Ответы к экзамену по программной инженерии

Дисциплина «Программная инженерия»

Единый конспект по вопросам

Содержание

Ответы подготовлены по материалам из директории `zelenko/`. Если в материалах курса вопрос раскрыт слабо или почти отсутствует, это отмечено в начале ответа. Список вопросов взят из документа `Список_вопросов_к_экзамену_по_ПИ_2025_2026.docx`.

Основы программной инженерии и требования

1. Основные понятия и определения курса

Недостаточно информации в предоставленных источниках: в локальных материалах есть отдельные определения модели, нотации, CASE-средств, UML и предметной области, но нет полного вводного конспекта по базовым понятиям программной инженерии.

Программная инженерия изучает, как создавать, сопровождать и развивать программные системы инженерным способом: через требования, проектирование, конструирование, тестирование, сопровождение, управление качеством, управление конфигурацией и управление проектом. В отличие от простого программирования, здесь важен не только код, но и весь жизненный цикл продукта.

Базовые понятия курса удобно держать в такой связке. Программное обеспечение - это программы, данные, документация и процедуры, которые вместе решают задачу пользователя или организации. Программная система - более широкое понятие: кроме ПО, в нее входят аппаратная среда, пользователи, внешние системы, регламенты и инфраструктура эксплуатации. Программный продукт - результат разработки, который можно поставлять, внедрять и сопровождать.

Требование описывает, что система должна делать или каким свойством должна обладать. Проектирование отвечает за устройство решения: архитектуру, компоненты, интерфейсы, данные и распределение ответственности. Конструирование - непосредственное создание работающего ПО: кодирование, локальное тестирование, сборка, интеграция. Тестирование проверяет соответствие продукта требованиям и ищет дефекты. Сопровождение начинается после передачи системы в эксплуатацию и включает исправления, адаптацию, улучшения и предотвращение будущих проблем.

В локальных материалах отдельно подчеркивается роль моделей. Модель - это абстракция физической или программной системы, рассматриваемая с определенной точки зрения и представленная на некотором языке или графически. Для сложных систем модели нужны не «для красоты», а чтобы справляться со сложностью, согласовывать требования, архитектуру и реализацию, контролировать изменения и улучшать коммуникацию между аналитиками, разработчиками, тестировщиками и руководителями.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 3-5, 19-20; SWEBOK Guide, Introduction и гл. 1-10.

2. IT-проекты: статистика успешности, причины неудач

В статистике IT-проектов обычно различают три исхода: успешный проект, проблемный проект и проваленный проект. Успешный проект укладывается в основные ограничения по срокам, бюджету и содержанию. Проблемный проект все же доходит до эксплуатации, но с перерасходом, задержками или урезанным функционалом. Проваленный проект отменяется либо дает результат, который невозможно нормально использовать.

Классическая статистика Standish Group CHAOS Report 1994 часто приводится как иллюстрация масштаба проблемы: около 16% проектов были успешными, около 53% - проблемными, около 31% - отмененными или проваленными. Поздние отчеты давали другие проценты, потому что менялись выборки, критерии и индустрия, но общий вывод остался тем же: программные проекты часто срываются не из-за одной ошибки, а из-за сочетания организационных, технических и коммуникационных причин.

В материалах курса перечислены типичные причины неудач:

- слабое управление требованиями;
- несогласованность требований, проектных решений и реализации;
- жесткая архитектура, которую трудно менять;
- рост сложности ПО;
- неточная и противоречивая коммуникация;
- недостаточное тестирование;
- субъективная оценка приоритетов артефактов проекта;
- игнорирование рисков и отсутствие процедур управления рисками;

- бесконтрольное внесение изменений;
- слабое использование CASE-средств, моделей и средств поддержки жизненного цикла.

Отдельно важно, что отсутствие моделей усиливает почти все эти проблемы. Когда команда не моделирует требования, архитектуру и поведение системы, участники проекта начинают по-разному понимать один и тот же продукт. Аналитик, разработчик, тестировщик и заказчик говорят вроде бы об одном, но подразумевают разные вещи. Поэтому визуальное моделирование, управление требованиями, контроль изменений и управление рисками в программной инженерии рассматриваются как практические способы снизить вероятность провала проекта.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 2-3; The Standish Group CHAOS Report 1994; SWEBOK Guide, гл. 7 и 10.

3. Инженерия программного обеспечения (программная инженерия): история, определения, инженерная деятельность, область действия программной инженерии

Недостаточно информации в предоставленных источниках: в локальных презентациях есть связь программной инженерии с UML, CASE-средствами и визуальным моделированием, но почти нет истории и общего определения дисциплины.

Термин «software engineering» закрепился после конференций НАТО 1968-1969 годов, где обсуждали «кризис программного обеспечения»: проекты становились крупнее, дороже и сложнее, а привычное программирование уже не давало управляемого результата. Главная идея была простой: ПО нужно разрабатывать не как набор отдельных программ, а как инженерный продукт - с требованиями, проектными решениями, проверками, измерениями, управлением качеством и ответственностью за эксплуатацию.

В классическом определении IEEE программная инженерия - это применение систематического, дисциплинированного и количественно измеримого подхода к разработке, эксплуатации и сопровождению программного обеспечения. В этом определении важны три слова. «Систематический» означает, что работа строится по процессу, а не только по вдохновению. «Дисциплинированный» - что команда соблюдает правила, договоренности и контроль изменений. «Измеримый» - что качество, трудоемкость, дефекты, сроки и риски нужно оценивать не только интуитивно.

Инженерная деятельность в программной инженерии охватывает не один этап кодирования. Она включает выявление и анализ требований, проектирование архитектуры и данных, конструирование, тестирование, интеграцию, поставку, сопровождение, управление конфигурацией, управление проектом, оценку качества, выбор методов и инструментов. Поэтому область действия программной инженерии шире, чем деятельность программиста: она связывает технические решения, процесс разработки, команду, заказчика и эксплуатацию.

Локальные материалы хорошо показывают прикладной смысл этой дисциплины: сложную систему нужно моделировать, документировать и обсуждать на языке, который понятен разным участникам проекта. UML, CASE-средства, репозитории проектных метаданных и визуальные модели нужны именно для этого.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 5-8, 19-20; SWEBOK Guide, Introduction; ISO/IEC/IEEE 12207:2017, Abstract.

4. Требования к ПО: функциональные и нефункциональные. Диаграмма Вигерса

Недостаточно информации в предоставленных источниках: отдельной лекции по требованиям в папке нет; в материалах требования упоминаются в связи с причинами провала проектов, UML и задачами проектировщика.

Требование к ПО - это зафиксированное ожидание к системе: что она должна делать, какие ограничения соблюдать и какими свойствами обладать. Ошибки в требованиях особенно опасны, потому что они переходят в архитектуру, код, тесты и документацию. Поэтому управление требованиями в материалах курса прямо названо одной из причин успеха или провала IT-проектов.

Функциональные требования описывают поведение системы: какие функции доступны пользователю, какие данные вводятся и выводятся, какие сценарии поддерживаются, какие правила выполняются. Примеры: «пользователь может оформить заказ», «администратор добавляет роль», «система формирует отчет за период».

Нефункциональные требования описывают качества и ограничения системы. Это производительность, надежность, удобство использования, безопасность, масштабируемость, сопровождаемость, совместимость, переносимость, требования к интерфейсам, платформам, законодательным ограничениям и т.п. Например: «ответ на запрос не дольше двух секунд», «доступ только по ролям», «система работает в браузерах последних двух версий».

Диаграмма Вигерса помогает не смешивать уровни требований. В верхней части находятся бизнес-требования: зачем продукт нужен организации. Ниже - пользовательские требования: какие цели и сценарии есть у пользователей. Еще ниже - функциональные требования, которые описывают конкретное поведение системы. Рядом обычно выделяют бизнес-правила, ограничения, внешние интерфейсы, атрибуты качества и требования к данным. Смысл диаграммы в том, что функциональность не должна возникать сама по себе: она должна трассироваться к пользовательским и бизнес-целям.

Для экзамена главное: функциональные требования отвечают на вопрос «что делает система», нефункциональные - «как хорошо и в каких условиях она это делает». Диаграмма Вигерса показывает уровни и типы требований, чтобы не терять связь между целями заказчика и деталями реализации.

Источники: 1 базовые принципы UML-2024.pptx, слайды 2, 8; Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx, слайд 3; SWEBOK Guide, гл. 1.

Проектирование программного обеспечения

5. Проектирование ПО: общие принципы разработки программных систем

Недостаточно информации в предоставленных источниках: локальные материалы раскрывают визуальное моделирование и UML, но не дают полного перечня общих принципов проектирования ПО.

Проектирование ПО - это переход от требований к устройству будущей системы. На этом этапе решают, какие подсистемы будут в продукте, какие у них обязанности, какие данные они хранят, как общаются между собой и какие технические ограничения нужно учесть. Хорошее проектирование уменьшает случайную сложность: система становится понятнее, ее легче менять, тестировать и развивать.

К общим принципам разработки программных систем относятся:

- абстракция - выделение существенных свойств объекта и отбрасывание деталей, неважных для текущего уровня рассмотрения;
- модульность - разбиение системы на части с понятными границами ответственности;
- инкапсуляция - сокрытие внутреннего устройства модуля или класса за интерфейсом;
- низкая связность - минимум лишних зависимостей между модулями;
- высокая связность внутри модуля - элементы одного модуля работают на одну понятную задачу;
- разделение ответственности - каждая часть системы должна отвечать за свой аспект поведения;
- проектирование через интерфейсы - зависимость от контрактов, а не от случайных деталей реализации;
- повторное использование - выделение общих компонентов, библиотек и шаблонов там, где это не усложняет систему.

В объектно-ориентированном проектировании эти принципы выражаются через классы, объекты, наследование, полиморфизм и композицию. В материалах курса ООП определяется как подход, где архитектура системы строится вокруг взаимодействия объектов, а ОО-анализ и проектирование - как представление предметной области через объекты и классы.

Визуальное моделирование в UML помогает зафиксировать проектные решения до кода. Диаграммы классов показывают статическую структуру, диаграммы состояний - жизненный цикл объектов, диаграммы компонентов и развертывания - физическую организацию системы. Поэтому проектирование в программной инженерии - это не только «нарисовать архитектуру», а согласовать структуру решения с требованиями, качеством и ограничениями проекта.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 7-10, 19-20; `8  Диаграмма классов 2025.pptx`, слайды 2-3; SWEBOK Guide, гл. 2.

6. Проектирование ПО: проблемы разработки сложных программных систем

Недостаточно информации в предоставленных источниках: в презентациях есть причины неудач и роль моделей, но нет отдельного подробного разбора сложности программных систем.

Сложная программная система трудна не только из-за большого объема кода. Основная проблема в количестве связей: между требованиями, пользователями, компонентами, данными, внешними сервисами, инфраструктурой, командными ролями и ограничениями эксплуатации. Чем больше таких связей, тем выше риск, что локальное изменение сломает поведение в другом месте.

