**СПИСОК ВОПРОСОВ**

«Программная инженерия»

для студентов очной формы обучения

1. **ПОНЯТИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ** - инженерная дисциплина, которая связана со всеми аспектами производства ПО от начальных стадий создания спецификации до интеграции и поддержки. **История -** Программная инженерия (технология промышленного программирования) как направление возникло и формировалось под давлением роста стоимости создаваемого программного обеспечения. Главная цель этой области знаний – сокращение стоимости и сроков разработки программы. **Модульное программирование –** высокая стоимость ПО была связана с разработкой одинаковых фрагментов кода в различных программах. Главный принцип модульного программирования состоял в выделении таких фрагментов и оформлении их в виде отдельных модулей. Каждый модуль снабжался описанием, в котором устанавливались правила его использования – интерфейс модуля. Интерфейс задавал связи модуля с основной программой – связи по данным, связи по управлению
2. **Рост сложности программ** - связан с переходом от разработки относительно простых программ к разработке программных комплексов. Следует отметить, что этот переход был вызван появлением вычислительной техники 3-го поколения. С переходом на использование интегральных схем производительность компьютеров возросла на порядки, что и создало предпосылки на решение разных сложных задач**.** *Сложные задачи*: система управления космическими объектами; управление оборонным комплексом; автоматизация финансовых учреждений и т.д.*Сложность можно оценить:* большой объём кода; большое кол-во связей м/у элементами кода; большое кол-во разработчиков и пользователей; длительное время использования. Для таких сложных задач оказалось, что основная часть их стоимости приходится не на создание программ, а на их внедрение и эксплуатацию. **Статистика успешности. Причин неудач.** (найти в интернете/придумать)
3. **Структурное программирование -** парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков. 3 структуры управления: последовательность, ветвление и повтор. **Основные принципы:**
4. Нисходящее функциональное проектирование
5. Применение специальных языков проектирования и средств автоматизации использования этих языков
6. Дисциплина проектирования и разработки
7. **Процедурно функциональное** (разные методы) **программирование.** Процедурное – позволяется изменение переменных внутри функции. Функциональное – выполнение функции без изменения состояния**. Основные принципы.** *Процедурное* – последовательность выполнения, использование переменных, код организуется в процедуры и функции, модификация состояния. *Функциональное* – чистые функции (Функции не изменяют состояние и не имеют побочных эффектов), иммутабельность (данные не изменяются после их создания), функции высшего порядка (функции могут принимать другие функции), ленивые вычисления (вычисления откладываются, пока результат не понадобится).
8. **Объектно-ориентированное программирование** - подход, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение. **Основные принципы:** 
   1. Абстракция. Моделирование требуемых атрибутов и взаимодействий сущностей в виде классов для определения абстрактного представления системы.
   2. Инкапсуляция. Скрытие внутреннего состояния и функций объекта и предоставление доступа только через открытый набор функций.
   3. Наследование. Возможность создания новых абстракций на основе существующих.
   4. Полиморфизм. Возможность реализации наследуемых свойств или методов отличающимися способами в рамках множества абстракций.
9. **Требования к ПО (функциональные, нефункциональные) –** Функциональные – удовлетворяют требованиям заказчика. Нефункциональные –
   1. Отказоустойчивость – возможность восстановления программы и данных в случае сбоя в работе.
   2. Безопасность – сбои в работе в программе не должны приводить к опасным последствиям. Защищённость от случайных и преднамеренных внешних воздействий.
   3. Эффективность – программное обеспечение не должно впустую тратить системные ресурсы.
   4. Удобство использования – именно тем типом пользователей, на которых рассчитано приложение
   5. Сопровождаемость – означает, что программа должна быть написана с расчётом на дальнейшее развитие.
10. **Кризис программирования -** В конце 1960-х, в начале 1970-х произошло событие, которое вошло в историю как первый кризис программирования. Событие состояло в том, что стоимость программного обеспечения стало приближаться к стоимости железа, а динамика роста этой стоимости позволяла прогнозировать, что к середине 90-х годов всё человечество будет заниматься разработкой ПО.
11. **Программная инженерия и программный продукт -** Программный продукт – это не только программы, а также вся связанная с ними документация и данные, необходимые для корректной работы программы. В зависимости от того, для кого разрабатываются программные продукты, бывают двух типов:

* Коробочные продукты
* Заказные продукты

Программная инженерия – инженерная дисциплина, которая связана со всеми аспектами производства ПО от начальных стадий создания спецификации до интеграции и поддержки

1. **Отличие информатики от программной инженерии -** Информатика (computer science) занимается теорией и методами вычислительных и программных систем, в то время как ПИ занимается практическими проблемами создания ПО. Информатика составляет теоретические основы ПИ, и инженер по ПО должен знать информатику. Программные инженеры зачастую используют приёмы, которые применимы только в конкретных условиях и не могут быть обобщены, а теории информатики не всегда могут быть применены к большим системам.
2. **Стоимость ПО. Распределение стоимости по основным этапам. -** Структура стоимости ПО существенно зависит от типа ПО, применяемых методов его разработки. Многие отмечают высокую долю стоимости этапа сопровождения (60% и более). Этап сопровождения включает выполнение двух видов работ: внесение изменений в программу, исправление ошибок в программе (несоответствие первоначальным требованиям).

Типовое распределение стоимости между основными этапами:

1. 15% - спецификация – формулировка требований и условий разработки
2. 25% - проектирование – разработка и верификация проекта
3. 20% - разработка – кодирование и тестирование
4. 40% - интеграция и тестирование – объединение и сборочное тестирование продукта

Для коробочного ПО характерна более высокая доля тестирования, но за счёт сокращения доли спецификации.

