**Вопросы для подготовки к экзамену**

**Перечень вопросов для оценки усвоенных знаний**

1. Что такое паттерны проектирования?

Шаблон проектирования (паттерн, от англ. design pattern) — **повторяемая архитектурная конструкция в сфере проектирования программного обеспечения, предлагающая решение проблемы проектирования в рамках некоторого часто возникающего контекста**.

1. Какие основные категории паттернов проектирования существуют?

порождающие, структурные и поведенческие.

1. Какие паттерны проектирования относятся к созданию объектов?

Как понятно из названия, данные паттерны в основном применяются для создания новых объектов. К ним относится такие паттерны, как Конструктор, Фабрика, Прототип и Синглетон.

1. Какие паттерны проектирования относятся к структурированию классов?

Эти паттерны связаны с классами и композицией объектов. Они помогают структурировать или реструктуризировать объекты или их части без нарушения работы всей системы.

1. Какие паттерны проектирования относятся к управлению поведением объектов?

Этот вид паттернов проектирования направлен на то, что бы улучшить коммуникацию между различными объектами. К паттернам проектирования поведения относятся Командир, Итератор, Медиатор, Обзервер, Паттерн Состояния, Стратегия и Шаблон.  
  
В данной статье мы разбираем только паттерны проектирования, связанные с созданием объектов — потому что они немного проще и лучше подходят новичкам, чтобы начать создавать свои to-do приложения, текстовые RPG в консоли, простые игры на Canvas, и т.д.

1. Какие паттерны проектирования относятся к взаимодействию объектов?

Поведенческие паттерны нужны для безопасного взаимодействия между объектами программы.

1. Какие роли и задачи могут быть назначены в команде, работающей с паттернами проектирования?

* Аналитик / Архитектор: Этот человек отвечает за анализ требований и определение наиболее подходящего паттерна проектирования для решения конкретной проблемы или задачи. Он также может помочь в выборе инструментов и технологий, которые будут использоваться в проекте.
* Разработчик / Инженер: Разработчик реализует паттерн проектирования, используя выбранные инструменты и технологии. Он пишет код, выполняет тестирование и отладку, а также обеспечивает интеграцию с другими частями системы.
* Тестировщик / QA инженер: Тестировщик проводит тесты для проверки корректности работы паттерна и выявления возможных ошибок. Он также должен убедиться, что паттерн соответствует требованиям и стандартам качества.
* Менеджер проекта: Менеджер проекта отвечает за планирование, организацию и контроль над выполнением задач в рамках проекта. Он должен следить за сроками, бюджетом и качеством работы команды.
* Консультант / Эксперт: Консультант может быть приглашен для помощи в выборе паттерна, его адаптации к конкретным требованиям и предоставления рекомендаций по использованию различных инструментов и технологий.
* Документатор: Документатор отвечает за создание документации по паттернам проектирования и их использованию в проекте. Это может включать в себя описание паттернов, примеры кода, инструкции по настройке и т.д.

1. Что такое событийно-управляемое программирование (event-drivenprogramming)?

Событийно-ориентированное **программирование** (**event**-**driven** **programming**) - парадигма **программирования**, в которой выполнение программы определяется событиями. **Event**-**driven** **programming** (EDP) ещё переводят как **программирование**, управляемое событиями. На фронтенде событиями будут являться, например: действия пользователя - клик мышкой по кнопке, отправка данных через форму, нажатие пальцем по сенсорному экрану и т.д. На бэкенде событиями будут являться, например: установление TCP-соединения между клиентом и сервером, обращение к базе данных, конец чтения файла, и т.д.

1. Какие компоненты могут генерировать события в программе?

Компоненты, которые могут генерировать события, включают в себя:

– Элементы пользовательского интерфейса (кнопки, текстовые поля, списки и т. д.)  
– Аппаратные устройства (например, мышь, клавиатура, камера, микрофон)  
– Системные события (например, изменение настроек системы, обновление программного обеспечения)  
– Внешние события (например, изменения на сервере, новые сообщения в социальных сетях)  
– Внутренние процессы (например, завершение загрузки файла, обновление данных в базе данных)

1. Какие обработчики событий могут быть присоединены к компонентам в программе?

Обработчики событий - это функции или методы, которые выполняются при возникновении определенного события. Они могут быть присоединены к различным компонентам в программе, таким как кнопки, текстовые поля, элементы пользовательского интерфейса и т.д.

1. Как можно отличить событие от обычного метода вызова в программе?

События в программировании вызываются в ответ на определенные действия пользователя, системы или внешнего источника, в то время как обычные методы вызова определяются и вызываются самим разработчиком.

Событие — это именованный делегат, при вызове которого, будут запущены все подписавшиеся на момент вызова события методы заданной сигнатуры.