В материалах курса прямо перечислены проблемы, которые ведут к неудачам: нарастающая сложность ПО, жесткая архитектура, несогласованность требований и реализации, слабая коммуникация, бесконтрольные изменения и недостаточное моделирование. Эти причины хорошо описывают природу сложных систем. Если архитектура не рассчитана на изменения, каждое новое требование превращается в борьбу с уже написанным кодом. Если требования противоречивы, команда строит разные части системы под разные ожидания. Если нет общей модели, участники проекта теряют единый язык.

Сложность бывает предметной и технической. Предметная сложность связана с реальными правилами бизнеса: исключениями, ролями, состояниями, процессами, правовыми ограничениями. Техническая сложность возникает из-за распределенности, конкуренции, отказов, интеграций, производительности, безопасности, поддержки разных платформ. Часть сложности неизбежна, но часть появляется из-за плохих решений: лишних зависимостей, дублирования, неясных интерфейсов, отсутствия тестов и документации.

Справляться со сложностью помогают декомпозиция, архитектурные уровни, четкие интерфейсы, моделирование, итеративная проверка решений, контроль изменений и управление рисками. В этом смысле UML и CASE-средства нужны не как формальность, а как способ сделать сложную систему обозримой для команды.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 2-8, 11; SWEBOK Guide, гл. 2 и 7.

7. Проектирование ПО: структура сложных систем

Недостаточно информации в предоставленных источниках: локальные материалы показывают, что визуальные модели представляют архитектуру ПС, но структура сложных систем раскрыта только фрагментарно.

Структура сложной программной системы - это способ организовать ее части и связи между ними. Обычно систему рассматривают на нескольких уровнях: предметная область, пользовательский интерфейс, прикладная логика, данные, интеграции, инфраструктура и развертывание. Такое разделение не обязательно строго одинаково во всех проектах, но принцип один: разные виды ответственности не должны хаотично смешиваться.

В презентации по UML показана типовая идея: визуальная модель системы не должна зависеть от языка реализации, а архитектуру можно представить слоями интерфейсов пользователя, бизнес-логики и баз данных. Это важный принцип: проектная структура должна быть понятна независимо от того, написана система на Java, C++, SQL или другом языке.

Сложные системы обычно строятся иерархически. На верхнем уровне выделяют подсистемы и крупные компоненты. Ниже - модули, классы, интерфейсы, операции и данные. Горизонтально систему делят по слоям: представление, предметная логика, доступ к данным, интеграция с внешними сервисами. Вертикально - по функциональным областям: заказы, платежи, пользователи, отчеты и т.д.

Хорошая структура должна обладать несколькими свойствами. Компоненты имеют понятные обязанности. Интерфейсы скрывают детали реализации. Зависимости направлены осмысленно, а не образуют случайную сеть. Изменение внутри одного модуля не должно требовать массовых правок в других. Критические решения - например выбор архитектурного стиля, способ хранения данных, протоколы взаимодействия - фиксируются в проектной документации и моделях.

UML помогает описывать структуру с разных сторон: диаграмма классов показывает статическую модель, диаграмма компонентов - программные компоненты и зависимости, диаграмма развертывания - физические узлы и размещение артефактов. Вместе они дают не одну «главную картинку», а набор согласованных представлений сложной системы.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 7, 19-20, 24-25; `8  Диаграмма классов 2025.pptx`, слайды 2, 19-32; `6  Диаграмма компонентов 2025.pptx`, слайды 2-4; SWEBOK Guide, гл. 2.

8. Проектирование ПО: виды дизайна, цели архитектуры, техники проектирования

Недостаточно информации в предоставленных источниках: в папке есть материалы по UML-диаграммам, но нет отдельной лекции, где системно перечислены виды дизайна и техники проектирования.

В программной инженерии слово «дизайн» означает не только внешний вид интерфейса. Проектирование охватывает несколько видов дизайна. Архитектурный дизайн определяет крупные подсистемы, компоненты, слои, внешние интеграции и ключевые технические решения. Детальный дизайн описывает внутреннюю структуру модулей, классов, алгоритмов и интерфейсов. Дизайн

данных определяет сущности, связи, схемы хранения и правила целостности. Дизайн интерфейса описывает взаимодействие пользователя с системой. Дизайн развертывания показывает, на каких узлах и в какой среде будет работать система.

Цели архитектуры - сделать систему пригодной к развитию и эксплуатации. Архитектура должна поддерживать функциональные требования, но ее главная нагрузка часто связана с нефункциональными свойствами: производительностью, надежностью, безопасностью, масштабируемостью, сопровождаемостью, переносимостью. Архитектура также служит средством коммуникации: она объясняет команде, где проходят границы компонентов и почему система устроена именно так.

К техникам проектирования относятся декомпозиция, абстрагирование, моделирование предметной области, проектирование интерфейсов, использование архитектурных стилей и шаблонов, прототипирование, пошаговое уточнение и анализ альтернатив. В объектно-ориентированном подходе часто применяют диаграммы классов, состояний, вариантов использования, последовательности, компонентов и развертывания.

В локальных материалах отдельно подчеркнуто, что UML не является методологией, процессом или языком программирования. UML - это нотация плюс семантика. Значит, сама нотация не гарантирует хорошего проектирования: она только дает общий язык, на котором можно зафиксировать и обсудить проектные решения.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 5, 7-8, 19-20; `2_1_Типы_диаграмм_в_UML_Диаграмма_вариантов_использования_2025.pptx`, слайды 2-3; SWEBOK Guide, гл. 2.

9. Проектирование ПО: ключевые вопросы проектирования, уровни архитектуры, оценка качества проектирования

Недостаточно информации в предоставленных источниках: локальные презентации дают материал по UML и моделированию, но не содержат отдельного списка ключевых вопросов проектирования и критериев оценки дизайна.

Ключевые вопросы проектирования сводятся к тому, как система будет выполнять требования и выдерживать ограничения. На практике архитектор и команда отвечают на такие вопросы: какие подсистемы нужны, где проходят их границы, какие данные где хранятся, какие интерфейсы будут между компонентами, как обрабатываются ошибки, как обеспечиваются безопасность и производительность, как система масштабируется, как ее тестировать, разворачивать и сопровождать.

Уровни архитектуры можно представить так:

1. Концептуальный уровень - основные понятия предметной области, роли пользователей, крупные сценарии и бизнес-правила.
2. Логический уровень - подсистемы, классы, компоненты, интерфейсы, состояния и взаимодействия.
3. Физический уровень - процессы, узлы, серверы, базы данных, сети, артефакты поставки и развертывание.
4. Технологический уровень - конкретные языки, фреймворки, СУБД, протоколы, инструменты сборки и эксплуатации.

В материалах курса эта идея видна через разные UML-представления. Диаграмма классов описывает статическую структуру, диаграмма компонентов - программные компоненты и зависимости, диаграмма развертывания - физические узлы выполнения и артефакты. То есть архитектура оценивается не по одной диаграмме, а по согласованности нескольких представлений.

Качество проектирования оценивают по нескольким признакам. Проект должен быть понятным, непротиворечивым, трассируемым к требованиям, достаточно простым, модульным, расширяемым, тестируемым и устойчивым к изменениям. В локальных материалах используется понятие непротиворечивой модели: модель считается корректной, если соответствует правилам нотации и семантики UML. Но этого мало: модель может быть формально корректной и все равно плохо

решать задачу. Поэтому дополнительно оценивают адекватность модели предметной области и соответствие целям проекта.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 19-25; `6 Диаграмма компонентов 2025.pptx`, слайды 2-4; `7 Диаграмма развертывания 2025.pptx`, слайды 2-8; SWEBOK Guide, гл. 2 и 10.

10. Проектирование ПО: стратегии и методы, нотации проектирования

Недостаточно информации в предоставленных источниках: материалы хорошо раскрывают UML и часть графических нотаций, но стратегии и методы проектирования даны не как отдельная тема.

Стратегия проектирования - это общий способ двигаться от требований к решению. При нисходящем проектировании сначала выделяют общую архитектуру, крупные подсистемы и интерфейсы, затем постепенно уточняют детали. При восходящем проектировании отталкиваются от уже имеющихся компонентов, библиотек или технических возможностей и собирают из них более крупное решение. В реальных проектах эти подходы часто смешиваются: архитектура задает рамки сверху, а ограничения технологий и готовых компонентов влияют снизу.

Методы проектирования зависят от выбранной парадигмы. Структурный подход использует функциональную декомпозицию, потоки данных, состояния и модули. Объектно-ориентированный подход строит модель через классы, объекты, их связи, состояния и взаимодействия. Компонентный подход рассматривает систему как набор заменяемых компонентов с предоставляемыми и требуемыми интерфейсами. Архитектурное проектирование добавляет стили и шаблоны: слоистая архитектура, клиент-сервер, микросервисы, событийная архитектура и т.д.

Нотация проектирования - это язык, на котором фиксируют модель. В локальных материалах названы UML, IDEF и ARIS. Также перечислены популярные исторические нотации: ERD для сущностей и связей, DFD для потоков данных, STD для переходов состояний, SADT для структурного анализа и проектирования, FDD для функциональной декомпозиции.

UML в курсе занимает центральное место. Он определен как унифицированный язык моделирования для описания, визуализации и документирования объектно-ориентированных систем при анализе и проектировании. При этом UML не является методологией, процессом, языком программирования или полностью формальным языком. Он задает нотацию и семантику, а метод разработки выбирается отдельно: например RUP, MSF, Agile-подходы или внутренний процесс организации.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 12-20; `2_1_Типы_диаграмм_в_UML_Диаграмма_вариантов_использования_2025.pptx`, слайды 2-8; SWEBOK Guide, гл. 2 и 9.

Конструирование, тестирование и сопровождение

11. Конструирование ПО: основы конструирования

Недостаточно информации в предоставленных источниках: отдельной лекции по конструированию ПО в папке нет; ответ опирается на SWEBOK и общую структуру курса.

Конструирование ПО - это часть жизненного цикла, где проектные решения превращаются в работающий программный продукт. Сюда входит кодирование, создание модулей, работа с данными, локальное тестирование, отладка, ревью, сборка, документирование кода, интеграция и подготовка исполняемых артефактов.

Конструирование не сводится к набору строк кода. Оно опирается на требования и проектную модель: разработчик должен понимать, какую ответственность несет модуль, какой интерфейс он реализует, какие ограничения по качеству существуют. Если проектирование отвечает на вопрос «как система должна быть устроена», то конструирование отвечает на вопрос «как это устройство реализовать надежно и проверяемо».

К основам конструирования относятся:

- выбор языка, библиотек, фреймворков и среды выполнения;
- соблюдение стандартов кодирования;
- использование понятных имен, простых структур данных и контролируемых зависимостей;
- обработка ошибок и исключительных ситуаций;
- защита данных и учет ограничений безопасности;
- модульное и интеграционное тестирование;
- автоматизация сборки и проверки;
- управление версиями исходного кода.

