

**ПРОФЕСИОНАЛНА ГИМНАЗИЯ**

**ПО ЕЛЕКТРОТЕХНИКА И ЕЛЕКТРОНИКА “АПОСТОЛ АРНАУДОВ”**

гр. Русе, ул. “Потсдам” № 3; п.к. 7005, тел. 082/84-60-96; e-mail: [info-1806301@edu.mon.bg](mailto:info-1806301@edu.mon.bg)

Д И П Л О М Е Н П Р О Е К Т

**ЗА ПРИДОБИВАНЕ НА**

**ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ**

**по професия код 481020 „Системен програмист“**

**специалност код 4810201 „Системно програмиране“**

**ТЕМА:**

**SOLID принципите при разработка на софтуер.**

**Ученик**: Юлиян Велиславов Наумов

**Ръководител-консултант:** Милена Дамесова-Христова

гр. Русе

2024

**СЪДЪРЖАНИЕ**

[УВОД 3](#_Toc164980420)

[ГЛАВА ПЪРВА 4](#_Toc164980421)

[1. Основи на езика C# и етапи при разработка на софтуер 4](#_Toc164980422)

[1.1 Основни възможности на програмния език C# 4](#_Toc164980423)

[1.2 Типове данни и колекции 6](#_Toc164980424)

[1.2.1 Прости типове 6](#_Toc164980425)

[1.2.2 Целочислени типове 6](#_Toc164980426)

[1.2.3 Оператори 10](#_Toc164980427)

[1.2.4 Управляващи инструкции 10](#_Toc164980428)

[1.3 Етапи при разработката на софтуер 14](#_Toc164980429)

[1.3.1 Събиране и анализ на изискванията 14](#_Toc164980430)

[1.3.2 Дизайн 14](#_Toc164980431)

[1.3.3 Кодиране 15](#_Toc164980432)

[1.3.4 Тестване 15](#_Toc164980433)

[1.3.5 Внедряване 15](#_Toc164980434)

[1.3.6 Поддръжка 15](#_Toc164980435)

[ГЛАВА ВТОРА 17](#_Toc164980436)

[2. SOLID принципите при разработка на софтуер. 17](#_Toc164980437)

[2.1 SOLID Принципи 17](#_Toc164980438)

[2.1.1 Single Responsibility Principle 17](#_Toc164980439)

[2.1.2 Open-Closed Principle 18](#_Toc164980440)

[2.1.3 Liskov Substitution Principle 18](#_Toc164980441)

[2.1.4 Interface Segregation Principle 19](#_Toc164980442)

[2.1.5 Dependency Inversion Principle 19](#_Toc164980443)

[ГЛАВА ТРЕТА 21](#_Toc164980444)

[3. ПРИЛОЖЕНИЕ ЗА УПРАВЛЕНИЕ НА ХОТЕЛСКИ РЕЗЕРВАЦИИ 21](#_Toc164980445)

[3.1 Среда за разработка VS 2022 21](#_Toc164980446)

[3.2 Архитектура на VS 2022 21](#_Toc164980447)

[3.3 Windows Forms 21](#_Toc164980448)

[3.3.1 Архитектура на WinForms 22](#_Toc164980449)

[3.3.2 Характеристики на WinForms 23](#_Toc164980450)

[3.4 Приложение за управлене на хотел 23](#_Toc164980451)

[3.4.1 Лого/Емблема на приложението 23](#_Toc164980452)

[3.4.2 Начална страница (Form1) 24](#_Toc164980453)

[3.4.3 Направи резервация (MakeReservationForm) 24](#_Toc164980454)

[3.4.4 Всички стаи (AllRoomsForm) 25](#_Toc164980455)

[3.4.5 Търсене на стая (FindingRoomForm) 26](#_Toc164980456)

[3.4.6 Изчисти стая (CleanRoomForm) 27](#_Toc164980457)

[3.4.7 Освободи (FreeingRoomForm) 27](#_Toc164980458)

[3.4.8 Изготвил проекта (CreditsForm) 28](#_Toc164980459)

[ЗАКЛЮЧЕНИЕ 29](#_Toc164980460)

[ИЗПОЛЗВАНА ЛИТЕРАТУРА 30](#_Toc164980461)

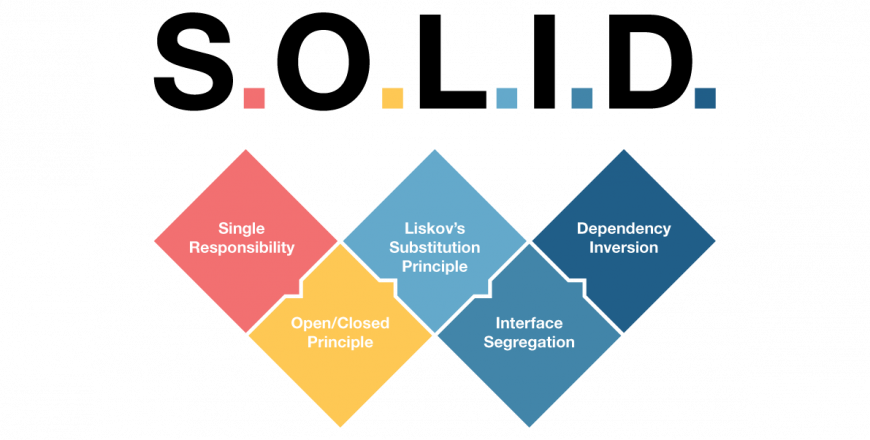
[ПРИЛОЖЕНИЯ 31](#_Toc164980462)

# УВОД

В настоящото технологично общество, където данните са от съществено значение, фокусът върху управлението на информацията в софтуерните приложения е от ключово значение. В този контекст, изучаваме и прилагаме принципите на SOLID в програмирането с помощта на езика C#.

Целта на нашето изследване е да разгледаме и използваме принципите на SOLID за разработване на софтуерни приложения, които се отличават с ясна структура, лесна поддръжка и гъвкавост. Проучваме методи и техники, които гарантират постигането на единен интерфейс, разделение на отговорностите и минимизиране на зависимостите между различните компоненти на приложението.

Дипломната работа си поставя за цел да представи най-добрите практики за разработване на софтуерни приложения с използване на C#, които следват принципите на SOLID. Целта ни е да демонстрираме какво е необходимо за постигане на ефективно и сигурно управление на данните в приложенията, като се спазват съвременни стандарти и норми за сигурност и производителност, базирани на SOLID принципите в програмирането с C#.



