Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное

учреждение высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра компьютерных систем в управлении и проектировании (КСУП)

**АНАЛИЗ ПРОГРАММНОГО КОДА НА СООТВЕТСТВИЕ ПРИНЦИПАМ SOLID**

Отчет по лабораторной работе номер 1

по дисциплине «Технология разработки программного обеспечения»

Студент гр. 588-М1

\_\_\_\_\_\_\_\_\_\_ А. А. Усольцева

\_\_\_\_\_\_\_\_\_\_

Руководитель

К.т.н., доцент каф. КСУП

\_\_\_\_\_\_\_\_\_\_ А. А. Калентьев

\_\_\_\_\_\_\_\_\_\_

**Оглавление**

[1 Введение 3](#_Toc6090820)

[2 Анализ программного кода 4](#_Toc6090821)

[2.1 Принцип единственной обязанности 4](#_Toc6090822)

[2.2 Принцип открытости/закрытости 5](#_Toc6090823)

[2.3 Принцип подстановки Лисков 8](#_Toc6090824)

[2.4 Принцип разделения интерфейса 9](#_Toc6090825)

[2.5 Принцип инверсии зависимостей 10](#_Toc6090826)

[3 Заключение 12](#_Toc6090827)

# **1 Введение**

Шаблоном проектирования или паттерном в разработке программного обеспечения называется повторяемая архитектурная конструкция, которая представляет собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

Одними из популярных шаблонов проектирования являются принципы SOLID (сокр. от англ. single responsibility, open-closed, Liskov substitution, interface segregation и dependency inversion) – первые пять принципов, названных Робертом Мартином в начале 2000-х, которые означали пять основных принципов объектно-ориентированного программирования и проектирования.

Принципы SOLID – это руководства, которые могут применяться во время работы над программным обеспечением для удаления «проблемного кода», предписывая программисту выполнять рефакторинг исходного кода, пока тот не станет разборчиво написанным и расширяемым. Это часть общей стратегии гибкой и адаптивной разработки. Использование в совокупности данных принципов позволяет повысить вероятность того, что программист создаст систему, которую будет легко поддерживать и расширять в течение долгого времени.

# **2 Анализ программного кода**

## **2.1 Принцип единственной обязанности**

Принцип единственной обязанности (Single-Responsibility Principle – SRP) обозначает, что каждый объект должен иметь одну обязанность, и эта обязанность должна быть полностью инкапсулирована в класс.

Относительно положительным примером класса программы, который соблюдает принцип единственной обязанности, можно назвать класс настроек точек PointsSettings, диаграмма которого представлена на рисунке 2.1. Данный класс представляет собой коллекцию точек с задаваемым лимитом. Фигуры программы композируют класс PointSettings.

Основной обязанностью данного класса является хранение точек. Дополнительными обязанностями являются добавление, удаление, изменение, информация о возможности добавления, расчет входят ли точки коллекции в прямоугольник.

Разбиение данного класса на более мелкие будет излишним, поскольку это приведет к разнесению обязанностей настроек точек фигуры.

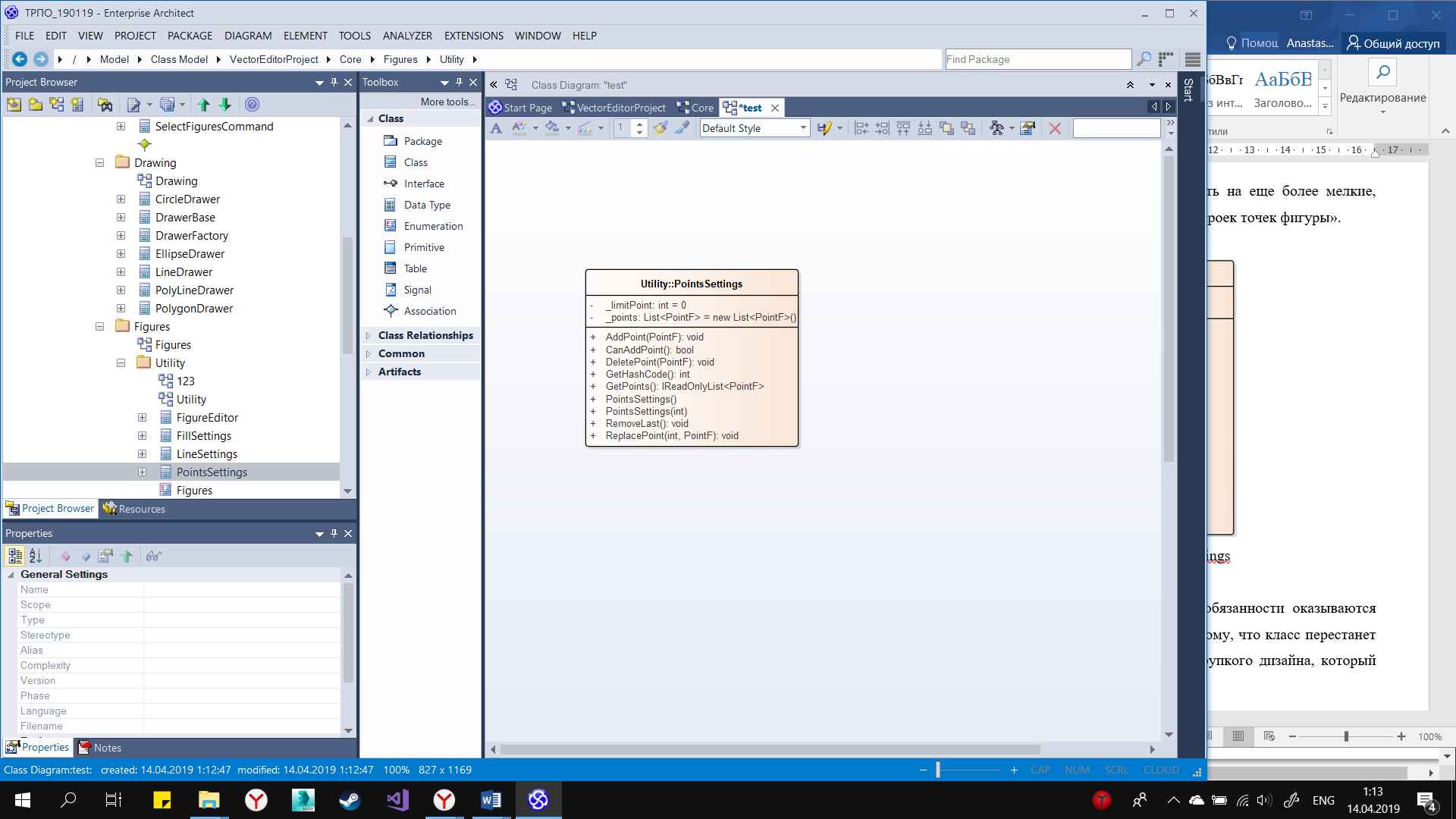


Рисунок 2.1 – Диаграмма класса PointsSettings

Если класс отвечает за несколько действий, то его обязанности оказываются связанными. Изменение одной обязанности может привести к тому, что класс перестанет справляться с другими. Такого рода связанность является причиной хрупкого дизайна, который неожиданным образом разрушается при изменении.

