Вопросы	ОТВЕТЫ			
Перечислите и кратко охарактеризуйте этапы разработки процесса ПО?	Типовой проект включает в себя следующие этапы разработки программного обеспечения: анализ требований к проекту; проектирование; реализация; тестирование продукта; внедрение и поддержка. Анализ требований к проекту На этом этапе формулируются цели и задачи проекта, выделяются базовые сущности и взаимосвязи между ними. То есть, создается основа для дальнейшего проектирования системы. В рамках данного этапа не только фиксируются требования заказчика, но и проводится их формирование – клиентам подбирается оптимальное решение их проблем, определяется необходимая степень автоматизации, выявляются наиболее актуальные для автоматизации бизнес-процессы. При анализе требований определяются сроки и стоимость разработки ПО, формируется и подписывается ТЗ на разработку программного обеспечения. Проектирование На основе предыдущего этапа проводится проектирование системы. Эта методология проектирования соединяет в себе объектную декомпозицию, приемы представления физической, логической, а также динамической и статической моделей системы. Во время проектирования разрабатываются проектные решения по выбору платформы, где будет функционировать система языка или языков реализации, назначаются требования к пользовательскому интерфейсу, определяется наиболее подходящая СУБД. Разрабатывается функциональная спецификация ПО: выбирается архитектура системы, оговариваются требования к аппаратному обеспечению, определяется набор орг. мероприятий, которые необходимы для внедрения ПО, а также перечень документов, регламентирующих его использование. Реализация Данный этап разработки программного обеспечения организован в соответствии с моделями эволюционного типа жизненного цикла ПО. При разработке применяются экспериментирование и анализ, строятся прототипы, как целой системы, так и ее частей. Прототипы дают возможность глубже вникнуть в проблему и принять все необходимые проектные решения еще на ранних этапах проектирования. Такие решения могут затрагивать разные части системы: внутреннюю организацию, пользовательский интерфейс, разграничение доступа и т.д. В результата этапа реализации появляется рабочая версия продукта. Тестирование продукта Тестирование тесно связано с такими этапами разработки программного обеспечения как проектирование и реализация. В систему встраиваются специальные механизмы, которые дают возможность производить тестирование системы на соответствие требований к ней, проверку оформления и наличие необходимого пакета документации. Результатом тестирования является устранение всех недостатков системы и заключение о ее качестве. Внедрение и поддержка Внедрения системы обычно предусматривает следующие шаги: установка системы, обучение пользователей, эксплуатация. К любой разработке прилагается полный пакет документации, который включает в себя описание системы, руководства пользователей и алгоритмы работы. Поддержка функционирования ПО должна осуществляться группой технической поддержки разработчика.			
Назовите известные Вам модели разработки процесса ПО? Краткая характеристика.	https://habrahabr.ru/company/edison/blog/269789/ 1. Waterfall 2. V-Model 3. Incremental 4. RAD 5. Agile 6. Iterative 7. Spiral			

	Объектно-ориентированное программирование - методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования **Функциональное программирование** - способ организации вычислений без состояния **Протокол-ориентированное программирование - наследование зло и пережиток прошлого, один из новых трендов - протоколы. Не нужно создавать наследника наследника наследника наследника одного base-класса, нужно создать протоколы, которые являеются расширяемыми - можно добавлять к протоколам методы и переменные.** **Императивное программирование** В этой парадигме вычисления описываются в виде инструкций, шаг за шагом изменяющих состояние программы. В низкоуровневых языках (таких как язык ассемблера) состоянием могут быть память, регистры и флаги, а инструкциями — те команды, что поддерживает целевой процессор. В более высокоуровневых (таких как Си) состояние — это только память, инструкции могут быть сложнее и вызывать выделение и освобождение памяти в процессе своей работы. В совсем высокоуровневых (таких как Python, если на нем программировать императивно) состояние ограничивается лишь переменными, а команды могут представлять собой комплексные операции, которые на ассемблере занимали бы сотни строк. **Структурное программирование** Эта парадигма вводит новые понятия, объединяющие часто используемые шаблоны написания императивного кода. В структурном программировании мы по прежнему оперируем состоянием и инструкциями, однако вводится понятие составной инструкции (блока), инструкций ветвления и цикла. Благодаря этим простым изменениям возможно отказаться от оператора goto в большинстве случаев, что упрощает код. Иногда goto все-же делает код читабельнее, благодаря чему он до сих пор широко используется, несмотря на все заявления его противников. **Процедурное программирование** Опять-же возрастающая сложность программного обеспечения заставила программистов искать другие способы описывать вычисления. Собственно еще раз были введены дополнительные понятия, которые позволили по-новому взглянуть на программирование. Этим понятием на этот раз была процедура. В результате возникла новая методология написания программ, которая приветствуется и по сей день — исходная задача разбивается на меньшие (с помощью процедур) и это происходит до тех пор, пока решение всех конкретных процедур не окажется тривиальным. Процедура — самостоятельный участок кода, который можно выполнить как одну инструкцию. В современном программировании процедура может иметь несколько точек выхода (return в C-подобных языках), несколько точек входа (с помощью yield в Python или статических локальных переменных в C++), иметь аргументы, возвращать значение как результат своего выполнения, быть перегруженной по количеству или типу параметров и много чего еще. **Модульное программирование** Который раз увеличивающаяся сложность программ заставила разработчиков разделять свой код. На этот раз процедур было недостаточно и в этот раз было введено новое понятие — модуль. Забегая вперед скажу, что модули тоже оказались неспособны сдержать с невероятной скоростью растущую сложность ПО и в последствии появились пакеты (это тоже модульное программирование), классы (это уже ООП), шаблоны (обобщенное программирование). Программа описанная в стиле модульного программирования — это набор модулей. Что внутри, классы, императивный код или чистые функции — не важно. Благодаря модулям впервые в программировании появилась серьезная инкапсуляция — возможно использовать какие-либо сущности внутри модуля, но не показывать их внешнему миру.			
Назовите известные Вам методологии тестирования ПО?. Краткая характеристика каждого.	Методологии тестирования - некорректный вопрос. Есть разделение тестирования по объектам тестирования, по уровням, по степени автоматизации и прочему. Однако одна из самых широко используемых классификаций обычно по уровню: **1. Модульное тестирование (Unit T**esting) Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и **т.д.).** **2. Интеграционное тестирование (Integr**ation Testing) Проверяется взаимодействие между компонентами системы после проведения компонентного **тестирования.** **3. Системное тестиров**ание (System Testing) Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство **использования и т.д.** **4. Операционное тес**тирование (Release Testing). Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес моделе системы. Следует учесть, что и бизнес модель может содержать ошибки. Поэтому так важно провести операционное тестирование как финальный шаг валидации. Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения — критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО. **5. Приемочное тестирование (Acceptance Testing)** Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью: • определения удовлетворяет ли система приемочным критериям; • вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.			
Дайте определение понятиям «статическая и динамическая типизация» и приведите примеры языков с данными видами типизации?	https://habrahabr.ru/post/161205/ Статическая определяется тем, что конечные типы переменных и функций устанавливаются на этапе компиляции. Т.е. уже компилятор на 100% уверен, какой тип где находится. В динамической типизации все типы выясняются уже во время выполнения программы. Примеры: Статическая: C, Java, C#; Динамическая: Python, JavaScript, Ruby. *Преимущества статической типизации* *Проверки типов происходят только один раз — на этапе компиляции. А это значит, что нам не нужно будет постоянно выяснять, не пытаемся ли мы поделить число на строку (и либо выдать ошибку, либо осуществить преобразование).* *Скорость выполнения. Из предыдущего пункта ясно, что статически типизированные языки практически всегда быстрее динамически типизированных.* *При некоторых дополнительных условиях, позволяет обнаруживать потенциальные ошибки уже на этапе компиляции.* *Ускорение разработки при поддержке IDE (отсеивание вариантов, заведомо не подходящих по типу).* *Преимущества динамической типизации* *Простота создания универсальных коллекций — куч всего и вся (редко возникает такая необходимость, но когда возникает динамическая типизация выручит).* *Удобство описания обобщенных алгоритмов (например сортировка массива, которая будет работать не только на списке целых чисел, но и на списке вещественных и даже на списке строк).* *Легкость в освоении — языки с динамической типизацией обычно очень хороши для того, чтобы начать программировать.*			

