ЛАБОРАТОРНАЯ РАБОТА №3
НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Цель работы: приобрести навыки использования механизмов наследования и полиморфизма при реализации объектно-ориентированной модели.

Задачи:

1. Научиться применять принципы SOLID на практике.
2. Доработать предметную модель из предыдущей лабораторной работы.

Результатами работы являются:

- Библиотека классов, моделирующая предметную область согласно варианту задания
- Набор модульных тестов для демонстрации возможностей разработанной библиотеки классов
- Подготовленный отчет

КРАТКАЯ ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

Наследование

Класс может быть унаследован от другого класса с целью расширения или настройки исходного класса. Наследование от класса позволяет повторно использовать функциональность этого класса вместо ее построения с нуля. Класс может наследоваться только от одного класса, но сам может быть унаследован множеством классов, формируя тем самым иерархию классов. В этом примере мы начнем с определения класса по имени Asset:

```
public class Asset
{
    public string Name;
}
```

Далее мы определим классы Stock и House, которые будут унаследованы от Asset. Классы Stock и House получат все, что имеет Asset, плюс любые дополнительные члены, которые в них будут определены:

```
public class Stock : Asset
{
 // унаследован от Asset
 public long SharesOwned;
}
public class House : Asset
{
 // унаследован от Asset
public decimal Mortgage;
}
```

Вот как можно работать с этими классами:

```
Stock rnsft = new Stock { Name="MSFT",
   SharesOwned=1000 };
Console.WriteLine (rnsft.Name);
Console.WriteLine (rnsft.SharesOwned);
House rnansion = new House { Name="Mansion",
   Mortgage=250000 };
Console.WriteLine (rnansion.Name);
Console.WriteLine (rnansion.Mortgage);
```

Производныее классы Stock и House наследуют свойство Name от базового класса Asset.

Производный класс также называется подклассом. Базовый класс также называется суперклассом.

Все остальные классы в .NET, даже те, которые создаются программистами, а также базовые типы, такие как System.Int32, являются неявно производными от класса Object. Даже если не указывается класс Object в качестве базового, по умолчанию неявно класс Object все равно стоит на вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object. Рассмотрим эти методы:

- ToString – служит для получения строкового представления данного объекта.
- GetHashCode – позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты.
- GetType – позволяет получить тип данного объекта.
- Equals – позволяет сравнить два объекта на равенство

Полиморфизм

Ссылки являются полиморфными. Это значит, что переменная типа x может ссылаться на объект, относящийся к подклассу x. Например, рассмотрим следующий метод:

```
public static void Display (Asset asset)
{
    Systern.Console.WriteLine (asset.Name);
}
```

Этот метод способен отображать значение свойства Name объектов Stock и House, т.к. они оба являются Asset:

```
Stock rnsft = new Stock …;
House mansion = new House…;
Display (rnsft);
Display (mansion);
```

В основе работы полиморфизма лежит тот факт, что подклассы (Stock и House) обладают всеми характеристиками своего базового класса (Asset). Однако обратное утверждение не верно. Если метод Display переписать так, чтобы он принимал House, то передавать ему Asset будет невозможно:

```
static void Main() { Display (new Asset() );
public static void Display (House house)
{
    Systern.Console.WriteLine (house.Mortgage};
}
```

Виртуальные функции-члены

Функция, помеченная как виртуальная (virtual), может быть переопределена в подклассах, где требуется предоставление ее специализированной реализации. Объявлять виртуальными можно методы, свойства, индексаторы и события:

```
public class Asset
{
 public string Name;
 public virtual decimal Liability => 0;
// Свойство, сжатое до выражения
}
```

(Конструкция Liability => 0 является сокращенной записью для { get return 0; }) .