# ГЛАВА ПЪРВА

## Основи на езика C# и етапи при разработка на софтуер

### Основни възможности на програмния език C#

C# (C Sharp, произнася се Си Шарп) е обектно ориентиран език за програмиране, разработен от Microsoft като част от софтуерната платформа .NET. Стремежът още при създаването на C# езика е бил да се създаде прост, модерен, обектно-ориентиран език с общо предназначение. Основа за C# са C++, Java и донякъде езици като Delphi, VB.NET и C. Проектиран е да балансира мощност (C++) с възможност за бързо разработване (Visual Basic и Java). Те представляват съвкупност от дефиниции на класове, които съдържат в себе си методи, а в методите е разположена програмната логика – инструкциите, които компютърът изпълнява. Програмите на C# представляват един или няколко файла с разширение .cs, в които се съдържат дефиниции на класове и други типове. Тези файлове се компилират от компилатора на C# до изпълним код и в резултат се получават асемблита – файлове със същото име, но с различно разширение (.exe или .dll).

Той е известен със своята синтаксисна яснота и гъвкавост, което го прави предпочитан избор сред програмистите по целия свят.

Сред основните възможности на C# са:

* **Обектно-ориентиран подход:** C# поддържа основните концепции на обектно-ориентираното програмиране, като класове, обекти, наследяване, полиморфизъм и инкапсулация. Това позволява по-лесно управление на сложни проекти и по-ефективно използване на кода.
* Мултиплатформеност: C# е разработен да бъде използван на различни операционни системи и платформи, включително Windows, Linux и macOS. Това осигурява гъвкавост и възможност за създаване на приложения, които могат да работят на различни устройства.
* Интеграция с .NET Framework и .NET Core: C# е основен език за програмиране в рамките на .NET Framework и .NET Core, които предоставят обширен набор от библиотеки и инструменти за разработка на разнообразни приложения. Това включва уеб приложения, десктоп приложения, мобилни приложения и много други.
* Съвременни функции на програмния език: C# постоянно се развива и включва съвременни функции и концепции на програмиране, като асинхронно програмиране, LINQ (Language Integrated Query), патерн съпоставяне и др. Тези функции улесняват разработката на по-ефективен и чист код.

Богата екосистема от инструменти и ресурси: Заедно с C# и .NET идва богата екосистема от инструменти за разработка, IDE (Integrated Development Environment) като Visual Studio, както и голямо количество отворени библиотеки и ресурси за подпомагане на разработката.

A computer chip with many squares and wires

Description automatically generated with medium confidence

Фигура 1 / Nuget.org – хранилище за множество пакети и библиотеки /

Тези основни възможности на C# правят езика предпочитан избор за множество програмисти при създаването на разнообразни приложения.

### Типове данни и колекции

#### Прости типове

Простите типове, който притежава C#, имат някои общи характеристики. Първо, всички те са псевдоними на .NET системните типове. Второ, изразите с константи от прост тип се изчисляват само при компилирането, не и при стартирането. На края, простите типове могат да бъдат инициализирани с литерали.

Простите типове в C# са групирани по следният начин:

 Целочислени типове

 Тип bool

 Тип char (специален случай на целичислен тип)

 Типове с плаваща запетая

 Типът decimal

#### Целочислени типове

Съществуват девет целочислени типа в C#: sbyte, byte, short, ushort, int, uint, long, ulong и char (обсъден в отделна секция). Те имат следните характеристики:

 Типът sbyte представлява 8 битови цели числа със знак, чиито стойности са между -128 и 127.

 Типът byte представлява 8 битови цели числа без знак, чиито стойности са между 0 и 255.

 Типът short представлява 16 битови цели числа със знак, чиито стойности са между -32,768 и 32,767.

 Типът ushort представлява 16 битови цели числа без знак, чиито стойности са между 0 и 65,535.

 Типът int представлява 32 битови цели числа със знак, чиито стойности са между -2,147,483,648 и 2,147,483,647.

 Типът uint представлява 32 битови цели числа без знак, чиито стойности са между 0 и 4,294,967,295.

 Типът long представлява 64 битови цели числа със знак, чиито стойности са между-9,223,372,036,854,775,808 и 9,233,372,036,854,755,807.

 Типът ulong представлява 64 битови цели числа без знак, чиито стойности са между 0 и 18,446,744,073,709,551,615

Може би програмистите на C и VB ще бъдат малко изненадани от новите обхвати на типовете int и long. За разлика от други програмни езици, в C# типът int вече не е зависим от размера на думата за съответната система, а типът long е установен на 64 бита.

###### Типът bool

Типът bool представлява булевите променливи true и false. На булева променлива може да се присвои една от двете стойности - true или false, или пък да се присвои израз, чийто резултат се свежда отново до една от тях:

bool bTest = (100 > 90);

За разлика от C и C++, в C# стойността true вече не се представя от която и да била нулева стойност. Не съществува възможност за конвертиране на целочислени типове в булев такъв, за да се нарушава тази конвенция.

###### Типът char

Типът char представлява един Unicode символ. Unicode символът е с 16 бита дължина и може да бъде използван за представянето на повече езици по света. Може да се присвои символ към променливата от тип char по следният начин:

char chSomeChar = 'A';

В допълнение към това, може да се присвои стойност на символна променлива посредством шестнадесетична стойност с префикс x или Unicode формат с префикс u:

char chSomeChar = 'x0065';

char chSomeChar = 'u0065';

Не съществува безусловно конвертиране от тип char в някакъв друг тип данни. Това означава, че не може да се третира променлива от този тип като някакъв друг целочислен тип в C#. Това е още една подробност, с която програмистите на C трябва да свикнат. Въпреки това, може да се извърши изрично преобразуване:

char chSomeChar = (char)65;

###### Типове с плаваща запетая

Два типа за данни спадат към категорията типове с плаваща запетая: float и double. Разликата между тях е в обхвата от сойности и броя цифри след десетичната запетая:

float: Обхватът от стойности е от 1.5 по 10 на -45 степен до 3.4 по 10 на 38 степен, с точност от 7 знака след десетичната точка.

double: Обхватът от стойности е от 5.0 по 10 на -324 степен до 1.7 по 10 на 308 степен с точност от 15-16 знака след десетичната точка.

При извършването на изчисления, с които и да било от двата типа с плаваща запетая, могат да се получат следните стойности:

 Положителна или отрицателна нула

 Положителна или отрицателна безкрайност

 Не числена стойност (Not-a-Number - NaN)

 Крайно множество от не нулеви стойности

Друго правило при изчисленията на стойностите е, че ако една променлива в даден израз е от тип плаваща запетая, всички останали типове се конвертират в тип с плаваща запетая преди извършването на изчисленията.

