Федеральное агентство по образованию
Псковский государственный политехнический институт

***Антонов И.В., Бруттан Ю.В.***

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ (C#)

***Учебно-методическое пособие***
*для студентов специальности 230201*
*"Информационные системы и технологии"*
*заочной формы обучения*

*Рекомендовано Научно-методическим советом*
*Псковского государственного политехнического института*

Псков
Издательство ППИ
2010

УДК 681.3.06
ББК 32.97
Б46

*Рекомендовано к изданию Научно-методическим советом Псковского государственного политехнического института*

Рецензенты:
– Воронов М.В. – д.т.н., профессор
– Плохов И.В. – д.т.н., профессор

**Антонов И.В., Бруттан Ю.В.**
**Объектно-ориентированное программирование (C#)**.
Учебно-методическое пособие для студентов специальности 230201 "Информационные системы и технологии" заочной формы обучения / Сост. И.В. Антонов, Ю.В. Бруттан. – Псковский государственный политехнический институт. – Псков: Издательство ППИ, 2010. — 50с.

В учебно-методическом пособии изложены теоретические и практические основы знания по изучению объектно-ориентированного подхода к программированию. Материал учебного пособия содержит изложение основ объектно-ориентированного программирования, общие рекомендации по изучению дисциплины студентами заочной формы обучения, контрольные вопросы, рекомендации по выполнению самостоятельной работы, задания для выполнения контрольных работ, рекомендации по выполнению контрольных работ, задания к 3 лабораторным работам вместе со справочными материалами по теоретическим аспектам выполнения заданий и подробные рекомендации по выполнению лабораторных работ.

Предназначено для студентов специальности 230201 "Информационные системы и технологии" заочной формы обучения.

УДК 681.3.06
ББК 32.97

# СОДЕРЖАНИЕ

стр.

# ВВЕДЕНИЕ

Объектно-ориентированное программирование (ООП) – это технология программирования, самым существенным моментом которой является объединение структур данных с операциями, которые можно осуществлять с этими структурами данных.

ООП дополняет обычные языки программирования механизмами, позволяющими использовать общие методы человеческого мышления при разработке программ, прежде всего – абстрагирование и построение моделей взаимодействия и поведения. Человек при решении задачи манипулирует абстракциями разного уровня, устанавливая между ними связи и выявляя характеризующие их закономерности. Набор понятий решаемой задачи часто представляется в виде дерева абстракций, верхние уровни которого являются обобщением понятий нижних уровней.

Центральным элементом при использовании объектно-ориентированного подхода является объект, представляющий собой модель некоторого элемента реального мира и содержащий в себе как данные, так и операции над ними. Объекты одного типа образуют классы, которые представляют собой основные "строительные блоки" разрабатываемых программ. В классах интегрируются как определения данных, так и определения функций, выполняемых над данными.

Применение нового подхода, основанного на ООП, позволяет в случае успеха получить следующие результаты – сокращение времени на разработку сложных проектов, облегчение повторного использования написанных модулей, снижение издержек на сопровождение и модификацию программного обеспечения.

Инструмент, на примере которого в данном курсе будут рассматриваться элементы ООП – это язык C# – результат эволюции языков C и C++. Являясь новейшей разработкой компании Microsoft, C# конструировался с учётом наилучших возможностей других языков, предназначенных для решения специфических проблем. C# является мощным языком программирования и имеет массу преимуществ: простота, объектная ориентированность, типовая защищенность, сборка мусора, поддержка совместимости версий и многое другое. Данные возможности позволяют быстро и легко разрабатывать приложения.

C# – один из языков программирования высокого уровня, который может использоваться для создания приложений на платформе .NET. Платформа .NET — многоязыковая среда, открытая для свободного включения новых языков, создаваемых не только Microsoft, но и другими фирмами. Все языки, включаемые в платформу .NET, должны опираться на единый каркас, роль которого играет .NET Framework. Главным достоинством языка C# можно назвать его согласованность с возможностями .NET Framework и вытекающую отсюда компонентную ориентированность. Компоненты позволяют решать проблему модульного построения приложений на новом уровне.

## РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ДИСЦИПЛИНЫ

Учебная дисциплина «Объектно-ориентированное программирование» изучается студентами в течение двух семестров. Программой предусмотрено проведение лекционных и лабораторных занятий, а также написание двух контрольных работ.

В весеннем семестре обучающиеся должны написать контрольную №1, выполнить лабораторные работы №1 и №2, сдать экзамен по теоретическим вопросам, а в осеннем семестре – выполнить лабораторную работу №3, написать контрольную №2, получить зачет по результатам защиты лабораторной работы.

Контрольные работы должны быть сданы преподавателю на рецензию до начала зачётной недели. Требования к оформлению контрольных работ приведены в данном пособии в разделе «Контрольные работы»

Лабораторные работы должны быть выполнены в среде Microsoft Visual Studio 2008.

Описание лабораторных работ и рекомендации по их выполнению, задания для контрольных работ, контрольные вопросы к экзамену приведены в данном учебном пособии.

Для подготовки к экзамену необходимо воспользоваться материалами лекций, описанием теоретических вопросов, приведенных в данном пособии. Для расширения познаний в области объектно-ориентированного подхода к программированию можно воспользоваться литературой, указанной в пособии.

## РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Самостоятельная работа студентов по дисциплине «Объектно-ориентированное программирование» заключается, во-первых, в самостоятельной проработке перечисленных ниже теоретических вопросов, а во-вторых, изучении интерфейса среды разработки приложений Microsoft Visual Studio 2008, используя список перечисленных ниже источников.

**Перечень теоретических вопросов для самостоятельной работы:**

1. Стандартные типы данных C#.
2. Перечисления.
3. Многомерные массивы в C#.
4. Преобразование типов.
5. Статические переменные и переменные экземпляра.
6. Переменные только для чтения.
7. Цикл foreach.
8. Свойства.
9. Класс Object.
10. Статические конструкторы.
11. Структуры.
12. Перегрузка операций.
13. Интерфейсы.
14. Уничтожение объектов в C#.
15. Обработка исключительных ситуаций.
16. Построение диаграмм UML.

**Список источников для выполнения самостоятельной работы**

1. Грейди Буч, Джеймс Рамбо, Айвар Джекобсон. Язык UML. Руководство пользователя. – СПб.: Питер, 2004. – 432 с.

2. Павловская Т.А. C#. Программирование на языке высокого уровня. – СПб.:Питер, 2007.

3. Троелсен Э. C# и платформа.NET. Библиотека программиста. – СПб.:Питер, 2007.

4. Трей Нэш. C# 2008: ускоренный курс для профессионалов. Язык программирования C# 3.0 для .NET 3.5. – М.: Издательский дом «Вильямс», 2008.

5. http://www.intuit.ru

6. http://ru.wikipedia.org

## ТЕОРЕТИЧЕСКИЕ ВОПРОСЫ

### *Объектно-ориентированный подход*

В 1980-х появились персональные компьютеры, компьютеры стали использоваться повсеместно. Появилось множество пользователей, желающих применять компьютеры в своей работе. Появилась реальная возможность автоматизации работы предприятий, внедрения «безбумажной» технологии управления. Компьютеры встраивались в контуры управления различных систем и принимали на себя задачи по рутинной обработке информации — делали это быстрее и точнее человека. Машина создавала в своей памяти информационную модель реальности, и синхронизировала эту модель с изменениями, происходящими во внешнем мире, вырабатывала управляющие воздействия для внешнего мира. Однако языковое обеспечение машин не очень подходило для удобного решения задач информационного моделирования. Основной концепцией оставался алгоритм, в лучшем случае — база фактов и правил вывода, предоставляемая логической парадигмой. Но ведь не всё в мире описывается в виде алгоритма. Чаще всего мир представляется в виде каких-то сущностей, обладающих свойствами и каким-либо поведением. Некоторые сущности активны — их поведение самостоятельно, другие — пассивны, и лишь реагируют на внешние воздействия. Самый человеческий язык отражает такой способ мышления в грамматике речи, в структуре предложения. Подлежащее — объект, делает то, что выражается сказуемым, с дополнениями и обстоятельствами. Ещё в Древней Греции философы занимались проблемой адекватности описания реальности на человеческом языке. Возникла мысль, а почему бы эту концепцию не перенести непосредственно в язык программирования. Первым таким языком стал Simula.

Объектно-ориентированный подход кардинально меняет наше представление о программе. Теперь основой программы является не алгоритм, а данные. Программа представляет собой систему взаимодействующих объектов. Каждый объект самостоятельно реагирует на внешние воздействия, если они есть, и не реагирует, если их нет. Реакция целой системы — это алгоритм действия на появление внешнего воздействия, но этот алгоритм собирается «на лету» из тех возможностей, которые предоставляют объекты, и зависит не только от вида воздействия, но и от состояния системы. Поведение объекта — это потенциал тех действий, которые он может выполнить. Можно на эти действия реагировать, а можно и не реагировать. Программа получается гибкой и достаточно легко расширяемой. За счёт соединения внутри концепции объекта поведения и свойств можно абстрагироваться от конкретных действий и данных, выполняемых конкретным объектом, обобщая одинаковость объектов в понятии класса. Каждый объект в системе — это автономный минимодуль, который может существовать во множестве экземпляров. Мы можем формировать в сознании объект целиком, не углубляясь в детали его

реализации. Мы можем разрабатывать объекты не столько в соответствии с потребностями их использующей программы, сколько в соответствии с их собственной природой, известной нам из реального мира. Появилось понимание, что программы, написанные в старых парадигмах — это программы, где *алгоритм управляет данными*. В объектно-ориентированных программах *данные управляют алгоритмом*.

Объектно-ориентированный подход позволил разрабатывать такие большие и сложные программы, которые не под силу создать в рамках алгоритмического подхода. Появились удобные средства для реализации пользовательского интерфейса на качественно новом уровне, новые возможности компьютерного моделирования очень сложных систем.

Рассмотрим основные принципы ООП.

1. ИНКАПСУЛЯЦИЯ – объединение в одном элементе и данных, и процедур их обработки. Инкапсуляция достигается использованием классов.

В традиционном С не обеспечивается однозначной и непосредственной связи между данными и кодом, любой программист может получить доступ к данным, минуя предназначенные для этого функции, изменения в структурах данных могут быть не учтены, во всех без исключения функциях, манипулирующих этими данными. В C# хорошим тоном считается защита данных класса от доступа извне, минуя компонентные функции класса.

2. НАСЛЕДОВАНИЕ – построение новых классов, которые наследуют данные и функции от одного или нескольких ранее определенных базовых классов, с возможным переопределением или добавлением новых данных и функций. Это создает иерархию классов.

3. ПОЛИМОРФИЗМ – единообразное обращение к одноименным функциям родственных объектов в тексте программы, при сохранении уникальности поведения объектов при вызове этих функций (слово полиморфизм происходит от греческого "имеющий много форм").

