**1. В чем идея инкапсуляции? Абстракция, интерфейс и реализация. Средства защиты и доступа.**

**Инкапсуляция (Tính đóng gói) — это заключение данных и функционала в оболочку. они не только собирают переменные и методы в одном месте, но и защищают их от вмешательства извне (сокрытие).**

**Абстрагирование - это обобщение данных и поведения для типа, находящегося выше текущего класса по иерархии**. *(Tổng quát hóa 1 cái gì đó lên, không cần chú ý bên trong)*

**Пример:** Абстракция означает предоставление только необходимого функционала конечному пользователю. Вам не нужно понимать, как работает двигатель для управления автомобилем. Все, что вам нужно знать, это как завести автомобиль, как добавить газу, притормозить и остановиться. Вам не нужно особо думать над тем, что под капотом автомобиля есть двигатель, топливный насос, фильтры, радиатор и т. д. *(Trừu tượng nghĩa là chỉ cung cấp chức năng cần thiết cho người dùng cuối. Bạn không cần phải hiểu động cơ hoạt động như thế nào để lái xe ô tô. Tất cả những gì bạn cần biết là khởi động xe, đổ xăng, giảm tốc độ và dừng xe. Bạn không cần phải suy nghĩ quá nhiều về động cơ, bơm nhiên liệu, các bộ lọc, bộ tản nhiệt,… dưới mui xe.)*

**Интерфейс (interface) – это внешний вид класса, выделяющий его существенные черты и не показывающий внутреннего устройства и секретов поведение.**Интерфейс – это набор методов класса, доступных для использования другими классами.  
**Примером простого** интерфейса может служить машина с коробкой-автоматом. Освоить её управление очень быстро сможет любой. С другой стороны, чтобы освоить управление современным пассажирским самолётом, необходимо несколько месяцев, а то и лет упорных тренировок. С другой стороны, вы никогда не заставите автомобиль подняться в воздух и перелететь из Москвы в Вашингтон  
*(****Interface****là 1 tập các thành phần chỉ có khai báo mà không có phần định nghĩa.* *Một interface được hiểu như là 1 khuôn mẫu mà mọi lớp thực thi nó đều phải tuân theo. Interface sẽ định nghĩa phần “****làm gì****” (khai báo) và những lớp thực thi interface này sẽ định nghĩa phần “****làm như thế nào****” (định nghĩa nội dung) tương ứng)*

**Реализация (implemention) – внутреннее представление класса, включая секреты его поведение.**

**Модификатор доступ**

**Public** – уровень предполагает доступ к компоненту с этим модификатором из экземпляра любого класса и любого пакета.

**Protected** – уровень предполагает доступ к компоненту с этим модификатором из экземпляров родного класса и классов-потомков, независимо от того, в каком пакете они находятся.

**Default** – уровень предполагает доступ к компоненту с этим модификатором из экземпляров любых классов, находящихся в одном пакете с этим классом.

**Private** – уровень предполагает доступ к компоненту с этим модификатором только из этого класса.

**internal -** Иногда нужно сделать компонент доступным только внутри одного файла — например, в Program.cs, Item.cs или любом другом. Для этого используется ключевое слово internal.

**2. В чем идея наследования? Простое, многоуровневое, множественное наследование.**

**Наследование**— это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.  
*(VD: 2 lớp Android, iPhone  
Mỗi lớp đều đại diện cho một loại smartphone khác nhau nhưng lại có những thuộc tính giống nhau như gọi điện, nhắn tin, chụp hình. Thay vì sao chép những thuộc tính này, ta nên đặt chúng vào một lớp chung gọi là lớp cha. Chúng ta có thể định nghĩa lớp cha – trong trường hợp này là Smartphone và có những lớp con kế thừa từ nó, tạo ra một mối quan hệ cha/con.)*

«**простого (или одиночного) наследования**» (англ. single inheritance), при котором класс может наследоваться только от одного суперкласса.  
*(Lớp con chỉ có duy nhất 1 lớp cha)*

#### Мно́жественное насле́дование (Multiple Inheritance) — свойство, поддерживаемое частью объектно-ориентированных языков программирования, когда класс может иметь более одного суперкласса (непосредственного класса-родителя), интерфейсы поддерживают множественное наследование во многих языках программирования. *(Đa kế thừa là một tính năng của ngôn ngữ C++. Trong đó một lớp có thể kế thừa từ nhiều hơn một lớp khác. Nghĩa là một lớp con được kế thừa từ nhiều hơn một lớp cơ sở.)*

**Многоуровневое наследование** появляется, когда потомок наследуется от родителя, а затем сам становится родителем.   
В многоуровневом наследовании последний класспотомок, наследует атрибуты и поведение всех классов в цепочке наследования.  
*(Kế thừa đa cấp xảy ra khi một đứa trẻ kế thừa từ cha mẹ và sau đó trở thành chính cha mẹ. Trong kế thừa lớp, lớp kế thừa cuối cùng kế thừa các thuộc tính và hành vi của tất cả các lớp trong chuỗi kế thừa.)*

**3. В чем идея полиморфизма? Раннее и позднее (динамическое) связывание.  
Полиморфизм —** это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.  
В более общем смысле, концепцией полиморфизма является идея “один интерфейс, множество методов“.  
*(Tính đa hình là một hành động có thể được thực hiện bằng nhiều cách khác nhau. Đây lại là một tính chất có thể nói là chứa đựng hầu hết sức mạnh của lập trình hướng đối tượng.)*  
  
**Например**: Ta có 2 con vật: chó, mèo. Cả 2 con vật này đều là lớp động vật. Nhưng khi ta bảo cả 2 động vật kêu thì con chó sẽ kêu gâu gâu, con mèo sẽ kêu meo meo. Vậy trong ví dụ chó, mèo xem như là các đối tượng. 2 con vật có thể hiểu cùng kêu nhưng theo các cách khác nhau.

**Раннее связывание** (статическое) осущесталяется на этапе компиляции, когда компилятор точно знает какой метод и из какого класса будет вызван.  
*(Liên kết sớm (tĩnh) xảy ra tại thời điểm biên dịch, khi trình biên dịch biết chính xác phương thức nào và từ lớp nào sẽ được gọi.)*

**Позднее связывание** (дианамическое) заключается в разрешении ссылки на функцию во время исполнения, а не на этапе компиляции, когда компилятор не знает из какого класса будет браться вызываемый метод.  
*(Liên kết muộn (dianamic) là về việc giải quyết một tham chiếu đến một hàm trong thời gian chạy, không phải tại thời điểm biên dịch, khi trình biên dịch không biết phương thức được gọi sẽ đến từ lớp nào.)*  
  
**Пример**: В приведенном ниже примере программа obj содержит данные целочисленного типа, а obj1 — данные двойного типа. Но компилятор не разрешает их во время компиляции. Во время выполнения эти динамические объекты обнаруживаются и преобразуются в System.Int32 и System.Double соответственно. Вот почему процесс разрешения во время выполнения называется поздним связыванием. dynamic obj = 4; dynamic obj1 = 5.678; Console.WriteLine(obj.GetType()); Console.WriteLine(obj1.GetType())

**4. Принцип “информационного эксперта”. В чем идея? С каким принципом SOLID коррелирует?**

**Information Expert -** Базовый принцип разделения ответственности. Ответственность на себя должен брать тот, кто обладает максимумом информации для исполнения.  
Если класс – «эксперт» над какими-то данными, то методы для обработки этих данных должны быть в этом классе.

*(Nguyên tắc cơ bản của việc phân chia trách nhiệm  
Người có thông tin tối đa phải chịu trách nhiệm thực hiện.  
Nếu lớp là "chuyên gia" về một số dữ liệu, thì các phương thức xử lý dữ liệu này phải ở lớp này.)  
(****VD****: Muốn lấy ra thông tin của tất cả các Sách đang có trong thư viện, ta tạm gọi hành động này là một method getAllBooks(), dựa vào nguyên tắc Information Expert thì chúng ta nên đặt method getAllBooks() vào trong class BookStore.)*

**Коррелирует с принципом SRP: The Single Responsibility Principle из Solid  
Single responsibility principle:** Существует лишь одна причина, приводящая к изменению класса. Один класс должен решать только какую-то одну задачу.