###### Типът double

Типът decimal е много прецизен, 128 битов тип за данни, който е предназначен за употреба при финансови и парични изчисления. Може да се представят стойности, вариращи приблизително от 1.0 по 10 на -28 степен до 7.9 по 10 на 28 степен с 28 до 29 значими цифри. Важно е да се отбележи, че точността се определя от цифри, а не от десетични знаци. Операциите са точни, максимум до 28 десетични знака.

Както може да се забележи, обхватът от стойности на типа char е по-малък от този на типа double, но с по-голяма прецизност. Поради тази причина безусловното конвертиране между типовете decimal и double не е възможно - в едната посока може да възникне препълване, а в другата може да се загуби точността. Затова е необходимо изрично да се заяви конвертирането чрез използване на образец.

Когато се дефинира променлива и се присвоява стойност на нея, се използва суфикса m, за да се обозначи, че стойността е от тип decimal.

decimal decMyValue = 1.0m;

###### Типът int

Типът данни int представлява цели числа в C#. Може да се присвои стойности на променлива от тип int, като се използват цели числа без десетична запетая. При дефиниране на променлива от тип int, може да се използва оператора за присвояване (=), за да се зададе стойност.

int age = 25;

int numberOfStudents = 50;

Целочислените типове данни в C# могат да представят както положителни, така и отрицателни числа, в рамките на определен обхват. В зависимост от конкретната имлементация, обхватът на типа int обикновено е от -2,147,483,648 до 2,147,483,647.

При използването на целочислени операции, трябва да се внимава за възможността за препълване или загуба на точност при извършването на операции, особено при деление или умножение на големи числа.

#### Оператори

A screenshot of a computer

Description automatically generated

Фигура 2 / Оператори /

#### Управляващи инструкции

##### if-else case

Условните конструкции if и if-else предоставят условен тип контрол, чрез който програмата може да изпълнява различни функции в зависимост от някакво условие, което се проверява по време на изпълнение на конструкцията.

public static void IfElse

{

if (булев израз)

{

тяло тип (1) на условната конструкция;

}

else if (нов булев израз)

{

тяло тип (2) на условната конструкция;

}

else

{

тяло тип (3) на else-конструкция;

}

}

If-else конструкцията се състои от запазената дума if, булев израз, тяло на условната конструкция, следвана от запазената дума else и тяло на else конструкцията. Тялото на else конструкцията може да се състои от един или няколко оператора, които са заградени в къдрави скоби, също както и тялото на условната конструкция.

##### switch case

Конструкцията switch-case избира измежду части от програмен код на базата на стойност на зададения селектор, който представлява променлива или израз (най-често целочислен). Форматът на конструкцията за избор на вариант е следният:

public static void SwitchCase

{

switch (селектор)

{

case стойност на селектора (1): конструкция; break;

case стойност на селектора (2): конструкция; break;

default: конструкция; break;

}

}

Операторът switch сравнява резултата от селектора с всяка една стойност от изброените в тялото на switch конструкцията в case етикетите. Ако се открие съвпадение с някой case етикет, се изпълнява съответната конструкция. Ако не се открие съвпадение, се изпълнява default конструкцията.

##### while loop

While цикълът в C# е основна конструкция за повтаряне на определен блок код. Използва се, за да повтаря определени действия, докато е изпълнено определено условие. Когато програмата достигне while цикъла, тя първо проверява условието. Ако това условие е верно, блокът от код в рамките на while цикъла се изпълнява. След като блокът приключи изпълнението си, програмата отново проверява условието. Ако то остане верно, цикълът се изпълнява отново. Този процес продължава, докато условието стане невярно, като по този начин while цикълът завършва.

public static void while

{

// Докато условието е вярно цикъла ще се изпълнява.

while (условие)

{

}

}

##### for loop

За разлика от while цикъла, for цикълът в C# изисква да бъде предварително определен брой итерации. Когато програматът достигне for цикъла, тя изпълнява инициализацията, след това проверява условието. След изпълнението на блока от код, програмата изпълнява стъпката за промяна и отново проверява условието. Този процес продължава, докато условието стане невярно, като по този начин завършва for цикълът.

public static void for

{

//Инициализация, следвана от проверка на условието, изпълнение на цикъла.

for (i=0; i<10; i++)

{

}

}

##### foreach loop

Foreach цикълът в C# се използва за итериране през елементите на колекция (като масив, списък и др.) без необходимост от указване на начало, край и стъпка за промяна. Вместо това, foreach цикълът автоматично преминава през всеки елемент от колекцията и изпълнява определен блок код за всяка итерация. Този вид цикъл е особено удобен, когато искаме да извършим действие върху всеки елемент на колекцията без да се налага да се грижим за индекси или броячи.

public static void foreach

{

//Изпълнение на цикъла за всеки елемент от масив или списък.

foreach (елемент in колекция)

{

}

}

### Етапи при разработката на софтуер

Създаването на софтуер може да бъде сложна задача, която отнема много време на цял екип от софтуерни инженери и други специалисти. Затова с времето са се обособили различни методики и практики, които улесняват работата на програмистите. Общото между всички тях е, че разработката на всеки софтуерен продукт преминава през няколко етапа, а именно:

#### Събиране и анализ на изискванията

От решаващо значение е групата софтуерни експерти да събере бизнес нуждите в тази първоначална стъпка, преди да може да се измисли широк дизайн на който и да е базирана програма. Основната цел на потребителите и ръководителите на проекти в този момент е да съставят списък на специфичните функции, изисквани от всяка програма, която се разглежда.

След отговорите на тези фундаментални въпроси се изготвя общ план, върху който софтуерните инженери да се концентрират. След това се проверява истинността на данните, както и възможността за включването им. Накрая се създава документ за спецификация на изискванията, който ще служи като ръководство за следващия етап от процеса на разработка на софтуер.

#### Дизайн

Разработването на софтуер сега преминава към следващия етап. Според изискванията от етап 1, на този етап се създава проект на програмата. Системните проекти помагат при спецификацията както на хардуера, така и на системните изисквания. Също така помага при дефинирането на цялостната система на софтуерния архитект.

Изискванията за проектиране на системата се използват като вход в следващата стъпка от методологията. Тестовите стратегии се разработват от тестери по време на тази фаза, която включва списък с елементи за тестване и как да ги проверите.

#### Кодиране

След получаване на проектната документация за изгражданата програма, работата се разделя равномерно на множество части и модули. Тук започва истинският код. Производството на отлични кодове от програмистите е основният акцент на тази фаза. Това е най-отнемащата време стъпка от целия процес.

