**Технологии создания программных продуктов**

**Экзаменационные вопросы**

1. **Инженерия программного обеспечения. Основные понятия**

Программная инженерия — это область компьютерной науки и технологии, которая занимается построением сложных программных систем, требующей слаженной работы команд программистов разных специализаций и квалификаций.

ТСПП − дисциплина, изучающая все аспекты создания программного продукта (организационные, технические, инженерные), включая методологии разработки ПП.

Цель дисциплины - эффективное создание качественного ПП.

1. **Понятие программного продукта, классификация программных продуктов**

Программный продукт (ПП) — это программное обеспечение, разработанное для решения определенной задачи или набора задач. Включает саму компьютерную программу, документацию к ней и конфигурационные (настроечные) файлы.

* общие − решают распространенные задачи, свободно-доступные на рынке ПО;
* заказные − закрывают коммерческие потребности заказчика;

1. **Технологический процесс создания ПО**

Этапы разработки (жизненный цикл ПП):

1. Спецификация требований. Определение целей и требований заказчика к создаваемому ПО.

2. Проектирование. Разработка архитектуры ПО, определение его компонентов и интерфейсов, выбор технологий и инструментов, которые будут использоваться при разработке.

3. Разработка. Сама разработка ПО, включая написание кода, тестирование и отладку.

4. Тестирование. Проверка работоспособности и соответствия ПО заданным требованиям. Тестирование может включать в себя различные виды тестирования, включая модульное, интеграционное, системное и приемочное.

5. Релиз и поддержка. После успешного прохождения всех этапов тестирования ПО выпускается в эксплуатацию. Важным этапом является поддержка ПО, которая может включать в себя исправление ошибок и обновление функциональности.

6. Вывод из эксплуатации. Ряд мероприятий, направленных на прекращение использования объекта или системы.

1. **Моделирование технологического процесса создания ПО**

Модель процесса создания ПП – упрощенное абстрагированное представление реального процесса разработки, использующееся для анализа и управления параметрами процесса.

Типы моделей:

1. Code & Fix – неформальный подход, не подходит для больших продуктов;

2. Классические модели: каскадная (waterfall), эволюционная;

3. Гибридные модели (пошаговая, спиральная);

4. Гибкие методологии (Agile, Scrum) – ориентированы на разработчика;

1. **Характеристики качественного программного обеспечения**

Показатели качества ПП:

1. Соответствие функциональным требованиям;
2. Скорость работы;
3. Удобство использования;
4. Надежность (работа в заданное время без сбоев, частота сбоев);
5. Защищённость от несанкционированного доступа;
6. Безопасность;
7. Удобство сопровождения;
8. **Жизненный цикл программного продукта**

Жизненный цикл ПО − весь период существования ПО, связанный с подготовкой к его разработке, разработкой и использованием. Начинается с момента, принятия решения разработать новую систему, заканчивается с полного прекращения её использования.

1. Спецификация требований (ТЗ);
2. Проектирование;
3. Разработка и тестирование отдельных модулей;
4. Сборка системы (build);
5. Аттестация ПП (проверка на соответствие с ожиданиями заказчика);
6. Ввод в эксплуатацию (развертывание ПП на системы заказчика);
7. Скорректировать продукт заказчика;
8. Обучение персонала;
9. Сопровождение ПП и эксплуатация;
10. Вывод из эксплуатации;
11. **Каскадная модель разработки ПО**

Эта модель предполагает последовательное выполнение различных видов деятельности, начиная с выработки требований и заканчивая сопровождением, с четким определением границ между этапами, на которых набор документов, созданный на предыдущей стадии, передается в качестве входных для следующей.

Особенность в том, что переход к следующему этапу происходит после завершения предыдущего.



Плюсы:

* Простота локализации ошибок во время отладки;
* Сроки и бюджет зафиксированы;
* Требования не меняются во время работы;

Минусы:

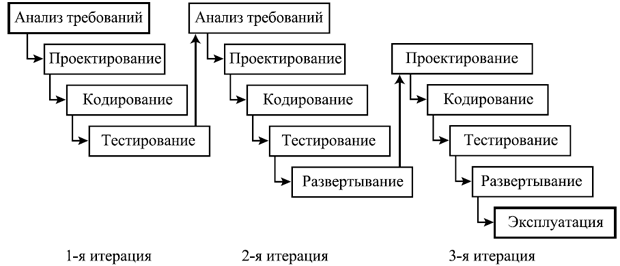
* Время разработки;
* Поздняя фиксация проблем и изменений (из-за последовательной работы фундаментальные проблемы проекта находят слишком поздно).
* Дорогостоящее исправление ошибок;

1. **Эволюционная модель разработки ПО**

Эволюционная модель разработки ПО предполагает постепенное уточнение требований и разработку системы в несколько итераций (решает проблемы каскадной модели). Каждая итерация включает в себя фазы, которые могут повторяться в каждой итерации:

Подходы:

1. Метод пробных разработок (постоянное взаимодействие с заказчиком, разработка того, что понятно);
2. Прототипирование (испытание прототипа для начальных требований);



Плюсы:

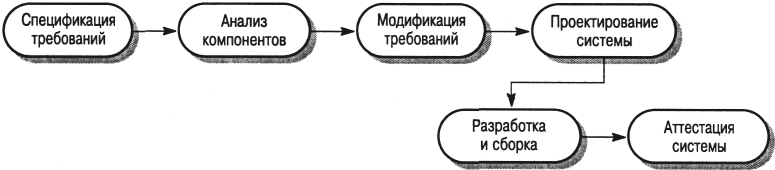
* Готовность к изменению требований;
* Раннее выявление ошибок;
* Полное удовлетворение требованиям;

Минусы:

* Нефиксированное время разработки;
* Сложная поддержка ПП;
* Недостаток документации

1. **Разработка ПО на основе ранее созданных компонентов**

Иногда при разработке собственного оригинального решения можно использовать чужие готовые наработки.



Плюсы:

* Экономия времени;
* Предсказуемость процесса разработки;
* Снижение рисков;

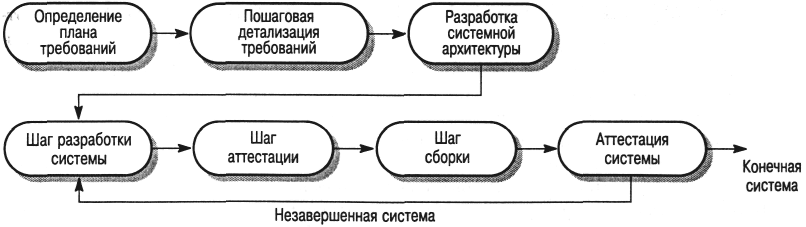
Минусы:

* Необходимость проверять надёжность компонентов

1. **Модель пошаговой разработки ПО**

- объединение каскадной и эволюционной модели

Модель пошаговой разработки – модель разработки, в которой процессы специфицирования требований, проектирования и написания кода разбиваются на последовательность небольших шагов, ведущих к созданию ПО.



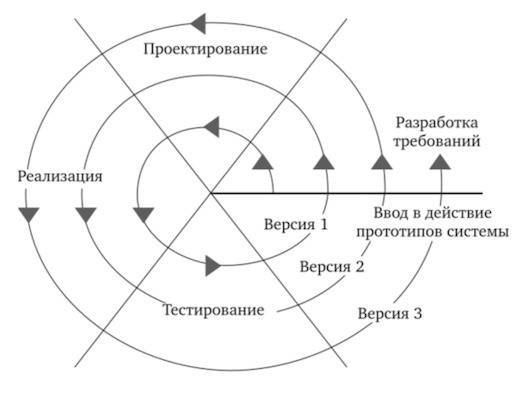
Даёт возможность выполнять отдельные процессы разработки подсистем и весь процесс создания ПО итерационно, когда в ответ на изменения требований повторно выполняются определенные этапы создания системы (чаще всего этапы проектирования и кодирования).

Плюсы: промежуточная оценка системы заказчиком, реализация наиболее важных требований в самом начале, использование прототипа для уточнения требований, наиболее важные компоненты тестируются многократно, снижение риска системных ошибок.

Минусы: ограничение на сложность функций, на каждый шаг 1 модуль, неэффективность при взаимодействии компонентов, сложность реализации общих системных функций.

1. **Спиральная модель разработки ПО**

- весь процесс создания ПО, от начального эскиза системы до ее конечной реализации, разворачивается по спирали.