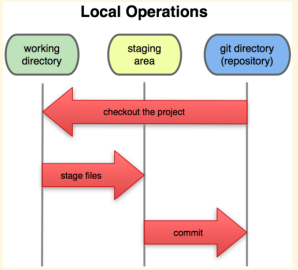
Дайте определение понятиям «сильная и слабая типизация» и приведите примеры языков с данными видами типизации?	https://habrahabr.ru/post/161205/ Сильная типизация выделяется тем, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например нельзя вычесть из строки множество. Языки со слабой типизацией выполняют множество неявных преобразований автоматически, даже если может произойти потеря точности или преобразование неоднозначно. Примеры: Сильная: Java, Python, Haskell, Lisp; Слабая: C, JavaScript, Visual Basic, PHP. *Преимущества сильной типизации* *Надежность — Вы получите исключение или ошибку компиляции, взамен неправильного поведения.* *Скорость — вместо скрытых преобразований, которые могут быть довольно затратными, с сильной типизацией необходимо писать их явно, что заставляет программиста как минимум знать, что этот участок кода может быть медленным.* *Понимание работы программы — опять-же, вместо неявного приведения типов, программист пишет все сам, а значит примерно понимает, что сравнение строки и числа происходит не само-собой и не по-волшебству.* *Определенность — когда вы пишете преобразования вручную вы точно знаете, что вы преобразуете и во что. Также вы всегда будете понимать, что такие преобразования могут привести к потере точности и к неверным результатам.* *Преимущества слабой типизации* *Удобство использования смешанных выражений (например из целых и вещественных чисел).* *Абстрагирование от типизации и сосредоточение на задаче.* *Краткость записи.*			
Дайте определение понятиям «явная и неявная типизация» и приведите примеры языков с данными видами типизации?	https://habrahabr.ru/post/161205/ Явно-типизированные языки отличаются тем, что тип новых переменных / функций / их аргументов нужно задавать явно. Соответственно языки с неявной типизацией перекладывают эту задачу на компилятор / интерпретатор. Примеры: Явная: C++, D, C# Неявная: PHP, Lua, JavaScript *Преимущества явной типизации* *Наличие у каждой функции сигнатуры (например int add(int, int)) позволяет без проблем определить, что функция делает.* *Программист сразу записывает, какого типа значения могут храниться в конкретной переменной, что снимает необходимость запоминать это.* *Преимущества неявной типизации* *Сокращение записи — def add(x, y) явно короче, чем int add(int x, int y).* *Устойчивость к изменениям. Например если в функции временная переменная была того-же типа, что и входной аргумент, то в явно типизированном языке при изменении типа входного аргумента нужно будет изменить еще и тип временной переменной.*			
Три кита ООП? Привести примеры инкапсуляции, наследования полиморфизма?	https://habrahabr.ru/post/148015/ Инкапсуляция Наследование Полиморфизм *+ Абстракция* 1. Наследование. Есть пекарь. Есть печь электрическая и газовая. Ваша задача смоделировать процесс приготовления пищи пекарем в каждой из печи. Решая задачу в лоб, у нас будет много дублирования кода из-за того, что сам процесс передачи пищи в печь и сама работа с печами идентичны для обеих печей. Но если мы включаем объектное мышление, и вспоминаем про инструмент наследование, то получаем примерно следующее (диаграмму лень рисовать, сорри): Есть печь (абстрактная печь). У нее есть поведение — включить, выключить, увеличить или уменьшить температуру, положить чего-то, достать чего-то и состояние — температура в печи, включена или выключена. Это отличный пример абстрактного объекта в котором соблюдены принципы инкапсуляции (при реализации я их обязательно буду соблюдать). И есть пекарь, конкретный такой пекарь Иван. Он умеет работать с абстрактной печью. Т.е. смотреть температуру, включать выключать и т.д. вы поняли. Сила наследования в том, что нам не придется переписывать нашего Ивана для каждой из печей, будь то электро или газовая печь. Я думаю всем ясно почему? Получается что инструмент применен правильно. 2. Полиморфизм. Печи ведь по-разному работают. Газовая потребляет газ, электро печь — электричество. Используя полиморфизм мы легко меняем поведение в наследниках абстрактной печи. 3. Инкапсуляция. Основная фишка инкапсуляции в том, что я не должен знать, что происходит внутри моей печи. Допустим, я вызываю не метод включить печь, а меняю ее свойство включена на значение true. Что произойдет в этот момент? Если принцип инкапсуляции не соблюден, то я буду вынужден печи сказать начинай потреблять горючее, т.к. я тебя включил. Т.е. пекарь знает, что печь потребляет горючее, знает, как печь работает. Или, например, мы не можем установить температуру печи ниже или выше определенного уровня. Если не соблюдать принцип инкапсуляции, то мы должны будем говорить печи проверь-ка текущую температуру, пойдет те такая? Т.е. пекарь опять слишком много знает о печи. Геттеры и сеттеры это средства языка, которые помогут нам легко реализовать отслеживание изменений состояния. Все. Если геттеры и сеттеры пустые, значит так надо на моем уровне абстракции. Геттеры и сеттеры — не могут мешать реализации инкапсуляции, криво реализовать инкапсуляцию может проектировщик/программист.			

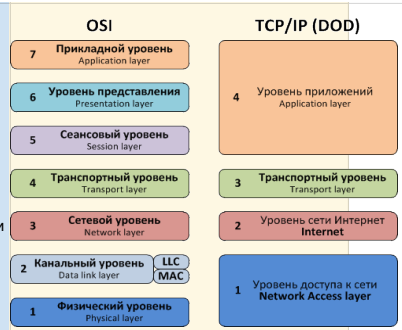
Абстрактные классы и интерфейсы?	Абстрактный класс — это класс, у которого не реализован один или больше методов (некоторые языки требуют такие методы помечать специальными ключевыми словами). Интерфейс — это абстрактный класс, у которого все методы не реализованы, все публичные и нет переменных класса. Интерфейс нужен обычно, когда описывается только интерфейс (тавтология). Например, один класс хочет дать другому возможность доступа к некоторым своим методам, но не хочет себя "раскрывать". Поэтому он просто реализует интерфейс. Абстрактный класс нужен, когда нужно семейство классов, у которых есть много общего. Конечно, можно применить и интерфейс, но тогда нужно будет писать много идентичного кода. В некоторых языках (C++) специального ключевого слова для обозначения интерфейсов нет. Можно считать, что любой интерфейс — это уже абстрактный класс, но не наоборот.	
Дайте определение понятию «структура данных»? Какие структуры данных вы знаете в чём их особенности?	Что такое структуры данных? По сути, это способы хранить и организовывать данные, чтобы эффективней решать различные задачи. Структуры данных: 1. Список (List) — представление пронумерованной последовательности значений, где одно и то же значение может присутствовать сколько угодно раз. 2. Хеш-таблица — неупорядоченная структура данных. Вместо индексов мы работаем с «ключами» и «значениями», вычисляя адрес памяти по ключу. 3. Стек - также упорядочен, но ограничен в действиях: можно лишь добавлять и убирать значения из конца списка. 4. Очередь - структура, комплементарная стеку. Разница в том, что элементы очереди удаляются из начала, а не из конца, т.е. сначала старые элементы, потом новые. 5. Графы 6. Связные списки - каждый элемент хранит ссылку на следующий элемент 7. Деревья https://habrahabr.ru/post/310794/	 **Рис. 3.1.** Классификация структур данных
Дайте характеристику структуре данных типа «массив»?	Массив (в некоторых языках программирования также таблица, ряд, матрица) — тип или структура данных в виде набора компонентов (элементов массива), расположенных в памяти непосредственно друг за другом. При этом доступ к отдельным элементам массива осуществляется с помощью индексации, то есть через ссылку на массив с указанием номера (индекса) нужного элемента. За счёт этого, в отличие от списка, массив является структурой данных, пригодной для осуществления произвольного доступа к её ячейкам ***Размерность массива*** — это количество индексов, необходимое для однозначной адресации элемента в рамках массива. Количество используемых индексов массива может быть различным: массивы с одним индексом называют одномерными, с двумя — двумерными, и т. д. Одномерный массив («колонка», «столбец») — нестрого соответствует вектору в математике; двумерный — матрице. Чаще всего применяются массивы с одним или двумя индексами; реже — с тремя; ещё большее количество индексов — встречается крайне редко. «Динамическим» называется массив такого размера, который может «динамически» меняться при выполнении программы. Динамические массивы делают работу с данными более гибкой, так как не требуют предварительного определения хранимых объёмов данных, а позволяют регулировать размер массива в соответствии с реальными потребностями. std::string students[10] = { "Иванов", "Петров", "Сидоров", "Ахмедов", "Ерошкин", "Выхин", "Андеев", "Вин Дизель", "Картошкин", "Чубайс" Сложность простейших операций: - вставка/удаление элемента - O(n) - поиск элемента - O(n) - индексация или доступ к элементу - O(1)	
Дайте характеристику структуре данных типа «список»?	**Список** — упорядоченный набор элементов, для каждого из которых хранится указатель на следующий (или для двусвязного списка и на следующий и на предыдущий) элементы списка. В односвязном списке каждый элемент состоит из двух различных по назначению полей: содержательного поля и поля указателя. В содержательном поле хранятся данные, ради которых создается список. В общем случае содержательное поле – это запись. Поле указателя хранит адрес следующего элемента списка. Пользуясь указателем, можно получить доступ к следующему элементу списка, а из следующего элемента – к очередному и так далее, пока не будет достигнут последний элемент. Поле указателя последнего элемента должно содержать специальный признак нулевого или пустого указателя – признак конца списка. list('список') ['с', 'п', 'и', 'с', 'о', 'к'] Сложность простейших операций: - вставка/удаление элемента - O(1) - поиск элемента - O(n) - индексация или доступ к элементу - O(n)	

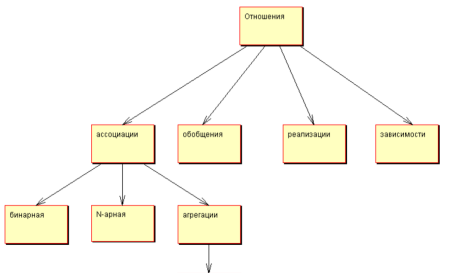
Дайте характеристику структуре данных типа «хэш-таблица»?	Хэш-таблица – это структура данных, реализующая интерфейс ассоциативного массива, то есть она позволяет хранить пары вида "ключ- значение" и выполнять три операции: o операцию добавления новой пары; o операцию поиска; o операцию удаления пары по ключу. Хэш-таблица является массивом, формируемым в определенном порядке хэш-функцией. С точки зрения практического применения, хорошей является такая хэш-функция, которая удовлетворяет следующим условиям: • функция должна быть простой с вычислительной точки зрения; • функция должна распределять ключи в хэш-таблице наиболее равномерно; • функция не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов; • функция должна минимизировать число коллизий,то есть ситуаций, когда разным ключам соответствует одно значение хэш-функции (ключи в этом случае называются синонимами). При возникновении коллизий (разным ключам соответствует одно значение хэш-функции) необходимо найти новое место для хранения ключей, претендующих на одну и ту же ячейку хэш-таблицы. Причем, если коллизии допускаются, то их количество необходимо минимизировать. В некоторых специальных случаях удается избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую инъективную хэш-функцию, которая распределит их по ячейкам хэш-таблицы без коллизий. Хэш-таблицы, использующие подобные хэш-функции, не нуждаются в механизме разрешения коллизий, и называются хэш-таблицами с прямой адресацией. Хэш-таблицы должны соответствовать следующим свойствам: • Выполнение операции в хэш-таблице начинается с вычисления хэш-функции от ключа. Получающееся хэш-значение является индексом в исходном массиве. • Количество хранимых элементов массива, деленное на число возможных значений хэш-функции, называется коэффициентом заполнения хэш-таблицы и является важным параметром, от которого зависит среднее время выполнения операций. • Операции поиска, вставки и удаления должны выполняться в среднем за время O(1). Однако при такой оценке не учитываются возможные аппаратные затраты на перестройку индекса хэш-таблицы, связанную с увеличением значения размера массива и добавлением в хэш-таблицу новой пары. Механизм разрешения коллизий является важной составляющей любой хэш-таблицы.			