#### Тестване

Този етап е наистина важен за разработчиците. Ако нещо се обърка по време на стъпката на тестване или ако бъдат открити грешки в кодовете, процесът на кодиране ще трябва да се повтори и цикълът ще продължи, докато проектът приключи. Всички видове функционални тестове, включително фаза на тестване, тестване на единици, автоматизация на тестове, валидиране на входа и нефункционално тестване, са завършени на този етап.

#### Внедряване

Етапът на внедряване започва, след като всички грешки в кода бъдат елиминирани по време на етапа на тестване. Попълненият код впоследствие се интегрира в програмата и се разпространява или предоставя на потребителите за използване.

Тъй като решението се разпространява сред бъдещи клиенти, първата стъпка е да се проведе бета тестване, за да се потвърди, че функционира правилно в широк мащаб. Ако някакви корекции са осъществими или ако по време на внедряването бъдат открити някакви неизправности, те незабавно се уведомяват до персонала, който гласят грешките, за да функционира правилно в реално време. Окончателното разпространение започва след като всички модификации са били приложени и всички проблеми са били адресирани.

#### Поддръжка

Това е продължителен процес на комуникация между потребителите и разработчиците по отношение на възникнали проблеми или нови функционални потребности. Често се сключва абонаментен договор за поддръжка, наречена support. Екип от специалисти отговарят за своевременно откриване на проблеми или разработката на нови компоненти за софтуера.

# 

# ГЛАВА ВТОРА

## SOLID принципите при разработка на софтуер.

### SOLID Принципи

В света на компютърното програмиране SOLID е мнемоничен акроним представен от Michael Feathers за „първите пет принципа“ основани и дефинирани от Роберт Мартин – Robert C. Martin в началото на 2000-ната година, който стои зад петте основни принципа на обектно ориентираното програмиране. Когато принципите се прилагат заедно при разработването на една система, програмиста създава програма, която е лесна за поддръжка и разширение с течение на времето. Принципите на SOLID са насоки, които могат да се прилагат по време на работа на софтуера за отстраняване на т.нар. „миризми по кода“ (код който не е написан качествено) от страна на програмиста при преработване на софтуерен код с цел той да е четим и разширяем. Всичко това е част от стратегията за това, че изходен софтуерен код е гъвкав “agile”.

A screenshot of a computer

Description automatically generated

Фигура 3 / SOLID принципите нагледно /

#### Single Responsibility Principle

Single Responsibility Principle (SRP) / Принцип за единствена отговорност – Един клас трябва да има една единствена причина да бъде променен. Този принцип може да определи дали един клас изпълнява твърде много задачи. Класът трябва да прави едно единствено нещо и да го прави добре. Тогава е много по-лесно кодът да се разбере, когато се чете от човек. Вероятността този код в бъдеще да се промени е минимална, но дори и това да се наложи, най-вероятно няма да възникнат непредвидени странични ефекти от тази промяна.

Добър пример: Разделяне на функционалността на класовете, така че всеки клас има само една причина да се променя. (Приложение 1 / SRP добра практика /)

Лош пример: Клас, който извършва повече от една операция. (Приложение 2 / SRP лоша практика /)

#### Open-Closed Principle

Open-Closed Principle (OCP) / Принцип отворен/затворен – Един клас трябва да бъде отворен за разширение, но затворен за модификации. Този принцип се отнася до добавянето на нова функционалност в една програма без да се променя вече написан код или дори асембли. Причината за това е, че всеки път, когато се променя даден код, се появява и риска от възникването на бъгове в съществуваща функционалност.

Добър пример: Класовете са отворени за разширяване, но затворени за модификация. ( Приложение 3 / OCP добра практика /)

Лош пример: Клас, който се променя, когато се добавя нова функционалност. ( Приложение 4 / OCP лоша практика /)

#### Liskov Substitution Principle

Liskov Substitution Principle (LSP) / Принцип на заместване на Лисков – Всеки един клас трябва да може да бъде заменен с подклас без извикващият го код да знае за промяната. Всеки наследник (подтип) трябва лесно да заменя всичките си базови типове. Подтипът не трябва да премахва нито една от функционалностите на базовия клас, а при нужда само да ги разширява, ако по някаква причина даден клас не поддържа някоя от функционалностите на своя базов клас (родител), трябва да се помисли, дали се използва правилен вид наследяване.

Добър пример: Подтиповете могат да бъдат заместени със своите супертипове без да се променя желаното поведение на програмата. (Приложение 5 / LSP добра практика /)

Лош пример: Подклас, който променя очакваното поведение на суперкласа. (Приложение 6 / LSP лоша практика /)

#### Interface Segregation Principle

Interface Segregation Principle (ISP) / Принцип за разделяне на интерфейсите – Повече специфични интерфейси са по-добре, отколкото един общ интерфейс. Този принцип гласи, че всеки интерфейс трябва да бъде разбит на много на брой малки интерфейси. Тези интерфейси трябва да отговарят за едно-единствено нещо. Може да се види пример в \* Приложение 4 / ISP /\*

Нито един клас не трябва да бъде принуждаван да имплементира методи, които няма да ползва никога.

„Дебел“ (“fat”), голям, пълен с различни методи интерфейс води до:

* Класовете да имплементират методи, които не са им нужни.
* Увеличена свързаност(coupling) между класовете.
* Намалена гъвкавост.
* Поддръжката става по-трудна.

Добър пример: Клиентите не трябва да бъдат принуждавани да зависят от интерфейси, които не използват. (Приложение 7 / ISP добра практика /)

Лош пример: Големи интерфейси, които изискват клиентите да имплементират методи, които те не използват. (Приложение 8 / ISP лоша практика /)

#### Dependency Inversion Principle

Dependency Inversion Principle (DIP) / Принцип на обръщане на зависимостите – Кодът (класът) трябва да зависи от абстракции, а не от конкретни имплементации. Принципът на обръщане на зависимостите е специфичен начин за отвързване (отделяне; decoupling) на софтуерните модули. Когато следваме този принцип, модулите на по-високо ниво не зависят от тези на по-ниско ниво, като и двата трябва да зависят само и единствено от абстракции. В същото време абстракциите не трябва да зависят от детайлите, а детайлите трябва да зависят от абстракциите. Този принцип е въведен от Роберт Мартин. Зависимост наричаме даден клас или модул, от който зависи друг клас или модул.

