Свою работу я начинаю с составления архитектуры.

Требования:

1. Отображение фигуры по входящим математическим параметрам:

* количество лучей;
* радиус окружности внешних вершин;
* радиус окружности внутренних вершин;
* угол поворота;
* координаты центра.

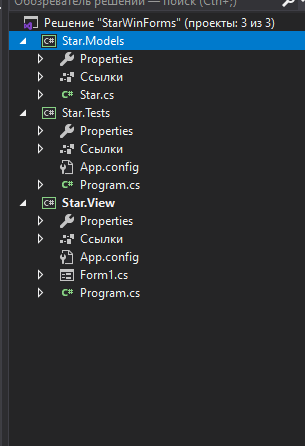
1. Отображение фигуры по входящим графическим параметрам:

* цвет линии.

1. Наличие тестов для проекта:

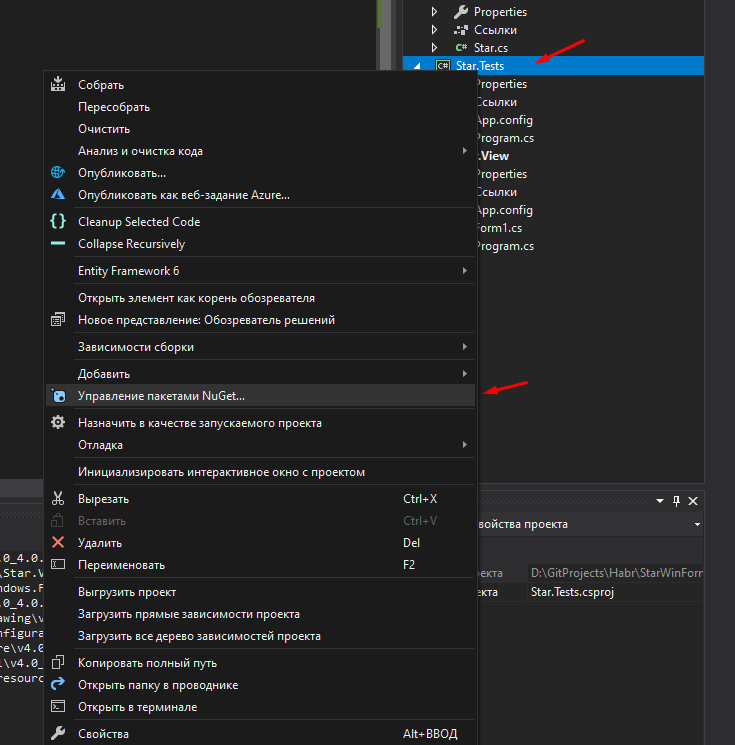
* Тесты должны поставить хорошие и плохие ситуации для модели

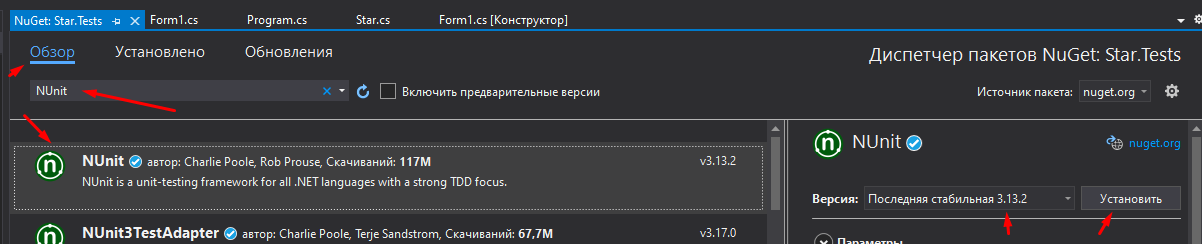
Создаем файл проекта и добавляем в него три решения (WindowsForms с названием Star.View, библиотека классов с названием Star.Models и консольное приложение с названием Star.Tests):



Почему я разделил все на три части? Потому что тесты должны тестировать только модели, а графическое отображение будет работать верно, если модель работает верно. Отсюда разделение на три проекта – тесты будут ссылаться только на Star.Models.

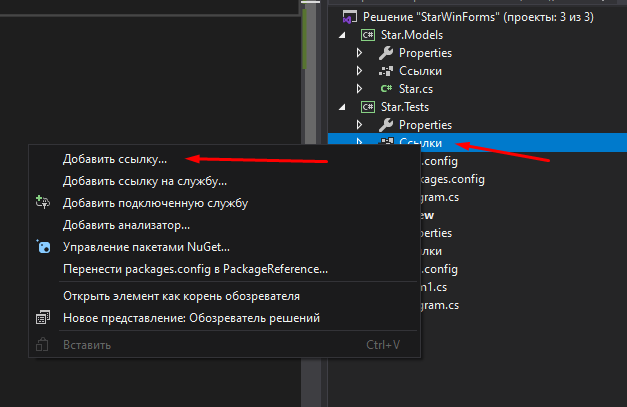
В Start.Tests добавляем пакет NUnit через менеджер пакетов Nuget:

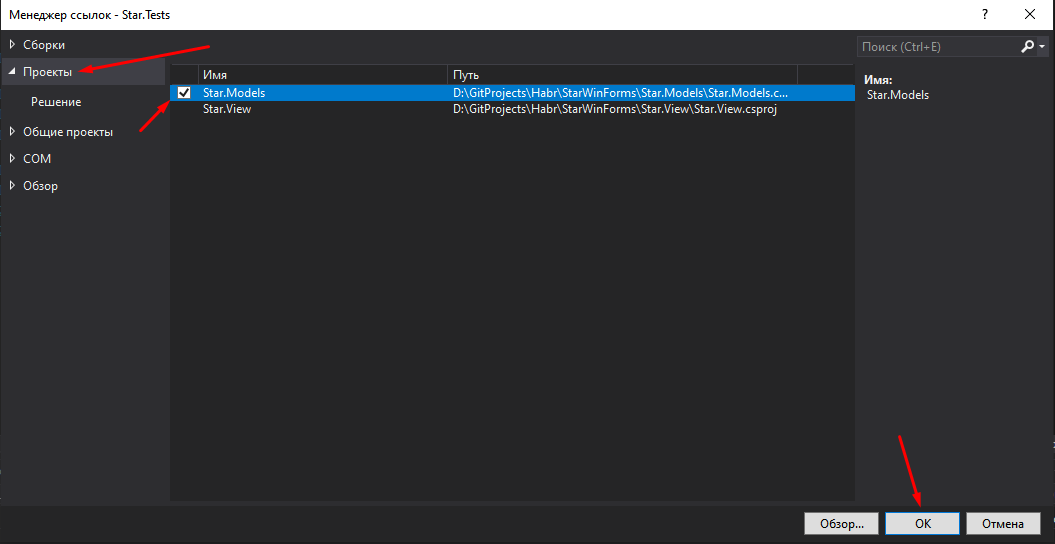




Это добавит в проект библиотеки, связанные с тестированием. Благодаря ним можно будет написать тестовый метод, пометить его аттрибутом [Test] и нам будет доступен его прямой вызов, без запуска консоли, без дополнительных телодвижений. Нажали на кнопку рядом с методом, которая в будущем появится, нажали на отладку или запуск и наслаждаемся результатом.