Дайте характеристику структуре данных типа «дерево»?	Деревом называется сетевая структура, характеризующаяся следующими свойствами:
	1.Существует единственный элемент, на который не ссылается никакой другой элемент. Он называется корнем.
	2.Начиная с корня и следуя по определенной цепочке указателей, содержащихся в элементах, можно осуществить доступ к любому элементу структуры.
	3.На каждый элемент, кроме корня, имеется единственная ссылка, то есть каждыйэлемент адресуется единственным указателем.
	Определенная таким образом структура называется также корневым деревом.
	Каждая вершина графа ассоциируется с элементом многосвязного списка и называется узлом ветвления, или промежуточным узлом. Узел, который не ссылается ни на какие другие узлы, называется листом. Дуги графа называются ветвями.
	У каждого промежуточного узла можно определить предка (родителя, отца) и потомка (сына). Предком данного узла считается тот узел, который ссылается на него. Потомком данного узла считается тот узел, на который он ссылается.
	В определении дерева не накладывается ограничений на степень исхода каждого узла, то есть на число сыновей у каждого родителя. Если это число не больше m для каждого узла, то соответствующее дерево называется m-арным. Если степень исхода равна m или 0 для каждого узла, то дерево называется полным m-арным деревом.
	В машинной памяти дерево представляется в виде многосвязного списка, в котором каждый указатель соответствует дуге. Это представление называется естественным представлением дерева.
	
	Tree {
	root: {
	value: 1,
	children: [{
	value: 2,
	children: [...]
	}, {
	value: 3,
	children: [...]
	}]
	}
	}			

Дайте определение понятию «система контроля версий»? Виды систем контроля версий и их особенности?	Система контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. Есть три вида систем контроля версий: 1. Локальные 2. Централизованные 3. Распределенные **Локальные системы контроля версий** Многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот файл, либо скопировать файлы не туда, куда хотел, и затереть нужные файлы. Пример утилиты для работы с локальнок СКВ основан на работе с наборами патчей между парами версий (патч — файл, описывающий различие между файлами), которые хранятся в специальном формате на диске. Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи. **Централизованные системы контроля версий** Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий **Распределённые системы контроля версий** В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных			

Система контроля версий «Git»? Понятие «слепка»? Диаграмма состояний репозитория? Объекты «Git»: рабочий каталог, область подготовленных файлов, каталог?	**Git — распределённая система управления версиями. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux** **Слепки вместо патчей** Главное отличие Git'a от любых других СКВ (например, Subversion и ей подобных) — это то, как Git смотрит на свои данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени. Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. **Диаграмма состояний** В Git'е файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит. Таким образом, в проектах, использующих Git, есть три части: каталог Git'a (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area). **Каталог Git'a** — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git'a, и именно она копируется, когда вы клонируете репозиторий с другого компьютера. **Рабочий каталог** — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git'a и помещаются на диск для того, чтобы вы их просматривали и редактировали. **Область подготовленных файлов** — это обычный файл, обычно хранящийся в каталоге Git'a, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).	Рабочий каталог, область подготовленных файлов, каталог Git'a.  		
Стандартный рабочий поток (Git workflow)? Основные команды: git init, git add, git commit, git push, git pull, git clone?	**Стандартный рабочий процесс** с использованием Git'a выглядит примерно так: Вы вносите изменения в файлы в своём рабочем каталоге. Подготавливаете файлы, добавляя их слепки в область подготовленных файлов. Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git'a на постоянное хранение. **git init - Создание репозитория в существующем каталоге** Эта команда создаёт в текущем каталоге новый подкаталог с именем .git содержащий все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. **git add - индексация файлов / Отслеживание новых файлов** Когда вы осуществляете команду git add file, вы говорите, что git надо отметить текущее состояние файла, коммит которого будет произведен позже. *git add . - проиндексировать все файлы проекта* **git clone - получить копию существующего репозитория Git** Клонирование репозитория осуществляется командой git clone [url]. **git commit - Фиксация изменений** Коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и торчит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние. **git push - отправка изменений на сервер** Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия: git push [удал. сервер] [ветка] **git pull - получений изменений с сервера (git push наоборот)**			

Объясните, что такое модель OSI? Стек протоколов TCP/IP? Кратко поясните процесс межпрограммного взаимодействия на основе данной модели?	Модель Взаимосвязи Открытых Систем (OSI), определявшую стандарты связи компьютеров от разных производителей. Модель OSI была подразделена на 7 уровней. 7ой — уровень: Приложение > Сервисы 6ой — уровень: Представление > Сервисы 5ый — уровень: Сессия > Связь 4ый — уровень: Транспортный > Связь 3ий — уровень: Сетевой > Связь 2ой — уровень: Данные > Физические соединения 1ый — уровень: Физический > Физические соединения Сетевая модель OSI разработана таким образом, чтобы вышестоящие уровни сетевой модели использовали нижестоящие уровни сетевой модели, для передачи своей информации. Правила, с помощью которых общаются уровни модели, называются сетевыми протоколами. Сетевой протокол определенного уровня модели может общаться либо с протоколами своего уровня, либо с протоколами соседних уровней. Здесь опять же можно провести аналогию с работой компании. В компании всегда есть четко установленная иерархия, хотя и не такая строгая как в сетевой модели. Работники одной ступени иерархии выполняют поручения, полученные от работников более высокого уровня иерархии. **Прикладной уровень** Прикладной уровень является точкой, через которую приложения общаются с сетью (точка входа в модель OSI). С помощью данного уровня модели OSI выполняется следующие задачи: управление сетью, управление занятостью системы, управление передачей файлов, идентификация пользователей по их паролям. Примерами протоколов данного уровня являются: HTTP, SMTP, RDP и д.р. Очень часто протоколы прикладного уровня выполняют одновременно функции протоколов представительского и сеансового уровней. **Представительский уровень** Данный уровень отвечает за формат представления данных. Грубо говоря, он преобразует данные полученные от уровня приложений к формату пригодному для передачи по сети (ну и соответственно выполняет обратную операцию преобразуя информацию, полученную из сети, к формату пригодному для обработки приложениями). **Сеансовый уровень** На данном уровне происходит установление, поддержание и управление сеансом связи между двумя системами. Именно данный уровень отвечает за поддержание связи между системами на весь промежуток времени в течение которого происходит их взаимодействие. **Транспортный уровень** Протоколы данного уровня сетевой модели OSI отвечают за передачу данных от одной системы другой. На данном уровне большие блоки данных разделяются на более мелкие блоки, пригодные для обработки сетевым уровнем (очень мелкие блоки данных объединяются в более крупные), данные блоки соответствующим образом маркируются для их последующего восстановления на принимающей стороне. Так же при использовании соответствующих протоколов данный уровень способен обеспечить контроль доставки пакетов сетевого уровня. Блок данных, которым оперируют данный уровень обычно называется сегментом. Примерами протоколов данного уровня являются: TCP, UDP, SPX, ATP и д.р. **Сетевой уровень** Данный уровень отвечает за маршрутизацию (определение оптимальных маршрутов от одной системы до другой) блоков данных данного уровня. Блок данных этого уровня обычно называется пакетом. Так же данный уровень отвечает за логическую адресацию систем (те самые IP адреса), на основе которой как раз и происходит маршрутизация. К протоколам данного уровня можно отнести: IP, IPX и др, к устройствам работающим на данном уровне – маршрутизаторы. **Канальный уровень** Данный уровень отвечает за физическую адресацию устройств сети (MAC адреса), управлением доступа к среде, а также коррекцией ошибок допущенных физическим уровнем. Блок данных, используемый на канальном уровне принято называть фреймом. К данному уровню относятся следующие устройства: коммутаторы (не все), мосты и д.р. Типичной технологией использующей данный уровень является Ethernet. **Физический уровень** Осуществляет передачу оптических или электрических импульсов по выбранной среде передачи. К устройствам данного уровня можно отнести всевозможные повторители и концентраторы.			
Понятие гипертекста? Что такое URL и их каких частей он состоит?	В компьютерной терминологии **гипертекст** — это текст, сформированный с помощью языка разметки (например, HTML) с расчетом на использование гиперссылок. Структура URL: В самом распространенном случае использования в интернете URL имеет следующую структуру: {протокол} :// {доменное имя сайта?} / {URL-путь} ? {параметры} # якорь Где «якорь» — это указатель для перемещения в пределах текущей страницы к заранее установленному флагу.				