### *Понятие класса*

Класс является основой для создания объектов. В классе определяются данные и код, который работает с этими данными. Объекты являются экземплярами класса. Непосредственно инициализация переменных в объекте (переменных экземпляра) происходит в конструкторе. В классе могут быть определены несколько конструкторов. Важно понимать разницу между классом и объектом: класс является абстракцией до тех пор, пока не будет создан объект и появится физическая реализация этого класса в памяти компьютера.

Методы и переменные, составляющие класс, называются членами класса.

### *Синтаксис определения класса*

При определении класса объявляются данные, которые он содержит, и код, работающий с этими данными. Самые простые классы могут содержать только код или только данные.

Данные содержатся в переменных экземпляра, которые определены классом, а код содержится в методах. Важно с самого начала отметить, что в C# определены несколько специфических разновидностей членов класса. Это — переменные экземпляра, статические переменные, константы, методы, конструкторы, деструкторы, индексаторы, события, операторы и свойства.

При создании (определении) класса вначале указывается ключевое слово **class**. Ниже представлен общий синтаксис определения класса, содержащий только переменные экземпляра и методы.

```
class <имя>
{
//   объявление переменных экземпляра
<доступ> <тип> <имя переменной1>;
<доступ> <тип> <имя переменной2>;
…
<доступ> <тип> <имя переменнойN>;
// объявление методов
<доступ> <тип_возвр> <имя метода1>(<список параметров>)
{
 // тело метода
}
<доступ> <тип_возвр> <имя метода2>(<список параметров>)
{
 // тело метода
}
// …

<доступ> <тип_возвр> <имя методаM>(<список параметров>)
{
 // тело метода
}
}
```

### *Управление доступом к компонентам класса*

Для указания уровня доступа при объявлении метода используются специальные модификаторы.

**private**: доступ к следующим за данным словом компонентам может быть осуществлен только через объявленные в этом классе компонентные функции.

**protected**: доступ к следующим за данным словом компонентам может быть осуществлен только через компонентные функции данного класса и через компонентные функции класса, порожденного данным.

**public**: модификатор общедоступности компонента. Доступ к следующим за данным словом компонентам может быть осуществлен отовсюду в сфере действия данного класса.

**internal**: компонент доступен внутри сборки (файла), в котором он определен.

### *Графический пример*

При работе с графикой естественным началом был бы класс, который моделирует физические пиксели экрана. Создадим класс с именем Point, который объединяет координаты X и Y точки как компоненты единого типа данных – модели точки:

```
class  Point
{
    int X;
    int Y;
    bool Visible;
}
```

С помощью ключевого слова class создается новый тип данных. В нашем случае новый тип данных называется Point. Это имя используется для объявления объектов типа Point. Помните, что объявление класса с помощью ключевого слова class даст только описание типа, но физический объект при этом не создается. Поэтому выполнение приведенного выше кода не приведет к возникновению объектов типа Point.

Для того чтобы создать объект класса, необходимо использовать оператор **new**. Например:

```
Point  p1;
p1 =   new  Point();   // Создание   объекта
```

После выполнения этого оператора будет создан экземпляр класса Point — объект p1.

При создании экземпляра класса создастся объект, который содержит собственную копию каждой переменной экземпляра, определенной классом.

Следовательно, каждый объект класса Point будет содержать свои собственные копии переменных экземпляра X и Y.

Для доступа к этим переменным используется оператор точка (**.**). Этот оператор связывает имя объекта с именем члена класса. Приведем общий синтаксис данного оператора:

**object. member**

Имя объекта указывается слева от точки, а имя члена класса — справа. Оператор точка может использоваться для получения доступа, как к переменным экземпляра, так и к методам.

При выполнении оператора new создается физический объект, а переменной p1 присваивается ссылка на этой объект. Следовательно, только после выполнения второй строки кода переменная p1 будет ссылаться на объект типа Point.

Оператор new динамически (во время работы программы) выделяет память для объекта и возвращает ссылку на эту область памяти. В дальнейшем эта ссылка хранится в переменной. Следовательно, всем объектам классов в C# память должна выделяться динамически.

В первой строке кода переменная p1 объявляется как ссылка на объект типа Point. Т.е. переменная p1 может ссылаться на объект, но не является самим объектом. На этом этапе данная переменная хранит значение null, означающее, что связь между ней и объектом отсутствует. В следующей строке кода создается новый объект класса Point, а переменной p1 присваивается ссылка на этот объект, теперь переменная p1 связана с объектом.

Поскольку доступ к объектам класса осуществляется с помощью ссылки, классы иначе называют ссылочными типами. Ключевое отличие между обычными и ссылочными типами состоит в значениях, хранимых переменными каждого типа. При выполнении операции присваивания переменные ссылочного типа действуют иначе, чем переменные обычного типа (например, типа int). Когда значение одной переменной обычного типа присваиваете другой переменной обычного типа, то переменная, находящаяся слева от оператора присваивания, получает копию значения переменной, указанной справа. Когда же значение одной переменной ссылочного типа присваиваете другой переменной, ситуация гораздо сложнее, поскольку вы присваиваете переменной ссылку на другой объект.

Например, рассмотрим следующий фрагмент кода:

```
Point p1 =   new   Point ();
Point p2  = pl;
```

Может показаться, что переменные p1 и p2 относятся к различным объектам, но это не так. На самом деле обе переменные pl и p2 ссылаются на один и тот же объект. При присваивании переменной p2 значения переменной p1, переменной p2 присваивается ссылка на тот же объект, на который ссылается переменная pl. Следовательно, доступ к объекту можно осуществлять как с помощью переменной pl, так и с помощью переменной p2.

### *Методы*

Методы – это подпрограммы, которые управляют данными, определенными в классе, и во многих случаях обеспечивают доступ к

данным. Обычно остальные части программы взаимодействуют с классом при помощи его методов. Метод содержит один и более операторов. В профессионально написанном коде каждый метод выполняет только одну задачу. В качестве имени метода может использоваться любой действительный идентификатор. Ключевые слова не могут быть именами методов, имя Main() является зарезервированным для метода, с которого начинается выполнение программы.

В тексте программы методы обозначаются следующим обратом: после имени метода следует пара круглых скобок. Например, если имя метода get, то при его вызове в программе он будет написан как **get()**. Такая форма написания применяется для того, чтобы в программах можно было различать имена переменных и методов.

Общий синтаксис метода выглядит следующим образом:

```
<доступ>  <возвр_тип>  <имя метода>(<список параметров>)
{
  // тело метода
}
```

<**доступ**> — это модификатор, который указывает, какие части программы могут иметь доступ к методу. Модификатор не является обязательным, если он отсутствует, то метод доступен только в пределах класса, в котором он объявлен (**private**). Методы, объявленные как **public**, могут быть вызваны кодом из любого места программы.

<**возвр_тип**> в синтаксисе указывает тип данных, возвращаемых методом. Это могут быть как данные любого стандартного типа, так и объекты любого, созданного класса. Если метод не возвращает никакого значения, он должен быть указан как имеющий тип **void**.

<**имя метода**> – это может быть любой действительный идентификатор, не повторяющий имена, которыми были названы другие члены класса в той же области видимости.

После имени метода следует (<**список параметров**>) — последовательность разделенных занятыми пар «**тип–идентификатор**». Каждый параметр состоит из имени типа параметра и имени, под которым он будет доступен внутри метода. То есть с помощью параметров методу передаются из программы значения, тип которых обязательно должен быть указан. Параметрами в основном являются переменные, которые принимают значения аргументов, передаваемых методу при вызове. Если метод не имеет параметров, то список параметров будет пустым.

Управление возвращается из метода в двух случаях. Это происходит, во-первых, когда встречается закрывающая фигурная скобка этого метода, во-вторых, когда выполняется оператор return. Существует две формы оператора return, одна форма используется в методах, имеющих тип void (которые не возвращают значения), а вторая — в методах, возвращающих

значения. Методы возвращают значение вызывающей подпрограмме, используя следующую форму оператора return: **return value;**

### *Вызов методов*

Для вызова (активизации) метода необходимо указать имя объекта, для которого вызывается метод, затем имя метода и список аргументов в скобках. Если функция возвращает результат, его можно сохранить в переменной, использовать в качестве аргумента другой функции или отбросить.

При вызове статического метода необходимо использовать ИМЯ типа класса этого метода, а не имя экземпляра класса. Отметим, что здесь синтаксис C# отличается от синтаксиса C++ и Java: эти языки позволят также вызвать статические члены путем указания имени объекта. C# не разрешает использовать такой синтаксис. Разумеется, если доступ к методам или полям класса осуществляется изнутри класса, можно указывать имя члена напрямую.

**Примечание 1**. Имя объекта не требуется указывать, если вы вызываете его из того же класса.

**Примечание 2**. Нельзя получить доступ к полям экземпляра из метода, объявленного как static.

### *Аргументы методов*

Аргументы могут быть переданы в методы по ссылке или по значению. Переменная, передаваемая в метод по ссылке, подвергается любым изменениям, которые производит с ней вызываемый метод, в то время как переменная, передаваемая по значению, не меняет свое значение в результате изменений, сделанных внутри метода. Это происходит потому, что в первом случае метод получает ссылку на саму переменную, а во втором случае он получает ее копию.

В C# все параметры передаются по значению, если только специально не указывается иное. Однако тип данных параметра определяет фактическое поведение передаваемых в метод параметров. Так как типы по ссылке хранят только ссылку на объект, они передадут в метод эту ссылку. Напротив, типы данных по значению содержат действительные данные, поэтому в метод будет передана копия самих данных. Например, int передается в метод по значению, и любые изменения, которые метод производит со значением этого int, не изменяют значение оригинального объекта int. Если же в метод передается массив или любой другой тип по ссылке, например класс, и метод изменяет значение в массиве, то новое значение записывается в оригинальный массив.

Мы рассмотрели поведение по умолчанию. Однако можно сделать так, чтобы переменные по значению передавались по ссылке. Для этого используется ключевое слово **ref**.

```
void  f(ref int i){…}
```

Если в метод передается параметр, а аргумент метода помечен как ref, то любое изменение переменной, сделанное методом, вызовет соответствующее изменение оригинальной переменной. Ключевое слово ref должно указываться и при вызове метода.

В языках С-стиля общей для функций является способность возвращать более одного значения. Это достигается с помощью выходных параметров: выходные значения присваиваются переменным, переданным в метод по ссылке. Обычно начальные значения передаваемых по ссылке переменных не важны. Эти значения будут затерты функцией, которая, возможно, даже ни разу не использует их. Было бы удобно применить то же самое в C#, но C# требует присвоения переменной начального значения перед ее использованием. Для того, чтобы обойти требование компилятора C#, касающееся инициализации передаваемых переменных, необходимо использовать следующий механизм. Если аргумент метода предварен ключевым словом **out**, то в этот метод может быть передана переменная, которая не была инициализирована начальным значением. Переменная передается по ссылке, так что все изменения этой переменной, произведенные методом, останутся после передачи управления из вызываемого метода. Ключевое слово out должно указываться и при определении метода, и при его вызове.