Виртуальный метод переопределяется в подклассе с применением модификатора override:

```
public class Stock: Asset
{
 public long SharesOwned;
}
 public class House: Asset
{
 public decimal Mortgage;
 public override decimal Liability => Mortgage;
}
```

По умолчанию свойство Liability класса Asset возвращает 0. Класс Stock не нуждается в специализации этого поведения. Тем не менее, класс House специализирует свойство Liability, чтобы возвращать значение Mortgage:

```
House mansion = new House {Name="McMansion",
   Mortgage=250000 ) ;
Asset a= mansion;
Console.WriteLine (mansion.Liability);
Console.WriteLine (a.Liability);
```

Сигнатуры, возвращаемые типы и доступность виртуального и переопределенного методов должны быть идентичными. Внутри переопределенного метода можно вызывать его реализацию из базового класса с помощью ключевого слова base.

Абстрактные классы и абстрактные члены

Класс, объявленный как абстрактный (abstract), не разрешает создавать свои экземпляры. Вместо этого можно создавать только экземпляры его конкретных подклассов.

В абстрактных классах есть возможность определять абстрактные члены. Абстрактные члены похожи на виртуальные члены за исключением того, что они не предоставляют стандартную реализацию. Реализация должна обеспечиваться подклассом, если только этот подкласс тоже не объявлен как abstract:

```
public abstract class Asset
{
    // Обратите внимание на пустую реализацию
    public abstract decimal NetValue { get; }
}
public class Stock: Asset
{
    public long SharesOwned;
    public decimal CurrentPrice;
    // Переопределить подобно виртуальному методу
    public override decimal NetValue =>
          CurrentPrice * SharesOwned;
}
```

Эквивалентность значений и ссылочная эквивалентность

Различают два вида эквивалентности.

- Эквивалентность значений: два значения эквивалентны в некотором логическом смысле.
- Ссылочная эквивалентность: две ссылки ссылаются в точности на один и тот же объект.

По умолчанию типы значений используют эквивалентность значений; ссылочные типы используют ссылочную эквивалентность.

На самом деле типы значений могут применять только эквивалентность значений (если они не упакованы). Простой демонстрацией эквивалентности значений может служить сравнение двух чисел:

```
int x = 5, y= 5;
Console. WriteLine (x == y) ;
// True (в силу эквивалентности значений)
```

Более сложная демонстрация предусматривает сравнение двух структур DateTimeOffset. Приведенный ниже код выводит на экран True, потому что две структуры DateTimeOffset относятся к одной и той же точке во времени, а, следовательно, считаются эквивалентными:

```
var dt1 = new DateTimeOffset (2010, 1, 1, 1, 1, 1,
   TimeSpan.FromHours(8));
var dt2 = new DateTimeOffset (2010, 1, 1, 2, 1, 1,
   TimeSpan.FromHours(9));
Console.WriteLine (dtl == dt2); // True
```

DateTimeOffset -это структура, семантика эквивалентности которой была настроена. По умолчанию структуры поддерживают специальный вид эквивалентности значений, называемый структурной эквивалентностью, при которой два значения считаются эквивалентными, если эквивалентны все их члены.

Ссылочные типы по умолчанию поддерживают ссылочную эквивалентность.

В следующем примере ссылки f1 и f2 не являются эквивалентными - несмотря на то, что их объекты имеют идентичное содержимое:

```
class Foo { public int X; }
…
Foo fl = new Foo { X = 5 };
Foo f2 = new Foo { X = 5 };
Console. WriteLine ( fl == f2} ; // False
```

В противоположность этому f3 и f1 эквивалентны, т.к. они ссылаются на тот же самый объект:

```
Foo f3 = fl;
Console.WriteLine (fl == f3); // True
```

Стандартные протоколы эквивалентности

Существуют три стандартных протокола, которые типы могут реализовывать для сравнения эквивалентности:

- операции == и ! =;
- виртуальный метод Equals в классе object;
- интерфейс IEquatable<T>.

Вдобавок имеются подключаемые протоколы и интерфейс IStructuralEquatable.

