ГБОУ ВО МО университет Дубна

Институт системного анализа и управления

Сеннер А.Е., Ушанкова М.Ю, Мельникова О.И., Самойлов Ю.Е.

Сборник практических заданий

и методических рекомендаций их выполнения по учебному курсу «Объектно-Ориентированное Программирование» (ООП)"

Дубна 2020

Оглавление

Авторы приносят свою искреннюю благодарность доценту Александру Михайловичу Задорожному за ряд ценных замечаний при рецензировании данного сборника практических задач по учебному курсу Объектно-ориентированное программирование.

Введение

Парадигма объектно-ориентированного программирования (ООП) не нова. Она приобрела популярность во второй половине 80-х годов прошлого века вместе с активным развитием языков *C++, Objective C, Object Pascal* и других. Базовые основы ООП – наследование, инкапсуляция и полиморфизм в настоящее время поддерживают практически все современные языки программирования. Эта парадигма стала неотъемлемой частью практически всех современных программных проектов.

Предлагаемый практикум направлен на закрепление теоретического материала, читаемого на лекциях учебного курса «Объектно-ориентированное программирование» для студентов очной формы обучения и для самостоятельного изучения данного курса студентами заочной и дистанционной форм обучения, направлений подготовки: «Прикладная математика и информатика», «Фундаментальные информатика и информационные технологии», «Информатика и вычислительная техника», «Программная инженерия» и «Информационные системы и технологии» Института системного анализа и управления. А также для других направлений, в учебном графике которых есть дисциплина «Объектно-ориентированное программирование».

Практикум включает в себя следующие разделы:

- Теоретический, в котором кратко определяются базовые понятия и термины ООП.
- Методический, содержащий общие рекомендации по выполнению практических заданий.
- Набор заданий по учебному курсу для студентов. Описание каждой задачи в практическом разделе состоит из:
    - цели задания;
    - описания решаемой задачи;
    - методических указаний к решению;
    - вопросов для самоконтроля.

Что необходимо знать и уметь

Прежде чем приступить к изучению курса «Объектно-ориентированное программирование» студентам направлений Института системного анализа и управления университета Дубна необходимы определенные знания и навыки, приобретаемые в рамках учебной дисциплины «Программирование на языке высокого уровня», изучаемой на первом курсе.

Студентам необходимо уверенно знать основы языка программирования *C#* и основы работы в среде разработки *Visual Studio*. А именно:

- Знать встроенные типы данных (*int*, *double*, *bool*, *string* и т.д.). Уметь объявлять, хранить, передавать и обрабатывать переменные этих типов, приводить один тип к другому. Уметь работать с коллекциями (массивами) и списками.
- Понимать функциональное назначение и знать синтаксис основных операторов. Уметь применять операторы выражений (*new*, =, ++ и т.д.), операторы выбора (*if, switch*), операторы итераций (*while, do, for* и *foreach*), операторы перехода (*break, continue, return* и т.д.). Уметь строить блоки операторов {} и понимать, как они работают [1].
- Иметь четкое представление о структуре разрабатываемого проекта. Что входит в понятие «проект» и «решение» в среде разработки *Visual Studio*. Уметь создавать решения и добавлять в них проекты. Не терять файлы проектов и решений, понимать, какие из них необходимы для компиляции [2].
- Владеть средствами отладки разрабатываемого программного продукта в случае ошибок при выполнении программы, получения не некорректных результатов. Уметь установить точку останова и запустить отладчик. Переходить по коду с помощью пошаговых команд (шаги с обходом модуля и шаги с заходом в модуль), быстрое выполнение до точки в коде с помощью мыши и т.д.) [3].
- Уметь программно реализовывать простейшие алгоритмы, работать с файлами и понимать основы работы в *Windows Forms*.

Теоретические основы ООП и его термины

Объектно-ориентированное программирование — один из самых эффективных подходов при разработке современного программного продукта.

Раньше программисты, в большинстве случаев, использовали функциональный или процедурный принцип программирования. С течением времени программы становились всё сложнее и больше, что доставляло проблемы разработчикам при поддержке таких программ и внесении изменений. Эту проблему во многом решило объектно-ориентированное программирование. ООП позволило объединить данные и методы, относящиеся к одной сущности, и работать с ними, как с одним целым.

ООП привносит два ключевых понятия: *класс* и *объект*.

Класс — это абстрактный тип данных. С помощью класса описывается некоторая сущность (ее характеристики и возможные действия).

Класс называется абстрактным типом данных потому, что, описывая реально существующий объект, мы не перечисляем все его возможные характеристики, а выбираем те необходимые, которые позволят решать определенные стоящие перед нами задачи (делать определенные действия) с объектом этого класса. В литературе также можно встретить следующее определение класса — User Defined Type (UDT). При этом пользователем здесь считается программист, создающий «свой» тип данных. Например, класс может описывать студента, автомобиль и т.д. Описав класс, мы можем создать его экземпляр – объект. Объект — это уже конкретный представитель класса.

Таким образом объект в программировании — некоторая сущность в виртуальном пространстве, обладающая определённым состоянием и поведением, имеющая заданные значения свойств (атрибутов) и операций над ними (методов). Как правило, при рассмотрении объектов выделяется то, что объекты принадлежат одному или нескольким классам, которые определяют поведение (являются моделью) объекта. Термины «экземпляр класса» и «объект» взаимозаменяемы.

Термин «объект» в программном обеспечении впервые был введен в языке *Simula* и применялся для моделирования реальности.

Пример:

Есть класс «Стиральная машина» в котором машина определяется следующими параметрами: компания-производитель, наименование модели, серийный номер изделия, загрузка и т.д.

Экземпляром класса «Стиральная машина» может быть ваша стиральная машина, имеющая следующие свойства: компания-производитель «Samsung», наименование модели «QuickDrive WW8800M», серийный номер изделия «WW10M86KNOA/LP», загрузка – 10 кг. и другие.

А еще одним экземпляром класса (наряду с уже существующим «Samsung») может быть: «Zanussi», наименование модели «Classic», серийный номер изделия «AA327/IWCD5085», загрузка – 7 кг.

Таким образом, однажды описанный класс может иметь какое угодно количество объектов (экземпляров класса).

Основные понятия ООП

В данном разделе формулируются и кратко описываются основные понятия ООП. **Обратите внимание**, что семантика ряда терминов ООП, хотя и совпадает по написанию с бытовыми терминами, но серьезно отличается от них по смыслу.

Абстрактный тип данных — может быть определён как множество объектов, определяемое списком компонентов (операций, применимых к этим объектам, и их свойств). Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — и в этом тоже заключается суть абстракции. Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями.

Например, есть тип данных *double*. В разных средах выполнения программы внутреннее представление числа в памяти вычислительного устройства отличается. Но это от нас спрятано, нам не надо знать, как оно хранится. Нам достаточно, что этот тип содержит рациональное число, записанное с точностью, которая имеется в описании данного типа.

Двумя основополагающими понятиями ООП являются класс и объект.

Класс — описание того, что содержит и что умеет делать объект. Такое упрощенное определение помогает начинающему осваивать ООП, и представляется наиболее доступным и понятным. Если сформулировать более профессионально, то класс — это абстрактный тип данных, описывающий характеристики и возможные действия некоторой сущности.

Объект — это конкретный представитель класса. Более краткая формулировка: объект — это экземпляр класса.

Хорошей иллюстрацией сути и взаимоотношений этих понятий является следующий пример. Допустим, нам в программе необходимо работать со странами. *Страна* — это абстрактное понятие. У нее есть такие характеристики, как название, население, площадь, флаг и многое другое. Для описания такой страны создается класс с соответствующими полями данных. Такие страны, как Россия и Украина, будут уже объектами (конкретными представителями класса *Страна*).

Динамическая память — участок оперативной памяти ЭВМ, которая предназначена, в частности, для размещения объектов. В среде разработчиков распространено еще одно название этого участка — куча (от англ. *heap*).

Ссылка на объект — переменная типа класса, хранящая физический адрес расположения самого объекта в оперативной памяти. Например, есть объявление переменной:

```
MyClass Obj;
```

Создаем новый объект и сохраняем на него ссылку (!!!) в этой переменной:

```
Obj = new MyClass();
```

В переменной *Obj* хранится адрес оперативной памяти, в котором физически располагается созданный объект. Обратите внимание, объект не расположен в переменной *Objj*. В этой переменной хранится адрес объекта! А сам объект реально расположен в динамической памяти.

Поле — так называются переменные, которые описаны в классе и относятся ко всему классу, ко всем методам в классе. Если переменная описана внутри метода класса, то это не поле, а локальная переменная метода. Все данные класса хранятся только в полях класса. Хороший стиль программирования требует — поля класса не должны быть напрямую доступны пользователю. Доступ к полям реализуется аппаратом свойств или методами.

Свойство — с функциональной точки зрения аппарат, обеспечивающий доступ к полю, путем созданных разработчиком класса произвольных алгоритмов обработки записи и чтения содержимого поля. Иногда свойство называют умным полем. Типичным примером алгоритма обработки записи значения может служить контроль диапазона допустимых значений. Например, блокирование записи в поле отрицательного значения веса человека. Свойство можно использовать и для непосредственного вычисления некоторых величин. Например, в классе круг определено поле радиус. Для вычисления площади круга нет необходимости писать отдельный метод. Это можно реализовать в виде свойства. Различные свойства могут относиться к одному и тому же полю.

Метод — синтаксически оформленный программный модуль, реализующий некоторый алгоритм. Наиболее ранние аналоги метода: функции и процедуры — различались тем, что модуль возвращал (или не возвращал) в качестве результата некоторое конкретное значение. Метод объединил эти различия за счет указания в явном виде:

- типа возвращаемого значения — если метод должен вычислить конкретное значение
- зарезервированного слова *void* — при отсутствии конкретного результата вычисления.

Код в виде методов, принадлежащих классу, может быть отнесен либо к самому классу, либо к экземплярам класса:

- Метод, принадлежащий только классу (**статический метод**) может быть вызван сам по себе без создания объекта. Он имеет доступ только к статическим переменным класса.
- Метод, соотнесенный с экземпляром класса (**обычный/экземплярный метод**), может быть вызван только у самого объекта, и имеет доступ как к статическим полям класса, так и к обычным полям конкретного объекта.

Существуют также операторные методы (см. ниже перегрузка операторов).

ООП базируется на трех основных принципах:

- ***инкапсуляция;***
- ***наследование;***
- ***полиморфизм***.

Инкапсуляция — объединение данных (полей) и алгоритмов их обработки (методов и свойств) в единую синтаксическую конструкцию. Данные хранятся в полях. Алгоритмы обработки реализуются методами и свойствами класса.

ВНИМАНИЕ! В литературе (Интернете) можно часто встретить, что инкапсуляция – это сокрытие (в разных источниках сокрытие разного). Это не совсем корректное утверждение. Сокрытие было и до ООП. А инкапсуляция появилась в ООП. Вспомните, как вы вычисляли, например, синус в ваших работах до того, как узнали о существовании ООП? Вызывали функцию *sin(...)*, тайны реализации которой для вас были сокрыты. Сокрытие появилось с появлением модульной технологии программирования. И никакого ООП тогда не было. Так что секрет инкапсуляции не в сокрытии, а в объединении — «Собрали в капсулу».

Наследование — механизм объектно-ориентированного программирования, позволяющий описать новый класс на основе уже существующего. Класс, на базе которого создается новый класс, называется базовым или родительским. Класс, который наследует базовый класс называется наследником. Все свойства и весь функционал родительского класса заимствуются наследником.

Пример:

Есть базовый класс «Животное». В нем описаны общие характеристики для всех животных (например, вес, возраст, ареал обитания и т.п.). На базе этого класса можно создать классы наследники «Собака», «Слон» и т.д. со своими специфическими свойствами (для собаки: порода, длина шерсти, родословная и т.п. т.е. то, что не характерно для всех остальных животных и, в частности, для слонов).

В этом случае базовый класс «Животное», классы наследники: «Собака» и «Слон».

Полиморфизм — возможность использовать тождественные имена для обозначения разных членов классов, как в рамках одного, так и в рамках нескольких классов. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций». Полиморфизм проявляется в трех видах: перезагрузка методов, тождественные имена в разных классах, виртуальные методы.

В рамках одного класса — возможность использовать перегружаемые методы. Например, в классе имеется вывод каких-либо результатов. Но вывод может осуществляться на разные устройства: на дисплей или на диск. В этом случае создаются два одноименных метода (условно *OutPut()*), различающиеся типом параметра или количеством параметров.

В рамках разных классов — использование одного и того же имени для функционально подобных членов класса. Например, в стандартных визуальных компонентах свойство *Text*. Или, например, есть два класса, «Круг» и «Квадрат». У обоих классов есть метод *GetSquare()*, который вычисляет и возвращает площадь фигуры. Но площадь круга и квадрата вычисляется по разным алгоритмам, соответственно, реализация методов различная, а имена тождественны.

Виртуальные методы. Суть виртуальных методов заключается в том, что имеются несколько различных классов наследников, имеющих тождественный по имени метод. Все классы наследники наследуют родительский класс, имеющий метод с аналогичным именем. Объекты этих классов в произвольном порядке расположены в массиве, имеющим тип базового класса. При выборе любого объекта из массива и вызове этого одноименного

метода на стадии выполнения приложения будет вызван тот метод, который относится к классу, объект которого выбран. Аппарат виртуальных методов очень эффективен при решении некоторого (не очень обширного) класса задач.

Конструктор — метод класса, обеспечивающий создание объекта (экземпляра класса). Единственный возможный способ создания объекта — выполнение конструктора.

Деструктор — метод класса, уничтожающий объект. В разных средах программирования уничтожение производится по-разному. В *C#* оно сводится к тому, что объект помечается как уничтоженный, но память, занимаемая им, не освобождается. Освобождение памяти производится в случае запуска специализированной программы «Сборщик мусора». Этот запуск осуществляет сама среда программирования, вашего участия в ее запуске не требуется.

ВНИМАНИЕ! После выполнения деструктора содержимое ссылки на объект не изменяется. Это надо учитывать при использовании вызова деструктора в явном виде.

Время жизни объекта — время с момента создания объекта конструктором до его уничтожения деструктором.

Инициализация объекта — присвоение начальных значений полям объекта.

Диаграмма классов — графическое представление некоторого количества классов и связей между ними.

Программный интерфейс — перечень (набор) функционала, который предоставляется пользователю класса. Включает описание того, какие аргументы и в каком порядке требуется передавать на вход алгоритмам из этого перечня и в каком виде будут возвращены результаты.

Интерфейс — абстрактный тип данных, созданный для формализованного описания перечня функционала программного интерфейса.

Реализация интерфейса — действительный программный код, который будет выполнять алгоритмы, которые объявлены в интерфейсе.

Агрегирование — методика создания нового класса из уже существующих классов путём включения в поля ссылок на объекты включаемых классов. Об агрегировании также часто говорят, как об «отношении принадлежности» по принципу «у машины есть корпус, колёса и двигатель».

Агрегированный класс — класс, в составе которого есть объекты других классов. Таким образом достигается возможность использования уже реализованных методов другого класса, а также сокрытие реализации этих методов.

Перегрузка операторов — это особая философия ООП, в основе которой лежит понятие операторного метода. Перегрузка операторов по сути является определением оператора (как его функционала, так и синтаксического описания самого оператора) для данного типа абстрактных данных (для объектов данного класса). Например, для численных типов в их классе определен операторный метод "+" и реализован алгоритм алгебраического суммирования. А для строкового типа (*string*) в операторном методе определён тот же символ "+", но реализующий алгоритм конкатенации строк. Чтобы перегрузить (определить) оператор в классе — необходимо описать **операторный метод** [6].

Сериализация – процесс преобразования данных объекта в некую стандартную форму, позволяющую при необходимости в последствии восстановить объект именно с этими данными. По сути это сохранение текущего состояния объекта (мгновенная фотография). Сериализация применяется к свойствам и полям класса. Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом *Serializable*. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом *NonSerialized*. Ярким примером служит запись данных объекта в файл.

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. В .NET можно использовать следующие форматы:

- бинарный
- SOAP
- xml
- JSON

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат - класс *BinaryFormatter*, для формата *SOAP* - класс *SoapFormatter*, для *xml* - *XmlSerializer*, для *json* - *DataContractJsonSerializer*.

Десериализация – построения из потока байтов ранее сохраненного объекта.

Структура (*struct*) – понятие очень схожее с понятием класса. Основным различием классов и структур является тип их экземпляров: экземпляры классов всегда будут являться объектами ссылочного типа, когда экземпляры структур – значимого. Отсюда следует, что экземпляры классов хранятся в динамической памяти, а экземпляры структур – в стеке вызовов. Так же структуры не могут наследовать и наследоваться, но могут реализовывать

интерфейсы. Экземпляры структур можно создавать без конструктора, но тогда все поля структуры будут заполнены стандартными значениями для их типа. Конструктор по умолчанию у структур перегрузить нельзя.

Методические рекомендации для студентов

Внимательно ознакомьтесь с этим разделом и следуйте указаниям ниже. Это СЭКОНОМИТ вам массу времени.

Поймите решаемую задачу

Приступив к выполнению задания, прежде всего ясно и четко поймите и представьте, ЧТО вы хотите сделать. Традиционная ошибка состоит в том, что студент с ходу начинает думать: КАК это сделать, не до конца понимая – ЧТО он должен сделать.

Каждое практическое задание — это необходимость объяснить машине некоторый алгоритм ее действия. Значит, надо сначала самому четко выработать и понять реализуемый алгоритм. Но для того, чтобы выработать алгоритм, надо определить, что ожидается получить. А для этого требуется самому обучаемому досконально понять ставящуюся перед ним задачу. Т.е. фактически в этом случае студент выступает как учитель, который объясняет материал новичку (то есть машине). Успех придет только тогда, когда учитель САМ разберется в материале и будет ясно понимать, что он хочет донести до обучаемого.

Внимательно читайте условия задания

Поверхностное чтение задания в огромном количестве случаев приводит к тому, что студент тратит время на работу, которая от него не требуется. Если вы не поняли, что требуется в задании — **задавайте вопросы преподавателю**. Уточняйте. Обсуждайте. Экономьте свое время!

Предусматривайте при разрабатываемых алгоритмах частные случаи

В подавляющем количестве алгоритмов есть частные случаи. Самые очевидный пример: если вводятся два числа и находится их частное, то деление на ноль недопустимо. Или сумма двух сторон вводимого треугольника не может быть меньше третьей стороны. Или год рождения ныне живущего субъекта не может быть больше текущего года.

Проверяйте входные данные и предусматривайте, по возможности, все особые случаи при программной реализации задания.

Составьте блок-схему

Итак, практическое задание осмыслено и выработан алгоритм его реализации. Если вы не обладаете достаточным опытом кодировки алгоритма на применяемом языке программирования — составьте блок-схему этого алгоритма. Т.е. переложите свое интуитивное представление об алгоритме на его формальное графическое изображение.

Это обеспечит вам комфортную возможность не держать при кодировке в голове всех необходимые связи, которых даже в алгоритмах средней сложности достаточно много.

Не пишите сразу весь код

Создавайте код от простого к сложному. Реализуйте сначала простейшие алгоритмы или их фрагменты (в зависимости от сложности задания) и отладьте их. На основе уже отлаженных добавляйте новые фрагменты кода и отлаживайте их. При таком подходе легче обнаруживаются ошибки или неточности в тех новых фрагментах кода, которые вы добавили. Иначе, чтобы найти эти ошибки, вам приходится тратить время и силы на локализацию мест, где проявляется ошибка.

Последовательность разработки класса

Придерживайтесь следующей последовательности действий при написании нового класса:

- Присвойте разрабатываемому классу некоторое мнемоничное имя и запишите синтаксически правильный пустой класс:

```
public class <имя класса>
{
}
```

- Определите минимально необходимые поля разрабатываемого класса.
  Поля являются единственным хранилищем данных, которые будут обрабатываться методами класса. Методы класса, в подавляющем большинстве случаев, содержат алгоритмы обработки информации, хранящейся в полях класса. Прежде чем реализовывать методы, необходимо определить те величины, которые будут обрабатываться алгоритмами методов.
- Если на этом первом этапе определены не все требуемые поля – их можно будет добавить в любой момент разработки.
- Создайте, если считаете нужным, конструктор: с параметрами, обеспечивающий ввод данных в поля; определяющий (инициирующий) начальные значения данных полей;
- Создайте метод(ы) или свойства, обеспечивающие доступ к данным, с которыми будет взаимодействовать пользователь класса.
- Начинайте реализовывать необходимые для решения поставленной задачи методы, начиная от простых и двигаясь к все более сложным. Не забывайте при этом отлаживать автономно создаваемые методы.

Пользуйтесь отладчиком

Отладчик (*Debuger*) является простым и мощным средством нахождения ошибок кодирования либо логики работы реализованного алгоритма. Его основные возможности просты (возможность пошагового выполнения программы и визуализация значений переменных), но очень эффективны. Овладение этим инструментом занимает несколько минут [3].

Одна из особенностей сложности поиска ошибок реализации состоит в том, что человеку свойственно видеть то, что он предполагает, а не то, что написано. Практически каждый, кто писал программы, сталкивался с этим. Т.е., например, нужно записать в *I*-ый элемент массива *Arr* некоторое значение (*Arr[I] = 5*). А в тексте программы написано *Arr[1] = 5*. Как показывает практика, очень немногие способны быстро увидеть эту ошибку при просмотре достаточно большого участка кода. А используя отладчик и выполнив этот оператор в пошаговом режиме сразу визуально станет видно, что *Arr[I]*не изменилось при *I* не равном *1*.

Помните, что программа делает то, что вы от нее просите,

а не то, что вы от нее хотите!

Практические задания

Класс «Квадратное уравнение»

Цель

Практическое освоение новых технологических приемов ООП: класс, объект, поле, конструктор, метод, перечислимые типы[1].

Практическое задание

Необходимо создать пользовательский класс, объекты которого хранят коэффициенты квадратного уравнения и способны вычислять его корни. Разработать тестирующее приложение.

