Формальные нотации могут быть подвергнуты автоматической проверке на соответствие заданным правилам и условиям. Это позволяет обнаружить противоречия, ошибки и несоответствия в документации, что способствует повышению качества системы.

*3. Формальная верификация и анализ:* Формальные нотации могут быть использованы для формальной верификации системы, то есть математического доказательства ее корректности или выполнения определенных свойств. Они также могут быть использованы для проведения анализа производительности, безопасности и других атрибутов качества системы.

Минусы частично формальных и формальных нотаций в документировании программных систем:

*1. Сложность и изучение:* Формальные нотации требуют более высокого уровня знаний и понимания для их использования и интерпретации. Изучение и применение формальных нотаций может быть сложным и требовать времени и усилий.

*2. Ограничения и ограниченная выразительность:* Некоторые формальные нотации могут иметь ограничения в выразительности и покрываемых аспектах системы. Они могут быть сфокусированы на конкретных аспектах, таких как модель данных или верификация, и не обладать полной покрытием всех аспектов системы.

Области применения частично формальных и формальных нотаций в документировании программных систем:

*1. Требования и анализ:* Формальные нотации могут быть использованы для формализации требований к системе и их анализа. Это позволяет уточнить требования, обнаружить противоречия и обеспечить более точное понимание системы.

*2. Архитектура и моделирование:* Формальные нотации могут быть использованы для описания архитектуры системы и ее компонентов. Они позволяют формализовать структуру системы и ее взаимодействия, что упрощает анализ, верификацию и сопровождение системы.

*3. Верификация и анализ:* Формальные нотации широко используются для формальной верификации системы и анализа ее свойств, таких как безопасность, производительность и надежность. Они позволяют проводить формальные доказательства и анализировать систему на соответствие заданным спецификациям.

**30. Нотации. Рассмотреть пример одной из нотаций на выбор. Основные типы диаграмм, области применения**

Нотации в программировании и системном анализе представляют собой символы, символьные системы или графические элементы, используемые для представления и описания различных аспектов программных систем. Нотации являются визуальным языком для коммуникации и передачи информации между участниками проекта.

Одной из наиболее распространенных нотаций является Unified Modeling Language (UML), который предоставляет набор графических элементов для создания различных типов диаграмм. Данный вид диаграмм относится к частично формальным нотациям. У UML есть два типа диаграмм – структурированные (Structural Diagrams) и поведенческие (Behavioral Diagrams).

Structural Diagrams включают в себя: Composite Structure Diagrams, Deployment Diagrams, Package Diagram, Profile Diagram, Class Diagram, Object Diagram, Component Diagram. Behavioral Diagrams включают в себя: State Machine Diagram, Communication Diagram, UseCase Diagram, Activity Diagram, Sequence Diagram, Timing Diagram, Interaction Overview Diagram.

Рассмотрим основные типы диаграмм UML и их области применения:

*1. Диаграмма классов (Class Diagram):* Используется для моделирования структуры классов в системе, их атрибутов, методов и отношений между классами. Диаграмма классов широко применяется при проектировании и анализе систем для описания структуры объектно-ориентированных программ.

*2. Диаграмма вариантов использования (Use Case Diagram):* Показывает набор действий которое система должна или может осуществлять взаимодействия с внешними пользователями и системами. Каждый сценарий UseCase должен нести внешнему потребителю наблюдаемый результат. Оперирует сценариями, акторами, ассоциациями.

*3. Диаграмма последовательности (Sequence Diagram):* Используется для описания взаимодействия объектов в системе в рамках конкретного сценария или функциональности. Диаграмма последовательности показывает временную последовательность сообщений между объектами и помогает визуализировать поведение системы.

*4. Диаграмма состояний (State Machine Diagram):* Показывает состояние элементов и переходы между ними. Детально описывает поведение системы. Дает руководство к реализации. Плохо подходит для описания взаимодействия между компонентами.

*5. Диаграмма компонентов (Component Diagram):* Показывает компоненты, порты, предоставляемые и требуемые интерфейсы и отношения между ними. Оперирует компонентами, интерфейсами, портами, коннекторами, зависимостями, использованием.

*6. Диаграмма развертывания (Deployment Diagram):* Используется для моделирования физической инфраструктуры и развертывания компонентов системы на аппаратном обеспечении. Диаграмма развертывания помогает планировать и визуализировать физическую конфигурацию системы.

*7. Диаграмма пакетов (Package Diagram):* Показывает структуру системы на уровне пакетов. Оперирует пакетами и зависимостями.

*8. Диаграмма активностей (Activity Diagram):* Описывает workflow и бизнес процессы. Описывает параллельные и условные потоки выполнения. Хороши для коммуникации с нетехническими заинтересованными сторонами. Детально показывает взаимодействие частей системы. Показывает временные зависимости. Не подходит для описания комплексного поведения системы.

Области применения диаграмм UML и других нотаций включают разработку программного обеспечения, анализ и проектирование систем, документирование и коммуникацию требований, моделирование и анализ бизнес-процессов, архитектурное проектирование и т.д. Диаграммы и нотации помогают участникам проекта лучше понимать и визуализировать различные аспекты программных систем, улучшая коммуникацию и снижая риски ошибок в разработке.

**31. Стили и шаблоны проектирования. Понятие архитектурного стиля и архитектурного шаблона**

Стили и шаблоны проектирования являются важными концепциями в области разработки программного обеспечения. Они представляют собой рекомендации, принципы и решения, которые помогают разработчикам создавать высококачественные и эффективные программные системы.

Архитектурный стиль — это набор принципов и ограничений, определяющих организацию компонентов, их взаимодействие и структуру программной системы. Архитектурный стиль обычно описывает общую концепцию системы и служит основой для принятия решений о структуре и дизайне.

Изображение выглядит как текст, Шрифт, снимок экрана, Прямоугольник

Автоматически созданное описание

Есть два уровня архитектурного стиля: уровень приложения (МОНОЛИТ, SOA, Микросервисы, Event-driven) и уровень компонент (слоистая архитектура, модульная архитектура, архитектура, архитектура каналов сообщений).

Архитектурный шаблон выражает фундаментальную структурную схему организации программных систем. Он предоставляет набор предопределенных подсистем, определяет их обязанности и включает правила и рекомендации по организации взаимосвязей между ними.

Изображение выглядит как текст, Шрифт, Самоклеющийся листок, снимок экрана

Автоматически созданное описание

Архитектурные шаблоны включают в себя: CRUD, CQRS, Integration Patterns, Fault tolerance patterns.

Основная идея стилей и шаблонов проектирования состоит в том, чтобы использовать уже существующие архитектурные решения и передовые практики для создания новых систем. Это позволяет повысить качество, модульность, переиспользуемость и сопровождаемость программного обеспечения.

Примеры архитектурных стилей включают:

*1. Клиент-серверный стиль:* Система разделена на клиентскую часть, которая обрабатывает пользовательский интерфейс, и серверную часть, которая предоставляет доступ к данным и бизнес-логике.

*2. N-уровневая архитектура:* Система разделена на несколько уровней, таких как представление, бизнес-логика и доступ к данным. Каждый уровень выполняет свою специфическую функцию и взаимодействует с соседними уровнями.

*3. Потоковый стиль:* Система организована в виде потоков обработки данных, где данные передаются от одного компонента к другому для обработки.