1. **Коробочное** - программное обеспечение с базовым набором функций, созданное для широкого круга клиентов **и индивидуальное –** программное обеспечение, которое разрабатывается специально для определенной организации или другого пользователя **ПО**
2. **Разные виды ПО и подходы программной инженерии к ним (web –** модульная архитектура, agile и scrum, devops**, системное ПО –** модульная разработка, тестирование на уровне системы, Методы разработки на основе спецификаций**, прикладное ПО -** User-Centered Design (UCD), Rapid Application Development, **мобильные приложения** - Кроссплатформенная разработка: Использование фреймворков (например, React Native, Flutter) для создания приложений, работающих на разных платформах. Agile методологии: Быстрая итерация и адаптация к изменениям в требованиях пользователей. **Игровое ПО** - Agile и Scrum: Гибкие методологии для быстрой адаптации к изменениям в игровом процессе или механике. Прототипирование: Быстрое создание прототипов игровых механик для тестирования идей.)
3. **ОС (webOS, Windows, Linux, MacOS и др.). Сравнение и анализ применительно к ПО** (Свободный ответ)**.**
4. **Машинный код** - набор инструкций, который может быть непосредственно выполнен процессором компьютера. Он состоит из двоичных данных (нулей и единиц) и зависит от архитектуры конкретного процессора (например, x86, ARM)**.** **Языки программирования и среды разработки IDE** - приложения, которые предоставляют разработчикам инструменты для написания, тестирования и отладки кода в одном интерфейсе (VSCode, Eclipse, PyCharm)**. История, развитие, перспективы. Низкий уровень** (ассемблер и машинный код) **и высокий** (python, java, C++, C#**, уровни абстракций** *низкий* – низкоуровневые языки предоставляют непосредственный контроль над аппаратным обеспечением. Позволяют оптимизировать производительность программ, но требуют глубоких знаний о системе**.**

*Высокий уровень* - Высокоуровневые языки программирования скрывают детали работы аппаратного обеспечения и предоставляют более удобные конструкции для разработки. Обеспечивают большую продуктивность и позволяют сосредоточиться на решении задач, а не на деталях реализации.

1. **Базы данных** - некоторый набор перманентных (постоянно хранимых) данных, используемых прикладными программными системами какого-либо предприятия**.**
2. **Синхронная и асинхронная программные модели** - *Синхронные* функции выполняют весь запрос, прежде чем потоку вызывающего объекта будет разрешено вернуться из вызова функции. *Асинхронные* функции возвращаются до выполнения запроса в полном объеме.
3. **Однопоточность** (задачи выполняются по порядку) **и многопоточность** (задачи выполняются одновременно, распределившись по потокам)**. Параллелизм -** способ обработки множественных запросом одновременно
4. **Библиотека** - коллекция предопределённых функций и процедур, которые разработчики могут использовать для выполнения общих задач.**, API** - интерфейс, который определяет, как различные программные компоненты взаимодействуют друг с другом. Он предоставляет набор правил и протоколов для создания и интеграции программного обеспечения.**, Фреймворк** - более комплексная структура, которая предоставляет основу для разработки приложений. Он определяет архитектуру приложения и включает в себя библиотеки и API, а также инструменты для разработки. **Разница** - библиотека — это часть приложения. Фреймворк — это скелет, API – внешние части указанного приложения. Фреймворк, ко всему прочему, в отличии от API использует инверсию управления
5. **Фронтенд** - клиентская часть продукта (интерфейс, с которым взаимодействует пользователь), **бэкенд** – программно-аппаратная часть веб-приложения, которую не видят пользователи: они не знают о работе внутренних процессов и не могут на них влиять. Бэкенд находится на сервере — мощном компьютере, который отвечает за хранение данных и обработку поступающих запросов **и фулстэк** – делает всё**.**
6. **Техники программирования (парное, пинг-понг, удаленное и т.д.).** *Парное* - практика, при которой два разработчика работают вместе за одним компьютером. Один из них (водитель) пишет код, а другой (наблюдатель) наблюдает, комментирует и предлагает улучшения*. Пинг-понг* – парное со сменой ролей. *Удалённое –* сотрудничество между разработчиками, которые находятся в разных географических локациях. *Код-ревью* - процесс проверки кода другими разработчиками перед его слиянием в основную ветку проекта.
7. **Принципы для разработки: KISS** – системы как можно проще**, DRY** – код не должен повторяться**, YAGNI** - разработчики не должны добавлять функциональность до тех пор, пока она не станет необходимой**, BDUF** - необходимо создать детальный проект всей системы**, SOLID** – одна ответственность класса, классы открыты для расширения, но закрыты для модификации, Объекты должны быть заменяемы экземплярами их подтипов без нарушения корректности программы, Клиенты не должны зависеть от интерфейсов, которые они не используют, Зависимости должны зависеть от абстракций, а не от конкретных реализаций**, APO** - оптимизация должна происходить только после того, как код уже работает
8. **Шаблон проектирования (паттерны) в разработке программного обеспечения** - повторяемые решения общих проблем, возникающих в процессе проектирования программного обеспечения. Они помогают разработчикам создавать более гибкие, поддерживаемые и понятные системы. Паттерны проектирования можно разделить на три основных категории: порождающие, структурные и поведенческие.
9. **архитектура ПО () -** высокоуровневое описание системы, которое определяет её основные компоненты, их взаимодействие и принципы организации. Она служит основой для проектирования и реализации программных систем и включает в себя как структурные, так и поведенческие аспекты.
10. **Виды архитектур приложений.** *Монолитная архитектура* – вся функциональность приложения объединена в одном едином модуле. *Микросервисная архитектура* - Приложение разбивается на независимые сервисы, каждый из которых отвечает за свою функциональность. *Сервисно-ориентированная архитектура* - Подход, основанный на использовании сервисов, которые взаимодействуют друг с другом через стандартизированные протоколы. *Клиент-серверная архитектура* - Разделение приложения на клиентскую часть (интерфейс пользователя) и серверную (логика обработки данных). *Архитектура на основе событий* - Приложение реагирует на события и изменения состояния, используя асинхронное взаимодействие.
11. **Методы программной инженерии** Процесс разработки, Моделирование, Управление требованиями, Проектирование, Кодирование, Тестирование, Сопровождение**. Компоненты метода. Case технологии** - набор инструментов и методов программной инженерии для проектирования программного обеспечения, которые помогают обеспечить высокое качество программ, отсутствие ошибок и простоту в обслуживании программных продуктов. **и Case средства** – upper case (Инструменты для ранних этапов разработки (анализ требований, проектирование)), Lower CASE - Инструменты для последующих этапов (кодирование, тестирование)**.**
12. **Понятие программного обеспечения и программного продукта** [(пункт 8)](#ПИ)
13. **Профессиональные** - Знания и навыки, Образование и квалификация, Опыт работы, Коммуникационные навыки, Постоянное обучение **и этические требования** - Честность и добросовестность, Конфиденциальность, Ответственность, Уважение к коллегам и клиентам, Социальная ответственность, Соблюдение законодательства **к специалистам.**
14. **Стандарты программной инженерии (корпоративные, отраслевые**- разрабатываются профессиональными ассоциациями или организациями в рамках конкретной отрасли**, государственные** - могут быть разработаны правительственными органами и могут касаться специфических требований к программному обеспечению, особенно в таких областях, как безопасность и защита данных**, международные** - разрабатываются международными организациями и могут быть применены в разных странах**.)**
15. **Организации международной стандартизации** - ISO (International Organization for Standardization): Разработка стандартов в области управления качеством (ISO 9001) и управления информационной безопасностью (ISO/IEC 27001).