Качественное конструирование связано с архитектурой. Если код нарушает границы компонентов, обходит интерфейсы и дублирует логику, архитектура быстро становится фикцией. Поэтому разработчик отвечает не только за локальную работоспособность функции, но и за сохранение структуры системы.

Для экзамена можно запомнить так: конструирование - это инженерная реализация проектных решений, где важны читаемость, надежность, тестируемость, повторное использование и управляемая интеграция. Чем лучше организовано конструирование, тем дешевле последующее тестирование и сопровождение.

Источники: SWEBOK Guide, гл. 3; ISO/IEC/IEEE 12207:2017, Abstract.

12. Конструирование ПО: управление конструированием, языки конструирования

Недостаточно информации в предоставленных источниках: локальные материалы почти не раскрывают управление конструированием и языки программирования как тему конструирования.

Управление конструированием нужно, чтобы разработка кода не превращалась в набор несвязанных индивидуальных решений. Команда заранее определяет стандарты кодирования, правила ветвления, порядок ревью, требования к тестам, правила сборки, способ регистрации дефектов и критерии готовности. Эти договоренности особенно важны в больших проектах, где один модуль зависит от другого, а ошибка в интерфейсе может задержать всю интеграцию.

В управлении конструированием обычно выделяют несколько задач. Во-первых, планирование работ: какие компоненты реализуются, в какой последовательности и кем. Во-вторых, контроль технических решений: соответствуют ли они архитектуре и требованиям. В-третьих, контроль качества кода: ревью, статический анализ, тесты, метрики сложности. В-четвертых, управление сборками и версиями: код должен воспроизводимо собираться, тестироваться и поставляться.

Языки конструирования - это языки, на которых создают программные артефакты. Обычно говорят не только о языках программирования общего назначения, но и о языках запросов, разметки, описания интерфейсов, конфигурации, сценариев сборки и моделирования. Например, Java, C#, C+

+, Python и Go используются для прикладной логики; SQL - для работы с базами данных; HTML и CSS - для интерфейсов; YAML и JSON - для конфигураций; UML - для визуального описания моделей.

Выбор языка влияет на архитектуру, производительность, безопасность, сопровождение и доступность специалистов. Но язык сам по себе не решает инженерные проблемы. Даже сильный язык не спасает проект, если нет управления требованиями, архитектурной дисциплины, тестирования и контроля изменений.

Источники: `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайд 4; SWEBOK Guide, гл. 3 и 6.

13. Конструирование ПО: кодирование, тестирование в конструировании, оценка качества, интеграция

Недостаточно информации в предоставленных источниках: вопрос относится к конструированию, а в локальной папке нет отдельного блока по этой теме.

Кодирование - центральная, но не единственная часть конструирования. Разработчик реализует модули, классы, функции, обработку ошибок, работу с данными и внешними сервисами. Хороший код должен быть не только рабочим, но и читаемым, проверяемым, изменяемым. Поэтому в профессиональной разработке кодирование сопровождается ревью, статическим анализом, автоматическими тестами и сборкой.

Тестирование в конструировании начинается до системного тестирования. На этом уровне проверяют отдельные функции, классы и компоненты: модульные тесты, компонентные тесты, тесты контрактов, проверки обработки ошибок. Их задача - быстро обнаружить дефект рядом с местом его появления. Чем позже ошибка найдена, тем дороже ее исправление, потому что она уже могла повлиять на другие части системы.

Оценка качества при конструировании включает несколько признаков: соответствие требованиям и архитектуре, отсутствие известных дефектов, читаемость, низкую сложность, покрытие тестами, отсутствие опасных зависимостей, корректную обработку исключений, безопасность и производительность. Метрики не заменяют инженерное мышление, но помогают увидеть риск: слишком длинные методы, высокая цикломатическая сложность, дублирование, нестабильные тесты, рост числа дефектов.

Интеграция - соединение реализованных частей в работающую систему. Она бывает разовой и постепенной. В современной практике предпочтительна непрерывная интеграция: изменения часто попадают в общую ветку, автоматически собираются и проверяются тестами. Это снижает риск «большого взрыва», когда модули долго разрабатывались отдельно и в конце внезапно не стыкуются.

Для экзамена важно связать эти части: кодирование создает программные элементы, тестирование в конструировании проверяет их локально, оценка качества показывает техническое состояние, интеграция подтверждает, что части работают вместе.

Источники: SWEBOK Guide, гл. 3, 4 и 6; ISO/IEC/IEEE 12207:2017, Abstract.

14. Тестирование ПО: основные концепции и определение тестирования, уровни тестирования

Недостаточно информации в предоставленных источниках: в локальных материалах тестирование упоминается как причина неудач и роль в команде, но теоретического блока по тестированию нет.

Тестирование ПО - это деятельность по проверке программного продукта с целью найти дефекты и получить информацию о его качестве. Тестирование не доказывает полного отсутствия ошибок. Оно снижает неопределенность: показывает, как система ведет себя в выбранных условиях, соответствует ли она требованиям и где остаются риски.

Основные концепции тестирования: тестовый случай, тестовые данные, ожидаемый результат, фактический результат, дефект, отказ, покрытие, тестовая среда, регрессия. Дефект - ошибка в артефакте, например в коде или требовании. Отказ - наблюдаемое неправильное поведение системы при

выполнении. Один дефект может проявляться не всегда, поэтому тестирование работает с конкретными условиями запуска.

Обычно выделяют несколько уровней тестирования:

1. Модульное тестирование - проверка отдельных функций, классов или модулей.
2. Интеграционное тестирование - проверка взаимодействия между модулями, компонентами и внешними сервисами.
3. Системное тестирование - проверка всей системы как единого продукта.
4. Приемочное тестирование - проверка пригодности продукта для заказчика или пользователя.

Дополнительно выделяют регрессионное тестирование, которое проверяет, что новые изменения не сломали старую функциональность. В больших проектах оно обычно автоматизируется.

В локальной лекции по команде тестировщик отвечает за удовлетворение функциональных и нефункциональных требований, план тестирования, базу зарегистрированных ошибок, разработку тестов, автоматизацию и организацию бета-тестирования. Это хорошо показывает, что тестирование - не финальная «проверка после разработки», а отдельная инженерная деятельность, встроенная в процесс проекта.

Источники: `1 базовые принципы UML-2024.pptx`, слайд 2; `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайд 3; SWEBOK Guide, гл. 4.

15. Тестирование ПО: техники тестирования, метрики тестирования, управление тестированием

Недостаточно информации в предоставленных источниках: локальные материалы дают обязанности тестировщика, но не раскрывают техники и метрики тестирования подробно.

Техники тестирования делят на несколько групп. Тестирование «черного ящика» строится по требованиям и внешнему поведению системы: тестировщик не опирается на внутренний код. Сюда относятся разбиение на классы эквивалентности, анализ граничных значений, таблицы принятия решений, тестирование состояний, тестирование сценариев использования. Тестирование «белого ящика» использует знание кода: проверяются ветви, условия, пути выполнения, обработка исключений. Есть также тестирование «серого ящика», когда тестировщик знает часть внутренней структуры.

По способу выполнения тестирование бывает ручным и автоматизированным. Ручное полезно для исследовательских проверок, UX, нестандартных сценариев. Автоматизация нужна для повторяемых проверок: модульных, интеграционных, регрессионных, нагрузочных.

Метрики тестирования помогают управлять качеством и ходом работ. Часто используют количество найденных и закрытых дефектов, плотность дефектов, долю успешных тестов, покрытие требований тестами, покрытие кода, среднее время исправления дефекта, число повторно открытых дефектов, стабильность автотестов. Метрики нужно трактовать осторожно: например, большое покрытие кода не гарантирует хороших тестов, а малое число дефектов может означать как высокое качество, так и слабое тестирование.

Управление тестированием включает планирование, подготовку тестовой среды и данных, распределение задач, ведение дефектов, контроль статусов, отчетность и принятие решения о готовности релиза. В локальной лекции отдельно указано, что тестировщик составляет план тестирования до начала реализации, контролирует выполнение плана и поддерживает целостность базы зарегистрированных ошибок: кто нашел, где, когда, при каких условиях, какой статус и приоритет.

Источники: `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайд 3; SWEBOK Guide, гл. 4.

16. Сопровождение ПО: основные концепции, категории сопровождения, ключевые вопросы сопровождения ПО

Недостаточно информации в предоставленных источниках: в локальной папке нет отдельной лекции по сопровождению ПО.

Сопровождение ПО - это изменение программного продукта после поставки в эксплуатацию. Оно нужно не только для исправления ошибок. Система живет в меняющейся среде: меняются требования бизнеса, законы, операционные системы, интеграции, нагрузки, угрозы безопасности и ожидания пользователей.

Классически выделяют четыре категории сопровождения:

- корректирующее - исправление обнаруженных дефектов;
- адаптивное - приспособление к изменившейся среде, например новой ОС, СУБД, API или законодательным требованиям;
- совершенствующее - улучшение функциональности, производительности, удобства, сопровождаемости;
- профилактическое - изменения, которые уменьшают вероятность будущих проблем: рефакторинг, обновление зависимостей, улучшение тестов, удаление технического долга.

Ключевые вопросы сопровождения связаны с управляемостью изменений. Нужно понять, какое изменение требуется, почему оно нужно, какие компоненты затрагивает, какие риски создает, как его протестировать и как поставить пользователям. Без управления конфигурацией и трассируемости сопровождение быстро становится опасным: исправление одной ошибки может породить несколько новых.

Большая часть стоимости ПО часто возникает именно после первого релиза. Поэтому сопровождаемость должна закладываться еще при проектировании и конструировании. Чистые интерфейсы, модульность, тесты, документация, понятные модели и автоматизированная сборка снижают цену будущих изменений.

Для экзамена удобно сказать так: сопровождение - это инженерная работа с уже поставленным программным продуктом, направленная на исправление, адаптацию, улучшение и предупреждение проблем. Его качество зависит от архитектуры, документации, тестов и дисциплины управления изменениями.

Источники: SWEBOK Guide, гл. 5; ISO/IEC/IEEE 12207:2017, Abstract.

17. Сопровождение ПО: процессы сопровождения, техники сопровождения

Недостаточно информации в предоставленных источниках: процессы и техники сопровождения в локальных материалах почти не представлены.

Процесс сопровождения начинается с регистрации запроса на изменение или сообщения о дефекте. Затем запрос анализируют: уточняют проблему, оценивают важность, воспроизводимость, затронутые требования, компоненты и риски. После этого принимают решение - отклонить запрос, отложить, включить в план релиза или исправить срочно. Далее выполняются изменение, ревью, тестирование, сборка, выпуск и обновление документации.