Примеры архитектурных шаблонов включают:

*1. MVC (Model-View-Controller):* Разделяет систему на модели данных, представления пользовательского интерфейса и контроллеры для управления взаимодействием между моделями и представлениями.

*2. Singleton:* Обеспечивает существование только одного экземпляра класса и предоставляет глобальную точку доступа к этому экземпляру.

*3. Фабричный метод (Factory Method):* Определяет интерфейс для создания объектов, но делегирует фактическую реализацию подклассам, позволяя изменять тип создаваемых объектов без изменения кода, использующего их.

**32. Архитектурные принципы проектирования. Основные принципы и подробно на примере 1 – 2 выбранных**

Архитектурные принципы проектирования представляют собой общие руководящие принципы, которые помогают разработчикам создавать гибкие, расширяемые и устойчивые программные системы. Эти принципы служат основой для принятия решений о структуре, организации и взаимодействии компонентов системы. Основные принципы:

1. *Принцип единственной ответственности:* Модуль, компонент, объект выполняет минимально возможную функциональность, обеспечивающую решение дискретной задачи.
2. *Принцип наименьшего знания:* Модуль, компонент, объект не знает о внутренней реализации других модулей, компонентов, объектов с которыми взаимодействует.
3. *Принцип слабой связности:* Связи модулей, совместно составляющих систему минимальны.
4. *Принцип сильного зацепления:* Внутренние компоненты модуля работают совместно для решения общих задач и сильно «сцеплены» друг с другом.
5. *Принцип инверсии зависимостей:* Зависимости компонент направлены в сторону абстракции а не в сторону деталей реализации

Рассмотрим два архитектурных принципа:

*1. Принцип единственной ответственности (Single Responsibility Principle - SRP):* Согласно этому принципу, каждый класс или модуль должен быть ответственен только за одну конкретную обязанность или функцию. Это означает, что класс или модуль должен иметь только одну причину для изменения. Соблюдение принципа SRP облегчает понимание, тестирование, поддержку и изменение кода, а также способствует повторному использованию.

Пример: Предположим, у нас есть класс "User" в системе управления пользователями. Принцип SRP гласит, что класс "User" должен отвечать только за функциональность, связанную с управлением пользователями, такую как регистрация, аутентификация, доступ к данным пользователя и т.д. Он не должен содержать логику, относящуюся к другим аспектам системы, например, отправку электронной почты или взаимодействие с внешними сервисами. Вместо этого, такая функциональность должна быть вынесена в отдельные классы с соответствующей ответственностью.

*2. Принцип слабой связности (COUPLING):* Программный модуль может иметь любую степень детализации: методы в классе, классы, подсистемы, пакеты, модули, сервисы и т.д. Модуль с высокой (или сильной) связью зависит от многих других модулей.

Некоторые примеры типов связности: Классы могут взаимно получать доступ к (частным) данным друг друга. Это очень сильная форма связи, поскольку вы больше не можете изменять один из классов, не рассматривая другой. Более слабая связь присутствует, если классы взаимодействуют через глобальную структуру данных. Прямые зависимости между классами освобождаются и передаются на аутсорсинг глобальной структуре данных. Несмотря на это, связь по-прежнему очень сильна: все изменения, которые влияют на глобальные данные, также влияют на все классы, которые работают с данными. Если классы взаимодействуют только через параметры метода, связь значительно ниже: задействованные методы содержат только необходимые данные. Таким образом, изменения в этой связке данных вызывают только локальные изменения в соответствующих методах задействованных классов.

**34. Архитектурные стили на уровне компонентов. Слоистая и модульная архитектура**

Архитектурный стиль — это набор принципов и ограничений, определяющих организацию компонентов, их взаимодействие и структуру программной системы. Архитектурный стиль обычно описывает общую концепцию системы и служит основой для принятия решений о структуре и дизайне. Есть два уровня архитектурного стиля: уровень приложения (МОНОЛИТ, SOA, Микросервисы, Event-driven) и уровень компонент (слоистая архитектура, модульная архитектура, архитектура, архитектура каналов сообщений). Рассмотрим слоистую и модульную архитектуры.

*1. Слоистая архитектура (Layered Architecture):*

Может быть логической и физической. Структура системы организована в набор слоев. Каждый слой укладывается поверх другого слоя. Четко определенные интерфейсы между слоями. Снижает сложность, улучшает модульность, возможность повторного использования, ремонтопригодность. Различные критерии для расслоения: в первую очередь абстракция

Классический пример слоистой архитектуры - веб-приложения с тремя основными слоями: представление (presentation layer), бизнес-логика (business logic layer) и слой доступа к данным (data access layer). Каждый слой отвечает за свои специфические задачи, такие как обработка пользовательского ввода, логика приложения и работа с базой данных соответственно.

*2. Модульная архитектура (Modular Architecture):*

Модульность -> более обслуживаемые и настраиваемые системы. Приложения должны быть расширяемыми без простоев. Модуль — это независимая единица развертывания. У него есть имя и версия, и он объявляет все необходимые зависимости. Для других модулей показан определенный интерфейс, детали реализации скрыты. Модульность времени разработки и времени выполнения - Одним из примеров модульной архитектуры является модульная система на языке Java. Здесь каждый модуль представляет собой отдельную единицу кода с явно определенными интерфейсами и зависимостями. Модули могут быть разработаны и собраны отдельно, а затем использоваться в различных приложениях.

Оба стиля - слоистая и модульная архитектуры - имеют свои преимущества и области применения. Слоистая архитектура обеспечивает хорошую модульность и отделение ответственностей, позволяя легко изменять и заменять компоненты на каждом слое. Она особенно полезна в больших системах, где требуется четкая организация функциональности. С другой стороны, модульная архитектура обеспечивает высокую переиспользуемость и гибкость, позволяя компонентам быть независимыми и легко собираемыми в различные конфигурации.

**33. Архитектурные стили. Уровни применения, типичные представителе (на примере выбранного - монолит, SOA, Микросервисы...)**

Архитектурные стили представляют собой общие подходы к организации компонентов и взаимодействию между ними в программных системах. Они определяют структуру системы, распределение функций и ответственностей, а также способы коммуникации между компонентами. В зависимости от требований и характеристик проекта, разработчики могут выбирать различные архитектурные стили. Архитектурный стиль — это набор принципов и ограничений, определяющих организацию компонентов, их взаимодействие и структуру программной системы. Архитектурный стиль обычно описывает общую концепцию системы и служит основой для принятия решений о структуре и дизайне.

Изображение выглядит как текст, Шрифт, снимок экрана, Прямоугольник

Автоматически созданное описание

Есть два уровня архитектурного стиля: уровень приложения (МОНОЛИТ, SOA, Микросервисы, Event-driven) и уровень компонент (слоистая архитектура, модульная архитектура, архитектура, архитектура каналов сообщений).

Приведем типичный архитектурный стиль с описанием их уровней применения и представителей:

*Event-driven архитектура:*

Максимально слабо связанная, состоит из процессинговых компонент, которые асинхронно получают, обрабатывают и (или) выставляют новые сообщения. Основная цель такой архитектуры – реализация высоконагруженных распределенных приложений. Две основные топологии: 1. Mediator когда нужна центральная оркестрация связанных событий. 2. Broker когда централизованная оркестрация не нужна.