Добър пример: Зависимостите на високо ниво не трябва да зависят от детайли на ниско ниво. Вместо това, и двете нива трябва да зависят от абстракции. (Приложение 9 / DIP добра практика /)

Лош пример: Високо ниво модули, които са твърде зависими от конкретни ниско ниво модули. (Приложение 9 / DIP лоша практика /)

# ГЛАВА ТРЕТА

## ПРИЛОЖЕНИЕ ЗА УПРАВЛЕНИЕ НА ХОТЕЛСКИ РЕЗЕРВАЦИИ

### Среда за разработка VS 2022

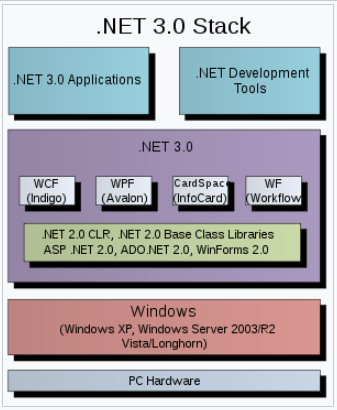
Visual Studio предоставя мощна интегрирана среда за писане на код, компилиране, изпълнение, дебъгване (както за високо така и за машинно ниво), тестване на приложения, дизайн на потребителски интерфейс (форми, диалози, уеб страници, визуални контроли и други), моделиране на данни, моделиране на класове, изпълнение на тестове, пакетиране на приложения и стотици други функции. Могат да се добавят и плъгини, които повишават функционалността на почти всяко ниво – включително добавянето на поддръжка за source-control системи (като Subversion и Visual SourceSafe), добавяне на нови инструменти като редактори и визуални дизайнери за domain-specific languages или инструменти за други аспекти (като например: Team Foundation Server, Team Explorer).

### Архитектура на VS 2022

Visual Studio е интегрирана среда за разработка, която не поддържа конкретен език или инструменти за програмиране направо, а посредством VSPackage плъгини. При инсталиране, функционалността се предоставя като услуга, като IDE-то предлага три основни услуги: SVsSolution, SVsUIShell и SVsShell. Всички инструменти и редактори се внедряват като VSPackages, като достъпът до тях става чрез COM.

### Windows Forms

Windows форми (на английски: Windows Forms) е графична (GUI) библиотека от класове в състава на Microsoft .NET Framework, която предоставя платформа за писане на клиентски приложения за настолни компютри, лаптопи и таблети. Формите са разглеждани като замяна на по-ранната и по-сложна C++ базирана библиотека Microsoft Foundation Class Library, въпреки че не предлагат съпоставима парадигма, а служат само като платформа за създаване на слоя потребителски интерфейс в многослойни решения.



Фигура 4 / Приложно-програмен интерфейс /

#### Архитектура на WinForms

Изработените с помощта на Windows форми приложения се задействат при настъпване на определено събитие или при определено действие от страна на потребителя, като например попълване на текстово поле или посочване и щракване на бутон.

Windows формите предоставят достъп до стандартните вградени контроли на Windows User Interface, като комбинира Windows API и т.нар. managed code (термин на Microsoft за програмен код, който се изпълнява под „управлението“ на виртуална машина .NET). В случая на Win32 API и Microsoft .NET Framework чрез Windows формите се постига по-разбираемо представяне, отколкото чрез Visual Basic или MFC.

#### Характеристики на WinForms

Всички визуални елементи в библиотеката Windows Forms са получени от класа Control. Това осигурява минималната необходима информация за всеки елемент от потребителския интерфейс, като например местоположение, размер, цвят, шрифт, текст, както и чести събития, като посочване и щракване и влачене и пускане. Класът Control поддържа докинг, което позволява приемственост при промяна на позицията, както и Microsoft Active Accessibility, което помага на потребители с увреждания.

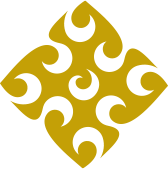
Освен че позволяват използването на контроли като бутони, текстови полета и др., Windows формите предоставят собствени контроли за ActiveX, оформление, валидация и свързване на данни. Тези контроли са представени чрез Graphics\_Device\_Interface (GDI+).

### Приложение за управлене на хотел

Приложението е предназначено да се използва от хотели по целия свят. Направено е да се използва от рецепционистите на хотелите, да регистрират и дерегистрират резервации и да се проверява дали дадена стая е изчистена след ползване. Също така хигиенистите на хотела имат достъп до софтуера и могат да маркират стаите като чисти след почистване. Но функциалноста му не приключва до тук. Може да се търси информация за стая по номера ѝ, както и да се извади общ оглед на всички стаи в един прозорец.

#### Лого/Емблема на приложението

Логото на софтуера служи за придобиването на уникалната му визия. Изработено е изцяло на GIMP (фигура 5), а на втория му вариант с името на холета, текста е добавен чрез Adobe Photoshop (фигура 6)

**A black and gold sign

Description automatically generated**

Фигура 5/ Лого /

Фигура 6/ Лого с текст /

#### Начална страница (Form1)

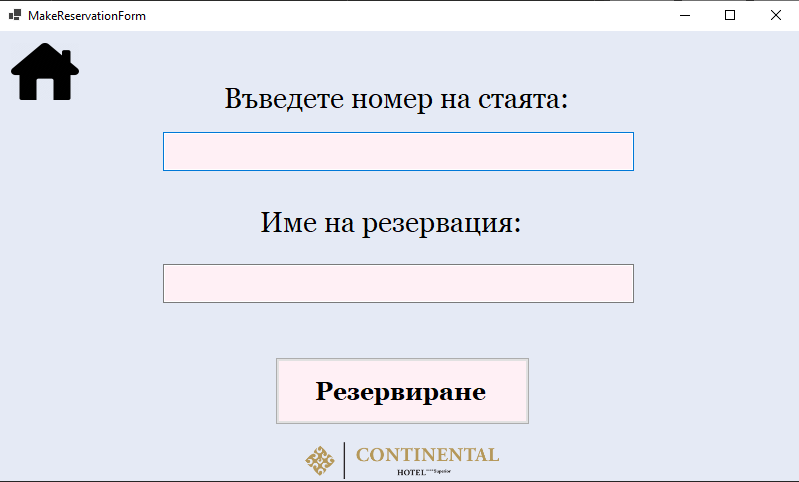
Както в повечето приложения за персонал, така и тук няма страница за вход с данни (log-in page), а директно се стартира главното меню. Чрез него се селектира (избира) кое подменю на приложението да се зареди.