На рисунке 2.2 представлена диаграмма класса ControlUnit, который является негативным примером применения принципа единственной обязанности.

Изначально данный класс задумывался как класс контроллер к стэку команд для Undo/Redo, но в итоге оказался перегружен обязанности, и на данный момент полностью или частично отвечает за следующее:

– Undo/Redo;

– обновление .NET контролов после исполнения команд;

– удаление, копирование, вырезание фигур;

– сериализация/десериализация.

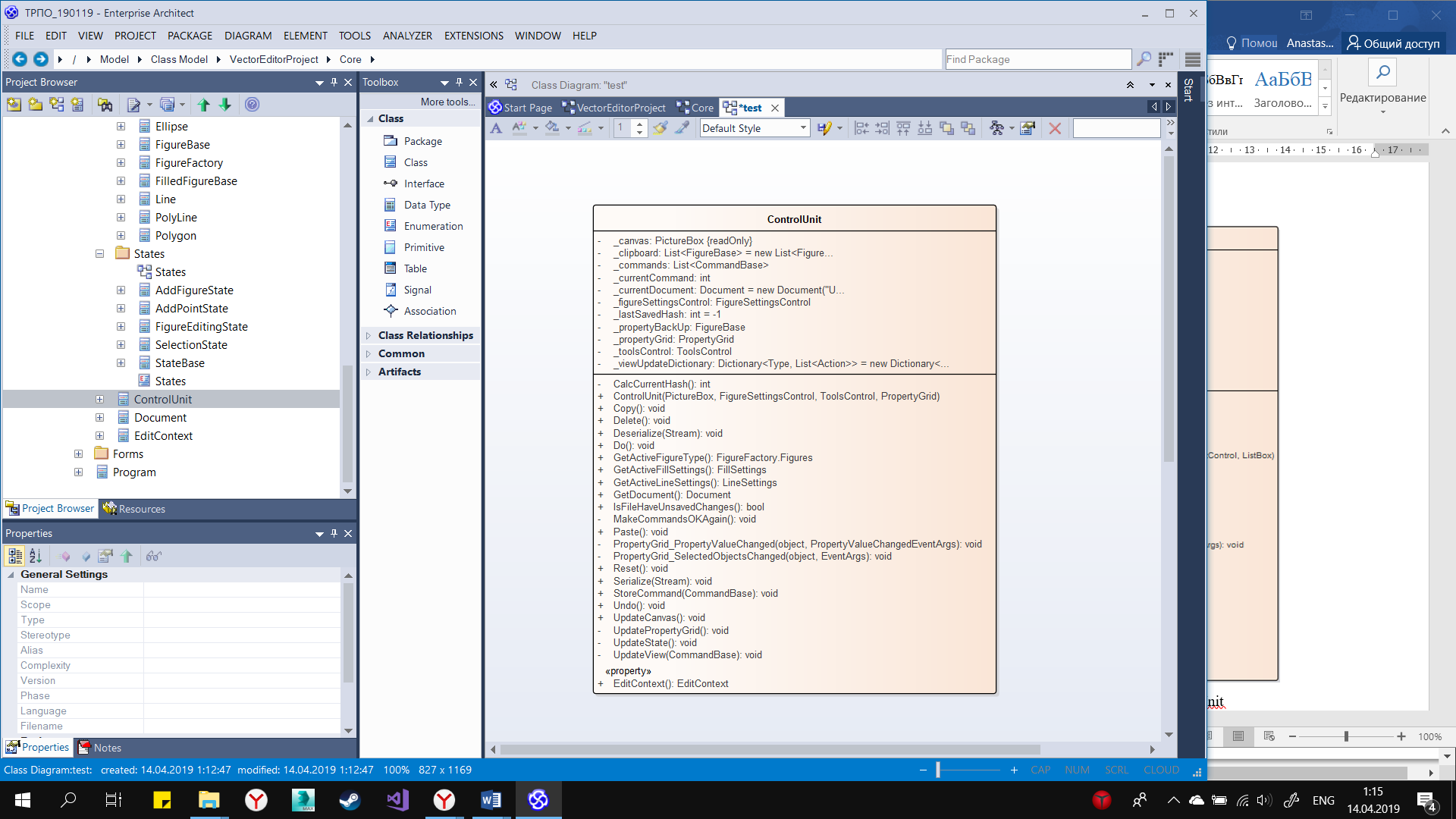


Рисунок 2.2 – Диаграмма класса ControlUnit

## **2.2 Принцип открытости/закрытости**

Принцип открытости/закрытости (Open/Closed Principle – OCP) подразумевает, что программные сущности (классы, модули, функции и т.п.) должны быть открыты для расширения, но закрыты для модификации.

У модулей, согласованных с принципом OCP, есть две основных характеристики.

Во-первых, они открыты для расширения. Это означает, что поведение модуля можно расширить. Когда требования к приложению изменяются, мы добавляем в модуль новое поведение, отвечающее изменившимся требованиям. Иными словами, мы можем изменить состав функций модуля.

Во-вторых, они закрыты для модификации. Расширение поведения модуля не сопряжено с изменениями в исходном или двоичном коде модуля. Двоичное исполняемое представление модуля, будь то компонуемая библиотека, DLL или EXE-файл, остается неизменным.

В C#, как и в любом другом объектно-ориентированном языке программирования, можно создавать абстракции, которые сами по себе фиксированы, но представляют неограниченное множество различных поведений. Абстракции – это абстрактные базовые классы, а поведения представляются производными от них классами. Модуль может манипулировать абстракцией. Такой модуль можно сделать закрытым для модификации, поскольку он зависит от фиксированной абстракции. И тем не менее поведение модуля можно расширять, создавая новые производные от абстракции.

В анализируемом программном коде положительным примером применения принципа OCP является реализация абстракций фигур. На рисунке 2.3 представлена UML-диаграмма классов фигур. Как видно из диаграммы, добавить новую фигуру в программу возможно будет без перекомпиляции основной программы.