Топология с медиатором полезна для событий, которые состоят из нескольких этапов и требуют определенного уровня организации для обработки события. В этой топологии есть четыре главных типа компонент: event queues, an event mediator, event channels, event processors. Обработка начинается, когда клиент посылает сообщение в event queue, которая перенаправляет сообщение в медиатор. Еvent mediator получает начальное сообщение и обрабатывает его, посылая асинхронно сообщения в event channels для выполнения каждого шага процесса. Event processor, слушает свой канал обрабатывая поступающие в него сообщения. The event-mediator отвечает за оркестрацию процесса обработки начального события. Event channels используются для доставки сообщений медиатора процессорам. The event channels can be either message queues or message topics. The event processor реализует логику обработки события. Эта топология полезна, когда у вас относительно простой поток обработки событий и вы не хотите (или нуждаетесь в) централизованной организации событий. Каналами событий, содержащимися в компоненте брокера, могут быть очереди сообщений, темы сообщений или комбинация того и другого. Не существует центрального компонента-посредника событий, контролирующего и организующего начальное событие; скорее, каждый компонент-обработчик событий отвечает за обработку события и публикацию нового события, указывающего на действие, которое оно только что выполнило.

За: Слабосвязанная архитектура; Легкое масштабирование и обеспечение любого уровня производительности; Хорошо с точки зрения поставки

Против: Сложная реализация; Распределенные компоненты (event processors); Нужно думать о обеспечении транзакций и sync/async обработке; Обеспечение отказоустойчивости для каждого компонента; Управление контрактами сообщений (input and output); Сложное тестирование и особенно отладка.

**35. Паттерны проектирования. Общие понятия, назначение, плюсы и минусы.**

Любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем такого решения, которое можно использовать потом множество раз

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Назначение паттернов проектирования заключается в том, чтобы предоставить стандартизированные решения для часто встречающихся проблем в проектировании ПО.

Есть три основные категории паттернов проектирования: порождающие паттерны (Creational patterns), структурные паттерны (Structural patterns), поведенческие паттерны (Behavioral patterns). Каждый паттерн решает определенную проблему в проектировании программного обеспечения и имеет свои специфические преимущества и контексты применения.

В общем случае паттерн состоит из четырех основных элементов:

*1. Имя.* Сославшись на имя, мы можем сразу описать проблему проектирования, ее решения и их последствия. Именованные паттерны позволяют проводить проектирование на более высоком уровне абстракции

*2. Задача.* Описание того, когда следует применять паттерн. Необходимо сформулировать задачу и ее контекст. Может описываться конкретная проблема проектирования, иногда отмечается какие структуры классов свидетельствуют о негибком решении

*3. Решение.* Описание элементов дизайна, отношений между ними и функций каждого элемента. Имеется в виду не конкретный дизайн и реализация, а абстрактное описание задачи и возможного решения с помощью обобщенного сочетания элементов

*4. Результаты* – следствия применения паттерна и различные компромиссы. При описании проектного решения о последствиях часто не упоминают, но иметь о них представление необходимо, чтобы более обоснованно выбирать между разными вариантами решения

*Плюсы:*

• Проверенные решения. Вы тратите меньше времени, используя готовые решения, вместо повторного изобретения велосипеда. До некоторых решений вы смогли бы додуматься и сами, но многие могут быть для вас открытием.

• Стандартизация кода. Вы делаете меньше просчётов при проектировании, используя типовые унифицированные решения, так как все скрытые проблемы в них уже давно найдены.

• Общий программистский словарь. Вы произносите название паттерна, вместо того чтобы час объяснять другим программистам, какой крутой дизайн вы придумали и какие классы для этого нужны.

*Минусы:*

• Применимы для «классических» объектно-ориентированных языков: Нужда в паттернах появляется тогда, когда люди выбирают для своего проекта язык программирования с недостаточным уровнем абстракции. В этом случае паттерны — это костыль, который придаёт этому языку суперспособности. Многие современные языки содержат возможности паттернов «из коробки»

• Неэффективные решения: Паттерны пытаются стандартизировать подходы, которые и так уже широко используются. Эта стандартизация кажется некоторым людям догмой, и они реализуют паттерны «как в книжке», не приспосабливая паттерны к реалиям проекта.

• Неоправданные решения: «Если у тебя в руках молоток, то все предметы вокруг начинают напоминать гвозди». Похожая проблема возникает у новичков, которые только-только познакомились с паттернами. Вникнув в паттерны, человек пытается применить свои знания везде. Даже там, где можно было бы обойтись кодом попроще.

**36. Паттерны проектирования. Порождающие паттерны (рассмотреть минимум два паттерна и сравнить их)**

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Порождающие паттерны проектирования (Creational patterns) занимаются созданием объектов и обеспечивают гибкость в создании и инициализации объектов. Они помогают сделать процесс создания объектов более гибким, удобным и расширяемым, скрывая конкретные детали их создания. Некоторые популярные порождающие паттерны включают Фабричный метод (Factory Method), Абстрактную фабрику (Abstract Factory), Строитель (Builder), Прототип (Prototype) и Одиночку (Singleton). Каждый из них имеет свои специфические особенности и контексты применения, и их выбор зависит от требований конкретного проекта.

Рассмотрим два порождающих паттерна проектирования - Фабричный метод (Factory Method) и Абстрактная фабрика (Abstract Factory) - и сравним их. Оба паттерна относятся к созданию объектов, но имеют некоторые различия в своей структуре и использовании.

Фабричный метод определяет интерфейс для создания объектов, но оставляет подклассам решение о том, какой класс конкретно создавать. Таким образом, он делегирует создание объектов подклассам. В центре Фабричного метода находится абстрактный класс, содержащий метод-фабрику, который подклассы могут переопределить для создания конкретных объектов.

Абстрактная фабрика предоставляет интерфейс для создания семейств взаимосвязанных объектов без указания их конкретных классов. Он позволяет создавать объекты семейства связанных продуктов, а не отдельные объекты. Абстрактная фабрика определяет интерфейс, содержащий методы для создания каждого из связанных продуктов. Затем создаются конкретные реализации абстрактной фабрики для каждого семейства продуктов.

*Сравнение:*

• Фабричный метод создает одиночные объекты, в то время как Абстрактная фабрика создает семейства взаимосвязанных объектов.

• Фабричный метод делегирует создание объектов подклассам, тогда как Абстрактная фабрика определяет семейства продуктов, и каждая конкретная реализация фабрики создает соответствующий продукт.

• Фабричный метод работает с одним уровнем абстракции, тогда как Абстрактная фабрика работает с несколькими уровнями абстракции (создание семейства связанных объектов).

• Фабричный метод может быть реализован в виде метода в абстрактном классе, в то время как Абстрактная фабрика требует создания интерфейса с методами для создания продуктов.

Выбор паттерна зависит от конкретных требований проекта и структуры объектов, которые необходимо создать. Фабричный метод применяется, когда необходимо делегировать создание объектов подклассам, а Абстрактная фабрика используется для создания семейств взаимосвязанных объектов. Фабричный метод более гибок в добавлении новых продуктов, но усложняет код создания объектов, тогда как Абстрактная фабрика гарантирует совместимость объектов, но может быть менее гибкой при добавлении новых продуктов.