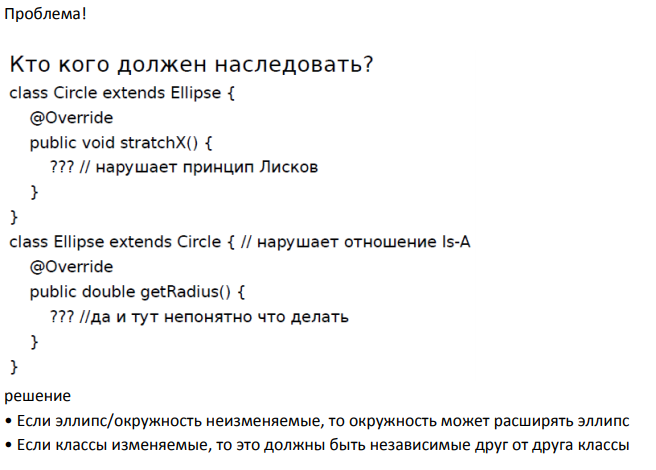
*(Mỗi lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi)*

**5. В чем заключается смысл принципов низкой связности и высокого зацепления (Low coupling & High cohesion)?**

**Low coupling -** модуль (класс) должен знать как можно меньше информации о другом модуле.*(Theo nguyên tắc này, một mô-đun (lớp) nên biết càng ít thông tin về mô-đun khác càng tốt.  
Nguyên tắc này nói chúng ta cần giảm các độ phụ thuộc xuống để ứng dụng của chúng ta có khả năng tái sử dụng cao hơn, ít bị ảnh hưởng hơn khi có bất cứ một thay đổi yêu cầu nào.)*

**High Cohesion -** Принцип декларирует, что ответственность модуля должна быть сосредоточена в одной области.*(Nguyên tắc tuyên bố rằng trách nhiệm của mô-đun nên được tập trung vào một khu vực)*

**Примером** хорошего дизайна системы может служить набор утилит GNU Binutils для Linux. В котором, каждая утилита (если ее рассматривать как модуль) выполняет лишь минимальные обязанности (высокое зацепление) и почти ничего не знает о природе других утилит (низкая связность), в связи с чем может быть легко заменена на аналог в некотором варианте использования.  
(*Một ví dụ về thiết kế hệ thống tốt là bộ tiện ích GNU Binutils dành cho Linux. Trong đó, mỗi tiện ích (nếu được coi là một mô-đun) chỉ thực hiện các nhiệm vụ tối thiểu (mức độ tương tác cao) và hầu như không biết gì về bản chất của các tiện ích khác (kết nối thấp), và do đó có thể dễ dàng thay thế bằng một tiện ích tương tự trong một số trường hợp sử dụng.)*

**6. В чем заключается смысл принципа подстановки Лисков. В каком случае и как нарушается принцип в проблеме эллипса-окружности (квадрата-прямоугольника)?  
Liskov substitution  
Согласно принципу LSP**, программы должны быть написаны таким образом, чтобы **в любом месте вместо базового типа мог быть подставлен подтип**. Это означает, что классы наследники должны реализовывать интерфейс согласовано с интерфейсом базового класса.  
**Принцип подстановки Лисков** требует использования открытого наследования лишь для реализации отношения «является разновидностью». Мы не можем установить факт нарушения принципа LSP до тех пор, пока не узнаем как именно будут использоваться наши классы.  
  
****

**7. Дайте описание принципам S, I, D из SOLID.  
Single responsibility principle:** Существует лишь одна причина, приводящая к изменению класса. Один класс должен решать только какую-то одну задачу.*(Mỗi lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi)*Он может иметь несколько методов, но они должны использоваться лишь для решения общей задачи. Все методы и свойства должны служить одной цели. Если класс имеет несколько назначений, его нужно разделить на отдельные классы **Пример**: модуль, формирующий отчеты, не должен отвечать за хранение и извлечение данных для отчетов**. (*Ví dụ****: Mô-đun báo cáo không nên chịu trách nhiệm lưu trữ và truy xuất dữ liệu cho các báo cáo.)*

**Interface segregation principle -** Много небольших интерфейсов для разных клиентов лучше, чем один большой общий интерфейс.  
Не стоит заставлять клиента реализовать интерфейс, которым он не пользуется.  
*(Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể)*  
**Пример:** работник-человек (работает и отдыхает) и работник-робот (отдых не нужен). Вместо одного интерфейса работник, делаем два интерфейса - работающий и отдыхающий. Для робота реализовываем только интерфейс работающий  
  
**Dependency inversion**   
- Высокоуровневые модули не должны зависеть от низкоуровневых. Оба вида модулей должны зависеть от абстракций.   
- Абстракции не должны зависеть от подробностей. Подробности должны зависеть от абстракций.   
Проще говоря: зависьте от абстракций, а не от чего-то конкретного.   
**Применяя** этот принцип, одни модули можно легко заменять другими, всего лишь меняя модуль зависимости, и тогда никакие перемены в низкоуровневом модуле не повлияют на высокоуровневый.  
*(1.Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.  
2.Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại (Các class giao tiếp với nhau thông qua interface (abstraction), không phải thông qua implementation.)****Ví dụ****, để kết nối tới Database, ta thường thiết kế lớp trừu tượng DataAccess có các phương thức phương thức chung như save(), get(), … Sau đó tùy vào việc sử dụng loại DBMS nào (vd: MySql, MongoDB, …) mà ta kế thừa và implement những phương thức này.)*

**8. Дайте описание принципам O, L из SOLID.**

**Open/Closed principle -** Программные сущности должны быть открыты для расширения, но закрыты для модификации  
*(Không được sửa đổi một Class có sẵn, nhưng có thể mở rộng bằng kế thừa.* ***Расширения****: mở rộng,* ***модификации****: sửa đổi  
Theo nguyên lý này, mỗi khi ta muốn thêm chức năng cho chương trình, chúng ta nên viết class mới mở rộng class cũ (bằng cách kế thừa hoặc sở hữu class cũ) chứ không nên sửa đổi class cũ. Việc này dẫn đến tình trạng phát sinh nhiều class, nhưng chúng ta sẽ không cần phải test lại các class cũ nữa, mà chỉ tập trung vào test các class mới, nơi chứa các chức năng mới.)*

**Liskov substitution principle** - Производный класс должен быть взаимозаменяем с базовым классом.  
*(Các đối tượng (instance) kiểu class con có thể thay thế các đối tượng kiểu class cha mà không gây ra lỗi.)*  
Для этого должны выполняться условия:  
1. Предварительные условия не могут быть усилены в подтипе.  
2. Постусловия не могут быть ослаблены в подтипе.  
3. Инварианты супертипа должны быть сохранены в подтипе.  
*(Quay trở lại ví dụ lớp Emloyee trong phần 1, ta giả sử có công ty sẽ điểm danh vào mỗi buổi sáng, và chỉ có các nhân viên thuộc biên chế chính thức mới được phép điểm danh. Ta bổ sung phương thức checkAttendance() vào lớp Employee.  
Hình dung có một trường hợp sau: công ty thuê một nhân viên lao công để làm vệ sinh văn phòng, mặc dù là một người làm việc cho công ty nhưng do không được cấp số ID nên không được xem là một nhân viên bình thường, mà chỉ là một nhân viên thời vụ, do đó sẽ không được điểm danh.  
Nguyên tắc này nói rằng: Nếu chúng ta tạo ra một lớp CleanerStaff kế thừa từ lớp Employee, và implement hàm working() cho lớp này, thì mọi thứ đều ổn, tuy nhiên lớp mới này cũng lại có hàm checkAttendance() để điểm danh, mà như thế là sai quy định dẫn đến chương trình bị lỗi. Như vậy, thiết kế lớp CleanerStaff kế thừa từ lớp Employee là không được phép.  
Có nhiều cách để giải quyết tình huống này ví dụ như tách hàm checkAttendance() ra một interface riêng và chỉ cho các lớp Developer, Tester và Salesman implements interface này.)*