Типовой процесс сопровождения можно представить так:

1. Получение запроса или инцидента.
2. Классификация: дефект, адаптация, улучшение, профилактика.
3. Анализ влияния на требования, архитектуру, код, тесты и эксплуатацию.
4. Планирование изменения и назначение ответственных.
5. Реализация и локальная проверка.
6. Регрессионное тестирование.
7. Выпуск версии и сопровождение внедрения.
8. Обновление документации и базы знаний.

К техникам сопровождения относятся анализ влияния, трассировка требований, рефакторинг, обратная инженерия, статический анализ, чтение и профилирование кода, регрессионное тестирование, мониторинг, логирование, управление дефектами и управление версиями. При работе со старым кодом особенно важны маленькие безопасные изменения: сначала покрыть критический участок тестами, затем исправлять или улучшать.

Сопровождение тесно связано с управлением конфигурацией. Команда должна понимать, какая версия стоит у пользователя, какие изменения в нее вошли, какие дефекты известны, какие компоненты и библиотеки использованы. Без этой информации сопровождение превращается в угадывание.

Смысл сопровождения не в бесконечном латании продукта, а в поддержании его полезности. Хорошее сопровождение продлевает срок жизни системы и позволяет ей меняться вместе с организацией.

Источники: SWEBOK Guide, гл. 5 и 6; ISO/IEC/IEEE 12207:2017, Abstract.

18. Управление конфигурацией ПО: основные понятия, составляющие части

Недостаточно информации в предоставленных источниках: тема управления конфигурацией в локальных презентациях раскрыта только косвенно через контроль изменений и управление исходными текстами.

Управление конфигурацией ПО - это дисциплина, которая помогает контролировать состав программного продукта и изменения в нем. Она отвечает на вопросы: какие артефакты входят в продукт, какие версии существуют, кто и когда внес изменение, как собрать конкретную версию, какие дефекты и требования в нее включены.

Конфигурационная единица - любой артефакт, который нужно контролировать: исходный код, требования, UML-модель, тесты, скрипты сборки, схема базы данных, документация, библиотека, контейнерный образ. Базовая линия - зафиксированное состояние набора конфигурационных единиц, от которого дальше контролируются изменения. Версия - конкретное состояние артефакта или продукта.

Основные составляющие управления конфигурацией:

- идентификация конфигурации - определение, какие элементы подлежат контролю;
- контроль изменений - правила внесения, согласования и утверждения изменений;
- учет статуса конфигурации - информация о версиях, изменениях, дефектах и релизах;
- аудит конфигурации - проверка, что продукт действительно соответствует заявленному составу;
- управление сборками и релизами - воспроизводимое создание и поставка версий.

В локальной лекции технолог разработки ПО отвечает за управление исходными текстами, сопровождение системы контроля версий, среду сборки и процедуру установки. Это практическая часть управления конфигурацией. Если сборка не воспроизводится, а изменения вносятся бесконтрольно, команда быстро теряет возможность надежно выпускать продукт.

Для экзамена важно: управление конфигурацией не равно только Git. Система контроля версий - один инструмент. Сама дисциплина шире: она связывает код, требования, модели, тесты, сборки, релизы и изменения в единый управляемый процесс.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 2-3; `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайд 4; SWEBOK Guide, гл. 6.

Управление, процессы, методы и инструменты

19. Управление проектами ПО: организационное, задачи, инженерия измерений

Недостаточно информации в предоставленных источниках: в локальных материалах подробно есть управление командой, но общая тема управления проектами и инженерии измерений раскрыта неполно.

Управление проектами ПО - это планирование, организация и контроль работ по созданию программного продукта. Его цель - получить нужный результат в условиях ограничений по срокам, бюджету, объему, качеству, людям и рискам. В программных проектах управление особенно трудно, потому что требования меняются, сложность растет, а результат долго остается нематериальным.

Организационная сторона управления включает выбор структуры команды, распределение ролей, коммуникацию, принятие решений, работу с заказчиком и внешними группами. В локальной лекции управление программным проектом сведено к трем ключевым задачам: подбор и управление командой, выбор процесса, выбор инструментальных средств. Это хорошее практическое ядро: даже сильная команда без процесса и инструментов работает хаотично, а хороший процесс без людей не дает результата.

Задачи управления проектом включают:

- определение целей, содержания и ограничений проекта;
- планирование работ, сроков и ресурсов;
- управление требованиями и изменениями;
- управление рисками;
- контроль выполнения и качества;
- организация коммуникации;
- подготовка поставки и внедрения;
- работа с метриками и отчетностью.

Инженерия измерений нужна, чтобы проектом управляли не только по ощущениям. Измеряют трудоемкость, скорость выполнения работ, количество дефектов, покрытие тестами, стабильность сборок, выполнение плана, загрузку команды, качество процессов. Но метрики должны помогать принятию решений, а не превращаться в самоцель. Например, число строк кода плохо показывает ценность продукта, а количество закрытых задач без учета дефектов может маскировать проблемы качества.

Источники: `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайды 2-4; SWEBOK Guide, гл. 7 и 12.

20. Процесс инженерии ПО: определения, составляющие части

Недостаточно информации в предоставленных источниках: в локальных материалах процесс явно упоминается через модели ЖЦ, UML и управление командой, но полного определения процесса инженерии ПО нет.

Процесс инженерии ПО - это организованная совокупность работ, ролей, артефактов, правил и критериев, по которым создается и сопровождается программный продукт. Процесс отвечает на вопросы: кто выполняет работу, в какой последовательности, какие входы и выходы у деятельности, какие документы или модели создаются, как проверяется качество, как принимаются решения.

Процесс не нужно понимать как жесткую бюрократию. Даже гибкая разработка имеет процесс: есть роли, правила планирования, доски задач, критерии готовности, проверки и регулярные встречи. Разница между процессами в степени формальности, частоте обратной связи, объеме документации и способе управления изменениями.

Составляющие части процесса обычно включают:

- деятельности: анализ требований, проектирование, конструирование, тестирование, сопровождение;

- роли: заказчик, аналитик, архитектор, разработчик, тестировщик, менеджер проекта, инженер по качеству и др.;
- артефакты: требования, модели, код, тесты, планы, отчеты, сборки, релизы;
- методы и техники: моделирование, ревью, тест-дизайн, анализ рисков;
- инструменты: системы управления требованиями, CASE-средства, Git, CI/CD, трекеры задач;
- метрики и критерии качества;
- правила управления изменениями и конфигурацией.

В локальных материалах хорошо видно, зачем процессу нужны модели. Без моделей команда хуже справляется со сложностью, хуже контролирует изменения и хуже согласует ожидания между ролями. Поэтому процесс инженерии ПО связывает технические и организационные действия в единый жизненный цикл.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 2-8, 19-20; `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайд 2; SWEBOK Guide, гл. 8.

21. Методы инженерии ПО: классификация методов, категории методов

Недостаточно информации в предоставленных источниках: локальные материалы называют UML, IDEF, ARIS, RUP, MSF и визуальное моделирование, но не дают полной классификации методов инженерии ПО.

Метод инженерии ПО - это упорядоченный способ выполнять инженерную работу: выявлять требования, анализировать предметную область, проектировать систему, проверять качество, управлять изменениями. Метод отличается от инструмента: метод объясняет, что и как делать, а инструмент помогает это выполнить.

Методы можно классифицировать по этапам жизненного цикла. Методы работы с требованиями включают интервью, анализ документов, сценарии использования, прототипирование, моделирование бизнес-процессов. Методы проектирования включают структурный анализ, объектно-ориентированное проектирование, компонентное проектирование, архитектурные шаблоны. Методы конструирования включают стандарты кодирования, TDD, ревью, парное программирование, непрерывную интеграцию. Методы тестирования включают классы эквивалентности, граничные значения, тестирование состояний, покрытие кода. Методы сопровождения включают анализ влияния, рефакторинг, обратную инженерию.

Другой способ классификации - по характеру представления. Формальные методы используют математические спецификации и доказательства. Полуформальные методы используют нотации с определенной семантикой, например UML. Неформальные методы опираются на текстовые описания, чек-листы, экспертные обсуждения и практики команды.

В локальных материалах особенно выделены методы визуального моделирования. UML представлен как отраслевой стандарт OMG, IDEF - как методология для анализа производственно-технических и организационно-экономических систем, ARIS - как методология и нотация для моделирования бизнес-процессов. Эти методы помогают превратить неструктурированную информацию в модели, пригодные для анализа, проектирования и коммуникации.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 4-6, 12-16; SWEBOK Guide, гл. 9.

22. Инструменты инженерии ПО: классификация, области применения

Недостаточно информации в предоставленных источниках: в локальных материалах хорошо описаны CASE-средства, но не весь спектр современных инструментов инженерии ПО.

Инструменты инженерии ПО - это программные средства, которые поддерживают процессы жизненного цикла: требования, моделирование, кодирование, тестирование, сборку, управление конфигурацией, проектное управление, эксплуатацию. Их задача - снизить ручную работу, повысить контролируемость и сохранить связь между артефактами проекта.

В материалах курса CASE-средство определяется как ПО, автоматизирующее совокупность процессов жизненного цикла. Отдельно указаны его возможности: мощные графические средства для описания и документирования программных систем, обеспечение управляемости процесса разра-

ботки, использование репозитория проектных метаданных. Также дана эволюция CASE-средств: от генерации схем БД к генерации кода, прямой и обратной кодогенерации, синхронизации кода и моделей.

Классифицировать инструменты можно по областям применения:

- управление требованиями: хранение требований, трассировка, согласование изменений;
- моделирование и проектирование: UML, BPMN, ERD, архитектурные диаграммы;
- разработка: IDE, редакторы, компиляторы, отладчики;
- управление версиями: Git и системы репозиториев;
- сборка и CI/CD: автоматическая сборка, тесты, публикация артефактов;
- тестирование: test management, автотесты, нагрузочное тестирование, статический анализ;
- управление проектом: трекеры задач, доски, планирование релизов;
- эксплуатация: мониторинг, логирование, алертинг, управление инцидентами.

Инструменты полезны только при наличии процесса. CASE-средство или трекер задач не исправят плохие требования и слабую коммуникацию сами по себе. Но при нормальной дисциплине они помогают сохранять целостность модели, кода, тестов и документации.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 5-6, 12-15; SWEBOK Guide, гл. 6, 8 и 9.

Качество, стандарты и жизненный цикл

23. Качество ПО: модель качества по МакКолу. Характеристики качества

Недостаточно информации в предоставленных источниках: в папке нет отдельного материала по модели качества МакКола; ответ опирается на общую теорию качества ПО.

Модель качества МакКола - одна из ранних и известных иерархических моделей качества программного обеспечения. Ее смысл в том, чтобы связать понятное пользователю качество с более конкретными инженерными характеристиками. Модель помогает не говорить абстрактно «программа хорошая», а разложить качество на факторы, критерии и метрики.

В модели МакКола выделяют три группы факторов качества.