### *Конструкторы*

Конструктор класса инициализирует объект при его создании. Он имеет то же имя, что и его класс и синтаксически похож на метод. Однако, в конструкторах тип возвращаемого значения не указывается явно. Общий синтаксис конструктора:

```
class-name()
{
        // код  конструктора
}
```

Как правило, конструкторы используются для присваивания начальных значений переменным экземпляра, определенным в классе, или для выполнения любых других процедур инициализации, необходимых для создания полностью сформированного объекта.

Все классы имеют конструкторы независимо от того, определен он или нет. Но по умолчанию в C# предусмотрено наличие конструктора, который присваивает нулевые значения всем переменным экземпляра (для переменных обычных типов) и значения null (для переменных ссылочного типа). Но если конструктор явно определен в классе, то конструктор по умолчанию использоваться не будет.

**Пример**.

```
class  Point
{
        int X;
        int Y;
        public  Point (int X, int Y)
        {       this.X=X;
                this.Y=Y;
        }
        bool Visible;
        public void MoveTo(int newX, int newY)
        {
                X=newX;
                X=newY;
        }
}
```

В приведенном примере конструктор класса Point присваивает переменным класса X и Y значения принимаемых параметров X и Y.

### *Оператор new*

Оператор new используется для создания объекта любого класса. Он имеет следующий синтаксис:

```
class-var =   new   class-name ();
```

Здесь словосочетание class-var — создаваемая переменная, тип которой (class) указывается перед именем переменной, а словосочетание class-name — это имя класса, экземпляр которого будет создан.

Имя класса, за которым следует пара круглых скобок, является конструктором класса. Если класс не имеет своего конструктора, оператор new будет использовать, конструктор по умолчанию, предоставленный C#. Следовательно, всем переменным экземпляра будут присвоены нулевые значения.

Поскольку размер памяти ограничен, существует вероятность, что оператор new не сможет выделить для объекта нужное количество памяти. Если это произойдет, то возникнет исключительная ситуация выполнения программы.

Один из ключевых компонентов схемы динамического выделения памяти – механизм освобождения памяти, занимаемой неиспользуемыми объектами, для того чтобы ее можно было выделить под вновь создаваемые объекты. Во многих языках программирования освобождение ранее выделенной памяти осуществляется вручную (например, в C++ для

освобождения памяти используется оператор delete). C# располагает для этого более эффективным механизмом, называемым «сборкой мусора».

Когда в C# программе отсутствуют обращения к объекту, то он рассматривается как не использующийся более, и система автоматически без участия программиста освобождает занимаемую им память, которая в дальнейшем может быть выделена под новые объекты.

«Сборка мусора» периодически осуществляется в ходе всей программы, причем для начала этой операции должны выполняться два условия: во-первых, наличие объектов, которые можно удалить из памяти, а во-вторых, необходимость такой операции. Поскольку процесс «сборки мусора» занимает некоторое время, система исполнения программы осуществляет ее только в случае необходимости, причем программист не может точно определить момент, когда это произойдет.

### *Деструкторы*

В C# предусмотрена возможность создания метода, который вызывается непосредственно перед удалением объекта из памяти при помощи операции «сборки мусора». Этот метод называется деструктором а может использоваться для гарантии корректного удаления объекта из памяти. Например, с помощью деструктора можно удостовериться, что файл, к которому имело место обращение из данного объекта, будет закрыт. Деструктор имеет следующий синтаксис:

```
~class-name()
{
// код деструктора
}
```

Здесь словосочетание class-name — это имя класса. То есть деструктор объявляется так же, как конструктор, за исключением того, что В деструкторе перед именем класса используется символ тильда (~). Отметим, что при определении деструктора не указывается тип возвращаемого значения.

Для добавления деструктора к классу нужно просто добавить его определение к коду этого класса, то есть деструктор является обычным членом класса. Внутри деструктора назначаются действия, которые должны быть выполнены перед возвращением памяти, выделенной для этого объекта. Деструктор вызывается во всех случаях, когда объект его класса должен быть удален из памяти.

Важно понимать, что деструктор вызывается не после, но и не перед выполнением операции «сборки мусора». (Он не вызывается, например, когда объект класса выходит из области видимости. Этим деструкторы в C# отличаются от деструкторов в C++, где они вызываются, когда объект выходит за пределы области видимости, в которой этот объект был создан.)

Это означает, что невозможно точно определить, когда будет выполняться код деструктора.

### *Ключевое слово this*

При вызове метода ему автоматически передается неявный аргумент, который является ссылкой на вызывающий объект (то есть на объект, с данными которого будет работать метод). Эта ссылка доступна через ключевое слово **this**.

Слово this часто используют в определениях пользовательских конструкторов для исключения конфликта между именами принимаемых параметров и именами внутренних членов класса (см. выше подраздел «Конструкторы»).

### *Основы наследования*

В C# при объявлении наследующего класса указывается имя наследуемого класса.

**Пример**.

```
class Figure
   {
    protected int X;
    protected int Y;
    protected bool Visible;
    public Figure(int X, int Y)
       {   this.X=X;
           this.Y=Y;
           Visible=false;
        }
   public int   GetX() {return X;}
   public int  GetY()  {return Y;}

  }

class Point : Figure
   {
     public Point(int X, int Y) : base(X, Y)    { }
      public void MoveTo(int newX, int newY)
     { X=newX;
       X=newY;
      }
   }
```

В приведенном примере ключевое слово **base** предназначено для указания компилятору того, что должен быть вызван метод базового класса.

В C# можно также создавать иерархию, содержащую любое количество уровней наследования.

### *Виртуальные методы*

Метод, при определении которого в наследуемом классе было указано ключевое слово **virtual** называется виртуальным методом. Следовательно, каждый наследующий класс может иметь собственную версию виртуального метода. В C# выбор версии виртуального метола, которую требуйся вызвать, осуществляется в соответствии с типом объекта, на который ссылается ссылочная переменная. Этот выбор (определение) осуществляется во время выполнения программы. Ссылочная переменная может ссылаться на различные типы объектов, следовательно, могут быть вызваны различные версии виртуальных методов, другими словами, именно тип объекта, на который указывает ссылка (а не тип ссылочной переменной), определяет вызываемую версию виртуального метода. Таким образом, если класс содержит виртуальный метод и от этого класса были наследованы другие классы, и которых определены свои версии метода, при ссылке переменной типа наследуемого класса на различные типы объектов вызываются различные версии виртуального метода. При определении виртуальною метода в составе наследуемого класса перед типом возвращаемого значения указывается ключевое слово virtual, а при переопределении виртуального метода в наследующем классе используется модификатор **override**. Процесс определения виртуального метода внутри наследуемого класса, при котором частично или полностью изменяется тело метода, а имя, параметры и их типы остаются прежними, называется переопределением метода. Виртуальный метод не может быть определен с модификатором static или abstract.

Переопределение метода положено в основу концепции динамического выбора вызываемого метода. Это механизм, с помощью которого выбор вызываемого переопределенного метода осуществляется во время выполнения программы, а не во время компиляции.

Переопределять виртуальный метод не обязательно. Если наследующий класс не предоставляет собственную версию виртуального метода, то используется метод наследуемого класса.

Виртуальные методы обеспечивают в C# поддержку полиморфизма во время выполнения программы. Полиморфизм очень важная составляющая объектно-ориентированного программирования, позволяющая определять в наследуемом классе методы, которые будут общими для всех наследующих классов, при этом наследующий класс может определять специфическую реализацию некоторых или всех этих методов. Переопределение методов представляет еще один способ реализации в C# принципа полиморфизма «один интерфейс, несколько методов».

Чтобы успешно использовать полиморфизм, вы должны понимать, что наследующий и наследуемый классы формируют иерархию, в которой осуществляется переход от меньшей специализации к большей. В наследуемом классе определены члены класса, вторые могут непосредственно использоваться наследующим классом, и методы, вторые в наследующем классе могут быть либо переопределены, либо оставлены без изменения. В результате наследующий класс получает определенную свободу при определении собственных версий методов, сохраняя при этом совместимый интерфейс, то есть возможность обращения к версии метода с использованием одного и то имени и одной и той же ссылочной переменной.

**Пример**.

```
class Figure
   {
    protected int X;
    protected int Y;
    protected bool Visible;
    public Figure(int X, int Y)
       {   this.X=X;
           this.Y=Y;
           Visible=false;
        }
   public int   GetX() {return X;}
   public int  GetY()  {return Y;}
  public int  GetVis()  {return Visible;}
   public virtual  void Show()
   { }
    public virtual void Hide()
     {}
  public  void MoveTo(int NewX, int NewY);
 {
  Hide();
  X = NewX;
  Y = NewY;
  Show();
 }
}

class Circle : Figure
{
  int Radius;
 public  Circle(int X, int Y, int Radius ):base(X,Y)
{this.Radius = Radius;}
```

```
public override void Show()
{   Visible = true;
   DrawCircle(X, Y, Radius, Color.Black);        // рисование окружности черным цветом
}
public override void Hide()
 {   Visible = false;
     DrawCircle(X, Y, Radius, Color.White);       // рисование окружности белым цветом

}
public  void Expand(int Delta )
{
  Hide();                    // удалить старый круг
   Radius += Delta;           // увеличить радиус
   if (Radius < 0)             // радиус должен быть > 0
     Radius = 0;
   Show();                    // рисовать новый круг

}
}
```

### *Абстрактные классы*

Для более эффективного использования наследования в базовом классе можно определить абстрактные методы, которые не имеют тела метода. При объявлении абстрактного метода используется модификатор **abstract**, поэтому встретив в списке членов класса метод с этим модификатором, программист знает, что он обязан реализовать (определить) этот метод в наследующем классе. Абстрактный метод автоматически становится виртуальным, так что модификатор virtual при объявлении такого метода не нужен.

Класс, содержащий один или более методов, должен быть объявлен как абстрактный (то есть при его объявлении указывается модификатор **abstract**). Поскольку абстрактный класс не определен полностью, объекты этого класса создать невозможно и попытка создания объекта абстрактного класса с помощью ключевого слова new приведет к ошибке компиляции.

Когда класс наследует абстрактный класс, он должен реализовать все абстрактные методы наследуемого класса. Если этого не происходит, то наследующий класс также должен быть объявлен с модификатором abstract. Таким образом, атрибут abstract наследуется до тех пор, пока методы, а значит и сам класс, не будут полностью реализованы.

**Пример**.

```
abstract class Figure
   {
    protected int X;
    protected int Y;
    protected bool Visible;
    public Figure(int X, int Y)
       {   this.X=X;
           this.Y=Y;
           Visible=false;
        }
   public int   GetX() {return X;}
   public int  GetY()  {return Y;}
  public int  GetVis()  {return Visible;}
   public abstract  void Show();
   public abstract  void Hide();
  public  void MoveTo(int NewX, int NewY);
 {
  Hide();
  X = NewX;
  Y = NewY;
  Show();
 }
}
```

### *Объектно-ориентированное проектирование*

Объектно-ориентированное проектирование – методология создания программного обеспечения на основе объектно-ориентированного подхода. Наиболее полно объектно-ориентированное проектирование реализует свой потенциал при разработке крупных программных комплексов, создаваемых коллективами программистов. Проектирование системы в целом, создание отдельных компонентов и их объединение в конечный продукт при этом часто выполняется разными людьми.