Операции = и ! =

Вы уже видели, каким образом стандартные операции == и != выполняют сравнения равенства/неравенства, во многих примерах. Тонкости с == и ! = возникают из-за того, что они являются операциями, поэтому распознаются статически (на самом деле они реализованы в виде статических функций). Следовательно, когда вы используете операцию == или ! =, то решение о том, какой тип будет выполнять сравнение, принимается на этапе компиляции, и никакое виртуальное поведение в игру не вступает. Как правило, это и желательно. В показанном далее примере компилятор жестко привязывает операцию == к типу int, потому что переменные x и y имеют тип int:

```
int x = 5;
int y= 5;
Console.WriteLine (x == y); // True
```

Но в приведенном ниже примере компилятор привязывает операцию == к типу object:

```
object x = 5;
object y = 5;
Console.WriteLine (x == y); // False
```

Поскольку тип object является классом (т.е. ссылочным типом), операция == типа object применяет для сравнения x и y ссылочную эквивалентность. Результатом будет false, т.к. x и y ссылаются на разные упакованные объекты в куче.

Виртуальный метод Object.Equals

Для корректного сравнения x и y в предыдущем примере мы можем использовать виртуальный метод Equals. Метод Equals определен в классе System.Object, поэтому он доступен всем типам:

```
object x = 5;
object y = 5;
Console.WriteLine (x.Equals (y)); // True
```

Метод Equals распознается во время выполнения - в соответствии с действительным типом объекта. В данном случае вызывается метод Equals типа Int32, который применяет к операндам эквивалентность значений, возвращая true. Со ссылочными типами метод Equals по умолчанию выполняет сравнение ссылочной эквивалентности; для структур метод Equals осуществляет сравнение структурной эквивалентности, вызывая Equals на каждом их поле.

Следовательно, метод Equals подходит для сравнения двух объектов в независимой от типа манере. Показанный ниже метод сравнивает два объекта любого типа:

```
public static bool AreEqual (object obj1,
    object obj2) => objl.Equals (obj2);
```

Тем не менее, имеется один сценарий, при котором такой метод не срабатывает. Если первый аргумент равен null, то сгенерируется исключение NullReferenceException. Ниже приведена исправленная версия метода:

```
public static bool AreEqual (object obj1,
    object obj2)
{
    if (obj1 = null) return obj2 = null;
    return objl.Equals (obj2);
}
```

Или более сжато:

```
public static bool AreEqual (object obj1,
   object obj2) => obj1 == null? obj2 == null:
   obj1.Equals (obj2);
```

Статический метод object.Equals

В классе object определен статический вспомогательный метод, который делает работу метода AreEqual из предыдущего примера. Его именем является Equals (точно как у виртуального метода), но никаких конфликтов не возникает, потому что он принимает два аргумента:

```
public static bool Equals (object objA, object objB)
```

Этот метод предоставляет алгоритм сравнения эквивалентности, безопасный к null, который предназначен для ситуаций, когда типы не известны на этапе компиляции. Например:

```
object x = 3, y= 3;
Console.WriteLine (object.Equals (x, y));
// True
x = null;
Console.WriteLine (object.Equals (x, y));
// False
y = null;
Console.WriteLine (object.Equals (x, y));
// True
```

Данный метод полезен при написании обобщенных типов. Приведенный ниже код не скомпилируется, если вызов object. Equals заменить операцией == или ! =:

```
class Test <T>
{
      T _value;
      public void SetValue (T newValue)
      {
            if (! object. Equals (newValue, _
         value))
         {
         _value = newValue;
         OnValueChanged();
         }
      }
      protected virtual void OnValueChanged() { ...
   }
}
```

Операции здесь запрещены, потому что компилятор не может выполнить связывание со статическим методом неизвестного типа.