• IEC (International Electrotechnical Commission): Стандарты для электротехнических и электронных систем.

• ISO/IEC 25010: Стандарт для оценки качества программного обеспечения.

1. **Жизненный цикл программного продукта** - процесс, который описывает все стадии, через которые проходит программное обеспечение от идеи до его снятия с эксплуатации**. История** жизненного цикла программного продукта началась с первых компьютерных систем в 1950-х годах. В то время разработка программного обеспечения была неформальной и чаще всего зависела от индивидуальных программистов. С увеличением сложности систем возникла необходимость в более структурированном подходе.

* 1970-е годы: Появление первых моделей жизненного цикла, таких как модель «водопад» (Waterfall), предложенная Уинстоном Ройсом в 1970 году. Эта модель описывала линейный процесс разработки, где каждая стадия должна быть завершена перед переходом к следующей.
* 1980-е - 1990-е годы: Разработка новых моделей, таких как спиральная модель и модели итеративной разработки, которые позволили более гибко подходить к изменяющимся требованиям.
* 2000-е годы: Внедрение Agile-методологий, которые акцентируют внимание на гибкости, взаимодействии с клиентом и быстрой поставке рабочего ПО. **Проблемы** - Неопределенные требования, Коммуникация, Управление изменениями, Качество продукта, Поддержка и обслуживание, Снятие с эксплуатации**.**

1. **Основные** - Сбор требований, Проектирование, Разработка, Тестирование, Внедрение, Эксплуатация, Поддержка и обслуживание, Снятие с эксплуатации **и вспомогательные процессы жизненного цикла** - Управление проектом, Управление качеством, Управление конфигурацией, Управление рисками**.**
2. **Языки Программирования**- формальные языки, используемые для написания программ, которые выполняют определенные задачи на компьютерах**. Среды разработки** – [(пункт 14)**.**](#ЯП)
3. **Каскадная (водопадная) модель жизненного цикла ПО** – процесс разработки ПО идёт строго по [пункт 31 “Основные”.](#Основные_процессы) Невозможно вернуться на предыдущий этап, либо пропустить один из.
4. **Спиральная модель жизненного цикла ПО ­**- подход к разработке ПО, который сочетает в себе элементы итеративного и каскадного подходов. Она была предложена Бэрри Бёмом в 1986 году и акцентирует внимание на управлении рисками и поэтапной разработке.

1. Планирование: Определение целей, сбор и анализ требований.

2. Анализ рисков: Выявление и оценка потенциальных рисков, разработка мер по их снижению

3. Разработка и тестирование: Проектирование, реализация и тестирование функциональности.

4. Оценка: Анализ результатов итерации и сбор обратной связи от пользователей.

1. **V-образная модель жизненного цикла ПО** - подход к разработке ПО, который подчеркивает важность тестирования на каждом этапе разработки. Модель визуализируется в виде буквы "V", где левая сторона представляет собой этапы проектирования и разработки, а правая — этапы тестирования и валидации.

Основные этапы:

1. Анализ требований: Сбор и документирование требований от заказчика.

2. Проектирование системы: Создание архитектуры и дизайна системы.

3. Проектирование модулей: Разработка детального проектирования отдельных компонентов.

4. Кодирование: Реализация программы на основе проектной документации.

На правой стороне "V" происходит:

5. Модульное тестирование: Проверка отдельных компонентов на соответствие требованиям.