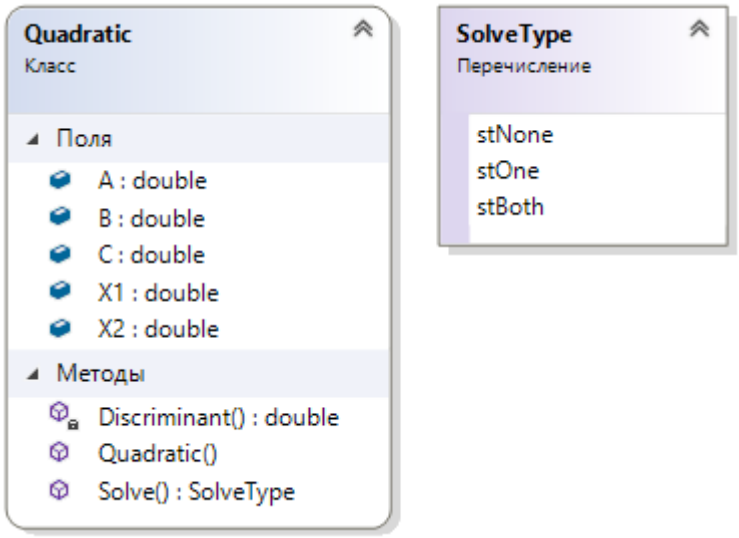

Рисунок 1. UML-диаграмма класса Quadratic и перечислимого типа SolveType

Поля:

A, B, C — числовые поля для хранения параметров уравнения;

X1, X2 — числовые поля для хранения корней уравнения.

Методы:

Quadratic() — конструктор по умолчанию, присваивает всем полям нулевые значения.

Discriminant() — метод, вычисляющий и возвращающий дискриминант квадратного уравнения. Дискриминант может возвращать объект перечислимого типа.

[1] Создание перечислимого типа – это дополнительное (необязательное) задание в рамках реализации класса «Квадратное уравнение». Без этого перечислимого типа метод Solve() может возвращать целое число – количество решений уравнения: 0, 1 или 2, соответственно.

Solve() — метод, вычисляющий корни квадратного уравнения. В зависимости от значения дискриминанта метод возвращает *stNone*, *stOne* или *stBoth*1, либо 0, 1 или 2, если перечислимый тип не применяется.

Применение класса:

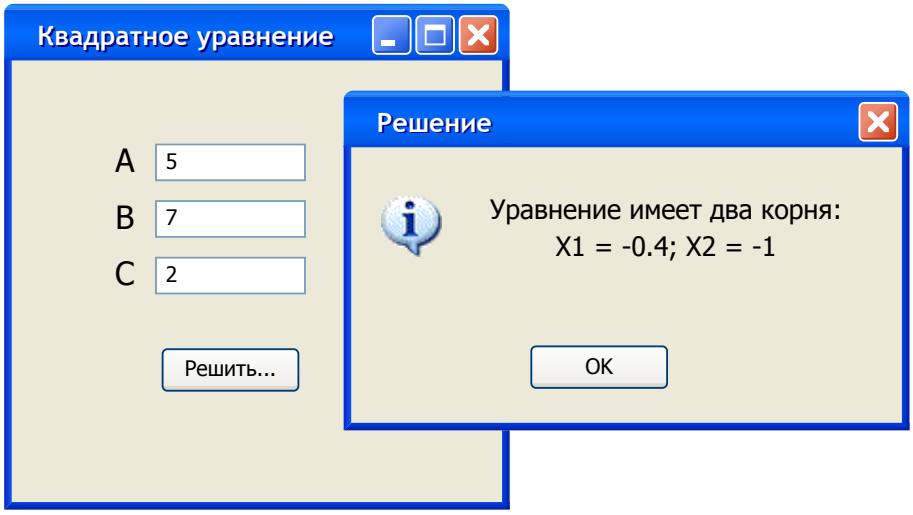

Рисунок 2. Применение класса Quadratic

В компоненты *TextBox* на форме вводятся значения полей – параметров уравнения. На кнопке «Решить» создается объект типа *Quadratic*, в поля которого присваиваются значения параметров. Далее вычисляются корни путем вызова метода *Solve()*. В зависимости от типа, возвращаемого методом *Solve()*, выводится окно сообщения (сколько корней и чему они равны).

Вопросы для самоконтроля

1. Что такое тип данных? Чем отличаются встроенные типы данных от пользовательских?
2. Что такое класс? Что такое объект класса? Чем класс отличается от объекта?
3. Объясните понятия «поле класса», «метод класса», «конструктор». Что они хранят, что делают, каким образом инициализируются?
4. Что такое «перечисление»[1] (перечислимый тип)? Как объявить и инициализировать объект перечислимого типа? Какие значения могут принимать объекты перечислимых типов?

Класс «Рациональное, представленное двумя целыми»

Цель задания

Практическое закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод.

Практическое освоение новых технологических приемов ООП: перегрузка методов, вызов методов класса из других методов класса, сокрытие вспомогательных методов, создание в методе класса новых объектов, класс как тип возвращаемого метода значения, возврат методом объекта, перегрузка операторов.

Задание рассчитано на выполнение в несколько этапов. Каждый этап преследуют освоение некоторых описанных в цели технологических средств ООП в порядке возрастания их сложности.

Практическое задание

В математике доказано, что любое рациональное число (*MF*) может быть представлено в виде дроби, знаменатель (*num*) и числитель (*den*) которой являются целыми: *MF = num / den.*

Необходимо создать класс, обеспечивающий:

- хранение рационального числа в виде двух целых, трактуемых как числитель и знаменатель дроби;
- реализующий возможность выполнения основных арифметических операций над числами, хранимыми в объектах этого класса. При этом результат выполнения арифметической операции сохраняется так же в объекте разрабатываемого класса.

Этап 1 Создание полей и двух методов

Необходимо разработать класс *MyFloat* (название условное, если предпочитаете другое, то предоставляется полная свобода действий), реализующий указанное выше представление рационального числа. Класс должен содержать:

- поля для хранения значений числителя и знаменателя;
- метод ввода значения числителя и знаменателя;
- метод визуализации хранимого рационального числа в виде десятичной дроби.

Создать программное приложение, которое используя класс *MyFloat*, обеспечивает:

- создание объекта класса *MyFloat*;
- ввод в созданный объект данных: значений числителя и знаменателя;
- хранение введенного числа;
- визуализацию рационального значения хранимого числа;
- сохранение числа в виде несокращаемой дроби путем создания метода нахождения наибольшего общего делителя и использования результата этого метода при заполнении значений полей.

Методические указания

В создаваемом классе, согласно заданию, не должны быть поля, хранящие заданное пользователем число в рациональном виде.

Желательно, чтобы метод визуализации имел тип возвращаемого значения *string*, т.к. практически все стандартные визуальные компоненты имеют такой тип для данных, которые планируется визуализировать.

Метод нахождения наибольшего общего делителя — вспомогательный метод класса и поэтому он не должен быть виден внешнему пользователю класса.

Этап 2. Расширить функционал класса созданием конструктора с параметрами

Необходимо:

- Добавить в описание класса конструктор с параметрами Параметры — значения числителя и знаменателя.
- В программном приложении использовать при создании объекта конструктор с параметрами.

Методические указания

Если вы создали ранее конструктор с параметрами, то создайте метод занесения значений числителя и знаменателя. Это обеспечит возможность внесение в существующий уже объект иного рационального числа без необходимости создания нового объекта.

Этап 3. Сложение двух чисел

Реализовать метод, позволяющий складывать два числа, хранимых в двух объектах класса, и их сумму поместить в новый объект этого же класса. В качестве формального параметра этого метода передавайте ссылку на объект, содержащий второе слагаемое.

При реализации расширьте пользовательский интерфейс возможностью ввода второго числа (слагаемого).

Методические указания

Не создавайте каждый раз по нажатию кнопки новые объекты, сохраняя ссылки на них в одно и тоже поле класса *Form1*. Это приводит к бесполезному расходованию ресурса как памяти, так и времени выполнения приложения. Конечно, в данном случае эти потери мизерны и не существенны. Но при создании больших приложений это может привести к значительному снижению скорости выполнения приложения.

Объявите ссылки на создаваемые объекты разрабатываемого класса как поля класса *Form1*. И при необходимости ввода новых данных проверяйте значения этих полей на *null*. Если они содержат *null*, то объект еще не создан и тогда надо его создать. В ином случае достаточно вызвать метод помещения данных в существующий объект.

Для реализации метода сложения используйте следующие возможности ООП:

- тип возвращаемого методом значения может быть ссылка на имя класса (класс — это равноправный тип данных);
- типы данных, передаваемых в формальных параметрах метода, так же могут ссылками на класс. При передаче фактических параметров в соответствующей позиции должна передаваться ссылка на объект данного класса;
- в методе класса можно создавать при необходимости новые объекты как данного класса, так и любого другого определенного в области видимости.

Обращение в главной программе к методу (условное имя метода *Add*) должно выглядеть следующим образом: *ObSum = Ob1.Add(Ob2);* где *ObSum* — объект, содержащий сумму чисел, хранимых в объектах *Ob1* и *Ob2*. Все эти объекты принадлежат разрабатываемому классу.

Этап 4. Реализация остальных арифметических действий

Реализуйте реализующие остальные три арифметических действия (вычитание, умножение и деление). Способ реализации подобен тому, как реализована на предыдущем этапе операция сложения.

Методические указания

Выполнение этого этапа позволит закрепить типичные приемы программной разработки в технологии ООП, которые освоены при выполнении предыдущего этапа.

Этап 5. Создание операторных методов

Реализуйте четыре основных арифметических действия над числами, хранимыми в разработанном классе, с сохранением результата в объекте этого же класса. Но при этом используйте перегрузку операторов (операторные методы).

Методические указания

Обратите внимание на то, что после выполнения этапа 4 разработанный наш класс способен выполнять вычисление алгебраических выражений в отличной от обычного синтаксиса форме. Например, нужно вычислить достаточно несложную формулу:

R = a + d*c – d/f

Допустим, что значения *a, b, c, d, f* содержаться в объектах разрабатываемого класса *Ob1, Ob2, Ob3, Ob4, Ob5* соответственно. Реализация вычисление указанного выше выражения методами класса *Add, Sub, Mult, Div* (реализующих соответственно сложение, вычитание, умножение и деление), приведет к такой записи вычислений:

```
ObTempMult = Ob2.Mult(Ob3);
ObTempDiv = Ob4.Mult(Ob5);
ObTempSub = ObTempMult.Sub(ObTempDiv);
ObResult = Ob1.Add(ObTempSub);
```

Наглядность такого способа вычисления и количество используемых операторов пробуждают желание записать это проще и понятнее.

После реализации Этапа 5, запись арифметических действий может быть выполнена в естественном для нас виде, а именно:

```
ObResult = Ob1 + Ob2*Ob3 – Ob4/Ob5;
```

Вопросы для самоконтроля

1. Дайте определение понятия ООП класс.
2. Дайте определение понятия ООП объект.
3. Что имеется в виду под термином ООП поле?
4. Дайте определение термина ООП метод.
5. Какое функциональное назначение каждого из выше приведенных понятий ООП?
6. Что такое конструктор?
7. Каково функциональное назначение конструктора?
8. Чем отличается конструктор от метода по сути?
9. Чем отличается конструктор от метода по синтаксису?

10. Чем обусловлено синтаксическое отличие конструктора от синтаксиса?
11. Конструктор по завершению выполнения возвращает какое-либо значение?

Класс «Комплексное число»

Цель 1

Практическое закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод.

Практическое освоение новых технологических приемов ООП: перегрузка методов, создание в методе класса новых объектов, возврат методом объекта, перегрузка операторов, работа с объектами перечислимых типов.

Практическое задание

Необходимо создать пользовательский числовой тип данных (Рисунок 3), обеспечивающий работу с комплексными числами: выполнение арифметических и логических операций с объектами этого типа. Обеспечить возможность использования объектов создаваемого типа данных в арифметических или логических выражениях:

```
Complex c0 = new Complex();
Complex c1 = new Complex();
Complex c2 = new Complex();
Complex c3 = c1 + c2 – c0;
if (c1 < c2)
{
}
```

Реализовать тестирующее приложение для класса комплексных чисел.

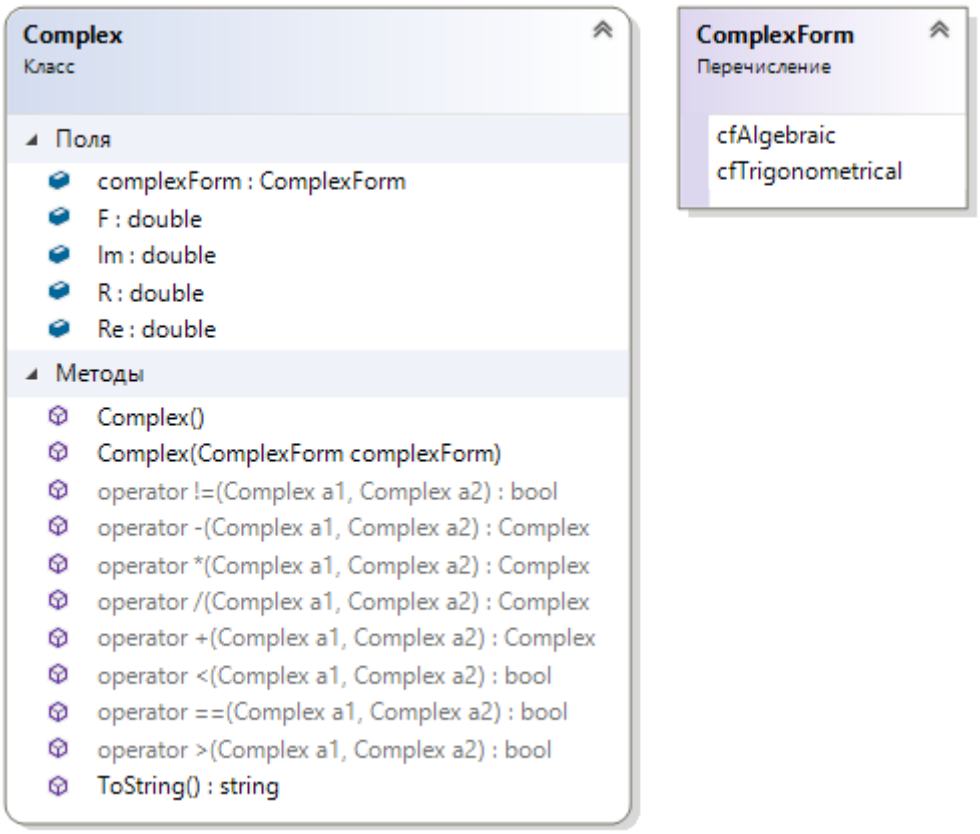

Рисунок 3. UML-диаграмма класса Complex и перечислимого типа ComplexForm

Поля:

complexForm — форма комплексного числа задается перечислимым типом ComplexForm (алгебраическая или тригонометрическая, задается при создании объекта, доступна только для чтения);

Re, Im — числовые поля для алгебраической формы (Re+Imi);

R, F — числовые поля для тригонометрической формы (r(cos(f)+sin(f)i).

Методы:

Complex() — конструктор по умолчанию, присваивает всем полям нулевые значения, форма комплексного числа по умолчанию – алгебраическая.

Complex(ComplexForm form) — конструктор с параметром, задающим форму числа.

operator +(Complex a1, Complex a2) — статические (*static* – неэкземплярные методы класса) методы, перегружающие операции сложения, вычитания, умножения и деления комплексных чисел – создают и возвращают (в качестве возвращаемого значения) комплексные числа в той форме, в которой создан первый передаваемый параметр (первое слагаемое, уменьшаемое, первый множитель и делитель соответственно).

operator == (Complex a1, Complex a2) — методы (тоже статические), перегружающие операции, определяющие правила сравнения двух чисел, возвращает *true* или *false*.

ToString() — метод перевода комплексного числа в строку – алгебраической и тригонометрической формы.

Применение класса:

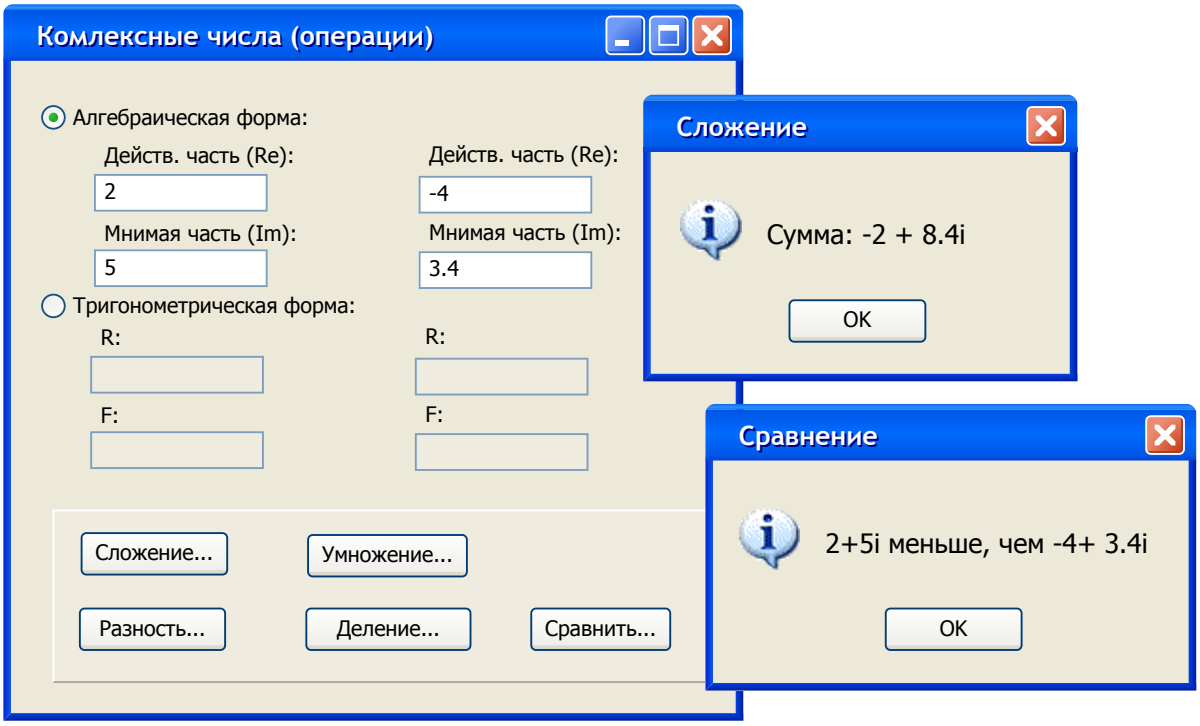

Рисунок 4. Применение класса Complex

Выбирается форма (Рисунок 4) задания комплексного числа (алгебраическая или тригонометрическая). Далее вводятся параметры двух комплексных чисел. Кнопки: «Сложение», «Разность», «Умножение», «Деление» и «Сравнить» для соответствующих операций. Результат выводится в окно сообщений в той форме, в которой числа были заданы изначально.

Для преобразования комплексного числа в строку необходимо использовать метод *ToString()*.

Вопросы для самоконтроля

1. Что такое тип данных? Чем отличаются встроенные типы данных от пользовательских?
2. Что такое класс? Что такое объект класса? Чем класс отличается от объекта?
3. Объясните понятия «поле класса», «метод класса», «конструктор». Что они хранят, что делают, каким образом инициализируются?

4. Что такое «перечисление»[1] (перечислимый тип)? Как объявить и инициализировать объект перечислимого типа? Какие значения могут принимать объекты перечислимых типов?
5. Для чего нужна перегрузка операций? Возможно ли перегрузить арифметическую операцию «+», но не перегружать арифметическую операцию «–»? Возможно ли перегрузить логическую операцию «больше», но не перегружать логическую операцию «меньше»?
6. Для чего нужен метод *ToString()* и что он возвращает?

Класс «Вектор»

Цель

Практическое закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод, перегрузка методов, создание в методе класса новых объектов, возврат объекта из метода, перегрузка операторов.

Практическое освоение новых технологических приемов ООП: создание и применение индексных свойств.

Практическое задание

Создать пользовательский тип данных *Vector* (Рисунок 5), обеспечивающий работу с объектами-векторами: инициализация векторов начальными значениями, сложение и умножение векторов, умножение вектора на число, вывод вектора в виде строки.

Обеспечить возможность непосредственной индексации объекта – ввести определение индексируемого свойства в класс:

пример 1 вектора без индексных свойств –

```
Vector v = new Vector(5);
v.Values[0] = 10;
```

пример 2 вектора с индексным свойством –

```
Vector v = new Vector(5);
v[0] = 10;
```

В обоих примерах *Values* по своей сути массив. Но в первом примере этот массив должен иметь область видимости *public* (или в крайнем случае *protected*, если массив расположен в родительском классе*)*. А это противоречит одному из основных принципов ООП - защиты данных. Использование индексатора во втором примере не только сохраняет приверженность к отмеченному принципу ООП, но и повышает наглядность и читабельность кода. А это очень важная компонента качества программного продукта.

Реализовать приложение для осуществления операций с представленными в векторном виде данными.

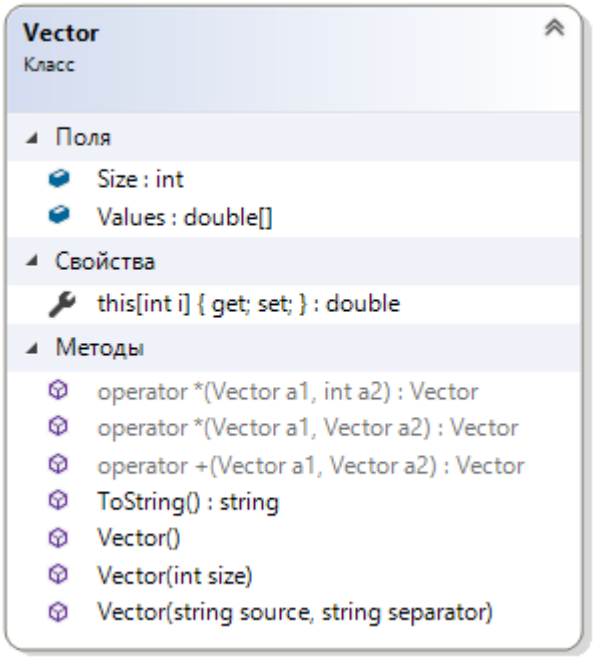

Рисунок 5. UML-диаграмма класса Vector

Поля и свойства:

Size — размер вектора. Задается либо в конструкторе, либо непосредственно присваивается;

Values — числовой массив для хранения элементов вектора.

this[int i] — индексированное свойство (индексатор).

