|  |
| --- |
| Министерство образования и науки РФ  Санкт-Петербургский государственный электротехнический университет  «ЛЭТИ» имени В.И. Ульянова (Ленина) |
| Факультет компьютерных технологий и информатики  Кафедра автоматики и процессов управления |

**Методические указания**

«Разработка простого MDA-приложения»

**РАЗРАБОТКА ПРОСТОГО MDA-ПРИЛОЖЕНИЯ**

**Цель работы:** ознакомиться с архитектурой, управляемой моделью MDA; научиться создавать простое приложение по технологии MDA.

**1. Краткие теоретические сведения**

## **1.1. Архитектура, управляемая моделью MDA**

**MDA** – Model Driven Architecture – Архитектура, управляемая моделью.

Идея заключается в том, что для конструирования программного приложения должна быть построена подробная, формально точная модель, из которой потом может быть автоматически генерирован исполняемый программный код приложения.

Основные шаги разработки:

1. Сначала разрабатывается модель предметной области проектируемого приложения, полностью независимая от имплементирующей технологии. Она называется Platform Independent Model (**PIM**).
2. Затем PIM автоматически трансформируется специальным инструментом в платформо-зависимую модель (Platform Specifical Model, **PSM**).
3. PSM переводится в исходный код на соответствующем языке программирования. И, при необходимости, генерируется база данных. При этом генерация кода и баз данных осуществляется автоматически, посредством специальных инструментальных программных средств.

Таким образом, в соответствии с концепцией MDA, главный акцент при разработке приложений переносится с этапа программирования на этап создания модели.

Преимущества такого подхода:

1. MDA-приложение полностью реализует бизнес–логику ввиду того, что приложение строго соответствует модели.
2. Существенно ускоряется процесс создания приложения, за счет автоматической генерации кода приложения и структур данных\*.

*\*Следует заметить, что на сегодня не существует продуктов, полностью реализующих концепцию MDA.*

**Модель приложения** – это взаимосвязанный набор визуальных диаграмм, наглядно описывающих внутреннюю структуру системы и принципы ее функционирования. Модель приложения не привязана к конкретному языку или конкретной среде программирования.

При разработке модели воспользуемся инструментом, который позволяет разрабатывать UML-модель приложения и по ней генерировать код, в данном примере будет использоваться бесплатный инструмент: <http://argouml.tigris.org/>

**ArgoUML** – средство UML моделирования. ArgoUML является открытым программным обеспечением и распространяется под лицензией EPL.

Функциональность ArgoUML включает в себя:

* Поддержку спецификаций UML 1.3, 1.4, XMI 1.0, 1.1, 1.2;
* 9 видов диаграмм UML (диаграммы классов, состояний, кооперации, последовательности, деятельности, прецедентов, объектов, компонентов, развёртывания);
* Поддержку OCL для классов;
* Генерацию исходного кода Java, C++, C# и PHP;
* Обратный инжиниринг из исходного кода и байткода Java;
* Автоматическую верификацию модели UML (design critics).

**1.2. Диаграмма классов**

**UML** (Unified Modeling Language – Унифицированный язык моделирования) – система обозначений для определения, визуализации и конструирования моделей системы в виде диаграмм и документов на основе объектно-ориентированного подхода.

Словарь UML включает три вида строительных блоков:

* диаграммы
* сущности
* связи

**Диаграммы классов** показывают набор классов, интерфейсов, а также их связи. При построении диаграммы классов окончательно должны быть определены атрибуты и операции классов.

**Класс** – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой.

Особенности задания атрибутов, методов и отношений между классами будут иллюстрироваться с учетом специфики (синтаксиса и семантики) языка программирования Java.

Графически **класс** отображается в виде прямоугольника, который может быть разделен горизонтальными линиями на секции. В этих секциях указывается имя, атрибуты (свойства) и операции (методы).

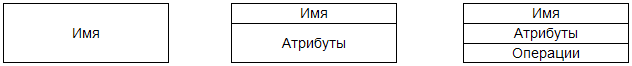


Рисунок 1.1 – Способы отображения класса

С точки зрения структурного подхода, **атрибуты – это переменные**, а **методы – это функции**, описанные в теле класса. Они могут быть доступны или не доступны для изменения (атрибуты) или выполнения (методы) внешними объектами.

Обязательным элементом обозначения класса на диаграмме является его имя. Оно должно быть уникальным в пределах пакета. Если класс является абстрактным, то его имя пишется курсивом. **Абстрактный класс** – это класс, на основе которого нельзя создать объекты. Такие классы используются в качестве шаблона для дочерних классов при наследовании.

В секции имени класса может быть указан стереотип: «entity», «boundary», «interface» и т.п.

Во второй секции каждому атрибуту соответствует отдельная строка со следующей спецификацией:

[видимость] имя [: тип [‘[‘кратность‘]‘] [ = исходное значение]] [‘{‘строка-свойство’}’]

Квадратные скобки означают, что соответствующий элемент спецификации может отсутствовать. Таким образом, при описании обязательным является только имя атрибута.

**Видимость** (visibility) характеризует возможность чтения и модификации значения атрибута для объекта, описываемого класса, из объектов других классов. Модификация значения возможна лишь при условии, что атрибут не является константой. Видимость отображается с помощью следующих символов:

* «+» – общедоступный атрибут (public) – доступен для чтения и модификации из объектов любого класса;
* «#» – защищенный атрибут (protected) – доступен только объектам описываемого класса и его потомкам при наследовании;
* «–» – закрытый атрибут (private) – доступен только объектам описываемого класса;
* «~» – пакетный атрибут (package) – доступен только объектам классов, входящих в тот же пакет.

**Имя** (name) атрибута представляет собой строку текста, которая используется для его идентификации. Оно должно быть уникальным в пределах класса.

**Тип** (type) атрибута выбирается исходя из семантики значений, которые должны храниться в атрибуте, и, как правило, возможностей целевого языка программирования по представлению этих значений. Он соответствует одному из стандартных типов, определенных в этом языке (например, String, Boolean, Integer, Color и т.д.) или имени класса, на объекты которого в этом атрибуте будет храниться ссылка. Во втором случае класс, имя которого указано в качестве типа, должен быть определен на диаграмме или в модели.