Объектно-ориентированное проектирование основывается на описании структуры и поведения проектируемой системы. При этом требуется найти ответы на два основных вопроса:

- Из каких частей состоит система.
- В чём состоит ответственность каждой из частей.

Выделение частей производится таким образом, чтобы каждая имела минимальный по объёму и точно определённый набор выполняемых функций, и при этом взаимодействовала с другими частями как можно меньше.

По мере детализации описания и определения ответственности выявляются данные, которые необходимо хранить, наличие близких по поведению объектов, которые становятся кандидатами на реализацию в виде классов с общими предками. После выделения компонентов и определения интерфейсов между ними реализация каждого компонента может проводиться практически независимо от остальных.

Большое значение имеет правильное построение иерархии классов. Одна из известных проблем больших систем, построенных по технологии объектно-ориентированного проектирования — так называемая проблема хрупкости базового класса. Она состоит в том, что на поздних этапах разработки, когда иерархия классов построена и на её основе разработано большое количество кода, оказывается трудно или даже невозможно внести какие-либо изменения в код базовых классов иерархии.

Объектно-ориентированный подход помогает справиться с такими сложными проблемами, как

- ♦ уменьшение сложности программного обеспечения;
- ♦ повышение надежности программного обеспечения;
- ♦ обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;
- ♦ обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

В объектно-ориентированном проектировании выделяют три основных этапа процесса создания прикладных программ: объектно-ориентированный анализ (Object-oriented analysis – OOA), объектно-ориентированное проектирование (Object-oriented design – OOD) и объектно-ориентированное программирование (Object-oriented programming – OOP).

**OOA** – методология создания моделей реальности и решаемых задач с использованием объектного подхода, при котором элементами моделей являются классы и объекты, составляющие словарь предметной области.

**OOD** – методология проектирования, реализующая объектную декомпозицию задачи (с определением состава и поведения объектов) и представление статической и динамической моделей проектируемой системы.

**OOP** – процесс реализации объектно-ориентированного проекта на каком-либо языке программирования.

На результатах OOA формируются модели, которые использует OOD для проектирования классов, создавая основу для окончательного решения задачи средствами OOP (на конкретном языке программирования, с учетом его инструментария).

Основной прием OOA – абстрагирование. Абстракция – это набор существенных характеристик некоторого объекта, которые отличают его от

других видов объектов и объединяют с объектами того же вида. Абстракция определяет специфику рассматриваемого объекта для его дальнейшего использования в проекте. Абстракции должны содержать самые существенные свойства объекта данного вида и игнорировать его случайные или второстепенные для заданной цели свойства. Для любого предмета возможно множество различных абстракций – все зависит от цели рассмотрения. Все человеческие слова и языки являются абстракциями - неполными описаниями мира.

В начале процесса анализа формируется набор объектов задачи. Объект обладает состоянием, проявляет четко выраженное поведение, индивидуальность и свойства. Структура и поведение схожих объектов определяют общий для них класс.

Поведение объекта определяется последовательностью совершаемых над ним действий. Действия приводят к изменению состояния объекта. Описывая состояние объекта, нужно перечислить все возможные свойства объекта с текущими значениями каждого из этих свойств. Эти значения могут быть простыми количественными характеристиками, а могут означать другой объект.

Объекты, как правило, не существуют изолировано. Они либо подвергаются воздействию, либо сами воздействуют на другие объекты. Взаимодействия между объектами определяют поведение объектов.

Абстракции обладают статическими и динамическими свойствами (содержание и поведение). Построение абстракций – до конца не формализованный процесс, опирающийся на ряд принципов. Абстракции образуют иерархии, по типам (отношения наследования) и объектам (отношения взаимодействия). Из абстракций формируется "словарь предметной области задачи", еще называемый "ключевыми абстракциями".

Завершается стадия анализа созданием модели системы. Моделью системы (или какого-либо другого объекта или явления) мы называем формальное описание системы, в котором выделены основные объекты, составляющие систему, и отношения между этими объектами.

Определение ключевых абстракций предметной области может рассматриваться и как часть анализа, так и как часть проектирования. Однако, цели этих этапов различны. При анализе моделируется окружающий мир, идентифицируя классы и объекты, составляющие словарь предметной области. При проектировании формируются абстракции и механизмы, обеспечивающие поведение, которое требует эта модель. Анализ определяем поведение создаваемой системы, а при проектировании создаем чертежи этой системы. Результаты анализа могут непосредственно использоваться на этапе проектирования.

На этапе OOD на базе абстракций OOA проектируются классы. Класс – определение набора характеристик множества объектов, связанных общим содержанием и поведением. Любой объект является экземпляром некоторого

класса. В составе класса обычно разделяют интерфейс и внутреннюю реализацию. Интерфейс обеспечивает внешнее поведение класса, реализация – внутреннюю организацию. Такое разделение, с защитой от доступа извне к внутренней реализации класса называется инкапсуляцией. Никакая часть сложной системы не должна находиться в зависимости от подробностей внутреннего устройства других частей системы. Ограничения доступа позволяют вносить в программу изменения, сохранять ее надежность и минимизировать затраты на этот процесс. Внутреннее строение (реализация) классов и объектов разрабатывается только после завершения проектирования их внешнего облика.

Набору абстракций разных уровней в соответствие ставится иерархия классов (наследование). Иерархия – это упорядоченная система абстракций. Основными видами иерархических структур применительно к сложным системам являются структура классов – иерархия по номенклатуре и структура объектов – иерархия по составу.

Иерархическая организация делает обозримыми и управляемыми отдельные классы, входящие в иерархию, т. е. сложность в отдельной точке ограничивается, за счет разнесения ее по уровням и инкапсуляции. Проектирование классов является итерационным процессом.

Качество проектируемых классов характеризуется рядом критериев, к соответствию которым классов проекта должен стремиться его разработчик:

1. Взаимозависимость – должна быть максимальной для связанных наследованием классов и минимальной для несвязанных. Означает влияние свойств одного объекта на свойства других.

2. Связность – степень взаимодействия элементов. Должна быть максимальной внутри класса и минимальной между классами.

3. Достаточность – наличие в классе всего необходимого для выполнения его функций.

4. Полнота – реализация в классе всех его потенциальных функциональных возможностей (важна мера).

5. Простота – означает безызбыточность, при которой все, что не необходимо для реализации, или уже реализовано иначе из класса исключается.

Процесс проектирования предполагает наличие системы обозначений, обеспечивающей документирование процесса проектирования и представляющей результаты проектирования. Для объектно-ориентированного проектирования в последнее десятилетие общепризнанным инструментом его документирования стал язык **UML**.

История развития языка UML берет начало с октября 1994 года, когда Гради Буч и Джеймс Румбах из компании Rational Software Corporation начали работу по унификации известных методов объектно-ориентированного проектирования Booch и OMT. Несмотря на то, что сами по себе эти методы были достаточно популярны, совместная работа была

направлена на изучение всех известных объектно-ориентированных методов с целью объединения их достоинств. При этом Г. Буч и Дж. Румбах сосредоточили усилия на полной унификации результатов своей работы. Проект так называемого унифицированного метода (Unified Method) был подготовлен и опубликован в октябре 1995 года. Осенью того же года к ним присоединился А. Джекобсон, главный технолог компании Objectory AB (Швеция), с целью интеграции своего метода OOSE с двумя предыдущими. В этот период поддержка разработки языка UML становится одной из целей консорциума OMG (Object Management Group), который был образован еще в 1989 году с целью разработки предложений по стандартизации объектных и компонентных технологий CORBA. Усилия группы разработчиков, в которую входили также Г. Буч, Дж. Румбах и А. Джекобсон, привели к появлению первых документов, содержащих описание языка UML (октябрь 1996 г.). Тогда же некоторые компании и организации увидели в языке UML стратегический интерес для своего бизнеса. Компания Rational Software вместе с несколькими организациями, изъявившими желание выделить ресурсы для разработки строгого определения версии 1.0 языка UML, учредила консорциум партнеров UML, в который первоначально вошли такие фирмы, как Digital Equipment Corp., HP, i-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI и Unisys. Эти компании обеспечили поддержку последующей работы по более точному определению нотации. В настоящее время язык UML получил всеобщее признание и поддерживается всеми современными средствами автоматизации объектно-ориентированного проектирования (IBM Rational Rose, Borland Together, Microsoft Visio и др.).

Язык UML предоставляет разработчикам объектно-ориентированных систем единую систему обозначений элементов объектно-ориентированных моделей и графических диаграмм. Диаграмма в UML – это графическое представление набора элементов, изображаемое чаще всего в виде связанного графа с вершинами (сущностями) и ребрами (отношениями).

В UML для документирования проекта используются девять типов основных диаграмм, а именно:

- диаграммы классов;
- диаграммы объектов;
- диаграммы прецедентов;
- диаграммы последовательностей;
- диаграммы кооперации;
- диаграммы состояний;
- диаграммы действий;
- диаграммы компонентов;
- диаграммы развертывания.

На **диаграмме классов** показывают классы, интерфейсы, объекты и кооперации, а также их отношения. При моделировании объектно-ориентированных систем этот тип диаграмм используют чаще всего. Диаграммы классов соответствуют статическому виду системы с точки зрения проектирования. Диаграммы классов, которые включают активные классы, соответствуют статическому виду системы с точки зрения процессов.

На **диаграмме объектов** представлены объекты и отношения между ними. Они являются статическими "фотографиями" экземпляров сущностей, показанных на диаграммах классов. Диаграммы объектов, как и диаграммы классов, относятся к статическому виду системы с точки зрения проектирования или процессов, но с расчетом на настоящую или макетную реализацию.

На **диаграмме прецедентов** представлены прецеденты и актеры (частный случай классов), а также отношения между ними. Диаграммы прецедентов относятся к статическому виду системы с точки зрения прецедентов использования. Они особенно важны при организации и моделировании поведения системы.

**Диаграммы последовательностей** и кооперации являются частными случаями диаграмм взаимодействия. На диаграммах взаимодействия представлены связи между объектами; показаны, в частности, сообщения, которыми объекты могут обмениваться. Диаграммы взаимодействия относятся к динамическому виду системы. При этом диаграммы последовательности отражают временную упорядоченность сообщений, а диаграммы кооперации - структурную организацию обменивающихся сообщениями объектов. Эти диаграммы являются изоморфными, то есть могут быть преобразованы друг в друга.

На **диаграммах состояний** представлен автомат, включающий в себя состояния, переходы, события и виды действий. Диаграммы состояний относятся к динамическому виду системы; особенно они важны при моделировании поведения интерфейса, класса или кооперации. Они акцентируют внимание на поведении объекта, зависящем от последовательности событий, что очень полезно для моделирования реактивных систем.