6. Интеграционное тестирование: Тестирование взаимодействия между модулями.

7. Системное тестирование: Оценка всей системы в целом.

8. Приемочное тестирование: Проверка соответствия системы требованиям заказчика.

▎Преимущества:

• Четкая структура и последовательность этапов.

• Раннее выявление ошибок через тестирование на каждом этапе.

• Улучшенная документация.

▎Недостатки:

• Меньшая гибкость к изменениям в требованиях.

• Может быть неэффективной для небольших проектов.

V-образная модель подходит для проектов с четкими требованиями и высоким уровнем ответственности за качество.

1. **Инкрементальная модель жизненного цикла ПО** - подход к разработке, при котором система создается и развивается поэтапно, с добавлением новых функций и улучшений в каждом инкременте. Вместо того чтобы разрабатывать весь продукт сразу, команда работает над его частями, которые постепенно интегрируются в общую систему.

▎Основные характеристики:

1. Этапы разработки: Проект разбивается на небольшие инкременты или версии, каждая из которых добавляет новые функции или улучшает существующие.

2. Постепенное тестирование: Каждый инкремент проходит тестирование, что позволяет выявлять ошибки на ранних стадиях.

3. Гибкость: Возможность вносить изменения в требования на основе отзывов пользователей после каждого инкремента.

4. Быстрая доставка: Пользователи могут получать рабочие версии продукта на более ранних этапах разработки.

▎Преимущества:

* + Быстрая реакция на изменения требований.
  + Упрощенное управление рисками.
  + Возможность получения обратной связи от пользователей на ранних стадиях.

▎Недостатки:

* Требует хорошей координации и управления проектом.
* Может привести к несогласованности между инкрементами, если не уделять должного внимания архитектуре.

Инкрементальная модель подходит для проектов, где важна гибкость и возможность адаптации к изменениям в требованиях.

1. **Экстремальное программирование модель жизненного цикла ПО** - методология разработки программного обеспечения, ориентированная на гибкость, качество и быструю доставку. Она акцентирует внимание на взаимодействии между разработчиками и заказчиками, а также на постоянном улучшении процесса разработки.

▎Основные характеристики:

1. Итеративный процесс: Разработка проходит через короткие итерации, каждая из которых включает в себя планирование, проектирование, кодирование и тестирование.

2. Частые релизы: Программное обеспечение выпускается часто (каждые 1-3 недели), что позволяет пользователям получать новые функции быстрее.

3. Тестирование: В XP тестирование является неотъемлемой частью процесса разработки. Все функции сопровождаются автоматизированными тестами.

4. Парное программирование: Разработчики работают в парах, что способствует обмену знаниями и повышению качества кода.

5. Простота: ориентируется на создание простых решений, которые легко поддерживать и развивать.

6. Обратная связь: Постоянное взаимодействие с заказчиком для получения обратной связи и корректировки требований.

▎Преимущества:

* Высокое качество программного обеспечения.
* Быстрая адаптация к изменениям требований.
* Улучшенное сотрудничество в команде.

▎Недостатки:

* Может быть сложно внедрить в крупные или традиционные организации.
* Требует высокой дисциплины и вовлеченности команды.

Экстремальное программирование подходит для проектов с высокими требованиями к качеству и изменчивым функционалом.

1. **гибкие методологии разработки** **по agile** - философия и набор принципов для разработки программного обеспечения, основанный на гибкости, сотрудничестве и быстром реагировании на изменения.**, scrum** - одна из наиболее популярных реализаций Agile. Он структурирует процесс разработки в виде коротких итераций, называемых спринтами (обычно 2-4 недели). Роли – Продуктовый владелец, Команда разработки, Скрам-мастер (помогает команде следовать принципам Scrum и устраняет препятствия). Процесс включает в себя регулярные встречи, такие как ежедневные стендапы, планирование спринта и ретроспективы.**, kanban** - метод управления потоком работы, который фокусируется на визуализации задач и ограничении количества задач в работе (WIP — Work In Progress).
   * Доска Kanban: визуальное представление задач, разделенных на колонки (например, "Запланировано", "В работе", "Готово").
   * Ограничение WIP: помогает избежать перегрузки команды и улучшить фокусировку на текущих задачах.
   * Непрерывное улучшение: акцент на постоянном совершенствовании процессов.
2. **GitHub. Основные понятия и возможности. Аналоги** – gitflic, Gitee, Gogs, Rhodecode**.**
3. **Автоматизация написания кода, рутинные задачи.** - Генерация кода (Spring Boot для Java или Yeoman для JavaScript), Шаблоны и библиотеки, Фреймворки, Автоматизация тестирования (JUnit, pytest или Jest), скрипты и макросы), Документация (Swagger для API-документации или Javadoc для Java)
4. **Управление проектами. Основные понятия, характеристики и ограничения.** –

*Цель проекта* – наличие чётко выраженного конечного результата, определяемого в терминах затрат, качества и времени реализации

*Уникальность*– проект – разовое начинание, которое не будет повторяться

*Ограниченность во времени*– проект имеет начало и конец

*Ограниченность ресурсов*– ресурсы, которые мы выделяем на выполнение проекта

*Сложность*– для достижения целей проекта необходимо решить множество задач

*Неопределённость* – возможность достижения цели в указанные сроки с выделенными ресурсами заранее не гарантировано

*Предсказуемость*– по мере реализации проекта меняется потребность в тех или иных ресурсах

*Проект* **–** достаточно сложный вид деятельности, которым сложно управлять в силу его уникальности и ограниченности во времени.