**Кратность** (multiplicity) атрибута характеризует количество значений, которые можно хранить в атрибуте. Если кратность атрибута не указана, то по умолчанию принимается ее значение равное 1, т.е. атрибут является атомарным. Такой вариант допускает и отсутствие значения в атрибуте (null). Для атрибута, представляющего собой массив, множество, список и т.п., требуется указание кратности, которая записывается после типа в квадратных скобках. Варианты указания кратности, имеющие смысл, могут быть следующие:

* [0..\*] или [\*] – количество хранимых значений может принимать любое положительное целое число, большее или равное 0. Такой вариант задания кратности характерен для множеств, списков и других атрибутов, допускающих добавление или удаление элементов;
* [0..<число>] – количество хранимых значений, может быть не более указанного числа. Данный вариант применяется при описании массивов фиксированного размера. При этом не обязательно, чтобы все элементы массива имели конкретные значения;
* [0..<число>] [0..<число>] – применяется при описании двумерных массивов. Аналогичным образом можно описать трехмерные, четырехмерные и т.д. массивы.

**Исходное значение** (default value) служит для задания некоторого начального значения атрибута в момент создания отдельного экземпляра класса (объекта).

**Строка-свойство** (property-string) описывает особенности реализации атрибута. В частности, в строке-свойстве можно указывать следующую информацию:

* {final} или {readOnly} – атрибут является константой, т.е. доступен только для чтения;
* {static} – атрибут в момент выполнения программы в конкретный момент времени будет иметь одно и то же значение для всех объектов класса;
* {transient} – атрибут и его значение при записи объекта в БД или файл (сериализации объекта) не должно запоминаться;
* {derived} – значение атрибута определяется по значениям других атрибутов (производный атрибут);
* {unique} – значения неатомарного атрибута должны быть уникальны;
* {ordered} или {unordered} – значения неатомарного атрибута должны быть отсортированы (ordered) или могут содержаться в произвольном порядке (unordered).

Допускается в строке-свойстве, через запятую, указывать сразу несколько модификаторов. Например, {ordered, unique} означает, что элементы массива должны быть уникальны и следовать в строго определенном порядке.

В Java типы, указанные с прописной буквы, называются примитивными (например, double или boolean). Значения атрибутов такого типа непосредственно хранится в объекте. Типы, указанные с заглавной буквы, называются ссылочными (например, String или ConnectDB). В атрибуте такого типа хранится ссылка на объект, созданный на базе соответствующего класса.

В третьей секции указывается перечень методов класса. Можно выделить шесть основных типов методов:

* **конструктор** – метод, создающий и инициализирующий объект. В Java имя конструктора совпадает с именем класса;
* **деструктор** – метод, уничтожающий объект. В некоторых языках программирования (и в Java) определение деструкторов не требуется, т.к. очистка памяти от неиспользуемых объектов (сборка мусора) выполняется автоматически;
* **модификатор** – метод, который изменяет состояние объекта (значения атрибутов). Имена модификаторов начинаются, как правило, со слова set (установить). Например, установить атрибуту Name новое значение setName(newName : String);
* **селектор** – метод, который может только считывать значения атрибутов объекта, но не изменяет их. Имена селекторов начинаются, как правило, со слов get (получить) или is при возврате логического результата. Например, считать значение атрибута Name - getName() или определить видимость на экране элемента графического интерфейса - isVisible();
* **итератор** – метод, позволяющий организовать доступ к элементам объекта. Например, для объекта, представляющего собой множество Set или список List, это могут быть методы перейти к первому элементу first(), следующему next(), предыдущему previous() и т.п.;
* **событие** – метод, запускаемый на выполнение автоматически при выполнении определенных условий.

Тип метода при построении диаграммы классов, как правило, указывается с помощью стереотипа.

Каждому методу соответствует отдельная строка со следующей спецификацией:

[видимость] имя ([список параметров]) [: тип] [‘{‘строка-свойство’}’]

**Имя** и **кратность** параметра задаются по тем же правилам, что и для атрибутов класса.

**Тип** **параметра** – тип значений, которые может принимать параметр.

**Значение по умолчанию** – значение, которое передается в метод, если при вызове метода данный параметр не определен.

**Тип метода** – тип результата, возвращаемого методом. Если тип не указан, то метод не возвращает никакого результата и обозначается с помощью модификатора void.

**Строка-свойство** метода служит для указания специфических свойств метода. В частности, в строке-свойстве можно указывать следующую информацию:

* {native} – реализация метода зависит от платформы (операционной системы);
* {abstract} – метод в описываемом классе не имеет тела. Код метода должен быть определен в дочерних классах;
* {synchronized} – при одновременном (параллельном) вызове метода с помощью разных сообщений, должна гарантироваться целостность объекта и корректность обработки информации.

В Java методы, не возвращающие результата, обозначаются с помощью модификатора void. Для остальных методов обязательно должен указываться тип возвращаемого результата. Исключение составляют конструкторы, для которых тип результата не указывается, а возвращаемое значение является ссылкой на объект, который был создан при вызове конструктора.

Далее иллюстрируется **связь** между графическим отображением классов и отношений на диаграммах и исходными текстами программ. Современные Case-средства при разработке классов, как правило, работают в режиме синхронизации диаграмм и исходных текстов. Т.е. если меняется диаграмма классов, то это приводит к автоматической корректировке текста программы и наоборот.

**Отношение** **ассоциации** означает наличие атрибута, в котором будет храниться ссылка (ссылки) на объект (объекты) класса, в сторону которого направлена стрелка ассоциации.

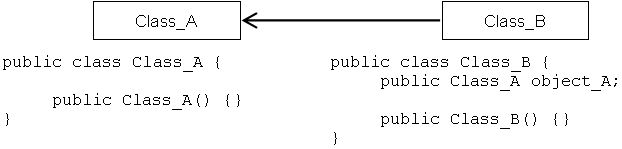


Рисунок 1.2 – Интерпретация ассоциации в тексте программы

Графический символ класса Class\_A преобразуется в строки определения самого класса «public class Class\_A» и его конструктора «public Class\_A() {}». Аналогично для Class\_B. Ассоциация от Class\_B в сторону Class\_A преобразуется в строку «public Class\_A object\_A;», описывающую атрибут object\_A, в котором будет храниться ссылка на объект класса Class\_A. Ввиду отсутствия указания кратности отношения, она по умолчанию принимается равной 1. На рисунке 1.3 приведен пример двунаправленной ассоциации и кратностью более 1.