Протокол HTTP? Структура заголовков запроса и ответа по протоколу HTTP? Особенности методов GET и POST протокола HTTP?	Общение между хостом и клиентом происходит в два этапа: запрос и ответ. Клиент формирует HTTP запрос, в ответ на который сервер даёт ответ (сообщение). Структура запроса: 1. **Строка запроса** – указывает метод передачи, URL-адрес, к которому нужно обратиться и версию протокола HTTP. 2. **Заголовки** – описывают тело сообщений, передают различные параметры и др. сведения и информацию. 3. **Тело сообщения** — это сами данные, которые передаются в запросе. Тело сообщения – это необязательный параметр и может отсутствовать. Запрос от браузера: GET / HTTP/1.1 Host: webgyry.info User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:18.0) Gecko/20100101 Firefox/18.0 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3 Accept-Encoding: gzip, deflate Cookie: wp-settings Connection: keep-alive Ответ сервера: HTTP/1.1 200 OK Date: Sun, 10 Feb 2013 03:51:41 GMT Content-Type: text/html; charset=UTF-8 Transfer-Encoding: chunked Connection: keep-alive Keep-Alive: timeout=5 Server: Apache X-Pingback: //webgyry.info/xmlrpc.php Существующие методы: GET: получить доступ к существующему ресурсу. В URL перечислена вся необходимая информация, чтобы сервер смог найти и вернуть в качестве ответа искомый ресурс. POST: используется для создания нового ресурса. POST запрос обычно содержит в себе всю нужную информацию для создания нового ресурса.			
Язык UML? Определение? Области применения? Прямое и обратное проектирование?	Язык UML представляет собой общецелевой язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес-процессов и других систем. Прямое и обратное проектирование: При прямом проектировании класса или кооперации вы на самом деле проектируете компонент, который представляет исходный код, двоичную библиотеку или исполняемую программу для этого класса или кооперации. Точно так же при обратном проектировании исходного кода, двоичной библиотеки или исполняемой программы вы в действительности проектируете компонент или множество компонентов, которые отображаются на классы или кооперации. Решая заняться прямым проектированием (созданием кода по модели) класса или кооперации, вы должны определиться, во что Их нужно преобразовать: в исходный код, двоичную библиотеку или исполняемую программу. Логическую модель имеет смысл превратить в исходный код, если вас интересует управление конфигурацией файлов, которые затем будут поданы на вход среды разработки. Логические, модели следует непосредственно преобразовать в двоичные библиотеки или исполняемые программы, если вас интересует управление компонентами, которые фактически будут развернуты в составе работающей системы. Иногда нужно и то, и другое. Классу или кооперации можно поставить в соответствие как исходный код, так и двоичную библиотеку или исполняемую программу. Прямое проектирование диаграммы компонентов состоит из следующих этапов: 1. Для каждого компонента идентифицируйте реализуемые им классы или кооперации. 2. Выберите для каждого компонента целевое представление. Это может быть либо исходный код (или иная форма, которой может манипулировать система разработки), либо двоичная библиотека, либо исполняемая программа (или иная форма, которая может быть включена в работающую систему). 3. Используйте инструментальные средства для прямого проектирования модели. Обратное проектирование (создание модели по коду) диаграммы компонентов - не идеальный процесс, поскольку всегда имеет место потеря информации. По исходному коду вы можете обратно спроектировать классы - это более или менее обычное дело. Обратное проектирование компонентов из исходного кода выявляет существующие между файлами зависимости компиляции. Обратное проектирование диаграммы компонентов осуществляется так: 1. Выберите целевое представление. Исходный код можно реконструировать в компоненты, а затем и в классы. Двоичные библиотеки можно подвергнуть обратному проектированию для раскрытия их интерфейсов. Исполняемые программы поддаются обратному проектированию в наименьшей степени. 2. С помощью инструментальных средств укажите на код, который вы хотите подвергнуть обратному проектированию. Воспользуйтесь инструментами для генерации новой модели или модификации существующей, для которой ранее было проведено прямое проектирование. 3. Воспользуйтесь инструментальными средствами для создания диаграммы компонентов путем сверки с моделью. Например, можно начать с одного или нескольких компонентов, а затем расширять диаграмму, следуя по связям или переходя к соседним компонентам. Раскройте или спрячьте детали этой диаграммы компонентов в соответствии с тем, что именно вы хотите донести до читателя.			
Основные строительный блоки языка UML: сущности, отношения, диаграммы?	**Диаграмма** в UML– это графическое представление набора элементов, изображаемое чаще всего в виде связанного графа с вершинами (сущностями) и ребрами (отношениями). Диаграммы рисуют для визуализации системы с разных точек зрения. **Диаграмма**– в некотором смысле одна из проекций системы. Как правило, за исключением наиболее тривиальных случаев, диаграммы дают свернутое представление элементов, из которых составлена система. Один и тот же элемент может присутствовать во всех диаграммах, или только в нескольких (самый распространенный вариант), или не присутствовать ни в одной (очень редко). Теоретически диаграммы могут содержать любые комбинации сущностей и отношений. На практике, однако, применяется сравнительно небольшое количество типовых комбинаций, соответствующих пяти наиболее употребительным видам, которые составляют архитектуру программной системы. *Отношения связывают различные сущности;* **Сущности**– это абстракции, являющиеся основными элементами модели. В UML имеется четыре типа сущностей: – структурные; – поведенческие; – группирующие; – аннотационные. Сущности являются основными объектно-ориентированными блоками языка. С их помощью можно создавать корректные модели.			

Назовите виды и особенности различных типов UML-диаграмм: логические и физические, структурные и поведенческие?	Чтобы пояснить отличие ***логического и физического*** представлений, рассмотрим в общих чертах процесс разработки некоторой программной системы. Ее исходным логическим представлением могут служить структурные схемы алгоритмов и процедур, описания интерфейсов и концептуальные схемы баз данных. Однако для реализации этой системы необходимо разработать исходный текст программы на некотором языке программирования (C++, Pascal, Basic/VBA, Java). При этом уже в тексте программы предполагается такая организация программного кода, которая предполагает его разбиение на отдельные модули. Тем не менее исходные тексты программы еще не являются окончательной реализацией проекта, хотя и служат фрагментом его физического представления. Очевидно, программная система может считаться реализованной в том случае, когда она будет способна выполнять функции своего целевого предназначения. А это возможно, только если программный код системы будет реализован в форме исполняемых модулей, библиотек классов и процедур, стандартных графических интерфейсов, файлах баз данных. Именно эти компоненты являются необходимыми элементами физического представления системы. Таким образом, полный проект программной системы представляет собой совокупность моделей логического и физического представлений, которые должны быть согласованы между собой. В языке UML для физического представления моделей систем используются так называемые диаграммы реализации (implementation diagrams), которые включают в себя две отдельные канонические диаграммы: диаграмму компонентов и диаграмму развертывания. ***Структурные диаграммы*** В UML существует четыре структурных диаграммы для визуализации, специфицирования, конструирования и документирования статических аспектов системы, составляющих ее относительно прочный "костяк". Статические аспекты программных систем отражают наличие и расположение классов интерфейсов, кооперацый, компонентов, узлов и других сущностей. Названия структурных диаграмм UML соответствуют названиям основных групп сущностей, используемых при моделировании системы: • диаграммы классов - классам, интерфейсам и кооперациям; • диаграммы объектов - объектам; • диаграммы компонентов - компонентам; • диаграммы развертывания - узлам. На диаграмме классов изображают множество классов, интерфейсов, кооперацый и их отношений. Это самый распространенный тип диаграмм, применяемый при моделировании объектно-ориентированных систем; он используется для иллюстрации статического вида системы с точки зрения проектирования. Диаграммы, на которых показаны активные классы, применяются для работы со статическим видом системы с точки зрения процессов. На диаграмме объектов показывают множество объектов и отношения между ними. Такие изображения используются для иллюстрации структуры данных, то есть статических "мгновенных снимков" экземпляров тех сущностей, которые представлены на диаграмме классов. Диаграммы объектов, так же как и диаграммы классов, относятся к статическому виду системы с точки зрения процессов, но заостряют внимание на реальных или модельных прецедентах. На диаграммах компонентов показаны множества компонентов и отношения между ними. С их помощью иллюстрируют статический вид системы с точки зрения реализации. Диаграммы компонентов соотносятся с диаграммами классов, так как обычно компонент отображается на один или несколько классов, интерфейсов или кооперацый. На диаграммах развертывания представлены узлы и отношения между ними. С помощью таких изображений иллюстрируют статический вид системы с точки зрения развертывания. Они соотносятся с диаграммами компонентов, так как узел обычно содержит один или несколько компонентов. ***Диаграммы поведения*** Пять основных диаграмм поведения в UML используются для визуализации, специфицирования, конструирования и документирования динамических аспектов системы. Можно считать, что динамические аспекты системы представляют собой ее изменяющиеся части. Динамические аспекты программной системы охватывают такие ее элементы, как поток сообщений во времени и физическое перемещение компонентов по сети. Диаграммы поведения в UML условно разделяются на пять типов в соответствии с основными способами моделирования динамики системы: • диаграммы прецедентов описывают организацию поведения системы; • диаграммы последовательностей акцентируют внимание на временной упорядоченности сообщений; • диаграммы кооперации сфокусированы на структурной организации объектов, посылающих и получающих сообщения; • диаграммы состояний описывают изменение состояния системы в ответ на события; • диаграммы деятельности демонстрируют передачу управления от одной деятельности к другой.