Ограничения – время, деньги, качество

1. **Ролевая модель команды. Основные роли, функции и компетенции. Ролевые группы** –

* *Менеджер проекта* **–** главное действующее лицо, обладающее знаниями и навыками, необходимыми для успешного управлением проекта
* *Проектировщик* **–** функция проектирования архитектуры высокого уровня и контроля её выполнения (анализ требований, разработка архитектуры, участие в планировании проекта, контроль выполнения проекта, участие в отборе кадров)
* *Разработчик* **–** роль, ответственная за непосредственное создание конечного продукта (помимо программирования в функции разработчика входит: контроль архитектурных и технических спецификаций продукта, подбор технологических инструментов из стандартов, диагностика и разрешение всех технических проблем, подбор инструментов разработки, метрик и стандартов, контроль за документацией, тестированием и другими этапами, мониторинг состояния продукта)
* *Тестировщик* **–** роль, ответственная за удовлетворение требований к продукту (составление плана тестирования, контроль выполнения плана, разработка теста, автоматизация тестирования, выбор инструментов, метрик стандартов для тестирования, организация бета-тестирования
* *Инженер по качеству* **–** Качество конечного продукта, качество процесса разработки, качество организации

В некоторых случаях функции инженера по качеству возлагаются на *тестировщика*

Отличие от тестировщика:

* + Составление плана качества, когда план тестирования
  + Описание процесса – все процессы должны быть оцифрованы, описание процессов являются их формализацией (метрики и т.д.)
  + Оценка процессов – включают регистрацию хода выполнения процессов и оценку метрик
  + Улучшение процессов – переопределение процессов, автоматизация части работ, обучение персонала
* *Технический писатель* – разработчик документации (разработка плана документирования, выбор и разработка стандартов и шаблонов подготовки документов, выбор средств автоматизации документирования, разработка документации, организация тестирования документации
* *Технолог разработки**ПО* – обеспечивает выполнение: поддержка модели жизненного цикла ПО, создание и сопровождение среды сборки продукта, создание и сопровождение процедуры установки так, чтобы каждая сборка устанавливалась автоматически с учётом версии и конфигурации, управление исходными текстами

1. **Градация разработчика:**

**Преджуниор**

* Опыт: менее 6 месяцев.
* Знания: Основы одного или нескольких языков программирования, базовые концепции алгоритмов и структур данных.
* Навыки: Умение работать с простыми задачами, понимание основ разработки, работа с системами контроля версий.
* Задачи: Выполнение простых задач под руководством более опытных коллег, участие в учебных проектах

**Джуниор**

* Опыт: 6 месяцев — 2 года.
* Знания: Знание одного или нескольких языков программирования, основ разработки ПО.
* Навыки: Способность разрабатывать простые приложения, работа с фреймворками и библиотеками, написание тестов.
* Задачи: Самостоятельное выполнение задач, работа в команде, поддержка кода и исправление ошибок

**Джуниор+**

* Опыт: 1 — 2 года.
* Знания: более глубокое понимание технологий и инструментов, используемых в команде.
* Навыки: Участие в проектировании и разработке более сложных функций, работа с базами данных и API.
* Задачи: Выполнение более сложных задач, помощь в обучении новых членов команды, участие в код-ревью.

**Мидл**

* Опыт: 2 — 5 лет.
* Знания: Глубокое понимание технологий, опыт работы с несколькими проектами.
* Навыки: Способность самостоятельно решать задачи, участие в проектировании архитектуры приложений.
* Задачи: Ведение небольших проектов или модулей, наставничество для джуниоров, активное участие в код-ревью

**Сеньор**

* Опыт: более 5 лет.
* Знания: Экспертное понимание технологий и инструментов, опыт работы с большими проектами.
* Навыки: Принятие архитектурных решений, управление проектами и командами.
* Задачи: Лидерство в команде, mentorship для других разработчиков, участие в стратегическом планировании и принятии важных технических решений.

1. **Модели организации команды (административная модель** - характеризуется строгой иерархией и четкой структурой управления. В команде есть четко определенные роли и обязанности, а также формализованные процессы и процедуры.**, модель хаоса** - предполагает низкий уровень структуры и формализации. Команда работает в условиях высокой неопределенности, что позволяет быстро адаптироваться к изменениям и экспериментировать с новыми подходами.**, открытая архитектура** - предполагает использование открытых стандартов и технологий, а также активное взаимодействие между командами и внешними участниками. Команды могут свободно обмениваться идеями, ресурсами и кодом.**)**
2. **Коммуникации в команде как основной фактор разработки ПО** - является одним из ключевых факторов успешной разработки программного обеспечения (ПО). Эффективное взаимодействие между членами команды может значительно улучшить качество продукта, ускорить процесс разработки и повысить удовлетворенность участников проекта. Рассмотрим несколько аспектов, подчеркивающих важность коммуникации в разработке ПО. Уточнение требований, Синхронизация работы, Устранение проблем, Стимулирование креативности, Создание культуры команды, Использование инструментов для коммуникации
3. **Планирование и управление рисками** Идентификация рисков, Оценка рисков, Разработка стратегий реагирования, Мониторинг и контроль рисков**. Примеры рисков** Технические риски, Риски, связанные с требованиями, Организационные риски, Человеческие риски, Внешние риски, Риски безопасности
4. **Управление качеством проекта** - систематический подход, который обеспечивает соответствие продукта или услуги установленным требованиям и ожиданиям заинтересованных сторон**. Качество ПО** - Качество ПО можно определить как степень, в которой программное обеспечение соответствует заданным требованиям и ожиданиям пользователей: Функциональность, Надежность, Удобство использования, Эффективность, Поддерживаемость, Переносимость **и качество процесса** - относится к эффективности и эффективности методов и практик, используемых для разработки ПО. Это включает в себя: Стандартизация процессов, Управление проектом, Контроль качества, Улучшение процессов**. ISO9000** - серия международных стандартов, которые определяют требования к системам управления качеством (СМК)
5. **Техническая документация** - включает в себя различные документы, которые помогают обеспечить понимание, управление и контроль над проектом. Рассмотрим основные виды технической документации, такие как техническое задание (ТЗ), технический проект, автодокументация и другие средства**. Техническое задание** - документ, который описывает требования к проекту или продукту. Он служит основой для разработки и определяет цели, задачи и ограничения проекта **и технический проект** - более детализированный документ, который разрабатывается на основе ТЗ и описывает, как именно будет реализован проект. **Автодокументация** - процесс автоматического создания документации на основе кода или других артефактов проекта.  **и другие средства** –