****

Фигура 7/ Начална Страница /

#### Направи резервация (MakeReservationForm)

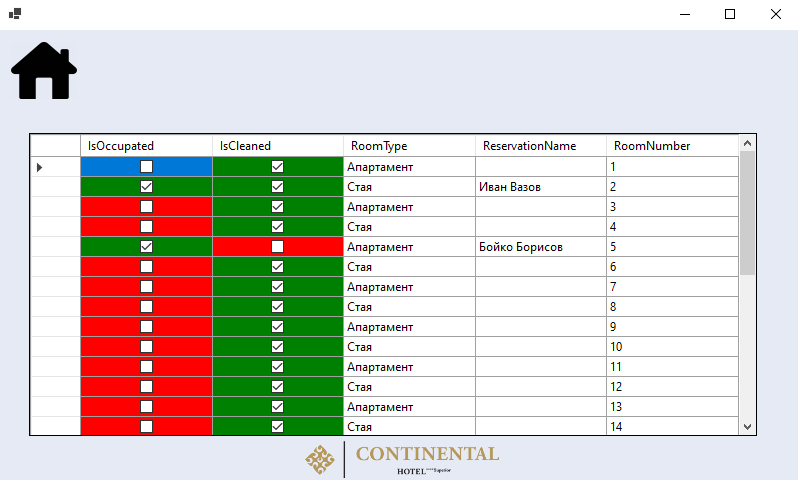
Това подменю извършва главната работа за приложението. Чрез него се правят резервации, които после могат да се видят в подменю „Всички стаи“. Всички полета са защитени от персонализирани изключения (Custom Exceptions)

****

Фигура 8 / Направи резервация /

#### Всички стаи (AllRoomsForm)

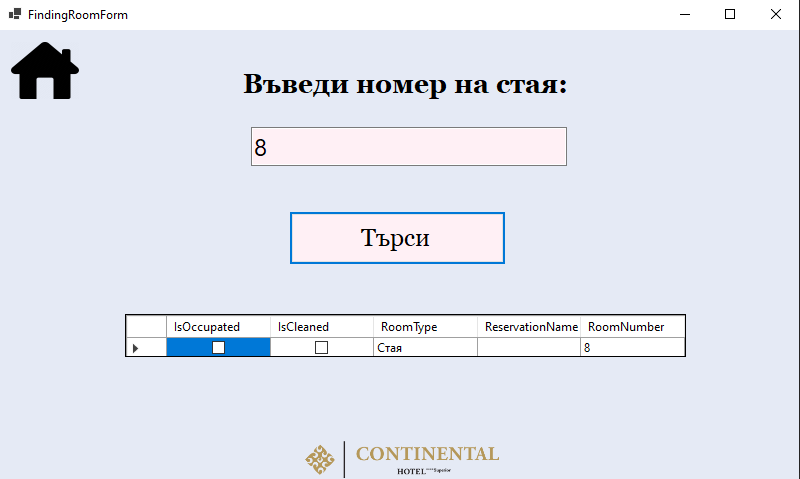
При стартиране на формата се зареждат стаите с информация за резервациите им чрез LocalStorage метода. Чете се от текстов файл с помощта на StreamReader и информацията се запазва в List<> от ръчно направен клас стаи Rooms. След като се зареди успешно, цялата информация излиза в графата DataGridView. Благодарение на нея става видимо коя стая е заета, дали е чиста, дали е тип „стая“ или „апартамент“, името на резервацията, ако има създадена и номера.

****

Фигура 9 / Всички стаи /

#### Търсене на стая (FindingRoomForm)

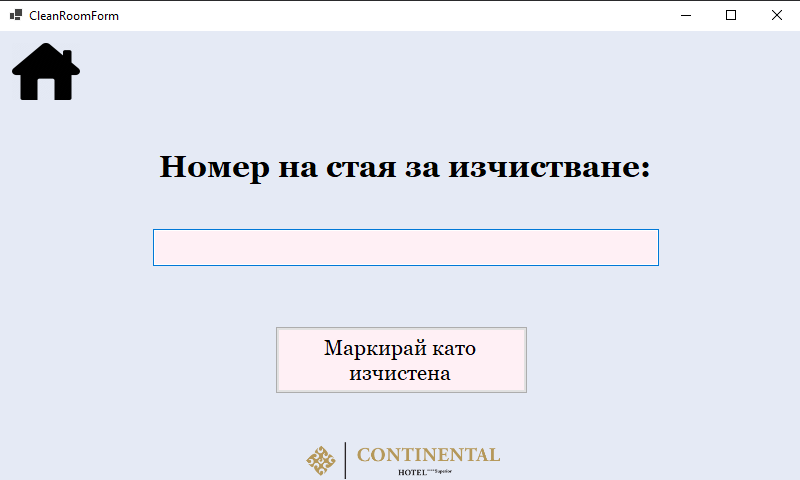
Това подменю преди да започне работа да търси по номер, трябва да зареди всички стаи във временен List<Rooms>. Когато вече търсенето стане възможно се въвежда номера на стаята и цялата информация за нея излиза в единична графа (DataGridView).

****

Фигура 10 / Търсене на стая /

#### Изчисти стая (CleanRoomForm)

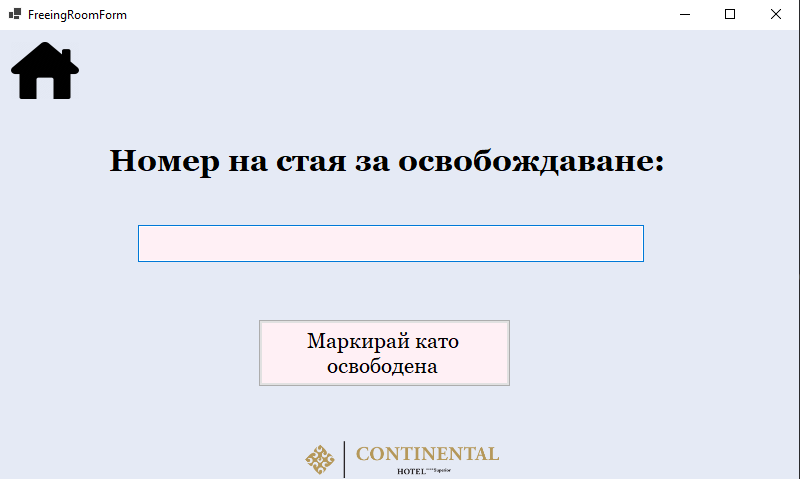
Тази форма прави точно това, което се очаква от нея по името ѝ. След като се заредят всички стаи, търси по даден номер, проверява дали е мръсна и ако е, я маркира като изчистена.