**37. Паттерны проектирования. Структурные паттерны (рассмотреть минимум два паттерна и сравнить их)**

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Структурные паттерны проектирования обеспечивают способы организации классов и объектов для создания более гибкой и эффективной структуры программы. Они фокусируются на компоновке объектов и классов в более крупные структуры, помогают определить отношения между объектами и упрощают взаимодействие между ними. Структурные паттерны помогают сделать код более гибким, модульным и понятным. Они позволяют создавать хорошо организованные и расширяемые системы, что упрощает их сопровождение и развитие в будущем. Некоторые популярные структурные паттерны включают Адаптер (Adapter), Декоратор (Decorator), Компоновщик (Composite), Фасад (Facade), Заместитель (Proxy) и Мост (Bridge). Каждый из них имеет свои специфические особенности и контексты применения, и их выбор зависит от требований конкретного проекта.

Рассмотрим два структурных паттерна проектирования - Адаптер (Adapter) и Декоратор (Decorator) - и сравним их. Оба паттерна помогают изменять поведение объектов, но они решают разные задачи и имеют различную структуру.

Адаптер преобразует интерфейс одного класса в интерфейс другого класса, чтобы сделать их совместимыми. Он позволяет объектам работать вместе, несмотря на несовместимость их интерфейсов. Адаптер состоит из двух основных компонентов: адаптируемого класса и адаптера. Адаптер реализует целевой интерфейс и содержит экземпляр адаптируемого класса, который он использует для преобразования вызовов методов.

Декоратор добавляет новые функциональные возможности или изменяет поведение существующего объекта, не изменяя его базовой структуры. Он предоставляет гибкую альтернативу наследованию для расширения функциональности объектов. Декоратор состоит из абстрактного компонента, конкретного компонента, декоратора и конкретных декораторов. Абстрактный компонент определяет общий интерфейс для всех компонентов, включая декораторы. Конкретный компонент представляет базовый объект, к которому могут добавляться декораторы. Декораторы имеют такой же интерфейс, что и абстрактный компонент, и содержат ссылку на компонент, который они декорируют.

*Сравнение:*

• Адаптер используется для обеспечения совместимости между несовместимыми интерфейсами, тогда как Декоратор используется для добавления функциональности или изменения поведения объекта без изменения его базовой структуры.

• Адаптер работает на уровне интерфейсов, преобразуя вызовы методов, в то время как Декоратор работает на уровне объектов, оборачивая их в другие объекты и добавляя новую функциональность.

• Адаптер изменяет только интерфейс объекта, в то время как Декоратор добавляет новую функциональность или изменяет поведение объекта.

• Адаптер используется для обеспечения взаимодействия между различными компонентами системы, тогда как Декоратор используется для добавления функциональности к отдельным объектам.

Выбор между этими паттернами зависит от конкретной задачи и требований проекта. Адаптер полезен, когда необходимо связать несовместимые компоненты, а Декоратор - для динамического расширения функциональности объектов.

**42. Методологии управления качеством. Классификация**

Методологии, целью которых является успешное выполнение отдельного проекта. К этой группе относится большинство проектных методологий и практически все адаптивные методологии. Логика успешного функционирования организации выглядит как рост компетенций, создание технических активов за счет последовательного выполнения успешных проектов.

Методологии, обеспечивающие устойчивое функционирование компании-разработчика ПО, нацелены на последовательное развитие компетенций. К этой группе относятся модели зрелости (CMM, CMMI). Логика успешного функционирования предполагает создание, контроль и непрерывное улучшение способностей организации к выполнению проектов, и, как следствие, успешное выполнение проектов

*По характеру знаний:*

• Инженерные методологии (software engineering) основываются на технологических принципах и направлены на совершенствование конечных продуктов, таких как программный код, тестовые примеры, прототипы, документация. Инженерные инструментальные необходимы квалифицированному разработчику.

• Управленческие методологии (software project management and process management) основываются на принципах теории управления (менеджмента), в их основе лежат такие концепции, как тотальное управление качеством, управление проектами, управление знаниями.

• Интегрированные методологии средства объединяют инженерные концепции и концепции управления.

Прогнозируемые методологии основываются на предпосылке о возможности и целесообразности детального планирования будущего. Для ИТ-проекта формулируются требования к разрабатываемой системе, формируется план проекта и определяется потребность в ресурсах. Изменения в плане проекта и требованиях считаются нежелательными. Проектные методологии этого класса используют каскадную модель жизненного цикла.

Адаптивные методологии нацелены на преодоление ожидаемой неполноты требований и их постоянного изменения. В основе адаптивных методологий лежит итерационная модель жизненного цикла. Примером адаптивных методологий являются Scrum, Канбан, Crystal, Extreme Programming. Адаптивные методологии учитывают психологические особенности процесса разработки ПО. Одним из значимых факторов успеха использования адаптивных методологий является высокая квалификация специалистов, в первую очередь – разработчиков.

**40. Управление качеством. Стандарты и модели**

Управление качеством (Quality Management) — это процесс планирования, контроля и обеспечения качества продукта или услуги в рамках проекта или организации. Оно включает в себя установление стандартов качества, определение процессов и методов для достижения и поддержания требуемого уровня качества, а также мониторинг и улучшение процессов для обеспечения постоянного улучшения качества.

Основными государственными стандартами, регламентирующими использование терминологии по качеству программного обеспечения в Российской Федерации, являются:

• ГОСТ 28806-90 «КАЧЕСТВО ПРОГРАММНЫХ СРЕДСТВ. Термины и определения (Software quality. Terms and definitions)»

• ГОСТ 28195-89 «ОЦЕНКА КАЧЕСТВА ПРОГРАММНЫХ СРЕДСТВ. Общие положения (Quality control of software systems. General principles)»

• ГОСТ Р ИСО/МЭК 25000. «Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов»

Модель качества (quality model) – это определенное множество характеристик и взаимосвязей между ними, которые обеспечивают основу для определения требований к качеству и оценки качества (ИСО/МЭК 25000).

Характеристики качества ПС часто называют свойствами, атрибутами, показателями или факторами качества. Характеристика качества ПС является структурным многоуровневым объектом, и определяется как набор свойств, которые могут быть выражены числовыми и (реже) лингвистическими оценками. Согласно ГОСТ 28195-89 [2] характеристика качества декомпозируется на 4 уровня:

• Фактор качества определяет группу основных показателей качества, характеризующую потребительские свойства (требования)

• Критерий качества задает программно-ориентированные свойства (требования) ПС. Для вычисления значения критерия используют одну или несколько метрик.

• Метрика используется для количественной оценки качества ПО по заданному критерию. Она определяется как шкала и метод измерения свойства.

• Оценочный элемент (элемент показателя качества) измеряет заданное в метрике свойство числом от 0 до 1 (ГОСТ 28195-89)

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Модель качества ПС стандартов ISO/IEC 9126-2, 9126-3, 9126-4 различает следующие понятия:

• внутреннее качество, связанное с характеристиками ПС самого по себе, без учета его поведения;

• внешнее качество, характеризующее ПС с точки зрения его поведения;

• качество процесса разработки ПС;

• эксплуатационное качества ПС при использовании в различных контекстах – того качества, которое ощущается пользователями при конкретных сценариях работы ПО.