**Диаграмма деятельности** – это частный случай диаграммы состояний; на ней представлены переходы потока управления от одной деятельности к другой внутри системы. Диаграммы деятельности относятся к динамическому виду системы; они наиболее важны при моделировании ее функционирования и отражают поток управления между объектами.

На **диаграмме компонентов** представлена организация совокупности компонентов и существующие между ними зависимости. Диаграммы компонентов относятся к статическому виду системы с точки зрения реализации. Они могут быть соотнесены с диаграммами классов, так как

компонент обычно отображается на один или несколько классов, интерфейсов или кооперaций.

На **диаграмме развертывания** представлена конфигурация обрабатывающих узлов системы и размещенных в них компонентов. Диаграммы развертывания относятся к статическому виду архитектуры системы с точки зрения развертывания. Они связаны с диаграммами компонентов, поскольку в узле обычно размещаются один или несколько компонентов.

**Примеры диаграмм UML приведены в приложении Б данного пособия.**

На рисунке 1 показаны отношения между различными видами диаграмм UML. Указатели стрелок можно интерпретировать как отношение "является источником входных данных для …" (например, диаграмма прецедентов является источником данных для диаграмм видов деятельности и последовательности). Приведенная схема является наглядной иллюстрацией итеративного характера разработки моделей с использованием UML.

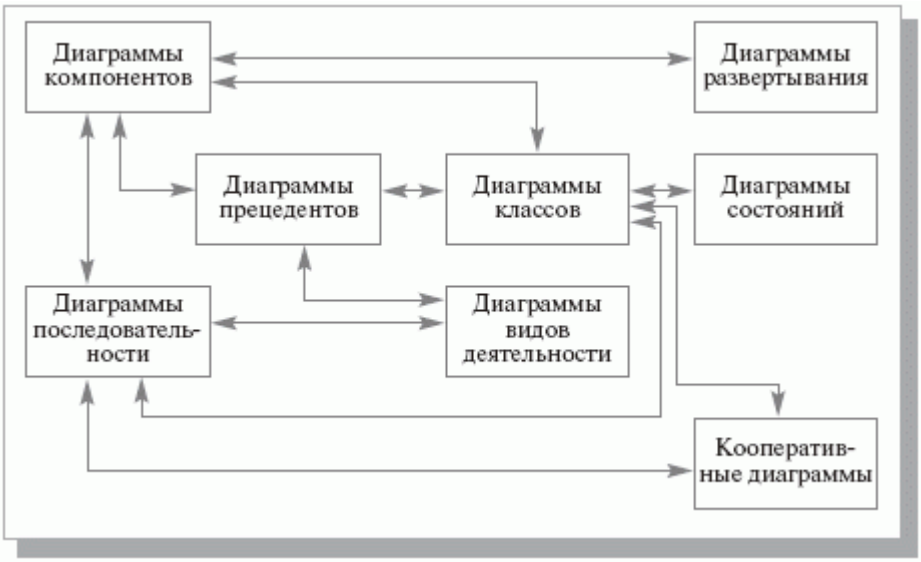


Рис. 1. Взаимосвязи между диаграммами UML

При графическом изображении диаграмм следует придерживаться следующих основных рекомендаций:

- Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Речь идет о том, что в процессе разработки диаграммы необходимо учесть все сущности, важные с точки зрения контекста данной модели и диаграммы. Отсутствие тех или иных элементов на диаграмме служит признаком неполноты модели и может потребовать ее последующей доработки.

- Все сущности на диаграмме модели должны быть одного уровня представления. Здесь имеется в виду согласованность не только имен одинаковых элементов, но и возможность вложения отдельных диаграмм друг в друга для достижения полноты представлений. В случае достаточно сложных моделей систем желательно придерживаться стратегии последовательного уточнения или детализации отдельных диаграмм.
- Вся информация о сущностях должна быть явно представлена на диаграммах. В языке UML при отсутствии некоторых символов на диаграмме могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), тем не менее, необходимо стремиться к явному указанию свойств всех элементов диаграмм.
- Диаграммы не должны содержать противоречивой информации. Противоречивость модели может служить причиной серьезных проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегирования или композиции приводит к ошибкам в программном коде, который будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами и различными атрибутами свойств в одном пространстве имен также приводит к неоднозначной интерпретации и может быть источником проблем.
- Каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов. Любые пояснительные тексты, которые не являются собственными элементами диаграммы (например, комментариями), не должны приниматься во внимание разработчиками. В то же время общие фрагменты диаграмм могут уточняться или детализироваться на других диаграммах этого же типа, образуя вложенные или подчиненные диаграммы. Таким образом, модель системы на языке UML представляет собой пакет иерархически вложенных диаграмм, детализация которых должна быть достаточной для последующей генерации программного кода, реализующего проект соответствующей системы.
- Количество типов диаграмм для конкретной модели приложения строго не фиксировано. Для простых приложений нет необходимости строить все без исключения типы диаграмм. Некоторые из них могут просто отсутствовать в проекте системы, и это не будет считаться ошибкой разработчика. Например, модель системы может не содержать диаграмму развертывания для приложения, выполняемого локально на компьютере пользователя.

Важно понимать, что перечень диаграмм зависит от специфики конкретного проекта системы.

- Любая модель системы должна содержать только те элементы, которые определены в нотации языка UML. Имеется в виду требование начинать разработку проекта, используя только те конструкции, которые уже определены в метамодели UML. Как показывает практика, этих конструкций вполне достаточно для представления большинства типовых проектов программных систем. И только при отсутствии необходимых базовых элементов языка UML следует использовать механизмы их расширения для адекватного представления конкретной модели системы. Не допускается переопределение семантики тех элементов, которые отнесены к базовой нотации метамодели языка UML.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Объектно-ориентированный подход к программированию.
2. Инкапсуляция.
3. Полиморфизм.
4. Наследование.
5. Понятие класса.
6. Синтаксис определения класса, пример определения класса.
7. Методы класса. Синтаксис метода. Вызов методов. Аргументы методов.
8. Модификаторы доступа.
9. Конструкторы и деструкторы.
10. Оператор new.
11. Ключевое слово this.
12. Ключевое слово base.
13. Оператор точка.
14. Виртуальные методы.
15. Абстрактные методы и классы.
16. Ключевое слово ref.
17. Ключевое слово out.
18. «Сборка мусора».
19. Технология объектно-ориентированного проектирования
20. Язык UML

## КОНТРОЛЬНЫЕ РАБОТЫ

### Контрольная работа №1

**Задание**

Написать реферат (объёмом 10 печатных страниц формата А4) на тему: «**C# и объектно-ориентированное программирование**». Реферат должен содержать описание (с примерами на языке C#) основных понятий и принципов объектно-ориентированного программирования.

**Основные разделы реферата**:

- Классы.
- Инкапсуляция.
- Наследование.
- Полиморфизм.

### Контрольная работа №2

**Задание**

1. Письменно спроектировать иерархию классов различных геометрических фигур, которая содержит базовый класс и производные от него классы: прямоугольник, эллипс, линия. Базовый класс для иерархии геометрических фигур рекомендуется сделать абстрактным классом. Он должен содержать:
   - данные, которые хранят информацию о двух точках, задающих размеры блока геометрической фигуры, цвет фигуры;
   - конструктор класса, который получает эти точки в качестве аргументов;
   - абстрактные методы: метод рисования фигуры сплошной линией, метод рисования фигуры пунктирной линией, метод стирания фигуры.

Абстрактные методы не содержат тела метода. А в производных классах эти методы будут получать реализацию, соответствующую типу конкретной фигуры.

2. Построить диаграмму спроектированных классов (используя язык UML).

### Общие требования к оформлению контрольных работ

1) Контрольную работу требуется представить на рецензию преподавателю в печатной форме, на белой бумаге формата А4 (210х297мм).

2) Контрольная работа должна состоять из следующих компонентов:

- титульный лист (пример титульного листа приведен в приложении А данного пособия);
- задание;
- содержательную часть в соответствии с заданием;

- список использованной литературы.

3) Контрольная работа должна быть напечатана в текстовом редакторе (например, Microsoft Word) в соответствии со следующими требованиями:

- шрифт: Times New Roman;
- размер:14 пт;
- межстрочный интервал: 1,5;
- поля: верхнее – 2,0 см; нижнее – 2,0 см; левое – 2,5 см; правое – 1,5 см;
- нумерация страниц внизу, от центра.

4) Рисунки должны быть ориентированы по центру страницы и подписаны. Например: *Рис.1 Диаграмма классов фигур*

**Список рекомендуемой литературы**

1. Павловская Т.А. C#. Программирование на языке высокого уровня. – СПб.:Питер, 2007.

2. Троелсен Э. C# и платформа.NET. Библиотека программиста. – СПб.:Питер, 2007.

## ЛАБОРАТОРНАЯ РАБОТА №1
## “Создание Windows-приложений. Обработка событий мыши. Вывод текста в окно”

### *Задание*

1. Ознакомиться со средой проектирования Visual Studio.NET
2. Создать простейшее приложение средствами Studio.NET.
3. Изменить размеры окна приложения, цвет фона и заголовок.
4. Реализовать обработку щелчка левой и правой кнопки мыши:
   - По левой кнопке - выводить координаты курсора мыши в точке его нахождения.
   - По правой – отображать диалоговое окно с сообщением «**Нажата правая кнопка мыши**» и очищать окно приложения от надписей.

### *Краткая справка*

В состав среды проектирования Microsoft Visual Studio.NET встроены средства, облегчающие программисту разработку приложений. Данная среда позволяет быстро создавать шаблоны новых приложений. При этом программисту не приходится писать ни одной строчки кода. Достаточно ответить на ряд вопросов, касающихся того, какое приложение требуется создать, и исходные тексты шаблона приложения вместе с файлами будут готовы.

Генерируемый средой Visual Studio.NET каркас стандартного приложения Windows содержит ряд классов. Класс формы (**Form1**) предназначен для создания интерфейса и программирования функциональности приложения.

С формой можно работать в следующих режимах:

1) в режиме дизайна или конструктора;

2) в режиме кодирования.

В режиме дизайна доступно **Окно свойств** формы, которое активизируется через команду меню, вызываемого по щелчку правой кнопкой мыши в области формы.

Используя доступные свойства можно определить положение, размер, цвет и особенности управления для создаваемого окна.

- Свойство Text позволяет изменить заголовок окна.
- Свойства Size и DesktopLocation задают размер и положение окна при его отображении.
- Свойство ForeColor служит для указания цвета переднего плана по умолчанию для всех элементов управления, размещенных в форме.
- Свойства BorderStyle, MinimizeBox и MaximizeBox определяют, можно ли будет свернуть, развернуть или изменить размер формы во время исполнения программы.

Для каждого изменяемого свойства, создается соответствующий код, который помещается в файл **Form1.Designer.cs,** в секцию, отмеченную как «**Код, автоматически созданный конструктором форм Windows»**. Можно раскрыть ее и просмотреть сгенерированный код.