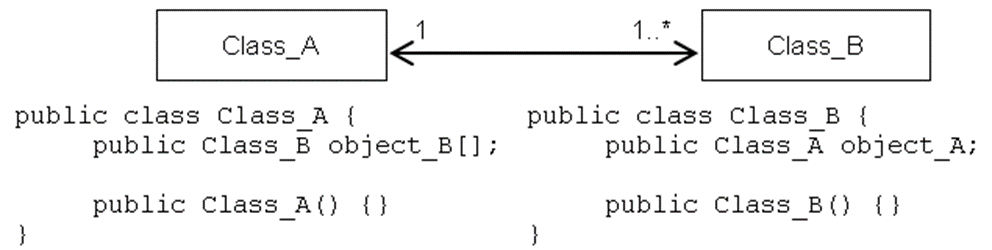


Рисунок 1.3 – Интерпретация двунаправленной ассоциации

Наличие двунаправленной ассоциации или ассоциации без стрелок свидетельствует о наличии в обоих классах атрибутов, содержащих ссылки на объекты. Кратность более 1 подразумевает хранения не одной, а нескольких ссылок. Таким образом, один объект класса Class\_A будет связан с несколькими объектами класса Class\_B. Ссылки на эти объекты будут храниться в массиве object\_B[]. Современные Case-средства позволяют вместо массива указывать другие варианты хранения набора объектов, такие как множества, списки, хешированные таблицы и т.д.

**Отношения** **агрегации** и **композиции** являются частными случаями ассоциации. В связи с этим интерпретация этих отношений с точки зрения текста программы совпадает с рассмотренной выше.

Окончание агрегации () или композиции () интерпретируется, как стрелка. Если требуется хранить ссылки на объекты-части в объекте-целом, то с обратной стороны отношения показывается стрелка.

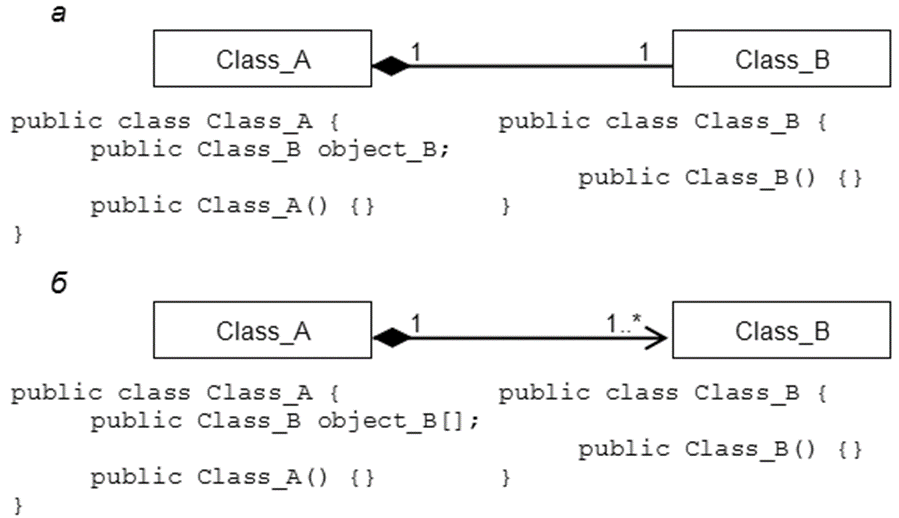


Рисунок 1.4 – Интерпретация композиции (агрегации) (а);

двунаправленной композиции (агрегации) (б)

**Отношение** **обобщения** в тексте программы на языке Java показывается ключевым словом «extends» (расширяет) в дочернем классе.

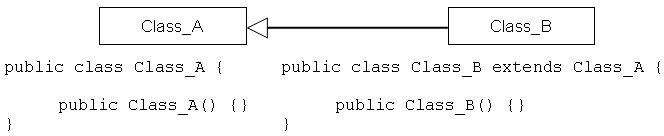


Рисунок 1.5 – Интерпретация обобщения

Отношение зависимости не приводит к автоматической генерации кода программы, но свидетельствует об обращение из объекта зависимого класса к атрибутам, методам или непосредственного к объектам независимого класса. Данное отношение в Case-средстве может автоматически отображаться на диаграмме при обратном проектировании или при синхронизации диаграммы и текста программы.

На рисунке 1.6 показан условный пример, свидетельствующий о наличии зависимости класса Class\_B от класса Class\_A. В строке «public obrabotka(Class\_A object\_A)» используется ссылка на объект класса Class\_A. В строке «String name = object\_A.name;» выполняется обращение к атрибуту объекта класса Class\_A. В строке «int age = object\_A.getAge();» выполняется обращение к методу объекта класса Class\_A.

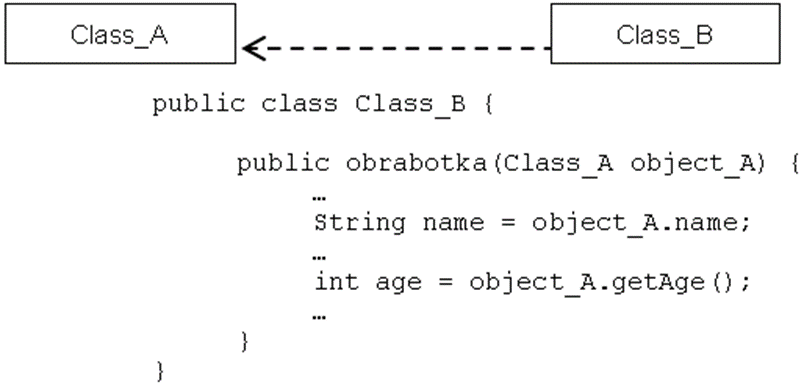


Рисунок 1.6 – Интерпретация зависимости

При одновременном наличии между классами отношений типа ассоциация и зависимость на диаграмме отображается ассоциация, как более сильная связь.

На диаграммах разного типа, в т.ч. и на диаграмме классов, для показа специфики связей, поведения или взаимодействия могут отображаться объекты. Вид объекта аналогичен классу, но при этом его имя должно быть подчеркнуто и, как правило, приведены атрибуты и их значения, вызвавшие необходимость отображения объекта на диаграмме.

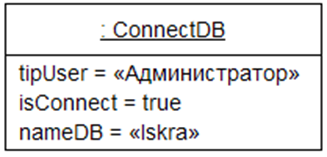


Рисунок 1.7 - Пример объекта

**1.3. Контрольные вопросы**

1. Что представляет собой технология MDA?

2. Что такое UML и зачем он используется?

3. Какая UML диаграмма наиболее часто используется для создания модели MDA?