**9. Паттерн одиночка. В чем идея? В чем опасность использования?  
Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.***(Single Pattern là một design pattern mà  
+ Đảm bảo rằng một class chỉ có duy nhất một instance (khởi tạo - mình xin phép để nguyên không dịch từ này)  
+ Và cung cấp một cách toàn cầu để truy cấp tới instance đó)* **Решение:** Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.  
Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.  
*(Tất cả các triển khai singleton đều ẩn đi hàm tạo mặc định và tạo một phương thức tĩnh công khai sẽ kiểm soát vòng đời của đối tượng singleton.  
Nếu bạn có quyền truy cập vào lớp singleton, thì bạn cũng sẽ có quyền truy cập vào phương thức tĩnh này. Từ bất kỳ điểm nào trong mã bạn gọi nó, nó sẽ luôn trả về cùng một đối tượng.)***Пример:****Правительство государства — хороший пример одиночки. В государстве может быть только одно официальное правительство. Вне зависимости от того, кто конкретно заседает в правительстве, оно имеет глобальную точку доступа «Правительство страны N».  
(+)   
- Гарантирует наличие единственного экземпляра класса.**  
*(Đảm bảo có một phiên bản duy nhất của lớp.)*  
- **Предоставляет к нему глобальную точку доступа**.  
*(Cung cấp một điểm truy cập toàn cầu cho nó.)***- Реализует отложенную инициализацию объекта-одиночки.**  
 *(Triển khai khởi tạo lười biếng của một đối tượng singleton.)*  
**(-)  
-Нарушает принцип единственной ответственности класса.**  
*(Vi phạm nguyên tắc chỉ trách nhiệm của lớp)*  
**- Маскирует плохой дизайн.**  
*(Cải trang thiết kế xấu)*  
**- Проблемы мультипоточности.**  
*(Vấn đề đa luồng)*  
**- Требует постоянного создания Mock-объектов при юнит-тестировании.**  
*(Yêu cầu liên tục tạo các đối tượng Mock trong quá trình thử nghiệm đơn vị.)*

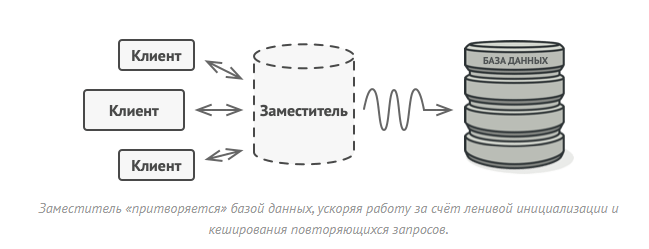
**10.** **Паттерн состояние. В чем идея? Какое ключевое отличие от паттерна стратегия?  
Состояние (State) —** это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.  
*(State là một mẫu thiết kế hành vi cho phép các đối tượng thay đổi hành vi dựa trên trạng thái của chúng. Nhìn từ bên ngoài, có vẻ như lớp của đối tượng đã thay đổi.)***Решение:***(Hành vi của một đội tượng phụ thuộc vào trạng thái của nó. Tại thời điểm runtime, khi đối tượng thực hiện hành vi, trạng thái của nó sẽ thay đổi theo.)*Паттерн Состояние предлагает создать отдельные классы для каждого состояния, в котором может пребывать объект, а затем вынести туда поведения, соответствующие этим состояниям.  
Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.  
(*State pattern gợi ý tạo các lớp riêng biệt cho mỗi trạng thái mà một đối tượng có thể cư trú, và sau đó chuyển đến đó các hành vi tương ứng với các trạng thái này.  
Thay vì lưu mã cho tất cả các trạng thái, đối tượng ban đầu, được gọi là ngữ cảnh, sẽ chứa một tham chiếu đến một trong các đối tượng trạng thái và ủy quyền công việc phụ thuộc vào trạng thái cho nó.)***Пример:** *Ví dụ ngân hàng có trạng thái rút tiền, chuyển tiền và nạp tiền. Thì đối với mỗi 1 trạng thái thì sẽ thực hiện hoạt động khác nhau.*  
**Паттерн состояние отличие от паттерна стратегия  
Состояние можно рассматривать как надстройку над Стратегией. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в Стратегии эти объекты не знают друг о друге и никак не связаны. В Состоянии сами конкретные состояния могут переключать контекст**. ***(****Một sắc thái rất quan trọng giúp phân biệt mô hình này với Chiến lược là bản thân bối cảnh và các trạng thái cụ thể có thể biết về nhau và bắt đầu chuyển đổi từ trạng thái này sang trạng thái khác.  
State có thể được xem như một cấu trúc thượng tầng so với Chiến lược. Cả hai mẫu đều sử dụng thành phần để thay đổi hành vi của đối tượng chính bằng cách ủy quyền công việc cho các đối tượng trợ giúp lồng nhau. Tuy nhiên, trong Chiến lược, các đối tượng này không biết về nhau và không được kết nối theo bất kỳ cách nào. Trong một state, các state cụ thể có thể chuyển đổi ngữ cảnh.)*

**11. Паттерн стратегия. В чем идея? Какое ключевое отличие от паттерна состояние?  
Стратегия —** это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.  
*(Chiến lược là một mẫu thiết kế hành vi xác định một nhóm các thuật toán tương tự và đặt mỗi thuật toán trong số chúng vào lớp riêng của nó, sau đó các thuật toán có thể được thay thế cho nhau ngay trong thời gian chạy.)***Решение:**  
Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.  
*(Mô hình Chiến lược gợi ý xác định một họ các thuật toán tương tự thường được thay đổi hoặc mở rộng, và chuyển chúng vào các lớp riêng của chúng, được gọi là chiến lược)*  
Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.  
*(Thay vì lớp ban đầu thực thi thuật toán này hoặc chính thuật toán đó, nó sẽ hoạt động như một ngữ cảnh, tham chiếu đến một trong các chiến lược và ủy quyền công việc cho nó. Để thay đổi thuật toán, bạn chỉ cần thay một đối tượng chiến lược khác vào ngữ cảnh.)*  
Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.  
*(Điều quan trọng là tất cả các chiến lược phải có một giao diện chung. Sử dụng giao diện này, ngữ cảnh sẽ độc lập với các lớp chiến lược cụ thể. Mặt khác, bạn có thể sửa đổi và thêm các loại thuật toán mới mà không cần chạm vào mã ngữ cảnh.)***Пример:**Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией. Вы выбираете конкретную стратегию в зависимости от контекста — наличия денег или времени до отлёта.  
*(Bạn cần phải đến sân bay. Bạn có thể đến đó bằng xe buýt, taxi hoặc xe đạp. Phương thức vận tải là một chiến lược ở đây. Bạn chọn một chiến lược cụ thể tùy thuộc vào bối cảnh - sẵn có tiền hoặc thời gian trước khi khởi hành.)*

**12. Паттерн легковес. В чем идея? Когда его следует применять?**

**Легковес** — это структурный паттерн проектирования, который позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.  
*(Flyweight Pattern được sử dụng để giảm sự căng thẳng cho bộ nhớ của nó. Điều này rất quan trọng đối với các thiết bị không có nhiều bộ nhớ, cũng như tối ưu hóa ứng dụng.  
Khi một ứng dụng nhất định cần tạo nhiều instance của cùng một class. Khi đó một nhóm chung được tạo để các ứng dụng tương tự có thể được sử dụng lại, thay vì phải tạo lại.)***Применимость**  
**Когда не хватает оперативной памяти для поддержки всех нужных объектов.** *(Khi không có đủ RAM để hỗ trợ tất cả các đối tượng cần thiết)*  
Эффективность паттерна Легковес во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены все перечисленные условия:  
*(Hiệu quả của mẫu Lightweight phần lớn phụ thuộc vào cách thức và vị trí nó được sử dụng. Áp dụng mẫu này khi tất cả các điều kiện sau được đáp ứng:)*  
+ в приложении используется большое число объектов;  
*(ứng dụng sử dụng một số lượng lớn các đối tượng)*  
+ из-за этого высоки расходы оперативной памяти;  
*(vì điều này, chi phí RAM cao)*  
+ большую часть состояния объектов можно вынести за пределы их классов;  
*(hầu hết trạng thái của các đối tượng có thể được đưa ra khỏi các lớp của chúng)*  
+ большие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено  
(Các nhóm đối tượng lớn có thể được thay thế bằng một số lượng tương đối nhỏ *các đối tượng được chia sẻ vì trạng thái bên ngoài được hiển thị.)*