Диаграмма «Классов» («Class Diagram»)? Основные типы сущностей и отношений? Сценарии применения?	Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними. Класс – это основной строительный блок. Каждый класс имеет название, атрибуты и операции. Класс на диаграмме показывается в виде прямоугольника, разделенного на 3 области. В верхней содержится название класса, в средней – описание атрибутов (свойств), в нижней – названия операций – услуг, предоставляемых объектами этого класса. Рис.1. Изображение класса в нотации UML **КАРТИНКИ ЗДЕСЬ: http://www.informicus.ru/default.aspx?SECTION=6&id=73&subdivisionid=3** Атрибуты класса определяют состав и структуру данных, хранимых в объектах этого класса. Каждый атрибут имеет имя и тип, определяющий, какие данные он представляет. При реализации объекта в программном коде для атрибутов будет выделена память, необходимая для хранения всех атрибутов, и каждый атрибут будет иметь конкретное значение в любой момент времени работы программы. Объектов одного класса в программе может быть сколь угодно много, все они имеют одинаковый набор атрибутов, описанный в классе, но значения атрибутов у каждого объекта свои и могут изменяться в ходе выполнения программы. Для каждого атрибута класса можно задать видимость (visibility). Эта характеристика показывает, доступен ли атрибут для других классов. В UML определены следующие уровни видимости атрибутов: • Открытый (public) – атрибут виден для любого другого класса (объекта); • Защищенный (protected) – атрибут виден для потомков данного класса; • Закрытый (private) – атрибут не виден внешними классами (объектами) и может использоваться только объектом, его содержащим. Последнее значение позволяет реализовать свойство инкапсуляции данных. Класс содержит объявления операций, представляющих собой определения запросов, которые должны выполнять объекты данного класса. Каждая операция имеет сигнатуру, содержащую имя операции, тип возвращаемого значения и список параметров, который может быть пустым. Реализация операции в виде процедуры – это метод, принадлежащий классу. Для операций, как и для атрибутов класса, определено понятие «видимость». Закрытые операции являются внутренними для объектов класса и недоступны из других объектов. Остальные образуют интерфейсную часть класса и являются средством интеграции класса в ПС. Отношения На диаграммах классов обычно показываются ассоциации и обобщения. Каждая ассоциация несет информацию о связях между объектами. Наиболее часто используются бинарные ассоциации, связывающие два класса. Ассоциация может иметь название, которое должно выражать суть отображаемой связи. Помимо названия, ассоциация может иметь такую характеристику, как множественность. Она показывает, сколько объектов каждого класса может участвовать в ассоциации. Множественность указывается у каждого конца ассоциации (полюса) и задается конкретным числом или диапазоном чисел. Множественность, указанная в виде звездочки, предполагает любое количество (в том числе, и ноль). Например, на рис.2 ассоциация связывает один объект класса «Набор товаров» с одним или более объектами класса «товар». Связаны между собой могут быть и объекты одного класса, поэтому ассоциация может связывать класс с самим собой. Например, для класса «Житель города» можно ввести ассоциацию «Соседство», которая позволит находить всех соседей конкретного жителя. Рис.2 Применение ассоциаций Ассоциация «включает» показывает, что набор может включать несколько различных товаров. В данном случае направленная ассоциация позволяет найти все виды товаров, входящие в набор, но не дает ответа на вопрос, входит ли товар данного вида в какой-либо набор. Ассоциация сама может обладать свойствами класса, то есть, иметь атрибуты и операции. В этом случае она называется класс-ассоциацией и может рассматриваться как класс, у которого помимо явно указанных атрибутов и операций есть ссылки на оба связываемых ею класса. В примере на рис.2 ассоциация «включает» по существу есть класс-ассоциация, у которой есть атрибут «Количество», показывающий, сколько единиц каждого товара входит в набор (см. рис.4). Обобщение на диаграммах классов используется, чтобы показать связь между классом-родителем и классом-потомком. Оно вводится на диаграмму, когда возникает разновидность какого-либо класса (например, при развитии ПС – см. рис.4), а также в тех случаях, когда в системе обнаруживаются несколько классов, обладающих сходным поведением (в этом случае общие элементы поведения выносятся на более высокий уровень, образуя класс-родитель – см. рис.3). Рис.3 Наследуются атрибуты и операции Как уже говорилось ранее, UML позволяет строить модели с различным уровнем детализации. На рис.4 показана детализация модели, представленной на рис.2. Рис. 4 Детализация модели набора товаров Обобщение показывает, что набор товаров – это тоже товар, который может быть предметом заказа, продажи, поставки и т.д. Набор включает опись, в которой указывается, какие товары входят в набор, а класс-ассоциация «включает» определяет количество каждого вида товаров в наборе. Стереотипы классов При создании диаграмм классов часто пользуются понятием «стереотип». В дальнейшем речь пойдет о стереотипах классов. Стереотип класса – это элемент расширения словаря UML, который обозначает отличительные особенности в использовании класса. Стереотип имеет название, которое задается в виде текстовой строки. При изображении класса на диаграмме стереотип показывается в верхней части класса в двойных угловых скобках. Есть четыре стандартных стереотипа классов, для которых предусмотрены специальные графические изображения (см. рис.5). Стереотип используется для обозначения классов-сущностей (классов данных), стреотип описывает пограничные классы, которые являются посредниками между ПС и внешними по отношению к ней сущностями – актерами, обозначаемыми стереотипом <>. Наконец, стереотип описывает классы и объекты, которые управляют взаимодействиями. Применение стереотипов позволяет, в частности, изменить вид диаграмм классов. Рис.5 Стереотипы для классов Применение диаграмм классов Диаграммы классов создаются при логическом моделировании ПС и служат для следующих целей: • Для моделирования данных. Анализ предметной области позволяет выявить основные характерные для нее сущности и связи между ними. Это удобно моделируется с помощью диаграмм классов. Эти диаграммы являются основой для построения концептуальной схемы базы данных. • Для представления архитектуры ПС. Можно выделить архитектурно значимые классы и показать их на диаграммах, описывающих архитектуру ПС. • Для моделирования навигации экранов. На таких диаграммах показываются пограничные классы и их логическая взаимосвязь. Информационные поля моделируются как атрибуты классов, а управляющие кнопки – как операции и отношения. • Для моделирования логики программных компонент (будет описано в последующих статьях). • Для моделирования логики обработки данных.			
Диаграмма «Компонентов» («Component Diagram»)? Основные типы сущностей и отношений? Сценарии применения?	Диаграммы компонентов показывают, как выглядит модель системы на физическом уровне. На диаграмме изображены компоненты программного обеспечения и связи между ними. При этом выделяют два типа компонентов: исполняемые компоненты и библиотеки кода. Каждый класс модели преобразуется в компонент исходного кода. После создания они сразу добавляются к диаграмме компонентов. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы. Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию системы. Из нее видно, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут созданы системой. На такой диаграмме показано соответствие классов реализованным компонентам. Она нужна там, где начинается генерация кода.			

Диаграмма «Развёртывания» («Deployment Diagram»)? Основные типы сущностей и отношений? Сценарии применения?	Когда мы пишем программу, мы пишем ее для того, чтобы запускать на компьютере, который имеет некоторую аппаратную конфигурацию и работает под управлением некоторой операционной системы. Корпоративные приложения часто требуют для своей работы некоторой ИТ-инфраструктуры, хранят информацию в базах данных, расположенных где-то на серверах компании, вызывают веб-сервисы, используют общие ресурсы и т. д. В таких случаях неплохо бы иметь графическое представление инфраструктуры, на которую будет развернуто приложение. Вот для этого-то и нужны диаграммы развертывания, которые иногда называют диаграммами размещения. Думаю, очевидно, что такие диаграммы есть смысл строить только для аппаратно-программных систем, тогда как UML позволяет строить модели любых систем, не обязательно компьютерных. Какую пользу можно извлечь из диаграмм развертывания? Во-первых, графическое представление ИТ-инфраструктуры может помочь более рационально распределить компоненты системы по узлам сети, от чего, как известно, зависит в том числе и производительность системы. Во-вторых, такая диаграмма может помочь решить множество вспомогательных задач, связанных, например, с обеспечением безопасности. Диаграмма развертывания показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения - маршруты передачи информации между аппаратными узлами. Это единственная диаграмма, на которой применяются "трехмерные" обозначения: узлы системы обозначаются кубиками. Все остальные обозначения в UML - плоские фигуры.			
Диаграмма «Прецедентов» («Use Case Diagram»)? Основные типы сущностей и отношений? Сценарии применения?	Диаграмма прецедентов (use case diagram) **КАРТИНКИ ЗДЕСЬ: http://studopedia.su/18_4323_ob-ektno-orientirovanniy-analiz-sistem-osnovi-UML.html Заодно может кому и др.варианты др.диаграмм понравятся.** Любые (в том числе и программные) системы проектируются с учетом того, что в процессе свой работы они будут использоваться людьми и/или взаимодействовать с другими системами. Сущности, с которыми взаимодействует система в процессе своей работы, называются экторами, причем каждый эктор ожидает, что система будет вести себя строго определенным, предсказуемым образом. Эктор (actor) - это множество логически связанных ролей, исполняемых при взаимодействии с прецедентами или сущностями (система, подсистема или класс). Эктором может быть человек или другая система, подсистема или класс, которые представляют нечто вне сущности. Графически эктор изображается либо "человечком", либо символом класса с соответствующим стереотипом. Обе формы представления имеют один и тот же смысл и могут использоваться в диаграммах. "Стереотипированная" форма чаще применяется для представления системных экторов или в случаях, когда эктор имеет свойства и их нужно отобразить. Дословный же перевод слова "эктор" - действующее лицо. Прецедент (use-case) - описание отдельного аспекта поведения системы с точки зрения пользователя. Прецедент (use case) - описание множества последовательных событий (включая варианты), выполняемых системой, которые приводят к наблюдаемому эктором результату. Прецедент представляет поведение сущности, описывая взаимодействие между экторами и системой. Прецедент не показывает, "как" достигается некоторый результат, а только "что" именно выполняется. Прецеденты обозначаются очень простым образом - в виде эллипса, внутри которого указано его название. Прецеденты и экторы соединяются с помощью линий. Часто на одном из концов линии изображают стрелку, причем направлена она к тому, у кого запрашивают сервис, другими словами, чьими услугами пользуются. Это простое объяснение иллюстрирует понимание прецедентов как сервисов, пропагандируемое компанией IBM. Прецеденты могут включать другие прецеденты, расширяться ими, наследоваться и т. д. Диаграммы прецедентов относятся к той группе диаграмм, которые представляют динамические или поведенческие аспекты системы. Это отличное средство для достижения взаимопонимания между разработчиками, экспертами и конечными пользователями продукта. Такие диаграммы очень просты для понимания и могут восприниматься и, обсуждаться людьми, не являющимися специалистами в области разработки ПО. Можно выделить такие цели создания диаграмм прецедентов: определение границы и контекста моделируемой предметной области на ранних этапах проектирования; формирование общих требований к поведению проектируемой системы; разработка концептуальной модели системы для ее последующей детализации; подготовка документации для взаимодействия с заказчиками и пользователями системы.			
Диаграмма «Последовательностей» («Sequence Diagram»)? Основные типы сущностей и отношений? Сценарии применения?	Для моделирования взаимодействия объектов в языке UML используются соответствующие диаграммы взаимодействия. Взаимодействия объектов можно рассматривать во времени, и тогда для представления временных особенностей передачи и приема сообщений между объектами используется диаграмма последовательности. Взаимодействующие объекты обмениваются между собой некоторой информацией. При этом информация принимает форму законченных сообщений. Другими словами, хотя сообщение и имеет информационное содержание, оно приобретает дополнительное свойство оказывать направленное влияние на своего получателя. ***Диаграммы последовательностей*** обычно содержат объекты, которые взаимодействуют в рамках сценария, сообщения, которыми они обмениваются, и возвращаемые результаты, связанные с сообщениями. Впрочем, часто возвращаемые результаты обозначают лишь в том случае, если это не очевидно из контекста. ***Диаграмма последовательности*** отражает поток событий, происходящих в рамках варианта использования. Все действующие лица показаны в верхней части диаграммы. Стрелки соответствуют сообщениям, передаваемым между действующим лицом и объектом или между объектами для выполнения требуемых функций. На диаграмме последовательности объект изображается в виде прямоугольника, от которого вниз проведена пунктирная вертикальная линия. Эта линия называется линией жизни (lifeline) объекта. Она представляет собой фрагмент жизненного цикла объекта в процессе взаимодействия. Каждое сообщение представляется в виде стрелки между линиями жизни двух объектов. Сообщения появляются в том порядке, как они показаны на странице сверху вниз. Каждое сообщение помечается как минимум именем сообщения. При желании можно добавить также аргументы и некоторую управляющую информацию. Можно показать самоделегирование (self-delegation) – сообщение, которое объект посылает самому себе, при этом стрелка сообщения указывает на ту же самую линию жизни. **https://habrastorage.org/getpro/habr/post_images/f61/f90/9d2/f61f909d28069d1e63703ac51b64c128.gif -> КАРТИНКА ДЛЯ ПРИМЕРА**			
Диаграмма «Состояний» («State Diagram»)? Основные типы сущностей и отношений? Сценарии применения?	Диаграммы состояний – хорошо известное средство описания поведения систем. Они определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий. В большинстве объектно-ориентированных методов диаграммы состояний строятся для единственного класса и отражают динамику поведения единственного объекта. Если метка перехода не содержит никакого события, это означает, что переход происходит, как только завершается любая деятельность, связанная с данным состоянием. Метка каждого из них включает только условие. Условие – это логическое условие, которое может принимать два значения: «истина» или «ложь». Условный переход выполняется только в том случае, если условие принимает значение «истина». Из конкретного состояния в данный момент времени может быть осуществлен только один переход. Таким образом, условия являются взаимно исключающими для любого события.			