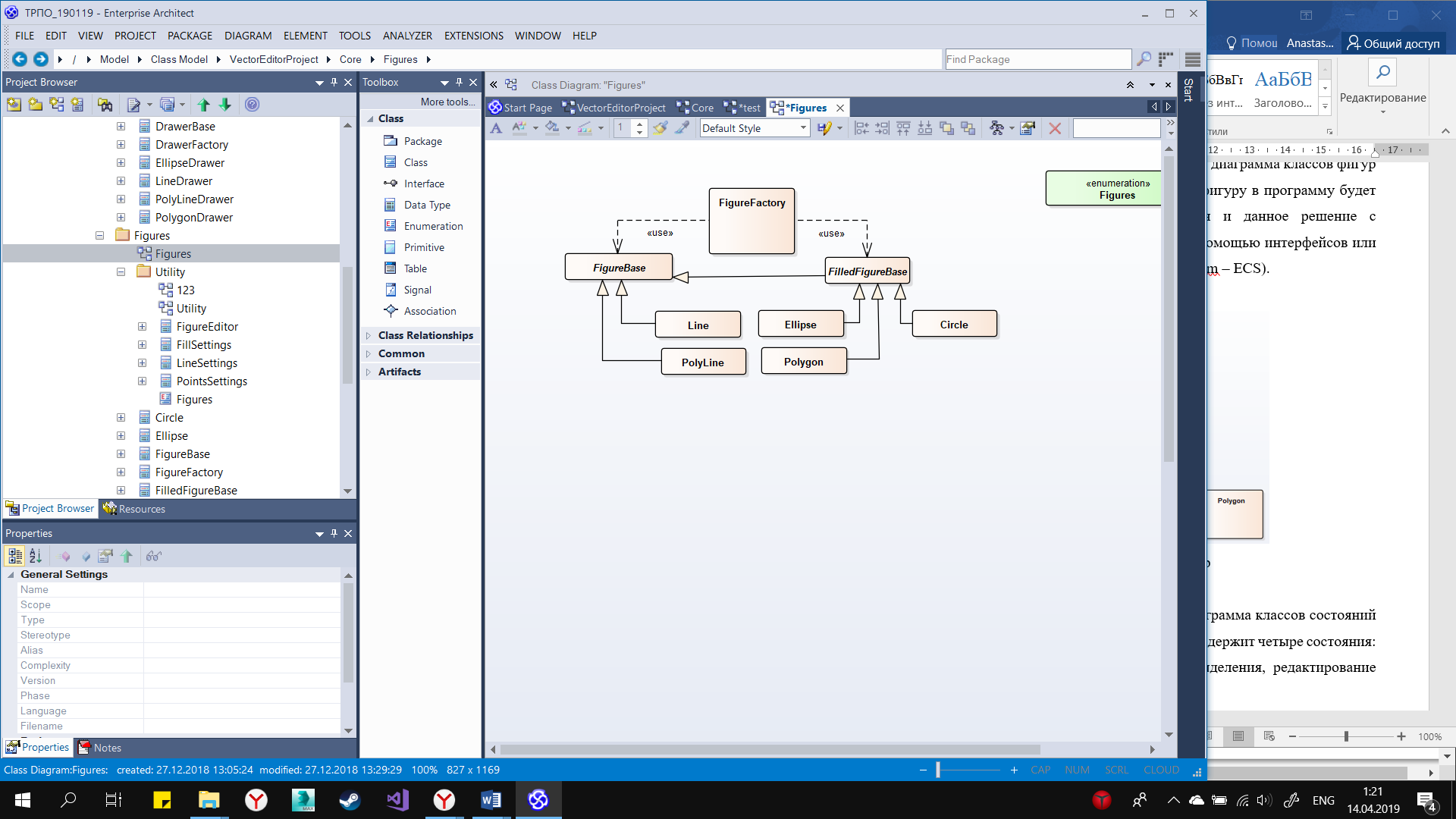


Рисунок 2.3 – Диаграмма классов фигур

Примером неудачного применения принципа открытости/закрытости является система состояний программы, диаграмма классов которых представлена на рисунке 2.4. В текущей реализации программа содержит следующие четыре состояния:

– добавление фигуры;

– добавление точки в фигуру;

– создание выделения;

– редактирование размеров выделенных фигур.