**13. Паттерн заместитель. В чем идея? Какие виды (типы) заместителей бывают? Для решения каких задач применяются?**

**Заместитель —** это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.  
*(Trình giữ chỗ là một mẫu thiết kế cấu trúc cho phép bạn thay thế các đối tượng giữ chỗ đặc biệt cho các đối tượng thực. Các đối tượng này chặn các cuộc gọi đến đối tượng gốc, cho phép bạn thực hiện điều gì đó trước hoặc sau khi chuyển lệnh gọi đến đối tượng gốc.)***Решение**  
Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.  
*(Mẫu Proxy gợi ý tạo một lớp sơ khai mới có giao diện giống với đối tượng dịch vụ ban đầu. Khi nhận được yêu cầu từ khách hàng, đối tượng proxy sẽ tự khởi tạo đối tượng dịch vụ và chuyển tiếp tất cả công việc thực sự tới nó.)***Применение**  
**1. Ленивая инициализация (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных***(Khởi tạo lười biếng (proxy ảo). Khi bạn có một đối tượng nặng đang tải dữ liệu từ hệ thống tệp hoặc cơ sở dữ liệu.).*  
Вместо того, чтобы грузить данные сразу после старта программы, можно сэкономить ресурсы и создать объект тогда, когда он действительно понадобится.  
*(Thay vì tải dữ liệu ngay sau khi bắt đầu chương trình, bạn có thể tiết kiệm tài nguyên và tạo một đối tượng khi thực sự cần.)*  
**2. Защита доступа (защищающий прокси). Прокси может проверять доступ при каждом вызове и передавать выполнение служебному объекту, если доступ разрешён.***(Bảo vệ truy cập (bảo vệ proxy). Một proxy có thể kiểm tra quyền truy cập trên mọi cuộc gọi và chuyển việc thực thi cho một đối tượng dịch vụ nếu quyền truy cập được cho phép.)***3. Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.**  
(Khởi động dịch vụ cục bộ (proxy từ xa). Khi đối tượng dịch vụ thực ở trên máy chủ từ xa.)  
**4. Кеширование объектов («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.**  
(Bộ nhớ đệm đối tượng (liên kết "thông minh"). Khi bạn cần lưu vào bộ nhớ cache các kết quả yêu cầu của khách hàng và quản lý vòng đời của họ.)

**14. Паттерн команда. В чем идея? Как его можно комбинировать с партнёром снимок?  
Команда — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.**  
*(Lệnh là một mẫu thiết kế hành vi biến các yêu cầu thành các đối tượng, cho phép chúng được truyền dưới dạng đối số khi gọi các phương thức, xếp hàng yêu cầu, ghi nhật ký chúng và hỗ trợ hủy các hoạt động.)***Пример:  
Вы заходите в ресторан и садитесь у окна. К вам подходит вежливый официант и принимает заказ, записывая все пожелания в блокнот. Откланявшись, он уходит на кухню, где вырывает лист из блокнота и клеит на стену. Далее лист оказывается в руках повара, который читает содержание заказа и готовит заказанные блюда**.  
*(Bạn đi vào một nhà hàng và ngồi bên cửa sổ. Một người phục vụ lịch sự đến gặp bạn và nhận món, ghi tất cả mong muốn của bạn vào một cuốn sổ. Nghỉ phép, anh ta đi vào bếp, nơi anh ta lấy ra một tờ từ một cuốn sổ và dán nó lên tường. Hơn nữa, tờ giấy này nằm trong tay của đầu bếp, người đọc nội dung của đơn đặt hàng và chuẩn bị các món ăn theo yêu cầu.)*  
В этом примере вы являетесь отправителем, официант с блокнотом — командой, а повар — получателем. Как и в паттерне, вы не соприкасаетесь напрямую с поваром. Вместо этого вы отправляете заказ с официантом, который самостоятельно «настраивает» повара на работу. С другой стороны, повар не знает, кто конкретно послал ему заказ. Но это ему безразлично, так как вся необходимая информация есть в листе заказа.  
*(Trong ví dụ này, bạn là người gửi, người phục vụ với sổ ghi chú là nhóm và đầu bếp là người nhận. Đối với mô hình, bạn không tiếp xúc trực tiếp với đầu bếp. Thay vào đó, bạn gửi đơn đặt hàng với một người phục vụ, người này độc lập "sắp đặt" đầu bếp đi làm. Mặt khác, đầu bếp không biết chính xác ai là người đã gửi đơn đặt hàng cho mình. Nhưng nó không quan trọng với anh ta, vì tất cả các thông tin cần thiết đều có trong tờ đơn đặt hàng.)***Как его можно комбинировать с партнёром снимок  
Команду и Снимок можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.**  
*(Command và Snapshot có thể được sử dụng cùng nhau để thực hiện các thao tác hoàn tác. Trong trường hợp này, các đối tượng lệnh sẽ chịu trách nhiệm thực hiện các hành động trên đối tượng và ảnh chụp nhanh sẽ lưu trữ một bản sao lưu trạng thái của đối tượng này, được thực hiện trước khi lệnh được chạy.)*

**15. Паттерн снимок. В чем идея? Как его можно комбинировать с партнёром команда?  
Снимок (Memento) — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.**  
*(Ảnh chụp nhanh là một mẫu thiết kế hành vi cho phép bạn lưu và khôi phục trạng thái trước đây của các đối tượng mà không tiết lộ chi tiết triển khai.)***Решение:**Паттерн Снимок поручает создание копии состояния объекта самому объекту, который этим состоянием владеет. Вместо того, чтобы делать снимок «извне», наш редактор сам сделает копию своих полей, ведь ему доступны все поля, даже приватные.  
(*Snapshot pattern hướng dẫn tạo bản sao trạng thái của đối tượng cho chính đối tượng sở hữu trạng thái này. Thay vì chụp ảnh “từ bên ngoài”, biên tập viên của chúng tôi sẽ tự tạo một bản sao các trường của anh ấy, vì tất cả các trường đều có sẵn cho anh ấy, ngay cả những trường riêng tư.)*Паттерн предлагает держать копию состояния в специальном объекте-снимке с ограниченным интерфейсом, позволяющим, например, узнать дату изготовления или название снимка. Но, с другой стороны, снимок должен быть открыт для своего создателя, позволяя прочесть и восстановить его внутреннее состояние.  
*(Mẫu đề xuất giữ một bản sao của trạng thái trong một đối tượng ảnh chụp nhanh đặc biệt với một giao diện hạn chế cho phép, ví dụ: ngày sản xuất hoặc tên của ảnh chụp nhanh. Nhưng, mặt khác, ảnh chụp nhanh phải được mở cho người tạo ra nó, cho phép anh ta đọc và khôi phục trạng thái bên trong của mình.)*Такая схема позволяет создателям производить снимки и отдавать их для хранения другим объектам, называемым опекунами. Опекунам будет доступен только ограниченный интерфейс снимка, поэтому они никак не смогут повлиять на «внутренности» самого снимка. В нужный момент опекун может попросить создателя восстановить своё состояние, передав ему соответствующий снимок.  
*(Sự sắp xếp này cho phép người sáng tạo chụp ảnh nhanh và đưa chúng để lưu trữ cho các thực thể khác được gọi là người giám hộ. Người giám hộ sẽ chỉ có quyền truy cập vào một giao diện ảnh chụp nhanh hạn chế, vì vậy họ sẽ không thể tác động đến "bên trong" của ảnh chụp nhanh theo bất kỳ cách nào. Vào đúng thời điểm, người giám hộ có thể yêu cầu tác giả khôi phục trạng thái của mình bằng cách đưa cho anh ta một bức ảnh chụp thích hợp.)*

***Cách hoạt động:***

*******+ Originator: là Object có trạng thái được lưu trữ hoặc khôi phục.  
+ Mementor: là trạng thái (State) của Object khi đang được lưu trữ.  
+ CareTaker: đóng vai trò lưu trữ và cấp phát các Memento. Nó có trách nghiệm lưu trữ các State ở dạng Memento và cấp phát các State cho các Object khi cần.*

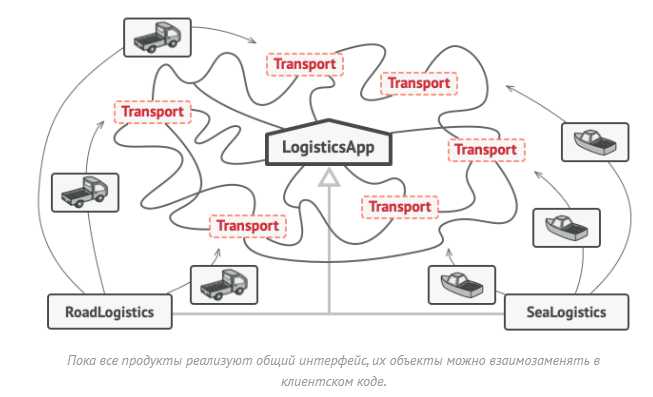
*Memento Pattern sẽ cấu trúc các dữ liệu cần lưu của một Object thành một State, sau đó sẽ lưu lại State này. Các State sau khi được lưu lại sẽ được gọi là các Memento. CareTaker sẽ đóng vai trò lưu trữ các State thành Memento và xuất các Memento thành State để có thể sử dụng. Do trạng thái của các Object đều được lưu trữ trong State nên khi State này được truyền qua các Object khác nhau thì sẽ không để lộ các implement chi tiết của các Object đó.*