Тем,что при обработке события вы заранее не знаете когда оно произойдет,если произойдет.

1. Какие преимущества имеет событийно-управляемое программирование по сравнению с процедурным подходом?

Расширяемость и обслуживание. Вместо того, чтобы возвращаться к методу и добавлять к нему каждый раз, когда вы хотите добавить нового "подписчика" в вашем примере без EDP, вы просто добавите метод, который хотите вызвать, в его список подписчиков.

1. Какие особенности разработки пользовательского интерфейса связаны с применением событийно-управляемого подхода?

* События: Разработка пользовательского интерфейса на основе событийно-управляемого подхода означает использование событий, которые происходят в результате действий пользователя, таких как нажатия на кнопки, перемещения мыши или ввода текста. Эти события могут вызывать определенные действия или изменения в интерфейсе.
* Обработчики событий: Обработчиками событий являются функции или методы, которые вызываются при возникновении определенного события. Например, при нажатии на кнопку может вызываться обработчик события, который выполняет определенную задачу, такую как открытие окна или обновление данных.
* Делегирование событий: Делегирование событий позволяет обрабатывать несколько событий одним обработчиком, что может упростить код и улучшить его читаемость. Например, можно использовать один обработчик для обработки нажатия на несколько кнопок, вместо создания отдельного обработчика для каждой кнопки.
* Отложенные вызовы: Отложенные вызовы используются для выполнения действий после того, как пользователь завершил определенное действие. Например, после того как пользователь ввел текст в поле, можно выполнить проверку на правильность введенных данных.
* Асинхронные вызовы: Асинхронные вызовы позволяют выполнять действия параллельно с основным потоком выполнения программы. Это может быть полезно для выполнения длительных операций, таких как загрузка данных из сети, без блокировки интерфейса.

1. Какие типы событий могут возникать в web-приложениях?

события мыши (Mouse Events), события клавиатуры (Keyboard Events), события объектов и фреймов (Frame/Object Events), события формы и элементов управления (Form Events), события перетаскивания (Drag Events), события анимации (Animation Events), события буфера обмена (Clipboard Events), события мультимедиа (Media Events), события трансформации (Transition Events), события, посылаемые сервером (Server-Sent Events), события касания (Touch Events), события печати (Print Events), разные события (Misc Events).

1. Как можно организовать обработку нескольких событий одновременно в программе?

Это можно сделать в представлении "События" окна свойств в C# или с помощью ключевого слова Handles и раскрывающихся списков **Имя класса** и **Имя метода** в редакторе кода Visual Basic.

Private Sub Button1\_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click, Button2.Click

' Add event-handler code here.

End Sub

1. Принципединственнойобязанности (Single Responsibility Principle)

* **Принцип единственной обязанности** (Single Responsibility Principle) можно сформулировать так:
* **Каждый компонент должен иметь одну и только одну причину для изменения.**.
* В C# в качестве компонента может выступать класс, структура, метод. А под обязанностью здесь понимается набор действий, которые выполняют единую задачу. То есть суть принципа заключается в том, что класс/структура/метод должны выполнять одну единственную задачу. Весь функционал компонента должен быть целостным, обладать высокой связностью (high cohesion).

1. Принцип открытости/закрытости (Open/ClosedPrinciple)

**Принцип открытости/закрытости** (Open/Closed Principle) можно сформулировать так:

**Сущности программы должны быть открыты для расширения, но закрыты для изменения**.

Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.

1. Принцип подстановки Лисков (LiskovSubstitutionPrinciple)

**Принцип подстановки Лисков** (Liskov Substitution Principle) представляет собой некоторое руководство по созданию иерархий наследования. Изначальное определение данного принципа, которое было дано Барбарой Лисков в 1988 году, выглядело следующим образом:

В общем случае данный принцип можно сформулировать так:

**Должна быть возможность вместо базового типа подставить любой его подтип**.

1. Принципразделенияинтерфейсов (Interface Segregation Principle)

**Принцип разделения интерфейсов** (Interface Segregation Principle) относится к тем случаям, когда классы имеют "жирный интерфейс", то есть слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы. Таким образом, интерфейс получатся слишком избыточен или "жирным".

Принцип разделения интерфейсов можно сформулировать так:

**Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.**

1. Принципинверсиизависимостей (Dependency Inversion Principle)

**Принцип инверсии зависимостей** (Dependency Inversion Principle) служит для создания слабосвязанных сущностей, которые легко тестировать, модифицировать и обновлять. Этот принцип можно сформулировать следующим образом:

**Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.**

**Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.**

1. Отношения между классами и объектами(наследование, реализация, ассоциация, композиция, агрегация)

**Наследование**

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением. Наследование определяет отношение **IS A**, то есть "является". Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class User  {  public int Id { get; set; }  public string Name { get; set; }  }  class Manager : User  {  public string Company{ get; set; }  } |

В данном случае используется наследование, а объекты класса Manager также **являются** и объектами класса User.