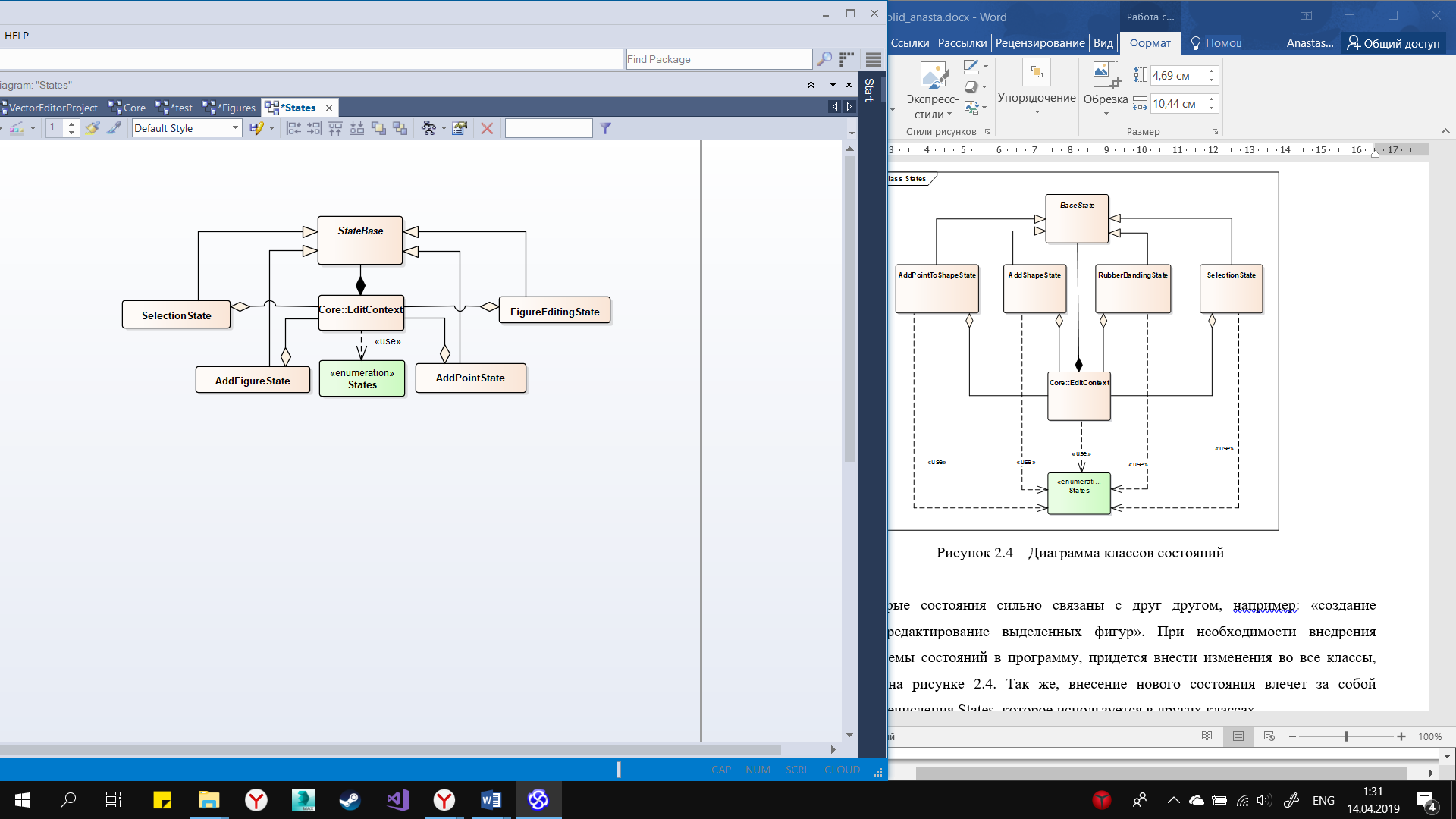


Рисунок 2.4 – Диаграмма классов состояний

Внесение каждого нового состояния повлечет за собой необходимость в дополнении перечисления States, которое используется в других классах

Также, некоторые состояния, как, например, «создание выделения – редактирование выделенных фигур» сильно связаны друг с другом. Из-за этого придется внести изменения во все классы, которые представлены на рисунке 2.4, при необходимости внедрения плагинной системы состояний в программу.

Решением данной ситуации будет использование словаря, в котором ключ – это название состояния, а значение – это ссылка на класс. Данный подход позволит перенести конкретные состояния в отдельные сборки, а StateBase – в SDK часть программы.

**2.3 Принцип подстановки Лисков**

Принцип подстановки Лисков (Liskov Substitution Principle – LSP) подразумевает, что должна быть возможность вместо базового типа подставить любой его подтип.

Принцип Лисков, сформулированный Р. Мартином: функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Иначе говоря, наследующий класс должен дополнять, а не замещать поведение базового класса.

Положительным примером применения принципа подстановки Лисков является реализация базовых классов фигур FigureBase и FilledFigureBase, диаграммы которых представлены на рисунке 2.5. Класс FigureBase композирует в себе классы настроек линии и настроек точек фигуры. Класс FilledFigureBase расширяет возможности FigureBase, реализуя возможность добавления свойства заливки.