Диаграмма «Активности» («Activity Diagram»)? Основные типы сущностей и отношений? Сценарии применения?	В отличие от большинства других средств UML диаграммы деятельностей(АКТИВНОСТИ) основаны на нескольких различных методах, в частности методе моделирования состояний SDL и сетях Петри. Эти диаграммы особенно полезны в описании поведения, включающего большое количество параллельных процессов. На концептуальной диаграмме деятельность – это некоторая задача, которую необходимо выполнить вручную или автоматизированным способом. На диаграмме, построенной в аспекте спецификации или реализации, деятельность представляет собой некоторый метод над классом. За каждой деятельностью может следовать другая деятельность. Такое следование образует простую последовательность. Например, за деятельностью Положить Кофе в Фильтр следует деятельность Вставить Фильтр в Автомат. Деятельность Поискать Напиток активизирует на выходе два действия. Каждое действие содержит условие – логическое выражение, которое может принимать одно из двух значений: "истина" или "ложь", так же как и на диаграмме состояний. В ситуации Личность осуществляет деятельность Поискать Напиток, выбирая между кофе и колой. Предположим, что мы отыскали кофе и идем вниз по «кофейному маршруту». Этот путь ведет к линейке синхронизации, с которой связана активизация трех деятельностей: Положить Кофе в Фильтр, Добавить Воду в Емкость и Достать Чашки. Диаграмма указывает на то, что эти три деятельности могут выполняться параллельно. По существу, это означает, что порядок их выполнения не играет роли. Можно сначала положить кофе в фильтр, затем добавить воды в емкость, потом достать чашки, а можно сначала достать чашки, а затем добавить кофе в фильтр. Можно также выполнять эти деятельности, чередуя их друг с другом. Можно было бы достать чашку, добавить немного воды в емкость, достать другую чашку, добавить еще немного воды и т. д. Можно также делать некоторые вещи одновременно: одной рукой наливать воду, а другой доставать чашку. В соответствии с диаграммой любой их этих вариантов является допустимым. Диаграмма деятельностей предоставляет свободу выбора порядка выполнения действий. Другими словами, она только устанавливает основные правила последовательности, которым необходимо следовать. Такая возможность важна при моделировании бизнес-процессов. Среди бизнес-процессов нередко встречаются такие, которые не обязаны выполняться последовательно. В таких ситуациях данный метод хорошо работает, так как он позволяет реализовывать процессы параллельно. Диаграммы деятельностей являются также полезными при параллельном программировании, поскольку можно графически изобразить все ветви и определить, когда их необходимо синхронизировать. Если при описании поведения системы имеются параллельные деятельности, то их необходимо синхронизировать. Так, кофейный автомат не будет включаться до тех пор, пока в него не вставлен фильтр и не добавлена вода в емкость. Именно поэтому на диаграмме результаты этих деятельностей сведены вместе к одной линейке синхронизации. Простая линейка синхронизации, подобная данной, показывает, что ее выходная деятельность активизируется только тогда, когда выполнены обе входные деятельности. Как можно увидеть в дальнейшем, эти линейки могут быть более сложными. Далее выполняется еще одна синхронизация: кофе должен быть готов и чашки должны стоять на месте перед тем, как мы сможем налить кофе. Теперь переместимся к другому маршруту. В данном случае мы имеем дело с составным решением. Первое решение принимается относительно кофе, оно определяется двумя выходами из деятельности Поискать Напиток. Если кофе нет, мы приходим ко второму решению, связанному с колой. При такой последовательности решений второе решение обозначается ромбом. Такое обозначение позволяет описывать вложенные решения, причем их количество может быть любым. Деятельность Выпить Напиток имеет два входа, что означает ее выполнение в любом случае. Данную ситуацию можно рассматривать как условие «ИЛИ» (выполняется, если выполняется хотя бы одна из двух деятельностей), а линейку синхронизации можно рассматривать как условие «И» (выполняется, если выполняются обе деятельности).			
Паттерны проектирования. Что это? История появления.	**Шаблоны проектирования** — это проверенные и готовые к использованию решения часто возникающих в повседневном программировании задач. Это не класс и не библиотека, которую можно подключить к проекту, это нечто большее. Шаблон проектирования, подходящий под задачу, реализуется в каждом конкретном случае. Кроме того, он не зависит от языка программирования. Хороший шаблон легко реализуется в большинстве, если не во всех языках, в зависимости от выразительных средств языка. Следует, однако, помнить, что такой шаблон, будучи примененным неправильно или к неподходящей задаче, может принести немало проблем. Тем не менее, правильно примененный шаблон поможет решить задачу легко и просто. Существует три типа шаблонов: *структурные;* *порождающие;* *поведенческие.* **Структурные шаблоны** определяют отношения между классами и объектами, позволяя им работать совместно. **Порождающие шаблоны** предоставляют механизмы инициализации, позволяя создавать объекты удобным способом. **Поведенческие шаблоны** используются для того, чтобы упростить взаимодействие между сущностями. Шаблон проектирования, по своей сути, это продуманное решение той или иной задачи. Если вы столкнулись с известной задачей, почему бы не использовать готовое решение, проверенное опытом?			
Паттерны GRASP. Общая характеристика.	https://habrahabr.ru/post/92570/ если кратенько: GRASP - General Responsibility Assignment Software Patterns (основные шаблоны распределения обязанностей в программном обеспечении). Это методический подход к объектному проектированию. Эти шаблоны называют также шаблонами распределения обязанностей. Под "шаблоном" в данном контексте, да и в любом контексте, связанном с разработкой ПО, понимается именованная пара "проблема/решение", содержащая рекомендации для применения в различных конкретных ситуациях. Шаблоны GRASP относятся к этапу проектирования и отвечают за взаимосвязь объектов в системе. GRASP состоит из 5 основных и 4 дополнительных шаблонов.			
GRASP. Information Expert.	Шаблон решает проблему распределения обязанностей между объектами в объектно-ориентированной системе. Под обязанностью в контексте GRASP понимается некое действие (функция) объекта. Т.о. Information Expert дает рекомендации касательно того, какие функции должен выполнять тот или иной объект. А решение очень простое и понятное без доказательств: назначать обязанность следует информационному эксперту - классу, у которого имеется информация, требуемая для выполнения обязанности. Конечно, не всегда бывает так, что вся необходимая информация заключена в одном классе (это, кстати, идет в противоречие с паттерном High Cohesion, о котором позже), чаще она распределена между различными классами, тогда каждый из этих классов будет являться частичным экспертом, т.е. будет предоставлять только доступную ему информацию, а при общем взаимодействии будет решена общая задача.			