**2. Создание простого MDA-приложения**

**2.1. Создание модели UML**

Для примера разработаем приложение, рассчитывающее площадь и периметр фигуры (прямоугольника, ромба и треугольника). При анализе было выявлено 4 основные сущности:

* **Форма**. Это абстрактный класс, для которой мы будем хранить информацию о площади и периметре.
* **Прямоугольник**. Содержит в себе информацию о длине и ширине и методы по расчету периметра и площади прямоугольника.
* **Ромб**. Содержит в себе информацию о длине и высоте и методы по расчету периметра и площади ромба.
* **Треугольник**. Содержит в себе информацию об основании, высоте и длинах трех сторон и методы по расчету периметра и площади треугольника.

Получим UML модель:

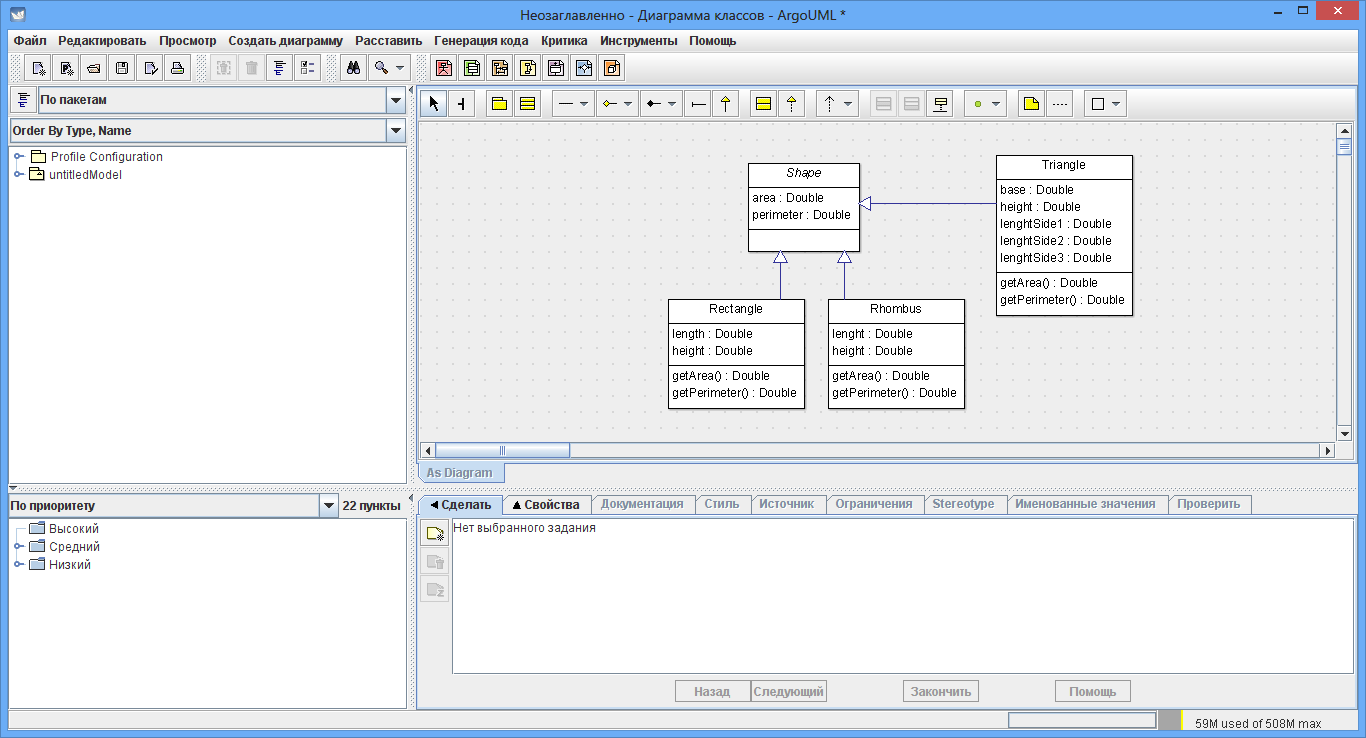


Рисунок 1 – Диаграмма классов

С помощью средств AgroUML получим классы:

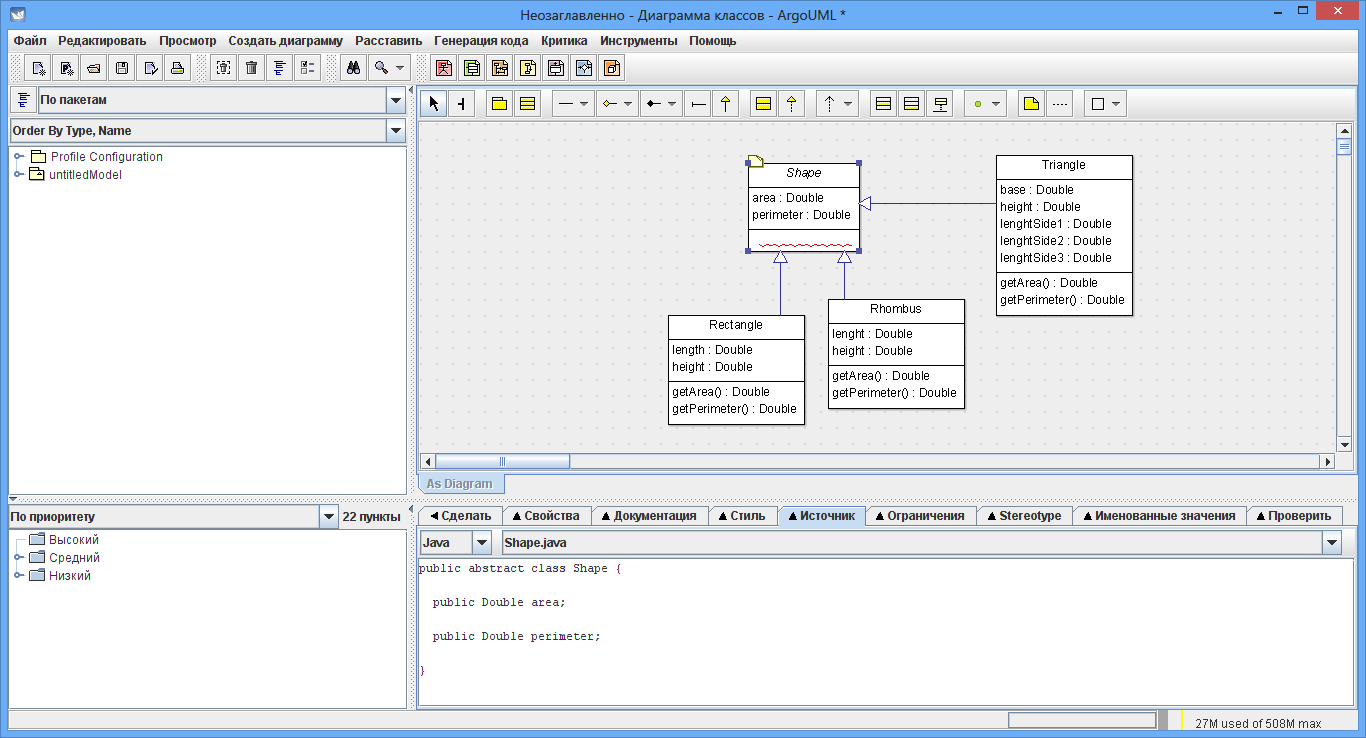


Рисунок 2 – Авто сгенерированный класс Форма

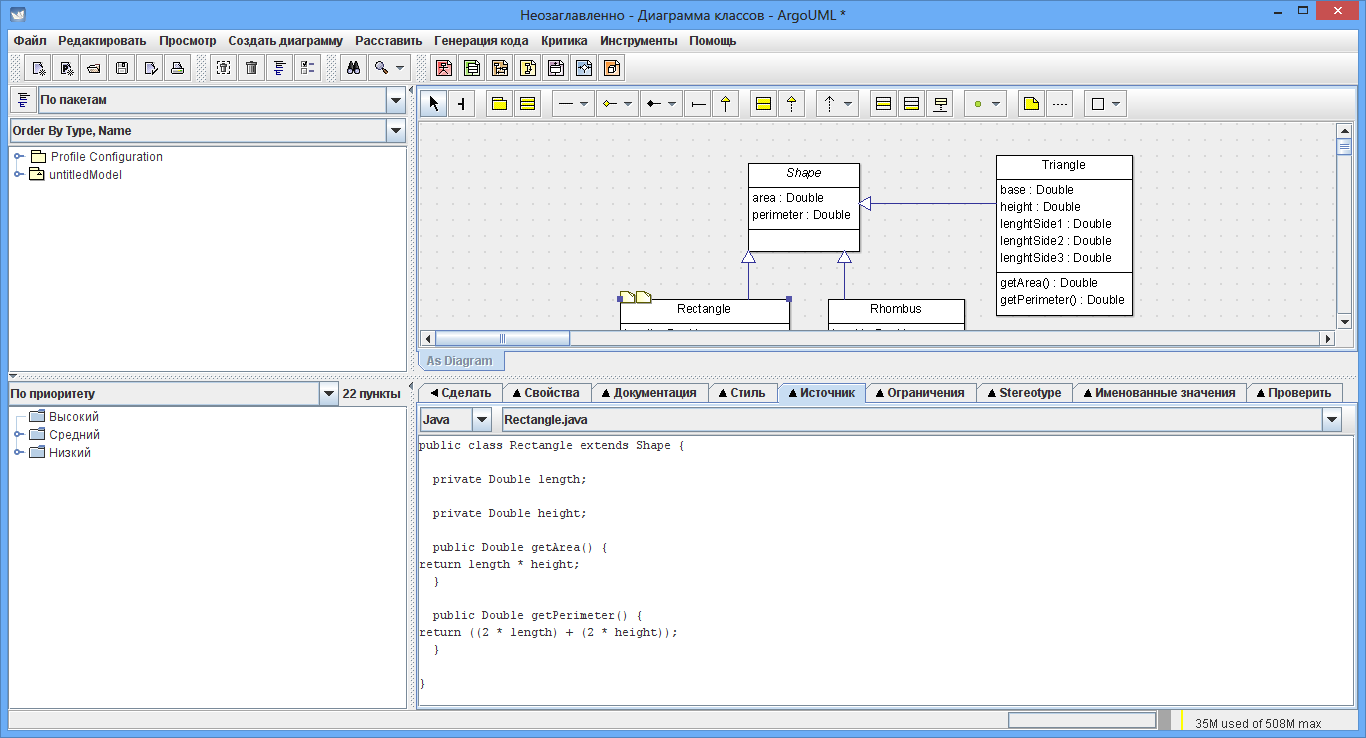


Рисунок 3 – Авто сгенерированный класс Прямоугольник

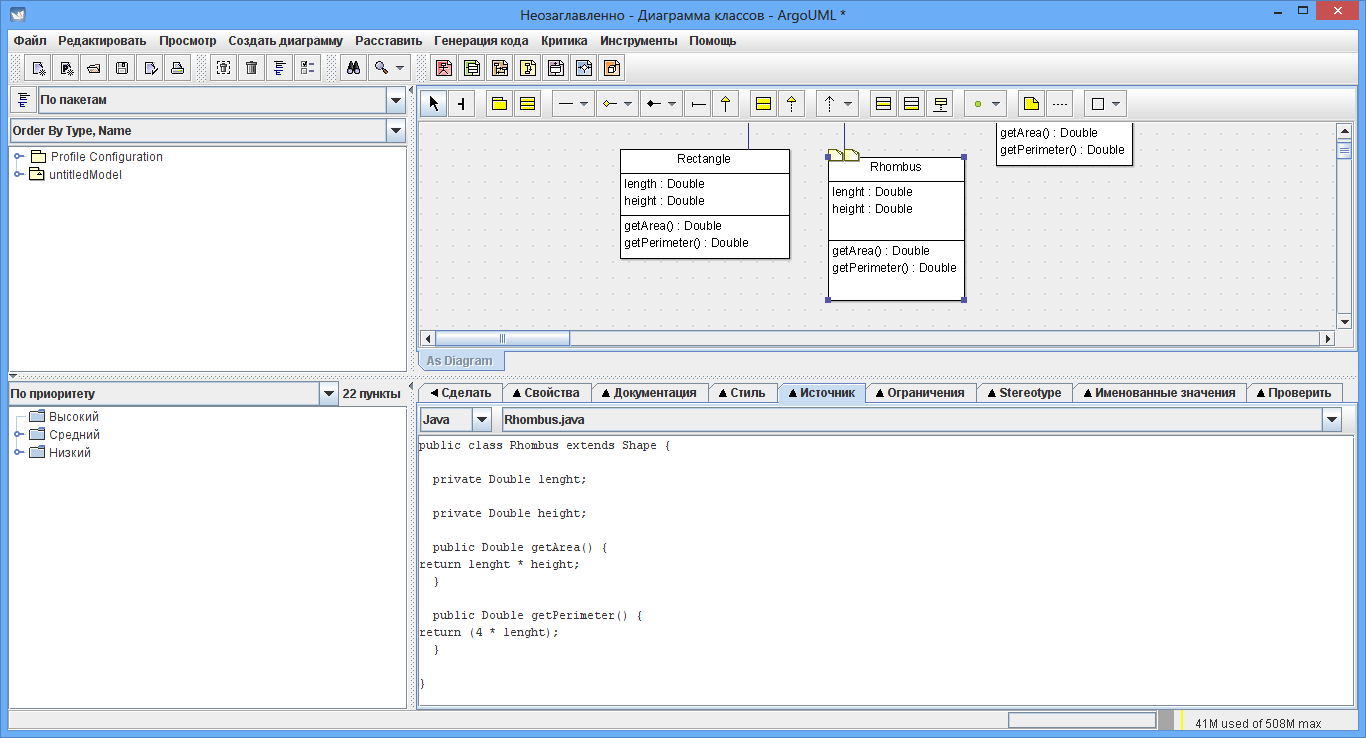


Рисунок 4 – Авто сгенерированный класс Ромб

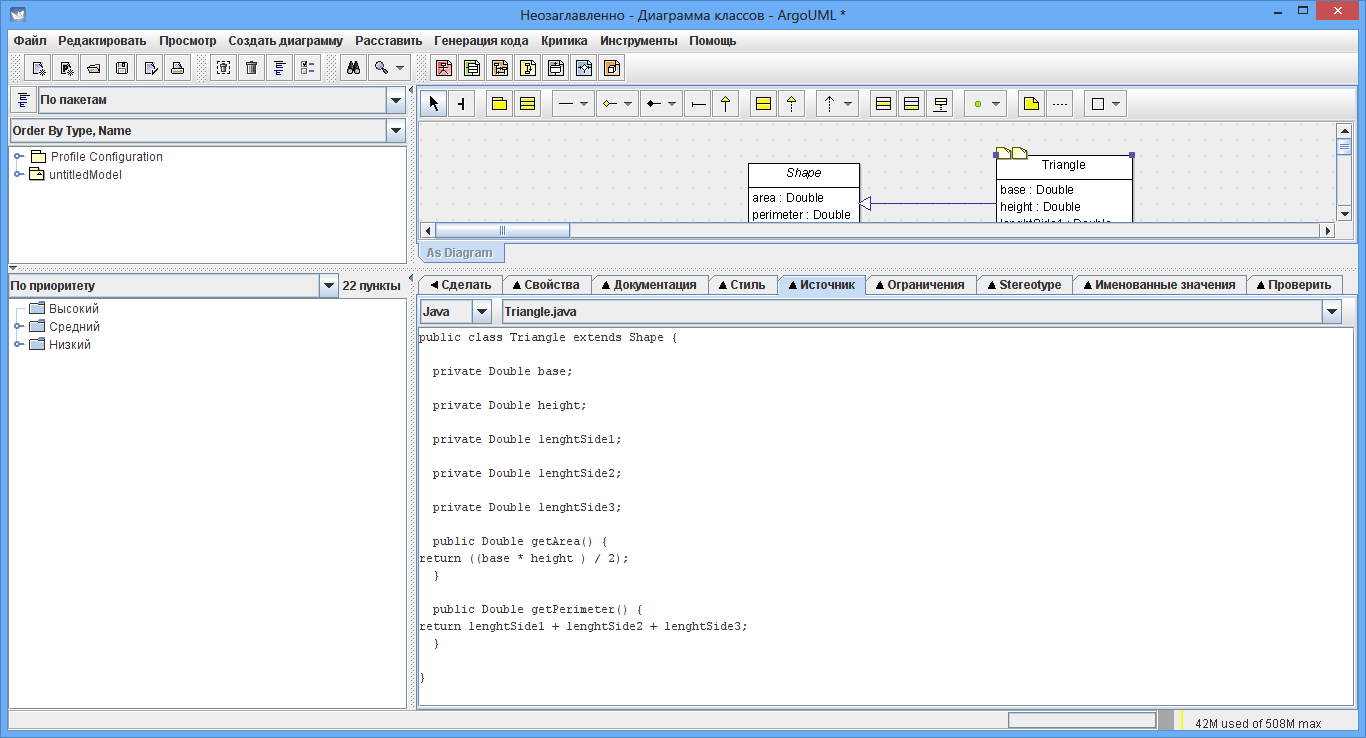
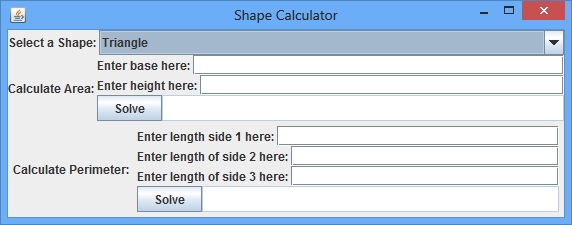


Рисунок 5 – Авто сгенерированный класс Треугольник

**2.2. Создание интерфейса**

Интерфейс представляет собой главное окно, на котором расположены:

* выпадающий список "Select a Shape" (для выбора фигуры пользователем);
* текстовые поля для ввода значений;
* кнопка "Solve" для выполнения расчетов;
* поля для вывода результата.

Рисунок 6 – Форма главного окна

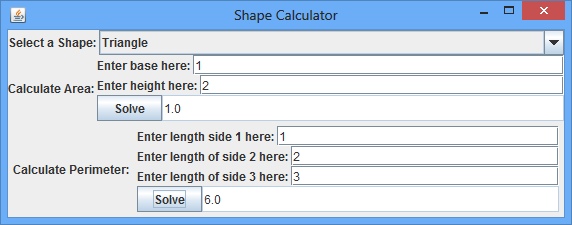


Рисунок 7 – Пример работы программы (фигура треугольник)

Так же ниже представлен код класса "Triangle" написанный на java:

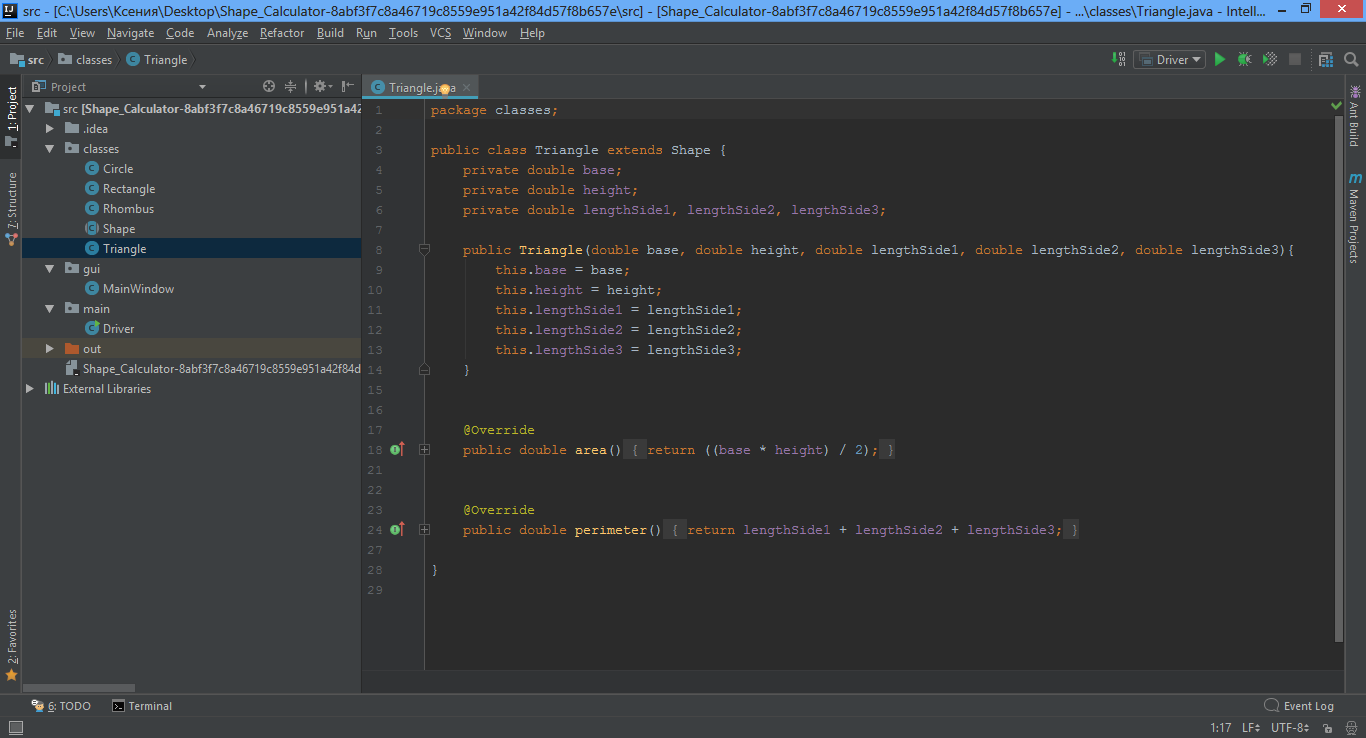


Рисунок 8 – Код класса "Triangle"

Код отрисовки формы для приложения (фигура треугольник) так же представлен ниже для разбора:

private void setupTriangle(JPanel areaParametersPanel, JPanel perimeterParametersPanel){  
 JPanel areaBasePanel = new JPanel();  
 areaBasePanel.setLayout(new BoxLayout(areaBasePanel, BoxLayout.X\_AXIS));  
 JLabel areaBaseLabel = new JLabel("Enter base here: ");  
 final JTextField areaBaseBox = new JTextField(24);  
 areaBasePanel.add(areaBaseLabel);  
 areaBasePanel.add(areaBaseBox);  
  
 JPanel areaHeightPanel = new JPanel();  
 areaHeightPanel.setLayout(new BoxLayout(areaHeightPanel, BoxLayout.X\_AXIS));  
 JLabel areaHeightLabel = new JLabel("Enter height here: ");  
 final JTextField areaHeightBox = new JTextField(24);  
 areaHeightPanel.add(areaHeightLabel);  
 areaHeightPanel.add(areaHeightBox);  
  
 JPanel areaSolutionPanel = new JPanel();  
 areaSolutionPanel.setLayout(new BoxLayout(areaSolutionPanel, BoxLayout.X\_AXIS));  
 JButton solveAreaButton = new JButton("Solve");  
 final JTextField areaSolutionField = new JTextField(24);  
 areaSolutionField.setBackground(Color.WHITE);  
 areaSolutionField.setEditable(false);  
 areaSolutionPanel.add(solveAreaButton);  
 areaSolutionPanel.add(areaSolutionField);  
  
 areaParametersPanel.add(areaBasePanel);  
 areaParametersPanel.add(areaHeightPanel);  
 areaParametersPanel.add(areaSolutionPanel);  
  
 solveAreaButton.addActionListener(new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent event) {  
 Triangle triangle = new Triangle(Double.parseDouble(areaBaseBox.getText()), Double.parseDouble(areaHeightBox.getText()), 0, 0, 0);  
 areaSolutionField.setText(triangle.area()+"");  
 }  
 });  
  
 JPanel perimeterLengthPanel = new JPanel();  
 perimeterLengthPanel.setLayout(new BoxLayout(perimeterLengthPanel, BoxLayout.X\_AXIS));  
 JLabel perimeterLengthLabel = new JLabel("Enter length side 1 here: ");  
 final JTextField perimeterLengthBox = new JTextField(24);  
 perimeterLengthPanel.add(perimeterLengthLabel);  
 perimeterLengthPanel.add(perimeterLengthBox);  
  
 JPanel perimeterLength2Panel = new JPanel();  
 perimeterLength2Panel.setLayout(new BoxLayout(perimeterLength2Panel, BoxLayout.X\_AXIS));  
 JLabel perimeterLength2Label = new JLabel("Enter length of side 2 here: ");  
 final JTextField perimeterLength2Box = new JTextField(24);  
 perimeterLength2Panel.add(perimeterLength2Label);  
 perimeterLength2Panel.add(perimeterLength2Box);  
   
 JPanel perimeterLength3Panel = new JPanel();  
 perimeterLength3Panel.setLayout(new BoxLayout(perimeterLength3Panel, BoxLayout.X\_AXIS));  
 JLabel perimeterLength3Label = new JLabel("Enter length of side 3 here: ");  
 final JTextField perimeterLength3Box = new JTextField(24);  
 perimeterLength3Panel.add(perimeterLength3Label);  
 perimeterLength3Panel.add(perimeterLength3Box);  
   