*Memento Pattern được sử dụng bất cứ khi nào chúng ta muốn lưu và sau đó khôi phục trạng thái của một Object. Ví dụ như khi chúng ta chơi game, chúng ta muốn lưu lại tất cả những trạng thái chúng ta đã chơi trước đó để sau khi quit game và mở lại thì chúng ta có thể tiếp tục chơi, khi đó ta có thể áp dụng Memento Pattern để giải quyết bài toán này.*

**16. Паттерн фабричный метод. В чем идея? В чем разница между паттернам фабричный метод и шаблонный метод?**

**Фабричный метод (Factory) —** это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов**.***(Phương thức nhà máy là một mẫu thiết kế tổ tiên xác định một giao diện chung để tạo các đối tượng trong lớp cha, cho phép các lớp con thay đổi loại đối tượng mà chúng tạo ra.)*

**Например, классы Грузовик и Судно реализуют интерфейс Транспорт с методом доставить. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда —по морю. Фабричный метод в классе ДорожнойЛогистики вернёт объект-грузовик, а класс МорскойЛогистики — объект-судно.**  
(*Ví dụ, các lớp Xe tải và Tàu biển triển khai giao diện Vận chuyển với một phương thức giao hàng. Mỗi lớp này thực hiện phương thức theo cách riêng của nó: xe tải vận chuyển hàng hóa bằng đường bộ và tàu - bằng đường biển. Phương thức factory trong lớp RoadLogistics sẽ trả về đối tượng xe tải và lớp MarineLogistics sẽ trả về đối tượng tàu.)*  
Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный Транспорт. Для него будет важно, чтобы объект имел метод доставить, а как конкретно он работает — не важно.  
*(Đối với client của phương thức factory, không có sự khác biệt giữa các đối tượng này, vì anh ta sẽ coi chúng như một loại Giao thông vận tải trừu tượng nào đó. Điều quan trọng đối với anh ta là đối tượng có một phương pháp để phân phối, nhưng nó hoạt động chính xác như thế nào không quan trọng.)*



**Различия фабричного и шаблонного метода:** разные типы паттернов(порождающий и поведенческий), фабричный - частный случай шаблонного.  
*(Sự khác biệt giữa các phương pháp nhà máy và khuôn mẫu: các loại mẫu khác nhau (phổ biến và hành vi), nhà máy là một trường hợp đặc biệt của khuôn mẫu.)* **17. Паттерн шаблонный метод. В чем идея? В чем разница между паттернам фабричный метод и шаблонный метод?**Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.  
*(Phương thức mẫu là một mẫu thiết kế hành vi xác định khung của một thuật toán, chuyển trách nhiệm đối với một số bước của nó cho các lớp con. Mẫu cho phép các lớp con xác định lại các bước của một thuật toán mà không thay đổi cấu trúc tổng thể của nó.)  
(Bạn sử dụng mẫu thiết kế Template Method khi bạn có một thuật toán với nhiều bước và bạn muốn cho phép tùy chỉnh chúng trong lớp con. Thật dễ dàng. Bằng cách viết lại các hàm đã được khai báo trong lớp trừu tượng, bạn sẽ thay đổi được theo cách mà bạn mong muốn. Mẫu thiết kế Template Method là ý tưởng tuyệt vời khi bạn có một thuật toán nhiều bước mà chính bạn có thể tùy chỉnh nó.)****Пример:*** *Bạn nhận được một hợp đồng về những con robot tự động lắp ráp xe hơi. Không vấn đề gì, các lập trình viên nói và bắt đầu bắt tay ngay vào công việc.  
Bạn sẽ tạo ra lớp Robot và có 4 hàm start(); getParts(); assemble();stop() (Starting...; Getting a carburetor...; Installing the carburetor...; Stopping...). Tiếp theo, bạn nhận được 1 hợp đồng mới, 1 phần mềm làm 1 con robot làm bánh nướng. Và các lập trình viên bắt đầu đắn đo, sẽ phải viết lại tất cả phần mềm từ đầu. Đây là thời điểm thích hợp để nói về mấu thiết kế Template Method. Có một rắc rối mà lập trình viên phải đối mặt, họ có 1 con robot lắp ráp xe hơi, nhưng bây giờ ho cần 1 con robot nướng bánh. Và họ phải viết lại toàn bộ mã nguồn. Con Robot nướng bánh có một số chức năng giống với con robot lắp ráp xe hơi như hàm start(), stop(), tuy nhiên nó cũng có 1 số khác biệt như hàm assemble() sẽ không hiển thị là Getting a carburetor... mà thay vào đó là Getting flour and sugar.... Mẫu thiết kế Template Method có thể giải quyết được vấn đề này. Mẫu này nói rằng, bạn có thể viết một phương thức, dùng để xác định một loạt các thuật toán, giống như hàm go mà bạn thấy trước đây, để chạy một loạt các chức năng cho robot. Trong trường hợp này, để làm một con robot nướng bánh, bạn sẽ viết lại các hàm geParts() và asemble(). Bạn nên sử dụng mẫu Template Method khi bạn có một thuật toán được tạo bởi nhiều bước, và bạn muốn tùy chỉnh một số bước trong đó.*

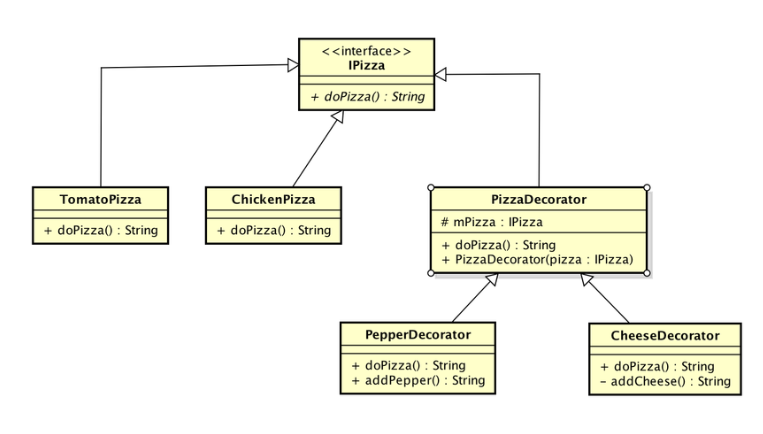
Строители используют подход, похожий на шаблонный метод при строительстве типовых домов. У них есть основной архитектурный проект, в котором расписаны шаги строительства: заливка фундамента, постройка стен, перекрытие крыши, установка окон и так далее.  
*(Các nhà xây dựng sử dụng cách tiếp cận giống như mẫu để xây dựng những ngôi nhà điển hình. Họ có một bản thiết kế kiến ​​trúc cơ bản trong đó phác thảo các bước xây dựng: đổ móng, xây tường, lợp mái, lắp cửa sổ, v.v.)*Но, несмотря на стандартизацию каждого этапа, строители могут вносить небольшие изменения на любом из этапов, чтобы сделать дом чуточку непохожим на другие.  
*(Nhưng bất chấp tiêu chuẩn hóa của từng giai đoạn, người xây dựng có thể thực hiện những thay đổi nhỏ ở bất kỳ giai đoạn nào để làm cho ngôi nhà có một chút khác biệt so với những ngôi nhà khác.)*