****

Фигура 11 / Изчисти стая /

#### Освободи (FreeingRoomForm)

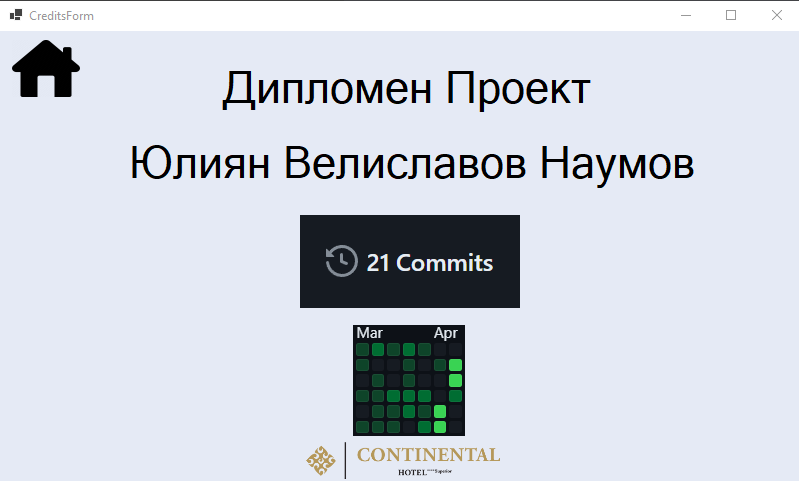
Това подменю извършва същата кодова дейност като (Фигура 11 / Изчисти стая /) но вместо да маркира стая като изчистена, я маркира като освободена. Освен, че стаята става свободна, изчезва и името на резервацията. И мръсна, и чиста стая може да се маркира като освободена и запазва променливата за чистота (IsCleaned). И двете подменюта са предназначени да бъдат достъпни и от хигиенистите на хотела.



Фигура 12 / Освободи стая /

#### Изготвил проекта (CreditsForm)

Тази форма накратко показва кой е изготвил Дипломния проект. Добавени са и екранни снимки (Screenshots) на статистики генерирани от личния GitHub профил на дипломанта. Първият показва колко пъти (а именно 21) е качвана нова подобрена версия на практическата част Hotel Continental в GitHub. Вторият показва в кои дни на двата месеца са били те и по колко пъти на ден.



Фигура 13 / Изготвил /

# ЗАКЛЮЧЕНИЕ

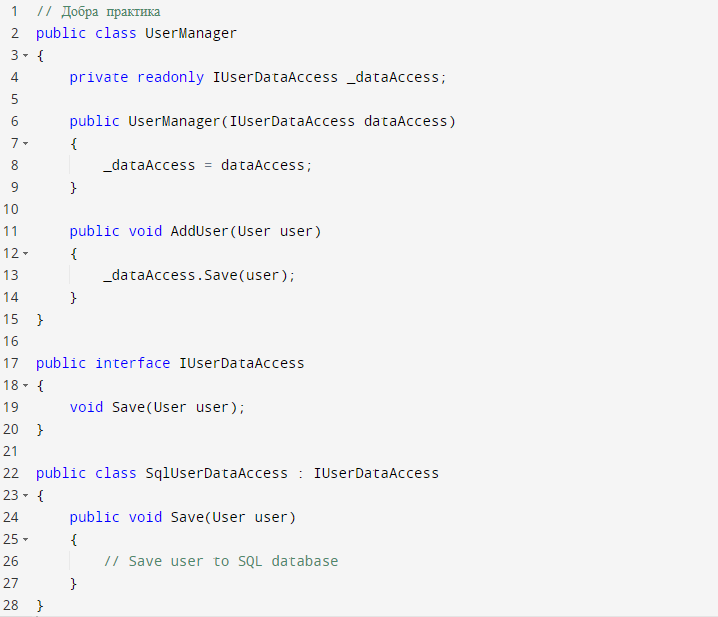
Като заключение може да се каже, че приложението е напълно защитето от негативна експлоатация (exploit/grief) и всякакви такива опити водят до ръчно писани съобщения за изключения (Custom exception messages) изкарани в отделен файл. Както така и файловите пътеки (FilePaths) за зареждане и запазване (import/export) на данните за резервации.

Бъдещо развитие би било създаването на база данни (Database) и да се запазват всички резервации и данни там, вместо локално (Local storage).

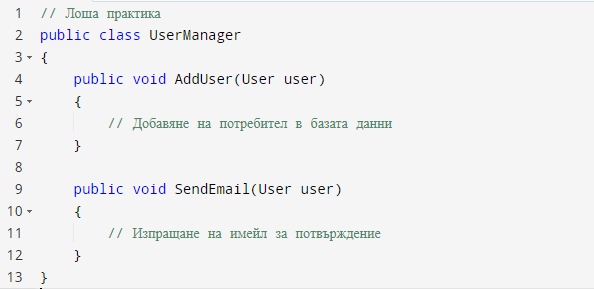
# ИЗПОЛЗВАНА ЛИТЕРАТУРА

1. Албахари Д. & Албахари, Б., (2020г.) C# джобен справочник. София, Асеневци
2. Наков С., Колев В. & Колектив, (2011г.) Въведение в програмирането със C#. София, СофтУни
3. https://bg.wikipedia.org/wiki/C\_Sharp
4. https://eurawest.bg/разработка-на-софтуер-6-етапа-които-тр/
5. https://gnnmobile.eu/какво-е-solid/
6. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/