**Программы Windows, основанные на графическом интерфейсе пользователя, управляются событиями**. C# выполняет функции обработки сообщений с помощью специальных функций–делегатов. Помимо методов и свойств формы содержат коллекции обработчиков событий. Если требуется обрабатывать событие, то вы должны выбрать в панели свойств формы закладку «**Events**» и двойным щелчком по пустой ячейке, расположенной справа от имени интересующего события, сгенерировать шаблон функции-обработчика события.

**События формы и элементов управления**

Таблица 1

| Событие | Описание |
|---|---|
| Click | Возникает при щелчке мыши |
| DoubleClick | Возникает при двойном щелчке мыши |
| KeyDown | Возникает при нажатии клавиши |
| KeyUp | Возникает при отпускании клавиши |
| MouseDown | Происходит, когда нажимается кнопка мыши, а указатель мыши находится над объектом |
| MouseEnter | Возникает, когда указатель мыши попадает в область объекта |
| MouseLeave | Возникает, когда указатель мыши покидает область объекта |
| MouseMove | Возникает при перемещении мыши над объектом |
| MouseUp | Возникает при отпускании кнопки мыши в области объекта |
| Resize | Возникает при изменении размера объекта |

Обработчик события получает два аргумента – ссылку на объект, к которому относится событие, и дополнительную информацию о событии (тип **EventArgs** или производный от него).

Обработчики событий от мыши получают в качестве второго аргумента объект типа **MouseEventArgs**.

**Свойства, определенные в классе MouseEventArgs**

Таблица 2

| | |
|---|---|
| Button | Позволяет получить информацию о том, какая кнопка мыши нажата (в виде значений перечисления MouseButtons) |
| Clicks | Позволяет получить информацию о том, сколько раз нажата и отпущена кнопка мыши |
| Delta | Позволяет получить информацию (в виде положительного или отрицательного числового значения) о повороте колесика мыши |
| X | Позволяет получить координату X для указателя мыши во время щелчка |
| Y | То же самое для координаты Y |

Операции рисования и вывода текста инкапсулирует класс **System.Drawing.Graphics**. В нём присутствуют методы для отображения линий, кривых, строк и других графических элементов.

Для выполнения вывода на экран в методах класса формы требуется предварительно получить объект **Graphics**. Это обеспечивается добавлением в метод следующей строки:

*Graphics g = CreateGraphics();*

Текстовые строки в C# хранятся в объектах типа **string**.

Для вывода текста в окно необходимо объявить переменную этого типа, проинициализировать её и передать методу **DrawString** класса Graphics.

Например:

```
string s = "Hello, World!";
g.DrawString(s, new Font("Times New Roman", 8),
new SolidBrush(Color.Black), new Point(100, 200));
```

В операции вывода строки дополнительно указываются три объекта типов **Font, SolidBrush** и **Point**, задающие соответственно шрифт, используемый для вывода, цвет шрифта и координаты точки привязки выводимого текста.

Со строками типа string можно выполнять операцию сложения, задаваемую знаком "+".

Числовые типы языка C# можно переводить в строковое представление вызовом для них функции **ToString**.

Например:

*string s = e.X.ToString();*

### *Рекомендации по выполнению задания*

Пункт 2

Создайте проект типа **Windows Application**, выбрав пункт меню **Файл | Создать | Проект | Visual C# | Windows**. В соответствующих полях диалогового окна **Создать проект** введите имя проекта и укажите путь к папке, рекомендованной для сохранения ваших проектов. Нажмите кнопку "**OK**".

Пункт 3

В **Окне свойств** формы задать значения свойств **BackColor**, **Size**, **Text**. Следует присвоить этим полям значения, задающие соответственно белый цвет, размер 600x450 и имя, подходящее для разрабатываемого вами графического редактора, функции которого будут реализовываться в дальнейших лабораторных работах.

Пункт 4

Добавить к классу **Form1** обработчик события **MouseDown** в соответствии с информацией, изложенной в справке к работе.

Для различения нажатий на левую и правую кнопки мыши следует проверять значение поля **Button** аргумента **MouseEventArgs**. Если оно равно **MouseButtons.Left**, то была нажата левая кнопка, если **MouseButtons.Right** – правая.

Пример выражения проверки:

*if (e.Button == MouseButtons.Left)*

Вывод на экран строки с координатами курсора мыши в точку, где находится курсор, должен быть реализован в соответствии с информацией, изложенной в справке к работе.

Диалоговое окно с сообщением создаётся с помощью функции **MessageBox.Show**(), которой передаётся два аргумента: строка выводимого в окне текста и строка заголовка окна.

Для очистки окна следует вызвать функцию **Clear()** объекта типа Graphics, передав ей в качестве аргумента значение цвета **Color.White**.

## ЛАБОРАТОРНАЯ РАБОТА №2
## “Создание многооконного приложения, имеющего меню. Рисование прямоугольников под управлением мыши”

### *Задание*

1. Сделать окно приложения MDI-контейнером.
2. Создать меню приложения, содержащее на верхнем уровне пункт «**Окно**», а в распахивающемся списке команду «**Новое**» и список открытых окон. Реализовать обработку команды создания нового окна.
3. Реализовать рисование на экране прямоугольников под управлением мыши. При нажатии левой кнопки мыши и ее удержании при перемещении мыши потенциальный прямоугольник должен отображаться пунктиром, а при отпускании кнопки мыши прямоугольник должен выводиться сплошной линией. Должно рисоваться произвольное число прямоугольников.

### *Краткая справка*

**MDI** (Multiple-document interface) приложения позволяют отображать несколько документов одновременно. Такая организация приложения является типичной для редакторов документов различных форматов. При этом каждый документ будет отображаться в своем собственном окне. Обычно MDI приложения имеют в основном меню подпункты для переключения между окнами. Основным окном MDI приложения является родительская форма. Она может содержать несколько дочерних окон. Только одно из дочерних окон может быть активно в один момент времени.

Программы в среде Windows могут иметь основное меню. Меню приложения позволяет создавать иерархию вложенных друг в друга команд меню любой степени сложности. С пунктами меню связываются реализуемые в классе формы методы-обработчики команд меню. Меню могут создаваться и назначаться формам под управлением программного кода и в режиме дизайна формы с использованием интерактивного редактора меню.

Для рисования графических объектов в Windows приложениях платформа .NET использует библиотеку GDI+

Пространство имен **Drawing** содержит множество объектов, которые облегчают программисту работу графикой. GDI+ включает возможности рисования простейших объектов (линии, эллипсы...), рисование различных объектов 2D графики, отображение файлов различных графических форматов (bmp, jpeg, gif, wmf, ico, tiff...) и многое другое.

Большинство функций рисования являются методами класса **Graphics**. Для рисования нужно создавать объект типа Graphics вызовом функции **CreateGraphics().** По завершении использования объекта Graphics программа также должна вызвать его метод **Dispose**.

Для рисования линий и контуров фигур в GDI+ используется объект перо (тип **Pen**). При создании пера можно задать его цвет и ширину:

Например:

*Pen p = new Pen(Color.Black, 1);*

Прямоугольники рисует метод класса Graphics **DrawRectangle**. Ему передаются объекты перо и прямоугольник (**Pen** и **Rectangle**).

Прямоугольник можно создать, вызвав метод класса **Rectangle FromLTRB**, которому передаются четыре числа, задающие координаты левого верхнего и правого нижнего углов прямоугольника:

Например:

*Rectangle r = Rectangle.FromLTRB(x1, y1, x2, y2);*

### *Рекомендации по выполнению задания*

Пункт 1

Для создания MDI-приложения откройте проект, созданный в первой работе и установите для формы свойство **IsMdiContainer** в **true**, это будет определять форму как родительское окно MDI приложения. Задайте через свойства формы размер родительского окна **1000x700.**

Пункт 2

Для создания меню приложения Visual Studio .NET имеет в **Панели элементов** компонент **MenuStrip**. Добавьте на форму родительского окна компонент MenuStrip. В панели компонентов ниже основной формы приложения появится объект **MenuStrip1**. В верхней части формы появится проект меню с единственным полем «**Вводить здесь**». Поле является редактируемым. Добавьте в меню верхнего уровня пункт «**Окно**» и в его подменю пункт «**Новое**». Для свойства **MainMenuStrip** в панели свойств формы должно быть установлено значение **menuStrip1**.

Для реализации окон документов MDI-приложения требуется добавить к проекту соответствующую новую форму (**Form2**). Для этого введите команду «**Проект**» | «**Добавить форму**» | «**Windows Forms**». Для созданной формы документа задайте размеры (**800x600**) и белый цвет фона.

Укажите, что в меню **Окно** следует отображать список созданных окон. Для этого зайдите в панель свойств меню для свойства **MdiWindowListItem** выберите значение **окноToolStripMenuItem.**

Сгенерируйте обработчик команды меню «Окно»–«Новое» двойным щелчком по пункту «**Новое**» на странице дизайна формы. В тело обработчика поместите следующий код создания и отображения нового окна документа:

```
Form f = new Form2();
f.MdiParent = this;
f.Text = "Рисунок " + this.MdiChildren.Length.ToString();
f.Show();
```

Пункт 3

Для решения задачи по рисованию прямоугольников в классе **Form2** потребуется обрабатывать события мыши **MouseDown**, **MouseMove**, **MouseUp**. Добавьте соответствующие методы-обработчики.

Затем требуется добавить поля данных в класс Form2:

- для хранения координат рисуемого прямоугольника;
- объекта Graphics;
- флага нажатой кнопки мыши (признака процесса рисования).

При нажатии на левую кнопку следует:

- инициализировать объект Graphics;
- устанавливать флаг рисования;
- инициализировать начальные координаты прямоугольника.

При перемещении мыши следует:

- проверять флаг рисования,
- и если он установлен, то рисовать промежуточный пунктирный контур прямоугольника, предварительно стирая (цветом фона) контур, нарисованный в предыдущем вызове этого обработчика..

При отпускании левой кнопки мыши следует:

- проверять флаг рисования, и, если он установлен, то стирать последний промежуточный контур,
- рисовать окончательный контур прямоугольника обычным пером,
- сбрасывать флаг рисования.

Стиль пунктирного пера для промежуточного рисования прямоугольников задаётся следующим образом:

*pen.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;*

Белое перо для стирания промежуточных контуров может быть создано следующим образом:

*pen=new Pen(Color.White, 1);*

Следует учесть, что методу **DrawRectangle** должен передаваться **нормализованный прямоугольник, у которого значения координат первой точки всегда меньше значений соответствующих координат второй точки**. Поскольку мышь в процессе рисования может оказаться левее и выше точки нажатия на левую кнопку, её текущие координаты нельзя непосредственно использовать в качестве координат второй точки, задающей прямоугольник. Перед передачей четырёх координат в метод Rectangle.FromLTRB их следует предварительно упорядочивать, используя вспомогательные переменные, таким образом, чтобы в первой паре аргументов передавались всегда меньшие значения координат.

