Изпитна тема № 7: Разработка на софтуер

1. **Дефиниране на трислойни модели. Представят се графично и се различават начините на изпълнението им. – 18т.**

- *Трислоен модел* – известен още като MVC модел. Разделя приложението на слоеве [презентационен слой (View), слой за услуги (Controller) и слой за данни (Model)] като всеки слой има строго определена задача.
  - *Презентационен слой (View)* – отговаря за въвеждане на данни от потребителите и визуализиране на данни към тях. Може да бъде GUI, уеб приложение, мобилно приложение и др.
  - *Слой за услуги (Controller)* – съдържа логиката на програмата. Приема потребителски команди, след което изпраща команди към слоя за данни за актуализиране на данните и накрая изпраща инструкции към презентационния слой за актуализиране на интерфейса.
  - *Слой за данни (Model)* – отговаря за връзката между базата данни и приложението. Служи за съхранение на данните и изпълняване на заявки от слоя за услуги върху тях.

Илюстрация на работа на трислойния модел

2. **Обяснява и модифицира готов примерен код.- 10т.**
3. **Описва и различава видовете тествания. – 6т.**

- *Компонентно тестване (unit testing)* – представлява код, който се пише от разработчика с цел да се провери и гарантира правилното поведение на дадена секция от код, която може да бъде самостоятелно тествана наричана компонент. Разделя се на два вида: ръчно и автоматично.
  - *Ръчно тестване* – това е когато разработчикът собственоръчно тества приложението. Този вид тестване е по-малко ефективен, неструктуриран е, тестовете не се повтарят и не покриват целия ни код.
  - *Автоматично тестване* – това е когато сме написали отделна програма/метод, която тества нашето приложение.
- *Интеграционен тест* – тестването се извършва от тестери и се тества интеграцията между софтуерните модули. При този вид тестове, отделни единици от програмата

се комбинират като група. За подпомагане на интеграционното тестване се използват тестови модули и тестови драйвери. Тестът за интеграция се извършва по два начина: *от долу нагоре* и *от горе надолу*.

- *Регресионен тест* – тества се скорошно направена в програмата промяна, за да се потвърди, че не е повлияла неприятно на съществуващите вече функции. Представлява пълна или частична селекция от вече изпълнени тестови случаи, които се изпълняват повторно, за да се гарантира, че съществуващите функционалности работят добре и няма някакви странични ефекти.
- *Системно тестване* – тества се, за да се подсигури, че ще работи добре и на други операционни системи. Обхванат е от така наречените „black box" тестове. При тях се съсредоточаваме само върху входните и изходните данни, без да ни интересува какво се случва с тях.

- *Разлики между различните видове тестване*
  - Компонентно: тестове на отделни компоненти;
  - Интеграционен: тестването се извършва от тестери и тества интеграция между софтуерни модули.
  - Регресионен: да се потвърди, че скорошна промяна на програмата или код не е повлияла неблагоприятно на съществуващите функции.
  - Системен: отнася се до цялостен подход за тестване на съответствието към софтуерното тестване, при което е показано, че тестваният модул отговаря изчерпателно на спецификация, до допусканията за тестване.

4. **Сравнява Code First и Database First подхода и прави изводи за разликите между тях. 12т.**

- *Code First* – когато чрез код създаваме база от данни заедно с нейните таблици. Това става като програмистът създаде entity класовете с необходимите им свойства. След това entity framework-а създава бази от данни и таблици, по информацията, която е дадена в класовете. Тук базата данни ще се създаде, когато се пусне програмата.

- *Database First* – когато имаме вече създадена база данни с таблици и работим по нея.

Разлики между Code First и Data First	
Code First	Data First
Представлява подход, наличен в entity framework, който позволява на разработчика да създава бази от данни, използвайки entity класове за изграждане на MVC приложения.	Представлява подход, наличен в entity framework, който позволява на програмиста първо да създаде база данни entity дата модела при разработване на MVC приложения.
За да създаде базата данни, разработчикът трябва да напише класове със свойства.	Програмистът може да използва програма с графичен потребителски интерфейс, за да създаде базата данни.
Подходящ за малки приложения, които не изискват интензивна работа с данни.	Подходящ за големи програми, които изискват интензивна работа с данни.

5. Разработва модела на база от данни чрез Code First. 6т.

```
using System;
using System.Collections.Generic;
using System.Data.Entity;

public class Book
{
   public int BookId { get; set; }
   public string Title { get; set; }
   public string Author { get; set; }
}

public class BookContext : DbContext
{
   public DbSet<Book> Books { get; set; }
}

class Program
{
   static void Main(string[] args)
   {
      using (var db = new BookContext())
      {
         db.Books.Add(new Book { Title = "The Great Gatsby", Author = "F. Scott Fitzgerald" });
         db.SaveChanges();

         var books = db.Books;
         foreach (var book in books)
         {
            Console.WriteLine("Title: {0}, Author: {1}", book.Title, book.Author);
         }
      }
   }
}
```

В този пример класът Book е дефиниран като модел, а класът BookContext е производен на DbContext. Свойството DbSet<Book> представлява таблицата books в базата данни. Методът Main добавя нова книга в базата данни, записва промените и след това извлича всички книги от базата данни и показва тяхното заглавие и автор.

6. Демонстрира познанията си за слоя с базите данни и дифе-ренцира SQL заявките от ORM подхода. – 20т.

- Един от начините за взаимодействие с база данни от приложението е чрез SQL, който е език за програмиране специално разработен за управление и манипулиране на данни, съхранявани в релационни бази данни.