* Системы управления документами: Платформы (например, Confluence, SharePoint), которые позволяют хранить, организовывать и совместно редактировать документы.
* Методологии разработки: Agile, Scrum и другие подходы, которые включают в себя создание документации как часть процесса разработки.
* Инструменты для отслеживания изменений: Системы контроля версий (например, Git), которые позволяют отслеживать изменения в документации и коде.

1. **Тестирование ПО** - процесс оценки функциональности и качества программного продукта с целью выявления дефектов, ошибок и несоответствий требованиям. Тестирование является важной частью жизненного цикла разработки ПО и помогает обеспечить надежность и соответствие продукта ожиданиям пользователей**. Цели** - Выявление дефектов, Проверка соответствия требованиям, Улучшение качества, Снижение рисков, Подтверждение готовности к выпуску, Обеспечение удовлетворенности пользователей**. Классификации видов и методов тестирования** - По типу тестирования По стадии тестирования (модульное, Интеграционное, Системное, Приёмочное), По методам тестирования (Чёрный ящик, Белый ящик, Серый ящик)**. Уровни тестирования** - Модульное тестирование (Unit Testing), Интеграционное тестирование (Integration Testing), Системное тестирование (System Testing)**.**
2. **Верификация** - процесс оценки и проверки того, соответствует ли продукт заданным требованиям и спецификациям на различных этапах разработки. Основная цель верификации — убедиться, что продукт был построен правильно, то есть соответствует проектной документации. **и валидация** - процесс оценки и проверки того, соответствует ли продукт требованиям и ожиданиям конечных пользователей. Основная цель валидации — убедиться, что продукт решает реальные задачи и удовлетворяет потребности пользователей.
3. **Отладка** - процесс выявления и устранения ошибок (багов) в программном обеспечении. Этот процесс может включать в себя использование различных инструментов и техник для анализа кода, выполнения программы и проверки ее поведения **и дебаггинг** (то же, что и отладка)**. Логирование** - процесс записи информации о работе приложения в специальные файлы или системы (логи). Это позволяет разработчикам и администраторам отслеживать события, ошибки и другую важную информацию о работе приложения **и профилирование** - процесс анализа производительности приложения для выявления узких мест и оптимизации его работы. Профилирование позволяет измерять время выполнения функций, использование памяти и другие ресурсы.
4. **Юнит тесты** - тесты, которые проверяют отдельные модули или компоненты программы (обычно на уровне функций или методов) на корректность их работы. Они предназначены для того, чтобы удостовериться, что каждый отдельный блок кода выполняет свою задачу правильно[**и пирамида тестирования**](#Тесты)**, программные средства для тестирования** ­– JUnit — для Java, pytest и unittest — для Python, Mocha и Jest — для JavaScript
5. **Фаза интеграционного тестирования** - Основная цель интеграционного тестирования — проверить взаимодействие между различными модулями или компонентами системы, чтобы убедиться, что они работают вместе корректно. Инструменты: JUnit: для Java-приложений, TestNG: для Java, поддерживает сложные сценарии тестирования, Postman: для тестирования API, SoapUI: для тестирования веб-сервисов.
6. **Модели** **белого**- Тестировщик имеет доступ к исходному коду и использует его для проектирования тестов. Это позволяет проверять внутренние структуры, алгоритмы и логику программы.

* Цель: Проверка логики работы программы, выявление ошибок в коде, оптимизация производительности.
* Примеры: Юнит-тестирование, интеграционное тестирование.
* Преимущества: позволяет находить ошибки на ранних этапах разработки и обеспечивать высокое покрытие кода.

**черного** - Тестировщик не знает внутренней структуры или реализации системы. Тестирование основано на спецификациях и требованиях, а не на коде.

* Цель: Проверка функциональности приложения, соответствия требованиям и ожиданиям пользователей.
* Примеры: Функциональное тестирование, системное тестирование, приемочное тестирование.
* Преимущества: позволяет сосредоточиться на конечном результате; не требует знаний о внутреннем устройстве системы.

**серого ящиков** - комбинированный подход, который объединяет элементы черного и белого ящика. Тестировщик имеет частичное понимание внутренней структуры системы, что позволяет разрабатывать более эффективные тесты

* Цель: Сочетание функционального и структурного тестирования для более глубокого анализа системы.
* Примеры: Интеграционное тестирование, тестирование API.
* Преимущества: позволяет использовать как спецификации, так и код для создания тестов, что улучшает качество тестирования.