**18. Паттерн посетитель. В чем идея?** **Какие требования предъявляются к классам, которые можно посещать?  
Посетитель (Visitor) — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.***(Khách truy cập là một mẫu thiết kế hành vi cho phép bạn thêm các thao tác mới vào chương trình mà không thay đổi các lớp đối tượng mà các thao tác này có thể được thực hiện.)***Решение:**Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.  
*(Mẫu khách truy cập đề xuất đặt hành vi mới trong một lớp riêng biệt, thay vì nhân nó trong một số lớp cùng một lúc. Các đối tượng mà hành vi lẽ ra được liên kết với sẽ không tự thực thi nó. Thay vào đó, bạn sẽ chuyển các đối tượng này cho các phương thức của khách truy cập.)***Например:**Представьте начинающего страхового агента, жаждущего получить новых клиентов. Он беспорядочно посещает все дома в округе, предлагая свои услуги. Но для каждого из посещаемых типов домов у него имеется особое предложение.  
*(Hãy tưởng tượng một đại lý bảo hiểm lần đầu tìm cách kiếm khách hàng mới. Anh ta ngẫu nhiên đến thăm tất cả các ngôi nhà trong khu vực, đưa ra các dịch vụ của mình. Nhưng đối với mỗi loại nhà được thăm viếng, anh ấy có một ưu đãi đặc biệt.)*  
+ Придя в дом к обычной семье, он предлагает оформить медицинскую страховку.  
*(Đến nhà của một gia đình bình thường, anh ta đề nghị mua bảo hiểm y tế.)*  
+ Придя в банк, он предлагает страховку от грабежа.  
*(Đến ngân hàng, anh ta cung cấp bảo hiểm cướp.)*+ Придя на фабрику, он предлагает страховку предприятия от пожара и наводнения.  
*(Khi đến nhà máy, anh ta đề nghị công ty bảo hiểm hỏa hoạn và lũ lụt.)***Какие требования предъявляются к классам, которые можно посещать**  
Когда над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но вы не хотите «засорять» классы такими операциями. Посетитель позволяет извлечь родственные операции из классов, составляющих структуру объектов, поместив их в один класс-посетитель. Если структура объектов является общей для нескольких приложений, то паттерн позволит в каждое приложение включить только нужные операции.  
*(Khi bạn cần thực hiện một số thao tác không liên quan trên các đối tượng có cấu trúc đối tượng phức tạp, nhưng bạn không muốn "xả rác" các lớp với các thao tác như vậy. Một khách truy cập cho phép bạn trích xuất các hoạt động liên quan từ các lớp tạo nên cấu trúc của các đối tượng bằng cách đặt chúng vào một lớp khách truy cập duy nhất. Nếu cấu trúc của các đối tượng là chung cho một số ứng dụng, thì mẫu sẽ cho phép mỗi ứng dụng chỉ bao gồm các hoạt động cần thiết.)*

Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии. Посетитель позволяет определить поведение только для этих классов, оставив его пустым для всех остальных.  
*(Khi hành vi mới chỉ có ý nghĩa đối với một số lớp từ hệ thống phân cấp hiện có. Người truy cập cho phép bạn xác định hành vi chỉ cho những lớp đó, để trống cho những người khác.)*

**19. Паттерн декоратор. В чем идея? Когда следует применять?  
Декоратор (Decorator) — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».**  
*(Decorator là một mẫu thiết kế cấu trúc cho phép bạn thêm động chức năng mới vào các đối tượng bằng cách gói chúng trong các "wrappers" hữu ích.)*

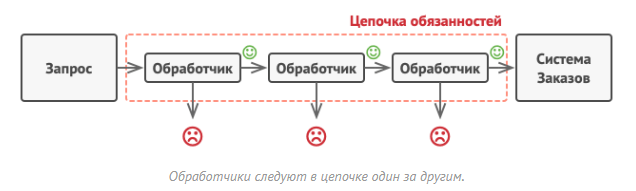
**Компонент:** общий интерфейс, поэтому объекты должны добавлять функциональные возможности во время выполнения, а затем реализовывать этот интерфейс.  
*(Component: giao diện (interface) chung để các đối tượng cần thêm chức năng trong quá trình chạy thì triển khai giao diện này.)*  
**ConcreteComponent:** реализация интерфейса Component, определяющая объект, которому требуется дополнительная функциональность во время выполнения.  
*(ConcreteComponent : Một cài đặt cho giao diện Component mà nó định nghĩa một đối tượng cần thêm các chức năng trong quá trình chạy.)*  
**Декоратор:** абстрактный класс, используемый для поддержки ссылки на объект компонента, а также для реализации компонентов интерфейса.  
*(Decorator : một lớp trừu tượng dùng để duy trì một tham chiếu của đối tượng thành phần và đồng thời cài đặt các thành phần của giao diện.)*

**ConcreteDecorator:** реализация Decorator, она устанавливает дополнительные компоненты в начале объектов компонентов.  
*(ConcreteDecorator : Một cài đặt của Decorator, nó cài đặt thêm các thành phần vào đầu của các đối tượng thành phần.)***Пример:**  


**Любая одежда — это аналог Декоратора. Применяя Декоратор, вы не меняете первоначальный класс и не создаёте дочерних классов. Так и с одеждой — надевая свитер, вы не перестаёте быть собой, но получаете новое свойство — защиту от холода. Вы можете пойти дальше и надеть сверху ещё один декоратор — плащ, чтобы защититься и от дождя** *(Bất kỳ quần áo nào cũng là một thứ tương tự của Trang trí. Bằng cách sử dụng Decorator, bạn không thay đổi lớp ban đầu hoặc tạo bất kỳ lớp con nào. Vì vậy, với quần áo - mặc một chiếc áo len, bạn không ngừng là chính mình mà còn có được một tài sản mới - sự bảo vệ khỏi cái lạnh. Bạn có thể đi xa hơn và đặt một đồ trang trí khác lên trên - một chiếc áo mưa để bảo vệ bạn khỏi mưa.)***Применимость**  
 **Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.**  
Объекты помещают в обёртки, имеющие дополнительные поведения. Обёртки и сами объекты имеют одинаковый интерфейс, поэтому клиентам без разницы, с чем работать — с обычным объектом данных или с обёрнутым. *(****Khi bạn cần thêm trách nhiệm cho các đối tượng đang bay, mã sử dụng chúng sẽ vô hình.*** *Các đối tượng được bao bọc với các hành vi bổ sung. Bản thân các trình bao bọc và các đối tượng có cùng một giao diện, vì vậy các máy khách không quan tâm đến việc làm việc với cái gì - với một đối tượng dữ liệu thông thường hay với một đối tượng được bao bọc.)*  
**Когда нельзя расширить обязанности объекта с помощью наследования.**Во многих языках программирования есть ключевое слово final, которое может заблокировать наследование класса. Расширить такие классы можно только с помощью Декоратора*. (****Khi bạn không thể mở rộng trách nhiệm của một đối tượng có tính kế thừa****. Nhiều ngôn ngữ lập trình có một từ khóa cuối cùng có thể chặn kế thừa lớp. Các lớp như vậy chỉ có thể được mở rộng bằng Decorator)*

**20. Паттерн адаптер. В чем идея? Когда следует применять?  
Адаптер (Adapter) — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.***(Bộ điều hợp là một mẫu thiết kế cấu trúc cho phép các đối tượng có giao diện không tương thích hoạt động cùng nhau.)*Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:  
*(Bộ điều hợp không chỉ có thể dịch dữ liệu từ định dạng này sang định dạng khác mà còn giúp các đối tượng có giao diện khác nhau hoạt động cùng nhau. Nó hoạt động như thế này:)*  
+ Адаптер имеет интерфейс, который совместим с одним из объектов.  
*(Bộ điều hợp có giao diện tương thích với một trong các đối tượng.)*  
+ Поэтому этот объект может свободно вызывать методы адаптера.  
*(Do đó, đối tượng này có thể tự do gọi các phương thức bộ điều hợp.)*+ Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.  
*(Bộ điều hợp nhận các cuộc gọi này và chuyển tiếp chúng đến đối tượng thứ hai, nhưng ở cùng một định dạng và trình tự mà đối tượng thứ hai hiểu được.)*

**Пример:** Когда вы в первый раз летите за границу, вас может ждать сюрприз при попытке зарядить ноутбук. Стандарты розеток в разных странах отличаются. Ваша европейская зарядка будет бесполезна в США без специального адаптера, позволяющего подключиться к розетке другого типа.  
*(Khi bạn bay ra nước ngoài lần đầu tiên, bạn có thể ngạc nhiên khi cố gắng sạc máy tính xách tay của mình. Tiêu chuẩn đầu ra khác nhau giữa các quốc gia. Bộ sạc châu Âu của bạn sẽ vô dụng ở Mỹ nếu không có bộ chuyển đổi chuyên dụng cho phép bạn cắm vào một ổ cắm khác.)***Применимость  
 Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.**Адаптер позволяет создать объект-прокладку, который будет превращать вызовы приложения в формат, понятный стороннему классу.   
*(****Khi bạn muốn sử dụng một lớp của bên thứ ba, nhưng giao diện của nó không khớp với phần còn lại của mã ứng dụng của bạn.*** *Bộ điều hợp cho phép bạn tạo một đối tượng shim sẽ biến các lệnh gọi ứng dụng của bạn thành một định dạng mà lớp bên thứ ba có thể hiểu được.)***Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете.**Более элегантным решением было бы поместить недостающую функциональность в адаптер и приспособить его для работы с суперклассом. Такой адаптер сможет работать со всеми подклассами иерархии. Это решение будет сильно напоминать паттерн Декоратор. *(Khi bạn cần sử dụng một số lớp con hiện có, nhưng chúng thiếu một số chức năng chung và bạn không thể mở rộng lớp cha. Một giải pháp thanh lịch hơn sẽ là đưa chức năng còn thiếu vào bộ điều hợp và điều chỉnh nó để hoạt động với lớp cha. Bộ điều hợp như vậy sẽ có thể hoạt động với tất cả các lớp con của hệ thống phân cấp. Giải pháp này sẽ gần giống với mẫu Decorator.)*