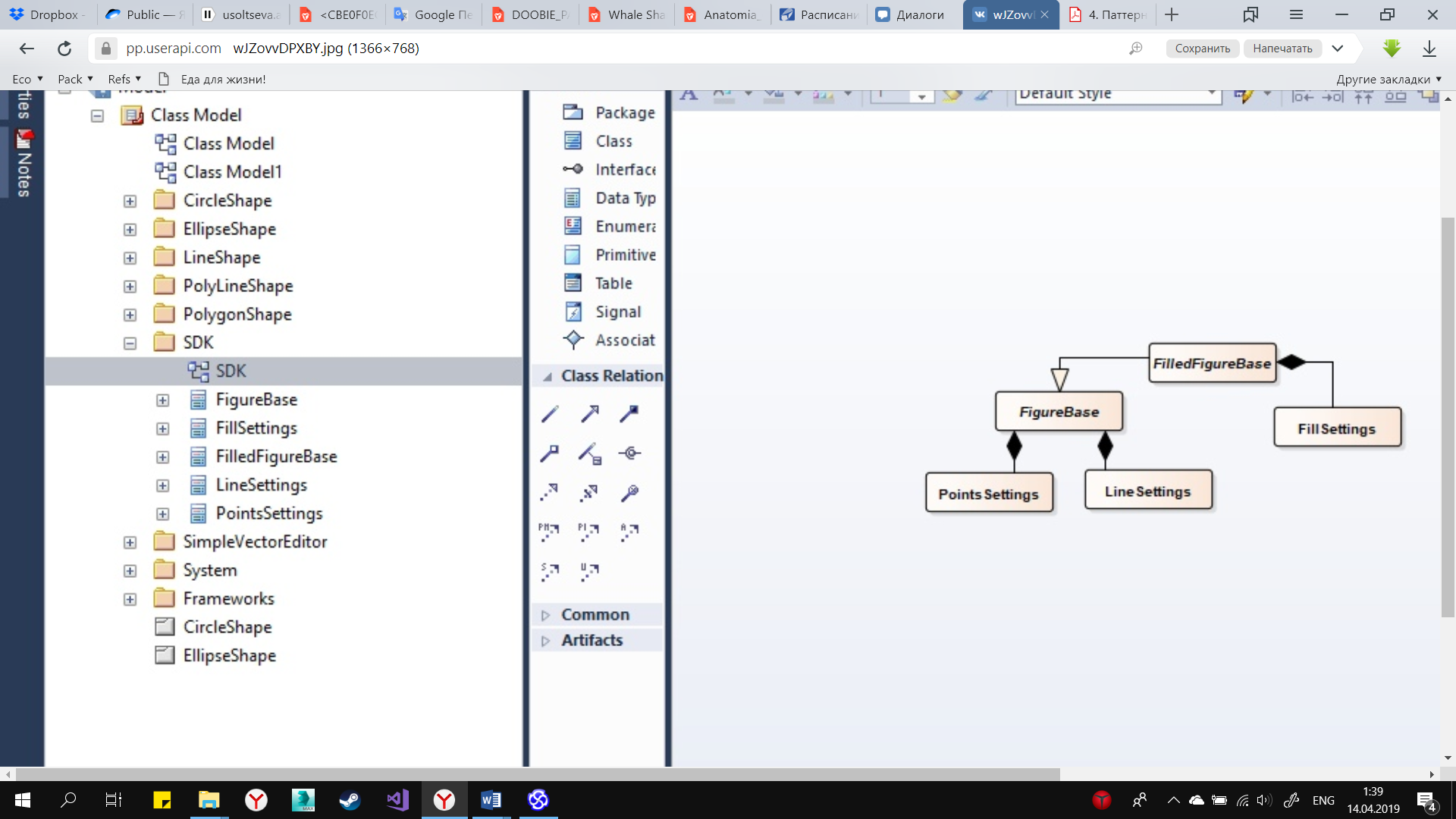
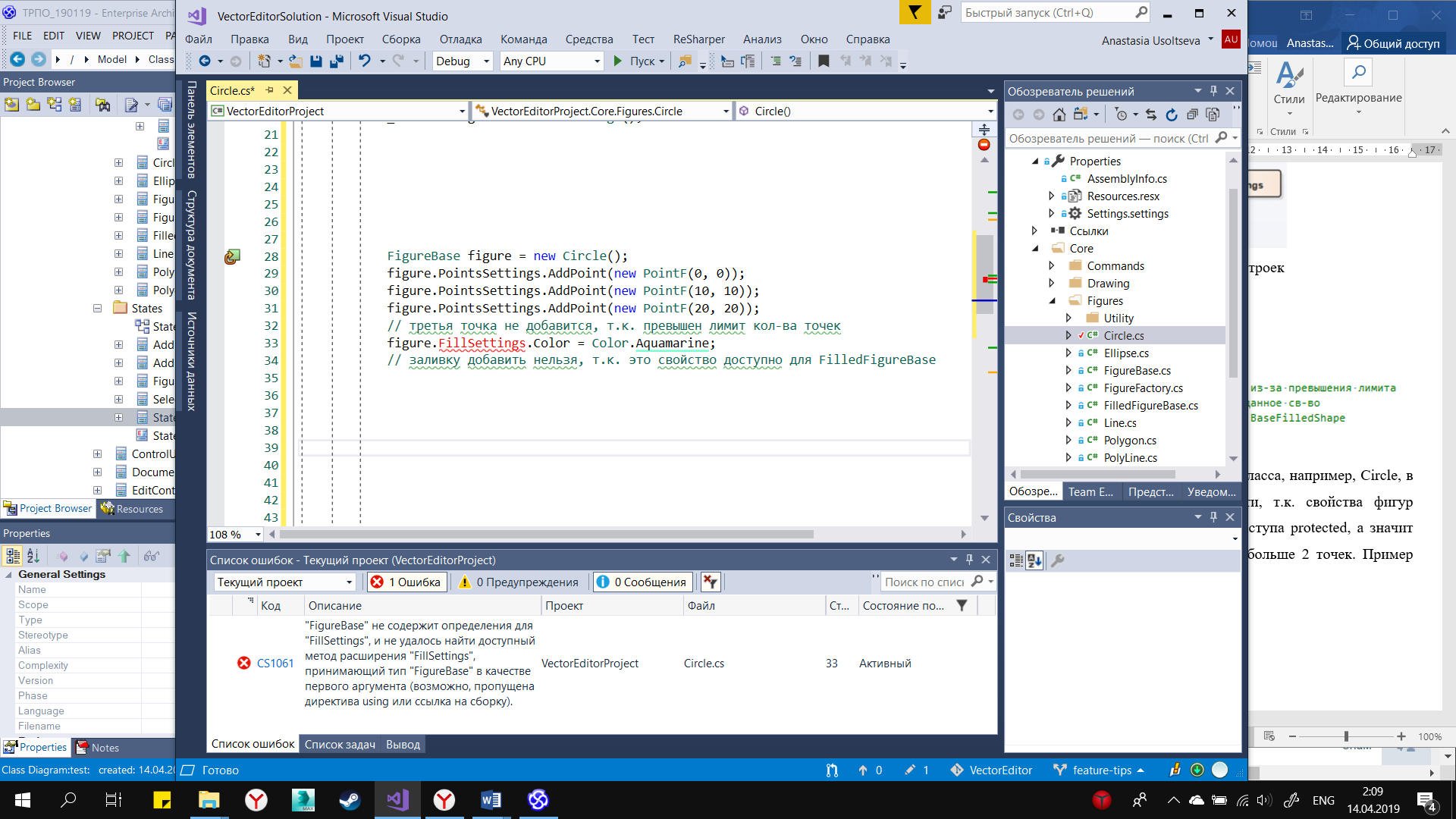
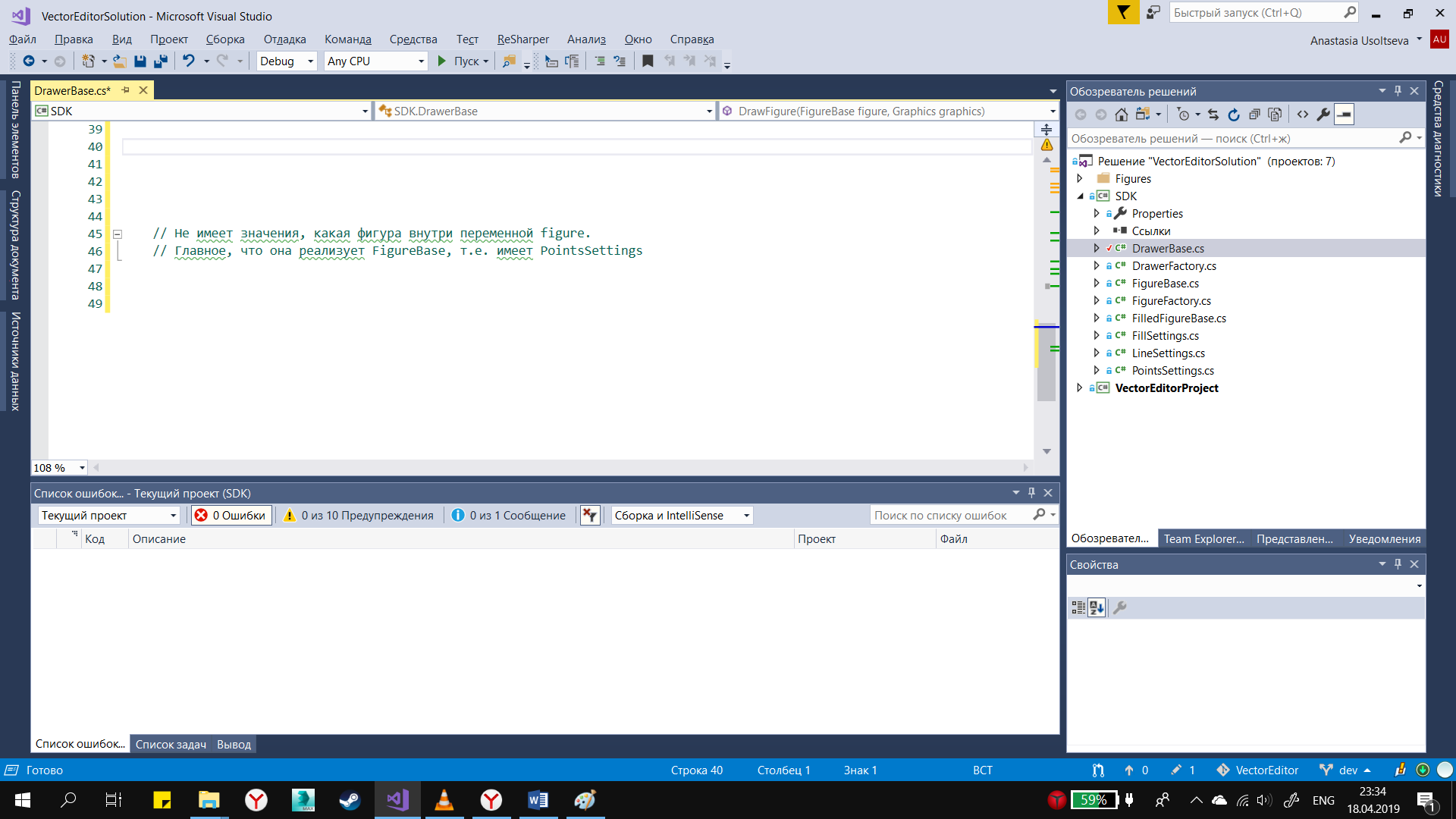