**в тестировании. Понятие покрытие кода** - метрика, которая используется для оценки качества тестирования программного обеспечения. Она показывает, какая часть исходного кода была выполнена во время тестирования. Покрытие кода помогает определить, насколько полно протестированное приложение

1. Покрытие операторов (Statement Coverage):

* Измеряет процент операторов (строк кода), которые были выполнены во время тестирования.
* Формула: (Количество выполненных операторов / Общее количество операторов) \* 100%

1. Покрытие ветвлений (Branch Coverage):

* Измеряет процент ветвлений (условных операторов), которые были протестированы.
* Формула: (Количество выполненных ветвлений / Общее количество ветвлений) \* 100%

1. Покрытие условий (Condition Coverage):

* Измеряет, были ли проверены все логические условия в коде.
* Формула: (Количество проверенных условий / Общее количество условий) \* 100%

1. Покрытие путей (Path Coverage):

* Измеряет процент всех возможных путей выполнения в коде.
* Это наиболее сложный тип покрытия, так как количество путей может расти экспоненциально с увеличением сложности программы.

5. Покрытие функций (Function Coverage**):**

* + Измеряет процент функций или методов, которые были вызваны во время тестирования.

1. **Статическое** - процесс проверки программного обеспечения без его выполнения. Оно включает в себя анализ документации, кода и других артефактов разработки на предмет соответствия стандартам, требованиям и возможным ошибкам. Методы статического тестирования:
   * Ревью: Оценка документации, кода или дизайна с участием команды или экспертов.
   * Анализ кода: Использование инструментов для автоматического анализа исходного кода на наличие ошибок, потенциальных уязвимостей или несоответствий стандартам.
   * Тестирование на основе требований: Проверка спецификаций и требований на полноту и корректность

**и динамическое тестирование** - процесс проверки программного обеспечения путем его выполнения. Оно включает в себя запуск приложения с различными входными данными и анализ поведения программы**.** Методы динамического тестирования:

* Функциональное тестирование: Проверка функциональности приложения на соответствие требованиям.
* Нагрузочное тестирование: Оценка производительности приложения при различных нагрузках
* Регрессионное тестирование: Проверка, что изменения в коде не повлияли на существующую функциональность.
* Тестирование безопасности: Проверка уязвимостей приложения.

1. **Рефакторинг. Понятие** - процесс изменения внутренней структуры программного кода без изменения его внешнего поведения**, цели** - Улучшение читаемости кода, Упрощение структуры, Увеличение производительности, Снижение технического долга, Подготовка к новым функциональным изменениям **и причины:**

* Накопление технического долга: Быстрая разработка может привести к созданию неэффективного или неаккуратного кода, который требует переработки.
* Изменения в требованиях: при изменении бизнес-требований может потребоваться адаптация существующего кода.
* Появление новых технологий: Появление новых библиотек, фреймворков или языков программирования может требовать обновления старого кода.
* Повышение качества кода: Регулярный рефакторинг помогает поддерживать высокий уровень качества кода, что снижает вероятность возникновения ошибок.

**Понятие плохого кода** - код, который трудно читать, поддерживать или расширять **и технического долга** - метафора, описывающая последствия быстрого и часто неаккуратного написания кода. Это "долг", который необходимо "возвращать" в будущем, когда потребуется исправить ошибки или улучшить код**.**

1. **Реинжиниринг. Понятие** - процесс радикального пересмотра и переработки бизнес-процессов с целью достижения значительных улучшений в таких аспектах, как производительность, качество, скорость и стоимость услуг или продукции. Это подход, который предполагает кардинальные изменения в организации, а не просто небольшие улучшения.**, цели** - Увеличение эффективности, Улучшение качества, Сокращение времени выполнения процессов, Повышение гибкости, Инновации **и причины** - Изменения в рыночной среде (конкуренция), Неэффективность текущих процессов, Технологические изменения, Необходимость снижения затрат**. Сложности реинжиниринга** – Сотрудники могут сопротивляться изменениям, Недостаток ресурсов, Неполное понимание текущих процессов, Риски и неопределенности, Необходимость обучения, Культурные барьеры**.**
2. **Профилирование** - процесс сбора и анализа данных о выполнении программы или системы с целью выявления узких мест, оптимизации производительности и повышения эффективности использования ресурсов. Профилирование позволяет разработчикам понять, как программа использует процессор, память и другие ресурсы, а также какие части кода требуют наибольшего времени выполнения**. Функциональные** –

* *Сбор данных о производительности*: Профилирование позволяет собирать информацию о времени выполнения функций, количестве вызовов, использовании памяти и других метриках,
* *Анализ узких мест*: Определение участков кода, которые требуют наибольших ресурсов или времени выполнения, что помогает сосредоточить усилия на оптимизации этих частей
* *Отчетность*: Генерация отчетов о производительности программы, которые могут включать графики, таблицы и другие визуальные представления данных.
* *Сравнительный анализ:* Возможность сравнивать производительность разных версий программы или различных алгоритмов.
* *Интерактивность*: Некоторые профилировщики позволяют интерактивно исследовать производительность программы во время ее выполнения.

**и аппаратные характеристики** –

* *Использование процессора*: Измерение загрузки процессора, чтобы понять, какие функции или потоки занимают наибольшее время выполнения
* *Использование памяти*: Анализ распределения памяти, включая количество выделенной и освобожденной памяти, а также выявление утечек памяти.
* *Доступ к диску*: Оценка времени доступа к дисковым операциям и их влияния на общую производительность программы.
* *Сетевые характеристики:* Измерение времени сетевых операций и их влияния на выполнение программы, особенно в распределенных системах.
* *Использование кэша:* Анализ работы кэша процессора для оценки эффективности использования памяти и потенциальных узких мест. Анализа покрытия.