Спиральная модель представляет собой гибридную модель, которая объединяет в себе преимущества моделей каскадной и эволюционной модели разработки. Она предполагает последовательное выполнение этапов, которые повторяются на каждой итерации.

Этапы:

1. Определение целей, альтернатив, ограничений, проектных рисков.
2. Анализ рисков. На этапе анализа рисков определяются возможные риски, которые могут возникнуть в процессе разработки, и планируются меры по их снижению и устранению.
3. Разработка и тестирование. На этом этапе происходит разработка и тестирование системы.
4. Оценка и планирование. На этом этапе проводится оценка качества системы, ее соответствия требованиям и возможности улучшения. Затем планируется следующая итерация.

Плюсы:

* Гибкость. Эта модель позволяет легко адаптироваться к изменяющимся требованиям и потребностям бизнеса.
* Раннее выявление рисков. Анализ рисков на каждой итерации позволяет на ранней стадии выявлять возможные проблемы и риски и устранять их.
* Постепенное уточнение требований. Эта модель позволяет уточнять требования к системе на протяжении всего процесса разработки.
* Возможность быстрого выпуска продукта. Эта модель позволяет выпускать продукт после каждой итерации.

Минусы:

* Высокие затраты на разработку
* Постоянный контроль над процессом
* Сложность модели (необходимость в высококвалифицированных работниках)
* Риск увеличения времени разработки

1. **Гибкие методологии разработки ПО. Основные принципы**

Гибкие методологии разработки ПО основаны на наборе принципов, которые включают:

• Гибкость и адаптивность к изменениям требований.

• Работа в сотрудничестве с заказчиком и заинтересованными сторонами.

• Итеративный подход к разработке, с акцентом на частую доставку ценных результатов.

• Самоорганизованные и многофункциональные команды разработчиков.

• Постоянное стремление к улучшению качества и процессов разработки.

Существует Agile – манифест для гибких методологий:

1. Люди важнее инструментов
2. Качество продукта важнее документации
3. Общение с заказчиком важнее контракта
4. Готовность к изменениям важнее установленного плана
5. **Гибкие методологии разработки ПО. Методология SCRUM**

SCRUM — это гибкая методология разработки ПО, основанная на итеративном и инкрементальном подходах. В SCRUM проект разделен на короткие циклы разработки, называемые спринтами, которые обычно длительностью от 1 до 4 недель. Команда разработчиков организована в многофункциональные самоорганизующиеся команды, и каждый спринт включает этапы планирования, разработки, тестирования и проверки результатов. SCRUM также поддерживает частую коммуникацию и взаимодействие между командой и заказчиком.

Роли:

Свиньи – полностью заинтересованные участники (Product Owner, Product Backlog, Scrum Master, разработчики)

Куры – частично заинтересованные (Пользователи, стейкхолдеры)

1. **Гибкие методологии разработки ПО. Экстремальное программирование**

Экстремальное программирование (Extreme Programming, XP) — это гибкая методология разработки ПО, которая акцентирует внимание на качестве кода, быстрой обратной связи и коллаборации внутри команды разработчиков. XP предлагает практики, такие как парное программирование, тестирование на каждой итерации, непрерывную интеграцию, короткие итерации, управление требованиями через истории пользователей и другие. Она стремится к быстрой итеративной разработке, акцентируя внимание на гибкости и быстрой адаптации к изменениям.

1. **Требования к ПО. Классификация требований**

Требования к ПО - определяют, какие свойства и характеристики он должно иметь для удовлетворения потребностей пользователей и других заинтересованных лиц.

Классификация требований:

• Пользовательские требования

• Системные требования

• Проектная системная спецификация

• Функциональные

• Нефункциональные

1. **Функциональные требования к ПО**

Функциональные требования — это описание функциональности, которую должно обеспечивать ПО. Это может быть список операций, которые должны быть выполнены, или список функций, которые ПО должно обеспечивать для пользователей.

1. **Нефункциональные требования к ПО**

Нефункциональные требования — это требования к качеству ПО, такие как производительность, надежность, безопасность, доступность и т.п. Эти требования часто связаны с аспектами, которые не являются прямо связанными с функциональностью ПО, но важны для его успешного функционирования.

1. **Пользовательские и системные требования к ПО**

Пользовательские требования — это требования, которые выражают потребности конечных пользователей. Это может быть простой интерфейс, удобство использования, легкость обучения и т.д.

Системные требования к ПО – это перечень минимальных и рекомендуемых характеристик компьютерной системы, необходимых для запуска и нормальной работы конкретного программного продукта.

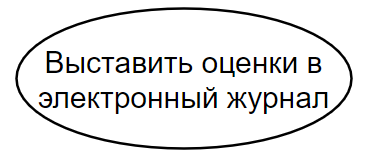
1. **Язык UML. Принципы построения диаграмм прецедентов**

Диаграмма вариантов использования (англ. use-case diagram) – диаграмма, описывающая, какой функционал разрабатываемой программной системы доступен каждой группе пользователей.

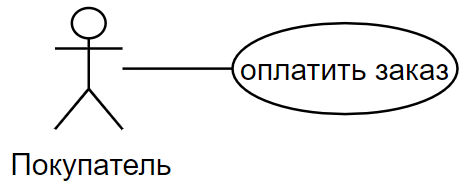
Каждая группа пользователей на диаграмме вариантов использования обозначается человечком. В терминологии UML, этот человечек называется актёром (англ. "actor"). В общем случае, актёр обозначает любые сущности, использующие систему. Этими сущностями могут быть люди, технические устройства или даже другие системы.



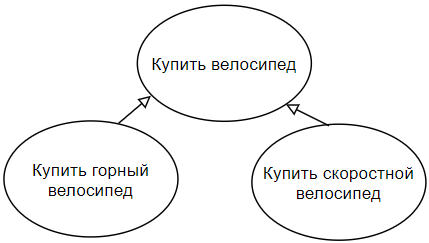
Каждая группа пользователей использует определённые функции системы. На диаграмме вариантов использования функция системы изображается эллипсом.



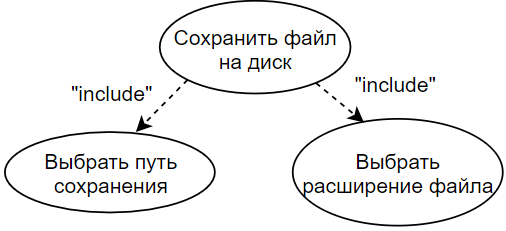
Отношение ассоциации предназначено для соединения актёров и вариантов использования. Отношение означает, что данный актёр может выполнять действия, описанные вариантом использования.



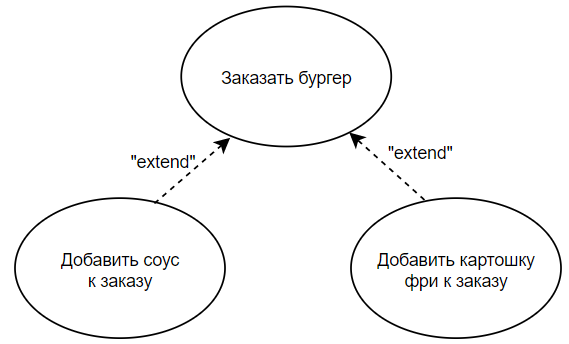
Отношение обобщения означает, что некоторый актёр (вариант использования) может быть обобщён до другого актёра (варианта использования). Стрелка направлена от частного случая (специализации) к общему случаю.



Отношение включения используется, чтобы показать, что некоторый вариант использования включает в себя другой вариант использования в качестве составной части.



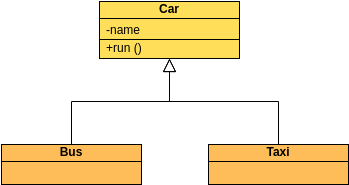
Отношение расширения — это выборочное отношение включения. Если отношение включения обозначает, что элемент обязательно включается в состав другого элемента, то в случае отношения расширения это включение необязательно.



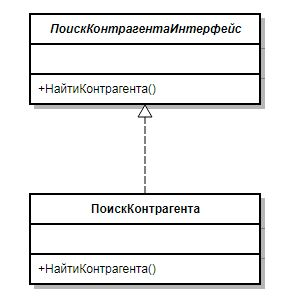
1. **Язык UML. Принципы построения диаграмм классов**

Существует шесть основных типов отношений между классами: наследование, реализация, композиция, агрегация, ассоциация и зависимость.