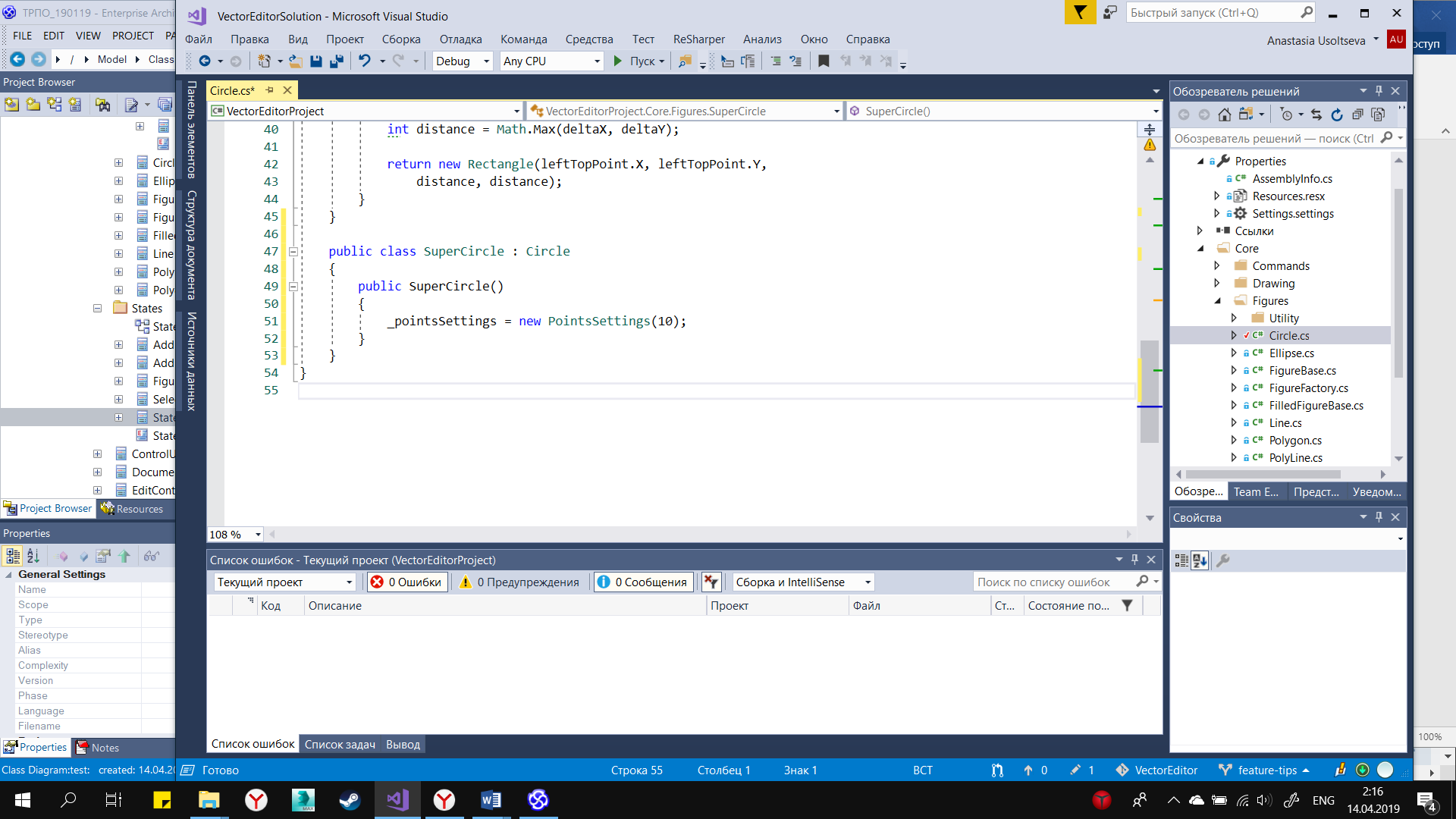
Рисунок 2.5 – Диаграмма классов фигур и настроек

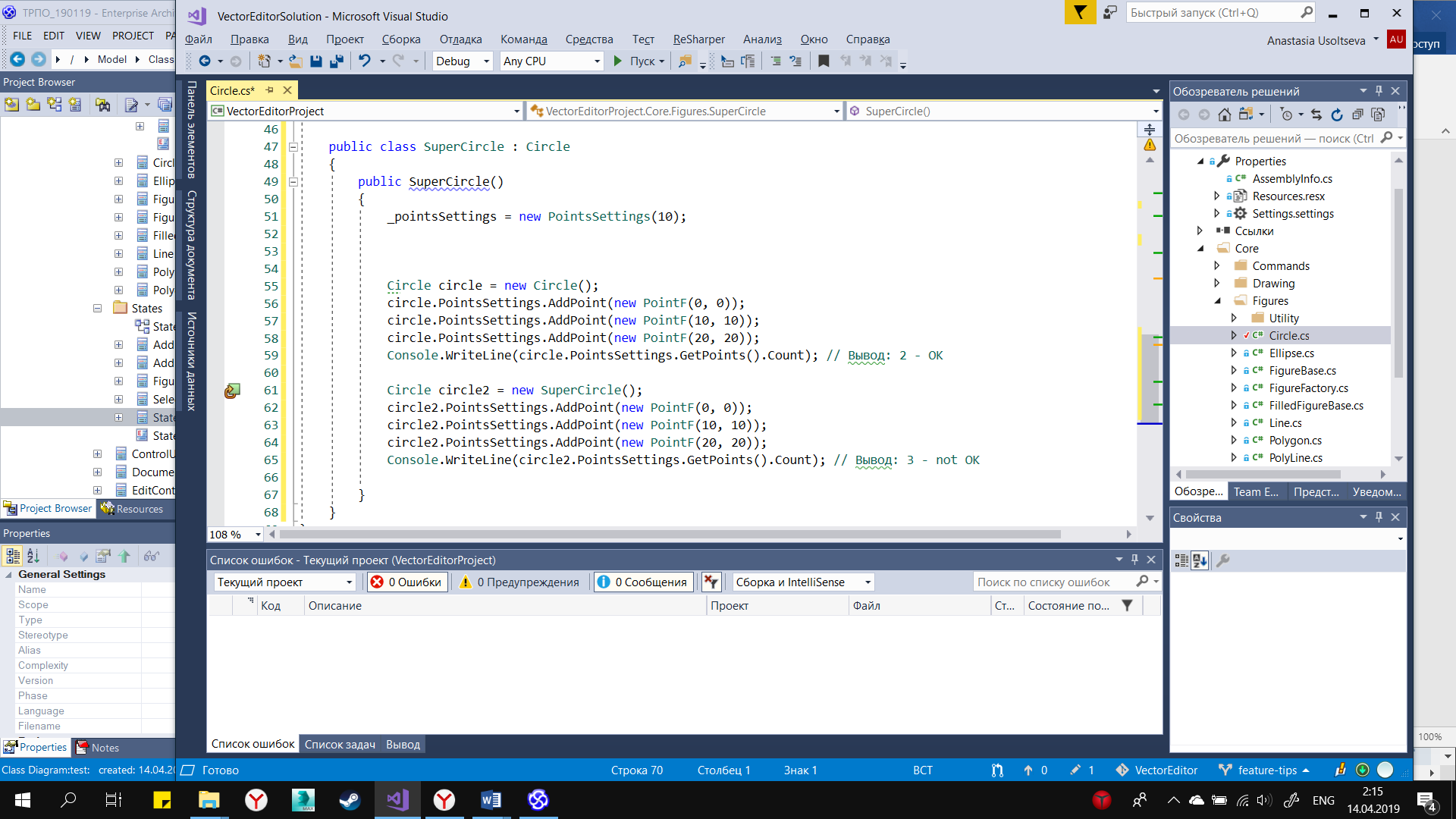
Ниже приведен пример пользовательского кода:





Примером негативного применения LSP является реализация наследования от конкретного класса. Например, Circle в большинстве случаев приведет к нарушению инвариантности, поскольку свойства фигур (LineSettings, PointsSettings, FillSettings) имеют модификатор доступа protected, что означает, что возможно создать наследник круга, который будет способен вместить в себя более двух точек. Далее представлен пример пользовательского кода, демонстрирующий данную проблему:





## **2.4 Принцип разделения интерфейса**

Принцип разделения интерфейса (Interface segregation principle – ISP). Много специализированных интерфейсов лучше, чем один универсальный.

Иначе этот принцип можно сформулировать так: клиенты не должны зависеть от методов, которые они не используют.

Положительным примером применения ISP является реализация абстракций фигур, а именно базовых классов FigureBase и FilledFigureBase. Однако текущая реализация базовых фигур и их свойств нельзя назвать гибкой, поскольку при добавлении новых типов настроек к фигурам, она перестанет быть такой же простой и понятной.

Аналогично можно было реализовать данное решение через использование интерфейсов, которые бы отражали наличие той или иной настройки у фигуры. В таком случае, интерфейс ILineSettings реализуется всеми фигурами, в то время как интерфейс IFillSettings реализуется только теми фигурами, которым нужна заливка: Polygon, Ellipse и тому подобные. Данная система со множеством узко специализированных интерфейсов была бы более гибкой.

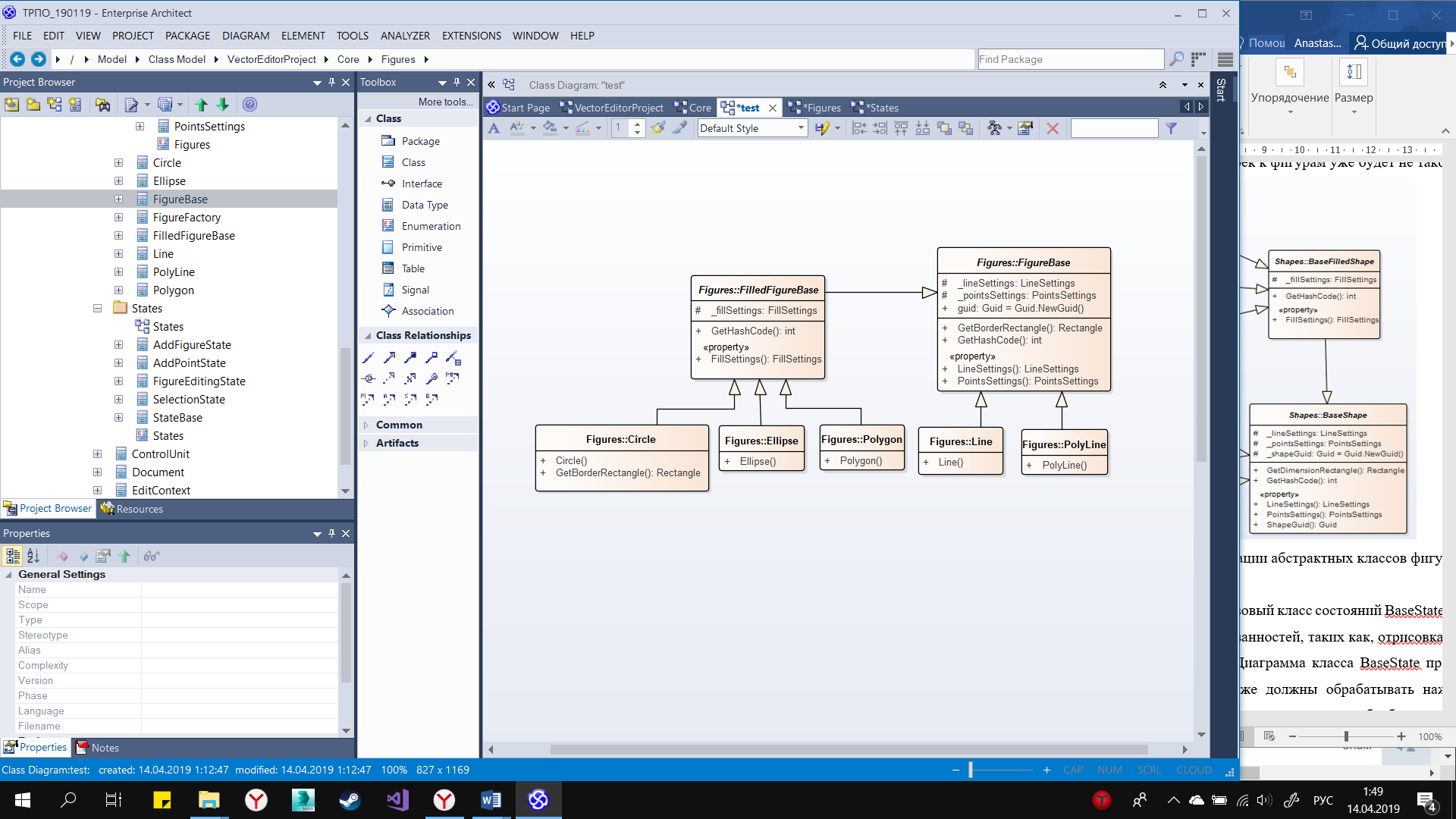


Рисунок 2.6 – Диаграмма реализации абстрактных классов фигур

Негативным примером применения ISP является базовый класс состояний StateBase, диаграмма которого представлена на рисунке 2.7. Данный класс вместил в себя различные обязанности, такие, например, как обработка пользовательского ввода мыши и отрисовка инструмента.