## ЛАБОРАТОРНАЯ РАБОТА №3
## “Проектирование иерархии классов. Использование контейнерных классов. Перерисовка графической информации”

### *Задание*

1. Спроектировать базовый класс для моделирования различных геометрических фигур.
2. Спроектировать класс, моделирующий прямоугольник, как производный от класса, спроектированного в пункте 1 задания.
3. Спроектировать класс, моделирующий эллипс, как производный от класса, спроектированного в пункте 1 задания.
4. Реализовать сохранение информации об изображении в динамическом массиве объектов, имеющих тип класса, спроектированного в пункте 1 задания.
5. Реализовать перерисовку графического изображения, создаваемого в программе.

### *Краткая справка.*

Классы являются шаблонами, на основе которых создаются объекты. Каждый объект содержит данные и методы для работы с этими данными. Класс определяет, какие данные может содержать каждый объект этого класса, но не содержит самих данных. При создании экземпляра объекта класса его поля заполняются конкретными данными.

Поля данных класса представляют собой любые переменные, связанные с классом. Если поля и функции объявлены как **public**, они доступны за пределами класса.

После того как будет создан экземпляр объекта, к полям класса можно осуществлять доступ с помощью синтаксиса **объект.имя_поля.**

Новые классы могут наследовать содержание (данные и методы) уже существующих классов. Такой новый класс называется **производным**. Класс, от которого он наследует, называется **базовым**. На основе наследования создаются иерархии классов. Проектирование иерархий классов – основа методологии объектно-ориентированного программирования.

**Абстрактные классы** содержат абстрактные методы – методы без реализации. Экземпляры таких классов не могут создаваться, но абстрактные классы часто используются в качестве базовых классов для иерархий классов, так как позволяют задать типовой набор функций, обязательный для реализации в производных классах. Такой абстрактный класс содержит обобщённую модель классов определённого назначения и ссылки на его тип могут использоваться в функциях, которым впоследствии будут передаваться экземпляры различных производных от него классов.

**Контейнерные классы** используются для хранения и обработки наборов объектов. Наиболее простой структурой данных такого рода

является обычный массив. **Массив** в C# является экземпляром класса **System.Array**. System.Array обеспечивает эффективный доступ к отдельному элементу по его индексу. Однако Array обладает недостатком, заключающимся в необходимости указания его размера при создании экземпляра класса. После его создания не существует возможности добавлять, вставлять или удалять элементы. .NET поддерживает множество других контейнерных классов, которые полезны в различных обстоятельствах.

За исключением System.Array, все классы структур данных располагаются в пространстве имён System.Collections. В .NET 2.0 добавилось пространство имён **System.Collections.Generic**, в котором находятся параметризованные реализации контейнерных классов. Таким классам при создании в качестве параметра, передаваемого в угловых скобках, указывается тип данных, который будет храниться в контейнере. В число этих классов входит класс **List**, реализующий **динамический массив** (массив переменного размера) для данных определённого типа. Если при инициализации такого массива указать в качестве параметра определённый класс, то массив можно будет использовать для хранения объектов этого класса и классов, производных от него.

Класс **List** может быть проинициализирован следующим образом для хранения динамического массива экземпляров любых классов, производных от класса **Figure**:

```
List<Figure> array;
array = new List<Figure>();
```

После этого в контейнер можно добавлять элементы при помощи метода **Add**(), которому передаётся ссылка на экземпляр добавляемого объекта.

Число элементов, в текущий момент содержащихся в List, может быть получено с помощью свойства **Count**:

```
int  n =  array.Count;
```

Для перебора всех элементов динамического массива удобно использовать оператор языка C# **foreach** :

```
foreach (Figure f in array) { … }
```

Блок в фигурных скобках выполняется для всех элементов **array**, ссылка на которые поочерёдно помещается в переменную (в данном случае - **f**), доступную внутри блока. Для доступа к элементам List можно использовать и обычную индексную операцию – **[]**.

Операционная система Windows и среда .NET не обеспечивают приложениям сохранение графической информации, выводимой ими в свои окна. В ситуациях перекрытия этих окон другими окнами и их сворачивания ранее выведенная информация будет исчезать, если приложение не

обеспечит восстановление содержания окна. Для этих целей предназначено событие **Paint**, которое должны обрабатывать классы форм и управляющих элементов для восстановления своего содержания. Данное событие генерируется, когда содержание окна должно быть обновлено, в первый раз – при первом отображении окна. Обработчик события Paint получает второй аргумент типа **PaintEventArgs**, поле **Graphics** которого содержит объект типа Graphics, через который должны выполняться в этом обработчике операции графического вывода.

**Graphics в обработчике события Paint не должен запрашиваться через метод CreateGraphics.**

### *Рекомендации по выполнению задания*

Для создания новых классов в проектах Visual Studio.NET можно использовать команду «**Проект**» –> «**Добавить класс…**» . Visual Studio предлагает указать имя файла, в котором будет сохранён новый класс с соответствующим именем.

Пункт 1

Базовый класс для иерархии геометрических фигур рекомендуется сделать абстрактным классом.

При этом заголовок класса может выглядеть следующим образом:

*abstract class Figure*

Класс должен как минимум содержать:

- данные, хранящие информацию о двух точках, задающих размеры блока геометрической фигуры;
- конструктор класса, получающий эти точки в качестве аргументов;
- абстрактные методы **Draw**, **DrawDash**, **Hide**, которые должны будут обеспечивать рисование, рисование пунктиром и стирание фигур соответственно.

Абстрактные методы задаются с ключевым словом **abstract** и не содержат тела метода.

В производных классах эти методы будут получать реализацию, соответствующую типу конкретной фигуры. При этом они объявляются с использованием ключевого слова **override**.

Например:

```
//объявление абстрактного метода в базовом классе
public abstract void Draw(Graphics g);
//реализация абстрактного метода в производном классе
public override void Draw(Graphics g) {…}
```

Функции рисования должны получать в качестве аргумента объект Graphics, который им должна передавать форма, в которой будет рисоваться фигура.

Для рисования эллипса используется метод класса Graphics **DrawEllipse,** который имеет те же аргументы, что и **DrawRectangle.**

Также в классе Figure можно уже поместить реализацию метода **MouseMove**, который получая от формы Graphics и координаты мыши, будет перемещать фигуру в новое положение, последовательно стирая её в исходном положении, изменяя координаты второй точки, задающей фигуру, и отображая фигуру в новом положении пунктиром.

Также можно поместить в этот класс вспомогательный метод для нормализации координат прямоугольника, которая может требоваться для корректного вызова функций рисования. Чтобы метод в C# мог изменять значения переменных, переданных ему через аргументы, при объявлении этих аргументов и их передаче в метод должен использоваться префикс **ref**, обеспечивающий передачу аргумента по ссылке, а не по значению.

Например, заголовок функции нормализации двух пар координат может быть следующим:

*public void norm(ref int x1, ref int y1, ref int x2, ref int y2)*

Пункт 2

Класс, моделирующий прямоугольник, должен быть объявлен производным от класса фигуры и должен содержать соответствующую реализацию абстрактных методов, объявленных в классе фигуры.

Конструктор класса должен получать координаты углов прямоугольника и передавать их классу точки. Передача аргументов конструктору базового класса выполняется с использованием ключевого слова **base**.

Например: *public Rect(Point point1, Point point2) : base(point1, point2) {}*

Пункт 3

Класс, моделирующий эллипс, проектируется аналогично классу, который моделирует прямоугольник.

Пункт 4

Динамический массив следует сделать полем класса Form2 и проинициализировать это поле в конструкторе класса Form2.

**При нажатии на левую кнопку мыши** должен создаваться экземпляр класса прямоугольник.

**При нажатии на правую кнопку мыши** должен создаваться экземпляр класса эллипс.

**При перемещении мыши,** необходимо проверять, какая кнопка мыши была нажата, чтобы определить экземпляр какого класса был создан, а затем созданный экземпляр соответствующего класса должен использоваться для прорисовки перемещения фигуры (прямоугольника либо эллипса).

**При отпускании левой кнопки** должен рисоваться окончательный вид прямоугольника и он должен добавляться к динамическому массиву.

**При отпускании правой кнопки** должен рисоваться окончательный вид эллипса и он должен добавляться к динамическому массиву.

**Примечание**. Код класса формы следует очистить от всех деталей, связанных с рисованием прямоугольников и обработкой их координат. Он должен лишь управлять объектами, выполняющими эти действия.

Пункт 5

Для восстановления информации в окне следует реализовать в классе Form2 обработчик события Paint. Этот метод должен перебирать (в цикле **foreach**) все элементы динамического массива и перерисовывать их.

После этого в конец блока, обрабатывающего добавление нового прямоугольника в массив, следует добавить вызов метода **Invalidate()**, который также инициирует перерисовку окна. Этот вызов обеспечит устранение дефектов рисунка, которые могут появляться в процессе добавления новых элементов к рисунку.

## СПИСОК БИБЛИОГРАФИЧЕСКИХ ИСТОЧНИКОВ

1. 1. Павловская Т.А. C#. Программирование на языке высокого уровня. – СПб.:Питер, 2007.

2. Троелсен Э. C# и платформа.NET. Библиотека программиста. – СПб.:Питер, 2007.

3. Антонов И.В., Бруттан Ю.В. Программирование на языке высокого уровня (C#). Методические указания к лабораторным работам.– Псков: Издательство ППИ, 2007.

**ПРИЛОЖЕНИЕ А**
**Пример титульного листа**

Псковский государственный политехнический институт

***Контрольная работа №1***

по учебной дисциплине

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ (C#)**

Выполнил: Иванов А.А.

Группа: 682-1003С

Проверил: Бруттан Ю.В.