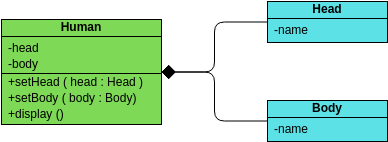
Наследование (обобщение) используется для описания отношений между родительским (базовым) и дочерним (производным) классами.



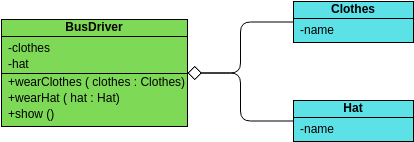
Реализация (имплементация) в основном используется для указания связи между интерфейсами и классами реализации. Интерфейс (включая абстрактный класс) — это набор методов. Класс реализует интерфейс, а методы в классе реализуют все методы объявления интерфейса.



Композиция: отношение между целым и частью, но целое и часть не могут быть разделены. Общее и часть имеют согласованное время жизни. Как только объект перестанет существовать, связанные с ним объекты умрут в одной и той же жизни.



Агрегация: отношение между целым и частью класса, объекты-члены являются частью общего объекта, но могут существовать независимо от общего объекта.



Ассоциация: указывает, что свойство класса содержит ссылку на экземпляр (или экземпляры) другого класса. Существует четыре вида ассоциаций: двусторонние ассоциации, односторонние, самоассоциация и многозначные ассоциации.



Зависимость: предположим, что изменение в классе А вызывает изменение в классе В, тогда скажем, что класс В зависит от класса А. В большинстве случаев зависимости отражаются в методах класса, использующих в качестве параметра объект другого класса.

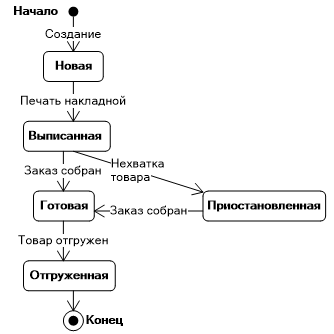


1. **Язык UML. Принципы построения диаграмм состояний**

Диаграмма состояний позволяет описывать поведение системы. В ООП разрабатывается диаграмма состояний единственного класса, демонстрирующая поведение одного объекта в течение его жизни.

Основные шаги построения диаграммы состояний:

* добавление состояний;
* указание переходов;
* добавление внутренних активностей;
* указание подсостояний и суперсостояний;



Каждый переход имеет метку, состоящую из следующих необязательных частей:

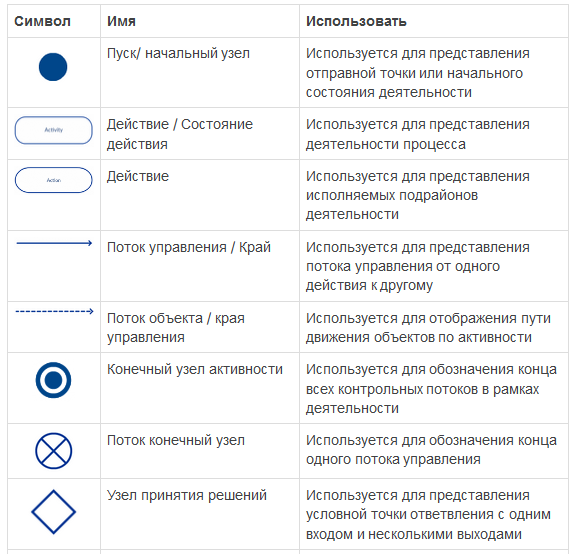
Триггер-идентификатор — единственное событие, способное вызвать изменение состояния. Пропуск этой части означает, что переход происходит немедленно

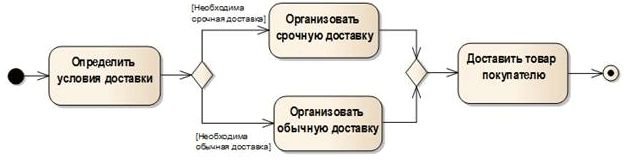
Защита — логическое условие, выполнение которого обязательно для осуществления перехода ( [Reason != null] ).

Активность — поведение системы во время перехода. Пропуск активности означает, что в процессе перехода ничего не происходит

1. **Язык UML. Принципы построения диаграмм видов деятельности**

Диаграмма активностей (видов деятельности) - как и диаграмма состояний, отражает динамические аспекты поведения системы. Представляет собой блок-схему, которая наглядно показывает, как поток управления переходит от одной деятельности к другой.





1. **Язык UML. Принципы построения диаграмм последовательностей**

Диаграммы последовательностей используются для уточнения диаграмм прецедентов, более детального описания логики сценариев использования.

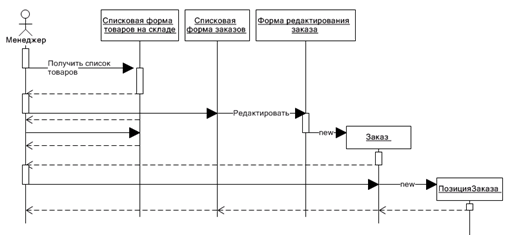
Они обычно содержат объекты, которые взаимодействуют в рамках сценария, сообщения, которыми они обмениваются, и возвращаемые результаты, связанные с сообщениями.

Объекты обозначаются прямоугольниками с подчеркнутыми именами (чтобы отличить их от классов).

Сообщения (вызовы методов) - линиями со стрелками.

Возвращаемые результаты - пунктирными линиями со стрелками.

Прямоугольники на вертикальных линиях под каждым из объектов показывают “время жизни” (фокус) объектов.



1. **Системы контроля версий и коллективная разработка ПО**

Системы контроля версий (СКВ) — это инструменты, которые помогают управлять изменениями в исходном коде и других файлах. Они позволяют отслеживать изменения в файлах, сохранять разные версии файлов и возвращаться к предыдущим версиям, если это необходимо.

СКВ позволяют командам разработчиков работать над проектом совместно, вносить изменения в код, не нарушая работу других членов команды, и отслеживать, кто и когда внес изменения.

Распределённые системы контроля версий (РСКВ) отличаются от централизованных тем, что каждый участник проекта имеет полную копию всего репозитория, а не только его часть. Это означает, что каждый участник может работать локально, сохранять изменения и обмениваться ими с другими участниками, не нуждаясь в непрерывном подключении к центральному серверу.

Примеры СКВ включают Git, Subversion (SVN), Mercurial, CVS и другие.



1. **Понятие архитектуры программной системы. Назначение, основные элементы, цели и задачи архитектуры ПО.**

Архитектура программной системы — это форма, которая придается системе её создателями. Эта форма образуется декомпозицией (делением) системы на компоненты, их организацией и определением способов взаимодействия между ними.

Главное предназначение архитектуры — поддержка жизненного цикла системы. Хорошая архитектура делает систему легкой в освоении, простой в разработке, сопровождении и развертывании. Конечная ее цель — минимизировать затраты на протяжении срока службы системы и максимизировать продуктивность программиста.

Главная стратегия - сделать связанность компонентов системы максимально низкой таким образом, чтобы компоненты могли быть легко заменяемыми и расширяемыми.

Хорошая архитектура должна обеспечивать:

* Разнообразие вариантов использования и эффективную работу системы;
* Простоту сопровождения системы;
* Простоту разработки системы;
* Простоту развертывания системы;

1. **Понятие и характеристика слоёв приложения, их роль в архитектуре программной системы.**

Приложение в программной системе может быть организовано в виде нескольких слоев, каждый из которых имеет свою специфическую функцию и является важным элементом архитектуры приложения. Ниже приведены характеристики каждого слоя:

1. Представление: слой обеспечивает пользовательский интерфейс и обработку запросов. Обычно состоит из веб-страниц, форм, контролов и других элементов пользовательского интерфейса. Роль этого слоя - обеспечить простой и понятный интерфейс для взаимодействия пользователя с приложением.
2. Бизнес-логика: этот слой обрабатывает бизнес-логику приложения. Он содержит правила и процедуры, необходимые для выполнения бизнес-функций приложения. Роль этого слоя-обеспечить логическую обработку данных и управление бизнес-процессами.
3. Доступ к данным: этот слой предоставляет доступ к хранилищу данных, которое используется в приложении. Содержит код для чтения и записи данных, а также для управления соединением с базой данных. Роль этого слоя - обеспечить безопасный и эффективный доступ к данным.
4. Инфраструктура: этот слой содержит компоненты, необходимые для работы приложения, не относящиеся к бизнес-логике приложения. Это может включать в себя сервисы, библиотеки, конфигурационные файлы и др. элементы. Роль этого слоя - обеспечить стабильность и надежность работы приложения.
5. **Понятие и отличительные особенности корпоративных программных приложений. Типы корпоративных приложений. Различия между B2B и B2C приложениями.**

К корпоративным приложениям относятся: бухгалтерский учет, ведение медицинских карт пациентов, экономическое прогнозирование, анализ кредитной истории клиентов банка, страхование, внешнеэкономические торговые операции и другие.