С помощью диаграмм UML отношение между классами выражается в незакрашенной стрелочке от класса-наследника к классу-родителю:

**Реализация**

Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется интерфейс IMovable с методом Move, который реализуется в классе Car:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public interface IMovable  {  void Move();  }  public class Car : IMovable  {  public void Move()  {  Console.WriteLine("Машина едет");  }  } |

С помощью диаграмм UML отношение реализации также выражается в незакрашенной стрелочке от класса к интерфейсу, только линия теперь пунктирная:

**Ассоциация**

Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Например, игрок играет в определенной команде:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Team  {  }  class Player  {  public Team Team { get; set; }  } |

Класс Player связан отношением ассоциации с класом Team. На схемах UML ассоциация обозначается в виде обычно стрелки:

Нередко при отношении ассоциации указывается кратность связей. В данном случае единица у Team и звездочка у Player на диаграмме отражает связь 1 ко многим. То есть одна команда будет соответствовать многим игрокам.

Агрегация и композиция являются частными случаями ассоциации.

**Композиция**

Композиция определяет отношение **HAS A**, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ElectricEngine  { }  public class Car  {  ElectricEngine engine;  public Car()  {  engine = new ElectricEngine();  }  } |

При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя - зависимой.

На диаграммах UML отношение композиции проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит, объект второй сущности, располагается закрашенный ромбик:

**Агрегация**

От композиции следует отличать агрегацию. Она также предполагает отношение **HAS A**, но реализуется она иначе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public abstract class Engine  { }  public class Car  {  Engine engine;  public Car(Engine eng)  {  engine = eng;  }  } |

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

Отношение агрегации на диаграммах UML отображается также, как и отношение композиции, только теперь ромбик будет незакрашенным:

При проектировании отношений между классами надо учитывать некоторые общие рекомендации. В частности, вместо наследования следует предпочитать композицию. При наследовании весь функционал класса-наследника жестко определен на этапе компиляции. И во время выполнения программы мы не можем его динамически переопределить. А класс-наследник не всегда может переопределить код, который определен в родительском классе. Композиция же позволяет динамически определять поведение объекта во время выполнения, и поэтому является более гибкой.

Наследование - это отношение между классами, когда один класс является потомком другого класса (родительский класс). В этом случае потомок наследует все атрибуты и методы родительского класса. Наследование может быть одиночным или множественным.

Реализация - это отношение между классом и интерфейсом. Класс может реализовать один или несколько интерфейсов, чтобы добавить функциональность, которую он предоставляет.

Ассоциация - это отношение между двумя объектами, которое показывает, что они связаны друг с другом. Ассоциация может быть однонаправленной или двунаправленной.

Композиция - это отношение, при котором один объект (композиция) состоит из других объектов (компонентов). Композиция всегда двунаправленная.

Агрегация - это разновидность композиции, при которой один объект (агрегат) содержит другие объекты (части), но не является их частью. Агрегация может быть однонаправленной или двунаправленной.

**Перечень заданий для оценки усвоенных умений**

1. Напишите программу на языке программирования, которая демонстрирует использование шаблона проектирования "Одиночка" (Singleton).
2. Реализуйте фабричный метод (Factory Method) для создания объектов разных классов, объявленных в интерфейсе.
3. Напишите программу, используя шаблон проектирования "Адаптер" (Adapter), чтобы адаптировать интерфейс одного класса к другому.
4. Разработайте программу с использованием шаблона проектирования "Стратегия" (Strategy), которая демонстрирует возможность изменения поведения приложения в зависимости от выбранной стратегии.
5. Напишите программу, используя шаблон «Фасад», который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней.
6. Напишите программу на языке программирования, которая использует события для реагирования на действия пользователя в окне приложения (например, щелчки мыши или нажатия клавиш).
7. Класс WeatherStation: генерирует температуру воздуха, имеет события на слишком высокую температуру, метод для ввода порога высокой температуры.
8. КлассCurrencyRate: генерирует курс валют, имеет события на слишком большое повышение или понижение курса, методы для ввода условия в процентах (например событие на повышение от +5%, понижение от -3%)
9. Класс Authentication: имеет секретный пароль и метод для его ввода, вызывает событие при вводе неверного пароля более 3-х раз.
10. Класс FileTracker: отслеживает изменения в файле, имеет события на любые изменения содержимого внутри файла, метод для ввода пути к отслеживаемому файлу.