Первая группа - эксплуатация продукта, то есть качество при непосредственном использовании:

- корректность - соответствие требованиям и ожиданиям пользователя;
- надежность - способность работать без отказов в заданных условиях;
- эффективность - рациональное использование времени и ресурсов;
- целостность - защита от несанкционированного доступа и повреждения данных;
- удобство использования - насколько легко пользователю освоить и применять систему.

Вторая группа - сопровождение продукта:

- сопровождаемость - насколько легко искать и исправлять дефекты;
- гибкость - насколько просто менять систему под новые требования;
- тестируемость - насколько легко проверять систему и ее части.

Третья группа - перенос и повторное использование:

- переносимость - возможность перенести систему в другую среду;
- повторное использование - пригодность частей системы для других задач;
- взаимодействие - способность работать совместно с другими системами.

Сильная сторона модели МакКола - практичность: она показывает разные стороны качества и помогает обсуждать компромиссы. Например, высокая эффективность может конфликтовать с простотой сопровождения, а безопасность может усложнять удобство использования. Слабая сторона - модель появилась в контексте старых систем и не полностью отражает современные аспекты, такие как облачная эксплуатация, UX-исследования или DevOps. Но как базовая классификация факторов качества она остается полезной.

Источники: SWEBOK Guide, гл. 10; McCall, Richards, Walters, Factors in Software Quality, 1977.

24. Качество ПО: модель качества по Боэму. Характеристики качества

Недостаточно информации в предоставленных источниках: модель Боэма в предоставленных презентациях не раскрыта.

Модель качества Боэма - иерархическая модель, предложенная Барри Боэмом в конце 1970-х годов. Она рассматривает качество через полезность программного продукта и удобна тем, что связывает высокоуровневые цели с более детальными характеристиками.

На верхнем уровне модель говорит об общей полезности продукта. Она раскладывается на три крупные характеристики:

1. Полезность «как есть» - насколько продукт можно эффективно использовать в текущем виде.
2. Сопровождаемость - насколько легко понимать, исправлять, проверять и изменять программу.
3. Переносимость - насколько легко перенести программу в другую среду.

Полезность «как есть» включает надежность, эффективность и удобство для человека. Надежность отвечает за выполнение функций без отказов, эффективность - за использование ресурсов, удобство - за понятность и приемлемость для пользователя.

Сопровождаемость включает понятность, модифицируемость и тестируемость. Это особенно важная часть модели Боэма: программа может работать сегодня, но быть настолько трудной для понимания и изменения, что ее дальнейшая жизнь станет слишком дорогой.

Переносимость связана с независимостью от конкретного оборудования и среды, а также с тем, насколько программа документирована, модульна и приспособлена к переносу.

По сравнению с моделью МакКола модель Боэма делает сильный акцент на иерархии и на том, что качество нельзя оценивать одной метрикой. Для пользователя важна полезность, для команды сопровождения - понятность и изменяемость, для организации - срок жизни продукта и возможность переносить его в новые условия.

Для экзамена можно сформулировать так: модель Боэма оценивает качество через общую полезность, которая достигается за счет текущей пригодности к использованию, сопровождаемости и переносимости.

Источники: SWEBOK Guide, гл. 10; Boehm et al., Characteristics of Software Quality, 1978.

25. Концепция качества ПО по стандарту ISO 9126-01: характеристикам качества (основные и дополнительные), деятельности и техники гарантии качества

Недостаточно информации в предоставленных источниках: ISO 9126-01 в локальных материалах не представлен; ответ основан на справочных данных по стандарту и его преемнику ISO/IEC 25010.

ISO/IEC 9126-1:2001 описывал модель качества программного продукта. Сейчас стандарт официально отозван и заменен семейством ISO/IEC 25010, но в учебных курсах ISO 9126 часто продолжают использовать как классическую модель качества.

В ISO 9126 основными характеристиками внутреннего и внешнего качества были:

- функциональность - способность предоставлять нужные функции и удовлетворять заданные потребности;
- надежность - способность сохранять работоспособность в заданных условиях;
- удобство использования - понятность, обучаемость, удобство работы для пользователя;
- эффективность - соотношение производительности и используемых ресурсов;
- сопровождаемость - удобство анализа, изменения, тестирования и стабилизации продукта;
- переносимость - способность переноситься в другую программно-аппаратную среду.

У каждой характеристики есть подхарактеристики. Например, надежность включает зрелость, отказоустойчивость и восстанавливаемость; сопровождаемость - анализируемость, изменяемость, стабильность и тестируемость; переносимость - адаптируемость, устанавливаемость, сосуществование и заменяемость.

Дополнительно в стандарте выделяли качество при использовании. Оно описывает не внутреннее устройство продукта, а результат для пользователя: эффективность достижения целей, продуктивность, безопасность и удовлетворенность.

Гарантия качества - это не только тестирование готовой программы. Она включает деятельность и техники на протяжении жизненного цикла: стандарты разработки, ревью требований и проектных решений, аудит процессов, статический анализ, тестирование, управление дефектами, управление конфигурацией, метрики качества, проверку документации и приемочные процедуры.

Главная идея ISO 9126: качество нужно описывать через набор измеримых характеристик. Тогда нефункциональные требования можно превращать в критерии приемки, а не оставлять в виде общих слов «быстро», «удобно», «надежно».

Источники: ISO/IEC 9126-1:2001, страница ISO; ISO/IEC 25010:2023, Abstract; SWEBOK Guide, гл. 10.

26. Стандарт ISO 12207. Основные определения. Процессы жизненного цикла (ЖЦ)

Недостаточно информации в предоставленных источниках: ISO 12207 не раскрыт в локальных презентациях. В ответе учтена классическая логика стандарта и современная редакция ISO/IEC/IEEE 12207.

ISO/IEC/IEEE 12207 - стандарт, который задает общую рамку процессов жизненного цикла программного обеспечения. Его задача - дать единый язык для описания работ, связанных с приобретением, поставкой, разработкой, эксплуатацией, сопровождением и выводом программных продуктов из эксплуатации.

Важно не путать стандарт с конкретной моделью разработки. ISO 12207 не говорит, что проект обязан быть водопадным, спиральным или Scrum. Он описывает процессы, виды деятельности и задачи, которые организация может адаптировать под свой жизненный цикл.

В классическом представлении ISO 12207 выделяли основные процессы жизненного цикла: приобретение, поставка, разработка, эксплуатация и сопровождение. Приобретение описывает действия заказчика, поставка - действия поставщика, разработка - создание ПО, эксплуатация - использование системы в рабочей среде, сопровождение - изменения после поставки.

Также выделялись вспомогательные процессы: документирование, управление конфигурацией, обеспечение качества, верификация, валидация, совместные просмотры, аудит, решение проблем. Организационные процессы включали управление, создание инфраструктуры, совершенствование процессов и обучение.

В современных редакциях структура процессов изменилась: используются группы процессов соглашения, организационного обеспечения проекта, технического управления и технические процессы. Но общий смысл остался прежним: жизненный цикл ПО нужно описывать как набор управляемых процессов, а не как случайную последовательность работ.

Для экзамена лучше сказать обе вещи: ISO 12207 задает общий framework процессов ЖЦ, а конкретная организация адаптирует его под свой проект, договор, модель разработки и уровень зрелости.

Источники: ISO/IEC/IEEE 12207:2017, страница ISO; SWEBOK Guide, гл. 8.

27. Модель жизненного цикла разработки ПО как процесс (обобщенная схема, иерархия элементов)

Недостаточно информации в предоставленных источниках: в локальной папке нет отдельной лекции по моделям жизненного цикла разработки ПО.

Модель жизненного цикла ПО описывает, как работа над программным продуктом проходит от идеи до вывода из эксплуатации. Она задает последовательность или повторяемый цикл процессов: требования, проектирование, конструирование, тестирование, внедрение, эксплуатация, сопровождение. Модель ЖЦ нужна, чтобы команда понимала, какие работы выполняются, какие артефакты создаются и где принимаются контрольные решения.

Обобщенная схема жизненного цикла обычно выглядит так: возникновение потребности, анализ предметной области, требования, проектирование, реализация, интеграция и тестирование, поставка, эксплуатация, сопровождение, завершение или замена системы. В разных моделях порядок и степень итеративности отличаются. Водопадная модель делает акцент на последовательных фазах, спиральная - на циклах и рисках, инкрементная - на поставке частей продукта, Agile - на коротких итерациях и обратной связи.

Иерархия элементов процесса может быть описана так:

- жизненный цикл - весь путь программного продукта;
- процесс - крупная область деятельности, например разработка или сопровождение;
- деятельность - группа связанных работ, например анализ требований;
- задача - конкретная работа с входами и выходами;
- операция или техника - способ выполнения задачи;

- артефакт - результат работы: документ, модель, код, тест, сборка, отчет.

Процесс также связан с ролями и ответственностью. Например, аналитик уточняет требования, проектировщик отвечает за архитектуру, разработчик реализует компоненты, тестировщик проверяет качество, менеджер проекта управляет ограничениями, инженер по качеству оценивает процесс.

Для экзамена важно: модель ЖЦ - это не просто схема этапов. Это процессная модель, которая связывает работы, роли, артефакты, контрольные точки и критерии качества.

Источники: `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайды 2-4; ISO/IEC/IEEE 12207:2017, Abstract; SWEBOK Guide, гл. 8.

Модели жизненного цикла

28. Каскадная (водопадная) модель жизненного цикла ПП

Недостаточно информации в предоставленных источниках: локальные материалы не содержат отдельного описания водопадной модели.

Каскадная, или водопадная, модель жизненного цикла представляет разработку как последовательность фаз. Обычно выделяют анализ требований, проектирование, реализацию, тестирование, внедрение и сопровождение. Каждая фаза должна завершиться определенными артефактами и согласованием, после чего команда переходит к следующей.

Главная идея модели - строгая последовательность и документированность. Сначала нужно понять требования, затем спроектировать решение, затем реализовать, затем протестировать и поставить. Такой подход хорошо выглядит для проектов с устойчивыми требованиями, понятной предметной областью, жесткими контрактами и высокой ценой неформальности.

Преимущества водопада:

- понятная структура работ;
- удобство планирования и отчетности;
- четкие контрольные точки;
- хорошая совместимость с договорной разработкой;
- сильная документация.

Недостатки связаны с поздней обратной связью. Пользователь часто видит работающую систему только ближе к концу, когда исправлять ошибки требований уже дорого. Если требования меняются, каскадная модель становится тяжелой: приходится возвращаться к уже закрытым фазам. Еще одна проблема - риск обнаружить архитектурные или интеграционные ошибки слишком поздно.

В реальности чистый водопад встречается редко. Даже в формальных проектах появляются итерации, прототипы, уточнения требований и промежуточные проверки. Но как базовая модель он важен: многие другие модели можно понимать как попытку исправить слабые места водопада - позднюю проверку, слабую гибкость и накопление рисков к концу проекта.

Источники: SWEBOK Guide, гл. 8; ISO/IEC/IEEE 12207:2017, Abstract.