* Предполагают долговременное хранение данных, которые могут пережить несколько поколений программ и оборудования.
* Система должна обеспечить возможность одновременной работы нескольких пользователей с одним и тем ресурсом, не блокируя и не вызывая коллизий.
* Система должна иметь уровни доступа, соответствующие бизнес-правилам.
* Система должна предусмотреть возможность интеграции с внешними сервисами.
* Система должна быть отказоустойчивой и иметь возможность быстрого восстановления данных при утере.
* Добавление нового функционала не должно влечь за собой расходы большого количества ресурсов на рефакторинг смежных подсистем.

Типы систем:

1. Бизнес-пользователь (В2С) − обслуживает множество пользователей и должна быть масштабируемой и эффективной в использовании ресурсов. Важным показателем для таких систем является время работы системы, идеальный показатель - 100%, но на текущий момент к такому результату могут приблизится только несколько крупных компаний. Важно, чтобы B2С системы имели отличный интерфейс для пользователей (UX), так как основная задача бизнеса - привлечь как можно больше пользователей.
2. Бизнес-бизнес (В2В) - обслуживание корпоративных клиентов в банке, эквайринг и оптовые закупки. Такие системы не имеют большой посещаемости, но потеря одного клиента бизнеса гораздо значительнее, чем потеря одного обычного потребителя. Поэтому важно, чтобы основной функционал системы работал без сбоев. Безопасность данных также важна для пользователей, так как через систему обеспечивается функционирование бизнеса клиента и взаимодействие с его пользователями.
3. **Понятие и свойства масштабируемых систем**

Время отклика — промежуток времени, который требуется системе, чтобы обработать запрос извне, подобный щелчку на кнопке графического интерфейса или вызову функции API сервера.

Быстрота реагирования — скорость подтверждения запроса. Чем раньше система способна подтвердить получение запроса, тем её быстрота реагирования выше. Не путать с временем отклика.

Время задержки — минимальный интервал времени между моментом отправки сигнала, запроса или команды, и моментом получения ответа на этот сигнал, запрос или команду. Может зависеть от разных факторов, таких как скорость передачи данных, расстояние и количество устройств на пути передачи, а также от характеристик самой системы и ее нагрузки.

Пропускная способность — количество данных (операций), передаваемых (выполняемых) в единицу времени. Например, может измеряться числом байтов в секунду или числом транзакций в секунду (tps).

Загрузка — значение, определяющее степень "давления" на систему и измеряемое, например, количеством одновременно подключенных пользователей. Параметр обычно служит контекстом для представления других функциональных характеристик.

Чувствительность к загрузке — выражение, задающее зависимость времени отклика от загрузки. Например, система А обладает временем отклика, равным 0,5 секунды для 10-20 пользователей, а система В — временем отклика в 0,2 секунды для 10 пользователей и 2 секунды для 20 пользователей.

Эффективность — удельная производительность в пересчете на одну единицу ресурса. Например, система с двумя процессорами, способная выполнить 30 tps, более эффективна по сравнению с системой, оснащенной четырьмя аналогичными процессорами и обладающей продуктивностью в 40 tps.

Мощность — наибольшее значение пропускной способности или загрузки. Это может быть как абсолютный максимум, так и некоторое число, при котором величина производительности все еще превосходит заданный приемлемый порог.

Способность к масштабированию — свойство, характеризующее поведение системы при добавлении ресурсов (обычно аппаратных). Масштабируемой принято считать систему, производительность которой возрастает пропорционально объему приобщенных ресурсов (скажем, вдвое при удвоении количества серверов).

* вертикальное – увеличение мощности отдельного сервера;
* горизонтальное – наращивание потенциала путём добавления новых серверов;

1. **Понятия "срочное и важное" в контексте приоритетов информационных систем. Факторы, определяющие приоритеты в архитектуре программной системы.**

Матрица Эйзенхауэра:



Первая ценность программного обеспечения — поведение — это нечто срочное, но не всегда важное. Вторая ценность — архитектура — нечто важное, но не всегда срочное.

Конечно, имеются также задачи важные и срочные одновременно и задачи не важные и не срочные. Все эти четыре вида задач можно расставить по приоритетам:

1. Срочные и важные.
2. Не срочные и важные.
3. Срочные и не важные.
4. Не срочные и не важные.

Разработчики программного обеспечения оказываются перед проблемой, обусловленной неспособностью руководителей оценить важность архитектуры. Но именно для ее решения они и были наняты. Поэтому разработчики должны всякий раз подчеркивать приоритет важности архитектуры перед срочностью поведения.

1. **Понятие парадигмы программирования, её роль в архитектурном проектировании информационных систем. Основные парадигмы программирования.**

Парадигма программирования — это основная концептуальная модель, которая определяет методы и подходы для разработки программного обеспечения. Она определяет как программисты должны думать и решать задачи при написании кода. Парадигмы программирования определяют способы организации кода, структуры данных и взаимодействия между компонентами системы.

Основные парадигмы программирования:

1. Процедурное программирование - программа разбивается на процедуры или функции.
2. Объектно-ориентированное программирование - программа строится на основе объектов, которые имеют свойства и методы.
3. Функциональное программирование - программа рассматривается как набор функций, которые выполняют определенные операции.
4. Логическое программирование - программа задается набором логических утверждений, а исполнительная система сама определяет, какие действия нужно выполнить.
5. Декларативное программирование - программа описывает, что нужно сделать, а не как это сделать.
6. Императивное программирование - программа описывает последовательность команд, которые нужно выполнить для получения результата.
7. **Структурное программирование: история, особенности, применение.**

Первой, получившей всеобщее признание (но не первой из придуманных), была парадигма структурного программирования, предложенная Эдсгером Вибе Дейкстрой в 1968 году. Дейкстра показал, что безудержное использование переходов (инструкций goto) вредно для структуры программы.

Дейкстра предположил программистам использовать проверенные структуры и связывать их с кодом, в правильности которого они хотели бы убедиться. Для этого было необходимо разработать методику написания доказательств на простых алгоритмах.

В связи с этим Дейкстра воспользовался результатами исследований Бёма и Якопини, доказавшими, что любую программу можно написать, используя всего три структуры: последовательность, выбор и итерации (if/then/else и do/while/until).

Управляющие структуры, делающие доказуемой правильность модуля, в точности совпадали с набором структур, минимально необходимым для написания любой программы.

Декомпозиция — это разделение программы на ряд функций верхнего уровня, а их в свою очередь - на еще более низкого, и так до момента, пока каждый “лист”, т.е. единичная функция, не станет атомарной.

1. **Объектно-ориентированное программирование: история, особенности, применение.**

В 1966 г. программисты Оле-Йохан Даль и Кристен Нюгор заметили, что в языке ALGOL есть возможность переместить кадр стека вызова функции в динамическую память (кучу), благодаря чему локальные переменные, объявленные внутри функции, могут сохраняться после выхода из нее. В результате функция превращалась в конструктор класса, локальные переменные — в переменные экземпляра, а вложенные функции — в методы. Это привело к открытию полиморфизма через строгое использование указателей на функции.

ОО даёт, посредством поддержки полиморфизма, абсолютный контроль над всеми зависимостями в исходном коде. Факт поддержки языками полиморфизма означает, что любую зависимость исходного кода, где бы она ни находилась, можно инвертировать. Это позволяет архитектору создать архитектуру со сменными модулями (плагинами), в которой модули верхнего уровня не зависят от модулей нижнего уровня.

Низкоуровневые детали не выходят за рамки модулей плагинов, которые можно развертывать и разрабатывать независимо от модулей верхнего уровня.