Псков
2009

## ПРИЛОЖЕНИЕ Б
## Примеры диаграмм UML

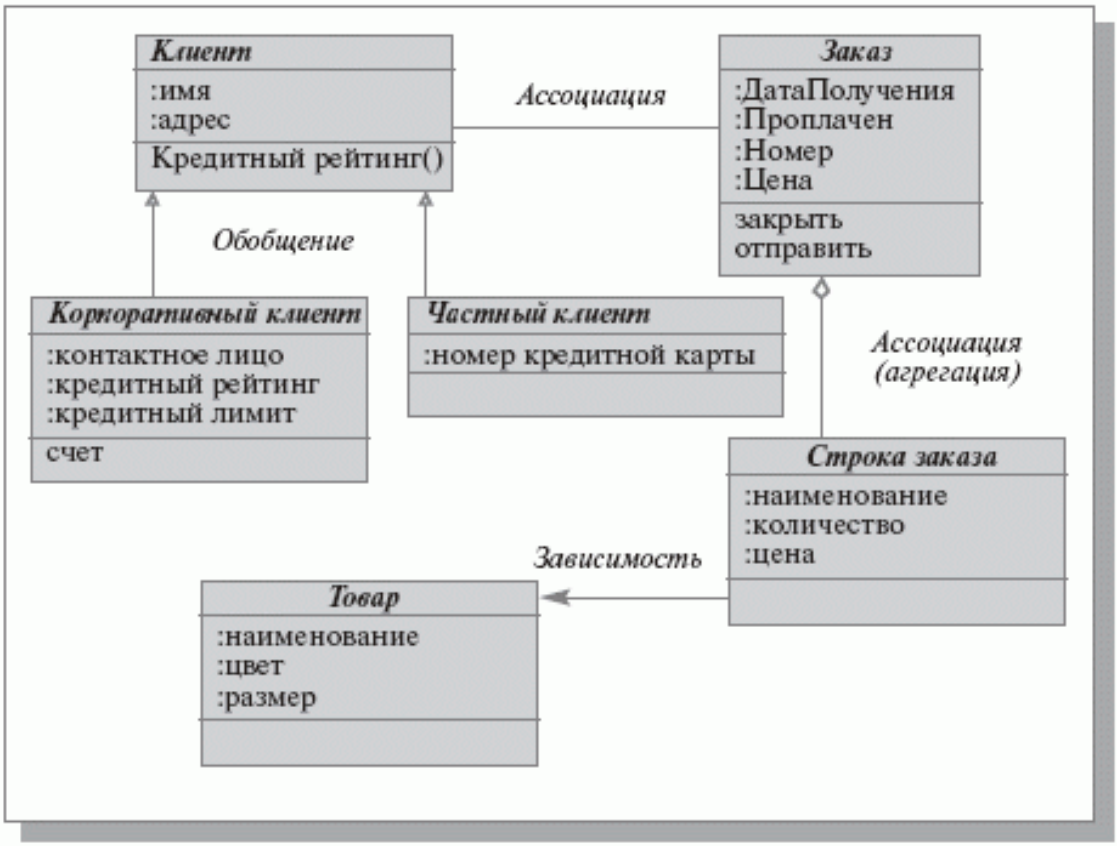


Рис. П1. Пример диаграммы классов

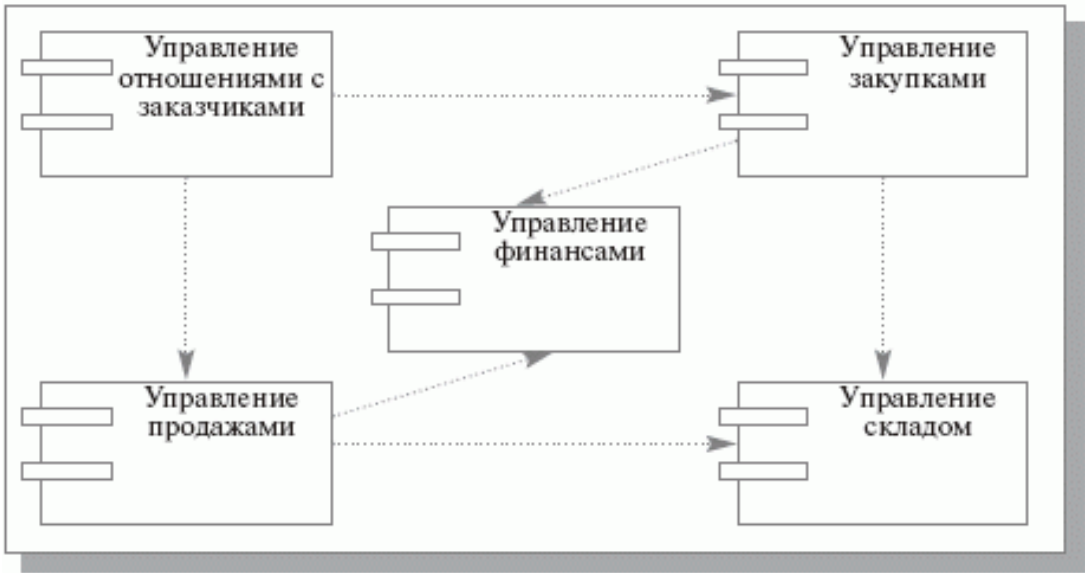


Рис. П2. Пример диаграммы компонентов


Рис. П3. Пример диаграммы деятельности

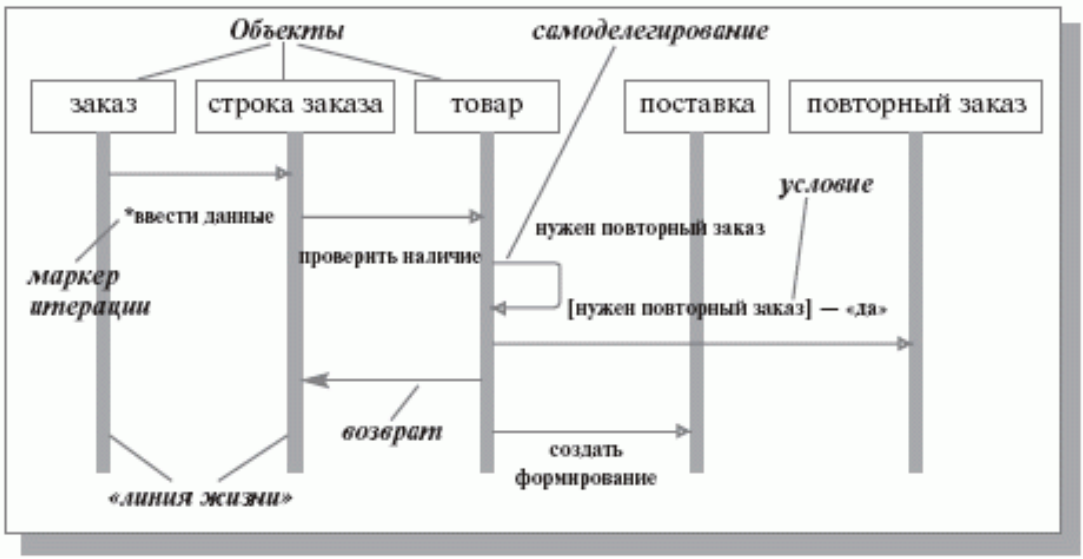


Рис. П4. Пример диаграммы последовательностей

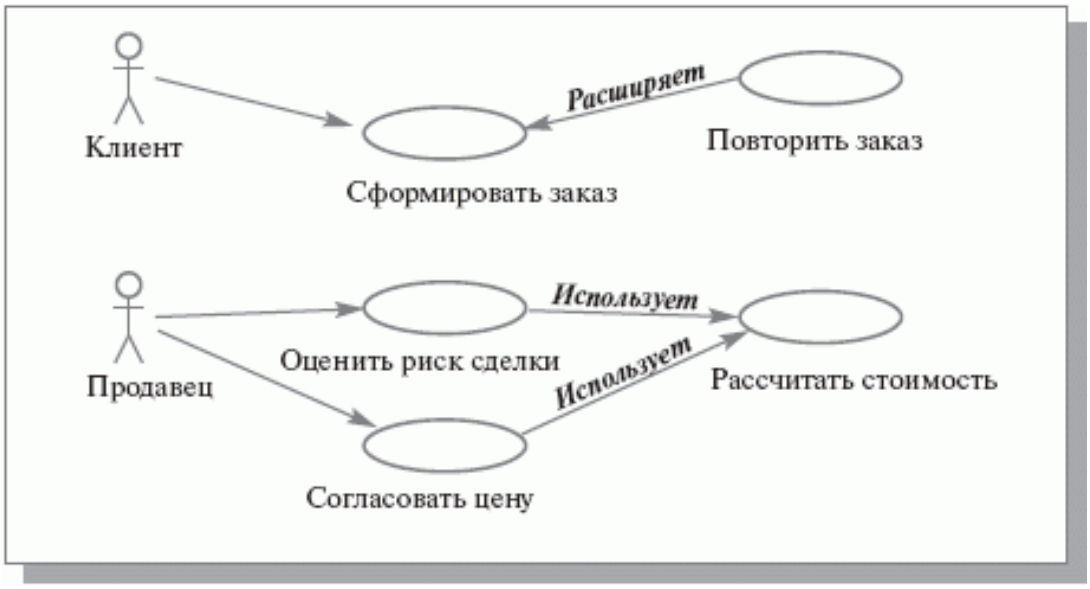


Рис. П5. Пример диаграммы прецедентов

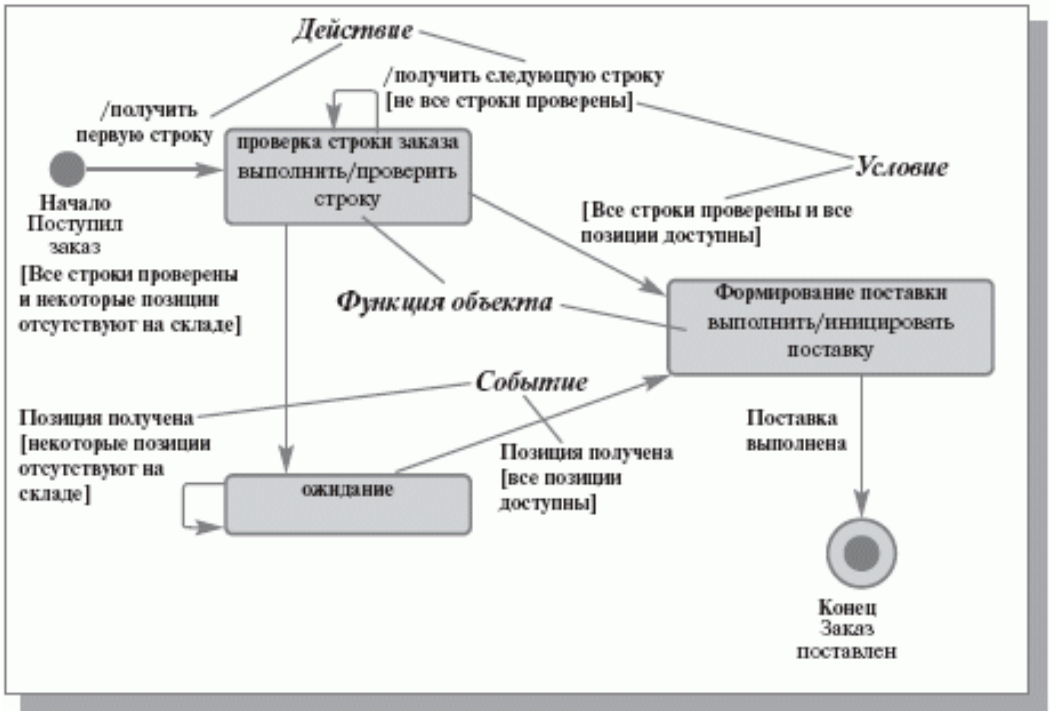


Рис. П6. Пример диаграммы состояний

***Антонов*** *Игорь Вадимович*
***Бруттан*** *Юлия Викторовна*

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ (C#)
*Учебно-методическое пособие*

Редактор Ю.В. Бруттан
Компьютерная верстка Ю.В. Бруттан

---

Подписано в печать 13.03.2010 г. Формат 60×90/16.
Гарнитура Times New Roman. Усл. п.л. 3,1.
Тираж 100 экз. Заказ №2500.

Адрес издательства:
Россия, 180000, Псков, ул. Л.Толстого 4.
Издательство ППИ