**39. Управление качеством. Основные понятия, процессы и практики**

Управление качеством (Quality Management) — это процесс планирования, контроля и обеспечения качества продукта или услуги в рамках проекта или организации. Оно включает в себя установление стандартов качества, определение процессов и методов для достижения и поддержания требуемого уровня качества, а также мониторинг и улучшение процессов для обеспечения постоянного улучшения качества.

*Основные аспекты управления качеством:*

1. Качество результата – как качество некоторого продукта, созданного в определенной среде в рассматриваемый период времени

2. Качество системы – среда в виде технических, информационных, организационных ресурсов и людей образует систему, в которой создается (программный) продукт

3. Качество процессов – совокупность изменений системы и производимых продуктов во времени образует процессы, анализ которых необходим для принятия решений по улучшению качества.

*Основными свойствами качества как для объектов, так и для процессов разработки являются:*

• Свойство измеримости. Одной из основных проблем управления качеством является выбор эффективных показателей. В любом случае, говоря качество может рассматриваться только через количественные показатели. Два подхода – комплексная оценка на основе показателей (простой) и разработка модели оценочных шкал в дополнение к показателям, для определения уровней качества.

• Свойство иерархичности. Сложное понятие качества следует декомпозировать на уровни иерархии.

• Свойство изменчивости. Качество — это динамическая категория. Качество создаваемых систем проектируется, устанавливается и реализуется, но при этом его показатели (целевые и фактические) меняются во времени, и процесс управления обязан это учитывать

*Процессы управления:*

• Подтверждение качества. — это процесс, в рамках которого осуществляется планирование, контроль и управление всеми деятельностями, направленными на обеспечение качества продукта или услуги.

• Проверка и аттестация. — это процессы, направленные на оценку и подтверждение соответствия продукта или услуги установленным стандартам, спецификациям и требованиям.

• Оценка и аудит. — это процессы, направленные на измерение, оценку и проверку процессов, систем, продуктов или услуг с целью определения их качества и соответствия установленным стандартам и требованиям.

*Практика применения:*

• Требования к качеству. Определение четких и измеримых требований к качеству продукта или услуги. Это может включать установление стандартов, спецификаций и метрик, по которым будет оцениваться качество.

• Характеристика дефектов. Идентификация и документирование дефектов и неполадок, которые возникают в процессе разработки или в конечном продукте. Характеристики дефектов помогают определить приоритеты по исправлению и улучшению качества продукта.

• Техника управления качеством. Применение методологий и инструментов для планирования, контроля и улучшения качества продукта или услуги. Это может включать управление конфигурацией, тестирование, код-ревью, аудит процессов и документации и другие практики.

• Количественная оценка качества. Использование метрик и измерений для количественной оценки качества продукта или услуги. Это может включать оценку производительности, надежности, безопасности, доступности и других характеристик продукта.

**38. Паттерны проектирования. Поведенческие паттерны (рассмотреть минимум два паттерна и сравнить их)**

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Поведенческие паттерны проектирования сосредоточены на организации взаимодействия и коммуникации между объектами, а также на управлении алгоритмами и обработкой различных поведенческих сценариев. Они помогают создавать более гибкие и расширяемые системы, где объекты могут взаимодействовать и менять свое поведение без прямой зависимости друг от друга. Некоторые популярные поведенческие паттерны включают Наблюдатель (Observer), Стратегия (Strategy), Команда (Command), Состояние (State), Цепочка обязанностей (Chain of Responsibility), Итератор (Iterator) и Шаблонный метод (Template Method). Каждый из них имеет свои специфические особенности и контексты применения, и выбор паттерна зависит от требований и целей проекта.

Рассмотрим два поведенческих паттерна проектирования - Наблюдатель (Observer) и Стратегия (Strategy) - и сравним их. Оба паттерна описывают способы организации взаимодействия объектов, но решают разные задачи и имеют различную структуру.

Наблюдатель определяет отношение "один ко многим" между объектами таким образом, что при изменении состояния одного объекта все зависимые от него объекты автоматически оповещаются и обновляются. В наблюдателе есть два основных компонента: субъект (subject) и наблюдатели (observers). Субъект имеет методы для добавления, удаления и уведомления наблюдателей. Наблюдатели определяют метод обновления, который вызывается субъектом при изменении его состояния.

Стратегия позволяет определить семейство алгоритмов, инкапсулировать каждый из них и делать их взаимозаменяемыми. Позволяет изменять алгоритмы независимо от клиентского кода, который их использует. В стратегии есть три основных компонента: контекст, стратегии и интерфейс стратегий. Контекст содержит ссылку на объект стратегии и имеет методы для выполнения алгоритма. Стратегии реализуют общий интерфейс и предоставляют конкретные реализации алгоритмов.

*Сравнение:*

• Наблюдатель используется для организации отношений "один-ко-многим" между объектами, тогда как Стратегия используется для предоставления взаимозаменяемых алгоритмов.

• Наблюдатель уведомляет зависимые объекты об изменениях в состоянии субъекта, тогда как Стратегия позволяет клиентам выбирать алгоритмы, независимо от контекста.

• Наблюдатель изменяет состояние объектов, в то время как Стратегия изменяет поведение объектов.

• Наблюдатель работает на основе паттерна "издатель-подписчик", в то время как Стратегия работает на основе паттерна "выделение алгоритмов".

Выбор паттерна зависит от специфических требований проекта и характера взаимодействия между объектами. Наблюдатель полезен, когда нужно обеспечить реакцию на изменение состояния объектов. Стратегия применяется, когда необходимо иметь возможность гибко выбирать и менять алгоритмы выполнения задачи.

**48. Процесс управления релизами. Выделите основные понятия и опишите процесс.**

**Процесс управления релизами (Release Management)** – это набор методов и процедур, которые позволяют управлять жизненным циклом программного обеспечения от разработки до выпуска в эксплуатацию.

*Основные понятия:*

1. Релиз: Релиз в программной разработке представляет собой выпуск новой версии программного продукта или изменения, вносимые в существующую версию. Релизы обычно включают в себя новые функции, исправления ошибок, улучшения производительности или другие изменения.

2. Управление релизами: Управление релизами — это процесс, осуществляемый командой разработки, который включает планирование, подготовку, тестирование, выпуск и отслеживание релизов программного продукта. Целью управления релизами является эффективное и контролируемое внедрение изменений в продукт.

*Процесс управления релизами:*

1. Планирование релизов: В этой фазе определяются цели релиза, функциональные требования, сроки и приоритеты. Команда разработки рассматривает функции, которые должны быть включены в релиз, а также определяет необходимые ресурсы, расписание и зависимости.

2. Разработка и тестирование: После завершения планирования команда разработки приступает к разработке новых функций или внесению изменений в программный продукт. Затем проводятся тесты, включая модульное, интеграционное и системное тестирование, чтобы обнаружить и исправить возможные ошибки.

3. Подготовка к выпуску: В этой фазе команда разработки готовит релиз к публикации. Включает в себя составление документации, подготовку инсталляционных пакетов, установку необходимых конфигураций и подготовку среды для развертывания релиза.

4. Выпуск и развертывание: После завершения подготовки команда выпускает релиз, загружая его на серверы или распространяя клиентам. Затем релиз разворачивается на целевых системах или предоставляется для загрузки.

5. Отслеживание и обратная связь: После выпуска релиза следует отслеживать его работу и собирать обратную связь от пользователей. Это позволяет выявить возможные проблемы, исправить ошибки и планировать будущие улучшения.