  SQL заявките се използват за създаване, модифициране и извличане на данни от база данни. Те са написани в специфичен синтаксис, който базата данни може да разбере и изпълни. Например следното е проста SELECT заявка в SQL, която извлича всички редове от таблица, наречена "customers":

  SELECT* FROM customers;

  Друг начин за взаимодействие между база данни и приложението е чрез инструмента Object-Relational Mapping (ORM). Инструментите за ORM осигуряват начин за съпоставяне на обекти в приложението с таблици в базата данни, което

позволява на разработчиците да работят с данни по обектно-ориентиран начин, вместо да пишат SQL заявки.

Използването на ORM може да опрости взаимодействието с базата данни и да намали количеството на кода, който трябва да се напише, но също така може да добави допълнително ниво на сложност към приложението. Инструментите ORM обикновено имат и наднормени разходи за производителност, тъй като трябва да превеждат между обектно-ориентираните и релационните модели на данни.

В езика C# един пример за ORM инструмент е Entity Framework, който предоставя начин за дефиниране на модели и връзки с помощта на класове на C# и след това автоматично генерира необходимите SQL заявки за създаване и манипулиране на съответните таблици в базата данни.

Ето един пример за използване на Entity Framework за извличане на данни от база данни в C#:

```
using (var context = new MyDbContext())
{
   var customers = context.Customers.ToList();
}
```

Този код създава екземпляр на класа context (който представлява връзката с базата данни) и го използва за извличане на списък с клиенти от базата данни. Контекстният клас се генерира от Entity Framework въз основа на моделите, дефинирани в приложението, и обработва създаването на необходимите SQL заявки зад кулисите.

7. Посочва принципите за правилно разделяне на компонентите на приложението по слоеве. Избира кои SOLID принципи да използва в конкретни ситуации. – 10т.

- В софтуерното инженерство SOLID е акроним за пет принципа на проектиране, предназначени да направят обектно-ориентираните проекти по-разбираеми, гъвкави и по-лесни за поддържане. Принципите са:
  - *S*ingle-responsibility – всеки клас трябва да има само една отговорност
  - *O*pen-closed – гласи, че софтуерните единици трябва да бъдат отворени за разширяване, но затворени за модифициране.
  - *L*iskov substitution – гласи, че функциите, които използват поинтъри или препратки към базови класове, трябва да могат да използват обекти от производни класове, без да знаят това.
  - *I*nterface segregation – гласи, че клиентите не трябва да бъдат принуждавани да зависят от интерфейси, които не използват
  - *D*ependency inversion – зависи от абстракции, а не от конкретизации.

  Използването на тези принципи не е задължително, но те могат да ви дадат добър ориентир за изграждане на поддържана и гъвкава база на кода.

  Например може да използвате:

  - Single-responsibility принципа, за да създадете малки, фокусирани класове, които изпълняват една единствена, добре дефинирана задача;
  - Open-Closed принципа, за да използвате абстракции и интерфейси, за да добавяте нова функционалност, без да променяте съществуващия код;

- Liskov substitution, за да гарантирате, че подкласовете са съвместими с интерфейсите и contract-тите на техните родителски класове.

Нека пак да отбележим, че принципите SOLID са насоки, а не строги правила и вие сами решавате кога и как да ги прилагате в кода си.

8. Описва и дава примери за графични компоненти, свойства и събития.- 12т.

- В софтуерното инженерство графичните компоненти са визуални елементи, които могат да се показват в потребителския интерфейс (UI). Примерите за графични компоненти в C#:
  - бутони
  - етикети
  - текстови полета
  - полета за отметка
  - падащи списъци

Тези компоненти могат да бъдат добавени към потребителския интерфейс с помощта на дизайнер на графичен потребителски интерфейс (GUI) или чрез написване на код, който създава и конфигурира компонентите програмно.

Свойствата (properties) задават начина, по който ще изглежда даден елемент. Примери за свойства са текстът, който се показва на бутон, цветът на фона на текстово поле и шрифтът, използван в етикет. Свойствата могат да се задават в дизайнера на графичния потребителски интерфейс или в кода. Например следният код задава текста, показван върху бутон:

```
myButton.Text = "Click Me";
```

Събитията (events) са действия, които възникват в отговор на действия на потребителя или на системата. Примери за събития са щракване върху бутон, въвеждане на текст в текстово поле и избор на елемент от падащ списък. Събитията могат да се обработват в кода чрез писане на обработчици на събития (event handlers), които са методи, изпълнявани при настъпване на събитие. Например следният код дефинира обработчик на събитие за когато бутон бъде натиснат:

```
myButton.Click += MyButton_Click;

private void MyButton_Click(object sender, EventArgs e)
{
      // Код, който ще бъде изпълнен при натискането на бутона
}
```

В този пример методът MyButton_Click се изпълнява всеки път, когато потребителят щракне върху бутона myButton. Параметърът “sender” е препратка към бутона, върху който е щракнато, а параметърът “e” е обект, който съдържа информация за събитието.

9. Демонстрира начина за свързване на данни към графичен компонент (Data Binding). - 6т.

```
public class Person // създаваме клас Person
{
    public string FirstName { get; set; } // създаваме свойство FirstName
    public string LastName { get; set; } // създаваме свойство LastName
}

// създава се инстанция на класа Person и се присвоява на частното поле
_person.
// Инстанцията е инициализирана с първото име "John" и фамилията "Doe"
private Person _person = new Person { FirstName = "John", LastName = "Doe" };

private void Form1_Load(object sender, EventArgs e)
{
    // Създава се нова връзка между текста, който ще се покаже в textBox1,
    // така че да се изведат данните от _person
    var binding = new Binding("Text", _person, "FirstName");
    textBox1.DataBindings.Add(binding);
}
```