Статический метод object.ReferenceEquals

Иногда требуется принудительно применять сравнение ссылочной эквивалентности. Статический метод object.ReferenceEquals делает именно это:

```
class Widget {…}
class Test
{
      static void Main ()
      {
            Widget w1 = new Widget ();
            Widget w2 = new Widget();
            Console.WriteLine
               (object.ReferenceEquals (w1, w2));
            // False
      }
}
```

Поступать подобным образом может быть необходимо из-за того, что в классе Widget возможно переопределение виртуального метода Equals, в результате чего wl.Equals(w2) будет возвращать true. Более того, в Widget возможна перегрузка операции==, из-за которой wl==w2 будет также давать true. В подобных случаях вызов object. ReferenceEquals гарантирует использование нормальной семантики ссылочной эквивалентности.

Интерфейсы

Интерфейс похож на класс, но предоставляет для своих членов только спецификацию, а не реализацию. Интерфейс обладает следующими особенностями.

- Все члены интерфейса являются неявно абстрактными. В противоположность этому класс может предоставлять как абстрактные члены, так и конкретные члены с реализациями.
- Класс (или структура) может реализовывать несколько интерфейсов. В противоположность этому класс может быть унаследован только от одного класса, а структура вообще не поддерживает наследование (за исключением того, что она порождена от System.ValueType).

Объявление интерфейса похоже на объявление класса, но при этом никакой реализации для его членов не предоставляется, т.к. все члены интерфейса неявно абстрактные. Эти члены будут реализованы классами и структурами, которые реализуют данный интерфейс. Интерфейс может содержать только методы, свойства, события и индексаторы, и это совершенно неслучайно в точности соответствует членам класса, которые могут быть абстрактными. Ниже показано определение интерфейса IEnumerator из пространства имен System. Collections:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset () ;
}
```

Члены интерфейса всегда неявно являются public, и для них нельзя объявлять какие-либо модификаторы доступа. Реализация интерфейса означает предоставление реализаций public для всех его членов:

```
internal class Countdown: IEnumerator
{
    int count = 11;
    public bool MoveNext () => count--> 0;
    public object Current => count;
    public void Reset()
    {
          throw new NotSupportedException();
    }
}
```

Объект можно неявно приводить к любому интерфейсу, который он реализует. Например:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
    Console.Write (e.Current); // 109876543210
```

Расширение интерфейса

Интерфейсы могут быть производными от других интерфейсов. Например:

```
public interface IUndoable { void Undo();}
public interface IRedoable: IUndoable
    { void Redo(); }
```

Интерфейс IRedoable "наследует" все члены интерфейса IUndoable. Другими словами, типы, которые реализуют IRedoable, должны также реализовывать члены IUndoable.

Явная реализация членов интерфейса

Реализация множества интерфейсов может иногда приводить к конфликту между сигнатурами членов. Разрешать такие конфликты можно за счет явной реализации члена интерфейса. Рассмотрим следующий пример:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }
public class Widget: I1, I2
{
      public void Foo ()
      {
         Console.WriteLine
         ("Widget's irnplernentation of l1.Foo");
      }
      int I2.Foo ()
      {
         Console.WriteLine
         ("Widget's implementation of I2.Foo");
         return 42;
      }
}
```

Поскольку интерфейсы I1 и I2 имеют методы Foo с конфликтующими сигнатурами, метод Foo интерфейса I2 в классе Widget реализуется явно. Это позволяет двум методам сосуществовать в рамках одного класса. Единственный способ вызова явно реализованного метода предусматривает приведение к его интерфейсу:

```
Widget w = new Widget();
w.Foo ();
((I1)w).Foo();
((I2)w).Foo();
```

Другой причиной явной реализации членов интерфейса может быть необходимость сокрытия членов, которые являются узкоспециализированными и нарушающими нормальный сценарий использования типа.

Реализация виртуальных членов интерфейса

Неявно реализованный член интерфейса по умолчанию является запечатанным. Чтобы его можно было переопределить, он должен быть помечен в базовом классе как virtual или abstract. Например:

```
public interface IUndoaЫe { void Undo();}
public class TextBox: IUndoable
{
   public virtual void Undo () => Console
         .WriteLine( "TextBox. Undo");
}
public class RichTextBox: TextBox
{
   public override void Undo() =>
         Console.WriteLine ("RichTextBox.Undo");
}
```

Обращение к этому члену интерфейса либо через базовый класс, либо интерфейс приводит к вызову его реализации из подкласса:

```
RichTextBox r = new RichTextBox();
r. Undo ();   // RichTextBox.Undo
((IUndoable)r).Undo(); // RichTextBox.Undo
((TextBox)r) .Undo(); // RichTextBox.Undo
```

Явно реализованный член интерфейса не может быть помечен как virtual, равно как и не может быть переопределен обычным образом. Однако он может быть повторно реализован.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РБОТУ

Доработать предметную модель согласно выданному варианту задания.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

При выполнении лабораторной работы необходимо использовать стандарт кодирования языка C#. Для разработанных классов создать набор модульных тестов, проверяющих корректность кода, а также демонстрирующих полученный API.

ВАРИАНТЫ ЗАДАНИЙ

Варианты № 1. «Служба доставки»

В качестве маркетингового хода менеджер службы доставки решил расширить систему скидок, предоставляемых клиентам компании. Теперь доступны следующие скидки:

- 5 %, если стоимость заказа превышает 1 000;
- 10 %, если стоимость заказа превышает 1 500;
- 7 %, если клиент – «привилегированный»;
- 15 %, если стоимость заказа превышает 1 500, и клиент – «привилегированный»;
- 12 %, если заказ был сформирован в период с 25 декабря по 7 января;

Необходимо модифицировать модель, реализованную в лабораторной работе № 2, так, чтобы полученное решение позволяло:

- определять, какие из доступных скидок могут быть применены к заказу;
- указывать скидку, предоставленную клиенту в рамках обработки заказа;
- определять общую стоимость заказа с учетом предоставленной скидки.

При этом решение должно обеспечивать гибкость для системы скидок, т. е. добавление новых видов скидок и удаление устаревших не должно приводить к переписыванию существующего кода обработки заказов.

Вариант № 2. «Управление задачами»

Компания, использующая приложение «Управление задачами», выполняет проекты по заказу других компаний. Разные контракты предполагают разные условия выплат неустойки, связанной с превышением сроков выполнения задач, причем для разных задач даже в рамках одного проекта могут действовать разные правила.

На настоящий момент встречались следующие условия:

- за каждый день просрочки начисляется пеня 1% от стоимости задачи, но не более 25%;
- за каждый день просрочки начисляется пеня 2% от стоимости задачи, но не более 40%;
- пеня начисляется фиксированного размера, вне зависимости от того, на сколько дней превышен срок;
- за каждые 2 дня просрочки начисляется пеня 2% от стоимости задачи;
- за каждый день просрочки начисляется пеня фиксированного размера, причем, при превышении срока 3 дня и менее неустойка не выплачивается (например, если фиксированная ставка равна 1 000 руб., то за превышение в 1-3 дня пеня не начисляется, за превышение в 4 дня начисляется пеня 4 000 руб. и т. д.).

Необходимо модифицировать модель, реализованную в лабораторной работе № 2, так, чтобы полученное решение позволяло:

- указывать правила начисления пени для задачи;
- определять общую стоимость задачи с учетом неустойки.

При этом решение должно обеспечивать гибкость системы правил начисления пени, т. е. добавление новых правил и удаление устаревших не должно приводить к переписыванию существующего кода работы с проектами.

Вариант № 3. «Учебная программа»

В Университете для получения разных квалификационных степеней разные требования:

- за все время обучения студент должен прослушать определенное количество спецкурсов, а также набрать в сумме некоторое количество кредитных единиц;
- студент должен прослушать заданное количество спецкурсов, причем в это число должны входить указанные обязательные спецкурсы (задан списан регистрационных номеров);
- студент в сумме должен набрать определенное количество кредитных единиц, а также прослушать указанные курсы (задан списан регистрационных номеров);
- учебная программа должна быть составлена в 2013 году или ранее; за все время обучения студент должен набрать в сумме некоторое количество кредитных единиц;
- за все время обучения студент должен набрать в сумме некоторое количество кредитных единиц; все прослушанные курсы должны завершаться экзаменом;