29. Спиральная модель жизненного цикла ПП

Недостаточно информации в предоставленных источниках: спиральная модель в локальных материалах не представлена.

Спиральная модель жизненного цикла, предложенная Барри Боэмом, строит разработку как последовательность циклов, где каждый виток спирали уточняет продукт и снижает риски. В отличие от водопада, здесь важен не разовый проход по фазам, а повторяющаяся работа: определить цели, оценить альтернативы и риски, разработать и проверить часть решения, спланировать следующий цикл.

Типовой виток спирали включает четыре области:

1. Определение целей, ограничений и вариантов решения.
2. Анализ рисков и выбор подхода.
3. Разработка и проверка очередной версии или прототипа.
4. Планирование следующего витка.

Главное достоинство модели - управление рисками. Если в проекте есть неизвестная технология, сложная интеграция, неясные требования или высокая цена ошибки, спиральная модель заставляет сначала исследовать риск, а не откладывать его до конца. Прототипы, эксперименты и архитектурные проверки становятся нормальной частью процесса.

Спиральная модель хорошо подходит для крупных, сложных и инновационных проектов, где нельзя заранее точно описать все требования и решения. Она позволяет постепенно уточнять систему, получать обратную связь и принимать решения на основе уже полученных результатов.

Недостатки тоже существенны. Модель сложнее в управлении, требует зрелой команды, хорошей оценки рисков и дисциплины планирования. Для маленьких типовых проектов она может быть избыточной. Если команда формально рисует «витки», но не анализирует риски, смысл модели теряется.

Для экзамена главное: спиральная модель - итерационная риск-ориентированная модель ЖЦ, где каждый цикл дает уточнение продукта и уменьшение ключевых неопределенностей.

Источники: SWEBOK Guide, гл. 8 и 9; Boehm, A Spiral Model of Software Development and Enhancement, 1988.

30. V-образная и инкрементная (пошаговая) модели жизненного цикла ПП

Недостаточно информации в предоставленных источниках: V-образная и инкрементная модели в локальных материалах не раскрыты.

V-образная модель похожа на водопадную, но специально показывает связь между этапами разработки и уровнями проверки. Левая ветвь V - уточнение и проектирование: требования, системная архитектура, детальный дизайн. Правая ветвь - тестирование: модульное, интеграционное, системное, приемочное. Каждому уровню разработки соответствует свой уровень проверки.

Смысл V-модели в том, что тестирование планируется не в конце, а параллельно с разработкой требований и проектных решений. Например, приемочные тесты связываются с пользовательскими требованиями, системные - с системной спецификацией, интеграционные - с архитектурой, модульные - с детальным дизайном. Это делает модель удобной для проектов, где важны трассируемость, регламенты и доказуемое качество.

Недостаток V-модели тот же, что у водопада: изменения требований обходятся дорого, а работающий продукт появляется поздно. Она лучше водопада с точки зрения проверки, но не всегда достаточно гибкая.

Инкрементная модель строит продукт частями. Сначала выбирают набор функций для первого инкремента, реализуют и поставляют его, затем добавляют следующие инкременты. Каждый инкремент расширяет уже существующую систему. Пользователь раньше получает полезный результат, а команда раньше получает обратную связь.

Преимущества инкрементной модели:

- ранняя поставка части функциональности;
- снижение риска за счет постепенной разработки;
- возможность менять приоритеты;
- более простая проверка отдельных частей.

Главный риск - архитектура должна выдерживать рост. Если первый инкремент сделан как временная заготовка без учета будущего развития, дальнейшие шаги становятся дорогими. Поэтому инкрементная разработка требует хорошей архитектурной основы и контроля технического долга.

Источники: SWEBOK Guide, гл. 8; ISO/IEC/IEEE 12207:2017, Abstract.

31. Модель быстрого прототипирования

Недостаточно информации в предоставленных источниках: модель быстрого прототипирования не описана в локальных презентациях.

Модель быстрого прототипирования используется, когда требования неясны, пользователю трудно заранее описать нужную систему или команда хочет проверить важное решение до полноценной разработки. Прототип - это упрощенная версия продукта или его части, созданная для уточнения требований, интерфейса, архитектурного риска или технической реализуемости.

Прототипирование бывает разным. Одноразовый, или throwaway, прототип создается для изучения задачи и затем выбрасывается. Его не нужно доводить до промышленного качества. Эволюционный прототип постепенно дорабатывается и может стать основой конечной системы. Горизонтальный прототип показывает широкий набор экранов или сценариев без глубокой реализации. Вертикаль-

ный прототип реализует небольшой кусок системы «в глубину», например полный путь от интерфейса до базы данных.

Преимущества быстрого прототипирования:

- ранняя обратная связь от пользователей;
- уточнение требований на видимом примере;
- снижение риска неудобного интерфейса;
- проверка технических гипотез;
- лучшее взаимопонимание между заказчиком и командой.

Недостатки связаны с ложным ощущением готовности. Заказчик может увидеть прототип и решить, что система почти сделана, хотя внутри нет надежности, безопасности, тестов и нормальной архитектуры. Еще один риск - использовать одноразовый прототип как промышленный код. Тогда временные решения попадают в продукт и создают технический долг.

Для экзамена главное: быстрое прототипирование помогает уточнить неизвестное до крупной разработки, но требует честного определения статуса прототипа - это эксперимент, демонстрация или будущая основа продукта.

Источники: SWEBOK Guide, гл. 1, 2 и 8.

32. Модель жизненного цикла MSF

Недостаточно информации в предоставленных источниках: локальная лекция раскрывает MSF в части команды и ролей, но модель жизненного цикла MSF описана не полностью.

MSF, или Microsoft Solutions Framework, - рамочный подход Microsoft к организации разработки и внедрения технологических решений. Он включает процессную модель, командную модель и дисциплину управления рисками. В курсе отдельно рассматривается MSF for Agile Software Development, но для понимания жизненного цикла полезно знать общую логику MSF.

Классическая процессная модель MSF строилась вокруг итераций и контрольных вех. Обычно выделяют фазы: выработка концепции, планирование, разработка, стабилизация и внедрение. На фазе концепции команда согласует видение продукта, бизнес-цели и границы решения. На планировании уточняются требования, архитектура, риски, график и ресурсы. На разработке создается решение. На стабилизации оно тестируется, исправляется и готовится к выпуску. На внедрении продукт передается в эксплуатацию.

Важная особенность MSF - не только последовательность фаз, но и командная организация. Модель делает акцент на общей ответственности, коммуникации, ориентации на заказчика, готовности к изменениям и постоянном управлении рисками. В локальной лекции управление программным проектом связывается с тремя задачами: команда, процесс, инструменты. Это хорошо совпадает с духом MSF.

MSF for Agile Software Development адаптирует эту логику к итерационной разработке. Работа идет короткими циклами, требования уточняются через сценарии, а роли распределяются так, чтобы покрыть управление продуктом, программой, разработку, тестирование, выпуск и пользовательский опыт.

Для экзамена можно сказать так: модель ЖЦ MSF - итерационная и веховая модель, где разработка решения проходит через видение, планирование, разработку, стабилизацию и внедрение, а управление рисками и командная ответственность действуют на протяжении всего цикла.

Источники: `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайды 2, 9-13; Microsoft Solutions Framework Core Whitepapers, MSF Team Model; SWEBOK Guide, гл. 7-8.

33. Модель жизненного цикла RUP

Недостаточно информации в предоставленных источниках: RUP в локальных материалах упоминается как методология, связанная с UML и Rational-средствами, но жизненный цикл RUP подробно не раскрыт.

RUP, Rational Unified Process, - итеративный процесс разработки ПО, тесно связанный с объектно-ориентированным подходом и UML. В локальных материалах показана связка: нотация UML, методология RUP, средство IBM Rational Rose или IBM Rational Software Architect. Это важно: UML дает язык моделей, а RUP задает процесс их использования.

Жизненный цикл RUP делится на четыре фазы:

1. Inception - начальная фаза. Определяются границы системы, бизнес-цели, основные участники, ключевые варианты использования, грубая оценка стоимости и рисков.
2. Elaboration - уточнение. Прорабатывается архитектура, уточняются требования, снимаются главные технические риски, создается архитектурная база.
3. Construction - построение. Основной объем функциональности реализуется, тестируется и интегрируется.
4. Transition - передача. Продукт вводится в эксплуатацию, исправляются остаточные дефекты, готовится документация, обучение и релиз.

Главная особенность RUP - итеративность внутри фаз. Это не простой водопад с четырьмя большими этапами. В каждой фазе могут быть итерации, которые дают проверяемые результаты. Кроме того, RUP описывает дисциплины: требования, анализ и проектирование, реализация, тестирование, развертывание, управление конфигурацией и изменениями, управление проектом, среда разработки.

RUP хорошо подходит для проектов, где нужны формальные модели, управляемая архитектура, трассируемость требований и контроль рисков. Недостаток - процесс может стать тяжелым, если команда пытается использовать все артефакты без адаптации под размер и риск проекта.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 13, 15; IBM Rational ClearQuest documentation, Project planning; SWEBOK Guide, гл. 8-9.

34. Модели ЖЦ при Agile-разработке ПС: модель XP

Недостаточно информации в предоставленных источниках: XP в локальных материалах не раскрыт.

XP, или Extreme Programming, - гибкая модель разработки, ориентированная на короткие циклы, постоянную обратную связь и высокую техническую дисциплину. Она возникла как реакция на тяжелые процессы, где команда поздно получала работающий продукт и плохо реагировала на изменение требований.

XP опирается на ценности: коммуникация, простота, обратная связь, смелость и уважение. Коммуникация нужна, чтобы разработчики, заказчик и тестировщики быстро уточняли смысл требований. Простота означает стремление реализовывать нужное сейчас, не усложняя систему гипотетическими будущими возможностями. Обратная связь достигается через тесты, короткие релизы и постоянное участие заказчика. Смелость нужна для рефакторинга и честного разговора о проблемах. Уважение удерживает команду от хаотичного индивидуализма.

К практикам XP обычно относят короткие релизы, планирование итераций, пользовательские истории, разработку через тестирование, парное программирование, коллективное владение кодом, непрерывную интеграцию, рефакторинг, простой дизайн, стандарты кодирования и постоянное присутствие представителя заказчика.

Сильная сторона XP - техническое качество. В отличие от поверхностного понимания Agile как «меньше документов», XP требует дисциплины: писать тесты, часто интегрировать код, улучшать дизайн, не накапливать дефекты. Это снижает стоимость изменений.

Слабые стороны связаны с требованиями к команде и заказчику. XP трудно внедрить, если нет доверия, автоматизированных тестов, готовности к частой коммуникации и культуры совместной работы. Парное программирование и коллективное владение кодом также подходят не любой организации.

Для экзамена: XP - Agile-модель с короткими итерациями и сильными инженерными практиками, направленная на быструю обратную связь и способность безопасно менять продукт.