   
 JPanel perimeterSolutionPanel = new JPanel();  
 perimeterSolutionPanel.setLayout(new BoxLayout(perimeterSolutionPanel, BoxLayout.X\_AXIS));  
 JButton solvePerimeterButton = new JButton("Solve");  
 final JTextField perimeterSolutionField = new JTextField(24);  
 perimeterSolutionField.setBackground(Color.WHITE);  
 perimeterSolutionField.setEditable(false);  
 perimeterSolutionPanel.add(solvePerimeterButton);  
 perimeterSolutionPanel.add(perimeterSolutionField);  
  
 solvePerimeterButton.addActionListener(new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent event) {  
 Triangle triangle = new Triangle(0, 0, Double.parseDouble(perimeterLengthBox.getText()), Double.parseDouble(perimeterLength2Box.getText()), Double.parseDouble(perimeterLength3Box.getText()));  
 perimeterSolutionField.setText(triangle.perimeter()+"");  
 }  
 });  
  
 perimeterParametersPanel.add(perimeterLengthPanel);  
 perimeterParametersPanel.add(perimeterLength2Panel);  
 perimeterParametersPanel.add(perimeterLength3Panel);  
 perimeterParametersPanel.add(perimeterSolutionPanel);  
  
 areaParametersPanel.revalidate();  
 areaParametersPanel.repaint();  
 perimeterParametersPanel.revalidate();  
 perimeterParametersPanel.repaint();  
 pack();  
}

**3. Задания и порядок выполнения работы**

Для вариантов 1-4: необходимо реализовать пример, приведенный в п.2, используя технологию MDA, и дополнить программу своим заданием.

Для вариантов 5-8: необходимо на основе примера, приведенного в п.2, выполнить задание, используя технологию MDA.

Список вариантов:

1. Добавьте четвертую фигуру.

1. На диаграмме необходимо добавить ещё один класс, задав ей необходимые атрибуты и методы.
2. Можно добавить: круг, овал, трапецию, параллелограмм или др.
3. Рассчитать для этой фигуры площадь и периметр.

2. Выводить тип фигуры.

1. На диаграмме необходимо добавить атрибут «type» и метод «setType» для определения типа фигуры.
2. Для Прямоугольника и Ромба проверить, являются ли они квадратами.
3. Для Треугольника вывести его тип: равносторонний (A=B=C), равнобедренный (напр., A=B≠C), остроугольный, тупоугольный, прямоугольный, произвольный или таково треугольника не существует (A<B+C и .т.д.). Треугольник может содержать комбинацию типов (например, равнобедренный и тупоугольный).

3. Найти и вывести радиус вписанной и описанной окружности.

1. На диаграмме необходимо добавить два атрибута «radiusExternal» и «radiusInternal» и добавить два метода «getRadiusExternal» и «getRadiusInternal» для расчета значений радиусов.
2. Для Прямоугольника найти радиус описанной окружности и найти (если есть) радиус вписанной окружности.
3. Для Ромба найти радиус вписанной окружности и найти (если есть) радиус описанной окружности.
4. Для треугольника найти радиус вписанной и описанной окружности, если треугольник с заданными длинами сторон существует.

4. Графически изобразить фигуры с заданными пользователем параметрами.

1. На диаграмме классам Прямоугольник, Ромб и Треугольник необходимо добавить атрибуты и методы для расчета точек (X, Y) каждой вершины фигуры.
2. После нажатия кнопки Solve на панели Canvas должна прорисоваться соответствующая геометрическая фигура с заданными длинами сторон.

5. Создать новую подобную диаграмму классов про Треугольники.

1. Класс Triangle (Треугольник) станет обобщенным абстрактным классом. Атрибутами класса останутся «area» и «perimeter».
2. Классы Equilateral (Равносторонний), Isosceles (Равнобедренный), Arbitrary (Произвольный) – зависимые классы от Triangle (т.е. наследуют его атрибуты).
3. Атрибуты, значения которых задаются пользователем:

Для класса Equilateral: «lengthAllSides».

Для класса Isosceles: «lengthSides1and2» и «lengthSide3».

Для класса Arbitrary: «lengthSide1», «lengthSide2», «lengthSide3».

1. Для каждого класса (кроме Triangle) есть атрибуты «base» и «height». Они не задаются пользователем, а рассчитываются по формулам с помощью методов «setBase», «setHeight».
2. Реализовать расчет площади и периметра для всех треугольников.
3. Осуществить поверку на существование треугольника с введенными длинами сторон. Если условие не выполняется (третья сторона меньше суммы двух других сторон), то выводить ошибку.

6. Создать новую подобную диаграмму классов про Треугольники.

1. Класс Triangle (Треугольник) станет обобщенным абстрактным классом. Атрибутами класса останутся «area» и «perimeter».
2. Классы Oxygon (Остроугольный), Obtuse (Тупоугольный), Right (Прямоугольный) – зависимые классы от Triangle (т.е. наследуют его атрибуты).
3. Атрибуты, значения которых задаются пользователем:

Для класса Oxygon: «angle1», «angle2» («angle3» рассчитывается системой, также делается проверка на то, что все углы <90°).

Для класса Obtuse: «angle1» и «angle2» («angle3» рассчитывается системой, также делается проверка на то, что один из углов >90°, а два других <90°).

Для класса Right: «angle1» («angle2» рассчитывается системой, («angle2» = 90° также делается проверка на то, что angle1<90°).

1. Реализовать расчет площади и периметра для всех треугольников.

7. Создать новую подобную диаграмму классов про Четырехугольники.

1. Класс Quadrilateral (Четырехугольник) станет обобщенным классом. В этом случае класс не абстрактный.
2. Классы Rectangle (Прямоугольник), Rhombus (Ромб), Square (Квадрат), Trapeze (Трапеция), Parallelogram (Параллелограмм) – зависимые классы от Quadrilateral (т.е. наследуют его атрибуты).
3. Атрибутами класса являются стороны фигуры и высота (для трапеции и параллелограмма). Все длины задаются пользователем.
4. Реализовать расчет площади и периметра для всех шести фигур (произвольный четырехугольник, прямоугольник, ромб, квадрат, трапеция, параллелограмм).

8. Создать новую подобную диаграмму классов про Многоугольники.

1. Класс Polygon (Многоугольник) станет обобщенным абстрактным классом.
2. Классы Triangle (Треугольник), Quadrilateral (Четырехугольник), Pentagon (Пятиугольник), Hexagon (Шестиугольник), Heptagon (Семиугольник), Octagon (Восьмиугольник) – зависимые классы от Polygon (т.е. наследуют его атрибуты).
3. Атрибутами класса являются стороны фигуры. Длины сторон задаются пользователем.
4. Реализовать расчет периметра для всех шести фигур (от треугольника, до восьмиугольника).

Итоговый отчет лабораторной работы должен содержать:

* Диаграмму классов приложения;
* Авто сгенерированную основу классов приложения;
* Оформленный отчет, включающий в себя ответы на контрольные вопросы;
* Десктоп-приложение, написанное на Java.

Правила программного кода

* Все основные классы должны быть созданы в модели приложения при помощи специальных UML инструментов (ArgoUML).