1. **Функциональное программирование: история, особенности, применение.**

Парадигма функционального программирования является прямым результатом работы Алонзо Чёрча, который в 1936 году изобрел лямбда-исчисление (λ-исчисление). Его λ-исчисление легло в основу языка LISP, изобретенного в 1958 году Джоном Маккарти.

Основополагающим понятием λ-исчисления является неизменяемость — то есть невозможность изменения значений символов. Фактически это означает, что функциональный язык не имеет инструкции присваивания. В действительности большинство функциональных языков обладает некоторыми средствами, позволяющими изменять значение переменной, но в очень ограниченных случаях.

Один из самых общих компромиссов, на которые приходится идти ради неизменяемости, — деление приложения или служб внутри приложения на изменяемые и неизменяемые компоненты.

Неизменяемые компоненты решают свои задачи исключительно функциональным способом, без использования изменяемых переменных. Они взаимодействуют с другими компонентами, не являющимися чисто функциональными и допускающими изменение состояний переменных.

1. **Факторы выбора парадигмы программирования при архитектурном проектирования информационной системы.**

Итоги:

1. Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления.
2. Структурное программирование накладывает ограничение на прямую передачу управления.
3. Функциональное программирование накладывает ограничение на присваивание.

Парадигмы говорят нам не столько что делать, сколько чего нельзя делать. Три парадигмы вместе отнимают у нас инструкции goto, указатели на функции и оператор присваивания. Если есть четкие ограничения — это лучше, чем невероятные возможности.

*Здесь представлен конспект (без сокращений) лекций 4.2 и 6 по ТСПП в формате ответов на экзаменационные вопросы (от 37 до 49 включительно). На некоторые вопросы не было дано ответа в лекциях, поэтому был использован gpt. А для некоторых тем, подробно описанных в лекциях не было вопросов, но так как на экзамене спрашивают не только по билетам, такие темы я выделил в раздел «Дополнительно» в конце файла.*

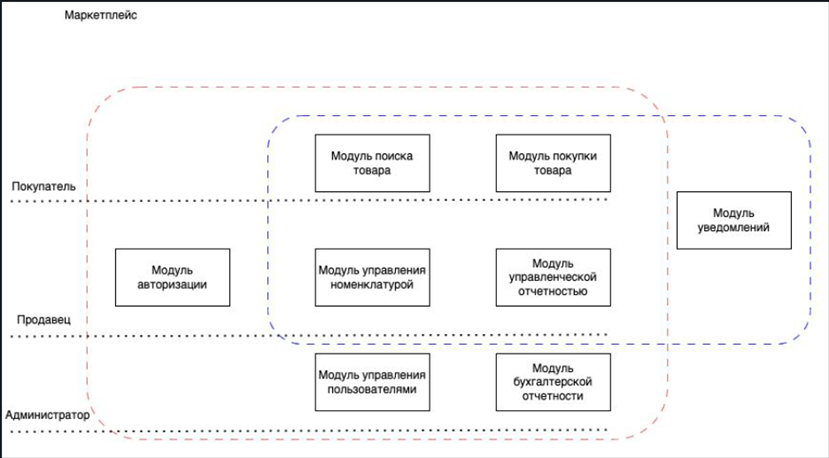
1. **Понятие монолитной архитектуры информационных систем. Основные принципы и особенности.**

Монолитная архитектура — это тип архитектуры программного обеспечения, при котором все компоненты приложения находятся в одной кодовом базе и выполняются как единое приложение.

Такой подход обеспечивает простоту развертывания и масштабирования приложения, так как все компоненты находятся в одном месте.

Однако, этот подход может приводить к проблемам, связанным с размером и сложностью кодовой базы, так как все компоненты находятся в одном месте.

Это может затруднять поддержку и разработку приложения на долгосрочной основе. Монолитная архитектура часто используется в небольших и средних проектах, где скорость разработки и простота развертывания являются более приоритетными, чем гибкость масштабирования и поддержки.



1. **Понятие микросервисной архитектуры информационных систем. Основные принципы и особенности.**

Микросервисная архитектура — это тип архитектуры программного обеспечения, при котором приложение разбивается на небольшие, автономные сервисы, каждый из которых отвечает за выполнение определенной функции.

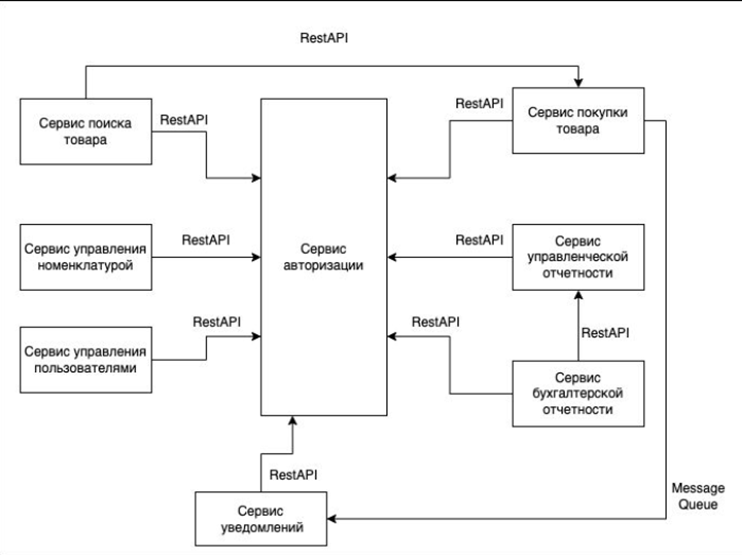
каждый сервис работает в своем собственном процессе и взаимодействует с другими сервисами через сетевые протоколы, такие как HTTP или Message Queuing

Такой подход обеспечивает гибкость, масштабируемость и легкость развертывания приложения, так как каждый сервис можно масштабировать и развертывать независимо от других сервисов.

Кроме того, это позволяет разработчикам использовать различные языки программирования и технологии для каждого сервиса, что может увеличить эффективность и гибкость разработки.

Однако, микросервисная архитектура может приводить к сложности в управлении и координации сервисов между собой, а также к проблемам при обработке транзакций и обеспечении целостности данных.

Кроме того, этот подход может быть избыточным для небольших проектов, где простота развертывания и управления являются более важными, чем гибкость масштабирования и поддержки.



1. **Сравнение монолитной и микросервисной архитектуры. Преимущества микросервисной архитектуры перед монолитной.**

В монолитной архитектуре все компоненты системы расположены в одной кодовой базе, а в микросервисах, кодовая база разбивается на сервисы, которые выполняют определенную задачу и общаются через разные протоколы.

Монолитная обеспечивает простоту развертывания и масштабирования приложения, так как все компоненты находятся в одном месте. Однако, этот подход может приводить к проблемам, связанным с размером и сложностью кодовой базы, так как все компоненты находятся в одном месте. Это может затруднять поддержку и разработку приложения на долгосрочной основе. Поэтому в больших проектах используют микросервисы, так как такой подход обеспечивает гибкость, масштабируемость и легкость развертывания приложения, так как каждый сервис можно масштабировать и развертывать независимо от других сервисов. Кроме того, это позволяет разработчикам использовать различные языки программирования и технологии для каждого сервиса, что может увеличить эффективность и гибкость разработки.

1. **Проблемы разработки и поддержки монолитной и микросервисной архитектуры программных продуктов.**

В монолитной архитектуре все компоненты приложения находятся в одной кодовом базе и работают в одном процессе операционной системы.

Это может привести к следующим проблемам:

1. Сложность масштабирования: монолитные приложения сложно масштабировать, поскольку все компоненты находятся в одном процессе. Это означает, что вы не можете масштабировать каждый компонент независимо, а вынуждены масштабировать всю систему целиком.
2. Ограниченная гибкость: монолитные приложения имеют ограниченную гибкость, поскольку все компоненты находятся в одном кодовом базе. Это значит, что вы не можете использовать различные технологии или языки программирования для каждого компонента, что может ограничивать возможности разработки.
3. Трудность в обновлении: обновление монолитных приложений может быть сложным, поскольку все компоненты находятся в одном кодовом базе. Это означает, что обновление одного компонента может повлиять на другие компоненты, что может привести к нежелательным последствиям.