**21. Паттерн цепочка обязанностей. В чем идея? Когда следует применять?  
Цепочка обязанностей (Chain of Responsibility) —** это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.  
*(Chuỗi trách nhiệm là một mẫu thiết kế hành vi cho phép các yêu cầu được chuyển một cách tuần tự thông qua một chuỗi các trình xử lý. Mỗi trình xử lý tiếp theo sẽ quyết định xem liệu nó có thể tự xử lý yêu cầu hay không và liệu nó có đáng để chuyển yêu cầu xuống chuỗi hay không.)*

**Пример:** *Gọi điện đến tổng đài khi máy tính bị hỏng. Đầu tiên sẽ là nghe giọng của robot tự động trả lời, sau đó sẽ là nhân viên chăm sóc khách hàng, sau khi nhân viên chăm sóc khách hàng nắm được vấn đề của bạn thì sẽ kết nối bạn với 1 chuyên gia để giúp bạn xử lý vấn đề.***службу поддержки: tong dai  
сотрудник службу поддержки: nv cham soc khach hang**

**Применимость****Когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно, какие конкретно запросы будут приходить и какие обработчики для них понадобятся.  
*(Khi một chương trình phải xử lý các yêu cầu khác nhau theo nhiều cách, nhưng không thể biết trước những yêu cầu cụ thể nào sẽ đến và những trình xử lý nào sẽ cần thiết cho chúng)***С помощью Цепочки обязанностей вы можете связать потенциальных обработчиков в одну цепь и при получении запроса поочерёдно спрашивать каждого из них, не хочет ли он обработать запрос.  
*(Với Chuỗi trách nhiệm, bạn có thể liên kết những người xử lý tiềm năng thành một chuỗi và khi nhận được yêu cầu, hãy luân phiên hỏi từng người trong số họ xem họ có muốn xử lý yêu cầu hay không.)***Когда важно, чтобы обработчики выполнялись один за другим в строгом порядке. *(Khi điều quan trọng là các trình xử lý được thực hiện lần lượt theo thứ tự nghiêm ngặt.)***Цепочка обязанностей позволяет запускать обработчиков последовательно один за другим в том порядке, в котором они находятся в цепочке.  
*(Chuỗi trách nhiệm cho phép bạn chạy các trình xử lý tuần tự lần lượt theo thứ tự mà chúng xuất hiện trong chuỗi.)***Когда набор объектов, способных обработать запрос, должен задаваться динамически. *(Khi một tập hợp các đối tượng có khả năng xử lý một yêu cầu phải được đặt động.)***В любой момент вы можете вмешаться в существующую цепочку и переназначить связи так, чтобы убрать или добавить новое звено.  
*(Bất cứ lúc nào, bạn có thể can thiệp vào chuỗi hiện có và gán lại các liên kết để xóa hoặc thêm một liên kết mới.)*

**22. Паттерн фасад. В чем идея? Когда следует применять?  
Фасад (Façade) —** это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.  
*(Façade là một mẫu thiết kế cấu trúc cung cấp một giao diện đơn giản cho một hệ thống lớp, thư viện hoặc khuôn khổ phức tạp.)*Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.  
*(Mặt tiền là một giao diện đơn giản để làm việc với một hệ thống con phức tạp chứa nhiều lớp. Mặt tiền có thể có một giao diện rút gọn thiếu 100% chức năng có thể đạt được bằng cách sử dụng trực tiếp một hệ thống con phức tạp. Nhưng nó cung cấp chính xác những tính năng mà khách hàng cần và ẩn tất cả phần còn lại.)*  
Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей  
*(Mặt tiền rất hữu ích nếu bạn đang sử dụng một thư viện phức tạp với nhiều bộ phận chuyển động, nhưng bạn chỉ cần một phần nhỏ khả năng của nó.)***Пример:**Когда вы звоните в магазин и делаете заказ по телефону, сотрудник службы поддержки является вашим фасадом ко всем службам и отделам магазина. Он предоставляет вам упрощённый интерфейс к системе создания заказа, платёжной системе и отделу доставки.  
*(Khi bạn gọi đến cửa hàng và đặt hàng qua điện thoại, nhân viên hỗ trợ khách hàng sẽ là người hỗ trợ bạn trước tất cả các dịch vụ và bộ phận của cửa hàng. Nó cung cấp cho bạn một giao diện đơn giản hóa cho hệ thống tạo đơn hàng, hệ thống thanh toán và bộ phận vận chuyển.)***Применимость**  
**Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме. *(Khi bạn cần trình bày một giao diện đơn giản hoặc rút gọn cho một hệ thống con phức tạp.)***  
Часто подсистемы усложняются по мере развития программы. Применение большинства паттернов приводит к появлению меньших классов, но в бóльшем количестве. Такую подсистему проще повторно использовать, настраивая её каждый раз под конкретные нужды, но вместе с тем, применять подсистему без настройки становится труднее. Фасад предлагает определённый вид системы по умолчанию, устраивающий большинство клиентов.

*(Các hệ thống con thường trở nên phức tạp hơn khi chương trình phát triển. Áp dụng hầu hết các mẫu dẫn đến các lớp nhỏ hơn, nhưng với số lượng lớn hơn. Việc sử dụng lại một hệ thống con như vậy sẽ dễ dàng hơn, điều chỉnh nó mỗi lần cho các nhu cầu cụ thể, nhưng đồng thời, việc sử dụng hệ thống con mà không điều chỉnh sẽ trở nên khó khăn hơn. Mặt tiền cung cấp một chế độ xem hệ thống mặc định nhất định phù hợp với hầu hết khách hàng.)*

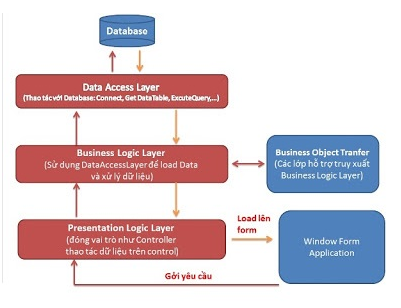
**Когда вы хотите разложить подсистему на отдельные слои.  
*(Khi bạn muốn phân rã một hệ thống con thành các lớp riêng biệt.)***Используйте фасады для определения точек входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.  
*(Sử dụng mặt tiền để xác định điểm vào cho mỗi cấp của hệ thống con. Nếu các hệ thống con phụ thuộc vào nhau, thì sự phụ thuộc có thể được đơn giản hóa bằng cách cho phép các hệ thống con chỉ trao đổi thông tin thông qua các mặt tiền.)*Например, возьмём ту же сложную систему видеоконвертации. Вы хотите разбить её на слои работы с аудио и видео. Для каждой из этих частей можно попытаться создать фасад и заставить классы аудио и видео обработки общаться друг с другом через эти фасады, а не напрямую.  
*(Ví dụ, hãy lấy cùng một hệ thống chuyển đổi video phức tạp. Bạn muốn chia nhỏ nó thành các lớp âm thanh và video. Đối với mỗi phần này, bạn có thể cố gắng tạo một mặt tiền và làm cho các lớp xử lý âm thanh và video giao tiếp với nhau thông qua các mặt này, thay vì trực tiếp.)*