Источники: Manifesto for Agile Software Development; SWEBOK Guide, гл. 8-9; Kent Beck, Extreme Programming Explained.

35. Модели ЖЦ при Agile-разработке ПС: модель SCRUM

Недостаточно информации в предоставленных источниках: Scrum в локальных презентациях не раскрыт.

Scrum - гибкий фреймворк для разработки сложных продуктов. Он не описывает все инженерные практики, как XP, а задает минимальную структуру ролей, событий и артефактов, через которые команда регулярно планирует, выполняет работу, проверяет результат и адаптируется.

В Scrum есть одна Scrum-команда, внутри которой выделяются три ответственности: Product Owner, Scrum Master и Developers. Product Owner отвечает за ценность продукта и Product Backlog. Scrum Master отвечает за понимание и применение Scrum, помогает команде и организации устранять препятствия. Developers создают инкремент продукта и сами организуют работу внутри спринта.

Работа идет спринтами - короткими фиксированными итерациями. Внутри спринта есть события:

- Sprint Planning - планирование цели и работы спринта;
- Daily Scrum - ежедневная синхронизация разработчиков;
- Sprint Review - демонстрация результата и получение обратной связи;
- Sprint Retrospective - анализ процесса и улучшения;
- сам Sprint - контейнер для всех этих событий.

Основные артефакты Scrum: Product Backlog, Sprint Backlog и Increment. У каждого есть обязательство: Product Goal, Sprint Goal и Definition of Done. Это нужно для прозрачности: команда и заинтересованные стороны должны понимать, что важно, что делается сейчас и когда результат считается готовым.

Преимущества Scrum - регулярная поставка инкрементов, прозрачность, быстрая обратная связь, адаптация планов. Ограничение - Scrum не гарантирует технического качества сам по себе. Если команда не использует тестирование, CI, ревью и архитектурную дисциплину, спринты могут просто ускорить накопление технического долга.

Источники: The Scrum Guide 2020; Manifesto for Agile Software Development; SWEBOK Guide, гл. 8.

36. Модели ЖЦ при Agile-разработке ПС: модель Kanban

Недостаточно информации в предоставленных источниках: Kanban в локальных материалах не раскрыт.

Kanban - подход к управлению потоком работ. В разработке ПО его используют, чтобы визуализировать работу, ограничивать незавершенные задачи и постепенно улучшать процесс. В отличие от Scrum, Kanban не требует фиксированных спринтов и заданных ролей. Он чаще начинается с существующего процесса команды и делает его видимым.

Базовая идея Kanban - работа проходит через поток состояний: например, backlog, анализ, разработка, ревью, тестирование, готово. Состояния отображаются на доске. Каждая задача движется по доске слева направо. Команда ограничивает WIP, то есть количество незавершенной работы в отдельных колонках или во всем потоке. Это защищает команду от ситуации, когда начато много задач, но мало что доведено до конца.

В официальном Kanban Guide выделяются три практики:

1. Определить и визуализировать workflow.
2. Активно управлять элементами в workflow.
3. Улучшать workflow.

Ключевые метрики Kanban: WIP, throughput, cycle time и work item age. WIP показывает объем начатой, но не завершенной работы. Throughput - сколько элементов завершается за период. Cycle time - сколько времени задача проходит от старта до завершения. Work item age - возраст незавершенной задачи. Эти метрики помогают видеть узкие места и прогнозировать поставку.

Kanban хорошо подходит для сопровождения, DevOps, продуктовых команд с непрерывным потоком задач и ситуаций, где сложно планировать работу спринтами. Его слабость - он не задает сам по себе продуктовую цель, роли и инженерные практики. Если просто повесить доску без WIP-лимитов и анализа потока, это будет визуальный список задач, а не Kanban.

Источники: The Kanban Guide, May 2025 и July 2020; Manifesto for Agile Software Development; SWEBOK Guide, гл. 7-8.

Команда проекта и MSF

37. Управление командой проекта: ролевая модель команды

Управление программным проектом в материалах курса связывается с тремя задачами: подбор и управление командой, выбор процесса и выбор инструментальных средств. Ролевая модель команды нужна, чтобы заранее определить зоны ответственности и не полагаться на неформальное «кто-нибудь сделает».

В лекции выделены роли: менеджер проекта, проектировщик, разработчик, технолог разработки ПО, тестировщик, инженер по качеству, технический писатель.

Менеджер проекта отвечает за управление кадрами, подготовку и исполнение плана, руководство командой, связь между подразделениями и готовность продукта. Проектировщик отвечает за архитектуру высокого уровня, основные интерфейсы и контроль выполнения архитектурных решений. В небольших командах эта функция может распределяться между менеджером и разработчиками, в больших - выделяться в отдельную группу.

Разработчик создает конечный продукт, но его функции шире кодирования. Он контролирует архитектурные и технические спецификации, выбирает инструменты и стандарты, решает технические проблемы, следит за состоянием продукта и обнаруженными ошибками.

Тестировщик отвечает за удовлетворение функциональных и нефункциональных требований. Он составляет план тестирования, ведет базу ошибок, разрабатывает тесты, организует автоматизацию и бета-тестирование. Инженер по качеству работает не только с продуктом, но и с процессами: составляет план качества, описывает процессы, вводит метрики, оценивает ход процесса и предлагает улучшения.

Технический писатель отвечает за пользовательскую и иную документацию: план документирования, стандарты, шаблоны, средства автоматизации, разработку и тестирование документации. Технолог разработки ПО поддерживает модель ЖЦ, среду сборки, процедуру установки и управление исходными текстами.

Главная мысль: роли не обязательно равны отдельным людям. В малом проекте один человек может совмещать несколько ролей, но ответственность все равно должна быть явно закрыта.

Источники: `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайды 2-4.

38. Модели организации команд: административная модель, модель хаоса, открытая архитектура

В материалах курса модели организации команд связаны с человеческим фактором. Люди в проекте одновременно разные и похожие: различаются характером, активностью, ответственностью, отношением к команде, но объединяются общей целью. Поэтому одна и та же методология может сработать в одном проекте и провалиться в другом.

Административная модель, или теория X, - иерархический командно-административный стиль управления. Ее девиз в лекции: «Люди делают только то, что вы контролируете». В такой модели сильна вертикаль власти, контроль, назначение задач сверху, формальные правила. Она может быть уместна в жестко регламентированных условиях, но плохо работает там, где нужна инициатива, творческое решение проблем и быстрая адаптация.

Модель хаоса, или теория Y, исходит из другой установки: «Работа - естественная и приятная деятельность». Здесь предполагается, что люди способны к самоорганизации, если понимают цель и имеют свободу действий. Такая модель может давать сильную мотивацию и инициативу, но без минимальных правил легко превращается в неуправляемость: цели размываются, ответственность неясна, решения конфликтуют.

Открытая архитектура, или теория Z, в лекции формулируется девизом: «Работаем спокойно. Работаем вместе». Это попытка соединить дисциплину и доверие. Команда строится на совместной ответственности, устойчивой коммуникации, уважении к ролям и готовности помогать друг другу.

Контроль остается, но не как микроменеджмент, а как прозрачность работы и согласованность целей.

Для экзамена важно не противопоставлять модели как «плохую» и «хорошую». Административная модель дает управляемость, но подавляет инициативу. Хаос дает свободу, но рискует потерять контроль. Открытая архитектура стремится к балансу: команда самоорганизуется, но работает в понятных рамках процесса, ролей и общей цели.

Источники: `Программная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайды 5-8.

39. Модель проектной группы MSF for Agile Software Development: основные принципы построения команды

Модель проектной группы MSF for Agile Software Development строится вокруг идеи команды равноправных ролей, которые совместно отвечают за успех решения. Это не означает, что в команде нет управления. Смысл в другом: разные роли представляют разные интересы проекта, и ни одна из них не должна полностью подавлять остальные.

В лекции показаны ролевые группы и роли MSF. Менеджер проекта отвечает за соблюдение ограничений и координацию процесса так, чтобы нужный продукт был выпущен в нужное время. Менеджер продукта представляет интересы заказчика и пользователей, помогает находить компромисс между функциональностью, сроками и финансированием. Релиз-менеджер отвечает за развертывание и сопровождение, понимает инфраструктуру и требования эксплуатации. Разработчик создает техническое решение. Тестировщик отвечает за проверку качества. Архитектор отвечает за технологические цели и архитектурные решения.

Основные принципы построения команды MSF можно сформулировать так:

- общая ответственность за результат;
- ориентация на потребности заказчика и пользователей;
- распределение ролей по целям, а не только по должностям;
- открытая коммуникация между ролями;
- баланс интересов продукта, сроков, качества, разработки и эксплуатации;
- масштабируемость: роли можно объединять в малой команде и разделять в большой;
- постоянное управление рисками.

Важная деталь: роли могут совмещаться, но цели ролей должны оставаться представленными. Если в проекте нет человека, который думает об эксплуатации, продукт могут написать, но плохо внедрить. Если нет сильного представления интересов пользователя, команда может успешно реализовать ненужные функции.

MSF полезен тем, что заставляет смотреть на проект не только глазами разработчика или менеджера. Успех решения зависит от продукта, программы, разработки, тестирования, выпуска, пользовательского опыта и внешних групп.

Источники: `Программная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайды 9-13; Microsoft Solutions Framework Core Whitepapers, MSF Team Model.

40. Управление рисками в MSF

Недостаточно информации в предоставленных источниках: в локальной лекции риск упоминается в контексте MSF и причин провалов, но подробный процесс управления рисками в MSF не дан.

В MSF риск понимается как неопределенное событие или условие, которое может отрицательно повлиять на проект. Примеры: ключевой специалист уходит из команды, технология не выдерживает нагрузку, требования меняются поздно, интеграция с внешней системой задерживается, поставщик не выполняет обязательства.

MSF делает акцент на проактивном управлении рисками. Это значит, что команда не ждет, пока проблема случится, а регулярно ищет риски, оценивает их и планирует действия. В старых материалах MSF Risk Management Discipline обычно описывается пятишаговый цикл:

1. Identify - выявить риски.

2. Analyze - оценить вероятность, влияние и приоритет.
3. Plan - подготовить стратегии смягчения и запасные планы.
4. Track/Control - отслеживать состояние рисков и выполнять действия.
5. Learn - извлекать уроки из сработавших и несработавших мер.

В управлении рисками важны два вида действий. Mitigation снижает вероятность или влияние риска до его наступления: например, сделать технический прототип, добавить автоматические тесты, обучить второго специалиста. Contingency - план на случай, если риск все же реализовался: например, переключиться на другой сервис или сократить объем релиза.

В MSF риск не является личной виной. Наоборот, раннее обнаружение риска считается полезным: команда получает меньше неприятных сюрпризов. Риск должен иметь владельца, описание, вероятность, влияние, приоритет, план действий и статус. Документ или реестр рисков пересматривается на контрольных точках проекта.

