Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

Высшего профессионального обучения

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра компьютерных систем в управлении и проектировании (КСУП)

Отчет по лабораторной работе №1 по дисциплине

«Технология разработки программного обеспечения»

«Анализ на соответствие SOLID»

Выполнил:

Студент гр. 588-М1

\_\_\_\_\_\_Д.А. Домаскин

. «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2019 г.

Проверил:

Доцент каф. КСУП

\_\_\_\_\_\_\_\_\_ А.А. Калентьев

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2019 г.

Томск 2019

**Оглавление**

**Оглавление**

[**1 Принцип единственной ответственности** 3](#_Toc6255798)

[**2 Принцип открытости/закрытости** 5](#_Toc6255799)

[**3 Принцип подстановки Барбары Лисков** 7](#_Toc6255800)

[**4 Принцип разделения интерфейсов** 8](#_Toc6255801)

[**5 Принцип инверсии зависимостей** 9](#_Toc6255802)

## **1 Принцип единственной ответственности**

**Положительный пример** – *DraftSerealizer*. На листинге 1.1 приведен код класса.

/// <summary>

/// Сериалайзер проекта

/// </summary>

public class DraftSerealizer

{

/// <summary>

/// Сериализовать стек команд

/// </summary>

/// <param name="stream">Поток</param>

/// <param name="stack">Сериализуемый стек</param>

public void Serialize(Stream stream, UndoRedoStack stack)

{

BinaryFormatter binaryFormatter = new BinaryFormatter();

binaryFormatter.Serialize(stream, stack);

}

/// <summary>

/// Десериализовать стек команд

/// </summary>

/// <param name="stream">Поток</param>

/// <returns>Выполненные команды</returns>

public UndoRedoStack Deserialize(Stream stream)

{

BinaryFormatter binaryFormatter = new BinaryFormatter();

return (UndoRedoStack)binaryFormatter.Deserialize(stream);

}

}

Листинг 1.1 – Код класса *DraftSerealizer*

Класс *DraftSerealizer* предназначен для одной задачи – сереиализация/десериализация стека команд. Это хорошо.

**Негативный пример** - *StorageManager*. На рисунке 1.1 отображена диаграмма класса *StorageManager*.

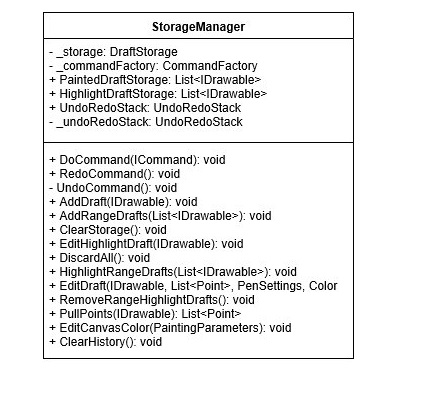


Рисунок 1.1 – Диаграмма класса *StorageManager*

Очевидно, что, к примеру, функция *PullPoints(IDrawable):List<Point>*, которая предназначена для получения точек из объекта, не связана логически с хранилищем объектов *DraftStorage*. Создавать отдельный класс под эту функцию посчиталось неоптимальным. Поэтому в ущерб принципу единственной ответственности функция помещена в класс *StorageManager*.

## **2 Принцип открытости/закрытости**

**Положительный пример** – класс *DrawerFacade*, который реализует общий интерфейс для доступа к объектам с базовым типом*BaseDrawer*. Класс *DraftPainter* служит для обработки данных, связанных с параметрами рисуемых объектов, порядком отрисовки объектов и прочего, но не используется для непосредственного рисования. Классы *LineDraver*, *CircleDriver* и тд. реализуют логику рисования фигур. Диаграмма описанных выше классов представлена на рисунке 2.1.

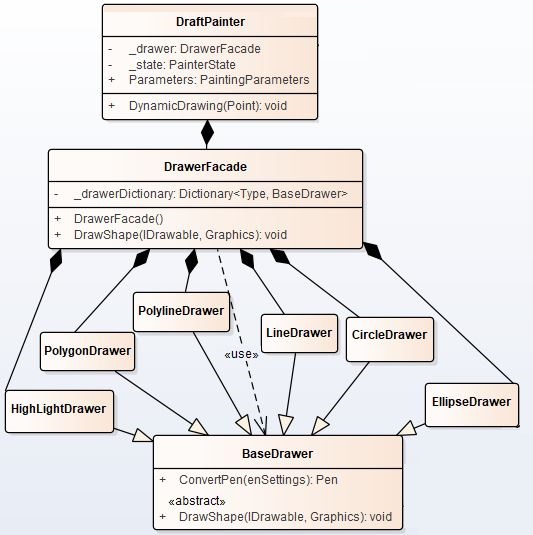


Рисунок 2.1 – Диаграмма классов причастных к рисованию объектов

На листинге 2.1 приведен пример участка кода демонстрирующий использование класса *DrawerFacade* классом *DraftPainter.*

private void DynamicDrawing(Point mousePoint)

{

/// . . .

\_drawer.DrawShape(State.CacheDraft, Painter);

/// . . .

}

Листинг 2.1 – Функция *DynamicDrawing*

Такая программная архитектура гарантирует, что в случае добавления дополнительных средств отрисовки, код, который вызывает отрисовку, не будет подвергнут изменению.