Для решения этих проблем стоит начать выделять из системы компоненты, которые могут и должны существовать, как независимые сервисы и выполнять одну узконаправленную функцию. Для этого нужно: Документация, Аналитика, Ключевые метрики, Стек технологий

Разработка микросервисной архитектуры может столкнуться с несколькими проблемами *(это выдал GPT)*:

1. Комплексность: Микросервисная архитектура может содержать множество небольших сервисов, что может сделать ее сложной для понимания и поддержки. Кроме того, необходимо убедиться, что каждый сервис выполняет свою задачу и взаимодействует с другими сервисами и компонентами системы правильным образом.
2. Непредсказуемость: В микросервисной архитектуре каждый сервис может быть разработан и развернут независимо от других сервисов. Это может привести к тому, что различные сервисы будут иметь разные версии, что может создать проблемы во время интеграции.
3. Управление данными: Микросервисная архитектура может привести к тому, что данные будут распределены по различным сервисам, что может сделать сложным управление данными и их консистентностью. Также может возникнуть проблема синхронизации данных между различными сервисами.
4. Безопасность: В микросервисной архитектуре необходимо убедиться, что каждый сервис защищен от несанкционированного доступа, а также что межсервисные связи защищены.
5. Тестирование: Тестирование микросервисной архитектуры может быть сложным, поскольку каждый сервис может иметь свои собственные зависимости и требования к окружению. Это может сделать тестирование настолько сложным, что приложение может быть запущено только в ограниченных условиях.

Эти проблемы могут быть решены с помощью правильного проектирования архитектуры, использования инструментов автоматизации и тестирования, а также управления версиями и мониторингом сервисов.

1. **Основные подходы к разделению функциональности в микросервисной архитектуре программных продуктов.**

*(Такой формулировки в лекциях нет, вот что выдал GPT на этот запрос:)*

Существует несколько подходов к разделению функциональности в микросервисной архитектуре:

1. Разделение по бизнес-функциям: Каждый микросервис выполняет определенную бизнес-функцию. Например, сервис, отвечающий за обработку платежей, может быть выделен в отдельный микросервис.

2. Разделение по данным: Каждый микросервис управляет определенным типом данных. Например, микросервис, отвечающий за управление информацией о клиентах, может быть выделен в отдельный сервис.

3. Разделение по границам контекста: Каждый микросервис управляет определенной областью бизнес-контекста. Например, микросервисы, отвечающие за управление заказами и управление складом, могут быть выделены в отдельные сервисы, поскольку они относятся к разным областям бизнеса.

4. Разделение по технологиям: Каждый микросервис использует определенные технологии и языки программирования. Например, микросервис, реализованный на Java, может быть выделен в отдельный сервис, а микросервис, реализованный на Python, может быть выделен в другой сервис.

5. Разделение по доступности: Каждый микросервис может быть развернут на отдельном сервере или в отдельном контейнере, что обеспечивает изолированную доступность и улучшает масштабируемость системы.

1. **Критерии выбора между монолитной и микросервисной архитектурой информационной системы.**

*(Опять же, вопрос повторяется, можно ответить исходя из особенностей данных архитектур, что описаны в ответах на прошлые вопросы, если резюмировать, то вот:)*

Монолитная архитектура часто используется в небольших и средних проектах, где скорость разработки и простота развертывания являются более приоритетными, чем гибкость масштабирования и поддержки. В случае, когда проект больше среднего и более приоритетна масштабируемость и гибкость проекта, стоит использовать микросервисы.

1. **Понятие архитектуры API информационной системы, основные принципы и цели. Виды API информационных систем.**

API, или Application Programming Interface, представляет собой набор методов, функций и протоколов, которые разработчики могут использовать для взаимодействия с программным обеспечением или веб-сервисами.

Он обеспечивает структурированный способ обмена данными и выполнения операций между различными компонентами программного обеспечения

Он определяет правила и форматы запросов и ответов, что делает взаимодействие между приложениями стандартизированным и удобным.

Клиент-серверная архитектура является распространенным подходом к разработке приложений, где клиентские приложения, такие как веб-браузеры или мобильные приложения, взаимодействуют с сервером, например, веб-сервером или базой данных.

API играет решающую роль в этой архитектуре, обеспечивая стандартизированный способ взаимодействия между клиентом и сервером.

Приложения могут отправлять запросы к API для выполнения определенных операций или получения доступа к данным на сервере. API обрабатывает эти запросы и возвращает соответствующие ответы, содержащие данные или результаты операций. Таким образом, API действует как посредник между клиентом и сервером, обеспечивая эффективную коммуникацию и передачу данных.

Правильно спроектированный API обеспечивает прозрачность и независимость между клиентом и сервером, позволяя разработчикам создавать разнообразные клиентские приложения, которые могут взаимодействовать с сервером без необходимости знать подробности его реализации.

Типы API:

* Веб-сервисы
* RESTful API (Representational State Transfer)
* SOAP (Simple Object Access Protocol)

Есть и другие типы API, такие как GraphQL, который предоставляет клиентам возможность запрашивать только нужные данные и избегать избыточности, а также множество проприетарных и специфических API, разработанных для конкретных приложений или сервисов. Понимание различных типов API позволяет разработчикам выбирать наиболее подходящий под их потребности. Каждый тип API имеет свои особенности, и выбор должен быть основан на требованиях проекта и целях взаимодействия клиентского и серверного приложений.

Основные принципы API:

*Прозрачность и независимость*

Один из ключевых принципов архитектуры API — это обеспечение прозрачности и независимости между клиентскими и серверными компонентами. API должен предоставлять абстракцию, скрывающую детали реализации сервера и позволяющую клиентским приложениям взаимодействовать с сервером без необходимости знать его внутреннюю структуру или технологии, используемые на стороне сервера. Прозрачность и независимость обеспечивают гибкость и упрощают сопровождение и развитие приложений.

*Единообразие интерфейса*

Другой важный принцип — это обеспечение единообразного интерфейса API. Клиентские приложения должны иметь возможность предсказуемо и последовательно взаимодействовать с различными частями API. Это достигается путем определения стандартных методов и форматов запросов и ответов, которые поддерживаются API. Единообразный интерфейс облегчает разработку клиентских приложений и интеграцию с различными сервисами и системами.

*Масштабируемость и гибкость*

API должен быть спроектирован с учетом масштабируемости и гибкости Масштабируемость обеспечивает возможность обрабатывать большой объем запросов и поддерживать растущее количество клиентов. Гибкость позволяет добавлять новые функциональные возможности или модифицировать существующие без нарушения совместимости с уже существующими клиентами. Гибкость API может быть достигнута, например, путем версионирования API или предоставления возможностей настройки и расширения.

*Безопасность и аутентификация*

Безопасность - важный аспект архитектуры API. API должен предоставлять механизмы для аутентификации и авторизации клиентских приложений. Это обеспечивает контроль доступа и защиту данных от несанкционированного использования. API должен также поддерживать безопасность передачи данных по сети путем использования шифрования и других механизмов защиты. Понимание и применение этих основных принципов архитектуры API позволяет разработчикам создавать надежные, гибкие и безопасные интерфейсы, которые легко масштабируются и поддерживаются в клиент-серверных приложениях.

1. **Понятие и основные принципы построения веб-сервиса.**

Веб-сервисы предоставляют средства для взаимодействия между различными приложениями через сеть, обычно используя протоколы HTTP. Они работают на основе клиент-серверной модели, где клиентские приложения могут отправлять запросы к веб-сервисам и получать ответы с данными или результатами операций

*(В лекции есть описание построения клиент-серверной модели, а не веб-сервисов, но так как веб-сервисы построены на ней, думаю оно будет уместно:)*

Клиент-серверная модель представляет собой способ организации взаимодействия между компьютерами в сети, где один компьютер выступает в роли клиента, а другой - в роли сервера. Клиентский компьютер обычно является пользовательским устройством, таким как персональный компьютер, мобильное устройство или веб-браузер. Клиентские приложения отправляют запросы к серверу, запрашивая определенные данные или операции. Сервер, в свою очередь, обрабатывает эти запросы и возвращает ответы с необходимыми данными или результатами операций.

*(В разделе «Дополнительно» есть плюсы и минусы клиент-серверной модели, а также протоколы и форматы данных, которые поддерживают веб-сервисы)*

1. **Понятие SOAP (Simple Object Access Protocol). Основные характеристики и принципы работы SOAP-веб-сервисов. Особенности архитектурного проектирования информационных систем на основе SOAP-веб-сервисов.**

SOAP (Simple Object Access Protocol) — это протокол обмена структурированными сообщениями, который используется для обмена данными между веб-сервисами.