1. **инструмент управления проектами Trello** - визуальный инструмент управления проектами, который использует методологию Kanban. Он позволяет командам организовывать задачи и проекты с помощью карточек и досок **/ jira** - более мощный инструмент управления проектами, разработанный компанией Atlassian, который в первую очередь ориентирован на команды разработки программного обеспечения. Он поддерживает методологии Agile, такие как Scrum и Kanban.
2. **Docker** - платформа для разработки, доставки и запуска приложений в контейнерах. Контейнеры позволяют упаковывать приложение и все его зависимости в единый образ, что обеспечивает его работоспособность в любой среде

Основные концепции Docker:

1. Образы (Images):

* Образ — это статичный файл, который содержит все необходимые компоненты для запуска приложения: код, библиотеки, зависимости и настройки.
* Образы создаются на основе Dockerfile — текстового файла, который содержит инструкции по сборке образа.

2. Контейнеры (Containers):

* Контейнер — это запущенный экземпляр образа. Он изолирован от других контейнеров и работает в своей собственной среде.
* Контейнеры легковесны и запускаются быстро, так как используют ядро хост-операционной системы.

3. Docker Hub:

* Это облачный репозиторий, где можно хранить и делиться образами. Docker Hub предоставляет доступ к множеству преднастроенных образов.

4. Docker Compose:

* Это инструмент для определения и запуска многоконтейнерных приложений. С помощью файла docker-compose.yml можно описать конфигурацию всех контейнеров, необходимых для приложения.

1. Преимущества Docker:

* Портативность: приложения могут быть запущены на любом сервере с установленным Docker.
* Изолированность: каждый контейнер работает в своей среде, что минимизирует конфликты между приложениями.
* Эффективность: контейнеры используют ресурсы более эффективно по сравнению страдиционными виртуальными машинами.
* **Kubernetes** – система оркестрации контейнеров, которая автоматизирует развертывание, масштабирование и управление контейнеризованными приложениями.

Основные концепции Kubernetes:

1. Под (Pod) — базовая единица развертывания в Kubernetes. Он может содержать один или несколько контейнеров, которые разделяют ресурсы (например, сеть и хранилище).

2. Сервисы (Services) — абстракция, которая определяет способ доступа к одному или нескольким подам. Он обеспечивает стабильный IP-адрес и DNS-имя для доступа к подам.

3. Деплойменты (Deployments) - управляет созданием и обновлением подов. Он позволяет легко масштабировать приложение и откатывать изменения в случае необходимости.

4. Состояние (StatefulSets) - используется для управления состоянием приложений, которые требуют уникальных идентификаторов и постоянного хранилища.

5. Кластеры (Clusters) - Kubernetes работает в кластере, который состоит из одного или нескольких узлов (серверов), на которых запускаются поды.

6. Преимущества Kubernetes:

* Автоматическое масштабирование: K8s может автоматически увеличивать или уменьшать количество подов в зависимости от нагрузки.
* Самовосстановление: если под выходит из строя, Kubernetes автоматически перезапускает его.
* Управление конфигурацией: K8s позволяет управлять конфигурацией приложения через ConfigMaps и Secrets. (образы, контейнеры, виртуализация и т.д.)

Виртуализация: использует гипервизоры для создания виртуальных машин (VM), которые эмулируют полные физические серверы. Каждая VM имеет свою операционную систему и ресурсы, что делает их более тяжелыми по сравнению с контейнерами.

Контейнеризация: использует общее ядро операционной системы для запуска изолированных приложений в контейнерах. Это делает контейнеры более легкими и быстрыми по сравнению с виртуальными машинами.

1. **Управление командой разработки ПО** - включает в себя множество аспектов, связанных с контролем качества, производительностью и развитием сотрудников. Рассмотрим подробнее ключевые показатели контроля качества и методы управления, такие как план повышения производительности (PIP), итоги Performance Review, SMP (Software Metrics Program), UEIP (User Experience Improvement Plan) и другие**.**

**Показатели контроля качества** –

* Время выполнения задач: Среднее время, необходимое для завершения задач или историй пользователя.
* Количество завершенных задач: Количество задач, выполненных за определенный период времени (спринт, месяц и т.д.).
* Качество кода: измеряется через количество ошибок или дефектов на единицу кода (например, количество багов на 1000 строк кода).
* Тестовое покрытие: Процент кода, покрытого автоматизированными тестами.

**план повышения производительности PIP** - документ, который описывает конкретные шаги и цели для улучшения производительности сотрудников, которые не достигают ожидаемых результатов (Описание текущих проблем с производительностью, Конкретные цели и ожидаемые результаты, План действий с четкими сроками, Ресурсы и поддержка, доступные сотруднику)**, итоги Performance Review** - процесс оценки работы сотрудников за определенный период (обычно год): Оценка достижения целей, Обратная связь от менеджера и коллег, Обсуждение карьерного роста и развития навыков, Установление целей на следующий период**, SMP** (Не SAMP ☹) - программа, направленная на сбор и анализ метрик разработки ПО для оценки качества и производительности команды (Время на исправление дефектов, Частота релизов, Уровень удовлетворенности пользователей, Количество автоматизированных тестов и их результаты)**, UEIP** - план, направленный на улучшение пользовательского опыта. Он может включать в себя исследования пользователей, тестирование интерфейсов и анализ отзывов (анализ текущего пользовательского опыта, Определение проблемных областей, Предложения по улучшению интерфейса и функционала, Метрики для оценки успешности внедренных изменений **и** [**т.д.**](#Agile)

1. **Свободный вопрос по курсу.**