Методы:

Vector(int size) — конструктор с параметром (длина вектора). Этот конструктор должен создать массив Values, заданной длины.

Vector(string source, string separator) — конструктор с параметрами: строка, содержащая элементы вектора, и строка-разделитель (например, запятая, пробел и т.д.). Этот конструктор должен разобрать передаваемую строку *source* на составляющие, где разделителем является строка *separator*. Далее посчитать, сколько составляющих получилось. Создать массив *Values* соответствующей длины и поместить в него все элементы, полученные из строки *source*.

operator(Vector a1, Vector a2) — статические методы, перегружающие операции сложения векторов, умножения векторов и умножения вектора на число – создают и возвращают (в качестве возвращаемого значения) результирующий вектор.

ToString() — метод перевода вектора в строку вида (1, 2, 3, 4, …).

Применение класса:

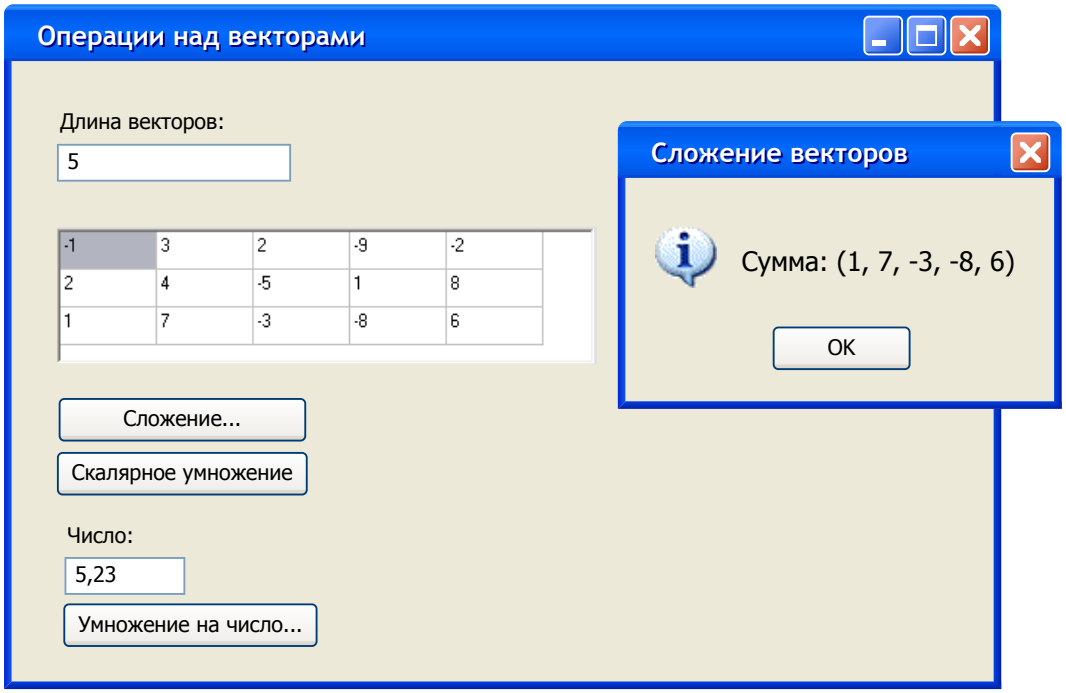

Рисунок 6. Применение класса Vector

Сначала вводится длина двух векторов (Рисунок 6). Потом в компоненте *DataGridView* поэлементно вводятся значения обоих векторов. Строка компонента *DataGridView* имеет тип *Rows* (набор строк), поэтому удобнее переписывать в цикле все значения из *DataGridView* в объект-вектор и обратно.

Либо, если с *DataGridView* нет желания работать, можно вводить в *TextBox* строку, в которой элементы разделены пробелом или запятыми. И далее применять конструктор *Vector(string source, string separator)* для создания объекта-вектора.

Кнопки: «Сложение», «Скалярное умножение» и «Умножение на число» для соответствующих операций сложения, умножения векторов и умножения вектора на число. Для умножения вектора на число можно использовать первый вектор (само число вводится дополнительно).

Результат операций отображается в окошках сообщений (для отображения использовать метод перевода вектора в строку). Также результат можно отобразить на *DataGridView* (дополнительной строкой), используя метод *ToString()*.

Вопросы для самоконтроля

1. Что такое класс? Что такое объект класса? Чем класс отличается от объекта?
2. Объясните понятия «поле класса», «метод класса», «конструктор». Что они хранят, что делают, каким образом инициализируются?

3. Зачем классу могут понадобиться несколько конструкторов? Должны ли эти конструкторы чем-либо отличаться? Если да, то чем?
4. Для чего нужны индексные свойства? Какие возможности они предоставляют?

Класс «Матрица»

Цель задания

Практическое закрепление навыков реализации базовых понятий и приемов создания программного обеспечения в ООП: класс, объект, поле, конструктор, метод, перегрузка методов, вызов методов класса из других методов класса, сокрытие вспомогательных методов, создание в методе класса новых объектов, класс как тип возвращаемого метода значения, возврат методом объекта, перегрузка операторов.

Задание рассчитано на выполнение в несколько этапов. Каждый этап преследуют закрепление и освоение некоторых описанных в цели технологических средств ООП в порядке возрастания их сложности.

Практическое задание

Матричная алгебра или матричное исчисление — раздел математики, посвященный работе с матрицами — один из самых важных, употребительных и содержательных понятий в математике. С помощью матричного аппарата легко и удобно производить различные действия при решении задач линейной алгебры, системного анализа, теории управления, экономики, статистики и других областей науки и знания. Обычно это реализуется решением системы линейных уравнений, векторными и линейными преобразованиями и т.д. Сейчас даже трудно себе представить области, где бы не применялись матричные методы при решении различных задач.

Большинство из вас использует операции с матрицами при практическом решении задач в ваших учебных курсах. Вполне возможно будет использовать их и в трудовой вашей деятельности. Поэтому результат выполнения данного задания полезен для вас и в чисто прикладном аспекте.

Итак, необходимо создать класс, обеспечивающий:

- возможность создания объекта, хранящего квадратную матрицу заданного пользователем размера;
- визуализацию матрицы;
- заполнение ее различными способами;
- реализацию матричных операций.

Этап 1. Создание полей класса и двух методов класса

Необходимо разработать класс *MyMatrix* (название условное), обеспечивающий:

- создание квадратной матрицы, для простоты реализации хранящей целые числа;
- имеющей указываемую пользователем размерность;
- заполняющий матрицу случайными значениями (с целью отладки класса);
- визуализацию матрицы.

Класс должен содержать:

- поле для хранения элементов матрицы;
- конструктор с параметром, задающим размерность матрицы;
- метод заполнения матрицы случайными значениями;
- метод визуализации содержимого матрицы.

Создать программное приложение, которое, используя класс *MyMatrix*, обеспечивает:

- создание объекта класса *MyMatrix* с указанием размерности матрицы;
- вызов метода заполнения случайными числами;
- визуализацию содержимого матрицы.

Методические указания

Хранение значений матрицы организовать с помощью двумерного динамического массива. Размерность массива задается при вызове конструктора класса, путем передачи параметра. Конструктор выполняется при нажатии кнопки «Создать матрицу». Определение значения размерности создаваемой матрицы возможно вводом из соответствующего *TextBox* на форме приложения либо непосредственно заданного в теле программы. Вызов метода заполнения случайными числами реализуется по нажатию кнопки «Заполнить случайно». Визуализация содержимого матрицы реализуется по нажатии кнопки «Визуализировать матрицу».

Естественным является вопрос: «А как ее визуализировать?» Часто первым предложением от студентов является «Создать много *TextBox-ов*. В каждом из них будет выводится отдельное значение матрицы». Вряд ли это хороший подход. Тем более, обращаем внимание: визуальная компонента *TextBox* предназначена для ввода данных. Для вывода представления данных создана визуальная компонента *Label*.

Наиболее продуктивным для визуализации матрицы является использование стандартной визуальной компоненты *DataGridView*. Это мощное гибкое средство, овладев

которым вы будете использовать его регулярно и при выполнении других учебных заданий. По сути, *DataGridView* это визуальная таблица, располагаемая на форме. Но это очень умная таблица. Она умеет не только осуществлять вывод данных. С ее помощью можно вводить данные. Можно привязывать произвольные объекты к ячейкам. Но на данном этапе достаточно овладеть только её возможностями визуализации данных. Освойте и используйте эти возможности используя справочные материалы, расположенные в Интернете.

Этап 2. Создание методов занесения и чтения значений в матрицу

Используйте для этого как методы, так и возможность ввода данных в матрицу непосредственно при помощи визуального компонента *DataGridView*.

Необходимо: в классе *MyMatrix* разработать функционал обеспечивающий возможность:

- заполнение матрицы заданными в программе значениями;
- извлечение значений, содержащихся в матрице;
- овладеть возможностью ввода значений с использованием *DataGridView*.

Методические указания

При практическом решении конкретной задачи, использующей матричную алгебру, участвующие в ней матрицы заполняются конкретными значениями. Необходимо создать аппарат, обеспечивающий это заполнение.

Поставленная задача может решаться в нашем проекте двумя способами. Вам требуется реализовать оба подхода.

Первый из них универсальный — как только нужно расширить функционал любого класса – надо создать метод. Матрица это прямоугольная (в данном проекте частный случай – квадратная) таблица, объединяющая некоторую совокупность чисел. Каждое значение, хранимое в матрице, располагается по уникальному адресу: номер строки и номер столбца. Если создать метод, который в указанную строку и в указанный столбец заносит конкретное значение, то задача заполнения матрицы, требующимися для решения любой задачи, решена. Значит надо создать такой метод — и аппарат заполнения матрицы создан! Не забудьте контролировать значения входных данных, а именно: передаваемые номер строки и столбца. Некорректно заданные значения могут привести к неверной работе приложения или даже к его отказу.

Следует отметить универсальность описанного аппарата. С помощью него можно заполнять матрицу данными, хранящимися в файлах во внешней памяти, получаемыми в результате программных вычислений, передающимися по сети и т.д.

Аналогично создается метод чтения данных, содержащихся в указанных строках и столбцах.

Второй подход заполнения матрицы требуемыми данными состоит в считывании их непосредственно из компонента *DataGridView*. Как сказано выше, компонент имеет возможность изменения (ввода) значений в произвольную ячейку его сетки. Поэтому создайте метод, входным параметром которого является ссылка на объект класса *DataGridView*, и внутри этого метода реализуйте последовательное считывание данных из *DataGridView* и занесение этих данных в матрицу. Используйте при этом созданный ранее метод занесения значений в указанные строку и столбец. С возможностью *DataGridView* изменять пользователем содержимое ячеек ознакомьтесь в Интернете и освойте его практически.

Обращаем внимание, что этот подход менее универсален. Он требует наличия в приложении компоненты *DataGridView*. Но более удобен в нашем проекте. Это наглядная иллюстрация того, что универсальность это, как известно, хорошо. Но она не учитывает специфику конкретного случая. И наличие универсальных средств ни в коей мере не отрицает возможности существования менее универсальных, но более эффективных в конкретном случае.

Метод вызывается по нажатию кнопки «Ввести данные из таблицы». Обращаем внимание, визуальный интерфейс должен быть рассчитан на пользователя. Пользователь не должен знать, что такое *DataGridView*. Избегайте в интерфейсе текстов, отражающих специфику программной реализации. Пользуйтесь терминологией из области решаемой задачи. Поэтому название данной кнопки типа «Ввести данные из *DataGridView*» – признак низкой квалификации разработчика.

Этап 3. Реализация матричных операций

Необходимо: расширить функционал класс *MyMatrix* реализовав:

- операцию сложения двух матриц;
- операцию умножения двух матриц;
- операцию транспонирования матрицы.

Методические указания

Операции реализовать с помощью использования операторных методов (перегрузки операторов).

Обычно в литературе иллюстрируются примеры создания операторных методов для операций с двумя операндами (бинарные операции). К ним относится привычные нам четыре основных арифметических действия (сложение, вычитание, умножение и деление). В этом смысле операции сложение и умножение матриц – именно такие операции.

Затруднение часто возникает у обучаемых при использовании операторного метода с одним параметром. Операция транспонирование матрицы – унарная операция (содержит один операнд). Но и здесь нет ничего сложного. В операторном методе просто записывается один параметр, а именно ссылка на сам объект – транспонируемую матрицу. А в качестве возвращаемого значения – ссылка на объект матрица. В качестве символа операции, например, можно использовать сивол «!» или другой, который допустим и вам нравится.

Для отладки и проверки методов обеспечить на форме возможность определения, как минимум, второй матрицы и визуализацию результатов операций в третью матрицу.

Проект «Матрица на основе класса Вектор»

Цель

Практическое закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод, перегрузка методов, создание в методе класса новых объектов, возврат объекта из метода, перегрузка операторов, создание и применение индексаторов.

Практическое освоение технологических приемов ООП: создание и применение свойств, доступных только для чтения, сокрытие полей класса.

Практическое задание

Создать пользовательский тип данных *Matrix* (Рисунок 7), обеспечивающий работу с объектами-матрицами: создание объектов, заполнение их числами, умножение матриц друг на друга, умножение матрицы на число. Свойства *Columns* и *Rows* должны представлять собой массивы векторов[2] и быть доступными только для чтения. Свойства *Height* и *Width* инкапсулируют поля класса *height* и *width*, соответственно, и доступны только для чтения.

Так же необходимо объявить индексное свойство:

пример матрицы без индексных свойств –

```
Matrix m = new Matrix(5, 5);
m.Values[1,1] = 123;
```

пример матрицы с индексным свойством –

```
Matrix m = new Matrix(5, 5);
m[1,1] = 123;
```

Для тестирования свойств и методов класса разработать проект матричный калькулятор.

[2] Тип данных *Vector* можно взять в разрабатываемый проект из задания

Класс «Вектор».

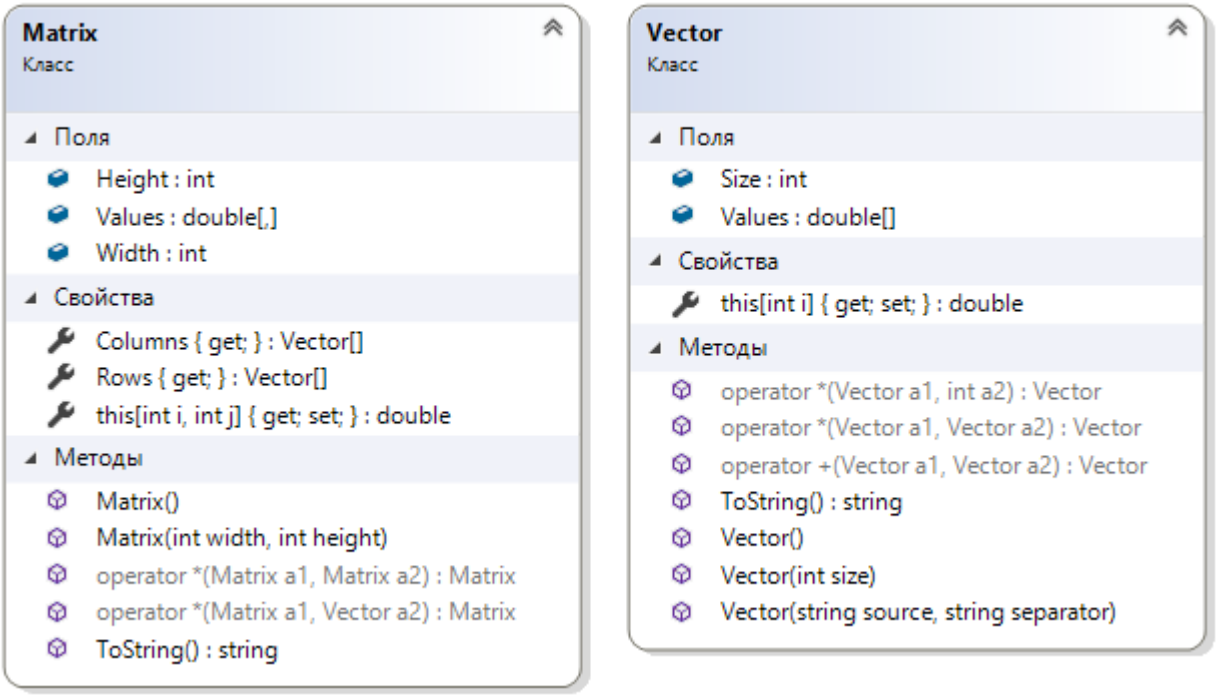

Рисунок 7.UML-диаграмма класса Matrix и класса Vector

Поля и свойства:

Width и *Height* — свойства, определяющие размер матрицы. Значения их задаются строго конструктором и свойства доступны только для чтения. Инкапсулируют приватные поля *width* и *height*, которые используются непосредственно в классе.

Values — двумерный числовой массив для хранения элементов матрицы. Индексное свойство, осуществляющее доступ к закрытому массиву элементов матрицы (доступно для чтения и записи).

Columns и *Rows* — количество колонок и строк матрицы (размерности двумерного массива матрицы). Доступны только для чтения.

this[int i, int j] — индексатор.

Методы:

Matrix() — конструктор по умолчанию. Создает двумерный массив *Values*, размером 3x3 и заполняет его нулями.

Matrix(int width, int height) — конструктор с параметрами, обозначающими ширину и высоту матрицы. Создает двумерный массив *Values* заданного размера и заполняет его нулями.

*operator *(Matrix a1, Matrix a2)* и *operator *(Matrix a1, float F)* — два перегруженных оператора умножения матриц: один умножает матрицу на матрицу, второй – матрицу на число. Возвращают результирующую матрицу.

ToString() — метод перевода матрицы в строку вида: ((A1,1, A1,2, ..., A1,n) (A2,1, A2,2, ..., A2,n) ... (An,1, An,2, …, An,n))

Применение класса:

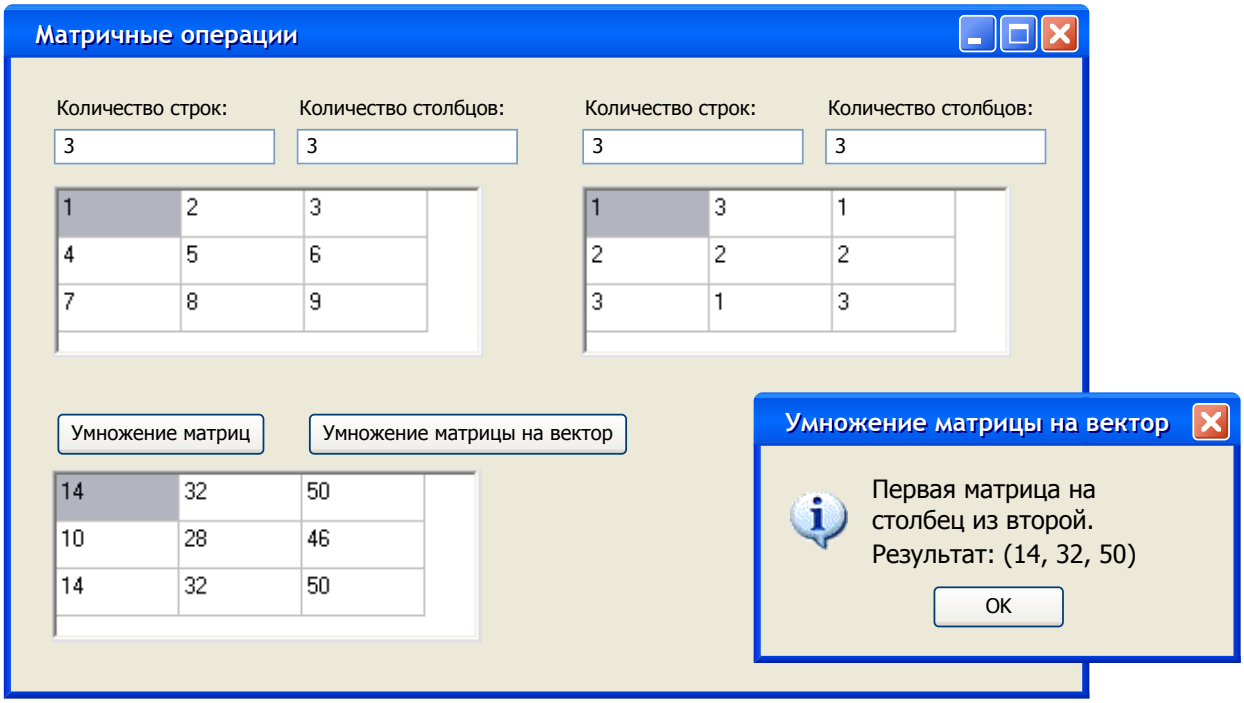

Рисунок 8. Применение класса Matrix

Вначале вводятся размеры двух матриц (Рисунок 8). В компонентах *DataGridView* вводятся элементы обеих матриц. Результирующая матрица (умножение) выводится в третий *DataGridView*. Результат от умножения матрицы (можно взять первую, к примеру) на вектор (например, первая строка/столбец из второй) выводится в окно сообщений или на *TextBox*.

Вопросы для самоконтроля

1. Что такое класс? Что такое объект класса? Чем класс отличается от объекта?
2. Объясните понятия «поле класса», «метод класса», «конструктор». Что они хранят, что делают, каким образом инициализируются?
3. Для чего необходимо скрывать (инкапсулировать) поля?
4. Для чего нужны свойства, доступные только для чтения? Как они объявляются?

Проект «Любопытные»

Цель задания

Практическое освоение технологических приемов ООП: наследование, перекрытие членов класса, вызов из класса методов другого класса, сокрытие вспомогательных методов, использование свойств, обращение к одноименному члену родительского класса из класса наследника.