*Примеры процесса управления релизами:*

Команда разработки мобильного приложения работает над выпуском новой версии. Они планируют добавить несколько новых функций, исправить известные ошибки и улучшить производительность. После завершения разработки и тестирования они подготавливают документацию, создают инсталляционные пакеты и выпускают релиз на платформах App Store и Google Play.

**41. Управление качеством. Эталонная модель**

Управление качеством (Quality Management) — это процесс планирования, контроля и обеспечения качества продукта или услуги в рамках проекта или организации. Оно включает в себя установление стандартов качества, определение процессов и методов для достижения и поддержания требуемого уровня качества, а также мониторинг и улучшение процессов для обеспечения постоянного улучшения качества.

В стандарте ГОСТ Р ИСО/МЕС 25010-2015 приведена эталонная модель измерения качества ПС, изображенная на рисунке. В этой модели рассматриваются характеристики (факторы), суб-характеристки (критерии), показатели качества (метрики) и элементы показателей качества (оценочные элементы).

Изображение выглядит как диаграмма, линия, План, Технический чертеж

Автоматически созданное описание

ГОСТ Р ИСО/МЕС 25010-2015 ("Системы и программное обеспечение. Системные и программные продукты. Оценка качества и свойств") определяет эталонную модель качества для системных и программных продуктов. Эта модель предоставляет набор характеристик и подходов для оценки качества продукта. Она определяет следующие основные характеристики качества ПО:

*• Functional Suitability (Функциональная пригодность):* Оценивает, насколько ПО соответствует функциональным требованиям и удовлетворяет потребностям пользователей.

*• Performance Efficiency (Эффективность производительности):* Оценивает производительность ПО в отношении времени отклика, скорости обработки и использования ресурсов.

*• Compatibility (Совместимость):* Оценивает способность ПО работать вместе с другими системами, платформами и компонентами.

*• Usability (Удобство использования):* Оценивает, насколько ПО удобно для использования, понятно и привлекательно для пользователей.

*• Reliability (Надежность):* Оценивает способность ПО выполнять требуемые функции с заданной надежностью и стабильностью.

*• Security (Безопасность):* Оценивает защищенность ПО от несанкционированного доступа, атак и утечек данных.

*• Maintainability (Поддерживаемость):* Оценивает возможность поддержки и изменения ПО с минимальными затратами на усилия и ресурсы.

*• Portability (Переносимость):* Оценивает способность ПО работать на различных платформах и в различных окружениях без потери функциональности.

Эталонная модель качества по ГОСТ Р ИСО/МЭК 25010-2015 используется для оценки качества системных и программных продуктов, а также для планирования, анализа и улучшения качества продукта в соответствии с международными стандартами.

**43. Рефакторинг. Назначение и принципы, применяемые подходы.**

Рефакторинг (Refactoring): изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения

Единственная цель рефакторинга – упростить понимание и модификацию программного обеспечения, и его следует четко отделять от других схожих манипуляций – например, оптимизации, когда наблюдаемое поведение программы также не меняется (кроме скорости работы).

*Назначение:*

• Рефакторинг улучшает композицию программного обеспечения

• Рефакторинг облегчает понимание программы

• Рефакторинг помогает найти ошибки

• Рефакторинг позволяет быстрее писать программы

Лучшей стратегией является постоянный рефакторинг, когда изменения небольшие, но вносятся сразу по мере необходимости. Конечно, для такого стиля работы необходима автоматизация тестирования.

*Общие правила:*

• Применяйте рефакторинг при добавлении функции

• Применяйте рефакторинг при разборе кода

• Применяйте рефакторинг, если требуется исправить ошибку

*Есть три случая, в которых работы по рефакторингу противопоказаны:*

1. Код настолько плох что лучшим решением является переписывание с нуля. Это нелегкое решение, но иногда его нужно принимать.

2. Код не работает должным образом. Если ошибок так много, что стабилизировать выполнение программы не удается. Перед началом рефакторинга код должен быть работоспособным. Компромиссное решение – рефакторинг состоящий в разделении системы на сильно инкапсулированные компоненты и принятие решения о замене только части компонентов

3. Проект близок к завершению. Рост производительности после рефакторинга не успеет «окупиться» до завершения проекта. В то же время успешный проект обязательно будет развиваться и результатами рефакторинга можно будет воспользоваться на следующей итерации разработки.

*Кандидаты на рефакторинг:*

• Раздуватели (Bloaters) – модули, классы, переменные, которые стали слишком большими и мешают эффективной работе.

• Нарушители OOD (Object-Orientation Abusers) – классы и модули, демонстрирующие неполное или неправильное применение принципов объектно-ориентированного проектирования

• Блокировщики изменений (Change Preventers) – ситуации, когда внесение любого изменения в код порождает дополнительные (массовые) изменения

• Пустой код (Dispensables) – когда вы видите что-то бесполезное и лишнее, от чего можно было бы полностью избавиться, сделав код чище, эффективней и проще для понимания

• Опутыватели (Couplers) – практики кодирования, которые приводят к избыточной связанности между классами, либо показывают, что бывает, если тесная связанность заменяется постоянным делегированием

• Неполнота библиотечного класса (Incomplete Library Class). Не относится ни к какой группе. В случае, когда в библиотечном классе не хватает функционала, естественное решение — внести изменения в библиотеку — очень часто оказывается недоступным, так как библиотека закрыта для записи

Рефакторинг дает возможность создания более простых, но при этом не менее гибких проектов. Процесс проектирования становится более легким, часто возможно применение наиболее простых решений с возможностью их замены в будущем (и только при необходимости такой замены).

**44. Рефакторинг. Техники рефакторинга (на примере одной из техник)**

Рефакторинг (Refactoring): изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения

*Техники рефакторинга:*

• Композиция методов (см. ниже)

• Перераспределение ответственности (Moving Features between Objects). Если вы разместили функциональность по классам не самым удачным образом — это еще не повод отчаиваться. Рефакторинги этой группы показывают как безопасно перемещать функциональность из одних классов в другие, создавать новые классы, а также скрывать детали реализации из публичного доступа.

• Организация данных

• Упрощение условных выражений (см. ниже)

• Упрощение вызова метода (Simplifying Method Calls). Эти рефакторинги делают вызовы методов проще и яснее для понимания. Это, в свою очередь, упрощает интерфейсы взаимодействия между классами.

• Обобщения (Dealing With Generalization). Обобщение порождает собственную группу рефакторингов, в основном связанных с перемещением функциональности по иерархии наследования классов, создания новых классов и интерфейсов, а также замены наследования делегированием и наоборот.

Композиция методов (Composing Methods). Значительная часть рефакторинга посвящается правильному составлению методов. В большинстве случаев проблема — это слишком длинные методы. Зависимости внутри такого метода, прячут логику выполнения и делают метод крайне сложным для понимания, а значит и изменнения. Рефакторинги этой группы призваны уменьшить сложность внутри метода, убрать дублирование кода и облегчить последующую работу с ним

• Извлечение метода (Extract Method)

Одной из распространенных техник рефакторинга является "Извлечение метода" (Extract Method). Эта техника используется для выделения повторяющегося фрагмента кода в отдельный метод, чтобы улучшить его читаемость, поддерживаемость и повторное использование.

