**Основные вопросы**

1) Объектно-ориентированное программирование (ООП)

ООП — это парадигма программирования, основанная на концепции "объектов", которые могут содержать данные в виде полей (атрибутов) и код в виде процедур (методов). Основными принципами ООП являются:

1. Инкапсуляция: механизм, объединяющий данные и методы, которые с ними работают, и защищающий их от внешнего вмешательства и неправильного использования.
2. Наследование: механизм, позволяющий создавать новые классы на основе существующих, перенимая их свойства и методы.
3. Полиморфизм: возможность объектов с одинаковым интерфейсом иметь различную реализацию.
4. Абстракция: выделение существенных характеристик объекта, отличающих его от всех других объектов.

ООП позволяет структурировать программы таким образом, чтобы свойства и поведение были сгруппированы в отдельные объекты, что облегчает понимание, разработку и поддержку кода.

2) Магическое число 7 Миллера

Психолог Джордж Миллер в 1956 году обнаружил, что кратковременная память человека может удерживать примерно 7 (±2) элементов информации. Примеры из IT:

1. Количество пунктов в меню: оптимальное количество пунктов меню в интерфейсе — 5-9.
2. Количество шагов в навигации: рекомендуемое количество шагов для достижения цели — не более 7.
3. Количество форм на странице: оптимальное количество форм ввода на одной странице — 5-9.
4. Количество вкладок: оптимальное количество вкладок в интерфейсе приложения — 5-9.
5. Количество цветов в палитре: эффективное количество основных цветов для интерфейса — 5-7.
6. Количество уровней в иерархической структуре: рекомендуемая глубина меню — не более 7 уровней.
7. Количество переменных в функции: оптимальное количество параметров функции — не более 7.
8. Количество элементов в чанке кода: разделение кода на логические блоки по 5-9 строк.

3) Энтропия ПО и негэнтропийные меры

Энтропия программного обеспечения — это мера беспорядка, хаоса и непредсказуемости системы, которая возрастает с течением времени и в процессе модификаций.

Негэнтропийные меры при разработке ПО:

1. Рефакторинг кода: регулярное улучшение структуры кода без изменения его внешнего поведения снижает энтропию.
2. Модульное тестирование: создание автоматизированных тестов, которые проверяют работу отдельных модулей, предотвращает деградацию системы.
3. Следование принципам SOLID: применение принципов проектирования, таких как единственная ответственность и открытость/закрытость, снижает сложность.
4. Непрерывная интеграция и доставка (CI/CD): автоматизированные процессы сборки и проверки кода помогают поддерживать качество.
5. Документирование кода и архитектуры: подробная документация помогает понимать систему и снижает вероятность внесения ошибок.
6. Статический анализ кода: использование инструментов для обнаружения потенциальных проблем снижает энтропию.

4) 5 признаков сложной системы по Гради Бучу

1. Сложные системы имеют иерархическую структуру
   1. Пример 1: В моем веб-приложении есть иерархия компонентов: корневой App содержит Layout, который включает Header, Main и Footer.
   2. Пример 2: В моей системе управления базой данных есть иерархия от соединения к базе данных, таблицам, записям и полям.
2. Относительная простота базовых компонентов
   1. Пример 1: Отдельные компоненты моего React-приложения выполняют простые функции (кнопка, поле ввода), но в совокупности образуют сложную систему.
   2. Пример 2: В моем микросервисном проекте каждый сервис решает одну задачу, но их взаимодействие образует комплексную архитектуру.
3. Отсутствие центрального управляющего элемента
   1. Пример 1: В моем приложении для управления задачами каждый модуль (задачи, пользователи, команды) работает независимо, взаимодействуя через события.
   2. Пример 2: В распределенной системе кэширования, которую я разрабатывал, каждый узел принимает решения локально, но общая система сохраняет целостность.
4. Эмерджентность (появление новых свойств)
   1. Пример 1: В моем проекте машинного обучения объединение нескольких простых моделей классификации создало систему с более высокой точностью.
   2. Пример 2: В игровом движке, над которым я работал, взаимодействие физики, рендеринга и AI создавало сложное поведение игрового мира.
5. Наличие контуров обратной связи
   1. Пример 1: В моей системе мониторинга сервера обратная связь от метрик производительности автоматически регулирует распределение ресурсов.
   2. Пример 2: В проекте аналитической платформы пользовательское поведение влияет на рекомендательную систему, что в свою очередь изменяет пользовательское поведение.

5) Закон иерархических компенсаций Седова

Закон Седова гласит, что в сложной иерархической системе увеличение разнообразия на верхнем уровне обеспечивается ограничением разнообразия на нижних уровнях, и наоборот.

Примеры применения в эволюции IT:

1. Стандартизация аппаратного обеспечения → разнообразие ПО: стандартизация архитектуры x86 позволила развиваться разнообразному программному обеспечению.
2. Стандартизация сетевых протоколов → разнообразие сетевых приложений: принятие TCP/IP как стандарта привело к появлению множества интернет-сервисов и приложений.
3. Ограничение возможностей языков программирования → увеличение надежности программ: языки со строгой типизацией (Rust, TypeScript) ограничивают свободу программиста, но увеличивают надежность и безопасность кода.
4. Контейнеризация → разнообразие микросервисов: стандартизация инфраструктуры с помощью Docker и Kubernetes позволила развиваться разнообразным микросервисным архитектурам.
5. Стандартизация интерфейсов API → многообразие интеграций: принятие REST и GraphQL как стандартов API позволило создавать разнообразные интеграции между системами.
6. Унификация пользовательских интерфейсов → разнообразие приложений: стандартизированные компоненты UI (Material Design, iOS Human Interface Guidelines) упростили разработку разнообразных приложений.