Задание рассчитано на выполнение в несколько этапов. Каждый этап преследует освоение некоторых описанных в цели задания технологических средств ООП в порядке возрастания их сложности.

Практическое задание

Создать класс ***Curious***, на котором расположено лицо с двумя глазами и с зрачками, следящими за движением мышки.

На основании этого класса создать класс ***Onlookers***, который объединяет произвольное количество лиц, синхронно следящими за движением мышки. **Ошибка! Источник ссылки не найден.** схематично иллюстрирует пример визуализации объекта класса ***Onlookers***.

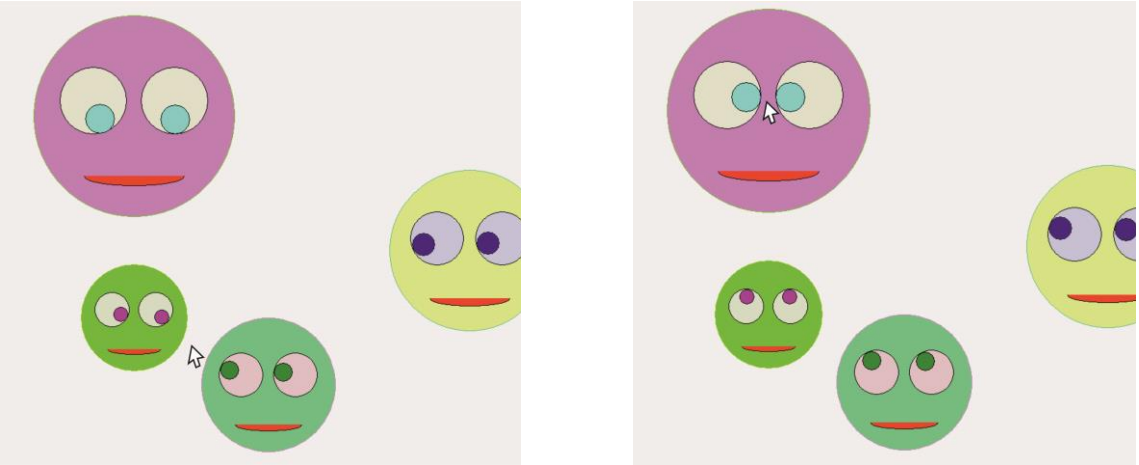

Рисунок 9 Пример визуализации класса ***Onlookers***

Методические рекомендации

Задание состоит из тех этапов:

1. Создание класса ***MyCircle.***
2. Создание класса ***MyEye***.

3. Создание класса ***Onlookers***.

Как видно из выше приведенного рисунка базовым элементом класса ***MyEye*** является круг. Соответственно имеет смысл разработать класс *Круг* и комбинируя наследование и агрегацию объектов этого класса существенно облегчить задачу разработки класса ***MyEye***.

Этап 1. Создание класса *MyCircle*.

Элемент *круг* встречается в постановке и решении многих задач. Даже в рамках этого пособия он используется и в других заданиях. Поэтому требования и рекомендации по его реализации описаны в первом этапе задания " Железнодорожный состав".

Этап 2. Создание класса *MyEye*

Разработать класс ***MyEye***, имитирующий глаз, состоящий из глазницы и зрачка. Обе эти сущности по сути экземпляры объектов ранее разработанного класса ***MyCircle***.

Параметрами класса ***MyEye*** являются:

- Координаты точки центра глаза ***X0*** и ***Y0***.
- Радиус глаза ***Reye***.

Все остальные необходимые величины рассчитываются относительно этих параметров глаза.

Значения параметров объекта глазница фиксированы, и по сути и являются задаваемыми значениями ***X0***, ***Y0***, ***Reye***.

Параметры объекта зрачок изменяются в зависимости от положения курсора мышки. Следовательно, объект глаз должен иметь метод, получающий текущее значение координат мыши и, в соответствии с этим, изменяющий значения параметров объекта зрачок.

Методические указания

Вычисление текущих значений параметров объекта зрачка может производится следующим образом (**Ошибка! Источник ссылки не найден.**). Текущие координаты мышки обозначены ***Xm***, ***Ym***. Параметры класса глазница – ***Rg***, ***Xg***, ***Yg*** (радиус и координаты центра соответственно). Параметры зрачка – ***Rz***, ***Xz***, ***Yz***.

Определим вспомогательную окружность - орбиту движения центра зрачка радиуса Ro относительно центра глазницы.

Тогда при заданных значениях Xm, Ym искомые значения Xz, Yz вычисляются как точка пересечения прямой (Xm, Ym)–(Xg, Yg) с орбитой зрачка.

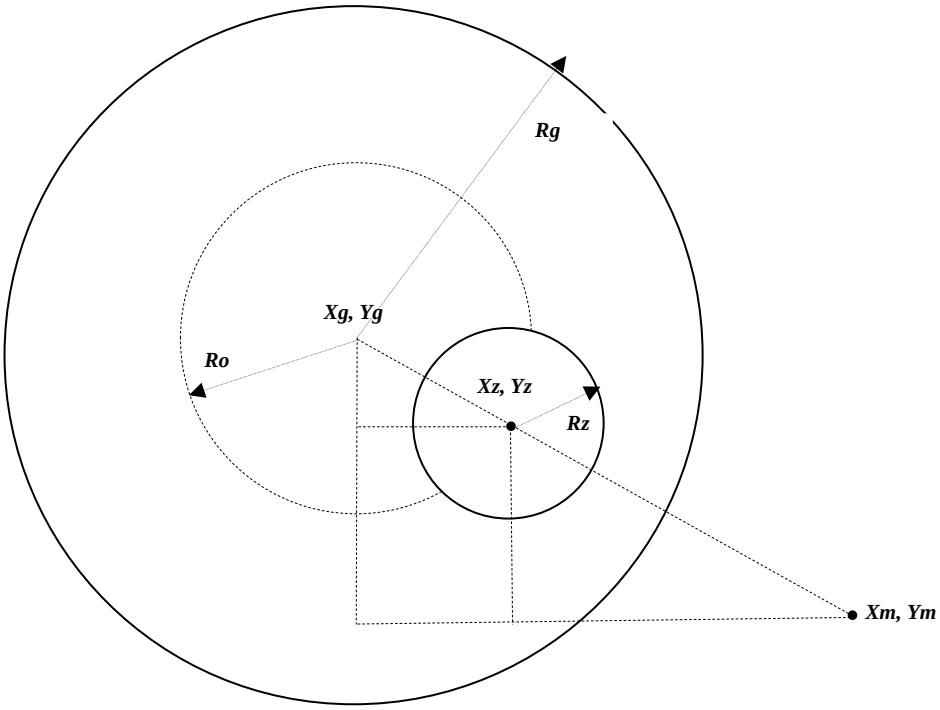

Рисунок 10. Геометрия объекта класса "Глаз"

Координаты этой точки находятся из подобия треугольников, обозначенных на рисунке. Обратите внимание, что расстояние между точками (Xg, Yg) и (Xz, Yz) равно Ro.

Этап 2

Разработать класс "Лицо" (***MyFace)***, объединяющий два глаза как показано выше (**Ошибка! Источник ссылки не найден.**).

Методические указания.

Основа лица - наследующий класс ***MyCircle***. Класс включает (агрегирует) два объекта класса "Глаз".

Класс лицо должен иметь возможность определять задаваемые пользователем размер и цвет лица.

Создание носа, рта, ушей и прочего не обязательно. Это ничего не добавляет к приобретению знаний и навыков ООП. Но с эстетической точки зрения, конечно, выигрышно. Каждый пусть сам решает за себя – сделать красиво или по минимуму (читай - лениво). ☺

Этап 3

Создать класс "Любопытные" (***Onlookers***).

Класс объединяет задаваемое пользователем количество объектов класса "Лицо" и визуализирует их на экране. Объекты различаются размерами и цветами лиц. Все они следят за передвижением мышки.

Особо интересно будет, если заменить изображение курсора на что-либо иное, например, ползающего жучка (или пирожок, или ... – фантазируйте тут самостоятельно). Это не добавит вам знаний по ООП, но добавит знаний по работе со средой разработки и выполнения программных приложений. И, несомненно, доставит больше позитива от работы над этим проектом. Но замена изображения курсора не входит в обязательную часть задания.

Методические указания

Создайте массив, объединяющий объекты класса ***MyFace***.

В цикле заполните этот массив объектами класса ***MyFace*** со значениями параметров, имеющими случайный характер.

При изменении координат положения курсора в цикле организуйте вызов соответствующего метода объектов ***MyFace*** для изменения значения параметров зрачков объектов.

Для управления определением и изменением цвета можно использовать стандартный метод ***RGB(int R, int G, int B)***. Метод формирует и возвращает цвет, с интенсивностью задаваемых компонентов цвета (соответственно красного, зеленого и синего). Интенсивность компонентов варьируется в диапазоне [0, 255]. Таким образом, черный представляется набором (0, 0, 0), а белый (255, 255, 255).

Вопросы для самоконтроля

1. Какое отличие между значимыми и ссылочными типами?
2. Как и зачем использовать конструкцию *Using* в *C#*?
3. Что подразумевается под свойствами в *C#*?
4. Что такое *namespace* в ООП?
5. Что нам дает существование механизма наследования?
6. Что определяет *protected*?

Проект «Зоопарк»

Цель

Практическое освоение механизма реализации интерфейсов.

Практическое закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод.

Практическое задание

Разработать проект «Зоопарк» — приложение, позволяющее из списка разных объектов («летучие мыши», «обезьяны», «утки», «пингвины», «орлы», «рыбы») выбрать летающих, ходящих и/или плавающих. Классы конкретных животных не обязательно должны иметь общего предка.

Все классы реализуют соответствующие по смыслу интерфейсы: летучая мышь летает; обезьяна ходит; утка ходит, летает и плавает и т.д. Реализация методов *Fly(), Walk()* и *Swim()* предельно проста: возвращается строка «*I can fly*…» («…*walk*» или «…*swim*» соответственно). Метод *Kind()* просто возвращает строку с названием животного.

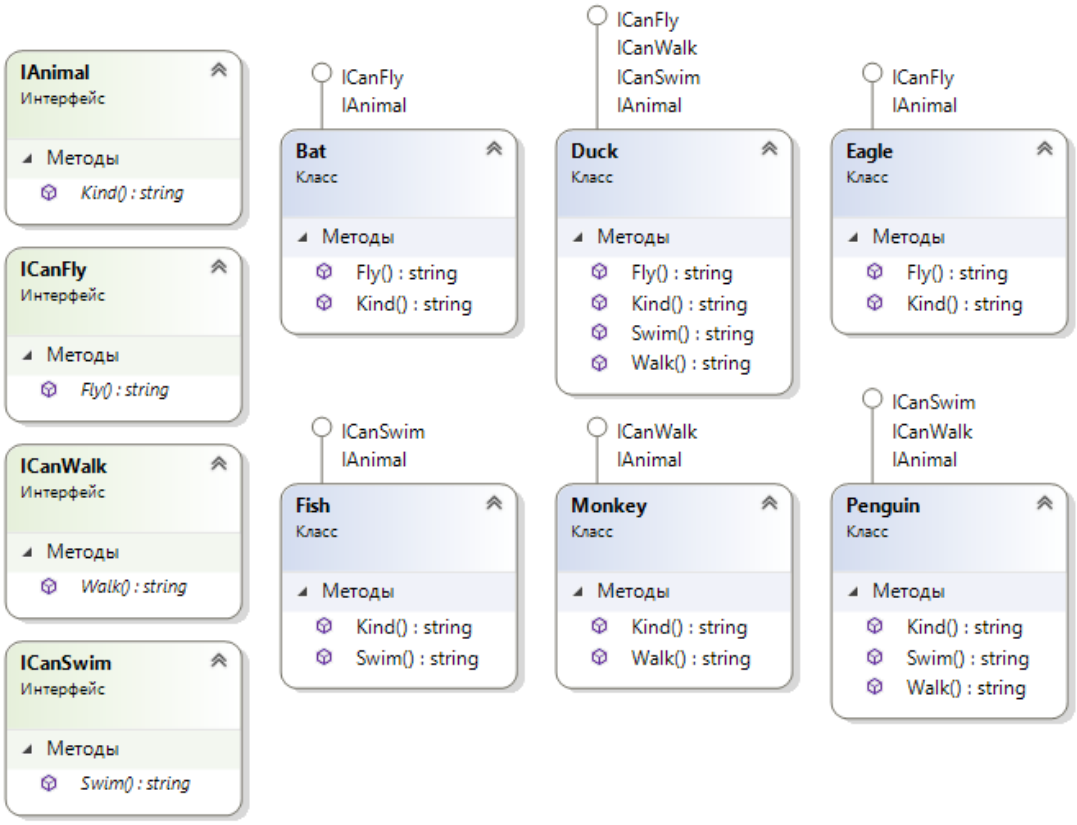

Рисунок 11. UML- диаграмма семейства классов для проекта "Полиморфизм и интерфейсы"

Применение классов:

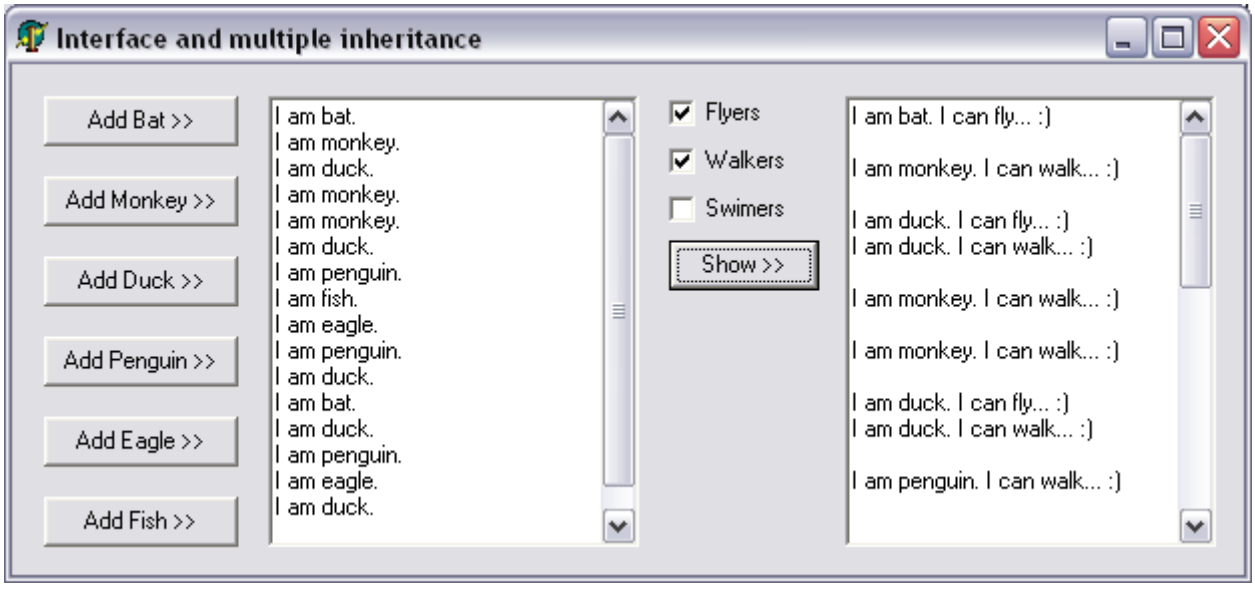

Рисунок 12. Форма проекта "Полиморфизм и интерфейсы"

По нажатию на кнопки «*Add*...» создаются соответствующие объекты и добавляются в массив типа *Animal*. Названия попадают одновременно в компонент *TextBox* (или *ListBox*). Далее галочками отмечаются желаемые «качества». И по нажатию на кнопку «*Show*» из массива показываются только «подходящие» животные в другой компонент *TextBox* (или *ListBox*). Выводятся также их названия и результаты вызовов методов *Fly()*, *Walk()* и *Swim()*.

Вопросы для самоконтроля

1. Что такое «интерфейсы» в ООП? Для чего нужен этот механизм? Как объявляются интерфейсы?
2. Как классы реализуют интерфейсы? Что это означает для класса?
3. Как избежать конфликта имен, если класс реализует два разных интерфейса с одинаковыми по сигнатуре методами?
4. Что будет, если класс «забудет» реализовать интерфейс, который объявлен в классе?

Проект «Железнодорожный состав»

Цель задания

Практическое освоение технологических приемов, связанных с использованием наследования, агрегации и виртуальных методов.

Задание рассчитано на выполнение в несколько этапов. Каждый этап преследуют освоение и закрепление некоторых описанных в цели технологических средств ООП в порядке возрастания их сложности.

Практическое задание

Задание сформировано таким образом, чтобы при минимальных временных затратах на кодирование, освоить ряд особенностей наследования, включения (агрегации) и традиционно достаточно сложного технологического приема – использование виртуальных методов.

Задание состоит из ряда этапов нарастающей сложности, выполняя которые создается класс, описывающий железнодорожный состав. Состав состоит из товарных вагонов, нагруженных грузами разных видов. При визуализации состава, каждый вагон отображается по-разному в соответствии с видом груза, которым он заполнен.

Для простоты реализации предполагается, что все числовые данные имеют тип целое.

Этап 1. Создание двух классов: класс круг и класс прямоугольник

Разработать класс *MyCircle* (название условное). Класс должен содержать:

- параметры круга: поля для хранения значений координат центра круга и радиус круга. Доступ к полям реализуется свойствами и разрешается только чтение данных;
- конструктор с параметрами, определяющими значения полей;
- метод записи данных в ранее созданный объект;
- метод визуализации круга.

Разработать класс *MyRectangle* (название условное, если предпочитаете другое, то предоставляется полная свобода действий). Класс должен содержать:

- параметры прямоугольника. Доступ к полям реализуется свойствами и разрешается только чтение данных;
- конструктор с параметрами, определяющими значения полей;
- метод записи данных в ранее созданный объект;

- метод визуализации прямоугольника.

Создать программное приложение, используя классы *MyCircle* и *MyRectangle*, обеспечивающее:

- определение пользователем при создании объектов значений параметров, задаваемых на форме;
- создание объектов этих классов с определенными пользователем параметрами по нажатию кнопки «Создать»;
- занесение в ранее созданные объекты этих классов значения параметров, определенных пользователем по нажатию кнопки «Задать параметры»;
- визуализацию созданных объектов по нажатию кнопки «Визуализировать».

Методические указания

Круг в классе *MyCircle* описывается тремя параметрами:

- координата центра круга по оси X;
- координата центра круга по оси Y;
- радиус круга.

Описание прямоугольника возможно двумя способами. Один из них состоит в задании пар координат двух диагонально удаленных углов прямоугольника (как правило, левого верхнего и правого нижнего). Второй — определение координат одного из углов и задание размеров прямоугольника по осям координат X и Y (ширина и высота). На практике используются оба. Первый более универсален, т.к. для второго надо знать еще дополнительную информацию о том, какой же из углов имеется в виду. Но при выполнении задания выбирайте любой способ описания.

Довольно часто различные затруднения вызывает реализация в классе визуализации. Поэтому следуйте рекомендациям ниже.

Во-первых, при реализации методов визуализации присвойте им мнемоничные тождественные имена (типа *Show*, *Draw* и т.п.).

Во-вторых, для качественного исполнения визуализации нужно помнить принцип: реализация класса не должна зависеть от конкретных элементов интерфейса. Т.е. недопустимо в классе использовать конкретный внешний объект (например, *panel1*). Так же не очень хороший подход — передача в качестве параметра типа *Panel*. Наиболее распространенный способ визуализации — передача в метод визуализации в качестве параметра объекта класса *Graphics*. Конкретный объект *Graphics* по сути можно рассматривать как конкретного художника. И пользователь вашего класса сам создает этого

художника для рисования там, где ему требуется (на форме, на панели и т.п.). Таким образом, ваш класс приобретает унификацию — он не будет привязан к особенностям интерфейса конкретной задачи.

Этап 2. Создание класса *MyVagon*, описывающего железнодорожный вагон

Необходимо разработать класс *MyVagon* (название условное, если предпочитаете другое, то предоставляется полная свобода действий). Класс должен содержать:

- поле – количество перевозимого в вагоне груза;
- свойство, обеспечивающее доступ к этому полю как на чтение, так и на запись;
- геометрические параметры, определяющие положение вагона на экране при необходимости его визуализации.
- метод визуализации вагона.

Методические указания

Примем, что изображение вагона представляет собой комбинацию объектов двух типов: круг и прямоугольник (Рисунок 13).

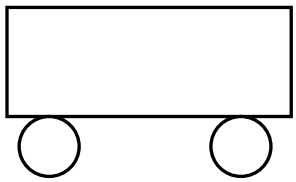

Рисунок 13. Отображение объекта класса *MyVagon*

Таким образом, изображения нашего вагона состоит из трех объектов: прямоугольника (кузов) и двух кругов (колес). Классы этих объектов созданы вами на предыдущем этапе проекта.

Для минимизации количества параметров, определяющих отображение вагона, выберите некоторую величин L — которую назовем характерный размер вагона. Пусть, например, это будет длина кузова. Тогда высоту кузова можно просто считать, например, как $L/2$, а радиус колеса $L/5$. Таким образом, чтобы задать место и размеры отображения вагона достаточно будет трех параметров: координат левого верхнего угла вагона ($X0, Y0$) и характерного размера (L). Значения коэффициентов 2 и 5, на которые делится L, приведены здесь условно.

Если уже есть классы необходимых нам объектов, то надо их использовать при разработке нового класса *MyVagon*. Использование существующих ранее классов при создании нового класса возможно в ООП с помощью двух технологий: наследование и включение (агрегирование).

В *C#* существует ограничение — класс может наследовать только один класс. Это не удовлетворяет полностью нашим потребностям (нам надо три объекта объединить в один), но все равно в учебных целях целесообразно этот механизм использовать. Для наследования выбирайте один из двух классов по вашему усмотрению. Два других объекта объединяются в создаваемом классе методом включения (агрегации). Т.е. в классе определяются два поля, имеющие тип объектов включаемого класса, в конструкторе создаваемого класса создаются объекты этих классов и ссылки, полученные в результате вызовов их конструкторов, помещаются в эти поля.