• Встраивание метода (Inline Method)

• Извлечение переменной (Extract Variable)

• Встраивание переменной (Inline Temp)

• Замена переменной вызовом метода (Replace Temp with Query)

• Расщепление переменной (Split Temporary Variable)

• Удаление присваиваний параметрам (Remove Assignments to Parameters)

• Замена метода объектом методов (Replace Method with Method Object)

• Замена алгоритма (Substitute Algorithm)

Упрощение условных выражений (Simplifying Conditional Expressions). Логика условного выполнения имеет тенденцию становиться сложной, поэтому ряд рефакторингов направлен на то, чтобы упростить ее.

• Объединение условных операторов (Consolidate Conditional Expression)

• Объединение дублирующихся фрагментов в условных операторах (Consolidate Duplicate Conditional Fragments)

• Разбиение условного оператора (Decompose Conditional)

• Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism)

• Удаление управляющего флага (Remove Control Flag)

• Замена вложенных условных операторов граничным оператором (Replace Nested Conditional with Guard Clauses)

• Введение Null-объекта (Introduce Null Object)

• Введение проверки утверждения (Introduce Assertion)

**45. Метрики программного кода. Основные принципы и подходы, инструменты, практика применения.**

Метрика — это количественная мера позволяющая оценить, в какой степени система, компонент системы или процесс обладают заданным качеством.

*Основные цели применения метрик программного кода:*

1. Измерение качества продукта либо процесса. 2. Прогнозирование качества продукта либо процесса. 3. Повышение качества продукта либо процесса.

*Метрики сложности программ принято разделять на основные группы:*

1. Количественные метрики

2. Метрики сложности потока управления программы

3. Метрики сложности потока управления данными

4. Метрики объектно-ориентированных систем

В общем случае применение метрик позволяет руководителям проектов и предприятий изучить сложность разработанного или даже разрабатываемого проекта, оценить объем работ, стилистику разрабатываемой программы и усилия, потраченные каждым разработчиком для реализации того или иного решения. В частности, применение метрик позволяет решать такие задачи как:

• Оценка производительности применения новых средств и методов

• Определение тенденций производительности с течением времени

• Оценка стоимости и графика будущих проектов

• Прогноз будущих потребностей в персонале

• Контроль стабильности качества ПО

• Прогноз стоимости владения (поддержки и развития) ПО

*Количественные метрики (они же Размерно-ориентированные):*

• LOC-оценка (Lines Of Code) — это самая простая и старая методика, основанная на подсчете строк кода (физических – SLOC или логических DSI). Показатели, по смыслу близкие к LOC-оценке и объединенные в одно «семейство»: количество пустых строк, количество комментариев, процент комментариев и др. Эти метрики чувствительны к используемому языку программирования, что накладывает ограничения на их применение.

• Метрики Хольстеда (Haulsted): словарь программы, длина программы, объем программы, уровень качества программирования, сложность понимания программы и др. При применении метрик Холстеда частично компенсируются недостатки, связанные с возможностью записи одной и той же функциональности разным количеством строк и операторов.

*Метрики сложности потока управления программы:*

Следующий большой класс метрик, основанный уже не на количественных показателях, а на анализе управляющего графа программы, называется метрики сложности потока управления программ.

• Самой распространенной оценкой, основанной на анализе графа управления, является цикломатическая сложность программы. Цикломатическая сложность = V(G) = e — n + 2p, где e — количество дуг, n — количество вершин, p — число компонент связности.

• метод Хансена. Мера сложности программы в данном случае представляется в виде пары . Преимуществом данной меры является ее чувствительность к структурированности ПО.

• Метрика Пивоварского. Модификация меры цикломатической сложности. Позволяет отслеживать различия не только между последовательными и вложенными управляющими конструкциями, но и между структурированными и неструктурированными программами.

• Мера Вудворда — количество пересечений дуг управляющего графа. Так как в хорошо структурированной программе таких ситуаций возникать не должно, то данная метрика применяется в основном в слабо структурированных языках (ассемблер)

Метрики сложности потока управления данными:

Метрика Чепина: суть метода состоит в оценке информационной прочности отдельно взятого программного модуля с помощью анализа характера использования переменных из списка ввода-вывода.

Метрики объектно-ориентированных программных систем: Объектно-ориентированные метрики вводятся с целью, улучшить понимание качества продукта, оценить эффективность процесса конструирования, улучшить качество работы на этапе проектирования.

*Набор метрик Чидамбера и Кемерера:*

• Взвешенные методы на класс WMC (Weighted Methods Per Class) учитывает комплексный показатель на основе количества методов в классе и их сложности

• Высота дерева наследования DIT (Depth of Inheritance Tree) максимальная длинна цепочки наследования классов. Рекомендация не более 7

• Количество детей NOC (Number of children) количество непосредственных потомков одного класса. Рекомендация не более 7 (9)

• Сцепление между классами объектов СВО (Coupling between object classes): Количество классов, о которых «знает» данный класс

*Метрики Лоренца и Кидда:* Метрики, ориентированные на классы:

• Размер класса CS (Class Size) – число операций (методов) класса. CS <=20

• Количество операций, переопределяемых подклассом, NOO (Number of Operations Overridden by a Subclass) – число унаследованных операций, переопределяемых классом. NOO <=3

• Количество операций, добавленных подклассом, NOA (Number of Operations Added by a Subclass) – число новых методов, вводимых подклассом. NOA <= 4 для листового класса

• Индекс специализации SI (Specialization Index) – насколько методы класса НЕ используют совместно атрибуты

*Операционно-ориентированные метрики:*

• Средний размер операции OSAVG (Average Operation Size) – количество сообщений, посылаемых операцией. <= 25

• Сложность операции ОС (Operation Complexityчисло унаследованных операций, переопределяемых классом. NOO <=3

• Среднее количество параметров на операцию NPAVG (Average Number of Parameters per operation)

**46. Мониторинг систем. Основные решаемые задачи, принципы организации, типы комплексных систем мониторинга.**

Мониторинг ИТ систем является составной частью управления информационной инфраструктурой предприятия, заключающейся в постоянном наблюдении и периодическом анализе ИТ объектов с отслеживанием динамики происходящих с ними изменений. Ключевой задачей систем мониторинга ИТ является получение, сохранение и анализ информации о состоянии подконтрольных элементов ИТ структуры компании. Как правило, выделяется два уровня мониторинга: Мониторинг инфраструктуры и Мониторинг сервисов.

Внедрение комплексной системы мониторинга ИТ помогает предприятию:

* снизить время простоя компонентов ИТ структуры;
* увеличить доступность программ для бизнеса;
* осуществлять проактивный анализ неполадок;
* повысить уровень производительности использования информационных ресурсов.

Внедрение автоматизированной системы мониторинга ИТ и контроль работы информационной инфраструктуры способно повысить уровень качества ее функционирования с помощью быстрого выявления и ликвидации сбоев и неполадок, а также предотвращения их возникновения в будущем, в первую очередь, для наиболее критичных для бизнеса компании сервисов. Внедрение комплексной системы мониторинга позволяет:

* своевременно фиксировать возникновение проблем в работе компонентов ИТ структуры;
* обнаруживать место и характер неполадки;
* определять влияние возникшей проблемы на возможность предоставления ИТ сервисов (это необходимо для верной расстановки приоритетов в работах по ликвидации сбоев;
* проактивно следить за изменениями в функционировании инфраструктуры;
* предотвращать вероятные сбои.