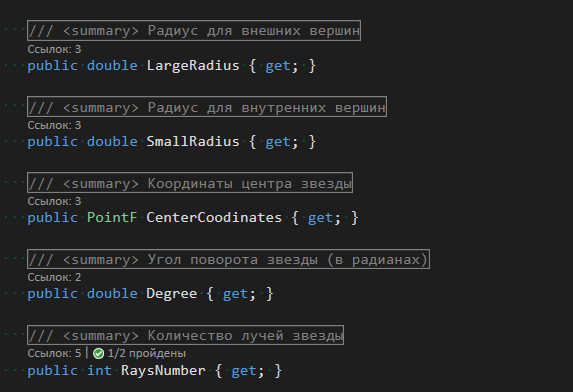
Теперь в Star.View и Star.Tests добавим ссылку на проект Star.Models, так как и там и там эта сборка будет использоваться. По шагам для Star.Tests:



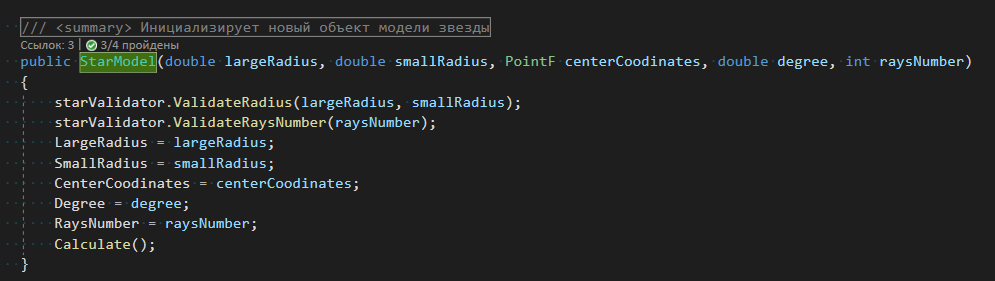


Аналогично делаем для Star.View.

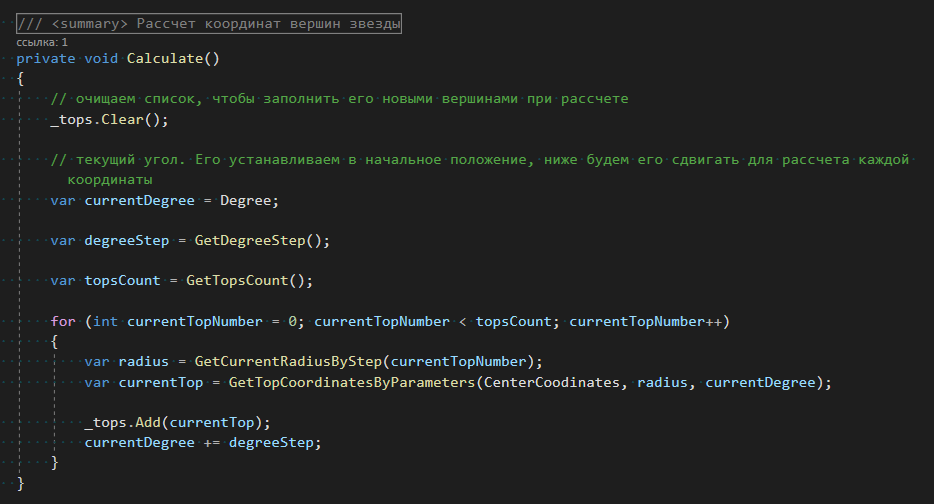
В Star.Models (далее - модели) создаем класс StarModel, это модель звезды. Она содержит такие свойства:

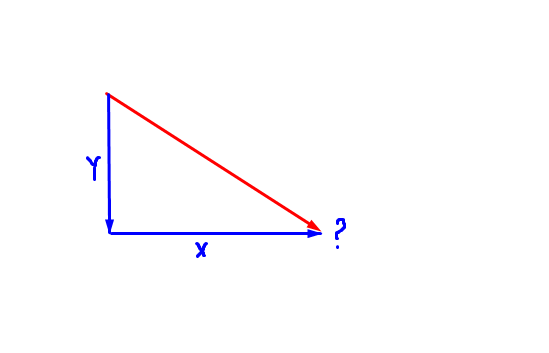


Все они настраиваются через конструктор:



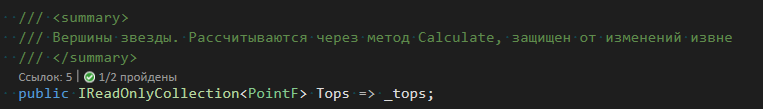
Также в конструкторе рассчитываются все вершины звезды:

Алгоритм прост. Получаем количество вершин исходя из количества лучей (вершин вдвое больше), высчитываем шаг сдвига луча для каждой вершины, и начинаем расчет каждой вершины. Идти будем по кругу против часовой стрелки (так как у нас угол измеряется в радианах). Для каждой вершины сначала определяем текущий радиус. Так как мы идем последовательно, то радиус каждой последующей вершины будет противоположным предыдущему. То есть если текущая вершина находится на расстоянии большего радиуса от центра, то следующая будет на расстоянии меньшего радиуса. Итак, что имеется, для расчета текущей вершины? Это начальные координаты, направление в виде угла (получается луч), и теперь нам нужно ограничить этот луч по длине, и тогда получаем отрезок, второй конец которого и будет нашей вершиной. Если представить, что каждая координата от вершины - это отдаление от центра по оси X или Y, тогда радиус будет гипотенузой получившегося прямоугольного треугольника. Составив проекции на каждую из плоскостей, все станет понятно:

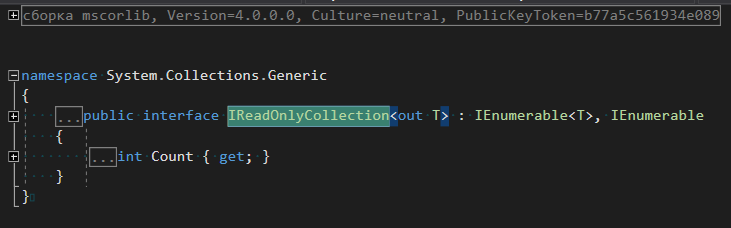


Рассчитать каждый катет по длине гипотенузы и угла не составляет труда.