Проблема появляется по причине того, что состояния также должны обрабатывать нажатия клавиш пользователем: одним состояниям нужна только обработка ввода мышью, другим – оба метода сразу. Создание специализированных интерфейсов позволило бы решить данную проблему.

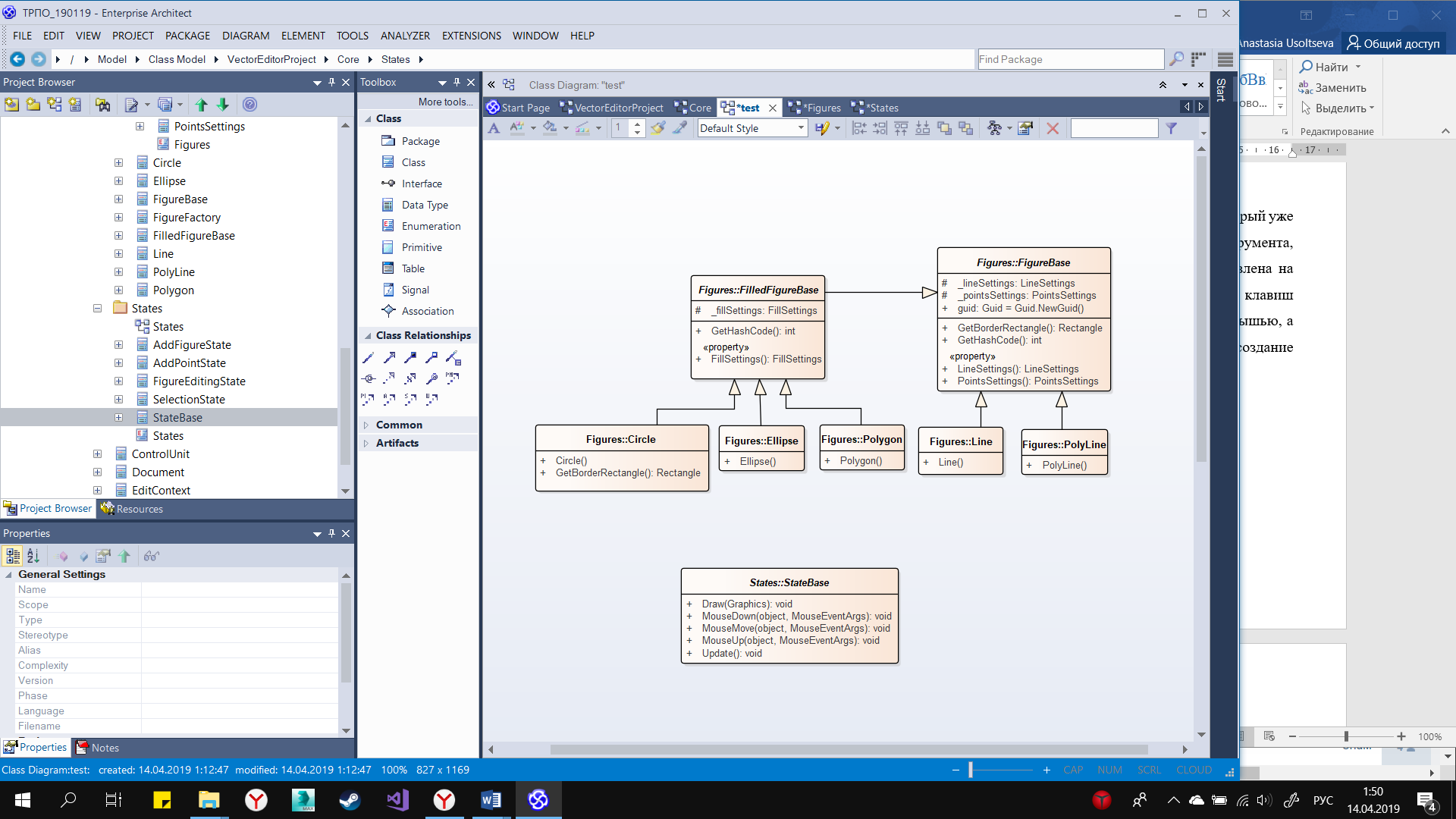
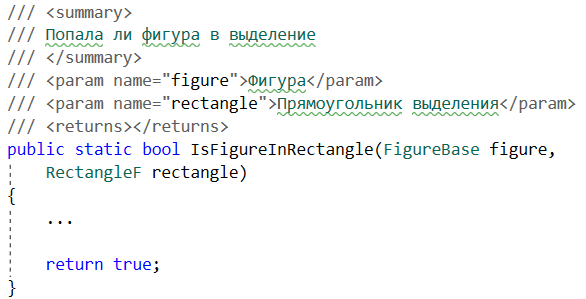


Рисунок 2.7 – Диаграмма класса StateBase

## **2.5 Принцип инверсии зависимостей**

Принцип разделения интерфейса (Dependency inversion principle – DIP). Суть принципа инверсии зависимостей проста: заменить композицию агрегацией. То есть вместо создания зависимостей напрямую, класс должен требовать их у более высокого уровня через аргументы метода или конструктора. При этом зависимость должна передаваться не в виде экземпляров конкретных классов, а в виде интерфейсов или абстрактных классов.

Положительным примером использования принципа разделения интерфейса в методе FigureEditor::IsFigureInRectangle является передача ссылок на конкретную фигуру по нужным абстракциям, в зависимости от контекста. Функция работает правильно вне зависимости от действительного типа объекта, поскольку ей требуются только настройки точек фигуры (PointsSettings), которые определены в классе-родителе (FigureBase).



Негативным примером использования принципа инверсии зависимостей является функция, которая создает копию фигуры – CopyFigure. Данная функция принимает объект по базовому классу FigureBase, но все равно приводит объект к FilledFigureBase и копирует свойства, которые соответствуют базовому классу FilledFigureBase.



# **3 Заключение**

В ходе работы был произведен анализ программного кода на соответствие паттернам SOLID.

В работе рассмотрены как положительные примеры применения шаблонов проектирования SOLID, так и негативные. В качестве примеров приведены образцы рассмотренного программного кода, а также соответствующие им UML-диаграммы.