**23. Паттерн посредник. В чем идея? Когда следует применять?  
Посредник (Mediator) — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник**.  
*(Dàn xếp là một mẫu thiết kế hành vi cho phép bạn giảm sự ghép nối của nhiều lớp với nhau bằng cách chuyển các mối quan hệ này vào một lớp dàn xếp duy nhất.)*Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.  
*(Mô hình Mediator buộc các đối tượng giao tiếp không trực tiếp với nhau mà thông qua một đối tượng hòa giải riêng biệt, đối tượng này biết ai cần chuyển hướng một yêu cầu cụ thể. Do đó, các thành phần của hệ thống sẽ chỉ phụ thuộc vào bên trung gian chứ không phụ thuộc vào hàng chục thành phần khác.)***Пример:  
Пилоты садящихся или улетающих самолётов не общаются напрямую с другими пилотами. Вместо этого они связываются с диспетчером, который координирует действия нескольких самолётов одновременно. Без диспетчера пилотам приходилось бы все время быть начеку и следить за всеми окружающими самолётами самостоятельно, а это приводило бы к частым катастрофам в небе**.  
*(Phi công hạ cánh hoặc khởi hành máy bay không liên lạc trực tiếp với các phi công khác. Thay vào đó, họ liên hệ với một nhân viên điều phối, người điều phối hoạt động của một số máy bay cùng một lúc. Nếu không có người điều phối, các phi công sẽ phải cảnh giác mọi lúc và tự mình theo dõi tất cả các máy bay xung quanh, và điều này sẽ dẫn đến những thảm họa thường xuyên xảy ra trên bầu trời.)*  
Важно понимать, что диспетчер не нужен во время всего полёта. Он задействован только в зоне аэропорта, когда нужно координировать взаимодействие многих самолётов.  
*(Điều quan trọng là phải hiểu rằng một bộ điều khiển là không cần thiết trong toàn bộ chuyến bay. Nó chỉ được sử dụng trong khu vực sân bay, khi cần phối hợp tác động qua lại của nhiều máy bay.)***Применимость**  
**Когда вам сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими классами**.  
*(Khi bạn khó thay đổi một số lớp do thực tế là chúng có nhiều mối quan hệ hỗn loạn với các lớp khác.)*

Посредник позволяет поместить все эти связи в один класс, после чего вам будет легче их отрефакторить, сделать более понятными и гибкими.  
*(Người trung gian cho phép bạn đặt tất cả các mối quan hệ này vào một lớp, sau đó bạn sẽ dễ dàng cấu trúc lại chúng, làm cho chúng dễ hiểu và linh hoạt hơn.)*   
**Когда вы не можете повторно использовать класс, поскольку он зависит от уймы других классов.  
*( Khi bạn không thể sử dụng lại một lớp vì nó phụ thuộc vào rất nhiều lớp khác.)***  
 После применения паттерна компоненты теряют прежние связи с другими компонентами, а всё их общение происходит косвенно, через объект-посредник.  
*(Sau khi áp dụng mẫu, các thành phần sẽ mất kết nối trước đó với các thành phần khác và tất cả giao tiếp của chúng diễn ra gián tiếp, thông qua đối tượng trung gian.)*

**24. Паттерн наблюдатель. В чем идея? Когда следует применять?  
Наблюдатель (obsever) — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.***(Observer là một mẫu thiết kế hành vi tạo ra một cơ chế đăng ký cho phép một đối tượng xem và phản hồi các sự kiện xảy ra trong các đối tượng khác.)  
(Định nghĩa mối phụ thuộc một - nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động. Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.)***Пример:  
После того как вы оформили подписку на газету или журнал, вам больше не нужно ездить в супермаркет и проверять, не вышел ли очередной номер. Вместо этого издательство будет присылать новые номера по почте прямо к вам домой сразу после их выхода**.  
*(Khi bạn đã đăng ký một tờ báo hoặc tạp chí, bạn không cần phải đến siêu thị và kiểm tra xem số tiếp theo đã ra mắt hay chưa. Thay vào đó, nhà xuất bản sẽ gửi các số báo mới qua đường bưu điện trực tiếp đến nhà bạn ngay sau khi chúng được phát hành.)***Издательство ведёт список подписчиков и знает, кому какой журнал высылать. Вы можете в любой момент отказаться от подписки, и журнал перестанет вам приходить.**   
*(Nhà xuất bản duy trì một danh sách những người đăng ký và biết sẽ gửi tạp chí nào cho ai. Bạn có thể hủy đăng ký bất cứ lúc nào và tạp chí sẽ ngừng nhận bạn.)***Применимость**  
**Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.   
*(Sau khi thay đổi trạng thái của một đối tượng, bạn cần làm điều gì đó ở những đối tượng khác, nhưng bạn không biết trước đối tượng nào sẽ phản ứng.)***  
 Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам. (*Sự cố được mô tả có thể phát sinh khi phát triển thư viện giao diện người dùng khi bạn cần kích hoạt các lớp của bên thứ ba để phản hồi các lần nhấp vào nút.)*  
Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.  
*(Mẫu Observer cho phép bất kỳ đối tượng nào có giao diện người đăng ký đăng ký nhận thông báo về các sự kiện xảy ra trong đối tượng nhà xuất bản.)*

**25. Какие слои присутствуют в классической 3х уровневой архитектуре. Дайте краткое описание и назначение каждого слоя.  
+ Слой клиента (Presentation tier): Самый верхний уровень приложение с интерфейсом пользователя. Главная функция интерфейса представление задач и результатов, понятных пользователю.***(Presentation tier: bao gồm các thành phần phần xử lý giao diện Graphic User Interface (GUI). Được dùng để giao tiếp với người dùng, nhiệm vụ chính là hiển thị dữ liệu và nhận dữ liệu từ người dùng.)*  
+ **Слой логики (Business tier): Этот слой координирует программу, обрабатывает логические решения и вычисления, выполняет расчет. Он также перемещается и обрабатывает данные между двумя окружающими слоями.** *(Lớp logic: Lớp này điều phối chương trình, xử lý các quyết định và tính toán logic và thực hiện các phép tính. Nó cũng di chuyển và xử lý dữ liệu giữa hai lớp xung quanh.) (Business tier: gồm các thành phần Business Logic Layer (BLL), Data Access Layer (DAL) và Data Tranfer Object (DTO). Được dùng để cung cấp các chức năng của phần mềm)*  
+ **Слой данных (Data tier)**: **Здесь хранится информация и извлекается из базы данных и файловой системы. Информация отправляется в логический слой для обработки и в конечном счете возвращается ползователю.***(Lớp dữ liệu: Đây là nơi thông tin được lưu trữ và truy xuất từ ​​cơ sở dữ liệu và hệ thống tệp. Thông tin được gửi đến lớp logic để xử lý và cuối cùng được trả lại cho người dùng.) (Data tier: lưu trữ dữ liệu, là các hệ quản trị CSDL như MS SQL Server, Oracle, SQLite, MS Access, XML files, text files,…Dùng để lưu trữ dữ liệu, cho phép lớp Business Logic có thể tìm kiếm, trích xuất, cập nhật…)*



**26. Какие обязанности берет на себя слой DAL? Какие контракты определяются в этом слое?**

**Data Access Layer (DAL) - это слой компьютерной программы, который предоставляет упрощенный доступ к данным, хранимым в постоянном хранилище какого-либо типа, таком как реляционная база данных.***(Lớp truy cập dữ liệu (DAL) là một lớp của chương trình máy tính cung cấp khả năng truy cập dễ dàng vào dữ liệu được lưu trữ trong một số kiểu lưu trữ liên tục, chẳng hạn như cơ sở dữ liệu quan hệ.)* **27.** **Какие обязанности берет на себя слой BLL? Какие контракты определяются в этом слое?**

**BLL (Bussines logic layer) - Это место обработки источника данных с Presentation Layer перед его передачей на Data Access Layer и сохранением в системе управления базой данных. Это также место для проверки ограничений, целостности и достоверности данных, выполнения вычислений и обработки бизнес-требований перед возвратом результатов на Presentation Layer.***(Đây là nơi đáp ứng các yêu cầu thao tác dữ liệu của GUI layer, xử lý chính nguồn dữ liệu từ Presentation Layer trước khi truyền xuống Data Access Layer và lưu xuống hệ quản trị CSDL.*

*Đây còn là nơi kiểm tra các ràng buộc, tính toàn vẹn và hợp lệ dữ liệu, thực hiện tính toán và xử lý các yêu cầu nghiệp vụ, trước khi trả kết quả về Presentation Layer.)*

**Lớp này gồm 4 thành phần:  
Service Interface**

**Bussiness Workflows**

**Bussiness Components**

**Bussiness Entities**

**28. Какие обязанности берет на себя слой Presentation?**

**Presentation Layer - У этого слой есть основная задача общаться с пользователем и выполнять такие задачи, как ввод, отображение данных, проверка правильности данных перед вызовом слоя Business Logic Layer (BLL).***(Lớp này có nhiệm vụ chính là giao tiếp với người dùng và thực hiện các công việc như nhập liệu, hiển thị dữ liệu, kiểm tra tính đúng đắn dữ liệu trước khi gọi lớp Business Logic Layer (BLL).)*