GRASP. Controller.	Контроллер (Controller) - GRASP **(вообще годен для небольших систем)** Шаблон controller решает давнюю проблему разделения интерфейса и логики в интерактивном приложении. Это не что иное, как C из аббревиатуры MVC. Этот шаблон отвечает за то, к кому именно должны обращаться вызовы из V (View), и кому C должен делегировать запросы на выполнение (какая модель M должна их обработать). Если обобщить назначение controller, то он должен отвечать за обработку входных системных сообщений. Под системными сообщениями здесь понимаются события высокого уровня, генерируемые внешним исполнителем. Чтобы решить поставленную проблему необходимо делегировать обязанности обработки системных сообщений классу, удовлетворяющему одному из следующих условий: **Проблема** "Кто" должен отвечать за обработку входных систем**ных соб**ытий? **Решение** Обязанности по обработке системных сообщений делегируются специальному классу. Контроллер - это объект, который отвечает за обработку системных событий и не относится к интерфейсу пользователя. Контроллер определяет методы для выполнения **системных оп**ераций. **Рекомендации** Для различных прецедентов логично использовать разные контроллеры (контроллеры прецедентов) - контроллеры не должны быть перегружены. Внешний контроллер представляет всю систему целиком, его можно использовать, если он будет не слишком перегруженным (то есть, если существует лишь нес**колько систе**мных событий). **Преимущества** Удобно накапливать информацию о системных событиях (в случае, если системные операции выполняются в некоторой определенной последовательности). Улучшаются условия для повторного использования компонентов (системные события обрабатываются Контроллером а не эле**ментами ин**терфейса пользователя).Использование контроллеров позволяет отделить логику от представления, тем самым повышая возможность повторного использования кода. **Недостатки** Контроллер может оказаться перегружен.			
GRASP. Creator.	**Шаблон Creator решает проблему о том, кто должен создавать экземпляры новых классов. Решение состоит в назначении классу. В обязанностей создавать экземпляры класса A, если выполняется одно из условий:** **1) Класс B агрегирует (aggregate) объекты A** **2) Класс B содержит (contains) объекты A** **3) Класс B записывает (records) экземпляры объектов A** **4)Класс B активно использует (closely uses) объекты A** **5)Класс B обладает данными инициализации (has the initializing data), которые будут передаваться объектам A при их создании (т.е. при создании объектов A класс B является экспертом)** **Преимущество: поддерживается шаблон Low Coupling, снижаются затрат на сопровождение и обеспечивается возможности повторного использования созданных компонентов в дальнейшем.**			
GRASP. Low Coupling. GRASP. High Cohesion.	**Low Coupling** **Шаблон решает проблему связности. Связность можно определить как количество точек соприкосновения классов между собой. Известно, что чем ниже связность классов, тем меньше их взаимовлияние, тем выше возможность повторного использования. А рекомендация здесь простая: распределять обязанности между классами надо таким образом, чтобы уменьшить связность.** **Не существует абсолютной меры для определения слишком высокой степени связывания. Важно понимать степень связанности на текущий момент и не упустить тот момент, когда дальнейшее повышение связанности может привести к появлению проблем. В целом следует руководствоваться таким соображением: классы, которые являются достаточно общими по своей природе и с высокой степенью вероятности будут повторно использоваться в дальнейшем, должны иметь минимальную степень связанности с другими классами.** **Высокая степень связывания сама по себе не является проблемой. Проблемой является жесткое связывание с неустойчивыми в некотором отношении элементами. High Cohesion** **Шаблон решает общую проблему управления сложностью за счет регулирования степени зацепления классов. Разница между связностью и зацеплением очень тонка, и понять ее необходимо на более глубоком интуитивном уровне. Зацепление (cohesion) (или более точно, функциональное зацепление) - это мера связанности и сфокусированности обязанностей класса. Считается, что элемент обладает высокой степенью зацепления, если его обязанности тесно связаны между собой и он не выполняет огромных объемов работы. Класс с низкой степенью зацепления выполняет много разнородных функций или несвязанных между собой обязанностей.** **High Cohesion твердит, что класс должен стараться выполнять как можно меньше не специфичных для него задач, и иметь вполне определенную область применения. Только с опытом приходит понимание балансировки между High Cohesion и Low Coupling. считается что связывание и зацепление это инь и янь проектирования ПО. Некорректное связывание порождает неправильное зацепление и наоборот.**			
GRASP. Pure Fabrication.	**Формулировки и мотивации дополнительных шаблонов GRASP не так очевидны(А это как раз доп.шаблон). И без примеров здесь трудно ориентироваться. Проблемой, решаемой Pure Fabrication, является обеспечение реализации шаблонов High Cohesion и Low Coupling или других принципов проектирования, если шаблон Expert (например) не обеспечивает подходящего решения. Решением может быть введение служебного класса, не представляющего понятия конкретной предметной области, однако имеющего высокую степень зацепления.** **Например, классу X необходимо сохранять информацию в реляционной базе данных. Согласно шаблону Information Expert, эту обязанность можно присвоить самому классу X. Однако следует принять во внимание, что Данная задача требует достаточно большого числа специализированных операций, поэтому класс X будет иметь низкую степень зацепления, класс X будет связан с интерфейсом БД, что повысит связность, кроме того задача записи в БД является довольно общей, поэтому необходимо обеспечить повторное использование кода. Естественным решением данной проблемы является создание нового класса, ответственного за сохранение объектов некоторого вида на постоянном носителе, например в реляционной базе данных. Его можно назвать PersistentStorage. Это чисто синтетический объект, что полностью соответствует Pure Fabrication. В итоге решаются задачи зацепления, связности и повторного использования.(Применение шаблона позволяет дизайну системы соответствовать принципам низкой связности и высокого зацепления.)**			
GRASP. Protected Variations.	**Основная проблема, решаемая шаблоном Protected Variations: как спроектировать объекты, подсистемы и систему, чтобы изменение этих элементов не оказывало нежелательного влияния на другие элементы? необходимо идентифицировать точки возможных вариаций или неустойчивости; распределить обязанности таким образом, чтобы обеспечить устойчивый интерфейс. (сущность шаблона заключается в устранении точек неустойчивости, путем определения их в качестве интерфейсов и реализации для них различных вариантов поведения.)** **Короче "Найди то, что меняется и инкапсулировать."**			
GRASP. Polymorphism.	Принцип полиморфизма является основополагающим в ООП. В данном контексте шаблон Polymorphism решает проблему обработки альтернативных вариантов поведения на основе типа. Решать эту проблему стоит с использование полиморфных операций, а не с помощью проверки типа и условной логики. Кроме того, при помощи полиморфизма легко создавать подключаемые компоненты. В итоге получаем следующие преимущества: 1.Впоследствии можно легко расширять систему, добавляя новые вариации. 2.Новые вариации можно вводить без модификации клиентской части приложения.			
GRASP. Indirection.	Данный шаблон помогает решить проблему совмещения низкой связанности и высоким потенциалом повторного использования.Суть: Присвоить обязанности промежуточному объекту, который, в свою очередь сотрудничает с двумя объектами избегая непосредственной связи т.е. косвенности. Похож на Pure Fabrication (разница лишь в том, что indirection более сфокусирован на повышении потенциала на повторное использование)			

Паттерны GoF. Общая характеристика. Классификация.	Банда Четырех являются авторами книги "Приёмы объектно-ориентированного проектирования.Паттерны проектирования". Эта книга описывает различные методы разработки и подводные камни, которые могут возникнуть при их использовании Всего их 23 . Эти 23 делятся на: **1)Порождающие шаблоны проектирования** - обеспечивают способы создания экземпляров отдельных объектов или групп связанных объектов. Есть пять таких моделей Abstract Factory(Абстрактная фабрика), Builder(Строитель), Factory Method(Фабричный метод), Prototype (Прототип), Singleton (Одиночка) **2)Структурные шаблоны проектирования** обеспечивают способ определения отношений между классами или объектами. Есть 7 паттернов: Adapter(Адаптер), Bridge (Мост), Composite (Компоновщик), Decorator (Декоратор), Facade (Фасад), Flyweight (Приспособленец), Proxy (Заместитель) **3)Поведенческие шаблоны проектирования** определяют "манеры общения" между классами и объектами. Их 11: Chain of responsibility (Цепочка обязанностей), Command (Команда), Interpreter (Интерпретатор), Iterator(Итератор), Mediator (Посредник), Memento (Хранитель), Observer (Наблюдатель), State (Состояние), Strategy (Стратегия), Template method (Шаблонный метод), Visitor (Посетитель)			
Паттерны GoF. Singleton.	Одиночка (англ. Singleton) — порождающий шаблон проектирования как он есть, гарантирующий, что в однопроцессном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру. Все последующие ссылки на объекты класса одиночки относятся к одному и тому же основному экземпляру. Одиночка полезен, когда требуется единая глобальная точка доступа к ограниченному ресурсу. Это более целесообразно, чем создание глобальной переменной, так как она может быть скопирована, что приводит к нескольким точкам доступа и риск того, что дубли становятся в ногу с оригиналом.			
Паттерны GoF. Factory Method.	Данный паттерн довольно сложно объяснить в метафорах, но всё же попробую. Ключевой сложностью объяснения данного паттерна является то, что это «метод», поэтому метафора метода будет использовано как действие, то есть например слово «Хочу!». Соответственно, паттерн описывает то, как должно выполнятся это «Хочу!». Допустим ваша фабрика производит пакеты с разными соками. Теоретически мы можем на каждый вид сока делать свою производственную линию, но это не эффективно. Удобнее сделать одну линию по производству пакетов-основ, а разделение ввести только на этапе заливки сока, который мы можем определять просто по названию сока. Однако откуда взять название? Для этого мы создаем основной отдел по производству пакетов-основ и предупреждаем все под-отделы, что они должны производить нужный пакет с соком про простому «Хочу!» (т.е. каждый под-отдел должен реализовать паттерн «фабричный метод»). Поэтому каждый под-отдел заведует только своим типом сока и реагирует на слово «Хочу!». Таким образом если нам потребуется пакет апельсинового сока, то мы просто скажем отделу по производству апельсинового сока «Хочу!», а он в свою очередь скажет основному отделу по созданию пакетов сока, «Сделай ка свой обычный пакет и вот сок, который туда нужно залить».			
Паттерны GoF. Abstract Factory.	Абстрактная фабрика - шаблон дизайна, который позволяет создание групп связанных объектов без необходимости указания точных конкретных классов, которые будут использоваться. Один из немногих фабричных классов генерирует наборы объектов.Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс			
Паттерны GoF. Адаптер.	Данный паттерн полностью соответствует своему названию. Чтобы заставить работать «советскую» вилку через евро-розетку требуется переходник. Именно это и делает «адаптер», служит промежуточным объектом между двумя другими, которые не могут работать напрямую друг с другом.			
Паттерны GoF. Facade.	Паттерн «фасад» используется для того, чтобы делать сложные вещи простыми. Возьмем для примера автомобиль. Представьте, если бы управление автомобилем происходило немного по-другому: нажать одну кнопку чтобы подать питание с аккумулятора, другую чтобы подать питание на инжектор, третью чтобы включить генератор, четвертую чтобы зажечь ламочку на панели и так далее. Всё это было бы очень сложно. Для этого такие сложные наборы действий заменяются более простыми и комплексные как «повернуть ключ зажигания». В данном случае поворот ключа зажигания и будет тем самым «фасадом» для всего обилия внутренних действий автомобиля.			
Паттерны GoF. Decorator.	Как понятно из названия, данный паттерн чаще всего используется для расширения исходного объекта до требуемого вида. Например мы условно можем считать «декоратором» человека с кистью и красной краской. Таким образом, какой бы объект (или определенный тип объектов) мы не передали в руки «декоратору», на выходе мы будем получать красные объекты.			
Паттерны GoF. Visitor.	Данный паттерн можно сравнить с прохождением обследования в больнице. Однако «посетителем» в терминах паттернов здесь будут сами врачи. Чтобы было понятнее: у нас есть больной которого требуется обследовать и полечить, но так как за разные обследования отвечают разные врачи, то мы просто присылаем к больному врачей в качестве «посетителей». Правило взаимодействия для больного очень простое «пригласите врача (посетителя) чтобы он сделал свою работу», а врач («посетитель») приходит, обследует и делает всё необходимое. Таким образом следуя простым правилам можно использовать врачей для разных больных по одним и тем же алгоритмам. Как уже было сказано, паттерном «посетитель» в данном случае является врач, который может одинаково обслуживать разные объекты (больных) если его позовут.			
Паттерны GoF. Observer.	Поведенческий шаблон проектирования. Один объект уведомляет другие объекты об изменениях своего состояния. Объектам, связанным таким образом, не нужно знать друг о друге — это и есть слабо-связанный (гибкий) код. Этот паттерн чаще всего используют, когда надо уведомить «заинтересованных лиц» об изменении свойств объекта. Наблюдатель «регистрирует» свой интерес о состоянии другого объекта. Когда состояние меняется, все объекты-наблюдатели будут уведомлены об изменении. Пример реализации паттерна у Apple - уведомления в коде(Notifications)() и Key-Value Observing (KVO).			