- учебная программа должна быть составлена в 2014 году или ранее; студент должен прослушать определенное количество спецкурсов, отработать не менее заданного количества часов практических занятий и выполнить не менее заданного количества курсовых работ.

Необходимо модифицировать модель, реализованную в лабораторной работе № 2, так, чтобы полученное решение позволяло:

- указывать правила получения квалификационной степени;
- определять допустимость составленной программы по правилам, установленным для квалификационной степени.

При этом решение должно быть таким, что добавление новых правил получения степени и удаление устаревших не должно приводить к переписыванию существующего кода обработки учебных программ.

Варианты № 4. «Начисление зарплаты»

Компания начисляет заработную плату своим сотрудникам по различным правилам (во всех схемах оплаты должен учитываться подоходный налог):

- штатные сотрудники получают заработную плату по правилам, описанным в лабораторной работе № 2;
- некоторые штатные сотрудники получают базовую заработную плату с учетом переработок и работы в выходные, а также за заключенные контракты 5% комиссионных от суммы контрактов;
- заработная плата начисляется за фактически отработанное время (вне зависимости от того, превышена ли норма рабочего времени в день, и была ли работа в выходные дни) с учетом должности и разряда сотрудника; профсоюзные отчисления составляют 1,5%;
- заработная плата начисляется за фактически отработанное время с учетом только должности сотрудника; профсоюзные отчисления составляют 0,5%;
- заработная плата начисляется за фактически отработанное время с учетом только должности сотрудника, за заключенные контракты 7% комиссионных от суммы контрактов; профсоюзные отчисления составляют 1%;
- сотрудник получает только комиссионные в размере 10% от суммы заключенных контрактов; профсоюзные отчисления не производятся.

Необходимо модифицировать модель, реализованную в лабораторной работе № 2, так, чтобы полученное решение позволяло:

- указывать контракты, заключенные за период, учитываемый в табеле (интерес представляют следующие атрибуты сущности «Контракт»: номер, дата, клиент, сумма, комментарий);
- указывать схему начисления заработной платы для каждого сотрудника;
- определять величину зарплаты за рабочее время, указанное в табеле.

При этом решение должно быть таким, что добавление новых схем начисления заработной платы не должно приводить к переписыванию существующего кода.

Вариант № 5. «Качество работы»

В целях обеспечения более детального мониторинга качества работы смен рабочих, предприятие начинает переработку системы присвоения категорий качества. Измененный вариант выглядит так:

- 1-я категория: не более 3% брака; 80% изделий изготовлены за установленное нормативом время; для остальных изделий среднее время переработки не превышает 5 минут (3 минуты для изделий с артикулом «SCOF-B1», 4 минуты для изделий «SCOY-ME2»);
- 2-я категория: не более 5% брака; 75% изделий изготовлены за установленное нормативом время; для остальных изделий среднее время переработки не превышает 5 минут (4 минуты для изделий «WVP-B2»);
- 3-я категория: не более 6% брака; 75% изделий изготовлены за установленное нормативом время; для остальных изделий среднее время переработки не превышает 6 минут (4 минуты для изделий «WVP-B1»);
- 4-я категория: не более 8% брака; 60% изделий изготовлены за установленное нормативом время; для остальных изделий среднее время переработки не превышает 6 минут (все изделия «WVTGN-MRL» изготовлены за нормативное время);
- 5-я категория: в остальных случаях.

Необходимо модифицировать модель, реализованную в лабораторной работе № 2, так, чтобы полученное решение учитывало новые правила присвоения категорий качества партии. При этом решение должно быть таким, что добавление новых схем категоризации партий (а это будет происходить часто) не должно приводить к переписыванию существующего кода.

Вариант № 6. «Лаборатория»

Для разных видов анализов правила определения нормального показателя различаются:

- показатель должен находиться в заданных пределах (это правило реализовано в лабораторной работе № 2);
- показатель не должен превышать заданного значения;
- показатель должен быть не меньше заданного значения;