Основные принципы организации системы мониторинга:

* Лучше использовать стандартные решения (Nagios, Zabbix и т.п.). Собственная система — это долго и дорого. Лучше потратить время на изучение какой-то стандартной системы и научиться писать к ней плагины, чем разрабатывать свою. Самописные системы тяжело развивать, особенно новым сотрудникам, не принимавшим участие в её создании.
* Система мониторинга не имеет смысла без организации дежурных смен (и/или мгновенных уведомлений администратору) для устранения проблем.
* Постоянная подстройка порогов реакции. Тысячи уведомлений будут бесполезны, правило должно быть таким: пришло уведомление — значит серьёзная проблема или авария.
* Автоматизация типовых реакций. Все что можно автоматизировать и нужно автоматизировать, в первую очередь некритичные рутинные операции.
* Нужна распределённая система мониторинига, которая может мониторить и саму себя. Если сама система мониторинга отвалится, уведомлений нет, вы считаете, что всё хорошо. А на самом деле проект - "лежит". Либо две системы мониторинга: одна мониторит одну часть проекта, вторая - другую, и каждая из них мониторит другую систему мониторинга.

**47. Процессы тестирования ПО. Опишите основные способы деления видов тестирования.**

Процессы тестирования программного обеспечения (ПО) играют важную роль в обеспечении качества и надежности системы. Основная цель тестирования состоит в проверке функциональности, соответствия требованиям, выявлении ошибок и улучшении качества ПО. Виды тестирования могут быть классифицированы по различным критериям, включая уровень абстракции, цель тестирования и способ выполнения. Вот основные способы деления видов тестирования:

*1. По уровню абстракции:*

- Юнит-тестирование (Unit Testing): Тестирование отдельных компонентов или модулей программы, обычно на уровне функций или методов.

- Интеграционное тестирование (Integration Testing): Тестирование взаимодействия между компонентами или модулями системы.

- Системное тестирование (System Testing): Тестирование системы в целом, чтобы проверить ее работоспособность и соответствие требованиям.

- Приемочное тестирование (Acceptance Testing): Тестирование системы на соответствие требованиям заказчика или пользователей.

*2. По цели тестирования:*

- Функциональное тестирование (Functional Testing): Тестирование функциональности системы с учетом ожидаемых результатов.

- Нефункциональное тестирование (Non-functional Testing): Тестирование нефункциональных аспектов системы, таких как производительность, безопасность, нагрузка и т.д.

- Регрессионное тестирование (Regression Testing): Повторное тестирование системы после внесения изменений для обнаружения возможных негативных влияний на существующую функциональность.

- Испытание граничных значений (Boundary Testing): Тестирование системы на пределах допустимых значений входных данных для проверки ее поведения.

*3. По способу выполнения:*

- Ручное тестирование (Manual Testing): Тестирование, выполняемое вручную тестировщиками без использования автоматизированных инструментов.

- Автоматизированное тестирование (Automated Testing): Тестирование, выполняемое с использованием специальных инструментов и сценариев тестирования, которые автоматизируют процесс проверки.

- Интерфейсное тестирование (UI Testing): Тестирование пользовательского интерфейса для проверки его удобства, функциональности и соответствия дизайну.

1. Жизненный цикл ПО. Понятие жизненного цикла ПО, этапы, их описание в терминологии различных стандартов (примеры).

2. Процессы создания ПО. Управление процессами, 5 групп процессов, их характерные особенности.

3. Проект. Управление проектами. Понятие проектного офиса, типы проектных офисов. Программы и портфели.

4. Требования к ПО. Виды требований. Техники сбора и описания.

5. Проектное управление. Ограничения и жизненный цикл проекта

6. Проектное управление. Методологии. PMBOOK и PRINCE2

7. Гибкие принципы управление. Agile – идеи и принципы.

8. Гибкие технологии. Scrum

9. Гибкие технологии. Kanban

10. Масштабирование гибкой разработки. Проблемы и решения. SAFe как пример масштабируемого фреймворка

11. Гибкое управление. Особенности организации работы дистанционно работающих команд

12. Поток поставки ценности. DevOps – понятия и основные принципы

13. DevOps. Компоненты и процессы. Понятие безопасной разработки

14. Документация. Типы, назначение. Принципы создания, группы заинтересованных лиц, методики формирования пакетов документации.

15. Тестирование ПО. Цели и принципы

16. Тестирование. Уровни и типы тестирования.

17. Тестирование. Методы черного и белого ящика. Статическое и динамическое тестирование

18. Тестирование. Методы и инструменты. Классификация дефектов

19. Сопровождение и поддержка ПО. ITSM и ITIL общие понятия и принципы

20. Сопровождение и поддержка. ITIL. Базовые процессы

21. Управление изменениями. ITIL и процесс управления изменениями

22. Процесс разработки. Архитектура решения. Назначение, принципы выбора. Анализ и оценка

23. Процесс разработки. Архитектура. Документирование и распространение сведений. Группы заинтересованных лиц.

24. Атрибуты качества систем и их связь с функциональными требованиями. Сценарии атрибутов качества

25. Атрибуты качества и сценарии - подробное описание на примере одного из типа сценариев (Модифицируемость, Ремонтопригодность, Производительность, Безопасность...)

26. Тактики обеспечения атрибутов качества (на примере одного из типов Модифицируемость, Ремонтопригодность, Производительность, Безопасность...)

27. Документирование программных систем. Заинтересованные лица, представления и нотации (с примерами).

28. Документирование программных систем. Неформальные нотации. Плюсы и минусы, области применения

29. Документирование программных систем. Частично формальные и формальные нотации. Плюсы и минусы, области применения

30. Нотации. Рассмотреть пример одной из нотаций на выбор. Основные типы диаграмм, области применения

31. Стили и шаблоны проектирования. Понятие архитектурного стиля и архитектурного шаблона

32. Архитектурные принципы проектирования. Основные принципы и подробно на примере 1 – 2 выбранных

33. Архитектурные стили. Уровни применения, типичные представителе (на примере выбранного - монолит, SOA, Микросервисы...)

34. Архитектурные стили на уровне компонентов. Слоистая и модульная архитектура

35. Паттерны проектирования. Общие понятия, назначение, плюсы и минусы.

36. Паттерны проектирования. Порождающие паттерны (рассмотреть минимум два паттерна и сравнить их)

37. Паттерны проектирования. Структурные паттерны (рассмотреть минимум два паттерна и сравнить их)

38. Паттерны проектирования. Поведенческие паттерны (рассмотреть минимум два паттерна и сравнить их)

39. Управление качеством. Основные понятия, процессы и практики

40. Управление качеством. Стандарты и модели

41. Управление качеством. Эталонная модель

42. Методологии управления качеством. Классификация

43. Рефакторинг. Назначение и принципы, применяемые подходы.

44. Рефакторинг. Техники рефакторинга (на примере одной из техник)

45. Метрики программного кода. Основные принципы и подходы, инструменты, практика применения.

46. Мониторинг систем. Основные решаемые задачи, принципы организации, типы комплексных систем мониторинга.

47. Процессы тестирования ПО. Опишите основные способы деления видов тестирования.

48. Процесс управления релизами. Выделите основные понятия и опишите процесс.