Паттерны GoF. Chain of responsibility.	Самым простым примером цепочки обязанностей можно считать получение какого-либо официального документа. Например вам требуется получить справку со счета из банка. Так или иначе, вы должны эту справку получить, однако кто именно ее должен вам дать — пока не ясно. Вы приходите в местное отделение банка, вам говорят что «мы сейчас заняты, идите в другое отделение», дальше вы идете в другое, там вам отвечают «мы этим не занимаемся», вы идете в региональное отделение и там получаете нужную справку. Таким образом паттерн реализует «цепочку обязанностей» отдельные объекты которой (отделения банка) должны обработать ваш запрос. Соответственно ваш запрос может быть обработан в первом же отделении, или же в нескольких, в зависимости от самого запроса и обрабатывающих объектов.			
Паттерны GoF. Strategy.	Используется для выбора различных путей получения результата. Вспомним пример с получением прав. Человек, который будет реализовывать паттерн «стратегия» будет действовать следующим образом: вы говорите ему «Хочу права, денег мало» в ответ вы получите права через длительное время и с большой тратой ресурсов. Если вы скажите ему «Хочу права, денег много», то вы получите права очень быстро. Что именно делал этот человек вы понятия не имеете, но вы задаете начальные условия, а как себя вести уже решает он сам (сам выбирает стратегию). Соответственно внутри «стратегии» хранятся различные способы поведения, и чтобы выбрать, ему нужны определенные параметры, в данном случае это объем денежных средств. Как устроена сама «стратегия» и какие алгоритмы внутри нее вам собственно знать и требуется.			
Компилятор, интерпретатор, препроцессор, линковщик. Назначение.	**Компилятор** транслирует высокоуровневый язык в машинный. Когда пользователь пишет код на языке высокого уровня, таком как Java, и хочет его выполнить, то прежде чем это может быть сделано, будет использован специальный компилятор разработанный для Java. Компилятор сначала сканирует всю программу, а потом транслирует ее в машинный код, который будет выполнен компьютерным процессором, после чего будут выполнены соответствующие задачи. **Интерпретаторы** не очень сильно отличаются от компиляторов. Они также конвертируют высокоуровневые языки в читаемые машиной бинарные эквиваленты. Каждый раз когда интерпретатор получает на выполнение код языка высокого уровня, то прежде чем сконвертировать его в машинный код, он конвертирует этот код в промежуточный язык. Каждая часть кода интерпретируется и выполняется отдельно и последовательно, и если в какой-то части будет найдена ошибка, она остановит интерпретацию кода без трансляции следующей части кода. **Препроцессор** — это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, компилятора). О данных на выходе препроцессора говорят, что они находятся в препроцессированной форме, пригодной для обработки последующими программами (компилятор). Результат и вид обработки зависят от вида препроцессора; так, некоторые препроцессоры могут только выполнить простую текстовую подстановку, другие способны по возможностям сравниться с языками программирования. Наиболее частый случай использования препроцессора — обработка исходного кода перед передачей его на следующий шаг компиляции. **Линковщик** - это программа, которая с объектных файлов собирает бинарник. Линковка это процесс компоновки различных кусков кода и данных вместе, в результате чего получается один исполняемый файл. Линковка может быть выполнена во время компиляции, во время загрузки (загрузчиком) и также во время исполнения (исполняемой программой).			
Система контроля SVN.	SVN - нераспределенная (GIT - распределенная) система контроля версий. Как это понять: если система версий распределенная, каждый разработчик копирует с сервера репозиторий и может с ним работать у себя локально (коммитить), без фиксирования изменений на сервере (без пушей), если например в данный момент у разработчика нет интернета. На сервере лежит версия ветки master, которую разработчики условно считают эталонной, но по сути у каждого на машине лежит равноценная версия репозитория. Если же система версий нераспределенная (SVN) каждый разработчик также копирует с сервера версию репозитория, но все изменения, которые он вносит, будут сразу же происходить на сервере (аналогично commit + push в GIT), то есть разработчик не может у себя на машине хранить версию отличающуюся от условной эталонной. SVN хранит файлы целиком, в отличии от GIT, который хранит данные об измененных файлах - в итоге SVN работает в разы медленне и требует больше памяти. В SVN ветки представляют собой папки, в которых лежит другая версия проекта			
Алгоритм, эффективность и сложность алгоритмов.	**Алгоритм** – это точное предписание, однозначно определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату Есть несколько подходов к оценке сложности алгоритмов: Big-O, Little-O, Theta, Omega **Самый употребляемый и полезный подход - О-нотация** O(f(x)), где f(x) — формула, выражающая сложность алгоритма. В формуле может присутствовать переменная n, представляющая размер входных данных. Показывает, как зависит скорость работы алгоритма от размера входных данных: Линейно — O(n); Логарифмически — O(log n); Линеарифметически — O(n·log n); Квадратично — O(n^2)			
Многопоточность. Понятие потока и процесса. Дедлоки.	**Многопото́чность** — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины. **Процесс** — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство **Поток** – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса. **Дедлок**: у нас есть два потока, которые поочередно ожидают два ивента: A и B. Возможна такая ситуация, при которой первый поток захватит событие A и будет ожидать освобождения события B, а второй поток, наоборот, захватит event B и будет ждать A. В этом случае мы получим взаимоблокировку, которая приведет к полному зависанию этих двух тредов.			
Методы синхронизации в многопоточной среде. Блокировки, мьютексы и семафоры.	**A lock** allows only one thread to enter the part that's locked and the lock is not shared with any other processes. **A mutex** is the same as a lock but it can be system wide (shared by multiple processes). **A semaphore** does the same as a mutex but allows x number of threads to enter.			
Кэширование. Популярные виды кэширования.	**Кэш** — промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Доступ к данным в кэше осуществляется быстрее, чем выборка исходных данных из более медленной памяти или удаленного источника, однако её объём существенно ограничен по сравнению с хранилищем исходных данных. Алгоритмы кеширования: **1. Алгоритм предвидения** - отбрасывать из кэша ту информацию, которая не понадобится в будущем дольше всего. **2. Least recently used** (Вытеснение давно неиспользуемых) - в первую очередь, вытесняется неиспользованный дольше всех. **3. Most Recently Used** (Наиболее недавно использовавшийся) - в отличие от LRU, в первую очередь вытесняется последний использованный элемент. **4. Least-Frequently Used** (Наименее часто используемый) - LFU подсчитывает как часто используется элемент. Те элементы, обращения к которым происходят реже всего, вытесняются в первую очередь. Есть еще много разных, но этого должно хватить			

Методологии разработки. XP.	Экстрема́льное программи́рование (англ. Extreme Programming, XP) — одна из гибких методологий разработки программного обеспечения. Название методологии исходит из идеи применить полезные традиционные методы и практики разработки программного обеспечения, подняв их на новый «экстремальный» уровень, заключается в 12 следующих принципах, включенных в четыре группы: *Короткий цикл обратной связи (Fine-scale feedback)* **1. Разработка через тестирование (Test-driven development)** - см следующий вопрос **2. Игра в планирование (Planning game)** - быстро сформировать приблизительный план работы и постоянно обновлять его по мере того, как условия задачи становятся всё более чёткими. Артефактами игры в планирование является набор бумажных карточек, на которых записаны пожелания заказчика (customer stories), и приблизительный план работы по выпуску следующих одной или нескольких небольших версий продукта. **3. Заказчик всегда рядом (Whole team, Onsite customer) -** не тот, кто оплачивает счета, а конечный пользователь программного продукта. XP утверждает, что заказчик должен быть всё время на связи и доступен для вопросов. **4. Парное программирование (Pair programming)** - предполагает, что весь код создается парами программистов, работающих за одним компьютером. Один из них работает непосредственно с текстом программы, другой просматривает его работу и следит за общей картиной происходящего. *Непрерывный, а не пакетный процесс* **5. Непрерывная интеграция (Continuous integration)** - Если выполнять интеграцию разрабатываемой системы достаточно часто, то можно избежать большей части связанных с ней проблем. **6. Рефакторинг (Design improvement, Refactoring)** - XP подразумевает, что однажды написанный код в процессе работы над проектом почти наверняка будет неоднократно переделан. **7. Частые небольшие релизы (Small releases)** - Версии (releases) продукта должны поступать в эксплуатацию как можно чаще. Работа над каждой версией должна занимать как можно меньше времени. При этом каждая версия должна быть достаточно осмысленной с точки зрения полезности для бизнеса. *Понимание, разделяемое всеми* **8. Простота (Simple design)** - XP исходит из того, что в процессе работы условия задачи могут неоднократно измениться, а значит, разрабатываемый продукт не следует проектировать заблаговременно целиком и полностью. **9. Метафора системы** - Разработчикам необходимо проводить анализ архитектуры программного обеспечения для того, чтобы понять, в каком месте системы необходимо добавить новую функциональность, и с чем будет взаимодействовать новый компонент. **10. Коллективное владение кодом (Collective code ownership) или выбранными шаблонами проектирования (Collective patterns ownership)** - Коллективное владение означает, что каждый член команды несёт ответственность за весь исходный код. Таким образом, каждый вправе вносить изменения в любой участок программы. **11. Стандарт кодирования (Coding standard or Coding conventions)** - Все члены команды в ходе работы должны соблюдать требования общих стандартов кодирования. *Социальная защищённость программиста (Programmer welfare):* **12. 40-часовая рабочая неделя (Sustainable pace, Forty-hour week)**			
Методологии разработки. CI - Continuous Integration.	**Непрерывная интеграция** (англ. Continuous Integration) — это практика разработки программного обеспечения, которая заключается в выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем. Русским языком: система, которая собирает билды самостоятельно при появлении изменений, сама прогоняет его через тесты и доставляет всем заинтересованным (тестер, заказчик). Если билд не проходит тесты или даже не компилируется, билд не доставляется дальше, а разработчикам приходят оповещения, что в последнем коммите что-то было не так - легко найти, чьи изменения принесли косяк. Систем CI довольно много: CruiseControl, CruiseControl.Net, Atlassian Bamboo, Hudson, Microsoft Team Foundation Server, TeamCity, пусть и не OpenSource, но все-же бесплатный для небольших проектов.			
Методологии разработки. TDD.	Разработка через тестирование (англ. test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам. **1. Добавление теста** Неизбежно этот тест не будет проходить, поскольку соответствующий код ещё не написан. **2. Запуск всех тестов: убедиться, что новые тесты не проходят** **3. Написать код** На этом этапе пишется новый код так, что тест будет проходить. Этот код не обязательно должен быть идеален. Допустимо, чтобы он проходил тест каким-то неэлегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его. Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней и, соответственно, не тестируемой функциональности. **4. Запуск всех тестов: убедиться, что все тесты проходят** **5. Рефакторинг** **6. Повторение цикла** *Разработка через тестирование тесно связана с такими принципами как «делай проще, дурачок» (англ. keep it simple, stupid, KISS) и «вам это не понадобится» (англ. you ain't gonna need it, YAGNI). Дизайн может быть чище и яснее, при написании лишь того кода, который необходим для прохождения теста.*			