SOAP определяет формат сообщений, включая XML (Extensible Markup Language) для описания данных, операций и ошибок. SOAP может использоваться с различными протоколами передачи, такими как HTTP, SMTP и другими.

Он обеспечивает надежность, безопасность и расширяемость взаимодействия между клиентом и сервером.

*(Это буквально всё, что описано про SOAP в лекции, также где-то говорится, что он является одним из протоколов веб-сервисов, а в 43 вопросе он выделяется как отдельный тип API. Для полноты информации, вставляю ответ на вопрос от GPT:)*

*Основные характеристики и принципы работы SOAP-веб-сервисов:*

1. Строгая типизация: В SOAP-сообщениях используется строгая типизация, что позволяет избежать ошибок при передаче данных между приложениями.
2. Использование XML: SOAP-сообщения представлены в формате XML, что обеспечивает удобство парсинга данных и обработки ошибок.
3. Использование WSDL: для описания интерфейса SOAP-веб-сервисов используется язык WSDL (Web Services Description Language), который позволяет клиентам получать информацию о доступных операциях и параметрах веб-сервиса.
4. Использование протокола HTTP: SOAP-сообщения обычно передаются через протокол HTTP, что обеспечивает удобство взаимодействия между клиентом и сервером.
5. Безопасность: SOAP-сообщения могут быть защищены с помощью различных механизмов, таких как SSL-шифрование и аутентификация.

*Особенности архитектурного проектирования информационных систем на основе SOAP-веб-сервисов:*

1. Централизованная архитектура: SOAP-веб-сервисы могут использоваться для построения централизованных информационных систем, в которых все приложения обмениваются данными через один или несколько веб-сервисов.
2. Может быть медленным: SOAP-сообщения могут быть более сложными и медленными, чем сообщения в других протоколах, таких как REST.
3. Широкая поддержка: SOAP-протокол поддерживается большинством языков программирования и платформ, что делает его удобным для использования в различных средах.
4. Большой объем сообщений: SOAP-сообщения могут быть более крупными, чем сообщения в других протоколах, что может быть неэффективно в случае передачи большого объема данных.

В целом, SOAP-веб-сервисы могут быть эффективным решением для построения централизованных информационных систем, которые должны обмениваться данными между различными приложениями на различных платформах. Однако, в случае множества распределенных систем, REST-архитектура может быть более подходящей.

1. **Понятие, концепция и основные принципы проектирования RESTful API.**

RESTful API (Representational State Transfer) — это стиль архитектуры API, основанный на наборе принципов и ограничений, которые позволяют создавать гибкие и масштабируемые веб-сервисы.

RESTful API использует протокол HTTP(S) (Hypertext Transfer Protocol) для обмена данными и операций между клиентом и сервером.

RESTful API становится все более популярным и широко применяется в различных веб-приложениях и мобильных приложениях.

Основные принципы:

1. Клиент-серверная архитектура: API разделяет клиентскую часть (которая отправляет запросы) и серверную часть (которая обрабатывает запросы и отправляет ответы).
2. Без состояния (stateless): Каждый запрос клиента к серверу должен содержать всю необходимую информацию, без сохранения состояния на стороне сервера. Сервер не должен запоминать информацию о предыдущих запросах.
3. Кэширование: RESTful API поддерживает кэширование, что позволяет клиентам сохранять локальные копии ресурсов и использовать их при необходимости.
4. Единообразный интерфейс: API должен иметь унифицированный набор методов и форматов для обмена данными. Например, использование стандартных HTTP-методов и формата данных, таких как JSON или XML.
5. Слой системы: RESTful API может быть организован в виде слоев, где каждый слой выполняет определенные функции и добавляет дополнительную функциональность.
6. **Характеристика и назначение HTTP-методов, применяемых в REST API. Понятие, способы представления и обработки ресурсов в контексте REST API.**

RESTful API использует различные методы HTTP для определения типа операции, которую клиент хочет выполнить над ресурсом. Некоторые из наиболее распространенных методов включают:

* GET: используется для получения данных с сервера. Клиент отправляет запрос на получение ресурса, и сервер возвращает запрошенные данные.
* POST: используется для создания нового ресурса на сервере. Клиент отправляет данные на сервер, и сервер создает новый ресурс на основе предоставленных данных.
* PUT: используется для обновления существующего ресурса на сервере. Клиент отправляет данные на сервер, и сервер обновляет соответствующий ресурс с новыми данными.
* DELETE: используется для удаления существующего ресурса на сервере. Клиент отправляет запрос на удаление, и сервер удаляет указанный ресурс.

Примеры использования:

* Получение информации о пользователях: Клиент может отправить GET-запрос на URL-адрес, предоставляющий информацию о пользователях, и получить список пользователей в формате JSON или XML.
* Создание новой записи: Клиент может отправить POST-запрос на URL-адрес, предоставляющий возможность создания новой записи, и передать данные новой записи.
* Обновление существующей записи: Клиент может отправить PUT-запрос на URL-адрес, предоставляющий возможность обновления существующей записи, и передать новые данные для обновления.
* Удаление записи: Клиент может отправить DELETE-запрос на URL-адрес, предоставляющий возможность удаления записи, и указать идентификатор удаляемой записи.

*(Про ресурсы в лекции опять таки ни слова, поэтому обратился к GPT:)*

В контексте REST API, ресурс — это любой объект, который может быть идентифицирован уникальным URI (Uniform Resource Identifier) и доступен через стандартные HTTP-методы, такие как GET, POST, PUT, DELETE. Ресурсы могут быть представлены различными типами данных, такими как HTML, XML, JSON и другими.

Примерами ресурсов могут быть:

* Конкретный объект, такой как пользователь, товар или заказ.
* Коллекция объектов, такая как список пользователей, каталог товаров или список заказов.
* Некоторое действие, которое может быть выполнено над объектом или коллекцией, такое как поиск, сортировка или фильтрация.

1. **Преимущества RESTful API по сравнению с другими подходами к созданию API. Способы поддержания состояния сеанса между клиентом и сервером RESTful API.**

*(Люблю лекции, опять ни слова ни про то, ни про другое 😊, вот что выдал GPT:)*

RESTful API имеет несколько преимуществ по сравнению с другими подходами к созданию API:

* Простота: RESTful API основан на стандартных протоколах и форматах данных, таких как HTTP и JSON, что делает его легким в использовании и понимании.
* Гибкость: RESTful API позволяет создавать гибкие и масштабируемые веб-сервисы, которые могут быть использованы в различных сценариях и средах.
* Быстрота: RESTful API обеспечивает быстрое взаимодействие между клиентом и сервером благодаря использованию стандартных протоколов и форматов данных.
* Легкость кеширования: RESTful API использует стандартные HTTP-методы, которые обеспечивают возможность кеширования данных на стороне клиента.
* Независимость от языка программирования: RESTful API не зависит от конкретного языка программирования и может быть использован с любой платформой и языком.

Способы поддержания состояния сеанса между клиентом и сервером RESTful API зависят от конкретного приложения и могут быть различными. В RESTful API, состояние сеанса может быть поддержано с помощью токенов доступа (access tokens, jwt токены) или куки (cookies). Токены доступа могут быть выданы сервером и использованы клиентом для выполнения последующих запросов, в то время как куки могут быть использованы для хранения данных о состоянии сеанса на стороне клиента.

Также, в RESTful API, можно использовать параметры запроса (query parameters) для передачи данных между клиентом и сервером, например, для фильтрации или сортировки ресурсов. Однако, в RESTful API, не используется поддержание состояния сеанса на стороне сервера, как, например, в SOAP-веб-сервисах

1. **Форматы данных для обмена информацией в RESTful API.**

*(Тут говорится в общем, для любого API, думаю для REST тоже подойдёт)*

Наиболее распространенными форматами данных являются JSON (JavaScript Object Notation) и XML (Extensible Markup Language).

JSON является легковесным и простым для чтения и записи, в то время как XML обладает более строгой структурой и поддерживает более широкий спектр типов данных.

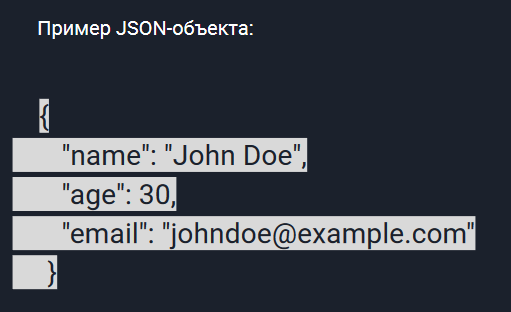
Выбор формата данных зависит от требований проекта, типов данных, которые будут передаваться, и совместимости с другими системами.