Связь с локальными материалами прямая: игнорирование рисков и отсутствие процедур управления рисками названы среди причин неудачных проектов. Поэтому MSF рассматривает управление рисками как постоянную дисциплину, а не разовую таблицу в начале проекта.

Источники: `1 базовые принципы UML-2024.pptx`, слайд 2; `Програмная_инженерия_лекция_4_ЧАСТЬ_2025.pptx`, слайды 9-13; Microsoft Solutions Framework Risk Management Discipline v1.1.

UML

41. Унифицированный язык моделирования UML. Диаграмма вариантов использования, назначение, применение. Примеры

UML, или Unified Modeling Language, в материалах курса определяется как унифицированный язык моделирования для описания, визуализации и документирования объектно-ориентированных систем в процессе анализа и проектирования. UML предоставляет стандартный способ описания как концептуальных аспектов системы, например бизнес-процессов и функций, так и более конкретных аспектов: выражений языков программирования, схем баз данных, компонентов ПО.

Важно: UML не является методологией, процессом, языком программирования или формальным языком. В лекции это сформулировано коротко: UML = нотация + семантика. То есть UML дает графические обозначения и смысл этих обозначений, но не диктует сам процесс разработки.

Диаграмма вариантов использования показывает отношения между актерами и вариантами использования. Актер - внешняя по отношению к системе роль: пользователь, другая система, устройство или организация. Вариант использования - внешняя спецификация последовательности действий, которую система выполняет при взаимодействии с актером и которая дает актеру значимый результат.

Цели диаграммы вариантов использования:

- определить границы и контекст предметной области;
- сформулировать общие требования к функциональному поведению системы;
- создать исходную концептуальную модель для дальнейшей детализации;
- подготовить документацию для взаимодействия разработчиков, заказчиков и пользователей.

Основные отношения на диаграмме: ассоциация между актером и вариантом использования, включение `include`, расширение `extend`, обобщение актеров или вариантов использования. Ассоциация показывает участие актера в сценарии. `include` выносит общий обязательный фрагмент поведения. `extend` добавляет необязательное или условное поведение. Обобщение показывает наследование свойств более общей роли или сценария.

Пример: для интернет-магазина акторы «Покупатель», «Администратор» и «Платежная система» связаны с вариантами использования «Оформить заказ», «Оплатить заказ», «Управлять каталогом», «Просмотреть статус доставки». Такая диаграмма не описывает внутренний алгоритм, а показывает, какие цели пользователей должна поддерживать система.

Источники: `1 базовые принципы UML-2024.pptx`, слайды 18-20, 24-25; `2_1_Типы_диаграмм_в_UML_Диаграмма_вариантов_использования_2025.pptx`, слайды 4-15; OMG UML 2.5.1 Specification.

42. Унифицированный язык моделирования UML. Диаграмма состояния: назначение, применение. Примеры

Диаграмма состояний, или state machine diagram, предназначена для описания поведения элемента модели в течение его жизненного цикла. В материалах курса прямо сказано: она применяется, чтобы объяснить, как работают сложные объекты и как они реагируют на события.

Диаграмма состояний представляет конечный автомат в виде ориентированного графа. Вершины графа - состояния, дуги - переходы. Состояние описывает ситуацию в жизненном цикле объекта, в течение которой выполняется некоторое условие инварианта. Переход - направленное отношение между исходным и целевым состоянием, которое срабатывает при событии, условии или завершении деятельности.

Для конечного автомата в лекции указаны обязательные условия: автомат не запоминает историю перемещения, если не используются специальные исторические состояния; в каждый момент находится в одном состоянии; количество состояний конечно; граф не должен иметь изолированных состояний и переходов; не должно быть конфликтующих переходов.

Основные элементы диаграммы: начальное состояние, конечное состояние, простое состояние, составное состояние, переход, выбор, соединение, разделение, слияние, историческое состояние. Простое состояние может иметь внутренние действия `entry`, `exit`, `do`, `include`, `defer`. `entry` выполняется при входе, `exit` - при выходе, `do` - пока объект находится в состоянии.

Составные состояния позволяют описывать вложенное поведение. Подсостояния могут быть последовательными или параллельными. Историческое состояние используется, чтобы запомнить, где объект находился до выхода из составного состояния; это полезно при обработке прерываний без потери выполненной работы.

Пример: заказ в интернет-магазине может иметь состояния «Создан», «Ожидает оплаты», «Оплачен», «Собирается», «Отправлен», «Доставлен», «Отменен». Переходы запускаются событиями: оплата получена, товар передан в доставку, заказ отменен, срок оплаты истек.

Источники: `3 Диаграмма состояний_2025.pptx`, слайды 2-29; OMG UML 2.5.1 Specification.

43. Унифицированный язык моделирования UML. Диаграмма деятельности: назначение, применение. Примеры

Недостаточно информации в предоставленных источниках: отдельной презентации по диаграмме деятельности в папке нет. В материалах есть только косвенные упоминания действий, сигналов и связи с диаграммами состояний.

Диаграмма деятельности UML описывает поток работ или алгоритм: какие действия выполняются, в каком порядке, где есть ветвления, параллельность, синхронизация и завершение. Ее удобно использовать для моделирования бизнес-процессов, сценариев использования, процедур обработки данных и логики операций.

Главный элемент диаграммы - действие. Действие выполняет некоторую работу: проверить данные, рассчитать сумму, отправить уведомление, сохранить запись. Действия соединяются потоками управления. Начальный узел показывает старт процесса, конечный - завершение. Ветвление выбирает один из путей по условию, слияние собирает альтернативные пути обратно. Разделение запускает параллельные ветви, синхронизация ждет завершения нескольких параллельных ветвей.

Диаграмма деятельности может также показывать потоки объектов: какие данные создаются, передаются и используются действиями. Для распределения ответственности применяют дорожки, или swimlanes: например «Пользователь», «Система», «Платежный сервис», «Склад». Это помогает увидеть, кто выполняет каждое действие и где процесс пересекает границы систем или подразделений.

Пример для оформления заказа: пользователь добавляет товары в корзину, система проверяет наличие, пользователь вводит адрес, система рассчитывает доставку, пользователь выбирает оплату, платежный сервис подтверждает платеж, система создает заказ и отправляет уведомление. Если оплаты нет, процесс уходит в ветвь ошибки или ожидания.

Диаграмма деятельности похожа на блок-схему, но имеет UML-семантику и лучше описывает параллельность, ответственность и связь с объектами. Для экзамена главное: она показывает не статическую структуру, а поток действий и решений.

Источники: `3 Диаграмма состояний_2025.pptx`, слайды 10-11, 19; OMG UML 2.5.1 Specification.

44. Унифицированный язык моделирования UML. Диаграмма последовательности: назначение, применение. Примеры

Недостаточно информации в предоставленных источниках: отдельной презентации по диаграмме последовательности в папке нет.

Диаграмма последовательности UML описывает взаимодействие объектов или участников во времени. Она показывает, кто кому посылает сообщения, в каком порядке и какие ответы получает. Если диаграмма классов отвечает на вопрос «из чего состоит система», то диаграмма последовательности отвечает на вопрос «как участники взаимодействуют при выполнении конкретного сценария».

Основные элементы диаграммы: участники взаимодействия, линии жизни, фокусы управления и сообщения. Линия жизни показывает существование объекта или роли во времени. Сообщения

изображаются стрелками между линиями жизни: синхронный вызов, асинхронное сообщение, ответ, создание или уничтожение объекта. Время обычно читается сверху вниз.

Диаграммы последовательности применяются для детализации вариантов использования, проектирования интерфейсов между компонентами, обсуждения протоколов обмена, поиска ответственности классов и проверки архитектурных решений. Они особенно полезны, когда важно увидеть порядок вызовов: авторизация, оформление заказа, платеж, обработка ошибки, запрос к внешнему сервису.

Пример для входа пользователя: пользователь вводит логин и пароль, интерфейс отправляет запрос контроллеру, контроллер обращается к сервису авторизации, сервис получает пользователя из репозитория, проверяет пароль, создает токен и возвращает результат интерфейсу. На диаграмме можно показать альтернативу: если пароль неверный, система возвращает ошибку; если верный - открывает личный кабинет.

На диаграммах последовательности часто используют комбинированные фрагменты: `alt` для альтернатив, `loop` для циклов, `opt` для необязательного поведения, `par` для параллельных ветвей. Это позволяет не рисовать несколько диаграмм для небольших вариантов одного сценария.

Для экзамена главное: диаграмма последовательности - поведенческая диаграмма взаимодействия, которая фиксирует временной порядок сообщений между участниками сценария.

Источники: OMG UML 2.5.1 Specification; SWEBOK Guide, гл. 2 и 9.

45. Унифицированный язык моделирования UML. Диаграммы реализации: назначение, применение. Примеры

Диаграммы реализации в UML показывают физическое представление модели. В материалах курса они определены как совокупность связанных физических сущностей, включая программное и аппаратное обеспечение. К диаграммам реализации относятся диаграмма компонентов и диаграмма развертывания.

Диаграмма компонентов - диаграмма физического уровня, которая представляет программные компоненты и зависимости между ними. Компонент - модульная часть системы с инкапсулированным содержимым; его спецификация должна быть взаимозаменяемой в окружении. Компонент может предоставлять интерфейсы и требовать интерфейсы других компонентов. Предоставляемый интерфейс показывает, какие сервисы компонент дает наружу, требуемый - какие сервисы ему нужны.

На диаграмме компонентов показывают компоненты, интерфейсы, зависимости, реализации, порты, собирающие и делегирующие соединители. Собирательный соединитель связывает требуемый интерфейс одного компонента с предоставляемым интерфейсом другого. Делегирующий соединитель связывает внешний контракт компонента с внутренними частями, которые реализуют это поведение. Пример: компонент «Веб-приложение» требует интерфейс «Оплата», который предоставляет компонент «Платежный шлюз».

Диаграмма развертывания показывает узлы выполнения программных компонентов реального времени, процессов и объектов. Узел - физически существующий элемент системы, обладающий вычислительным ресурсом или являющийся техническим устройством: сервер, рабочая станция, датчик, принтер, модем. На диаграмме также показывают соединения между узлами, зависимости, размещенные компоненты и артефакты.

Артефакт - физически существующая часть информации, используемая или производимая при разработке или эксплуатации: исполняемый файл, библиотека, архив, конфигурация, скрипт, таблица БД. Пример диаграммы развертывания: браузер пользователя соединен с веб-сервером, веб-сервер - с сервером приложений, сервер приложений - с сервером БД и внешним платежным сервисом.

Назначение диаграмм реализации - связать логическую архитектуру с физической поставкой и эксплуатацией: какие компоненты есть, от чего они зависят, где они выполняются и какие артефакты должны быть развернуты.

Источники: 6 Диаграмма компонентов 2025.pptx, слайды 2-16; 7 Диаграмма развертывания 2025.pptx, слайды 2-11; OMG UML 2.5.1 Specification.