О методе визуализации. Присвойте ему имя тождественное имени методов визуализации круга и прямоугольника. Не пытайтесь снова написать алгоритмы визуализации включенных (агрегированных) объектов. Вы уже научили их себя отображать. Просто обратитесь к их методам визуализации.

Немного сложнее обстоит дело с наследованным объектом. Обратите внимание, что при создании наших классов все методы визуализации имеют тождественные имена. Поэтому при попытке обращения к методу визуализации родителя из потомка будет, естественно, вызван метод потомка и приложение просто зациклится. А если подождать определенное время, то произойдёт исключительная ситуация — переполнение стека (*Stackoverflow*). Для того чтобы вызвать одноименный метод родителя используйте конструкцию *base*.

Этап 3. Создание класса *MyTrain*, описывающего железнодорожный состав, состоящий из вагонов (объектов класса *MyVagon)*

Необходимо разработать класс *MyTrain* (название условное, если предпочитаете другое, то предоставляется полная свобода действий). Класс должен содержать:

- поле — количество вагонов в составе;
- поле — динамический массив ссылок на объекты класса *MyVagon*;
- поля — геометрические параметры, определяющие положение первого вагона на экране при визуализации состава;
- метод заполнения с помощью датчика псевдослучайных чисел тоннажа, перевозимого каждым из вагонов груза;
- метод подсчета общего перевозимого тоннажа состава;
- метод визуализации состава.

Расширить возможности программного приложения за счет:

- возможности ввода пользователем определения количества вагонов в состав;

- создания объекта *MyTrain* по нажатию соответствующей кнопки;
- загрузки вагонов состава по нажатию соответствующей кнопки;
- визуализацию перевозимого составом общего тоннажа груза по нажатию кнопки;
- визуализацию состава по нажатию соответствующей кнопки.

Методические указания

Рисунок 14 представляет возможную визуализацию объекта класса *MyTrain*.

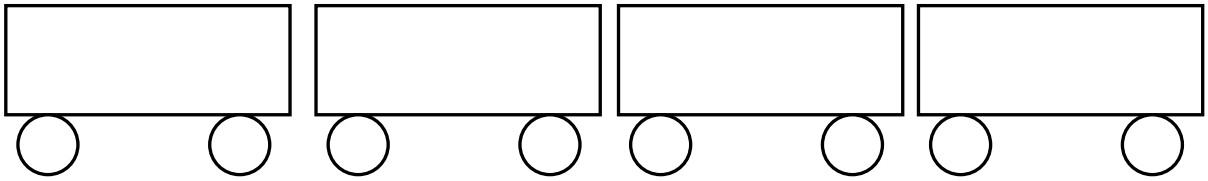

Рисунок 14. Отображение объекта класса *MyTrain*

Формальными параметрами конструктора класса *MyTrain* являются:

- количество вагонов в составе;
- характерный размер вагона;
- координаты левого верхнего угла первого вагона на экране.

В теле конструктора организуется цикл по количеству вагонов в котором:

- определяются численные значения геометрических параметров расположения вагонов (объектов класса *MyVagon*);
- создаются объекты с этими значениями;
- сохраняется ссылки на созданные объекты в массиве, являющимся полем разрабатываемого класса.

Обратите внимание на следующие два обстоятельства.

Первое — созданный объект класса *MyTrain* существует независимо от того визуализируем мы его или нет. Очень часто у многих магия рисования на форме вагона и состава создает иллюзию, что проект направлен на рисование. Нет! Эта иллюзия сразу же развеивается, если убрать (достаточно мысленно убрать) из требований к этому этапу программного приложения последний пункт: «визуализацию состава по нажатию соответствующей кнопки». Тогда не будет никакого визуального отображения состава. Но описывающий состав объект будет существовать, и будут существовать объекты вагоны, и будут существовать объекты круги и прямоугольники, составляющие эти вагоны, и будет храниться тоннаж перевозимого каждым вагоном груза, и можно будет получить тоннаж всего состава. Т.е. визуализация состава — это только одна возможность из ряда функционала разрабатываемого класса. С технологической точки зрения визуализация

помогает убедиться в том, что приложение создано корректно. В том, что, например, колеса вагона расположены внизу, а не вверху, в том, что те же колеса примыкают к вагону, а не расположены где-то в стороне. А такие случаи наблюдаются достаточно часто при отладке класса *MyVagon*. И, кроме того, визуализация в данном проекте просто делает его веселее, что тоже не маловажно в процессе овладения новым знанием.

Второе — все действия с совокупностью вагонов скрыты в методах разрабатываемого класса. Пользователь, создавая объект этого класса, не представляет всей внутренней его реализации. Для него это черный ящик, имеющий только те методы, которые ему предоставлены, а именно: конструктор, заполнение вагонов грузом, получение общего тоннажа груза в составе, визуализация состава. А вся сложная структура реализации совершенно от него скрыта и ему не требуются эти знания, для того, чтобы работать с объектом класса *MyTrain*.

Этап 4. Практическое освоение технологии виртуальных методов

Необходимо на базе класса *MyVagon*:

- разработать класс *MyVagonCoal* (название условное, если предпочитаете другое, то предоставляется полная свобода действий), описывающий вагон, нагруженный углем. Класс должен включать все члены класса *MyVagon*, но метод визуализации вагона должен быть соответственно изменен.
- разработать класс *MyVagonSand* (название условное, если предпочитаете другое, то предоставляется полная свобода действий), описывающий вагон, нагруженный песком. Класс должен включать все члены класса *MyVagon*, но метод визуализации вагона должен быть соответственно изменен.
- модифицировать алгоритм формирования состава в классе *MyTrain* таким образом, чтобы состав формировался случайным образом из объектов, принадлежащих классам *MyVagon*, *MyVagonCoal*, *MyVagonSand*.
- используя механизм виртуальных методов реализовать отображение состава, состоящего из вагонов различных типов так, чтобы вагоны с различным грузом отображались в соответствии с их методами отображения.

Методические указания

Наиболее продуктивным путем создания классов *MyVagonCoal* и *MyVagonSand* является создание их на базе класса *MyVagon* путем наследования. Тогда достаточно изменить только алгоритм метода визуализации для того, чтобы можно было бы наглядно убедиться в корректности работы разработанного программного продукта.

Для этого, прежде всего, надо определиться с тем, как будут отображаться вагоны с углем и песком. Рисунок 15 представляет возможные простейшие варианты отображения объектов создаваемых классов (отображения приведены как пример, если предпочитаете другие, то предоставляется полная свобода действий).

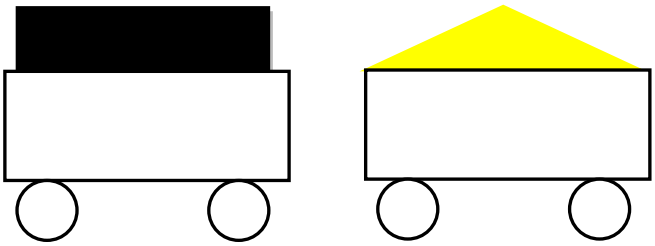

Рисунок 15. Возможная форма визуализации вагонов

При реализации методов визуализации разрабатываемых классов используйте вызов метода визуализации родительского класса.

При модификации алгоритма формирования состава в классе *MyTrain* обратите внимание на то, что массив класса *MyTrain*, хранивший объекты вагоны классов *MyVagon*, может так же хранить и объекты классов *MyVagonCoal* и *MyVagonSand*. Это может показаться на первый взгляд странным, т.к. массив по определению суть структура, объединяющая элементы одного типа. Но среди допустимых случаев корректного использования оператора присваивания есть такое положение: тип, расположенный в операторе присваивания справа, является прямым или косвенным потомком класса, стоящего слева. И именно это положение дает, в частности, возможность использования виртуальных методов.

После выполнения первых трех из четырех пунктов задания запустите приложение на выполнение. Объясните полученный результат.

Выполните последний пункт задания. Запустите приложение на выполнение. Объясните полученный результат.

Объясните разницу между результатами без использования виртуальных методов и с использование их.

Вопросы для самоконтроля

1. Что обеспечивает в *C#* ключевое слово *base*?
2. В каких случаях в массиве могут храниться объекты различного типа?
3. Что дает использование виртуальных методов?
4. Чем агрегация отличается от наследования?
5. Что означает ключевое слово *virtual*?
6. Как из конструктора потомка обратиться к конструктору родителя?

7. Зачем используется ключевое слово *override*?
8. Приведите пример эффективности использования виртуальных методов (на данное задание не ссылаться).
9. Перечислите модификаторы доступа и когда они используются?

Проект «Площади фигур»

Цель

Практическое закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод.

Практическое освоение новых технологических приемов ООП: наследование, отношение «предок-потомок» между классами, виртуальные методы, перегрузка виртуальных методов.

Практическое задание

Создать четыре пользовательских типа: класс-предок «Фигура» и три класса-потомка «Прямоугольник», «Круг» и «Треугольник». Базовый класс никак не определяет свойства фигур-потомков, зато определяет их методы: расчет площади фигуры и вывод информации о фигуре и ее площади на экран. В классах-потомках необходимо задать свойства каждой фигуры и реализовать подсчет площади по соответствующим формулам.

Разработать тестирующее приложение, позволяющее набирать коллекцию разных фигур и фильтровать её по значению площади этих фигур.

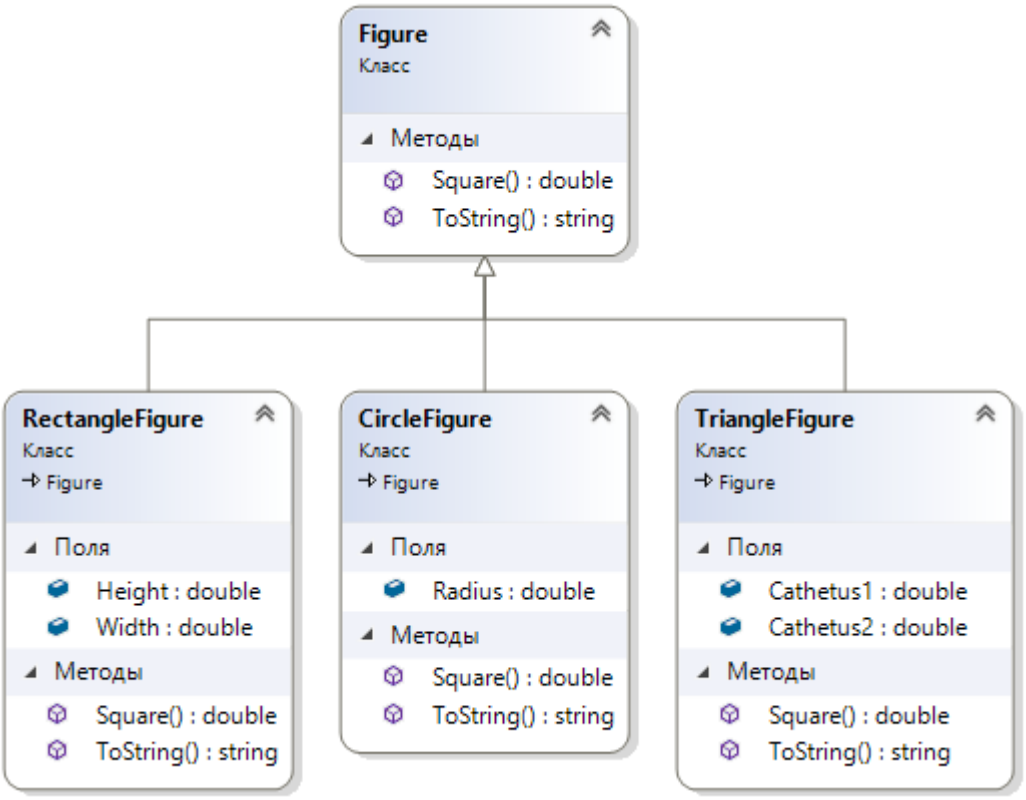

Рисунок 16. UML- диаграмма семейства классов для проекта «Площади фигур»

Поля и свойства:

Height и *Width* — свойства класса прямоугольника *RectangleFigure*, определяющие высоту и ширину будущих объектов этого класса.

Radius — свойство класса *CircleFigure*, определяющее радиус будущих объектов этого класса.

Cathetus1 и *Cathetus2* — свойства класс *TriangleFigure*, определяющие катеты будущих объектов этого класса.

Методы:

Square() — метод, рассчитывающий и возвращающий площадь фигуры. В базовом классе Figure этот метод объявлен как виртуальный (*virtual*) и не имеет реализации. В классах-потомках он перезаписывается (*override*) и, соответственно, реализуется каждым потомком по-своему, в зависимости от математической формулы площади.

ToString() — метод, возвращающий строку вида «Название фигуры: Свойство = значение. Площадь = значение». Например:

Треугольник: первый катет = 23,4; второй катет = 56. Площадь = 655,2
Круг: радиус = 31,5. Площадь = 3115,66

Применение класса:

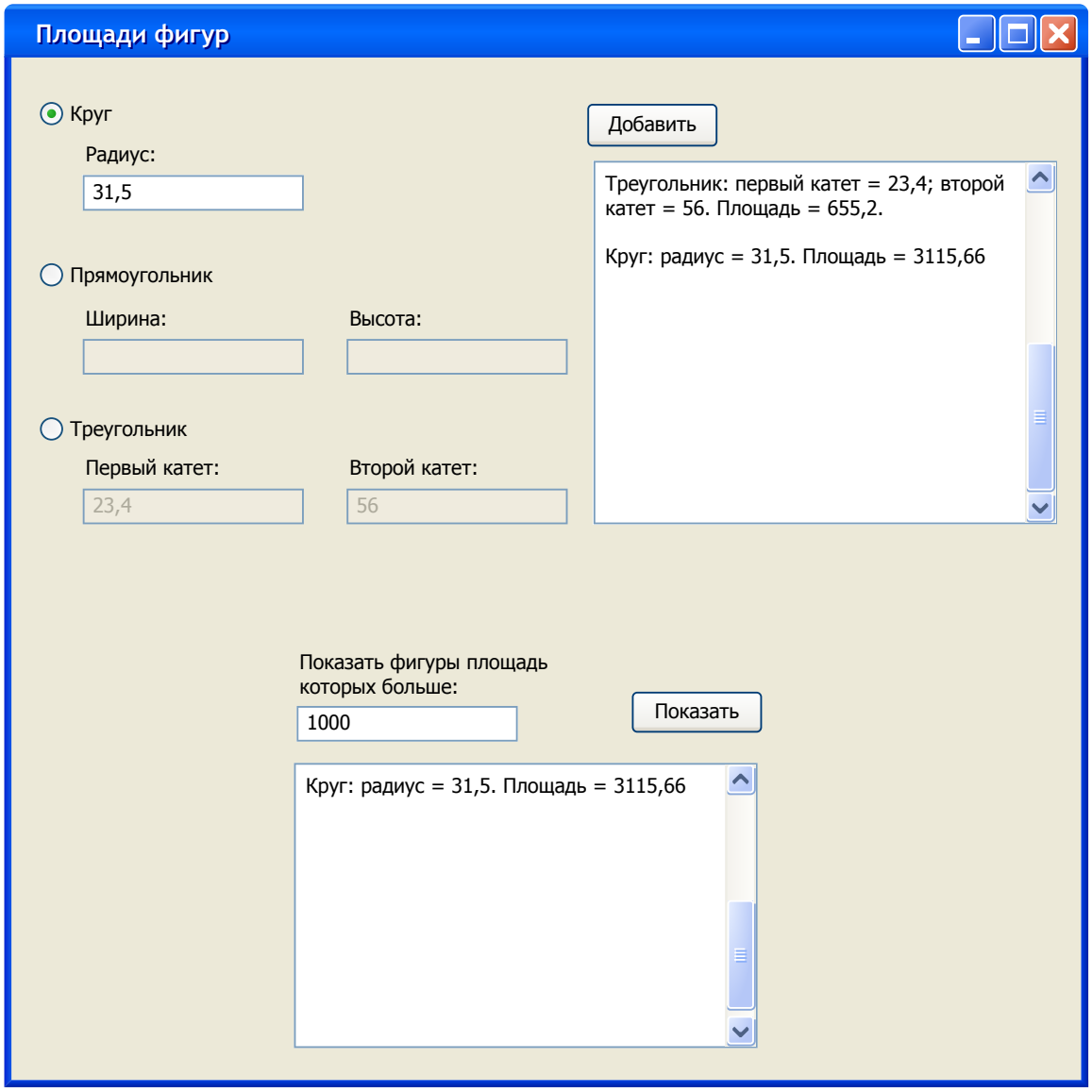

Рисунок 17. Тестирование семейства классов проекта «Площади фигур»

На форме сначала выбирается тип фигуры, затем вносятся ее данные (радиус, ширина-высота или катеты). По кнопке «Добавить» создаваемая фигура помещается в массив (или список). Одновременно с этим информация о добавляемой фигуре отображается в компонент *ListBox*.

Далее в *TextBox* вводится нижняя граница отображаемой площади. На кнопке «Показать» собранный массив (или список) фигур фильтруется по заданному значению: на *ListBox* отображаются только те фигуры, площадь которых больше заданной.

Вопросы для самоконтроля

1. Что такое «наследование классов» в ООП? Для чего нужен этот механизм? Как объявляются классы-наследники?

2. Что такое виртуальные методы? Чем они отличаются от обычных (не виртуальных)?
3. Что такое «перегрузка виртуальных методов» в классе-потомке?
4. Что происходит с методами при наследовании? Как ведут себя методы класса-предка в классе-потомке: обычные, виртуальные?
5. Что будет, если в классе-потомке объявить метод, по сигнатуре полностью совпадающий с методом класса-предка?
6. Что будет, если в классе-потомке объявить метод, одноименный методу класса-предка, но разный с ним (методом) по сигнатуре?

Проект «Список студентов»

Цель

Практическое освоение новых технологических приемов: механизма сериализации[3] (сохранения) коллекции объектов в файл.

Практическое закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод.

Практическое задание

Разработать проект «Список студентов» — приложение, позволяющее создавать объект-студент и вносить данные об этом объекте: имя, возраст, группу, адрес и фотографию. Объекты сохраняются в типизированный список *List<Student>*, который затем нужно сохранить в файл на диск. Так же необходимо предусмотреть загрузку списка студентов из файла и отображение информации о каждом, ранее созданном объекте-студенте.

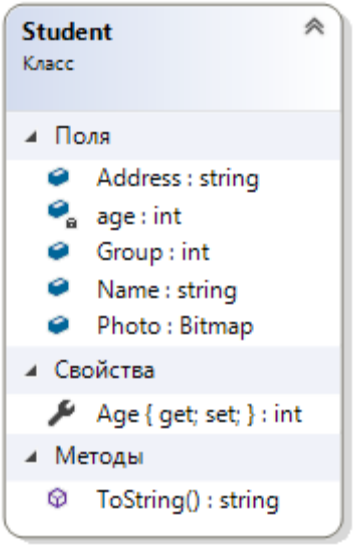

Рисунок 18. UML диаграмма класса Student

Поля и свойства:

Name, *Address*, *Group* — поля, хранящие имя, адрес и группу студента, соответственно.

Age — свойство, инкапсулирующее скрытое поле *age* для хранения возраста студента. Необходимо предусмотреть обработку входного значения свойства (аксессор *set*): если введено число меньше 14 или больше 65, то выдавать исключение

[3] Для бинарной сериализации в *C#* применяется тип *BinaryFormatter*.

ArgumentOutOfRangeException с сообщением о том, что возраст должен находиться в диапазоне от 14 до 65.

Photo — поле для хранения изображения студента, представляющее собой объект типа *Bitmap*[4]. Необходимо предусмотреть возможность загрузки изображения из графического файла **.bmp*, **.jpeg* или **.png*.

Методы:

ToString() — перезаписанный (override) метод, возвращающий короткую информацию о студенте в виде «Имя, возраст».

Применение класса:

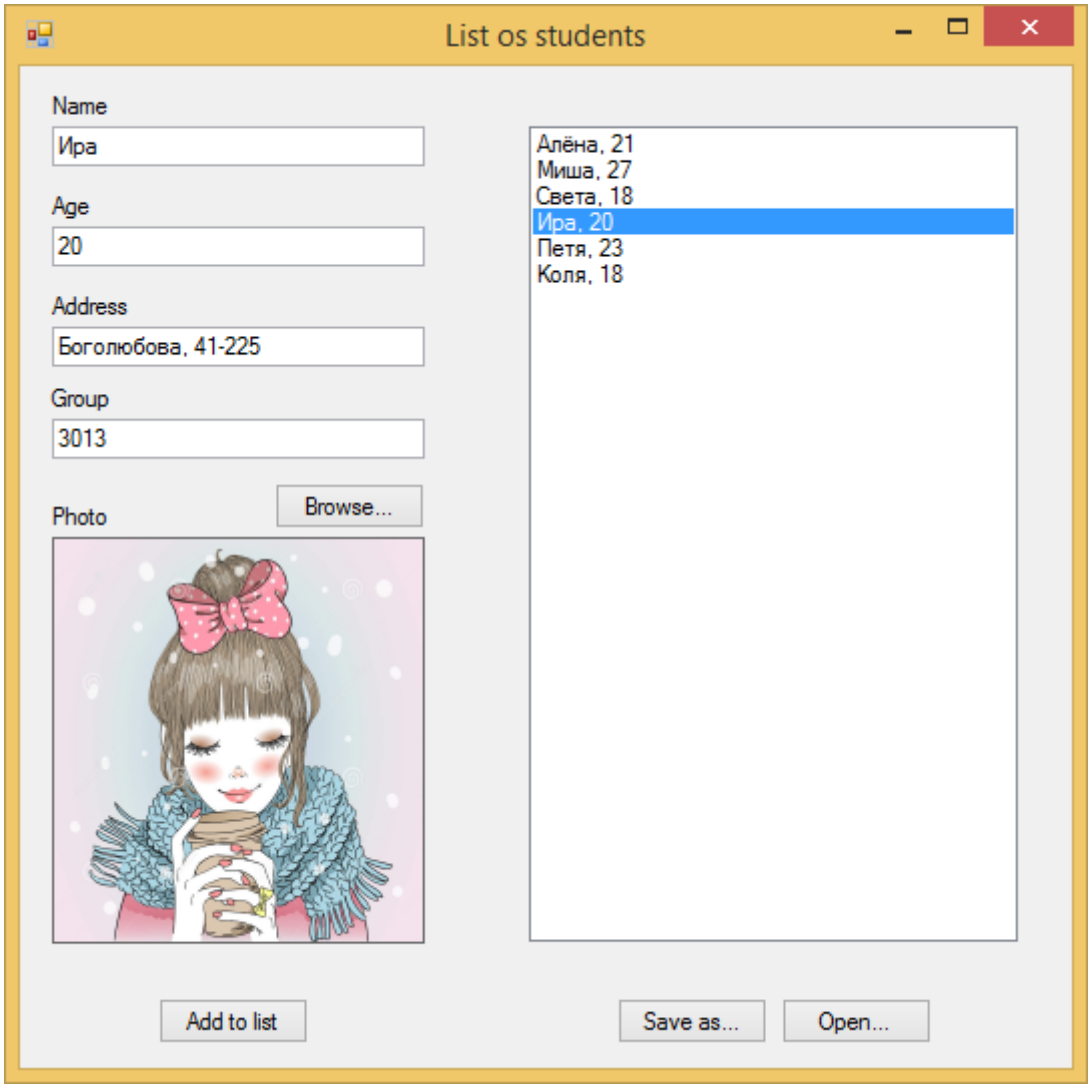

Рисунок 19. Форма проекта «Список студентов»

[4] Тип *Bitmap* находится в пространстве имен *System.Drawing*.

На форме располагаются *TextBox* для ввода имени, возраста, адреса, группы и *PictureBox* для отображения загруженного изображения. Для загрузки изображения удобно применять *OpenFileDialog*.

По нажатию на кнопку «*Add to list*» формируется объект типа *Student* и добавляется в список студентов *List<Students>*. Имя и возраст добавляемого студента можно дополнительно вывести на компонент *ListBox* (справа).

Если в *ListBox* выделяется какая-либо строка с именем, то информация о выделенном студенте должна отображаться в *TextBox* и в *PictureBox* слева. Для этого используется событие *SelectedIndexChanged* у *ListBox*.

Кнопки «*Save as…*» и «*Open…*» реализуют, соответственно, сохранение списка *List<Students>* в файл и загрузку его из файла. После загрузки имена студентов из списка необходимо перечислить на *ListBox*.

Вопросы для самоконтроля

1. Что такое класс? Что такое объект класса? Чем класс отличается от объекта?
2. Объясните понятия «поле класса», «конструктор». Что они хранят, что делают, каким образом инициализируются?
3. Как объекты сохраняются на диск? Что такое «сериализация» и «десериализация»? Какие виды сериализации бывают? Почему для этого задания не подходит *XML*-сериализация?
4. Что необходимо сделать, чтобы класс был сериализуемым?

Класс «Календарная дата»

Цель задания

Практическое освоение и закрепление базовых понятий ООП: класс, объект, поле, метод.

Задание рассчитано на выполнение в два этапа с целью акцентирования внимания на базовом принципе ООП — функциональное разделение реализации.

Практическое задание

Создаем простое приложение — календарь. Календарь обеспечивает следующие возможности:

- установка текущей даты;
- хранение текущей даты;
- изменение текущей даты на следующую дату;
- контроль корректности ввода даты.

Календарная дата состоит из трех целых чисел: день, месяц и год.

Необходимо создать класс *MyDate* календарной даты, в котором реализуется:

- хранение трех целых чисел, определяющих день, месяц и год;
- метод для «перелистывания» даты календаря *NextDate()*, изменяющий текущую календарную дату на следующую дату.

Этап 1

Создание полей для хранения значений конкретной даты, внутренних вспомогательных данных и методов, реализующих перечисленный ниже функционал.

Необходимо разработать класс *MyDate*, реализующий описанное выше представление даты. Класс должен содержать:

- поля для хранения значений дня, месяца, года;
- метод *SetDate* для занесения в объект класса значений день, месяц, год.
- метод *NextDate*, изменяющий дату текущую дату на следующую календарную.
- метод *Visual*, отображающий дату на форме приложения.

Создать программное приложение, которое используя класс *MyDate*, обеспечивает:

- создание объекта класса *MyDate*;
- ввод в созданный объект данных: числа, месяца, года;
- хранение созданного объекта с введенными данными;
- изменение по нажатию кнопки текущей даты на следующую.

Методические указания

Количество дней в месяцах целесообразно хранить массиве. Номер месяца ассоциируется с индексом массива. Заполнение массива данными можно реализовать при инициализации объекта (в конструкторе).

Получение количества дней в заданном месяце заданного года рекомендуется реализовать отдельным внутренним (невидимым извне) методом, который учитывает для февраля високосный или не високосный год.

Проверку, является ли високосный ли текущий год рекомендуется осуществлять в отдельном методе, возвращающем *true*, если год високосный и *false*, в противном случае. Такой метод позволит сильно упростить код алгоритма изменения даты.

Наиболее интересная часть данного задания состоит в реализации алгоритма «перелистывания» дня календаря. Это связано с различным количеством дней не только в различных месяцах, но и в различных годах, так как существуют високосные и не високосные годы. Природа при создании Солнечной системы не позаботилась о том, чтобы синхронизовать скорость вращения Земли вокруг своей оси и период обращения Земли вокруг Солнца. Поэтому человечество, создавая календарь, вынуждено применять ряд ухищрений для того, чтобы календарь стабильно соответствовал реалиям.

История создания современного календаря прошла ряд этапов, каждый из которых старался точнее провести формальное описание природного процесса. У разных народов в различные исторические эпохи существовали и существуют различные календари. Остановимся на кратком историческом экскурсе возникновения календаря, которым пользуется в настоящее время большинство стран мира и, в частности, Россия.

По древнейшему римскому календарю год состоял из десяти месяцев, причём первым месяцем считался март. Этот календарь был заимствован у греков. Согласно принятой традиции, считается, что его ввёл основатель и первый царь Рима – Ромул, в 738 году до н.э.. Восемь названий месяцев этого календаря (март, апрель, май, июнь, сентябрь, октябрь, ноябрь, декабрь) сохранились во многих языках до настоящего времени. На рубеже VII и VI веков до н. э. из Этрурии был заимствован календарь, в котором год делился

на 12 месяцев: добавленные январь и февраль следовали после декабря. Год состоял из 354 дней: 6 месяцев по 30 дней и 6 месяцев по 29 дней, но каждые несколько лет добавлялся дополнительный месяц.

На смену римскому календарю пришел Юлианский календарь, введённый Юлием Цезарем с 1 января 45 года до н. э. Год по юлианскому календарю начинается 1 января. В юлианском календаре обычный год состоит из 365 дней и делится на 12 месяцев. Раз в 4 года объявляется високосный год, в который добавляется один день — 29. Таким образом, юлианский год имеет продолжительность в среднем 365,25 дней, что больше на 11 минут продолжительности тропического года. В Киевской Руси календарь был известен под названием «Миротворного круга», «Церковного круга». Юлианский календарь в современной России обычно называют «старым стилем».

Следующим приближением стал григорианский календарь, который и используется в настоящее время в большинстве стран мира. Календарь был введён папой римским Григорием XIII в католических странах 4 октября 1582 года вместо прежнего юлианского: следующим днём после четверга 4 октября стала пятница 15 октября. В григорианском календаре длительность года принимается равной 365,2425 суток. Длительность не високосного года — 365 суток, високосного — 366. Определение високосного и не високосного года осуществляется по следующему алгоритму:

- год, номер которого кратен 400, — високосный;
- остальные годы, номер которых кратен 100, — не високосные;
- остальные годы, номер которых кратен 4, — високосные.

Таким образом, 1600 и 2000 годы были високосными, а 1700, 1800 и 1900 годы високосными не были.

Количество дней во всех месяцах кроме февраля фиксировано:

- 31 день в январе, марте, мае, июль, август, октябрь, декабрь;
- 30 дней в апреле, июне, сентябре, ноябре.

В феврале в не високосном году 28 дней, в високосном 29.

Этап 2.

Добавить в класс:

- конструктор с параметрами, передающими значения дня, месяца, года.

- метод *IsValid*, реализующий проверку корректности вводимой даты, возвращающий *false*, если значение даты недопустимо (например, 32 января 2015 года, 29 февраля 2011 года).

В приложении ввести средства, демонстрирующие работу новых членов класса.

Методические указания

Метод проверки корректности вводимой даты реализовать по общепринятому правилу оформления методов проверки. А именно, при положительном результате проверки (дата введена правильно) метод возвращает значение *true*. В противном случае возвращаемое значение — *false*.

Вопросы для самоконтроля

1. В чем разница между интерфейсом и абстрактным классом?
2. Что такое деструктор и зачем он нужен?
3. Что такое указатель *this*?
4. Что такое в ООП статический элемент?
5. В чем разница инкапсуляции и сокрытия?

Проект «Календарь»

Цель задания

Практическое освоение и закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод, создание в методе класса новых объектов.

Задание рассчитано на выполнение в два этапа. Каждый этап преследуют освоение некоторых описанных в цели технологических средств ООП в порядке возрастания их сложности.

Практическое задание

Создать простое приложение — календарь.

Календарь содержит в себе:

- текущую дату;
- даты дней рождения (например, студентов группы);
- даты праздников.

Программа-календарь, позволяет:

- установить текущую дату;
- изменить текущую дату на следующую дату;
- выводить на экран текущую дату;
- добавлять новый праздник или день рождения;
- выводить на экран ближайшие праздники и дни рождения.

Этап 1

При разработке любого приложения надо двигаться от простого к сложному поэтапно. Поэтому на первом этапе реализуем класс праздник *MyCelebration*, содержащий:

- дату;
- название праздника;
- конструктор с параметрами, обеспечивающими описание праздника.

И класс календарь *MyCalendar*, содержащий:

- текущую дату;
- список объектов праздник;
- метод установки в объекте класса текущей даты;
- метод добавления нового объекта в список праздник;
- метод, возвращающий дату ближайшего праздника.

Далее реализуем приложение, обеспечивающее:

- при запуске создание объекта класса *MyCalendar*;
- ввод текущей даты и передачу ее в объект класса *MyCalendar*;
- создание объекта *MyCelebration* с заданными на форме значениями параметров;
- занесение созданного объекта *MyCelebration* в список объектов праздников объекта *MyCalendar*.

Методические указания

Для хранения даты используйте объекты класса *MyDate*, разработанного при выполнении задания «Календарная дата».

Хранение множества объектов праздников можно организовать на базе поля класса *MyCalendar*, представляющего собой массив объектов *MyCelebration*. Но более продуктивно воспользоваться, например, классом списка: *System.Collections.Generic.List*.

Для реализации поиска ближайшего к текущей дате праздника расширьте класс *MyDate* методом, возвращающим в количестве дней разницу между датой, хранимой в объекте и датой, передаваемой этому методу в качестве параметра. Если возвращаемый результат отрицателен, то в вашем календаре нет еще внесенных праздников. Реализовав такой метод и использовав его при циклическом просмотре всех членов списка праздников, вы легко определите ближайший праздник.

Этап 2

На этапе 1 выполнено более половины общего задания. Остался не выполненным пункт, связанный с возможностью заносить в календарь дни рождения.

Необходимо создать класс *MyBirthdays*, содержащий:

- дату дня рождения;
- имя;
- фамилию.
- конструктор с параметрами, обеспечивающими определение полей класса.

В приложении ввести средства, демонстрирующие работу класса *MyBirthdays*.

Методические указания

Создание класса *MyBirthdays* технологически аналогично созданию класса *MyCelebration*. То же самое можно сказать о включении объектов класса *MyBirthdays* в класс *MyCalendar*.

Например, если вы для хранения праздников ввели массив, то достаточно расширить класс *MyCalendar* массивом, содержащим объекты класса *MyBirthday*. И далее реализуемые методы работы с днями рождения будут взаимодействовать с этим массивом. Но есть другое решение, которое упростит и уменьшит код приложения в целом. Оно состоит в использовании того факта, что в цепочке иерархии наследования любого создаваемого пользователем класса лежит стандартный системный класс *Object*. Поэтому, если объявить массив, в который ранее заносились праздники, типом *Object*, то теперь в него можно помещать объекты любых классов. В нашем случае объекты классов *MyCelebration* и *MyBirthdays*.

Если для хранения использовался класс списка *System.Collections.Generic.List* (или любой другой класс, поддерживающий список), то его члены, предназначенные для хранения объектов, уже имеют тип *Object* и туда могут помещаться объекты любых типов.

В чем же тогда разница между использованием массива, для хранения объектов или списка? Когда вы создаете массив, то определяете его конкретный размер. А, следовательно, при помещении в него очередного объекта вынуждены каждый раз проверять, не заполнен ли массив максимально и есть ли место для помещения нового очередного в него элемента. Если нет — то это не фатально, т.к. массив динамический и его в любой момент можно при надобности расширить. Но это надо реализовывать. Т.е. тратить на это ресурс времени. В предлагаемых средой программирования классах, реализующих списки, это делается автоматически. В этом состоит одно из преимуществ использования этих классов.

Мы обсудили, как используя один и тот же список или массив, организовать хранение в них объектов различного класса. Но есть еще одна проблема — распознавание: объект какого класса содержится в произвольно извлекаемом из списка или массива элементе? Здесь есть также два решения.

Первое — простое. Использовать оператор *is*, позволяющий определить принадлежность объекта к конкретному классу.

Второе — использовать механизм виртуальных методов. В этом случае надо будет создать ещё некий родительский класс общий для классов праздники и день рождения.

В заключении необходимо отметить предложенный поэтапный подход реализации задания. На первом этапе мы абстрагировались от реализации функционала, связанного с учетом дней рождения. Создали приложение в усеченном варианте по сравнению с заданием, реализовали его, отладили и убедились в корректности его работы. А на втором

этапе просто расширили его функционал в соответствии с заданием. Это принципиальный технологический подход, используемый при реализации различных приложений. Представьте себе, что теперь вы хотите добавить в класс календаря некоторые другие памятные даты. Методика уже вами отработана — пишите соответствующий класс и в классе календаря организуйте метод, возвращающий дату ближайшего праздника.

Вопросы для самоконтроля

1. Что такое агрегация (включение) объектов.
2. Что такое наследование?
3. Чем отличается наследование от агрегации.
4. Могут ли классе агрегированного объекта существовать члены класса одноименные с членами класса объекта агрегировавшего класса.
5. Класс ***B*** наследует класс ***A***. Класс ***C*** наследует класс ***B***. Сколько объектов будет создано при после выполнения оператора *object Ob = new B();* ?
6. Класс ***B*** наследует класс ***A***. Класс ***C*** наследует класс ***B***. Сколько объектов будет создано после выполнения оператора *object Ob = new C();* ?
7. Класс ***D*** агрегирует класс ***E***. Класс ***F*** агрегирует класс ***D***. Сколько объектов будет создано при после выполнения оператора *object Ob = new D();* ?
8. Класс ***D*** агрегирует класс ***E***. Класс ***F*** агрегирует класс ***D***. Сколько объектов будет создано при после выполнения оператора *object Ob = new F();* ?

Проект «Графический редактор»

Цель

Практическое закрепление базовых понятий ООП: класс, объект, поле, конструктор, метод. Закрепление механизмов сериализации (сохранения) коллекции объектов в файл.

Практическое освоение азов проектирования архитектуры классов, построения относительно сложного проекта с множеством классов, взаимодействующих между собой. Реализация интерфейса *IComparable* в создаваемом классе.

Практическое задание

Разработать проект «Графический редактор» — приложение, позволяющее рисовать несколько примитивных фигур (прием рисования – растягивание мышью), выбирать их цвет, сохранять рисунок в файл[5], выделять нарисованную фигуру щелчком мыши, перекрашивать выделенную фигуру, организовывать фигуры на рисунке по возрастанию/убыванию площади.

Разработать шесть классов:

1. *Shape* — класс-предок для всех графических примитивов.
2. *Rectangle, Triangle, Circle* (прямоугольник, треугольник, круг) и какой-либо собственный класс, описывающий любой графический объект, сложнее предыдущих: например, смайлик, домик или звезду.
3. *Collection* — класс, хранящий коллекцию нарисованных фигур и предоставляющий методы добавления фигур в коллекцию, сохранения коллекции в файл и загрузки из файла, а так же сортировки фигур.

[5] Усложнение задания – реализовать возможность сохранения в файл двумя способами: растровый (*.bmp, *.jpeg) и векторный. Для векторных рисунков придумать свой формат сохранения файла. Важно! Если файл загружен в векторном формате, то объекты можно выделять, перекрашивать, перетаскивать и т.д.

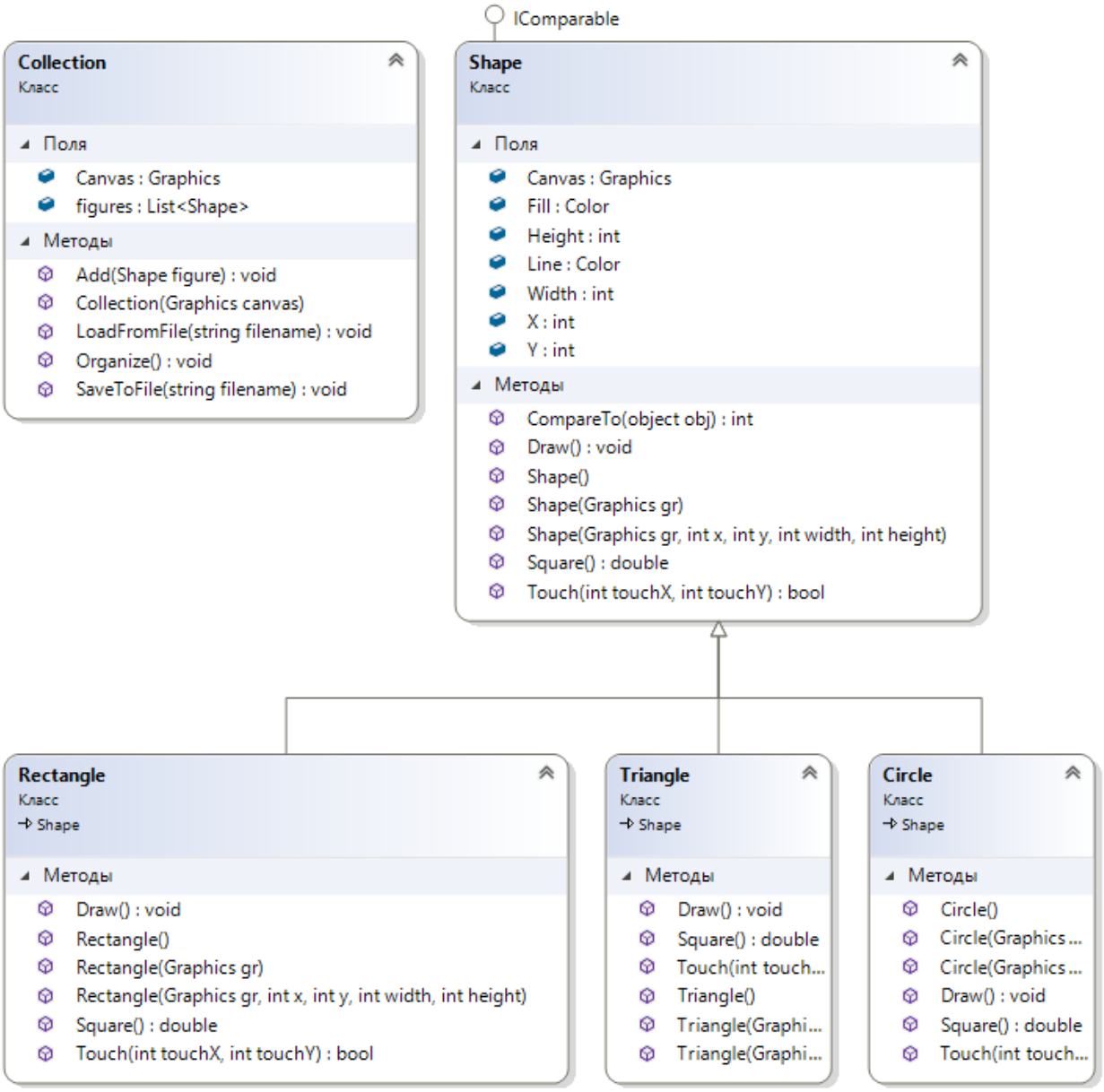

Рисунок 20. UML- диаграмма семейства классов для проекта "Графический редактор"

Поля и свойства класса Shape и его потомков:

Canvas — объект типа *Graphics*, на котором будут рисоваться все фигуры. Так называемый «холст». Связывается с компонентом формы *Panel* (или *PictureBox*).

Хотя лучше бы объект *Canvas* оформить в качестве параметра. Это даст возможность работать с конкретным экземпляром редактора в разных окнах. Иначе для каждого окна надо создавать свой индивидуальный экземпляр редактора.

X, *Y*, *Width* и *Height* — параметры объектов-фигур: координаты левого верхнего угла, ширина и высота фигуры.

Fill и *Line* — цвет заливки и цвет линии, соответственно.

По праву наследования, классы-потомки получат все незакрытые поля и свойства класса-предка. Необходимо позаботиться о том, чтобы они были объявлены, как *public* или *protected*.

Методы класса Shape и его потомков:

Shape() — три конструктора: первый – по умолчанию, без параметров; второй – принимает объект типа *Graphics*; третий – помимо *Graphics*, принимает координаты, ширину и высоту создаваемой фигуры. Конструкторы инициализируют значения свойств создаваемой фигуры: холст, координаты, размер, цвета линии и заливки. В классах-потомках следует вызывать конструктор предка, используя оператор *base()*.

Draw() — метод визуализации фигуры. В классе *Shape* этот метод объявлен как виртуальный (*virtual*) и не имеет реализации. В классах-потомках этот метод перезаписывается (*override*) – в нем реализуется рисование конкретных фигур на объекте *Canvas*: кругов, прямоугольников, треугольников, смайликов, домиков и т.д.

Touch(int touchX, int touchY) — метод проверяет факт попадания точки с координатами (*touchX, touchY*) в фигуру. В классе *Shape* этот метод объявлен как виртуальный (*virtual*) и не имеет реализации. В классах-потомках этот метод перезаписывается (*override*) – в нем реализуется алгоритм попадания точки в конкретный замкнутый контур[6]. Метод возвращает *true*, если точка, определенная координатами (*touchX, touchY*), принадлежит контуру конкретной фигуры. В противном случае, метод возвращает *false*.

Square() — метод, рассчитывающий площадь фигуры. В классе *Shape* этот метод объявлен как виртуальный (*virtual*) и не имеет реализации. В классах-потомках этот метод перезаписывается (*override*) – в нем реализуется алгоритм вычисления площади конкретной фигуры: круга, прямоугольника, треугольника – по соответствующим математическим формулам. Площади сложных объектов считаются по сумме площадей составляющих их фигур.

CompareTo(object obj) — метод сравнения объектов *Shape* между собой. Реализует интерфейс *IComparable*. Реализация этого интерфейса необходима для того, чтобы на форме можно было бы применить метод сортировки *Sort()* списка фигур *List<Shape>* для того, чтобы упорядочить ранее нарисованные фигуры.

Поля и свойства класса Collection:

[6] Для реализации алгоритма попадания точки в замкнутый контур рекомендуется применять функционал *GraphicPath*.

Canvas — объект типа *Graphics*, на котором будут рисоваться все фигуры коллекции. Так называемый «холст». Связывается с компонентом формы *Panel* (или *PictureBox*).

figures — закрытый (*private*) список, содержащий в себе все фигуры, добавляемые на рисунок.

Методы класса Collection:

Collection(Graphics canvas) — конструктор, создающий объект-коллекцию. Инициализирует объект *Canvas*, на котором будут рисоваться все объекты коллекции и создает список *figures*.

Add(Shape figure) — метод добавления фигуры в список *figures*.

SaveToFile(string filename) — метод сохранения коллекции фигур *figures* в файл, указанный в параметре *filename*. Для сохранения объекта необходимо использовать бинарную сериализацию в файловый поток[7].

LoadFromFile(string filename) — метод загрузки коллекции фигур *figures* из файла, указанного в параметре *filename*. Для сохранения объекта необходимо использовать бинарную десериализацию из файлового потока. Полученный из файла список фигур должен быть отражен на *Panel*.

Organize() — метод упорядочивания фигур по убыванию (или по возрастанию – тут решение остается за студентом).

Применение класса:

[7] Для бинарной сериализации в *C#* применяется тип *BinaryFormatter*.

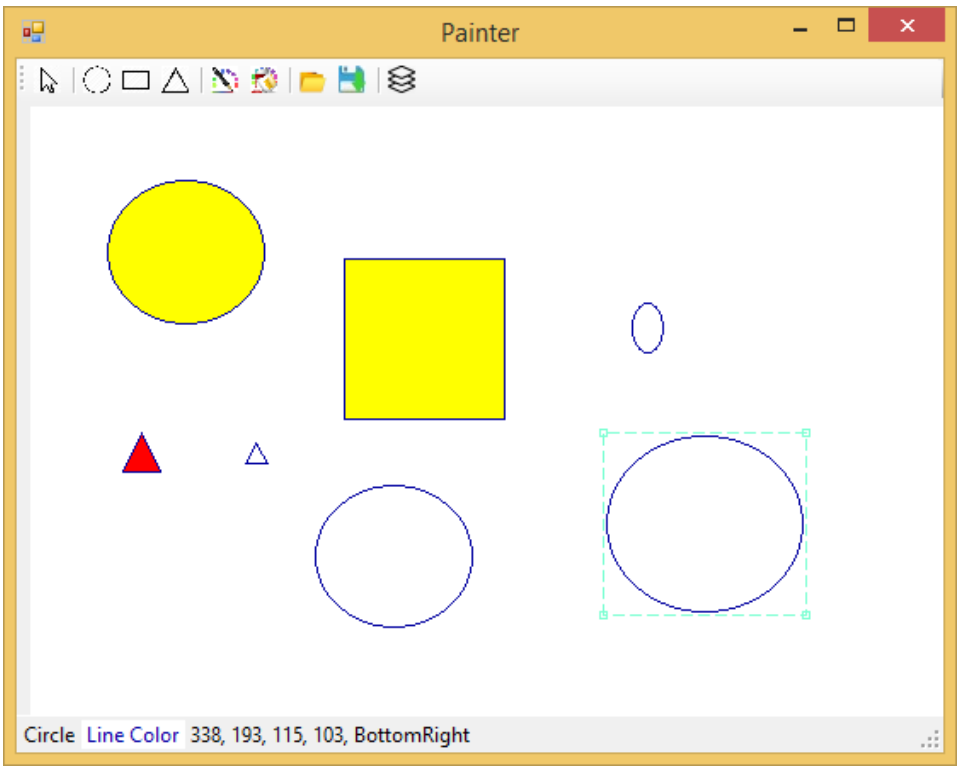

Рисунок 21. Форма проекта «Графический редактор»

На форме добавлен компонент – панель инструментов *ToolStrip*, на котором расположены кнопки: выбора фигуры, выбора цвета, сохранения и загрузки рисунка, а также кнопка, упорядочивающая фигуры на рисунке. Коллекция фигур отображается на компоненте *Panel*. Снизу формы предусмотрена статусная строка *StatusStrip* для вывода информации о рисуемой фигуре: координаты, цвета и т.д.

Сценарий рисования фигуры следующий: вначале выбирается тип фигуры на панели инструментов, затем на поле нажимается левая кнопка мыши, мышь движется и потом кнопка отпускается. Этими двумя точками: *MouseUp* и *MouseDown* – определяется размер рисуемой фигуры.

Для выбора цвета линии и цвета заливки фигуры применяйте *ColorDialog*.

Вопросы для самоконтроля

1. Что такое «наследование классов» в ООП? Для чего нужен этот механизм? Как объявляются классы-наследники?
2. Что такое виртуальные методы? Чем они отличаются от обычных (не виртуальных)?
3. Что такое «перегрузка виртуальных методов» в классе-потомке?
4. Что происходит с методами при наследовании? Как ведут себя методы класса-предка в классе-потомке: обычные, виртуальные?

5. Что будет, если в классе-потомке объявить метод, по сигнатуре полностью совпадающий с методом класса-предка?
6. Что будет, если в классе-потомке объявить метод, одноименный методу класса-предка, но разный с ним (методом) по сигнатуре?
7. Что такое «типизированный список» объектов? Как объявить такой список? Объекты каких типов могут быть добавлены в такой список?
8. Какие списки могут быть отсортированы встроенным методом сортировки *List.Sort()*? Что необходимо предпринять, чтобы типизированный список из объектов пользовательских классов мог быть отсортирован?
9. Как объекты сохраняются на диск? Что такое «сериализация» и «десериализация»? Какие виды сериализации бывают? Какие объекты не могут быть сериализованы? Как исключить некоторые поля объекта из сериализации?
10. Как организуется проверка попадания точки в замкнутую область фигуры?

Проект «Микроорганизмы»

Цель задания

Практическое решение комплексной задачи, закрепляющей не только владение основными базовыми элементами ООП, но и развивающими способность фрагментировать сложную задачу на отдельные меньшие не зависимые при реализации подзадачи.

Задание рассчитано на выполнение в несколько этапов.

Практическое задание

Моделирование жизни различных одноклеточных микроорганизмов в одной пробирке. Любой одноклеточный организм способен к размножению путем деления. Популяция может расти или уменьшаться со временем, перемещаться по пробирке.

Для простоты модель пробирки представляет собой двумерную (плоскостную) структуру, разделённую на квадратные ячейки. Размеры микроорганизма таковы, что он может занимать одну или более ячеек.

Каждый микроорганизм может совершать три действия:

- перемещаться в пределах пробирки;
- размножиться;
- гибнуть.

В пробирке находятся микроорганизмы трех видов: амебы, бактерии и инфузории. Количество и положение микроорганизмов задается изначально при запуске программы.

Моделирование осуществляется пошагово. Переход к следующему шагу моделирования происходит по нажатию пользователем клавиши или по истечению заданного периода времени секунд с момента предыдущего шага.

Текущее положение микроорганизмов отображаются на экране.

Исходные данные:

- Пробирка и ее размеры.
- Микроорганизмы.

Общие методические указания

Это комплексное задание достаточно сложное и объемное. Помните, любая самая сложная задача состоит из ряда более частных, которые, в свою очередь, так же могут быть представлены как еще более частные и т.д. Надо двигаться от более простого к более сложному шаг за шагом.

Начинаем с анализа поставленной задачи.

Есть пробирка — это отдельная сущность. Со своими параметрами. Со своим функционалом. Значит надо создавать такой класс. Назовем его, например, *MyTesttube*. Любое моделирование разбивает непрерывные процессы на дискретные шаги. Поэтому разделим модель пространства пробирки на квадратные ячейки, в которых будут находится и передвигаться микроорганизмы.

Есть различные типы микроорганизмов. Они имеют различные размеры и занимают определенные ячейки пробирки. При этом любая ячейка может быть занята только одним микроорганизмом. Микроорганизмы передвигаются. Каждый по своему алгоритму.

Возможна при передвижении попытка одного из них перейти на место уже занятое другим. Эта коллизия может разрешаться при моделировании различными способами. Можно, например, для такого микроорганизма пропустить передвижение на данном ходе (итерации). Это выглядит естественно — как в жизни: если мы с кем-то столкнулись, то замираем на время. Можно повторить попытку движения в другом направлении. Хотя здесь есть опасность (подумайте в чем опасность).

Микроорганизмы также могут по своим правилам размножаться и погибать со временем.

Т.к. каждый микроорганизм живет по своим правилам, естественно, надо для каждого из них создавать свой класс. Обратим внимание на то, что параметры и функциональность различных микроорганизмов во многом тождественны, а именно: все они имеют некие размеры, все находятся в каких-то определенных ячейках пробирки, все они двигаются, размножаются, погибают. Т.е. у них очень много общих признаков и действий. Значит с точки зрения реализации целесообразно создать базовый класс, который содержит эти общие для них элементы. Этот базовый класс будет наследоваться конкретным классом микроорганизма, а в последнем будут определены параметры, добавлена специфика и определены алгоритмы движения, деления и гибели конкретного микроорганизма.

При такой организации реализации, достаточно будет разработать решение для одного микроорганизма, а потом уже по образу и подобию легко добавлять остальные.

Реализацию задачи для одного микроорганизма целесообразно тоже разбить на ряд этапов.

Для выбранного микроорганизма нужно создать отдельный класс, наследующий базовый класс. В этом классе научить его помещаться в пробирку в указанное ему место.

Реализовать отображение пробирки и выбранного микроорганизма. Отладить правильность кода на этой стадии. Это обеспечит визуальный контроль корректности нашей реализации.

Далее нужно научить микроорганизм двигаться и отображать его движение. При движении необходим контроль за тем, чтобы микроорганизм не выходил за пределы пробирки. Тут возникает принципиальный вопрос: как производить этот контроль? Кто должен реализовывать эту функцию проверки: пробирка или микроорганизм? Это достаточно сложный типичный практический вопрос, постоянно появляющийся в решении практических задач. Т.е. при наличии взаимодействия двух классов какой из них должен реализовывать это взаимодействие? Подумайте над этим, примите свое решение, представьте его преподавателю, обоснуйте свое решение и послушайте его соображения и предложения.

Базовый класс *MyCell* с общими для всех клеток виртуальными методами: делиться, перемещаться, отобразиться на экране.

Классы-наследники: *MyAmoeba*, *MyBacteria*, *MyInfusoria*. Объекты этих классов по-разному отображаются на экране, по-разному перемещаются по пробирке.

MyAmoeba имеет форму большого шара, движется быстро в случайном направлении;

MyBacteria имеет форму маленького эллипса и медленно движется в случайном направлении,

MyInfusoria имеет прямоугольную форму и движется по прямой.

Для каждого вида микроорганизма скорость передвижения постоянна, но различна для разных видов микроорганизмов. Предлагается определять скорость движения величинами смещениями по осям за шаг. Вероятность деления и завершения жизненного цикла микроорганизма также индивидуальны для отдельного вида, но различны для разных видов.

Этап 1. Создание пробирки, одного микроорганизма и их отображения

Необходимо разработать класс пробирка *MyTesttube*, содержащий:

- поля описания границ пробирки;
- массив, содержащий объекты микроорганизмы;
- конструктор;
- метода помещения начального набора микроорганизмов;
- метода, реализующего очередной шаг жизни микроорганизмов.

Разработать базовый класс описания микроорганизма *MyCell*, содержащий:

- поля *X* и *Y* координат нахождения микроорганизма в пробирке;
- поля *DX* и *DY* – приращения координат *X* и *Y* для организации перемещения микроорганизмов при каждом шаге;
- конструктор;
- виртуальный метод перемещения микроорганизма.
- виртуальный метод визуализации микроорганизма;

Разработать один из классов-наследников, например, *MyAmoeba*. Класс должен переопределять (*override*) виртуальные методы базового класса.

Создать приложение, создающее 10 микроорганизмов и отображающие их на экране.

Методические указания

Класс пробирка *MyTesttube*. Удачным выглядит решение организовать отображение пробирки на базе отдельной панели. При этом конструктор в качестве формального параметра получает объект типа панель и из него определяет ее размеры по вертикали и горизонтали. Такая реализация удобна тем, что при изменении размеров и положения панели на форме автоматически можно настраивать и все остальные параметры создаваемых классов. На этом этапе можно в конструкторе определить и количество изначально задаваемых амеб.

Базовый класс *MyCell* содержит вышеперечисленные поля и не требующие реализации виртуальные методы.

Класс амебы *MyAmoeba* наследует класс *MyCell*. Конструктор лучше создавать с параметрами, включающими координаты положения амебы в пробирке, ведь эта характеристика амебы определяется пробиркой. А поля, определяющие скорость и направление движения — это относится к самой амебе. Поэтому их место в этом классе.

Этап 2. Создание, визуализация, движение микроорганизмов

Необходимо разработать один из классов-наследников: *Infusoria, Bacteria*. Классы должны переопределять (*override*) виртуальные методы базового класса *Cell*.

Создать приложение, создающее (в общем массиве) 20 разных микроорганизмов и рисующее их на экране.

Добавить в приложение перемещение микроорганизмов:

- Определить метод перемещения для каждого из микроорганизмов
- Задать алгоритм перемещения для каждого из типов микроорганизмов. Убедиться, чтобы микроорганизмы не могли вылезти за края пробирки.
- Задать основной цикл программы, который перемещает микроорганизмы, поочередно вызывая метод перемещения для каждого из них.
- Добавить в приложение таймер, на каждое срабатывание которого исполнять цикл перемещения микроорганизмов, а также визуализацию их в новом положении.

Этап 3. Реализация размножения

Добавить виртуальный метод деления в *Cell*. Определить его для классов-наследников.

Добавить в основной цикл программы деление всех микроорганизмов.

Методические указания

Метод должен возвращать новый объект — новый микроорганизм, получившийся после деления исходного. Координаты нового микроорганизма должны отличаться от координат породившего его микроорганизма (чтобы они не сливались).

Вопросы для самоконтроля

1. Назовите преимущества объектно-ориентированного подхода по сравнению со структурным программированием.
2. Перечислите недостатки ООП парадигмы.
3. Что такое раннее и позднее связывание?
4. Что подразумевается под свойствами в *C#*?
5. Что такое упаковка и распаковка в ООП?

Проект «Тараканьи бега»

Цель задания

Практическое решение комплексной задачи:

- закрепляющей владение основными базовыми элементами ООП;
- обучающей умению проектирования и разработки структуры данных, состоящей из массива объектов класса;
- развивающей способность фрагментировать сложную задачу на отдельные подзадачи, реализуемые классами.

Практическое задание

Моделирование процесса движения объектов («тараканов», условно показанных как шарики) по своим дорожкам.

Каждый шарик:

- обладает своей скоростью и цветом;
- скорость изменяется (может изменяться) в некоторые моменты времени независимо для каждого шарика.

В проекте должен быть реализован контроль времени прохождения дорожки для каждого шарика, а также определения лидера данного запуска.

На форме должны быть изображены десять дорожек, на каждой из которых свой шарик. По нажатии кнопки «Старт» шарики начинают движение к вертикальной черте (линии) в правой части формы, которая обозначает финиш. Для каждого шарика непрерывно должно отображаться его положение и время его движения. При достижении шариком финишной черты – его время останавливается. При достижении всеми шариками финиша – справа от финиша должны расставиться места, ими занятые.

Исходные данные

Количество шариков не меньше 10 (можно реализовать вариант ввода с экрана их количества пользователем).

Общие методические указания

В данной задаче есть две явно выделяющиеся сущности:

- единичный объект таракан;
- объединение отдельных тараканов в общем забеге.

В соответствии с этим создать два класса: первый описывает параметры и алгоритмы движения отдельного таракана, второй – параметры и алгоритмы их движения в рамках отдельного забега.

Необходимо реализовать проект, который будет совмещать в себе разработку этих классов.

Создать класс *Cockroach*, который должен содержать следующие поля:

- Координату *Y* объекта-шарика
- Координату *X* объекта-шарика
- *Radius* радиус объекта-шарика (удобнее всего сделать одинаковое значение для всех объектов)
- *Distant* длину дистанции (как и с радиусом – можно сделать одинаковой для всех шариков)
- *Speed* скорость движения шарика. При умножении на время движения дает новую координату. Тогда можно добавить еще одно поле – момент изменения скорости. У каждого шарика он будет свой, естественно
- *BugColor* цвет шарика
- *BugBrush* кисточка для закрашивания шарика его цветом
- *BrushColorBackGround* кисточка для закрашивания шарика цветом фона – необходима при перерисовке местоположения шарика
- *ActiveRectangle* область (прямоугольник) для рисования текущего положения шарика
- *PreviousRectangle* область (прямоугольник) для сохранения предыдущего положения шарика
- *Time_of_Finish* время окончания движения шарика.

Методы класса *Cockroach* без описания параметров:

- *Cockroach()* конструктор, задающий случайные значения для некоторых полей (скорость, цвет и т.д.) и определенные значения (см. описание выше).
- *Move()* расчет новой координаты (X) шарика, возможного изменения скорости и визуализация нового положения шарика.
- *ReRectangle()* расчет новых координат прямоугольных областей.
- *ReSpeed()* расчет нового значения скорости.
- *GetTimeOfFinish()* возвращение значения времени пробега шарика.

Создать класс *Race*, который должен организовывать процесс игры.

Данный класс должен быть агрегированным – он должен включать массив объектов (с их инициализацией) класса *Cockroach*, а также массив хранения времени пробега каждого шарика.

Методы объекта класса *Race* без описания параметров:

- *SetNCockroach() – установка/изменение количества участников забега.*
- *Race()* –создание объектов массива *Cockroach* со всеми необходимыми настройками (скорости, цветов и т.п.), инициализировать массив хранения пробегов;
- *Field()* – должен реализовать игровое поле (например, рассчитать ширину дорожки в зависимости от количества объектов (элементов массива) *Cockroach*, определить их длину (она может отличаться от длины забега *Distant*), нарисовать и т.п.);
- *Start()* – запуск игры: движение (вызов методов перерисовки шарика) и проверка окончания движения (вызов метода *Stop()* при окончании движения очередным шариком). По окончании движения всех шариков вызвать метод *Result()*;
- *Stop()* – метод заполнения очередного элемента массива хранения времени пробега. Вызывается, когда очередной шарик заканчивает бег
- *Rezult()* – метод окончания сеанса игры и расчета занятых «мест» каждым шариком;
- *Sort()* – метод сортировки времен пробега каждого шарика и определения каждому шарику занятого место.

В пространстве *Form1*:

- Создать массив *TextBox*-ов – для показа времени пробега каждого таракана.
- Создать необходимые кнопки для запуска и окончания игры (вызова соответствующих методов класса *Race*).
- Создать кнопку показа результатов «соревнований».

Методические указания

Данный проект может быть разработан по структуре, описанной выше. Однако, вы можете сделать и свое решение. Требования к своему решению – наличие массива объектов вашего класса, а также агрегированного класса.

Рисунок 22 и Рисунок 23 иллюстрируют одну из практических реализаций задания. В этой реализации студент постарался и у него получилось сделать красивый интерфейс. Но это уже на любителя. На того, кто хочет сделать не только правильно, но и красиво.

Но суть задания не в товарном виде интерфейса. Если вы реализуете это и с более простым интерфейсом – задание выполнено. Здесь основной упор сделан на использовании ООП.