**Негативный пример –** метод *IDrawable PointSearch(Point, List<IDrawable>)* класса *Selector.* Данный метод реализует логику поиска объекта по заданной точке. На листинге 2.2 приведен участок кода, где есть явное нарушение принципа открытости/закрытости.

public IDrawable PointSearch(Point mousePoint, List<IDrawable> draftList)

{

///...

for (var i = draftList.Count - 1; i > -1; i--)

{

///...

if (draftList[i] is IMultipoint multipoint)

{

foreach (var point in multipoint.DotList)

{

sx = Math.Min(sx, point.X);

ex = Math.Max(ex, point.X);

sy = Math.Min(sy, point.Y);

ey = Math.Max(ey, point.Y);

}

}

else if (draftList[i] is Circle)

{

///...

///...

}

///...

}

return null;

}

Листинг 2.2 – Функция *PointSearch*

Приведенный выше код, завязан на конкретных классах, лишь частично присутствует завязка на интерфейсе *IMultipoint.* Добавление в приложение объектов, не реализующих интерфейс *IMultipoint,* с высокой долей вероятности приведет к необходимости изменения кода функции *PointSearch* класса *Selector.*

## **3 Принцип подстановки Барбары Лисков**

В качестве позитивного примера обратимся к диаграмме классов 3.1.

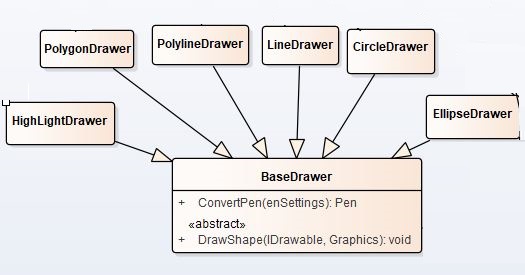


Рисунок 3.1 – Диаграмма классов отрисовщиков

На листинге 3.1 приведен код примера дочернего класса LineDrawer.

/// <summary>

/// Отрисовщик линий

/// </summary>

public class LineDrawer: BaseDrawer

{

/// <summary>

/// Отрисовать линию

/// </summary>

/// <param name="shape">Линия</param>

/// <param name="graphics">Ядро рисования</param>

public override void DrawShape(IDrawable shape, Graphics graphics)

{

var factory = new DrawFactory();

graphics.DrawLine(ConvertPen(shape.Pen),

shape.StartPoint,

shape.EndPoint);

}

}

Листинг 3.1 – Класс *LineDrawer*

Для **негативного примера** обратимся к уже ранее описанному методу *IDrawable PointSearch(Point, List<IDrawable>)* класса *Selector -* листинг 2.2*.* Для поиска объекта дочернего класса *Circle*, который реализует интерфейс *IDrawable*, требуется отличная от родительского интерфейса логика поиска, что нарушает принцип подстановки.

## **4 Принцип разделения интерфейсов**

**Положительный пример** архитектуры приведен на рисунке 4.1.

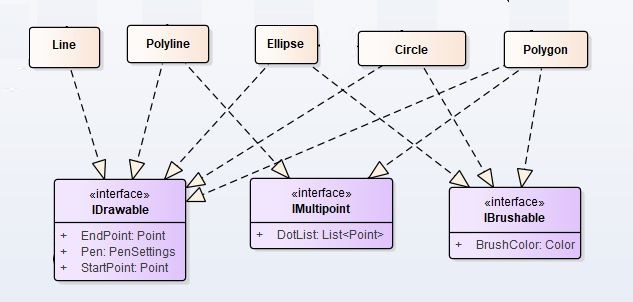


Рисунок 2.1 – Диаграмма классов причастных к рисованию объектов

Интерфейс *IDrawable* является обобщением всех фигур и содержит только максимально общие свойства. Интерфейс *IMultipoint* является обобщением многоточечных фигур, и предоставляет доступ к списку точек объекта. Интерфейс *IBrushable* является обобщением замкнутых фигур и предоставляет доступ к свойству цвет заливки.

## **5 Принцип инверсии зависимостей**

В качестве **позитивного решения** рассмотрим метод *SoloDraw(IDrawable draft*) класса *DraftPainter*, код которого на листинге 5.2.

/// <summary>

/// Отрисовка одного объекта

/// </summary>

/// <param name="draft"></param>

public void SoloDraw(IDrawable draft)

{

\_drawer.DrawShape(draft, Painter);

}

Листинг 5.2 – Фрагмент функции *BiasObject*

Данный метод вызывает отрисовку объекта, ничего о нем не зная. Благодаря тому, что в качестве типа входного параметра используется не конкретный тип, а абстракция *IDrawable,* добавление в систему дополнительных классов фигур не повлечет за собой переработку метода *SoloDraw* .

В качестве **негативного примера** рассмотрим метод RepairCommands(ICommand[] commands) класса DrawManager. Ниже приведен фрагмент метода RepairCommands.

/// <summary>

/// Актуализировать комманды, работающие по ссылкам

/// </summary>

/// <param name="commands">Массив комманд</param>

private void RepairCommands(ICommand[] commands)

{

foreach (var cmd in commands)

{

switch (cmd)

{

case AddDraftCommand addDraftCommand:

addDraftCommand.DraftList =

DraftStorageManager.PaintedDraftStorage;

continue;

case AddRangeDraftCommand addRangeDraftCommand:

addRangeDraftCommand.TargetStorage =

DraftStorageManager.PaintedDraftStorage;

continue;

case ClearStorageCommand clearStorageCommand:

clearStorageCommand.TargetStorage =

continue;

///...

///...

}

}

Листинг 5.1 – Фрагмент функции RepairCommands

Данный метод использует конкретные классы для работы. Расширение функционала приложения может привести к необходимости переработке реализации данного метода.