1. JSON (JavaScript Object Notation):

JSON — это легковесный формат данных, основанный на синтаксисе JavaScript. Он прост для чтения и записи как людьми, так и компьютерами.

JSON использует простую структуру данных, состоящую из пар ключ-значение, которые могут быть вложенными. Основные преимущества JSON включают:

* Легкость использования: JSON предоставляет простой и понятный синтаксис, который легко читается и пишется. Это делает его идеальным выбором для многих разработчиков.
* Высокая производительность: Парсинг и сериализация JSON происходят очень быстро, что делает его эффективным во время обмена данными между клиентом и сервером.
* Поддержка различных языков программирования: JSON поддерживается большинством современных языков программирования, что облегчает интеграцию API с различными технологиями.



2. XML (Extensible Markup Language):

XML — это язык разметки, который предоставляет гибкую структуру для представления данных. XML использует теги для определения элементов и атрибуты для хранения значений.

Основные преимущества XML включают:

* Гибкость и расширяемость: XML позволяет создавать сложные иерархические структуры данных, которые могут быть легко расширены или изменены без нарушения совместимости.
* Возможность описания схемы данных: XML схемы (XSD) позволяют определить строгую структуру данных и типы, что способствует более строгой валидации данных.
* Поддержка для различных языков и кодировок: XML поддерживает различные кодировки и может быть использован с любым языком программирования

*(Про HTML не уверен, выдала нейронка)*

3.HTML (Hypertext Markup Language) - это текстовый формат обмена данными,который используется для представления документов в Интернете. HTML может быть использован для передачи данных в формате веб-страницы.

**ДОПОЛНИТЕЛЬНО**

**способы построения архитектуры:**

1. Событийно-ориентированная архитектура (Event-Driven Architecture, EDA): приложение разбивается на отдельные компоненты, каждый из которых реагирует на определенные события и взаимодействует с другими компонентами через обмен сообщениями.

2. Сервер-клиентская архитектура (Client-Server Architecture): приложение разбивается на клиентскую часть, которая предоставляет пользовательский интерфейс, и серверную часть, которая обрабатывает запросы и управляет данными.

3. Архитектура, основанная на контейнерах (Container-Based Architecture): приложение разбивается на независимые контейнеры, каждый из которых выполняет определенную функцию и может масштабироваться независимо.

4. Слоистая архитектура (Layered Architecture): приложение разбивается на слои, каждый из которых выполняет определенную функцию и взаимодействует только с более близкими к нему слоями.

5. Архитектура, основанная на событийных потоках (Event-Streaming Architecture):

приложение разбивается на отдельные компоненты, каждый из которых обрабатывает события и передает результаты другим компонентам через событийные потоки.

6. Архитектура, основанная на графах (Graph-Based Architecture): приложение

представляет собой граф, в котором узлы представляют компоненты, а ребра – связи между ними.

**Роль API в клиент-серверной модели**

API играет важную роль в клиент-серверных приложениях. Он предоставляет стандартизированный интерфейс, через который клиенты могут взаимодействовать с сервером. API определяет доступные операции и функциональные возможности, а также форматы запросов и ответов.

Он обеспечивает прозрачность и независимость между клиентскими и серверными компонентами, позволяя разработчикам создавать различные клиентские приложения, которые могут взаимодействовать с сервером, не заботясь о его внутренней реализации. API обеспечивает удобный и стандартизированный способ коммуникации между клиентом и

сервером. Клиентские приложения могут использовать API для отправки запросов на выполнение операций или получения данных с сервера. API обрабатывает эти запросы, взаимодействуя с соответствующими компонентами на стороне сервера, и возвращает ответы клиенту

**Преимущества клиент-серверной модели**

* Распределение нагрузки: Сервер может обрабатывать множество запросов от различных клиентов, что позволяет распределить нагрузку и повысить производительность системы.
* Централизованное управление данными: Сервер может хранить и управлять данными, обеспечивая централизованный доступ и поддержку целостности данных.
* Улучшенная безопасность: Клиентские приложения могут обращаться к серверу для проверки доступа и выполнения безопасных операций, что способствует защите данных и ресурсов.

**Недостатки клиент-серверной модели**

* Единственная точка отказа: Если сервер перестает работать, клиентские приложения могут потерять доступ к функциональности или данным.
* Зависимость от сети: Клиентским приложениям требуется постоянное подключение к сети для взаимодействия с сервером, что может вызывать проблемы в случае неполадок или ограниченной пропускной способности сети.
* Распределение версий: Если клиент и сервер разрабатываются и обновляются независимо друг от друга, может возникнуть проблема с несовместимостью версий API.

**Протоколы и форматы, поддерживаемые веб-сервисами *(кроме SOAP, он был описан в вопросе 45)***

2. WSDL (Web Services Description Language)

WSDL (Web Services Description Language) - это язык описания веб-сервисов. Он предоставляет формальное описание доступных операций и данных, которые могут быть использованы клиентскими приложениями.

WSDL использует XML для определения структуры и типов данных, а также для описания доступных операций и методов. Он позволяет клиентам автоматически генерировать код для взаимодействия с веб-сервисом и облегчает интеграцию различных приложений.

3. UDDI (Universal Description, Discovery, and Integration)

UDDI (Universal Description, Discovery, and Integration) - это стандартный протокол и реестр, используемый для обнаружения и регистрации веб-сервисов. UDDI позволяет разработчикам и организациям находить и использовать доступные веб-сервисы, обеспечивая централизованное хранение информации о сервисах, их описаниях и доступных операциях. UDDI облегчает интеграцию и повторное использование веб-сервисов в различных приложениях.

**Определение требований. Методы и подходы**

1. Исследование рынка и анализ конкурентов: изучение существующих API в смежных отраслях и конкурирующих продуктах может помочь определить общие стандарты и лучшие практики. Это также может помочь выявить преимущества и недостатки существующих решений, чтобы создать более привлекательное и конкурентоспособное API.

2. Обратная связь от потенциальных пользователей: важно обратиться к потенциальным пользователям API, провести опросы, собрать обратную связь и учесть их потребности и предпочтения. Это может помочь определить необходимые функции, типы данных, методы взаимодействия и другие аспекты API, которые будут наиболее полезными и удобными для пользователей.

3. Коллективное обсуждение с командой разработчиков: вовлечение команды разработчиков в процесс определения требований позволяет объединить различные точки зрения и экспертизу, чтобы определить наиболее эффективные и реалистичные требования. Регулярные совещания, мозговые штурмы и обсуждения могут помочь выявить потенциальные проблемы и найти наилучшие решения.

4. Проектирование прототипов и MVP (минимально жизнеспособного продукта): Создание прототипов и MVP позволяет быстро протестировать и визуализировать основные функции API. Это помогает лучше понять, как API будет использоваться и какие требования к нему предъявляются. Прототипирование и MVP также

позволяют собрать обратную связь от пользователей и внести коррективы в ранней стадии разработки.

При определении требований к API важно учесть бизнес-цели, потребности пользователей и технические возможности.

Комбинация методов, средств и подходов позволяет создать эффективное, удобное и гибкое API, которое будет успешно интегрироваться и использоваться другими системами и разработчиками.

**Модульность и версионирование API**

При проектировании API рекомендуется использовать модульный подход, разделяя функциональность на небольшие и самодостаточные модули.

Модульность облегчает разработку, тестирование и поддержку API, а также позволяет более гибко вносить изменения и расширять функциональность в дальнейшем.

Кроме того, важно предусмотреть механизм версионирования API, чтобы обеспечить совместимость существующих клиентов и позволить внесение изменений в API без нарушения работы клиентских приложений.

**Документация и описание API**

Хорошо задокументированное и описанное API является ключевым аспектом успешной разработки.

Документация API должна содержать информацию о доступных ресурсах, операциях, форматах данных, параметрах запросов и ответах, а также примеры использования.

Четкая и подробная документация помогает разработчикам быстро разобраться в использовании API, снижает вероятность ошибок и облегчает интеграцию с другими системами.

Кроме того, описание API может включать спецификации и схемы данных, такие как OpenAPI (ранее известный как Swagger), которые облегчают автоматическую генерацию клиентского кода и инструментов для работы с API.