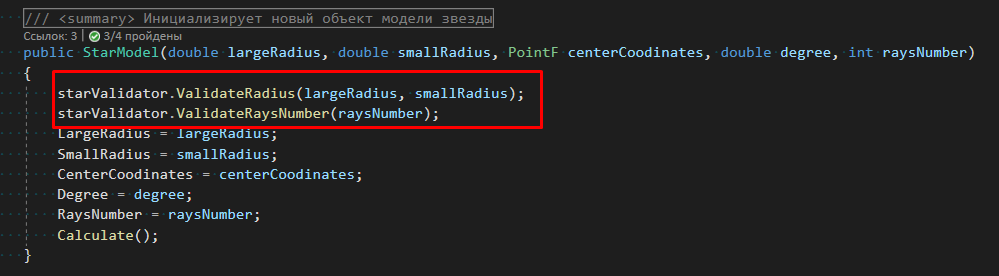
Итак, мы получили координаты каждой вершины, отправляем их в поле с коллекцией вершин, а свойство для доступа извне будет ссылаться на это поле:



Здесь типом коллекции является IReadOnlyCollection, он нужен для того, чтоб извне нашу звезду не могли исправить. Вся логика по расчету вершин звезды содержится только в звезде. Вот так выглядит этот интерфейс:



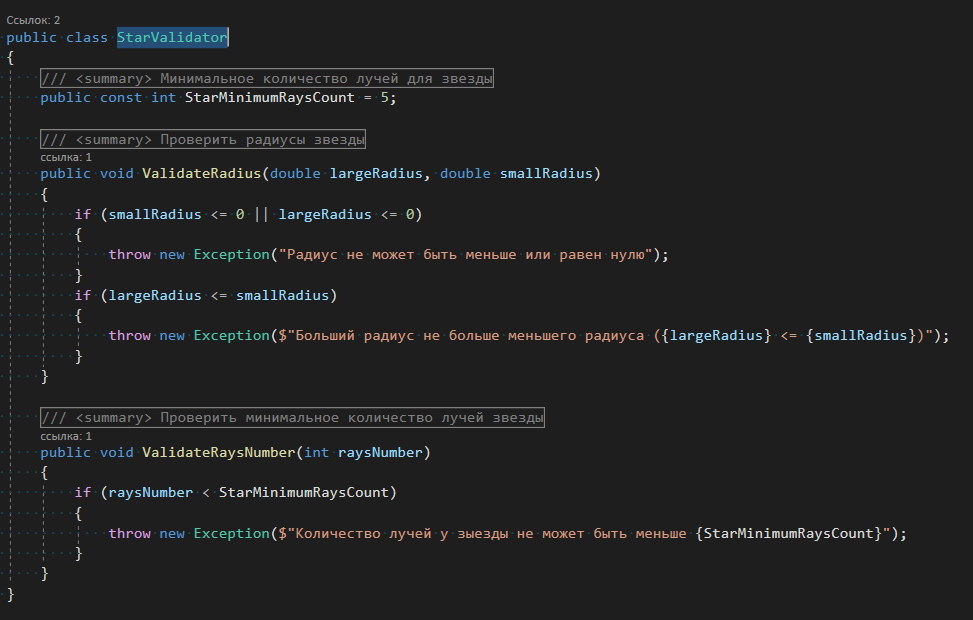
Enumerable, от которого он наследуется, не имеет методов доступа к коллекции. А значит, нашу коллекцию с вершинами никто извне класса не изменит, но внутри мы можем спокойно менять их через поле. Если присмотреться к модели звезды, то можно увидеть, что она “одноразовая”. Это значит, что извне звезду можно только создать и удалить, ничего больше.   
Я намеренно пропустил две строки в конструкторе, пришло время объяснить их:



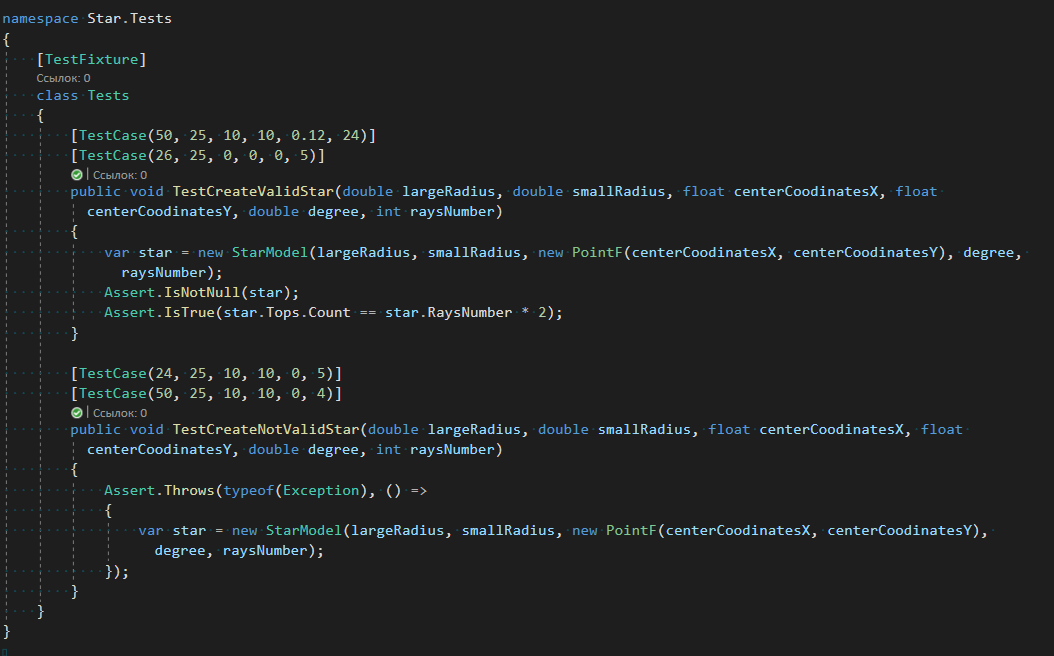
Это методы класса StarValidator. Этот класс нужен для того, чтоб проверять входящие данные на их правильность. Если что-то не так – выбрасываем исключение наверх. Экземпляр этого класса содержится внизу, рядом с другим полем.

Также тут переопределен метод ToString(), чтоб извне мы могли получать структурированную информацию о звезде.

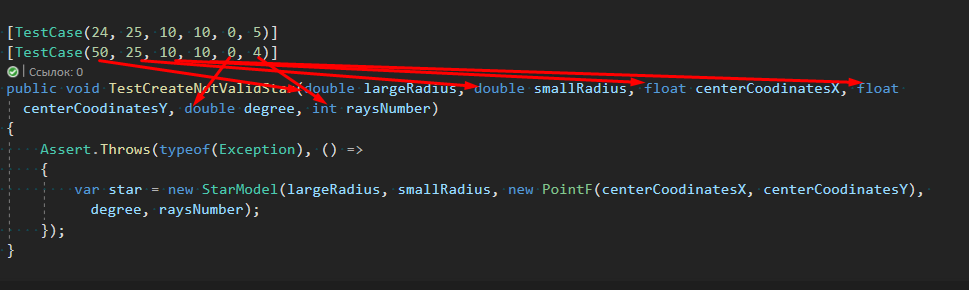
Ниже показан класс-валидатор данных для звезды. Исключения создаются благодаря конструкции throw, она пробрасывает их наверх по стеку вызовов, добавляя нужную информацию в них.