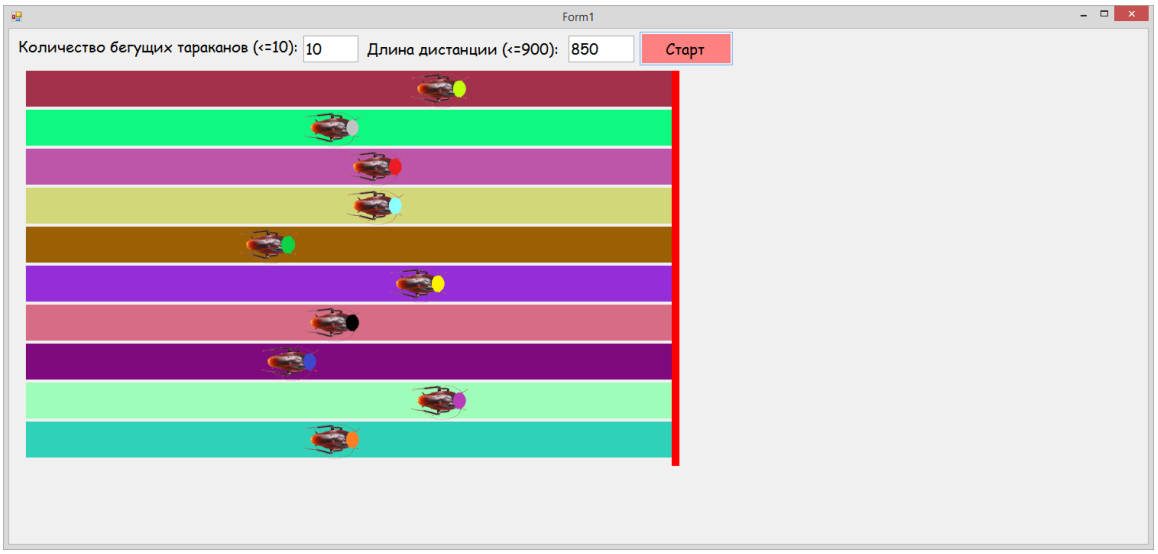

Рисунок 22 Скриншоа забега

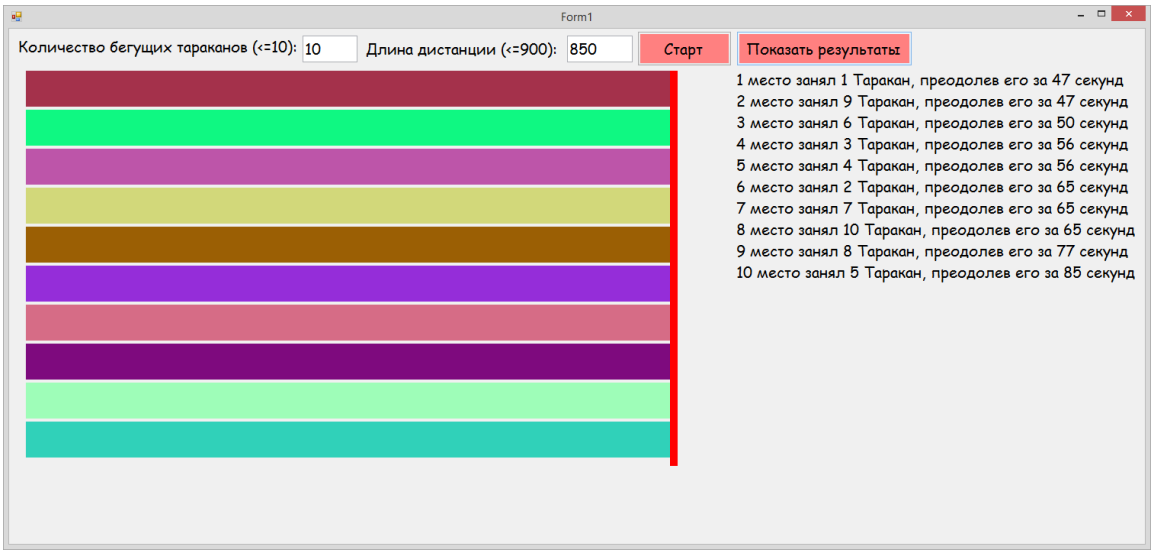

Рисунок 23 Скриншот результатов забега

Рисунок 24 – скриншот еще одной реализации. Может не такой яркий, но тоже красивый.

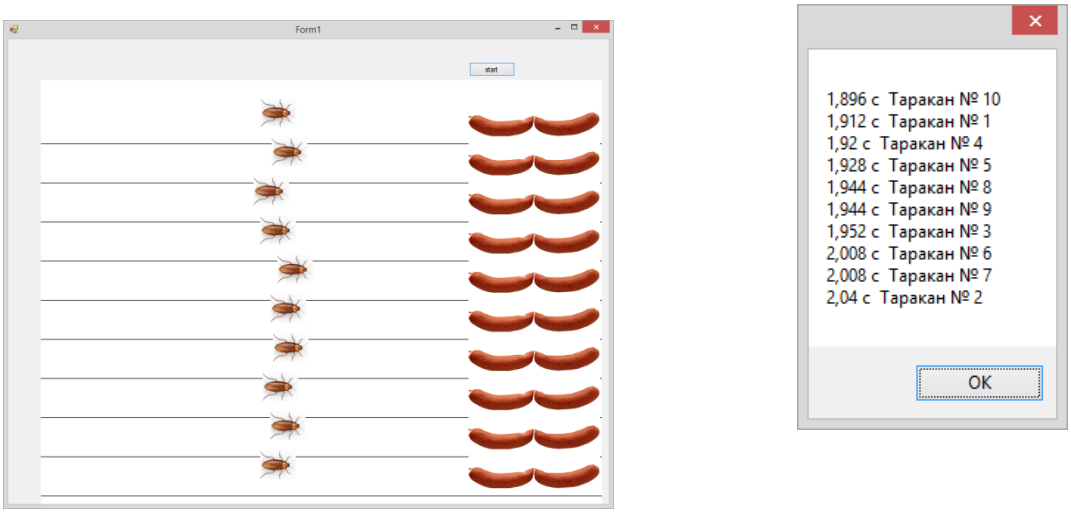

Рисунок 24 Скриншот реализации

Вопросы для самоконтроля

1. Как создать массив объектов класса?
2. Как задать определенные значения элементам массива объекта класса?
3. Что такое агрегированный класс?
4. В каком порядке создаются объекты агрегированного класса – внутреннего класса и непосредственно самого класса?
5. Каким образом взаимодействуют объекты *Form* и методы объектов вашего класса?
6. Данная игра может/должна предоставлять возможность повторных сеансов игры в рамках одного запуска приложения. Нужно в этом случае использовать вызов каких-либо деструкторов в явном виде? Если да - то почему? Если нет - то почему?

Проект «Танчики»

Цель задания

Получение навыка создания приложения, объединяющего комплекс взаимосвязанных объектов различных классов. Проектирование и построение архитектуры достаточно сложного программного приложения. Практическое освоение работы со структурами.

Практическое задание

Разработка игры, в которой сражаются 2 танка на плоскости. Управление каждым происходит автоматически на основе заданного алгоритма.

Индивидуальные параметры танка следующие:

- определенное количество жизнестойкости (жизни);
- определенный типом брони;
- вес брони;
- набор снарядов;
- скорость передвижения;
- местоположение на карте (координаты).

Типов брони три:

- тяжелая (самая прочная, но самая тяжелая);
- средняя (средняя по прочности и весу);
- легкая (менее прочная, но самая легкая).

Прочность брони определяет процент наносимого урона танку, а вес – скорость его передвижения. Чем более тяжелая броня, тем меньше скорость передвижения.

Типов снарядов три:

- Дальнего поражения (летит далеко, но минимальный урон).
- Среднего поражения (средняя по дальности и урону).
- Ближнего поражения (летит не далеко, максимальный урон).

Боекомплект танка не превышает 10 снарядов. Формирование боекомплекта производится случайным образом. При окончании боекомплекта производится его перезарядка, на которую тратится очередной ход.

Игра заканчивается, когда в живых остается один танк.

Исходные данные

Два танка (минимум), каждый из которых совершает действия на основе заданного алгоритма.

Этап 1. Разработка интерфейсов и структур

В задании предлагается освоить работу со структурами. Структуры, как правило, используются для небольших по размеру типов данных с целью повышения эффективности и производительности программ. Такое использование структур обусловлено тем, что они являются значимым, а не ссылочным типом данных. Структуры хранятся в стеке вызова и обращение к ним идет напрямую в отличие от классов, в которых обращение к объекту идет с помощью ссылки на участок динамической памяти. Так же структуры имеют преимущество перед набором переменных в том, что они представляют собой группу логически связанных вместе данных, которые хранятся под одним именем.

В задании с помощью структуры предлагается реализовать типы снарядов и брони. Для них объявляется интерфейс, который определяет общие свойства структур – дальность и урон для снарядов и вес и прочность у брони. *UML* диаграмма интерфейсов и структур представлена ниже (см. Рисунок 25).

Для того, чтобы отсутствовала возможность изменения параметров структур можно создавать свойства только с аксессором *get*, который будет возвращать конкретное значение.

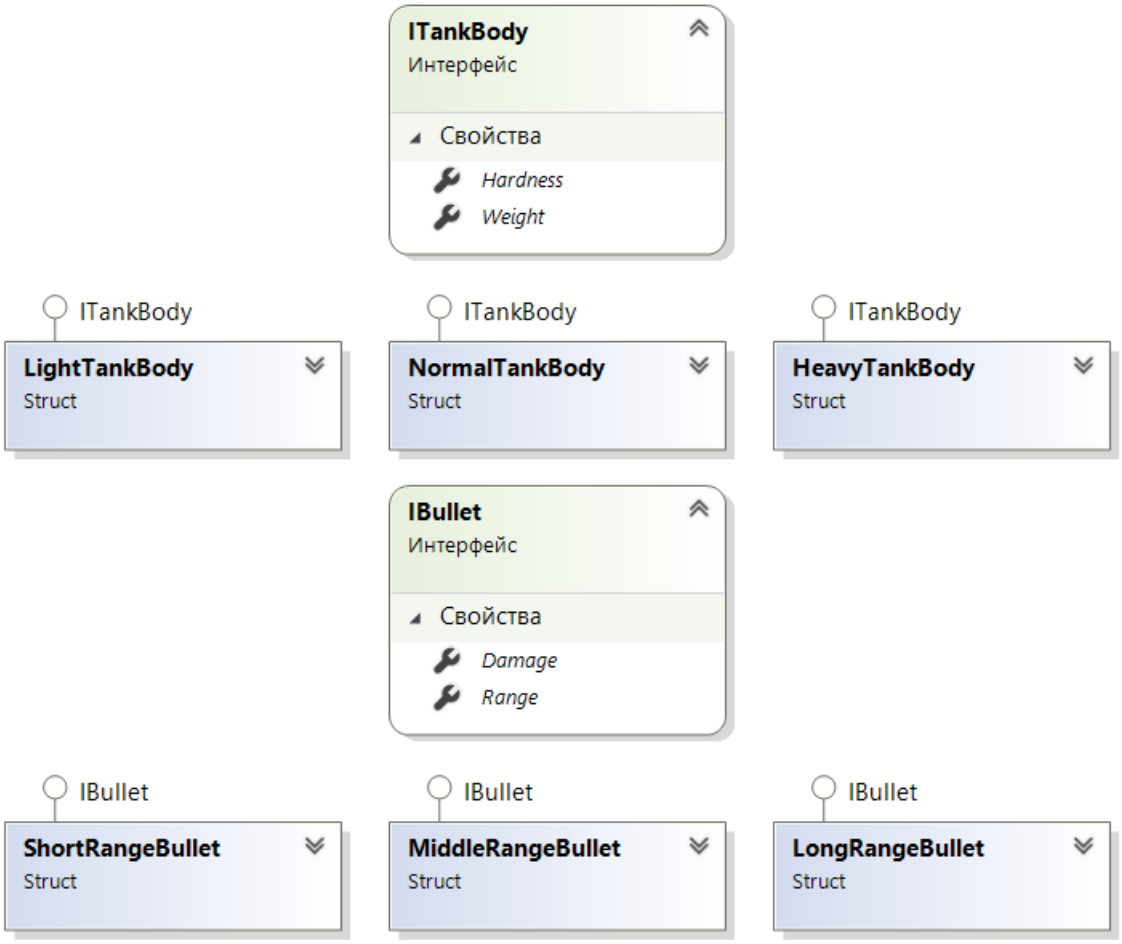

Рисунок 25. UML диаграммы структур и интерфейсов для видов снарядов и брони

В качестве примера значений для всех типов брони и снарядов можно использовать значения, представленные ниже (см. Таблица 1 и Таблица 2Таблица 1).

Таблица 1. Значения для типов брони

Тип брони	**Вес**	**Прочность**
Легкая броня	300	0,2
Средняя броня	600	0,6
Тяжелая броня	900	0,9

Таблица 2. Значения для типов снарядов

Тип снаряда	**Урон**	**Дальность**
Ближнего поражения	80	20
Среднего поражения	50	60
Дальнего поражения	25	100

Этап 2. Разработка класса *Tank*

Класс танка определяется значением жизнестойкости, местоположения, экземпляром типа брони, набором имеющихся снарядов и рядом методов (см. Рисунок 26). Начальное значение жизнестойкости у всех танков одинаково – 100 единиц.

Набор снарядов можно реализовать с помощью массива, списка или очереди. При создании экземпляра танка тип брони определяется случайным образом, а координаты передаются как входные параметры.

Скорость передвижения задается свойством *Speed* (зависимость от веса брони) и определяет расстояние, которое танк может пройти за один ход.

Текущий боекомплект задается методом *Reload,* который очищает текущую коллекцию снарядов и добавляет новую.

Очередной снаряд извлекается из боекомплекта и отправляется в противника методом *Shoot*.

Успех выстрела обрабатывается методом *GetHit*, который в качестве параметров принимает тип снаряда и расстояние с которого произведен выстрел. Выстрел может попасть в цель либо промахнуться. Успех попадания зависит от расстояния между танками – чем больше расстояние, тем меньше вероятность попадания.

Метод *GetHit* принимает прилетевший в танк тип снаряда и расстояние пролета снаряда, Метод определяет вероятность попадания и величину жизнестойкости после обстрела.

Вероятность попадания и нанесенный урон предлагается вычислить по следующей формуле:

$$\frac{range}{distance} * rnd.Next(20) > 13, \quad \text{где:}$$

range – дальность полета снаряда;

distance – расстояние, преодоленное снарядом;

rnd.Next(20) – случайное значение в диапазоне от 0 до 19.

Константу для определения вероятности попадания можно варьировать.

Формулу расчета нанесенного урона можно записать следующим образом:

$$Health -= bullet.Damage * (1 - body.Hardness);$$

Метод *Move* сдвигает танк в указанном направлении на значение свойства *Speed*.

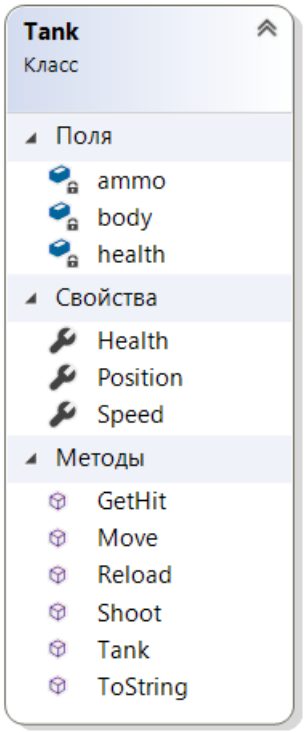

Рисунок 26. UML диаграмма класса Tank

Этап 3. Разработка класса *TankController*

Класс *TankController* предназначен для управления объектом танка, его отображением, алгоритмом поведения на каждом ходу. Диаграмма класса представлена ниже (см. Рисунок 27).

Конструктор данного класса принимает 2 параметра – координаты изначального расположения танка.

Поле *tank* является экземпляром класса *Tank*.

Свойство *Alive* возвращает логическое значение, обозначающее жив ли танк.

Свойство *Position* возвращает положение текущего танка.

Метод *Draw* отображает танк на игровом поле. Для отображения состояния танка можно, например, менять его цвет от зеленого до красного. Зеленый цвет – жизнестойкость танка максимальная, красный цвет – нулевая.

Метод *Hit* принимает снаряд как входной параметр и вызывает метод *GetHit* у экземпляра танка.

Метод *TakeTurn* описывает алгоритм поведения танка. Для примера предлагается следующий алгоритм:

1. Если нет патронов – перезарядка.
2. Если патроны есть – проверка на дистанцию попадания верхнего патрона до врага.
    a. Если снаряд может долететь будет – выстрел.
    b. Если снаряд не может долететь – движение в случайную сторону на плоскости, движение происходит по координатам *X* и *Y*.

Метод *TakeTurn* можно сделать виртуальным и сделать несколько классов-контроллеров с разными алгоритмами поведения, наследованными от класса *TankController*.

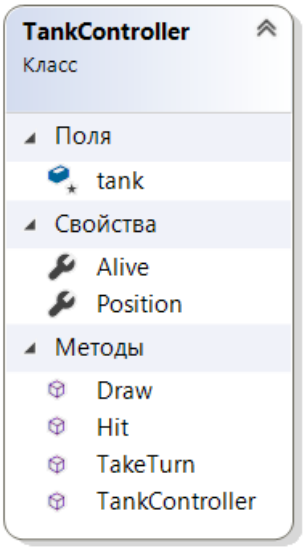

Рисунок 27. UML диаграмма класса TankController

Этап 4. Разработка класса *Game*

Класс *Game* отвечает за логику всей игры. Представленный пример рассчитан на двух игроков. Описание класса в виде *UML* диаграммы представлено ниже (см. Рисунок 28).

Поля *player1* и *player2* – экземпляры класса *TankController*.

Поле *turnOne* типа bool определяет чей сейчас ход.

Метод *CheckWin* проверяет живы ли игроки. Если остался только – то возвращает значение *true*, иначе – *false*.

Метод *Draw* обеспечивает визуализацию всех игроков.

Поле *NextTurn* определяет ход по полю *turnOne* игрока, вызывает у соответствующего игрока метод *TakeTurn* и изменяет переменную *turnOne.*

Рекомендуется создать конструктор, в котором будут указаны границы игрового поля. Там же будут задаваться изначальные настройки игры, создаваться экземпляры игроков и т.д. При создании игроков стоит передать входные параметры расположения танков, которые будут генерироваться случайным образом в диапазоне указанного игрового поля.

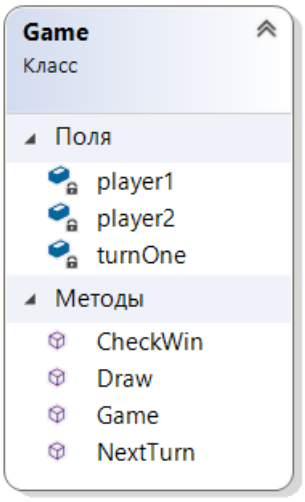

Рисунок 28. UML диаграмма класса Game

Методические указания

Данный проект может быть разработан по структуре, описанной выше. Однако, вы можете сделать и свое решение. Требования к своему решению – использование структур для решения задачи. На рисунке ниже представлен простой графический интерфейс готового проекта (см. Рисунок 29).

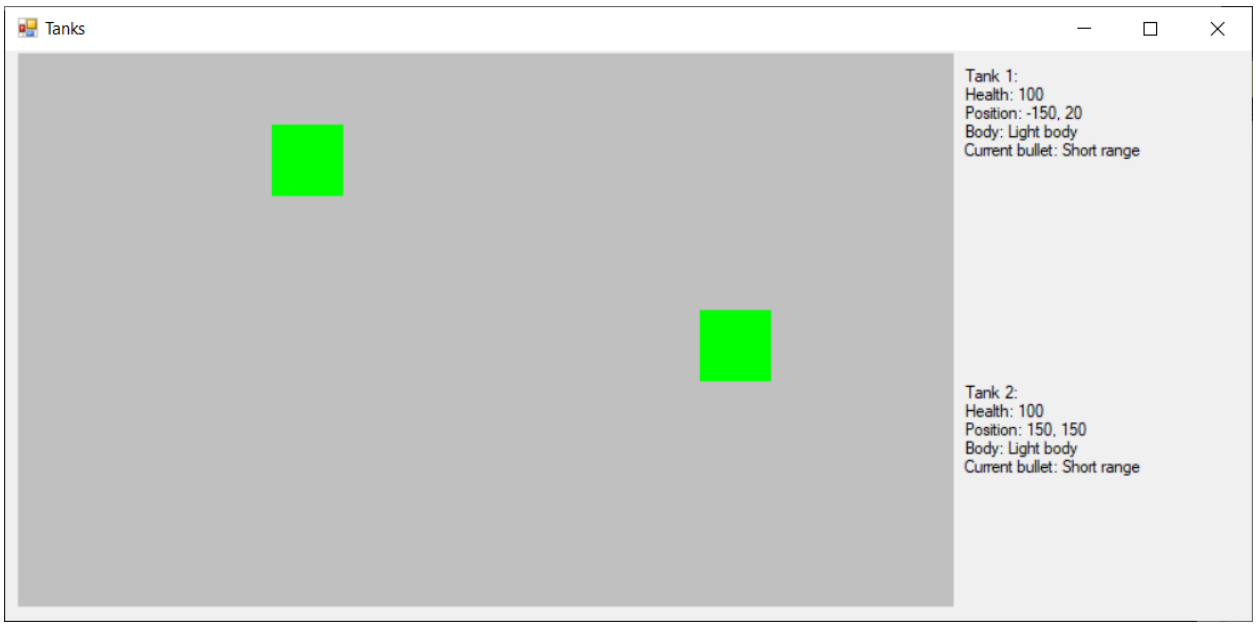

Рисунок 29. Возможный интерфейс программы

Данная игра имеет интересный потенциал для своего развития.

Например, можно реализовать различные алгоритмы тактики ведения боя для тяжелых, средних и легких танков. Ввести возможность в класс *Game* при запуске выполнять не единственный бой, а автоматически заданное количество сражений подряд и сравнить успешность того или иного алгоритма тактики по соотношению побед и поражений. Другими словами, это уже будет задача не просто из области освоения навыков ООП, а из области создания искусственного интеллекта! А это интересно!

Можно и посоревноваться с товарищем, загрузив разработанные им методы ведения боя в один танк и свои в другой. И выяснить кто из вас будет выдающимся полководцем, а кто великим! ☺

Дерзайте! Развивайте свое мышление, свои навыки, свои знания!

Вопросы для самоконтроля

1. Что такое структура?
2. Чем структура отличается от класса?
3. В чем особенность передачи экземпляра структуры как входного параметра?
4. В каких случаях выгоднее использовать структуры, а в каких классы?

Список литературы и источников

1. Краткий обзор языка C# https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/
2. Сведения о проектах и решениях https://docs.microsoft.com/ru-ru/visualstudio/get-started/tutorial-projects-solutions
3. Знакомство с отладчиком Visual Studio https://docs.microsoft.com/ru-ru/visualstudio/debugger/debugger-feature-tour
4. Руководство по программированию на C# https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide
5. Сериализация https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/serialization/
6. Васильев А. C# Объектно-ориентированное программирование: Учебный курс. – СПб.: Питер, 2012. – 320 с.: ил.