- показатель должен находиться вне заданного диапазона значений (например, ≤ 3,50 или ≥ 5,42);
- показатель должен принадлежать одному из двух заданных диапазонов (например, 12,30–14,98 или 16,01–19,49);
- показатель должен иметь одно из трех заданных значений (0,57, или 0,93, или 1,20).

Необходимо модифицировать модель, реализованную в лабораторной работе № 2, так, чтобы полученное решение учитывало новые правила нормы показателей (и, как следствие, успешности прохождения теста). При этом решение должно быть таким, что добавление новых правил определения нормы не должно приводить к переписыванию существующего кода.

Вариант № 7. «Учебные сертификаты»

Каждый профессор сам определяет требования, которые необходимо выполнить, чтобы получить сертификат о прохождении курса. В настоящее время для разных курсов действуют следующие правила:

- правила, реализованные в лабораторной работе № 2;
- успешно принятые работы не могут пересдаваться; если работа сдана не в срок, то при подсчете учитываются только 50% набранных за нее баллов (округление вниз);
- все работы могут пересдаваться только один раз (кроме экзамена – экзамен пересдавать нельзя), при этом всегда учитывается последний результат; сдача работ с опозданием не штрафуется (кроме экзамена – экзамен может быть сдан только в срок);

– все работы могут пересдаваться до трех раз, при этом всегда учитывается максимальный результат; сдача работ с опозданием на 1 неделю не штрафуется; при сдаче с опозданием более 1 недели при подсчете учитываются только 80% набранных за работу баллов;
– все работы могут пересдаваться неограниченное количество раз, при этом всегда учитывается последний результат; при сдаче с опозданием не более 1 недели при подсчете учитываются только 85% набранных за работу баллов, работы с опозданием более 1 недели не принимаются;
– домашние задания могут пересдаваться до пяти раз; рубежные тесты и экзамен – до трех раз; при подсчете учитывается только 90% максимального результата (если была хотя бы одна пересдача); если работа сдана не в срок, то при подсчете учитываются только: 80% набранных баллов для домашнего задания, 75% баллов для рубежного теста; 60% баллов для экзамена (округление вверх).

Необходимо модифицировать модель, реализованную в лабораторной работе № 2, так, чтобы полученное решение учитывало новые правила выдачи сертификатов. При этом решение должно быть таким, что добавление новых правил не должно приводить к переписыванию существующего кода.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Дайте определение термина наследование. Для чего наследование может использоваться?
2. Дайте определение термина полиморфизм. Для чего полиморфизм может использоваться?
3. Сформулируйте принцип открытости/закрытости.
4. Сформулируйте принцип подстановки Лискоу.
5. Опишите синтаксис наследования в C#.
6. Назовите основные особенности виртуальных методов. Для чего используются виртуальные методы.
7. Опишите синтаксис определения виртуальных методов и их переопределения в производных классах на языке C#.
8. Раскройте значение абстрактного класса и абстрактного метода. В каких ситуациях применяются абстрактные классы, абстрактные методы?
9. Перечислите методы, наследуемые всеми классами от класса System.Object.
10. Каково назначение метода Object.Equals? Назовите, в каких случаях требуется его переопределение. Приведите общую схему его переопределения.
11. Каково назначение метода Object.GetHashCode? Назовите, в каких случаях требуется его переопределение. Приведите общую схему его переопределения.
12. Каково назначение метода Object.ToString? Назовите, в каких случаях требуется его переопределение.
13. Что такое интерфейс? Для чего используются интерфейсы?
14. Как объявить и реализовать интерфейс на языке C#?

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 10 часов: 9 часов на выполнение работы, 1 час на подготовку и защиту отчета по лабораторной работе.

Номер варианта студенту выдается преподавателем.

Отчет на защиту предоставляется в печатном виде.

Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), UML-диаграмма классов модели, этапы выполнения работы (со скриншотами), результаты выполнения работы. выводы.