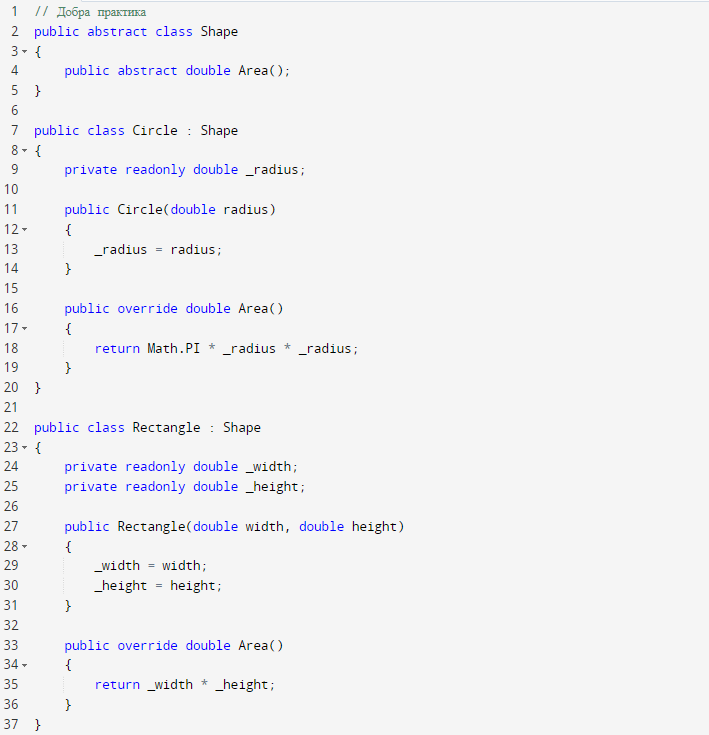
# ПРИЛОЖЕНИЯ



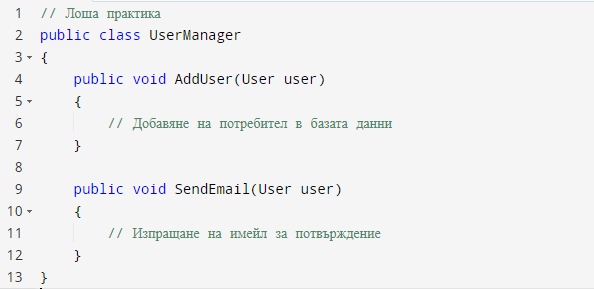
Приложение 1 / SRP добра практика /



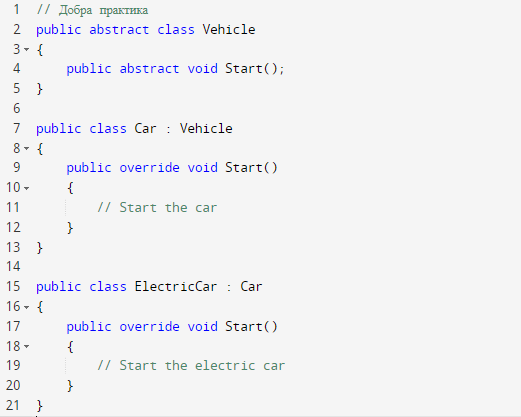
Приложение 2 / SRP лоша практика /



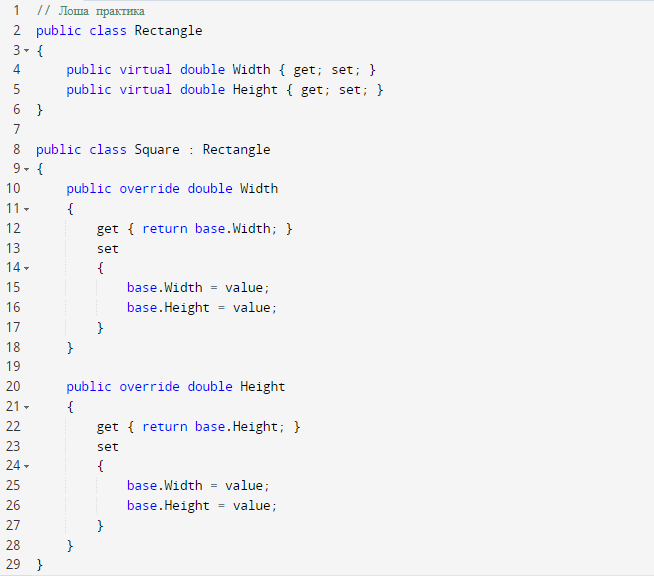
Приложение 3 / OCP добра практика /



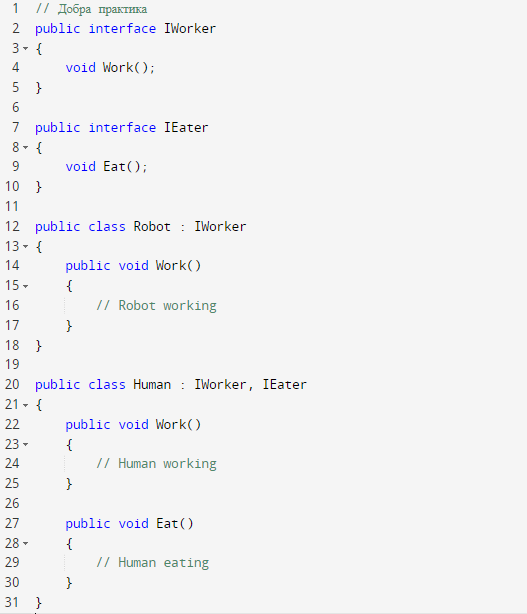
Приложение 4 / OCP лоша практика /



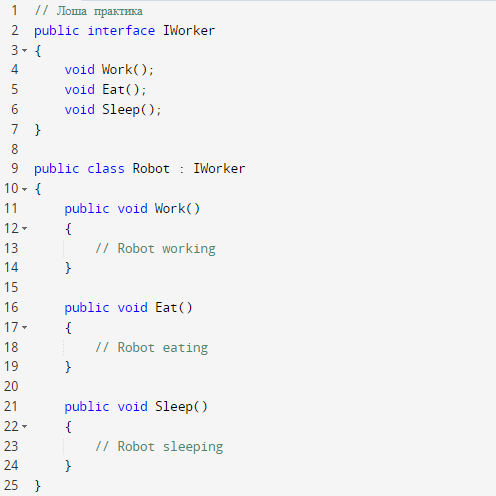
Приложение 5 / LSP добра практика /



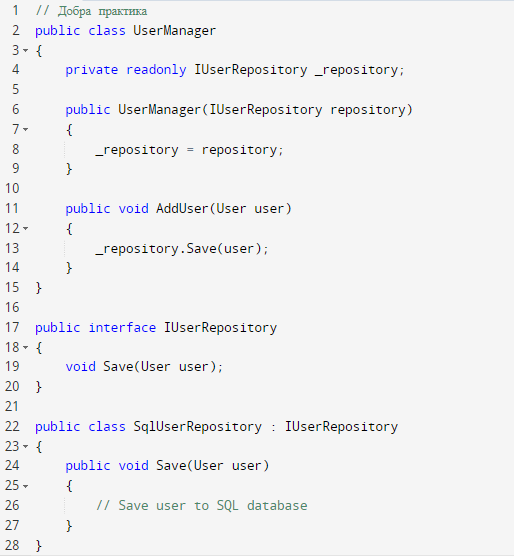
Приложение 6 / LSP лоша практика /



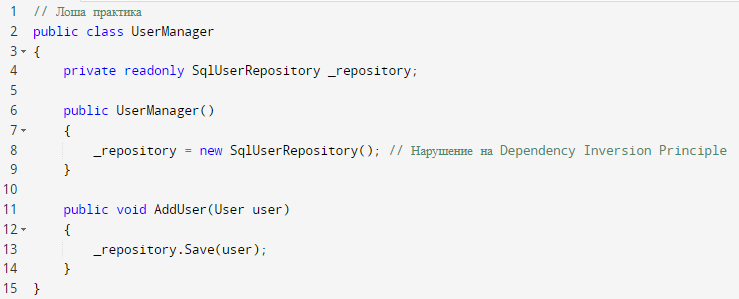
Приложение 7 / ISP добра практика /



Приложение 8 / ISP лоша практика /



Приложение 9 / DIP добра практика /



Приложение 10 / DIP лоша практика /