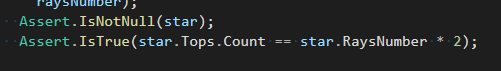
С моделями мы закончили. Теперь перейдем к тестам (Star.Tests). Для тестирования я создал класс Tests и в нем два метода (создание звезды с правильными входными данными и с неправильными). Правильный должен создать звезду и не поймать ни одного исключения, а неправильный наоборот. Вот так выглядит этот класс:



Сам класс помечен атрибутом TestFixture. В этой сборке средство, которое запускает тесты бегло просматривает все классы и оставляет себе только те, которые помечены этим атрибутом. Этот атрибут говорит о том, что в этом классе содержатся методы для тестирования. Каждый метод помечается другим атрибутом – Test, если на вход он не принимает никакие данные, или TestCase(params object[] value) для методов, которые принимают в себя какие-то данные. У меня оба метода принимают данные. Можно пометить несколько раз этим атрибутом, тогда метод протестируется несколько раз. Каждый входной параметр будет получать данные из конструктора этого атрибута. Выглядит это вот так:



Assert. Класс Assert предназначен для сравнения каких-то условий в коде. Если какое-то условие не пройдет, тест будет считаться проваленным. В первом методе у меня есть два метода от класса Assert:

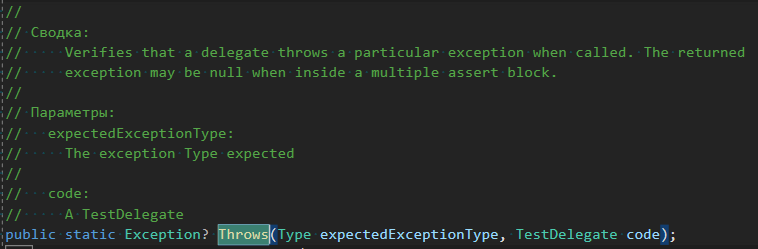


Первый говорит о том, что для прохождения теста звезда не должна быть Null, а второй требует, чтоб количество вершин звезды было равно удвоенному количеству лучей. Если условия проходят – тест успешно выполнен.

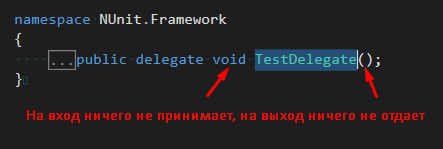
Во втором методе используется Assert.Throws(…)

Чтоб пройти это место, код внутри метода обязан вызвать исключение определенного типа. В нашем случае исключение может вызвать только валидатор звезды, и все исключения, которые он вызывает, базовые – Exception.

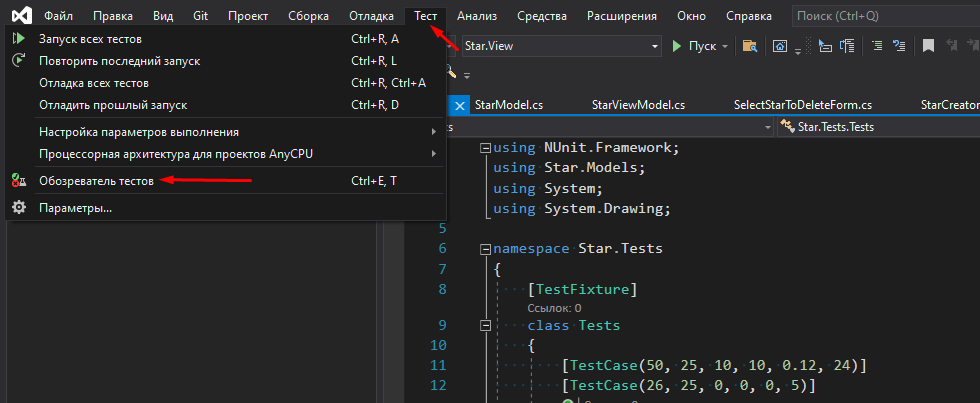
Поэтому первым параметром в Throws мы передаем тип исключения через конструкцию typeof(Exception), а вторым параметром передаем лямбду. Лямбда, она же анонимный метод, может быть заменена обычным методом, схожим по сигнатуре с тем, что требуется (сигнатура метода – соблюдение правильного порядка и типа входящих параметров метода и типа выходящего параметра метода). Почему здесь нельзя написать через метод? Потому что, если взглянуть на сигнатуру метода Throws,



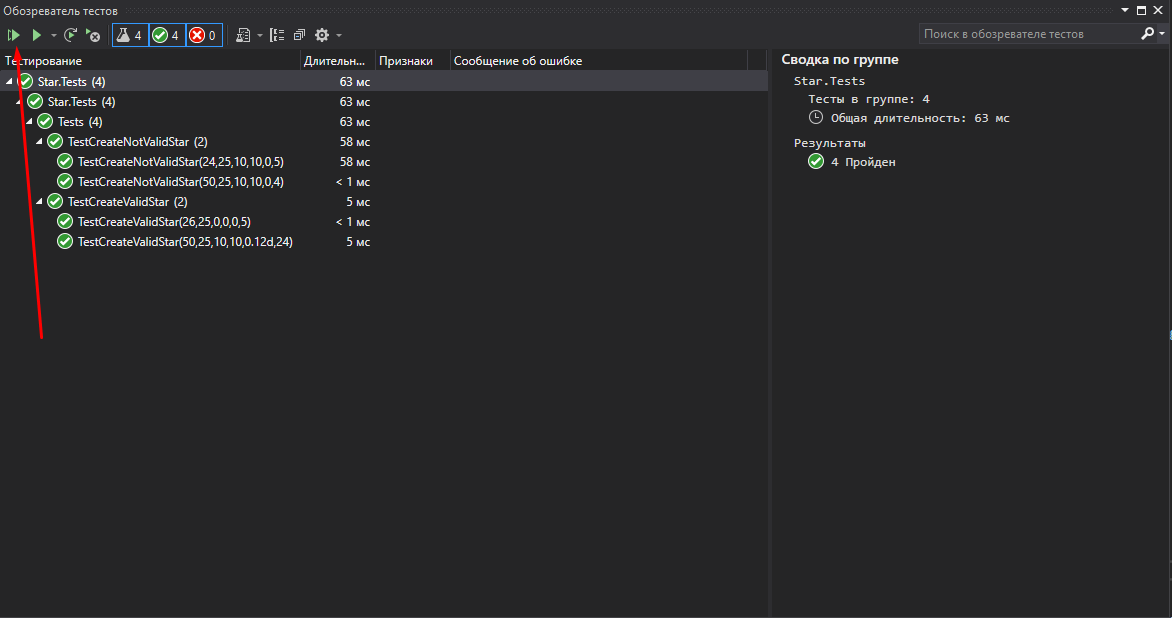
то можно увидеть, что он требует в себя метод с сигнатурой делегата TestDelegate, а его сигнатура – просто пустой метод без входящих параметров:



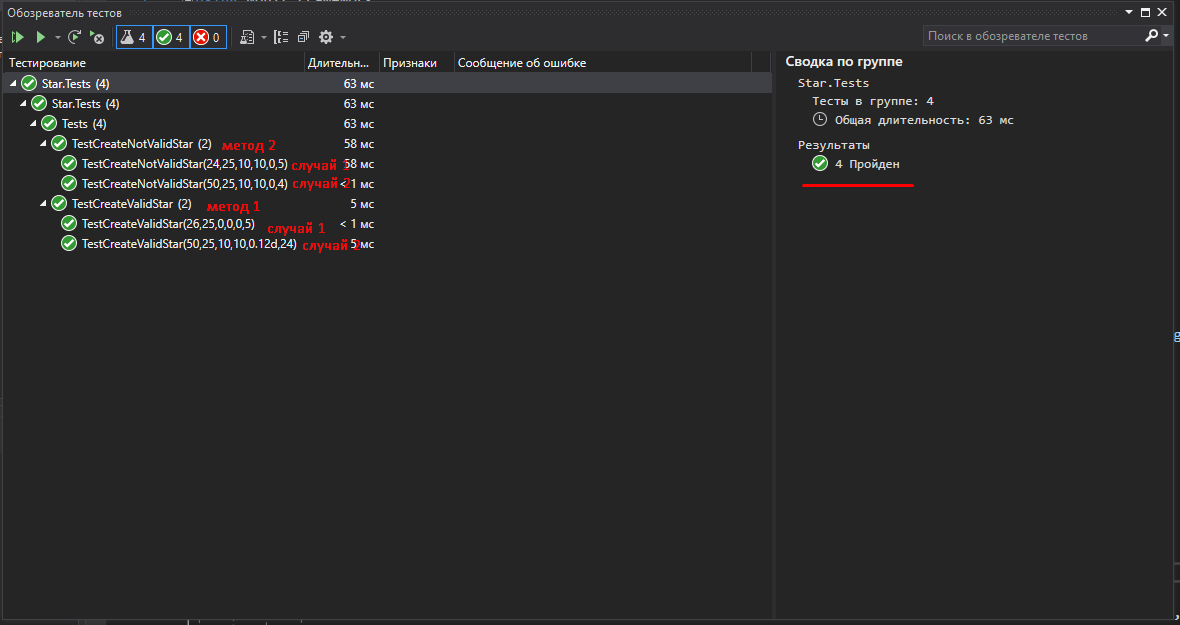
И куда бы тогда пошли наши данные? Никуда. У нас бы не получилось написать через внешний метод, поэтому тут лямбда-выражение.

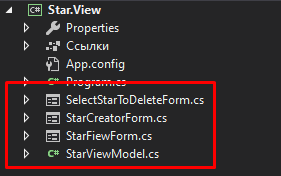
Самое главное – как запустить тесты? 

Раскрываем все ветви, тыкаем сюда:

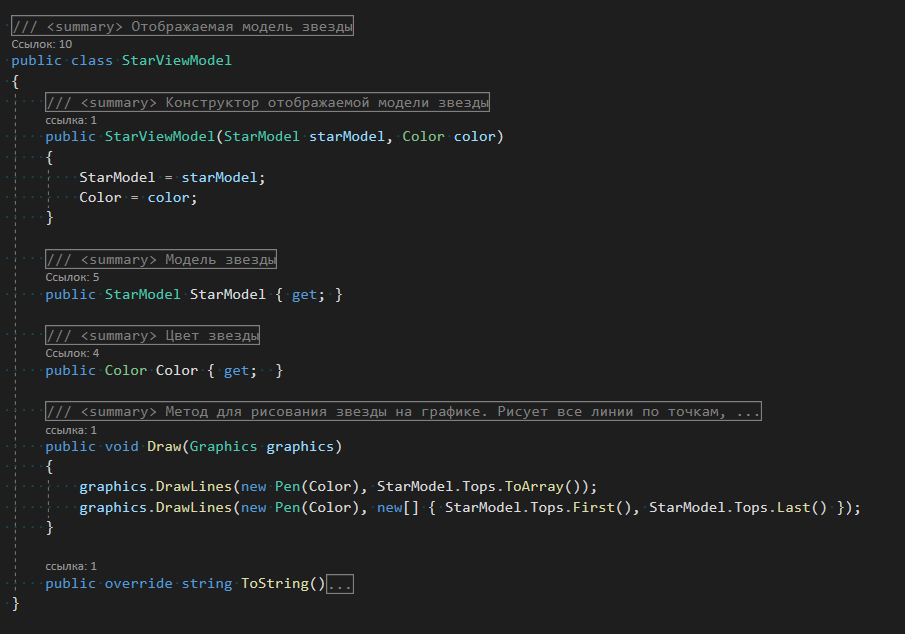


Ждем результатов и видим, что наши методы успешно протестированы для каждого случая:

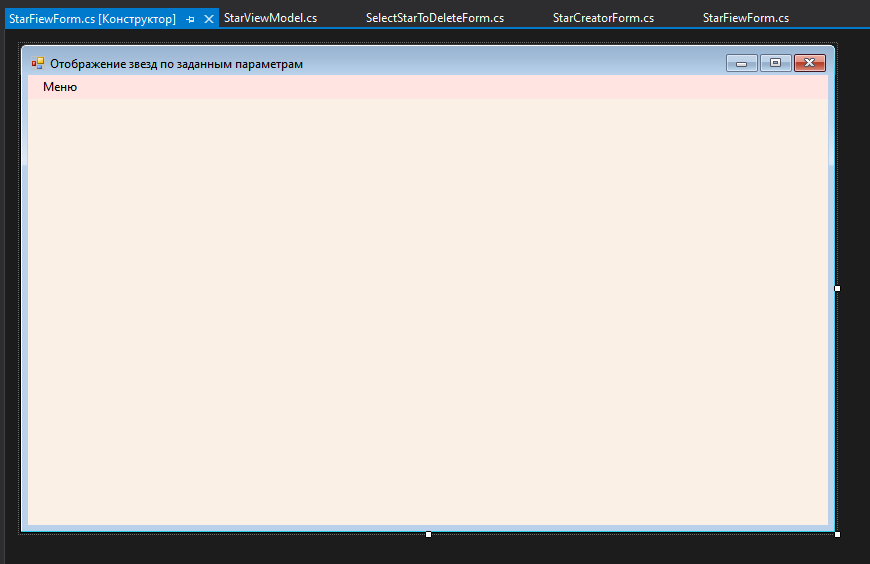
С тестами закончили, теперь переходим в вид (Star.View):



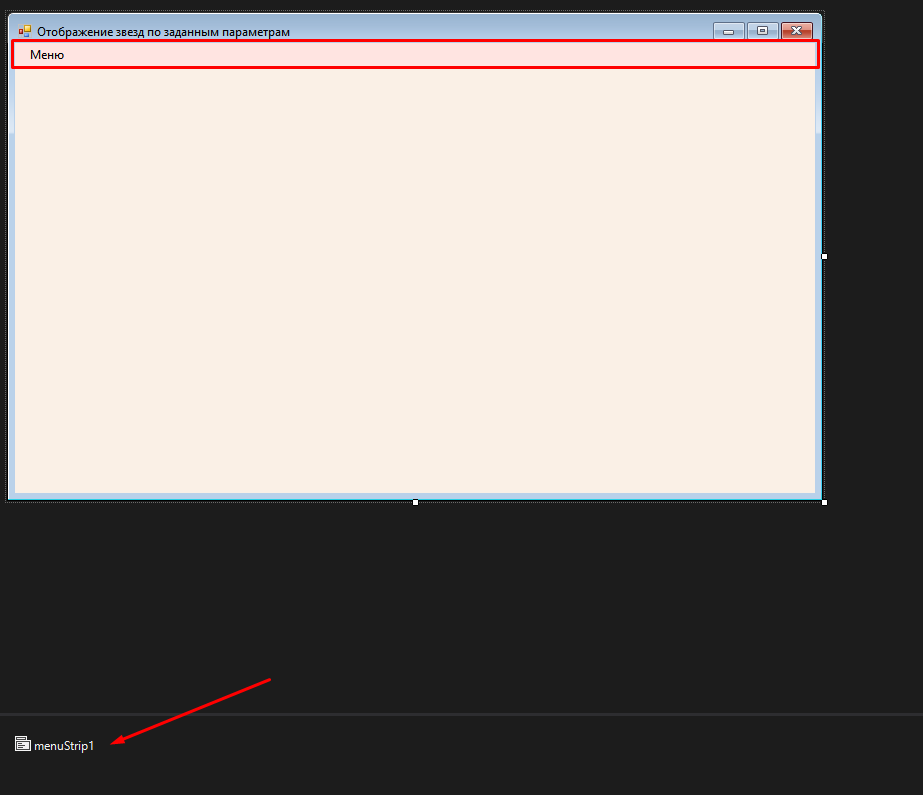
Здесь есть три класса-формы и один класс-обертка над нашей моделью. Формы нужны для отображения пользовательских данных (об этом далее), а обертка StarViewModel нужна для совмещения логики модели и логики ее отрисовки. Посмотрим внутрь:



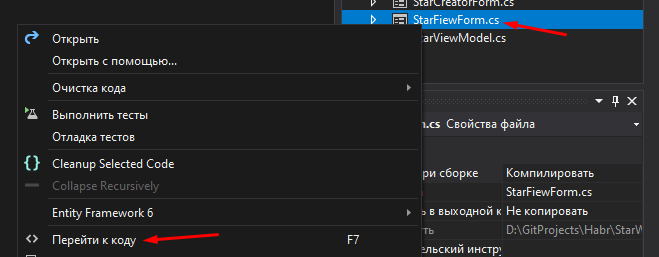
Очень простой класс, как по мне. Хранит внутри себя модель звезды и цвет, которым она отрисована. Умеет рисовать звезду с помощью графики. Почему тут вызывается два раза Draw? Потому что первый раз он рисует линии между всеми вершинами, и не соединяет последние две точки. Второй метод создает мини-массив из этих двух точек (первой и последней) и соединяет их единой линией. Получается замкнутая звезда. Перейдем в главную форму (StarViewForm):



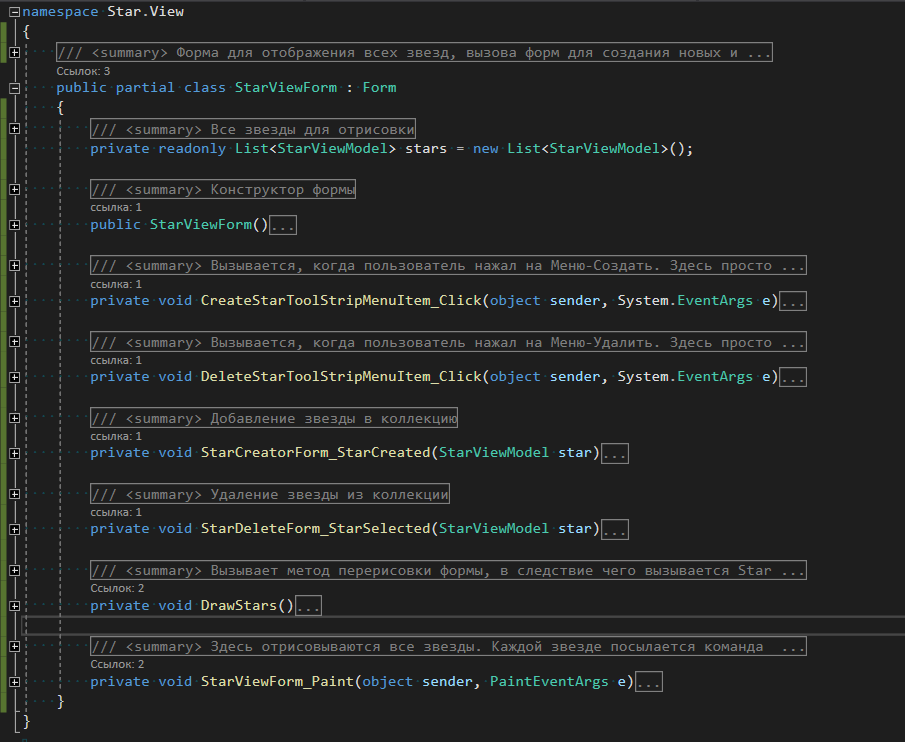
Вот так она выглядит в конструкторе. На форме лежит один элемент – menuStrip.



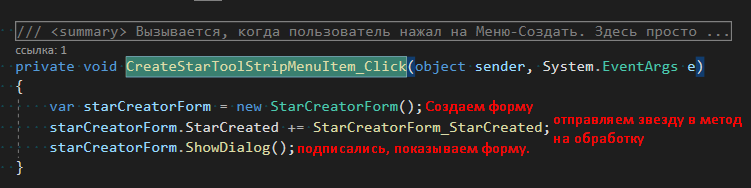
Он создает меню в топе формы. В нем два действия – создать и удалить. Каждое предназначено для, соответственно, создания новой звезды, или удаления уже существующей. Чтоб перейти к коду формы нужно нажать сюда:

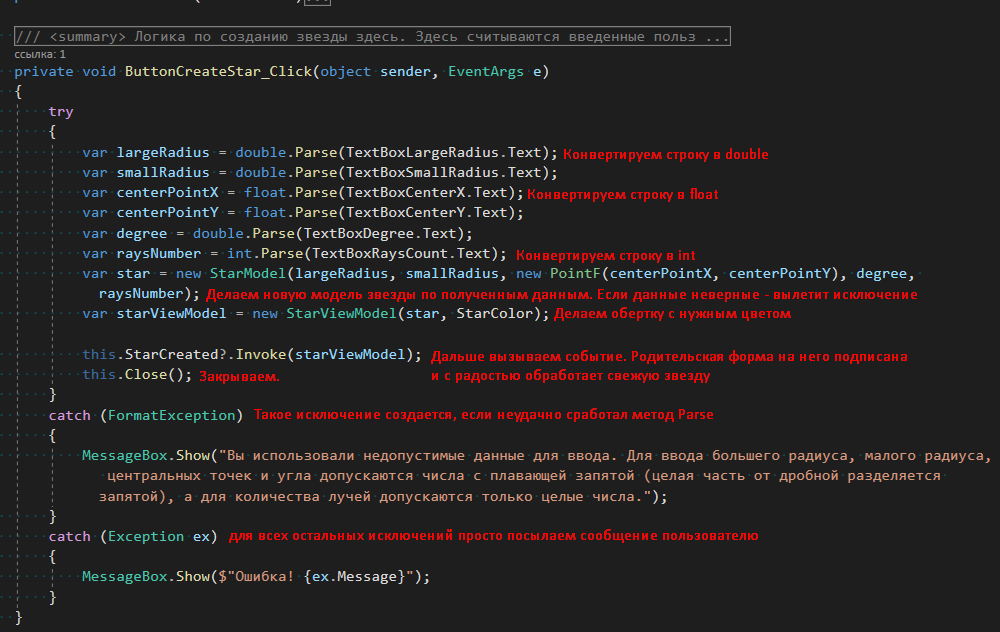


А вот и весь класс:



Внутри есть список всех звезд. Он помечен модификатором readonly, чтоб после объявления его нельзя было изменить. Метод CreateStarToolStripMenuItem\_Click вызывает новую форму, в которой пользователь заполняет данные и которая возвращает новую звезду через подписку на событие:

Что же внутри этой формы? Почти ничего интересного. Только обработка исключений требует внимания:



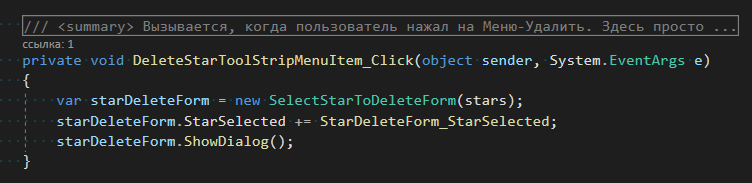
Конструкция try-catch. В блоке try мы кладем код, который может вызвать исключения, и если они вылетят, то автоматически отправятся на обработку в блок catch.

Первый блок catch ловит только исключения типа FormatException. Они вызовутся, если преобразование из строки в другие типы пошло неудачно (пользователь поставил точку вместо запятой, или вообще ничего не ввел в поле).

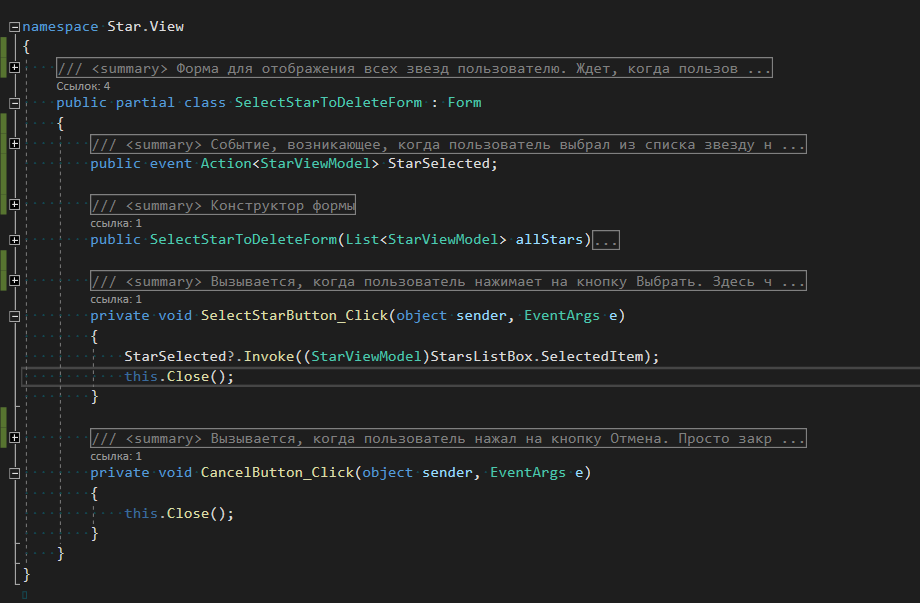
Второй блок catch ловит все остальные исключения. MessageBox.Show() показывает маленькую формочку пользователю. Никакой нагрузки, кроме оповещения о чем-то пользователя она не несет.

Конвертация из строки в другой тип проходит в методе Parse. Он как раз вызывает исключение типа FormatException.

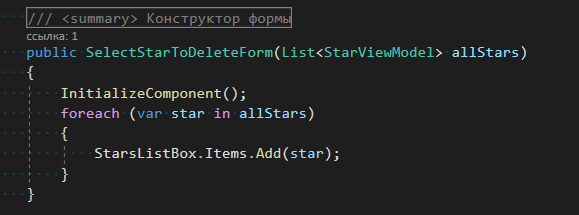
Вернемся в StarViewForm. Метод по удалению звезды:

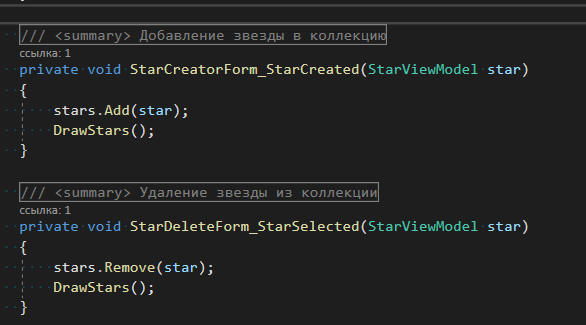


Создаем новую форму для выбора звезды на удаление из списка звезд. Подписываемся на событие этой формы, которое возникает, когда пользователь выбрал звезду на удаление. Что происходит в этой форме:

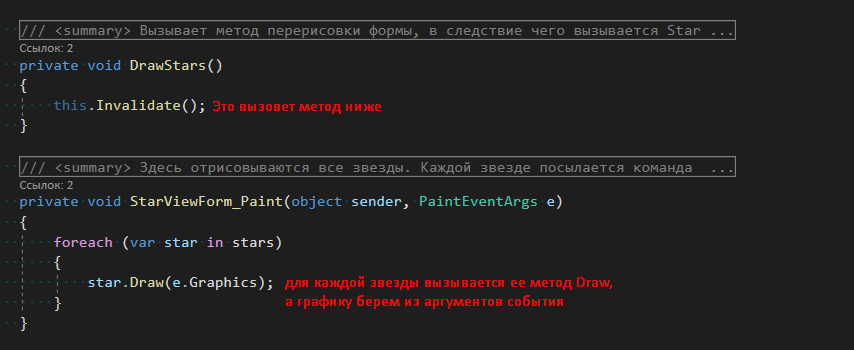


На нажатие кнопки принять берется текущий выбранный элемент, вызывается событие и выбранная звезда (текущий элемент) отправляется наверх. В конструкторе просто заполняется список, из которого выбирать:



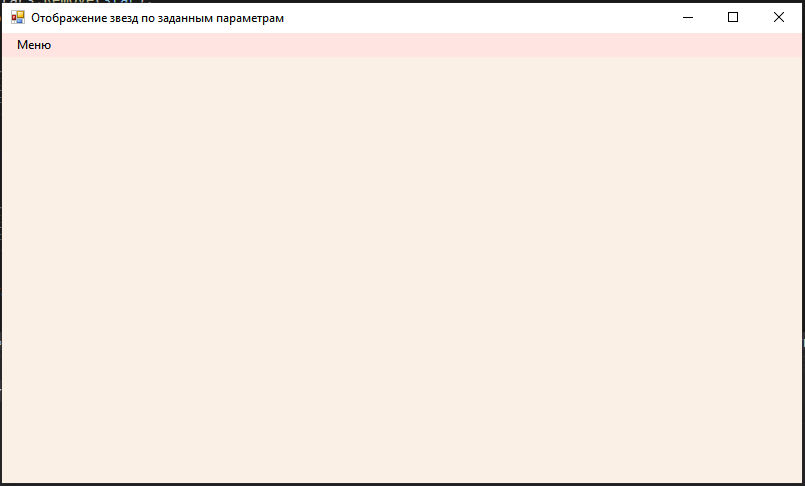
Возвращаемся к главной форме. Обработчики событий по добавлению и удалению звезды просто добавляют или удаляют ее из списка и вызывают перерисовку формы:

Перерисовка формы:

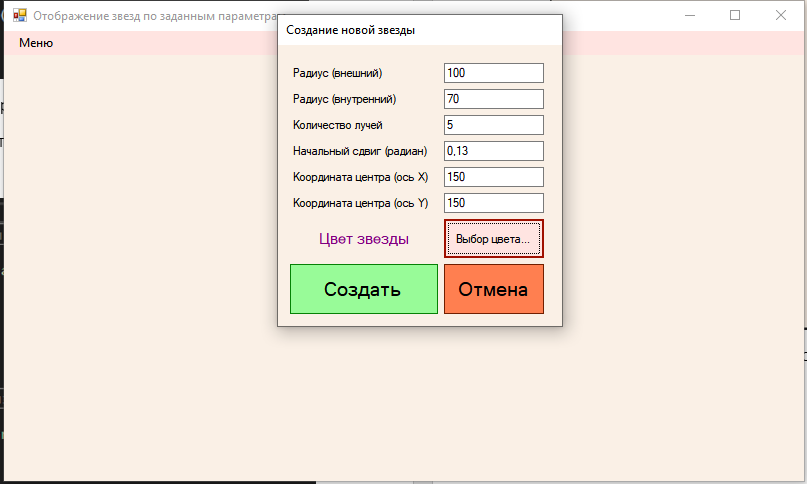


Переходим к результатам работы.

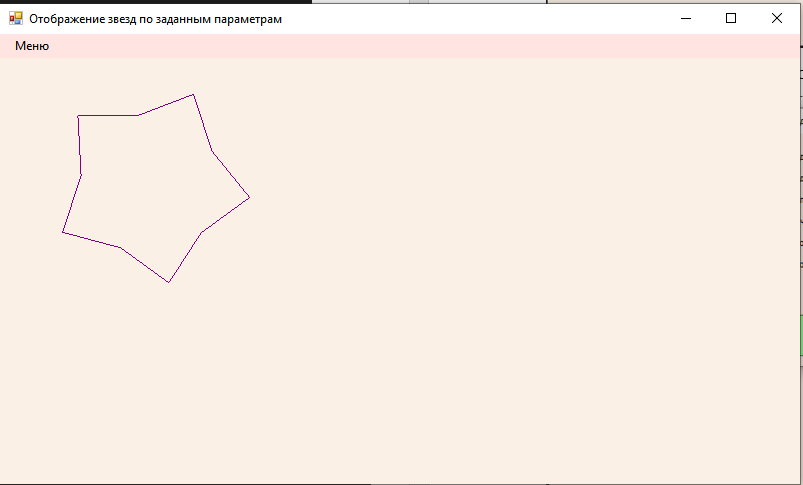
Запуск:



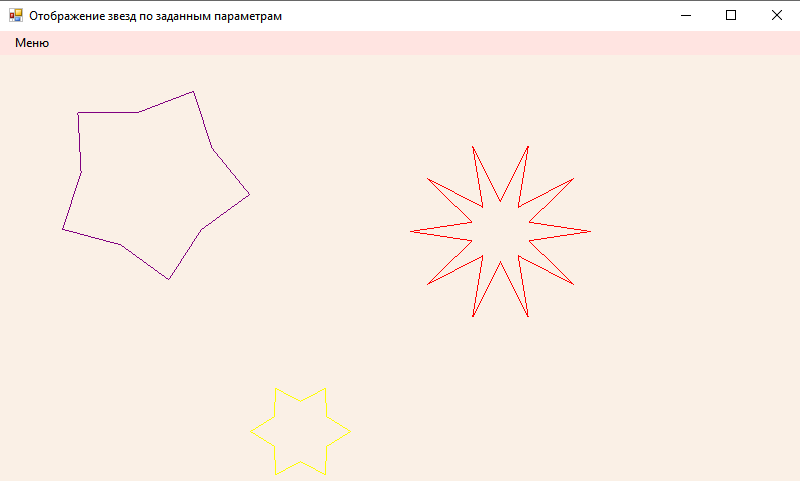
Просто пустая форма. Нужно добавить звезду:



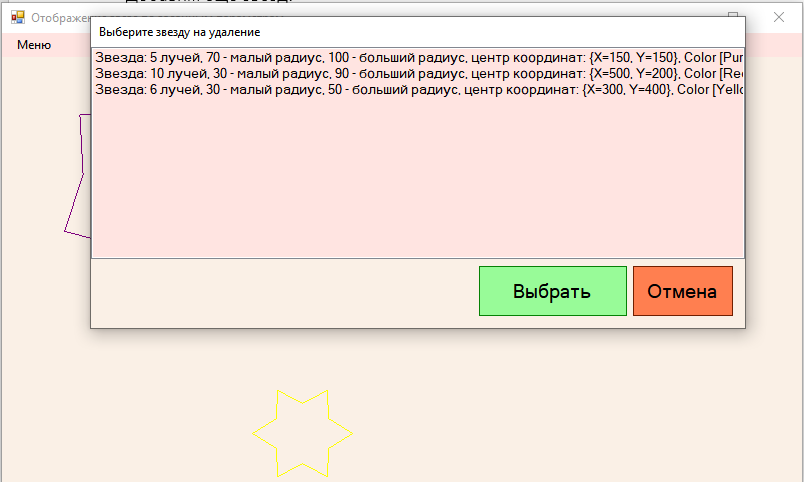
Выбираем все нужные параметры и нажимаем Создать:



Добавим еще звезд:



Удалим какую-нибудь. Выбираем здесь любую и нажимаем Выбрать:



Все успешно удалилось:

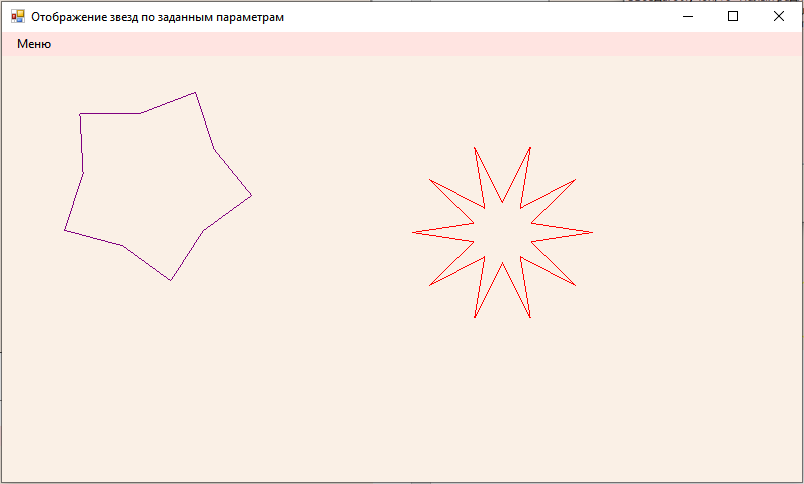


Диаграмма классов:

